Spring MVC et Thymeleaf

par l'exemple

serge.tahe at univ-angers.fr, janvier 2015

Table des matières

1 INTRODUCTION....8
1.1 Sources....8
1.2 Les outils utilisés....8
1.3 Les exemples....8
1.4 La place de Spring MVC dans une application Web....11
1.5 Le modèle de développement de Spring MVC....12
1.6 Un premier projet Spring MVC....14
1.6.1 Le projet de démonstration....14
1.6.2 Configuration Maven....15
1.6.3 L'architecture d'une application Spring MVC....17
1.6.4 Le contrôleur C....17
1.6.5 La vue V....19
1.6.6 Exécution....20
1.6.7 Création d'une archive exécutable....23
1.6.8 Déployer l'application sur un serveur Tomcat....25
1.7 Un second projet Spring MVC....28
1.7.1 Le projet de démonstration....28
1.7.2 Configuration Maven....29
1.7.3 L'architecture d'un service Spring [web / jSON]....30
1.7.4 Le contrôleur C....31
1.7.5 Le modèle M....32
1.7.6 Exécution....33
1.7.7 Exécution du projet....33
1.7.8 Création d'une archive exécutable....36
1.7.9 Déployer l'application sur un serveur Tomcat....38
1.8 Conclusion....39
2 LES BASES DE LA PROGRAMMATION WEB....41
2.1 Les échanges de données dans une application Web avec formulaire....42
2.2 Pages Web statiques, Pages Web dynamiques....43
2.2.1 Page statique HTML (HyperText Markup Language)....43
2.2.2 Une page Thymeleaf dynamique....48
2.2.3 Configuration de l'application Spring Boot....51
2.3 Scripts côté navigateur....52
2.4 Les échanges client-serveur....53
2.4.1 Le modèle OSI....54
2.4.2 Le modèle TCP/IP....55
2.4.3 Le protocole HTTP....57
2.4.4 Conclusion....61
2.5 Les bases du langage HTML....61
2.5.1 Un exemple....62
2.5.2 Un formulaire HTML....64
2.5.2.1 Le formulaire....67
2.5.2.2 Les champs de saisie texte....67
2.5.2.3 Les champs de saisie multilignes....68
2.5.2.4 Les boutons radio....68
2.5.2.5 Les cases à cocher....68
2.5.2.6 La liste déroulante (combo)....69
2.5.2.7 Liste à sélection unique....69
2.5.2.8 Liste à sélection multiple....70
2.5.2.9 Bouton de type button....70
2.5.2.10 Bouton de type submit....71
2.5.2.11 Bouton de type reset....71
2.5.2.12 Champ caché....71
2.5.3 Envoi à un serveur Web par un client Web des valeurs d'un formulaire....72
2.5.3.1 Méthode GET....72
2.5.3.2 Méthode POST....76
2.6 Conclusion....76
3 ACTIONS : LA RÉPONSE....78
3.1 Le nouveau projet....78
3.2 [/a01, /a02] - Hello world....81
3.3 [/a03] : rendre un flux XML....83

3.4 [/a04, /a05] : rendre un flux jSON....85
3.5 [/a06] : rendre un flux vide....87
3.6 [/a07, /a08, /a09] : nature du flux avec [Content-Type]....88
3.7 [/a10, /a11, /a12] : rediriger le client....90
3.8 [/a13] : générer la réponse complète....93
4 ACTIONS : LE MODÈLE....95
4.1 [/m01] : paramètres d'un GET....96
4.2 [/m02] : paramètres d'un POST....96
4.3 [/m03] : paramètres de mêmes noms....97
4.4 [/m04] : mapper les paramètres de l'action dans un objet Java....98
4.5 [/m05] : récupérer les éléments d'une URL....99
4.6 [/m06] : récupérer des éléments d'URL et des paramètres....100
4.7 [/m07] : accéder à la totalité de la requête....100
4.8 [/m08] : accès à l'objet [Writer]....101
4.9 [/m09] : accéder à un entête HTTP....102
4.10 [/m10, /m11] : accéder à un cookie....103
4.11 [/m12] : accéder au corps d'un POST....105
4.12 [/m13, /m14] : récupérer des valeurs postées en jSON....106
4.13 [/m15] : récupérer la session....108
4.14 [/m16] : récupérer un objet de portée [session]....110
4.15 [/m17] : récupérer un objet de portée [application]....113
4.16 [/m18] : récupérer un objet de portée [session] avec [@SessionAttributes]....114
4.17 [/m20-/m23] : injection d'informations avec [@ModelAttribute]....115
4.18 [/m24] : validation du modèle de l'action....118
4.19 [m/24] : personnalisation des messages d'erreur....121
4.20 [/m25] : internationalisation d'une application Spring MVC....125
4.21 [/m26] : injection de la locale dans le modèle de l'action....128
4.22 [/m27] : vérifier la validité d'un modèle avec Hibernate Validator....129
4.23 [/m28] : externalisation des messages d'erreur....134
5 LES VUES THYMELEAF....138
5.1 Le projet STS....138
5.2 [/v01] : les bases de Thymeleaf....140
5.3 [/v03] : internationalisation des vues....144
5.4 [/v04] : création du modèle M d'une vue V....145
5.5 [/v05] : factorisation d'un objet dans une vue Thymeleaf....150
5.6 [/v06] : les tests dans une vue Thymeleaf....151
5.7 [/v07] : itération dans une vue Thymeleaf....152
5.8 [/v08-/v10] : @ModelAttribute....153
5.9 [/v11] : @SessionAttributes....156
5.10 [/v13] : générer un formulaire de saisie....159
5.11 [/v14] : gérer les valeurs postées par un formulaire....161
5.12 [/v15-/v16] : validation d'un modèle....162
5.13 [/v17-/v18] : contrôle des messages d'erreur....166
5.14 [/v19-/v20] : usage de différents validateurs....170
5.15 [/v21-/v22] : gérer des boutons radio....180
5.16 [/v23-/v24] : gérer des cases à cocher....184
5.17 [/25-/v26] : gérer des listes....187
5.18 [/v27] : paramétrage des messages....190
5.19 Utilisation d'une page maître....192
5.19.1 Le projet....192
5.19.2 La page maître....194
5.19.3 Les fragments....195
5.19.4 Les actions....196
6 VALIDATION JAVASCRIPT CÔTÉ CLIENT....197
6.1 Les fonctionnalités du projet....197
6.2 Validation côté serveur....200
6.2.1 Configuration....200
6.2.2 Le modèle du formulaire....204
6.2.3 Le contrôleur....206
6.2.4 La vue....209
6.2.5 La feuille de style....211
6.3 Validation côté client....212
6.3.1 Rudiments de jQuery et de Javascript....212

6.3.2 Les bibliothèques JS de validation....215
6.3.3 Import des bibliothèques JS de validation....218
6.3.4 Gestion de la locale côté client....218
6.3.5 Les fichiers de messages....220
6.3.6 Changement de locale....222
6.3.7 Le POST des valeurs saisies....225
6.3.8 Validateur [required]....230
6.3.9 Validateur [assertfalse]....231
6.3.10 Validateur [asserttrue]....234
6.3.11 Validateurs [date] et [past]....235
6.3.12 Validateur [future]....237
6.3.13 Validateurs [int] et [max]....238
6.3.14 Validateur [min]....240
6.3.15 Validateur [regex]....242
6.3.16 Validateur [email]....242
6.3.17 Validateur [range]....243
6.3.18 Validateur [number]....244
6.3.19 Validateur [custom3]....247
6.3.20 Validateur [url]....249
6.3.21 Activation / Désactivation de la validation côté client....249
7 AJAXIFICATION D'UNE APPLICATION SPRING MVC....253
7.1 La place d'AJAX dans une application web....253
7.2 Mise à jour d'une page avec un flux HTML....254
7.2.1 Les vues....254
7.2.2 L'action [/ajax-01]....254
7.2.3 La vue [vue-01.xml]....257
7.2.4 Le formulaire....258
7.2.5 L'action [/ajax-02]....262
7.2.6 Le POST des valeurs saisies....265
7.2.7 Tests....267
7.2.8 Désactivation du Javascript avec la culture [en-US]....268
7.2.9 Désactivation du Javascript avec la culture [fr-FR]....272
7.2.10 Gestion du lien [Calculer]....277
7.3 Mise à jour d'une page HTML avec un flux JSON....279
7.3.1 L'action [/ajax-04]....279
7.3.2 La vue [vue-04.xml]....280
7.3.3 La fonction JS [postForm]....282
7.3.4 L'action [/ajax-05]....283
7.3.5 La fonction JS [postForm] - 2....287
7.3.6 Tests....288
7.4 Application web à page unique....289
7.4.1 Introduction....289
7.4.2 L'action [/ajax-06]....290
7.4.3 La vue [vue-06.xml]....290
7.4.4 La vue [vue-07.xml]....291
7.4.5 La fonction JS [gotoPage]....291
7.4.6 L'action [/ajax-07]....291
7.4.7 La vue [vue-08.xml]....292
7.5 Embarquer plusieurs flux HTML dans une réponse JSON....292
7.5.1 Introduction....292
7.5.2 L'action [/ajax-09]....294
7.5.3 Les vues XML....295
7.5.4 Le code JS de gestion du bouton [Rafraîchir]....296
7.5.5 L'action [/ajax-10]....298
7.5.6 Traitement de la réponse de l'action [/ajax-10]....303
7.5.7 Affichage de la page [Page 2]....305
7.5.8 L'action [ajax-11A]....306
7.5.9 Traitement de la réponse de l'action [/ajax-11A]....308
7.5.10 Retour vers la page n° 1....312
7.5.11 L'action [/ajax-11B]....312
7.5.12 Traitement de la réponse de l'action [/ajax-11B]....313
7.6 Gérer la session côté client....314
7.6.1 Introduction....314

7.6.2 L'action [/ajax-12]....314
7.6.3 Le code JS de gestion du bouton [Rafraîchir]....316
7.6.4 L'action [/ajax-13]....317
7.6.5 Traitement de la réponse de l'action [/ajax-13]....320
7.6.6 Affichage de la page [Page 2]....321
7.6.7 L'action [/ajax-14]....322
7.6.8 Traitement de la réponse de l'action [/ajax-14]....323
7.6.9 Retour à la page n° 1....324
7.6.10 Conclusion....324
7.7 Structuration du code Javascript en couches....324
7.7.1 Introduction....324
7.7.2 La page de démarrage....325
7.7.3 Implémentation de la couche [DAO]....326
7.7.4 Interface....326
7.7.5 Implémentation de l'interface....326
7.7.5.1 La fonction [updatePage1]....326
7.7.5.2 La fonction [getPage2]....327
7.7.6 La couche [présentation]....328
7.7.6.1 La fonction [postForm]....328
7.7.6.2 Le rôle du paramètre [sendMeBack]....329
7.7.7 La fonction [valider]....330
7.7.8 Tests....331
7.8 Conclusion....331
8 ETUDE DE CAS....333
8.1 Introduction....333
8.2 Fonctionnalités de l'application....334
8.3 La base de données....343
8.3.1 La table [MEDECINS]....344
8.3.2 La table [CLIENTS]....345
8.3.3 La table [CRENEAUX]....345
8.3.4 La table [RV]....346
8.3.5 Création de la base de données....346
8.4 Le service web / jSON....348
8.4.1 Introduction à Spring Data....348
8.4.1.1 La configuration Maven du projet....349
8.4.1.2 La couche [JPA]....351
8.4.1.3 La couche [DAO]....352
8.4.1.4 La couche [console]....354
8.4.1.5 Configuration manuelle du projet Spring Data....356
8.4.1.6 Création d'une archive exécutable....361
8.4.1.7 Créer un nouveau projet Spring Data....363
8.4.2 Le projet Eclipse du serveur....365
8.4.3 La configuration Maven....366
8.4.4 Les entités JPA....368
8.4.5 La couche [DAO]....374
8.4.6 La couche [métier]....376
8.4.6.1 Les entités....376
8.4.6.2 Le service....377
8.4.7 La configuration du projet....380
8.4.8 Les tests de la couche [métier]....381
8.4.9 Le programme console....384
8.4.10 La couche [web / jSON]....385
8.4.10.1 Configuration Maven....386
8.4.10.2 L'interface du service web....387
8.4.10.3 Configuration du service web....393
8.4.10.4 La classe [ApplicationModel]....395
8.4.10.5 La classe Static....397
8.4.10.6 Le squelette du contrôleur [RdvMedecinsController]....398
8.4.10.7 L'URL [/getAllMedecins]....402
8.4.10.8 L'URL [/getAllClients]....402
8.4.10.9 L'URL [/getAllCreneaux/{idMedecin}]....403
8.4.10.10 L'URL [/getRvMedecinJour/{idMedecin}/{jour}]....406
8.4.10.11 L'URL [/getAgendaMedecinJour/{idMedecin}/{jour}]....408

8.4.10.12 L'URL [/getMedecinById/{id}]....410
8.4.10.13 L'URL [/getClientById/{id}]....411
8.4.10.14 L'URL [/getCreneauById/{id}]....412
8.4.10.15 L'URL [/getRvById/{id}]....414
8.4.10.16 L'URL [/ajouterRv]....415
8.4.10.17 L'URL [/supprimerRv]....418
8.4.10.18 La classe exécutable du service web....419
8.4.11 Introduction à Spring Security....423
8.4.11.1 Configuration Maven....424
8.4.11.2 Les vues Thymeleaf....425
8.4.11.3 Configuration Spring MVC....428
8.4.11.4 Configuration Spring Security....429
8.4.11.5 Classe exécutable....430
8.4.11.6 Tests de l'application....430
8.4.11.7 Conclusion....432
8.4.12 Mise en place de la sécurité sur le service web de rendez-vous....433
8.4.12.1 La base de données....433
8.4.12.2 Le nouveau projet STS du [métier, DAO, JPA]....434
8.4.12.3 Les nouvelles entités [JPA]....434
8.4.12.4 Modifications de la couche [DAO]....436
8.4.12.5 Les classes de gestion des utilisateurs et des rôles....438
8.4.12.6 Tests de la couche [DAO]....440
8.4.12.7 Conclusion intermédiaire....444
8.4.12.8 Le projet STS de la couche [web]....444
8.4.12.9 Tests du service web....447
8.4.13 Mise en place des requêtes inter-domaines....452
8.4.13.1 Le projet du client....453
8.4.13.2 L'URL [/getAllMedecins]....457
8.4.13.3 Les autres URL [GET]....465
8.4.13.4 Les URL [POST]....468
8.4.13.5 Conclusion....471
8.5 Client programmé du service web / JSON....472
8.5.1 Le projet du client console....473
8.5.2 Configuration Maven....473
8.5.3 Le package [rdvmedecins.client.entities]....474
8.5.4 Le package [rdvmedecins.client.requests]....475
8.5.5 Le package [rdvmedecins.client.responses]....475
8.5.6 Le package [rdvmedecins.client.dao]....476
8.5.7 Le package [rdvmedecins.client.config]....476
8.5.8 L'interface [IDao]....477
8.5.9 Le package [rdvmedecins.clients.console]....480
8.5.10 Implémentation de la couche [DAO]....485
8.5.11 Anomalie....491
8.6 Ecriture du serveur Spring / Thymeleaf....494
8.6.1 Introduction....494
8.6.2 Le projet STS....495
8.6.3 Les fonctionnalités de l'application....498
8.6.4 Étape 1 : introduction au framework CSS Bootstrap....505
8.6.4.1 Le projet des exemples....505
8.6.4.1.1 Configuration Maven....506
8.6.4.1.2 Configuration Java....507
8.6.4.1.3 Le contrôleur Spring....507
8.6.4.1.4 Le fichier [application.properties]....508
8.6.4.2 Exemple n° 1 : le jumbotron....509
8.6.4.3 Exemple n° 2 : la barre de navigation....510
8.6.4.4 Exemple n° 3 : le bouton à liste....512
8.6.4.5 Exemple n° 4 : un menu....515
8.6.4.6 Exemple n° 5 : une liste déroulante....519
8.6.4.7 Exemple n° 6 : un calendrier....521
8.6.4.8 Exemple n° 7 : une table HTML 'responsive'....526
8.6.4.9 Exemple n° 8 : une boîte modale....531
8.6.5 Étape 2 : écriture des vues....536
8.6.5.1 La vue [navbar-start]....536

8.6.5.1 La vue [jumbotron]....537
8.6.5.2 La vue [login]....538
8.6.5.3 La vue [navbar-run]....538
8.6.5.4 La vue [accueil]....539
8.6.5.5 La vue [agenda]....540
8.6.5.6 La vue [erreurs]....543
8.6.5.7 Résumé....543
8.6.6 Étape 3 : écriture des actions....544
8.6.6.1 Les URL exposées par le service [Web1] 544
8.6.6.2 Le singleton [ApplicationModel]....545
8.6.6.3 La classe [BaseController]....547
8.6.6.4 L'action [/getNavBarStart]....551
8.6.6.5 L'action [/getNavbarRun]....552
8.6.6.6 L'action [/getJumbotron]....553
8.6.6.7 L'action [/getLogin]....553
8.6.6.8 L'action [/getAccueil]....554
8.6.6.9 L'action [/getNavbarRunJumbotronAccueil]....555
8.6.6.10 L'action [/getAgenda]....556
8.6.6.11 L'action [/getNavbarRunJumbotronAccueilAgenda]....558
8.6.6.12 L'action [/supprimerRv]....558
8.6.6.13 L'action [/validerRv]....559
8.6.7 Étape 4 : tests du serveur Spring/Thymeleaf....562
8.6.7.1 Configuration des tests....562
8.6.7.2 L'action [/getNavbarStart]....562
8.6.7.3 L'action [/getNavbarRun]....563
8.6.7.4 L'action [/getJumbotron]....564
8.6.7.5 L'action [/getLogin]....564
8.6.7.6 L'action [/getAccueil]....565
8.6.7.7 L'action [/getAgenda]....566
8.6.7.8 L'action [/getNavbarRunJumbotronAccueil]....568
8.6.7.9 L'action [/getNavbarRunJumbotronAccueilAgenda]....569
8.6.7.10 L'action [/supprimerRv]....570
8.6.7.11 L'action [/validerRv]....570
8.6.8 étape 5 : Écriture du client Javascript....572
8.6.8.1 Le projet JS....572
8.6.8.2 L'architecture du code....573
8.6.8.3 La couche [présentation]....573
8.6.8.4 Les fonctions utilitaires de la couche [événements]....575
8.6.8.5 Connexion d'un utilisateur....577
8.6.8.6 Changement de langue....578
8.6.8.7 La fonction [getAccueilAvecAgenda-one]....580
8.6.8.8 La fonction [getAccueilAvecAgenda-parallel]....580
8.6.8.9 La fonction [getAccueilAvecAgenda-sequence]....581
8.6.8.10 La couche [DAO]....582
8.6.8.11 La page de boot....585
8.6.8.12 Tests....588
8.6.8.13 Conclusion....592
8.6.9 étape 6 : génération d'une application native pour Android....593
8.6.10 Conclusion de l'étude de cas....597
9 ANNEXES....599
9.1 Installation d'un JDK....599
9.2 Installation de Maven....599
9.3 Installation de STS (Spring Tool Suite)....600
9.4 Installation d'un serveur Tomcat....603
9.5 Installation de [WampServer]....605
9.6 Installation du plugin Chrome [Advanced Rest Client]....606
9.7 Gestion du jSON en Java....607
9.8 Installation de [Webstorm]....609
9.8.1 Installation de [node.js]....610
9.8.2 Installation de l'outil [bower]....610
9.8.3 Installation de [Git]....610
9.8.4 Configuration de [Webstorm]....611
9.9 Installation d'un émulateur pour Android....612

1 Introduction

Nous nous proposons ici d'introduire à l'aide d'exemples les notions importantes de **Spring MVC**, un framework Web Java qui fournit un cadre pour développer des applications Web selon le modèle MVC (Modèle – Vue – Contrôleur). Spring MVC est une branche de l'écosystème Spring [http://projects.spring.io/spring-framework/]. Nous présentons également le moteur de vues **Thymeleaf** [http://www.thymeleaf.org/].

Ce cours est à destination de lecteurs ayant une vraie maîtrise du langage Java. Il n'est pas nécessaire de connaître la programmation web.

Bien que détaillé, ce document est probablement incomplet. Spring est un framework immense avec de nombreuses ramifications. Pour approfondir Spring MVC, on pourra utiliser les références suivantes :

- le document de référence du framework Spring [http://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/spring-framework-reference.pdf] ;
- de nombreux tutoriels Spring sont trouvés à l'URL [http://spring.io/guides]
- le site de [developpez.com] consacré à Spring [http://spring.developpez.com/].

Le document a été écrit de telle façon qu'il puisse être lu sans ordinateur sous la main. Aussi, donne-t-on beaucoup de copies d'écran.

1.1 Sources

Ce document a deux sources principales :

- [Introduction à ASP.NET MVC par l'exemple]. Spring MVC et ASP.NET MVC sont deux frameworks analogues, le second ayant été construit bien après le premier. Afin de pouvoir comparer les deux frameworks, j'ai repris la même progression que dans le document sur ASP.NET MVC ;
- le document sur ASP.NET MVC ne contient pas pour l'instant (déc 2014) d'étude de cas. J'ai repris ici celle du document [Tutoriel AngularJS / Spring 4] que j'ai modifiée de la façon suivante :
  - l'étude de cas dans [Tutoriel AngularJS / Spring 4] est celle d'une application client / serveur où le serveur est un service web / jSON construit avec Spring MVC et le client, un client AngularJS,
  - dans ce document, on reprend le même service web / jSON mais le client est une application web 2tier [client jQuery] / [service web / jSON] ;

En-dehors de ces sources, je suis allé chercher sur Internet les réponses à mes questions. C'est surtout le site [http://stackoverflow.com/] qui m'a alors été utile.

1.2 Les outils utilisés

Les exemples qui suivent ont été testés dans l'environnement suivant :

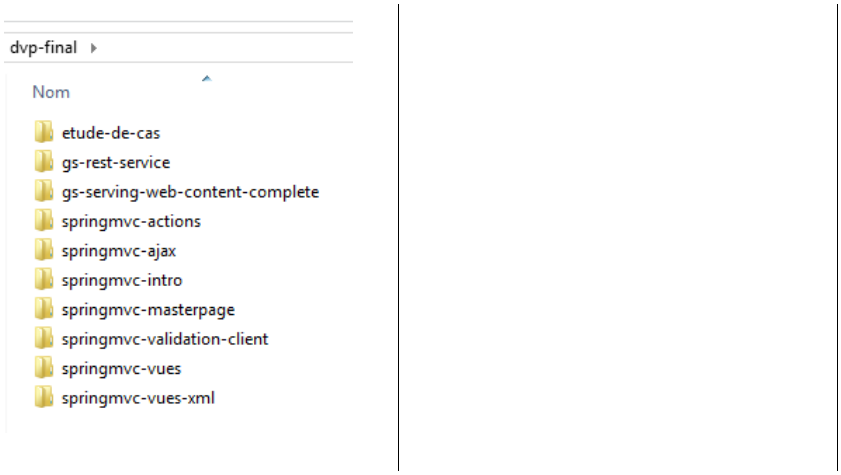
- machine Windows 8.1 pro 64 bits ;
- JDK 1.8 ;
- IDE Spring Tool Suite 3.6.3 (cf paragraphe 9.3, page 600) ;
- navigateur Chrome (les autres navigateurs n'ont pas été utilisés) ;
- extension Chrome [Advanced Rest Client] (cf paragraphe 9.6, page 606) ;

Attention au JDK 1.8. L'une des méthodes de l'étude de cas utilise une méthode du package [java.lang] de Java 8.

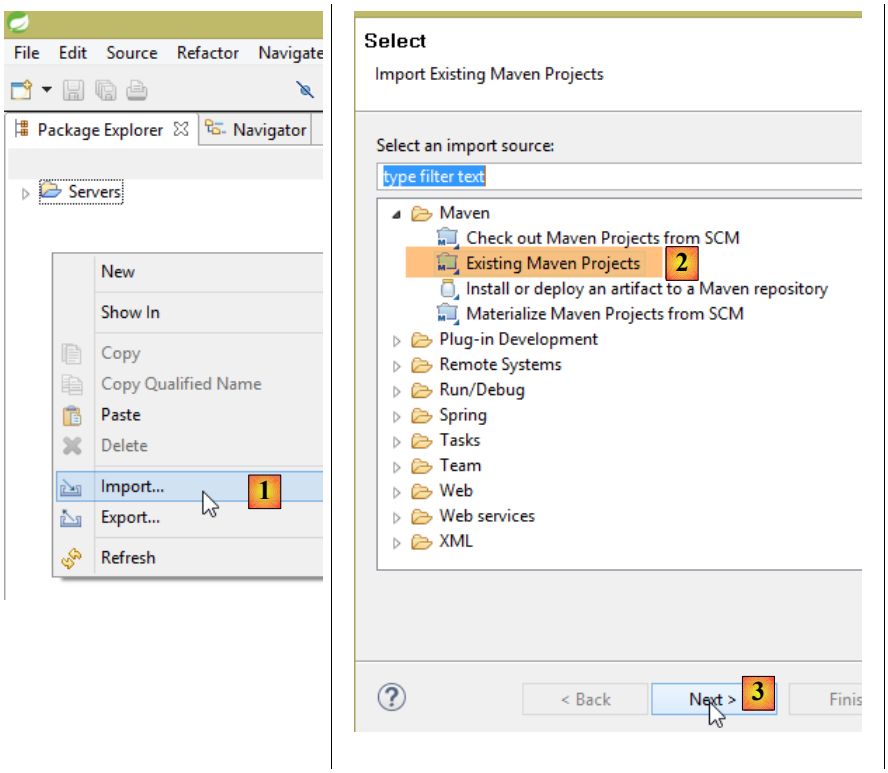
Tous les exemples sont des projets Maven qui peuvent être ouverts indifféremment par les IDE Eclipse, IntellijIDEA, Netbeans. Dans la suite, les copies d'écran proviennent de l'IDE Spring Tool Suite, une variante d'Eclipse.

1.3 Les exemples

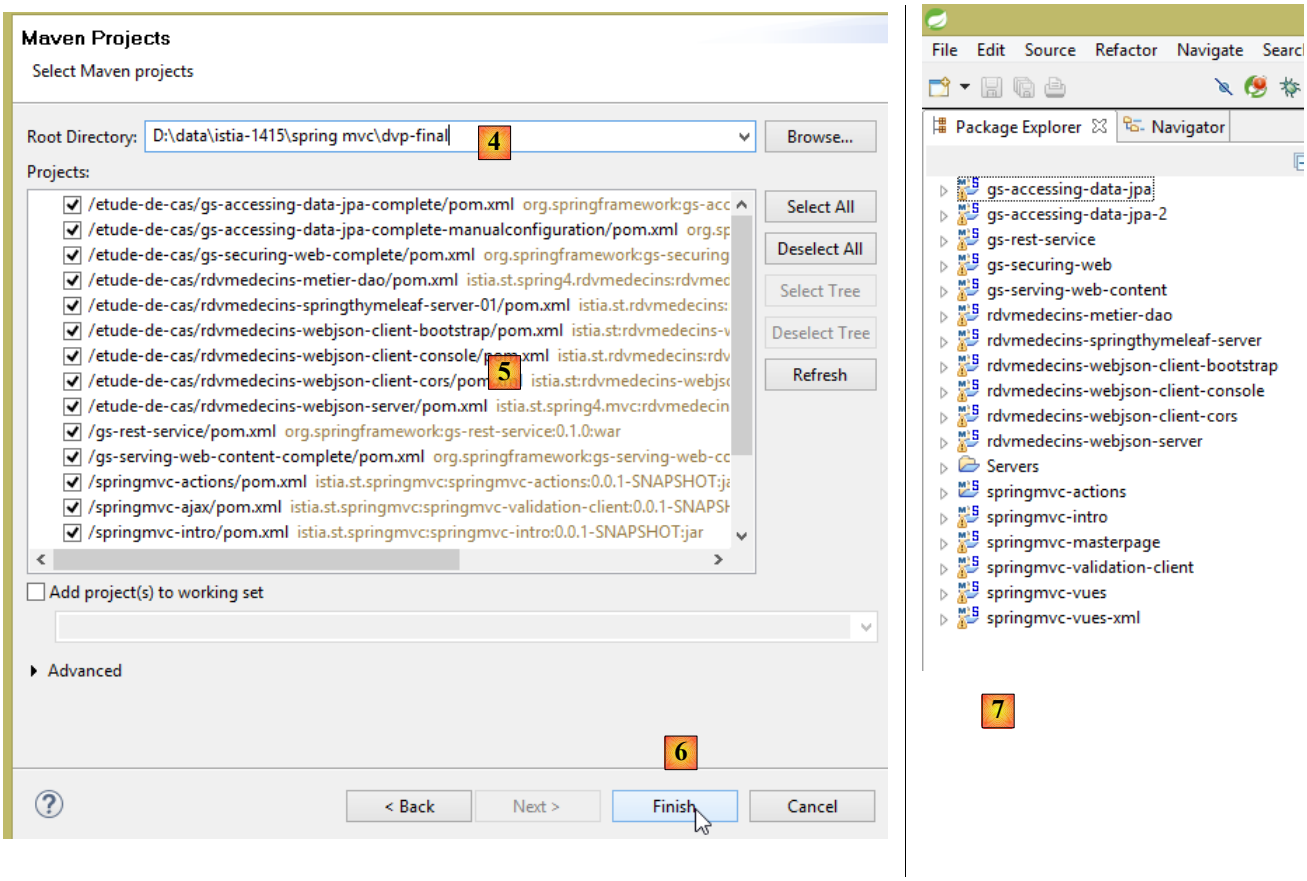
Les exemples sont disponibles à l'URL [http://tahe.developpez.com/java/springmvc-thymeleaf] sous la forme d'un fichier *zip* à télécharger.



Pour charger tous les projets dans STS on procèdera de la façon suivante :



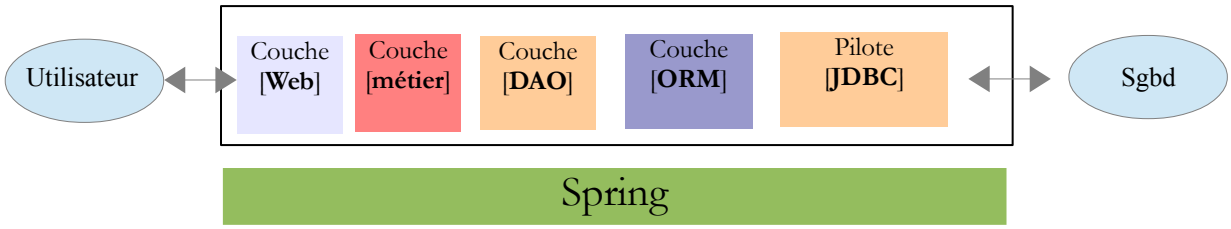
- en [1-3], importez des projets Maven ;



- en [4], désignez le dossier des exemples ;
- en [5], sélectionnez tous les projets du dossier ;
- en [6], validez ;
- en [7], les projets importés ;

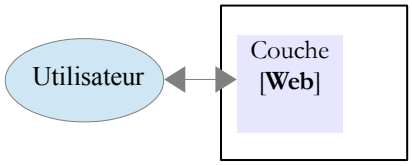
1.4 La place de Spring MVC dans une application Web

Situons Spring MVC dans le développement d'une application Web. Le plus souvent, celle-ci sera bâtie sur une architecture multicouche telle que la suivante :

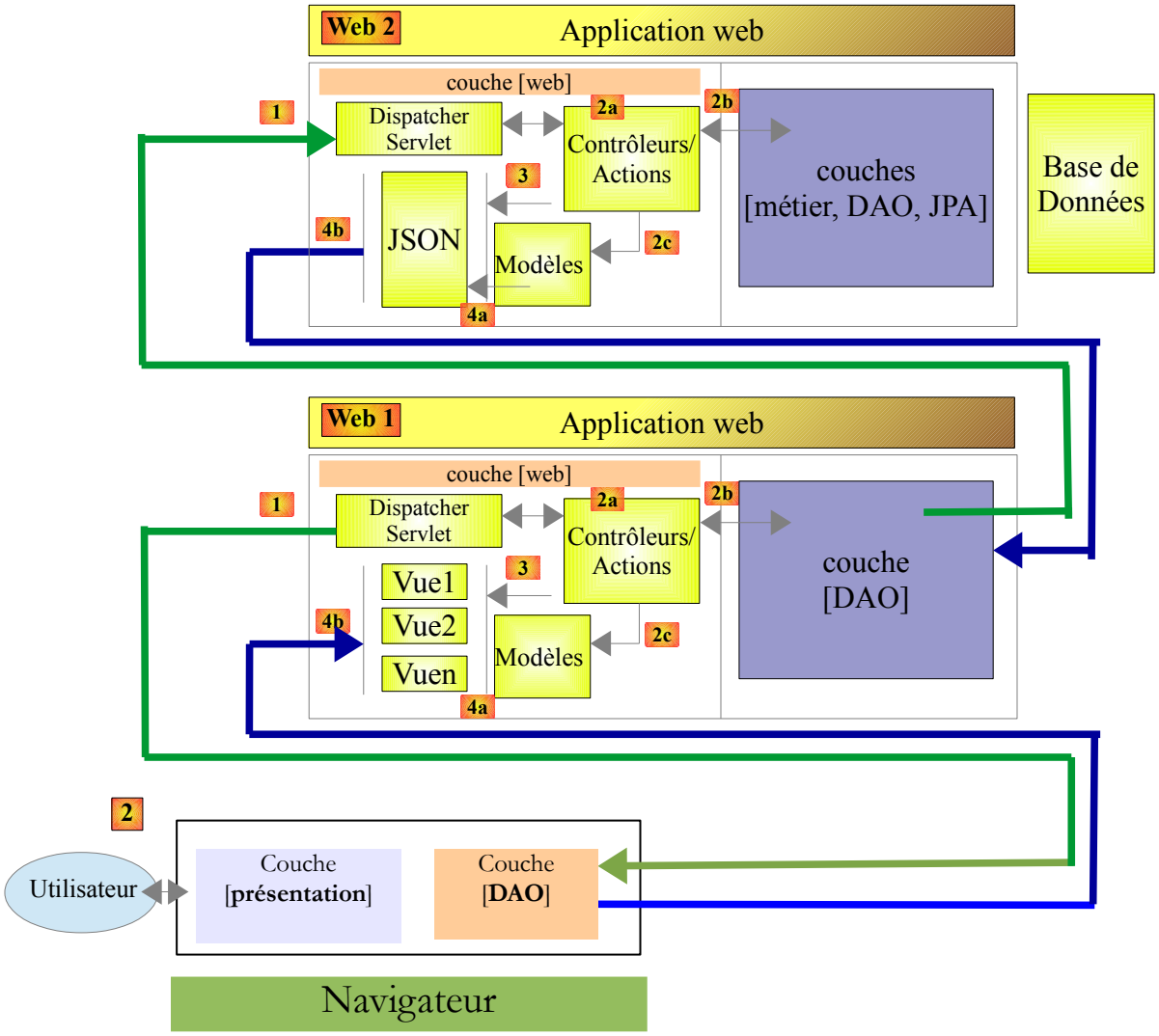


- la couche [**Web**] est la couche en contact avec l'utilisateur de l'application Web. Celui-ci interagit avec l'application Web au travers de pages Web visualisées par un navigateur. **C'est dans cette couche que se situe Spring MVC et uniquement dans cette couche ;**
- la couche [**métier**] implémente les règles de gestion de l'application, tels que le calcul d'un salaire ou d'une facture. Cette couche utilise des données provenant de l'utilisateur via la couche [Web] et du SGBD via la couche [DAO] ;
- la couche [**DAO**] (Data Access Objects), la couche [**ORM**] (Object Relational Mapper) et le pilote JDBC gèrent l'accès aux données du SGBD. La couche [**ORM**] fait un pont entre les objets manipulés par la couche [DAO] et les lignes et les colonnes des tables d'une base de données relationnelle. Nous utiliserons ici l'ORM Hibernate. Une spécification appelée JPA (Java Persistence API) permet de s'abstraire de l'ORM utilisé si celui-ci implémente ces spécifications. C'est le cas d'Hibernate et des autres ORM Java. On appellera donc désormais la couche ORM, la couche JPA ;
- l'intégration des couches est faite par le framework Spring ;

La plupart des exemples donnés dans la suite, n'utiliseront qu'une seule couche, la couche [Web] :



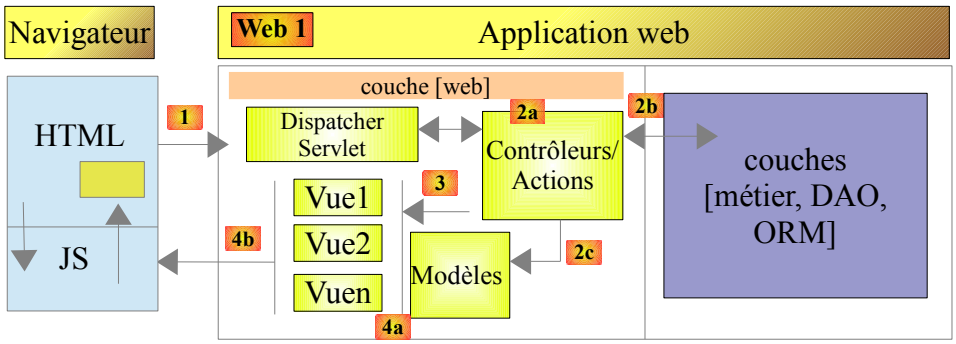
Ce document se terminera cependant par la construction d'une application Web multitier :



Le navigateur se connectera à une application [Web1] implémentée par Spring MVC / Thymeleaf qui ira chercher ses données auprès d'un service web [Web2] lui aussi implémenté avec Spring MVC. Cette seconde application web accédera à une base de données.

1.5 Le modèle de développement de Spring MVC

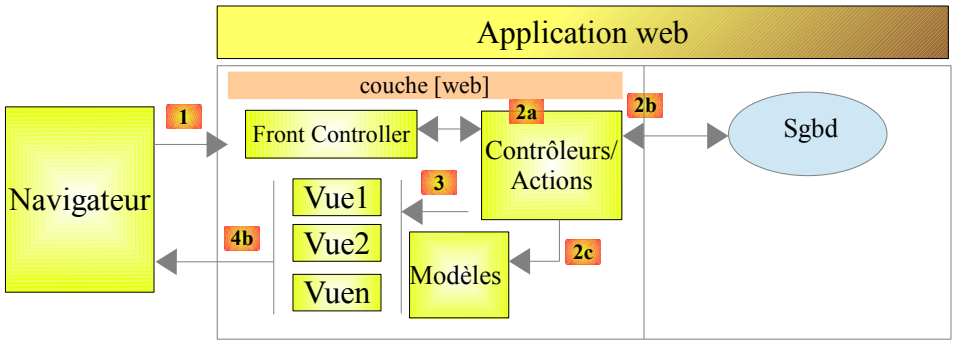
Spring MVC implémente le modèle d'architecture dit MVC (Modèle – Vue – Contrôleur) de la façon suivante :



Le traitement d'une demande d'un client se déroule de la façon suivante :

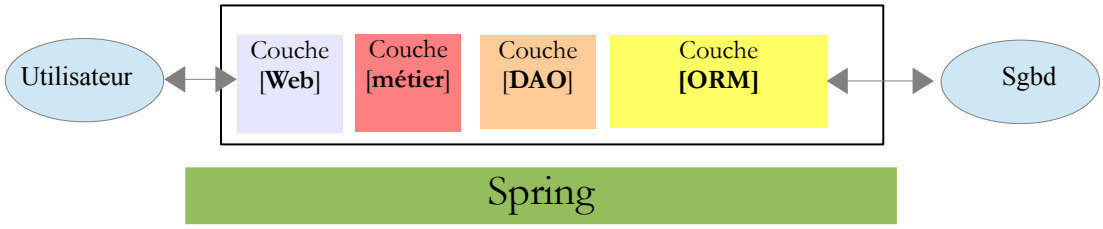
1. **demande** - les URL demandées sont de la forme *http://machine:port/contexte/Action/param1/param2/....?p1=v1&p2=v2&...* Le [Front Controller] utilise un fichier de configuration ou des annotations Java pour " router " la demande vers le bon contrôleur et la bonne action au sein de ce contrôleur. Pour cela, il utilise le champ [Action] de l'URL. Le reste de l'URL [*/param1/param2/...*] est formé de paramètres facultatifs qui seront transmis à l'action. Le **C** de MVC est ici la chaîne [Front Controller, Contrôleur, Action]. Si aucun contrôleur ne peut traiter l'action demandée, le serveur Web répondra que l'URL demandée n'a pas été trouvée.
2. **traitement**
   - l'action choisie peut exploiter les paramètres *parami* que le [Front Controller] lui a transmis. Ceux-ci peuvent provenir de plusieurs sources :
     - du chemin [*/param1/param2/...*] de l'URL,
     - des paramètres [*p1=v1&p2=v2*] de l'URL,
     - de paramètres postés par le navigateur avec sa demande ;
   - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche [métier] [2b]. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
     - une page d'erreur si la demande n'a pu être traitée correctement
     - une page de confirmation sinon
   - l'action demande à une certaine vue de s'afficher [3]. Cette vue va afficher des données qu'on appelle le **modèle de la vue**. C'est le **M** de MVC. L'action va créer ce modèle M [2c] et demander à une vue **V** de s'afficher [3] ;
3. **réponse** - la vue **V** choisie utilise le modèle **M** construit par l'action pour initialiser les parties dynamiques de la réponse HTML qu'elle doit envoyer au client puis envoie cette réponse.

Maintenant, précisons le lien entre architecture web MVC et architecture en couches. Selon la définition qu'on donne au **modèle**, ces deux concepts sont liés ou non. Prenons une application web Spring MVC à une couche :



Si nous implémentons la couche [Web] avec Spring MVC, nous aurons bien une architecture web MVC mais pas une architecture multicouche. Ici, la couche [web] s'occupera de tout : présentation, métier, accès aux données. Ce sont les actions qui feront ce travail.

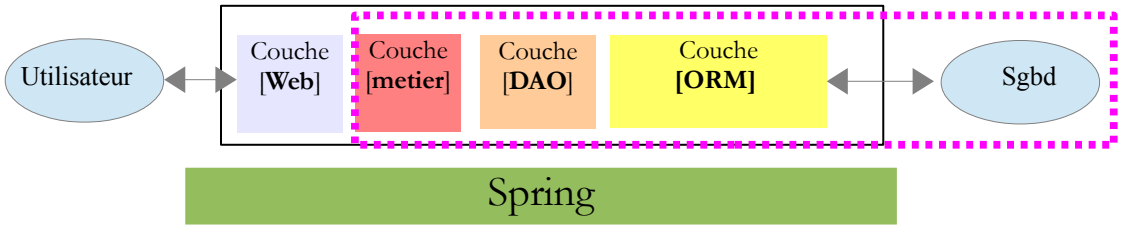
Maintenant, considérons une architecture Web multicouche :



La couche [Web] peut être implémentée sans framework et sans suivre le modèle MVC. On a bien alors une architecture multicouche mais la couche Web n'implémente pas le modèle MVC.

Par exemple, dans le monde .NET la couche [Web] ci-dessus peut être implémentée avec ASP.NET MVC et on a alors une architecture en couches avec une couche [Web] de type MVC. Ceci fait, on peut remplacer cette couche ASP.NET MVC par une couche ASP.NET classique (WebForms) tout en gardant le reste (métier, DAO, ORM) **à l'identique**. On a alors une architecture en couches avec une couche [Web] qui n'est plus de type MVC.

Dans MVC, nous avons dit que le modèle M était celui de la vue V, c.a.d. l'ensemble des données affichées par la vue V. Une autre définition du modèle M de MVC est donnée :



Beaucoup d'auteurs considèrent que ce qui est à droite de la couche [Web] forme le modèle M du MVC. Pour éviter les ambigüités on peut parler :

- du **modèle du domaine** lorsqu'on désigne tout ce qui est à droite de la couche [Web]
- du **modèle de la vue** lorsqu'on désigne les données affichées par une vue V

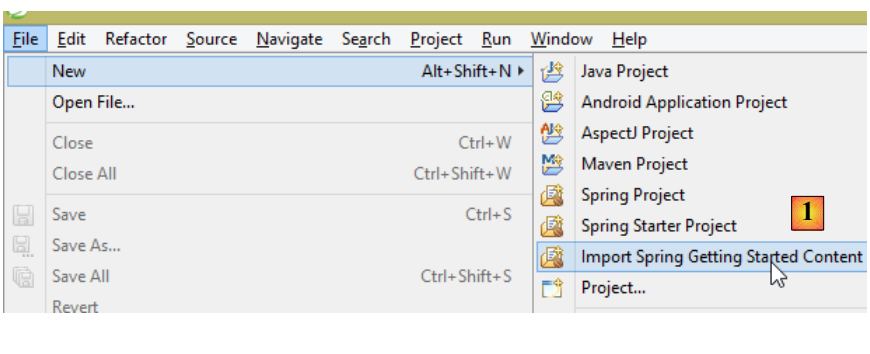
Dans la suite, le terme " modèle M " désignera exclusivement le **modèle d'une vue** V.

1.6 Un premier projet Spring MVC

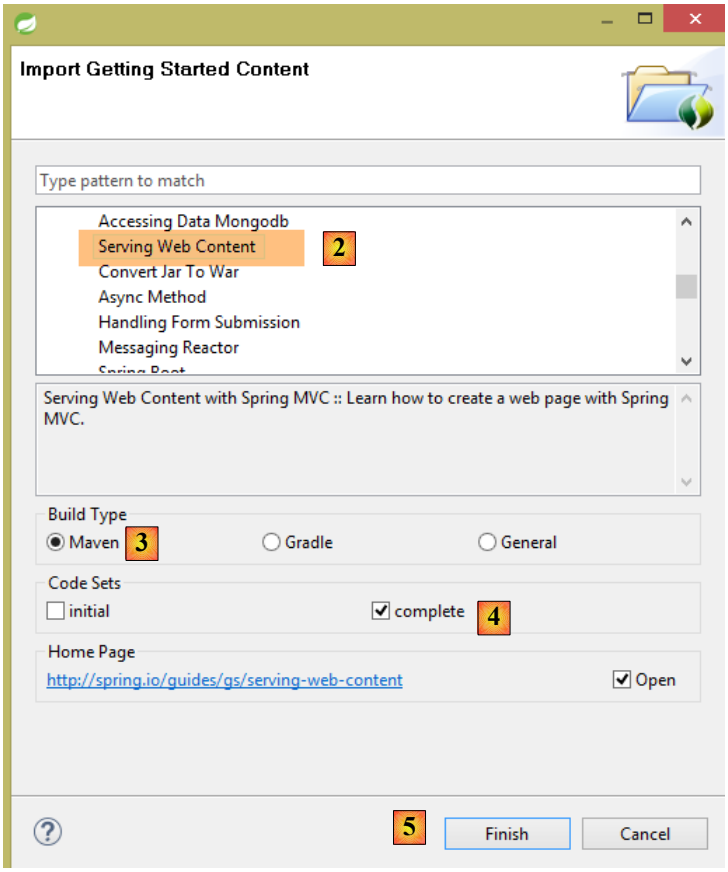
A partir de maintenant, nous travaillons avec l'IDE Spring Tool Suite (STS), une variante d'Eclipse personnalisée pour Spring. Le site [http://spring.io/guides] offre des tutoriels de démarrage pour découvrir l'écosystème Spring. Nous allons suivre l'un d'eux pour découvrir la configuration Maven nécessaire à un projet Spring MVC.

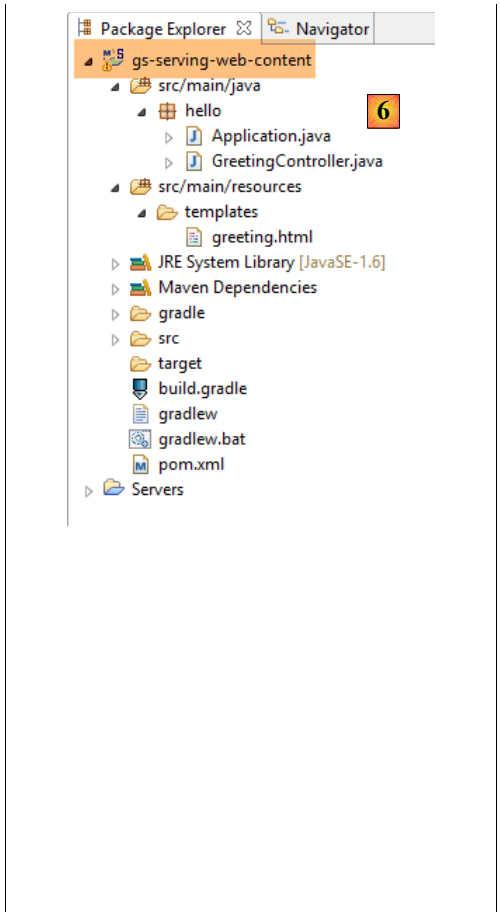
Note : la compréhension des détails du projet échappera à la plupart des débutants. Ce n'est pas important. Ces détails sont expliqués dans la suite du document. On se contentera de reproduire les manipulations.

1.6.1 Le projet de démonstration



- en [1], nous importons l'un des guides Spring ;





- en [2], nous choisissons l'exemple [Serving Web Content] ;
- en [3], on choisit le projet Maven ;
- en [4], on prend la version finale du guide ;
- en [5], on valide ;
- en [6], le projet importé ;

Examinons le projet, d'abord sa configuration Maven.

1.6.2 Configuration Maven

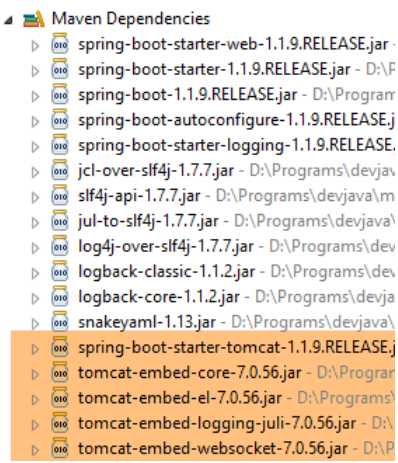
Le fichier [pom.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>org.springframework</groupId>
7.     <artifactId>gs-serving-web-content</artifactId>
8.     <version>0.1.0</version>
9.
10.    <parent>
11.       <groupId>org.springframework.boot</groupId>
12.       <artifactId>spring-boot-starter-parent</artifactId>
13.       <version>1.1.9.RELEASE</version>
14.    </parent>
15.
16.    <dependencies>
17.       <dependency>
18.          <groupId>org.springframework.boot</groupId>
19.          <artifactId>spring-boot-starter-thymeleaf</artifactId>
20.       </dependency>
21.    </dependencies>
22.
```

```
23.     <properties>
24.        <start-class>hello.Application</start-class>
25.     </properties>
26.
27.     <build>
28.        <plugins>
29.           <plugin>
30.              <groupId>org.springframework.boot</groupId>
31.              <artifactId>spring-boot-maven-plugin</artifactId>
32.           </plugin>
33.        </plugins>
34.     </build>
35.
36.     <repositories>
37.        <repository>
38.           <id>spring-milestone</id>
39.           <url>https://repo.spring.io/libs-release</url>
40.        </repository>
41.     </repositories>
42.
43.     <pluginRepositories>
44.        <pluginRepository>
45.           <id>spring-milestone</id>
46.           <url>https://repo.spring.io/libs-release</url>
47.        </pluginRepository>
48.     </pluginRepositories>
49.
50. </project>
```

- lignes 6-8 : les propriétés du projet Maven. Manque une balise [<packaging>] indiquant le type du fichier produit par la compilation Maven. En l'absence de celle-ci, c'est le type [jar] qui est utilisé. L'application est donc une application exécutable de type console, et non une application web où le packaging serait alors [war] ;
- lignes 10-14 : le projet Maven a un projet parent [spring-boot-starter-parent] C'est lui qui définit l'essentiel des dépendances du projet. Elles peuvent être suffisantes, auquel cas on n'en rajoute pas, ou pas, auquel cas on rajoute les dépendances manquantes ;
- lignes 17-20 : l'artifact [spring-boot-starter-thymeleaf] amène avec lui les bibliothèques nécessaires à un projet spring MVC utilisé conjointement avec un moteur de vues appelé [Thymeleaf]. Cet artifact amène avec lui un très grand de bibliothèques dont celles d'un serveur Tomcat embarqué. C'est sur ce serveur que l'application sera exécutée ;

Les bibliothèques amenées par cette configuration sont très nombreuses :



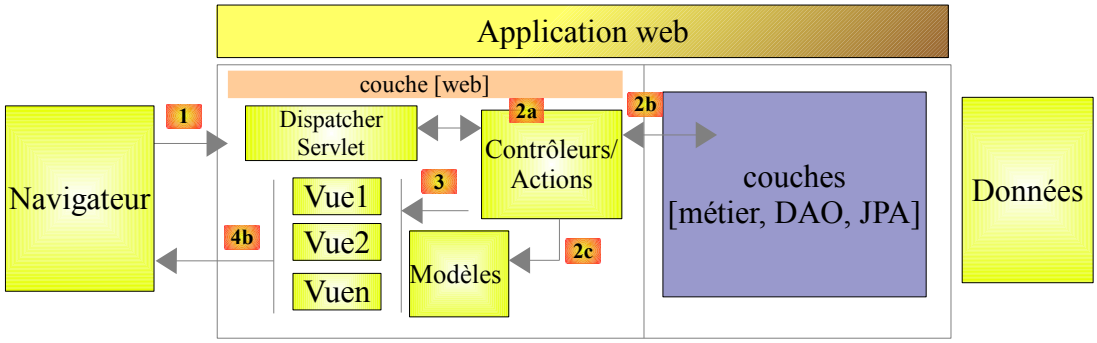
Ci-dessus on voit les archives du serveur Tomcat.

Spring Boot est une branche de l'écosystème Spring [http://projects.spring.io/spring-boot/]. Ce projet vise à diminuer au maximum la configuration des projets Spring. Pour cela, Spring Boot fait de l'auto-configuration à partir des dépendances présentes dans le Classpath du projet. Spring Boot fournit de nombreuses dépendances prêtes à l'emploi. Ainsi la dépendance [spring-boot-starter-thymeleaf] trouvée dans le projet Maven précédent amène toutes les dépendances nécessaires à une application Spring MVC utilisant le moteur de vues [Thymeleaf]. Avec ces deux caractéristiques :

- dépendances prêtes à l'emploi ;
- auto-configuration faite à partir de ces dépendances et de valeurs par défaut 'raisonnables', on peut avoir très rapidement une application Spring MVC opérationnelle. C'est le cas du projet étudié ici ;

1.6.3 L'architecture d'une application Spring MVC

Spring MVC implémente le modèle d'architecture dit MVC (Modèle – Vue – Contrôleur) :

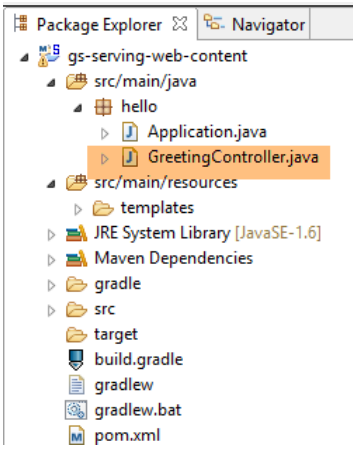


Le traitement d'une demande d'un client se déroule de la façon suivante :

1. **demande** - les URL demandées sont de la forme *http://machine:port/contexte/Action/param1/param2/....?p1=v1&p2=v2&...* La [Dispatcher Servlet] est la classe de Spring qui traite les URL entrantes. Elle "route" l'URL vers l'action qui doit la traiter. Ces actions sont des méthodes de classes particulières appelées [Contrôleurs]. Le **C** de MVC est ici la chaîne [Dispatcher Servlet, Contrôleur, Action]. Si aucune action n'a été configurée pour traiter l'URL entrante, la servlet [Dispatcher Servlet] répondra que l'URL demandée n'a pas été trouvée (erreur 404 NOT FOUND) ;
2. **traitement**
    - l'action choisie peut exploiter les paramètres *parami* que la servlet [Dispatcher Servlet] lui a transmis. Ceux-ci peuvent provenir de plusieurs sources :
        - du chemin [*/param1/param2/*...] de l'URL,
        - des paramètres [*p1=v1&p2=v2*] de l'URL,
        - de paramètres postés par le navigateur avec sa demande ;
    - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche [metier] [2b]. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
        - une page d'erreur si la demande n'a pu être traitée correctement
        - une page de confirmation sinon
    - l'action demande à une certaine vue de s'afficher [3]. Cette vue va afficher des données qu'on appelle le **modèle de la vue**. C'est le **M** de MVC. L'action va créer ce modèle M [2c] et demander à une vue **V** de s'afficher [3] ;
3. **réponse** - la vue **V** choisie utilise le modèle **M** construit par l'action pour initialiser les parties dynamiques de la réponse HTML qu'elle doit envoyer au client puis envoie cette réponse.

Nous allons regarder ces différents éléments dans le projet étudié.

1.6.4 Le contrôleur C

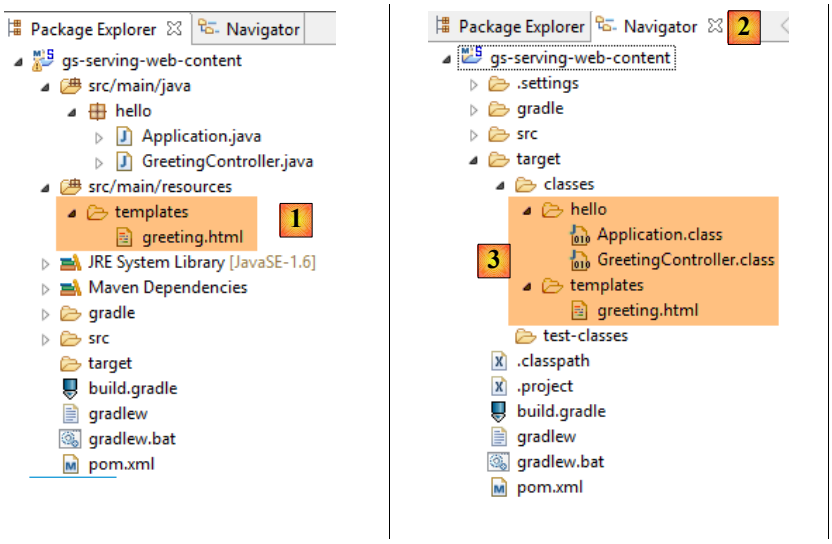


L'application importée a le contrôleur suivant :

```
1.  package hello;
2.
3.  import org.springframework.stereotype.Controller;
4.  import org.springframework.ui.Model;
5.  import org.springframework.web.bind.annotation.RequestMapping;
6.  import org.springframework.web.bind.annotation.RequestParam;
7.
8.  @Controller
9.  public class GreetingController {
10.
11.     @RequestMapping("/greeting")
12.     public String greeting(@RequestParam(value="name", required=false, defaultValue="World") String name,
    Model model) {
13.         model.addAttribute("name", name);
14.         return "greeting";
15.     }
16.
17. }
```

- ligne 8 : l'annotation [**@Controller**] fait de la classe [GreetingController] un contrôleur Spring, ç-à-d que ses méthodes sont enregistrées pour traiter des URL ;
- ligne 11 : l'annotation [**@RequestMapping**] indique l'URL que traite la méthode, ici l'URL [/greeting]. Nous verrons ultérieurement que cette URL peut être paramétrée et qu'il est possible de récupérer ces paramètres ;
- ligne 12 : la méthode admet deux paramètres :
  - [String name] : ce paramètre est initialisé par un paramètre de nom [name] dans la requête traitée, par exemple [/greeting?name=alfonse]. Ce paramètre est facultatif [required=**false**] et lorsqu'il n'est pas là, le paramètre [name] prendra la valeur 'World' [defaultValue="World"],
  - [Model model] est un modèle de vue. Il arrive vide et c'est le rôle de l'action (la méthode *greeting*) de le remplir. C'est ce modèle qui sera transmis à la vue que va faire afficher l'action. C'est donc un modèle de vue ;
- ligne 13 : la valeur de [name] est mis dans le modèle de la vue. La classe [Model] se comporte comme un dictionnaire ;
- ligne 14 : la méthode rend le nom de la vue qui doit afficher le modèle construit. Le nom exact de la vue dépend de la configuration de [Thymeleaf]. En l'absence de celle-ci, la vue affichée ici sera la vue [/templates/greeting.html] ou le dossier [templates] doit être à la racine du Classpath du projet ;

Examinons notre projet Eclipse :



Les dossiers [src/main/java] et [src/main/resources] sont tous deux des dossiers dont le contenu sera mis dans le Classpath du projet. Pour [src/main/java] ce sera les versions compilées des sources Java qui y seront mis. Le contenu du dossier [src/main/resources] est lui mis dans le Classpath sans modification. On voit donc que le dossier [templates] sera dans le Classpath du projet [1].

On peut vérifier cela [2-3] dans la fenêtre [Navigator] d'Eclipse [Window / Show view / Other / General / Navigator]. Le dossier [target] est produit par la compilation (appelée *build*) du projet. Le dossier [classes] représente la racine du Classpath. On voit que le dossier [templates] y est présent.

1.6.5 La vue V

Dans le MVC, nous venons de voir le contrôleur C et le modèle de vue M. La vue V est ici représentée par le fichier [greeting.html] suivant :

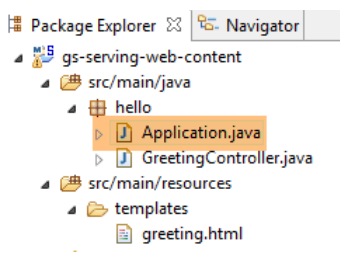
```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title>Getting Started: Serving Web Content</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.      <p th:text="'Hello, ' + ${name} + '!'" />
9.  </body>
10. </html>
```

- ligne 2 : l'espace de noms des balises Thymeleaf ;
- ligne 8 : une balise <p> (paragraphe) avec un attribut Thymeleaf. L'attribut [th:text] fixe le contenu du paragraphe. A l'intérieur de la chaîne de caractères on a l'expression [${name}]. Cela signifie qu'on veut la valeur de l'attribut [name] du modèle de la vue. Or on se souvient que cet attribut a été placé dans le modèle par l'action :

```
model.addAttribute("name", name);
```

Le premier paramètre fixe le nom de l'attribut, le second sa valeur.

1.6.6 Exécution

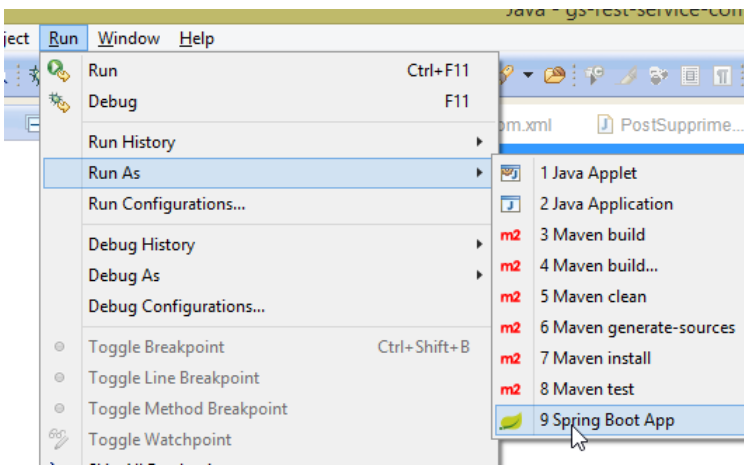


La classe [Application.java] est la classe exécutable du projet. Son code est le suivant :

```
1.  package hello;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.boot.SpringApplication;
5.  import org.springframework.context.annotation.ComponentScan;
6.
7.  @ComponentScan
8.  @EnableAutoConfiguration
9.  public class Application {
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(Application.class, args);
13.     }
14.
15. }
```

- ligne 11 : la classe est exécutable avec une méthode [main] propre aux applications console. La classe [SpringApplication] de la ligne 12 va lancer le serveur Tomcat présent dans les dépendances et déployer le service web dessus ;
- ligne 4 : on voit que la classe [SpringApplication] appartient au projet [Spring Boot] ;
- ligne 12 : le premier paramètre est la classe qui configure le projet, le second d'éventuels paramètres ;
- ligne 8 : l'annotation [@EnableAutoConfiguration] demande à Spring Boot de faire la configuration du projet ;
- ligne 7 : l'annotation [@ComponentScan] fait que le dossier qui contient la classe [Application] va être exploré pour rechercher les composants Spring. Un sera trouvé, la classe [GreetingController] qui a l'annotation [@Controller] qui en fait un composant Spring ;

Exécutons le projet :



On obtient les logs console suivants :

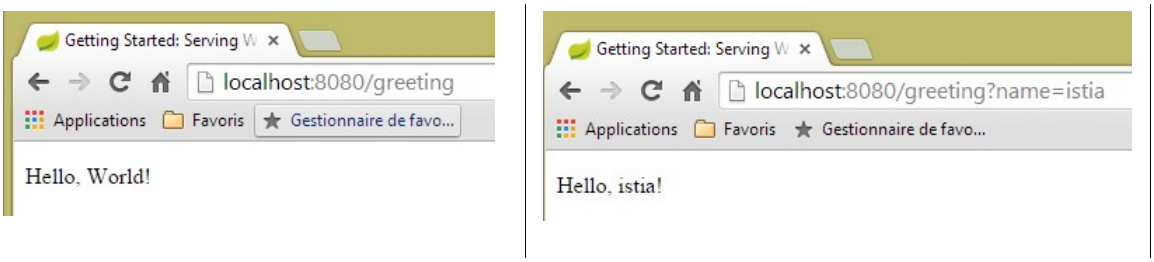
```
1.  .   ____          _            __ _ _
```

```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.1.9.RELEASE)

2014-11-27 16:48:12.567  INFO 3908 --- [           main] hello.Application                        :
Starting Application on Gportpers3 with PID 3908 (started by ST in D:\data\istia-1415\spring mvc\dvp\gs-
serving-web-content-complete)
2014-11-27 16:48:12.723  INFO 3908 --- [           main] ationConfigEmbeddedWebApplicationContext :
Refreshing
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@1a38c59b:
startup date [Thu Nov 27 16:48:12 CET 2014]; root of context hierarchy
2014-11-27 16:48:13.813  INFO 3908 --- [           main] o.s.b.f.s.DefaultListableBeanFactory     :
Overriding bean definition for bean 'beanNameViewResolver': replacing [Root bean: class [null]; scope=;
abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true;
primary=false;
factoryBeanName=org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorView
Configuration; factoryMethodName=beanNameViewResolver; initMethodName=null;
destroyMethodName=(inferred); defined in class path resource
[org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration.c
lass]] with [Root bean: class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3;
dependencyCheck=0; autowireCandidate=true; primary=false;
factoryBeanName=org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurati
onAdapter; factoryMethodName=beanNameViewResolver; initMethodName=null; destroyMethodName=(inferred);
defined in class path resource
[org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter.class
]]
2014-11-27 16:48:15.247  INFO 3908 --- [           main] .t.TomcatEmbeddedServletContainerFactory :
Server initialized with port: 8080
2014-11-27 16:48:15.574  INFO 3908 --- [           main] o.apache.catalina.core.StandardService   :
Starting service Tomcat
2014-11-27 16:48:15.575  INFO 3908 --- [           main] org.apache.catalina.core.StandardEngine  :
Starting Servlet Engine: Apache Tomcat/7.0.56
2014-11-27 16:48:15.955  INFO 3908 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
Initializing Spring embedded WebApplicationContext
2014-11-27 16:48:15.955  INFO 3908 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root
WebApplicationContext: initialization completed in 3236 ms
2014-11-27 16:48:16.918  INFO 3908 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
Mapping servlet: 'dispatcherServlet' to [/]
2014-11-27 16:48:16.922  INFO 3908 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2014-11-27 16:48:17.354  INFO 3908 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
Mapped URL path [/**/favicon.ico] onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2014-11-27 16:48:17.679  INFO 3908 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/greeting],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
java.lang.String hello.GreetingController.greeting(java.lang.String,org.springframework.ui.Model)
2014-11-27 16:48:17.681  INFO 3908 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequ
est)
2014-11-27 16:48:17.682  INFO 3908 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[text/html],custom=[]}" onto
public org.springframework.web.servlet.ModelAndView
org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServlet
Request)
2014-11-27 16:48:17.696  INFO 3908 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
Mapped URL path [/webjars/**] onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2014-11-27 16:48:17.697  INFO 3908 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
Mapped URL path [/**] onto handler of type [class
org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2014-11-27 16:48:18.159  INFO 3908 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
Registering beans for JMX exposure on startup
2014-11-27 16:48:18.491  INFO 3908 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 8080/http
2014-11-27 16:48:18.493  INFO 3908 --- [           main] hello.Application                        :
Started Application in 6.833 seconds (JVM running for 8.658)
```

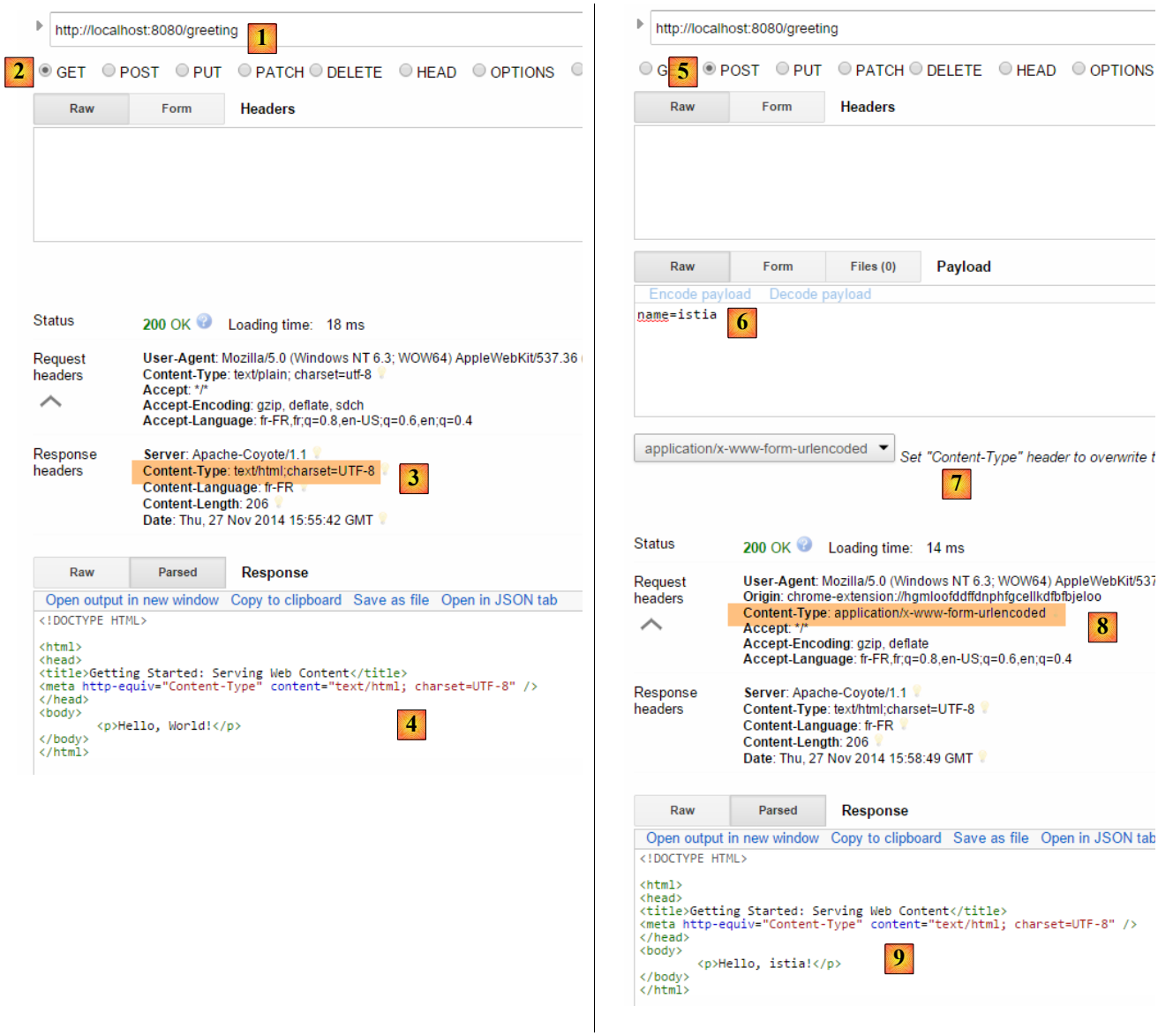
- ligne 13 : le serveur Tomcat démarre sur le port 8080 (ligne 12) ;
- ligne 17 : la servlet [DispatcherServlet] est présente ;

- ligne 20 : la méthode [hello.GreetingController.greeting] a été découverte ainsi que l'URL qu'elle traite [/greeting] ;

Pour tester l'application web, on demande l'URL [http://localhost:8080/greeting] :



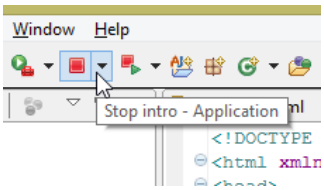
Il peut être intéressant de voir les entêtes HTTP envoyés par le serveur. Pour cela, on va utiliser le plugin de Chrome appelé [Advanced Rest Client] (cf paragraphe 9.6, page 606) :

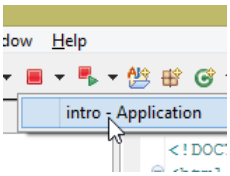


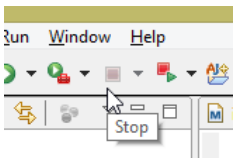
- en [1], l'URL demandée ;
- en [2], la méthode GET est utilisée ;
- en [3], le serveur a indiqué qu'il envoyait une réponse au format HTML ;
- en [4], la réponse HTML ;
- en [5], on demande la même URL mais cette fois-ci avec un POST ;

- en [7], les informations sont envoyées au serveur sous la forme [urlencoded] ;
- en [6], le paramètre **name** avec sa valeur ;
- en [8], le navigateur indique au serveur qu'il lui envoie des informations [urlencoded] ;
- en [9], la réponse HTML du serveur ;

Pour arrêter l'application :







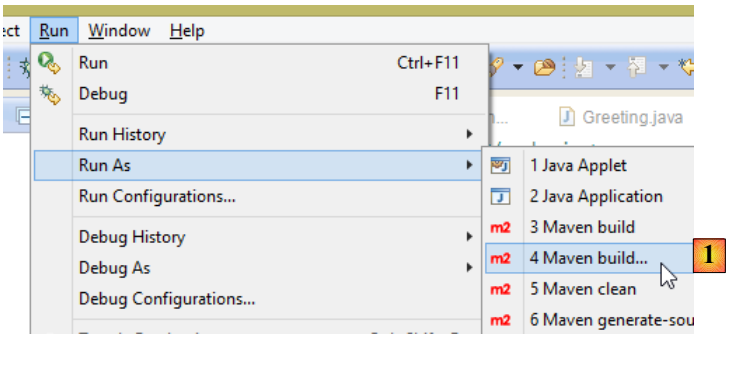
1.6.7 Création d'une archive exécutable

Il est possible de créer une archive exécutable en-dehors d'Eclipse. La configuration nécessaire est dans le fichier [pom.xml] :

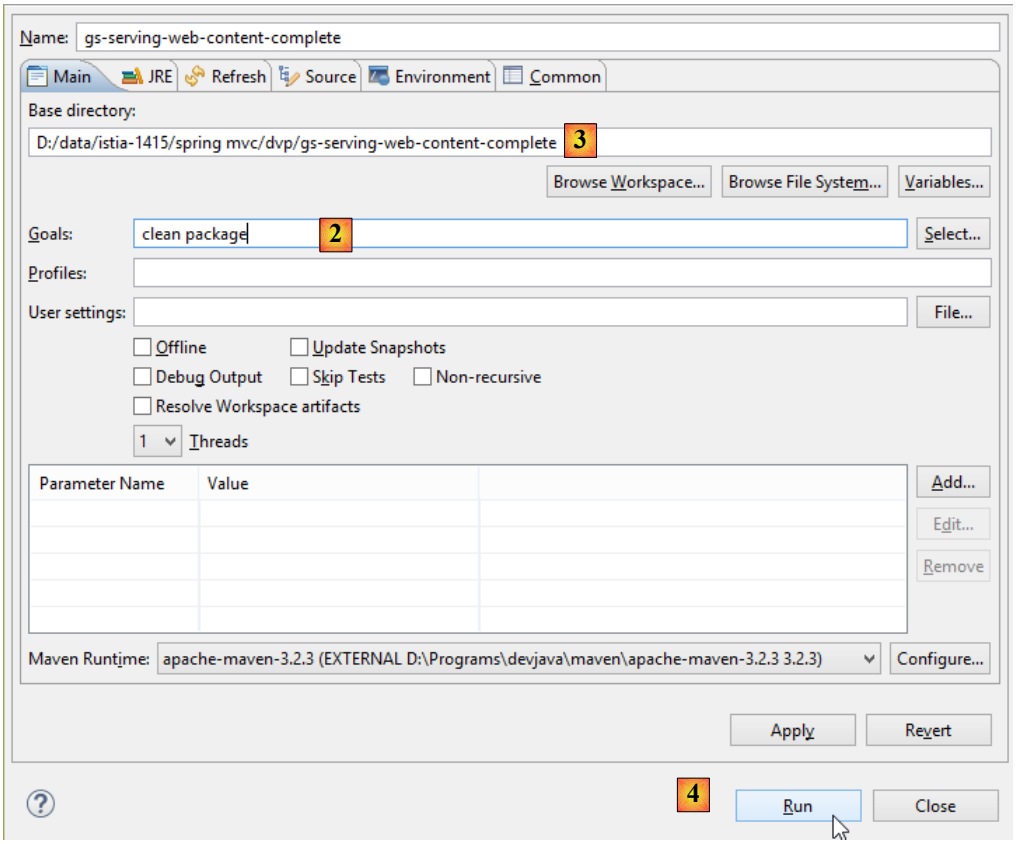
```
1.      <properties>
2.         <start-class>hello.Application</start-class>
3.      </properties>
4.
5.      <build>
6.         <plugins>
7.            <plugin>
8.               <groupId>org.springframework.boot</groupId>
9.               <artifactId>spring-boot-maven-plugin</artifactId>
10.           </plugin>
11.        </plugins>
12. </build>
```

- les lignes 7-10 définissent le plugin qui va créer l'archive exécutable ;
- la ligne 2 définit la classe exécutable du projet ;

On procède ainsi :

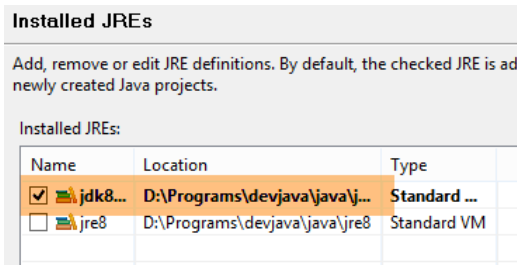


- en [1] : on exécute une cible Maven ;



- en [2] : il y a deux cibles (goals) : [clean] pour supprimer le dossier [target] du projet Maven, [package] pour le régénérer ;
- en [3] : le dossier [target] généré le sera dans ce dossier ;
- en [4] : on génère la cible ;

Note : pour que la génération réussisse, il faut que la JVM utilisée par STS soit un **JDK** [Window / Preferences / Java / Installed JREs] :



Dans les logs qui apparaissent dans la console, il est important de voir apparaître le plugin [`spring-boot-maven-plugin`]. C'est lui qui génère l'archive exécutable.

```
[INFO] --- spring-boot-maven-plugin:1.1.9.RELEASE:repackage (default) @ gs-serving-web-content ---
```

Avec une console, on se place dans le dossier généré :

```
1.  gs-serving-web-content-complete\target>dir
2.   ...
3.
4.   Répertoire de D:\data\istia-1415\spring mvc\dvp\gs-serving-web-content-complete
5.  \target
6.
7.  27/11/2014  17:07    <DIR>          .
```

```
8.  27/11/2014  17:07    <DIR>          ..
9.  27/11/2014  17:07    <DIR>          classes
10. 27/11/2014  17:07    <DIR>          generated-sources
11. 27/11/2014  17:07        13 419 551 gs-serving-web-content-0.1.0.jar
12. 27/11/2014  17:07             3 522 gs-serving-web-content-0.1.0.jar.original
13. 27/11/2014  17:07    <DIR>          maven-archiver
14. 27/11/2014  17:07    <DIR>          maven-status
```

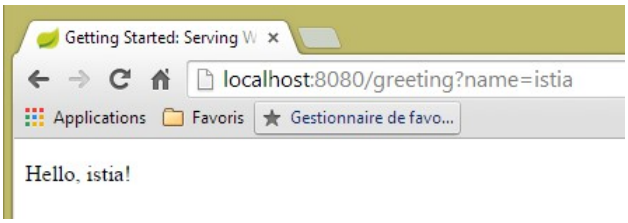
- ligne 12 : l'archive générée ;

Cette archive est exécutée de la façon suivante :

```
1.  gs-serving-web-content-complete\target>java -jar gs-serving-web-content-0.1.0.jar
2.
3.    .   ____          _            __ _ _
4.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
5.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
6.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
7.    '  |____| .__|_| |_|_| |_\__, | / / / /
8.   =========|_|==============|___/=/_/_/_/
9.   :: Spring Boot ::        (v1.1.9.RELEASE)
10.
11. 2014-11-27 17:14:50.439  INFO 8172 --- [           main] hello.Application                        :
    Starting Application on Gportpers3 with PID 8172 (D:\data\istia-1415\spring mvc\dvp\gs-serving-web-
    content-complete\target\gs-serving-web-content-0.1.0.jar started by ST in D:\data\istia-1415\spring
    mvc\dvp\gs-serving-web-content-complete\target)
12. 2014-11-27 17:14:50.491  INFO 8172 --- [           main] ationConfigEmbeddedWebApplicationContext :
    Refreshing
    org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@12f4ec3a: startup
    date [Thu Nov 27 17:14:50 CET 2014]; root of context hierarchy
```

Note : il faut auparavant arrêter le service web éventuellement lancé dans Eclipse (cf page 23).

Maintenant que l'application web est lancée, on peut l'interroger avec un navigateur :



1.6.8 Déployer l'application sur un serveur Tomcat

Si Spring Boot s'avère très pratique en mode développement, une application en production sera déployée sur un vrai serveur Tomcat. Voici comment procéder :

Modifier le fichier [pom.xml] de la façon suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>org.springframework</groupId>
7.     <artifactId>gs-serving-web-content</artifactId>
8.     <version>0.1.0</version>
9.     <packaging>war</packaging>
10.
11.    <parent>
12.       <groupId>org.springframework.boot</groupId>
13.       <artifactId>spring-boot-starter-parent</artifactId>
14.       <version>1.1.9.RELEASE</version>
15.    </parent>
16.
```

```
17.     <dependencies>
18.        <!-- environnement Thymeleaf -->
19.        <dependency>
20.           <groupId>org.springframework.boot</groupId>
21.           <artifactId>spring-boot-starter-thymeleaf</artifactId>
22.        </dependency>
23.        <!-- génération du war -->
24. <!--        <dependency>
25.           <groupId>org.springframework.boot</groupId>
26.           <artifactId>spring-boot-starter-tomcat</artifactId>
27.           <scope>provided</scope>
28.        </dependency> -->
29.     </dependencies>
30.
31.     <properties>
32.        <start-class>hello.Application</start-class>
33.     </properties>
34.
35.     <build>
36.        <plugins>
37.           <plugin>
38.              <groupId>org.springframework.boot</groupId>
39.              <artifactId>spring-boot-maven-plugin</artifactId>
40.           </plugin>
41.        </plugins>
42.     </build>
43.
44.     <repositories>
45.        <repository>
46.           <id>spring-milestone</id>
47.           <url>https://repo.spring.io/libs-release</url>
48.        </repository>
49.     </repositories>
50.
51.     <pluginRepositories>
52.        <pluginRepository>
53.           <id>spring-milestone</id>
54.           <url>https://repo.spring.io/libs-release</url>
55.        </pluginRepository>
56.     </pluginRepositories>
57.
58. </project>
```

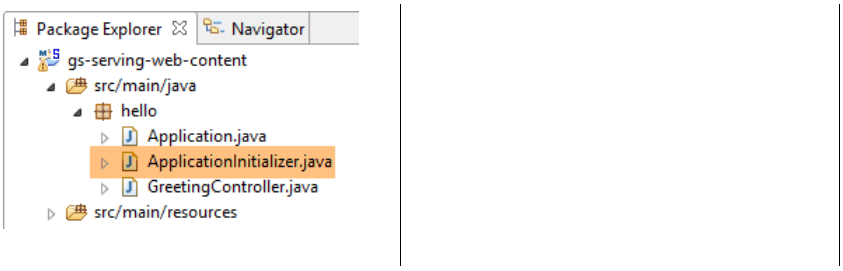
Les modifications sont à faire à deux endroits :

- ligne 9 : il faut indiquer qu'on va générer une archive war (Web ARchive) ;
- lignes 24-28 : il faut ajouter une dépendance sur l'artifact [spring-boot-starter-tomcat]. Cet artifact amène toutes les classes de Tomcat dans les dépendances du projet ;
- ligne 27 : cet artifact est [provided], ç-à-d que les archives correspondantes ne seront pas placées dans le war généré. En effet, ces archives seront trouvées sur le serveur Tomcat sur lequel s'exécutera l'application ;

En fait, si on regarde les dépendances actuelles du projet, on constate que la dépendance [spring-boot-starter-tomcat] est déjà présente :

Il n'y a donc pas lieu de la rajouter dans le fichier [pom.xml]. On l'a mise en commentaires pour mémoire.

Il faut par ailleurs configurer l'application web. En l'absence de fichier [web.xml], cela se fait avec une classe héritant de [`SpringBootServletInitializer`] :

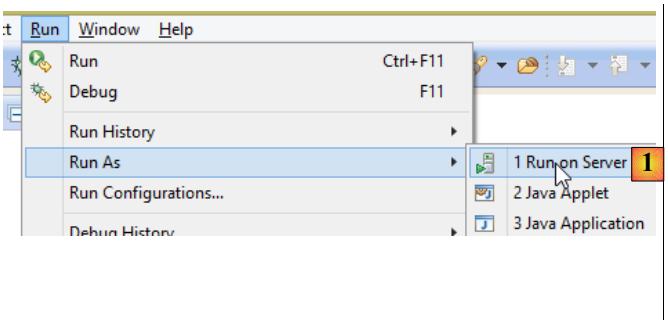


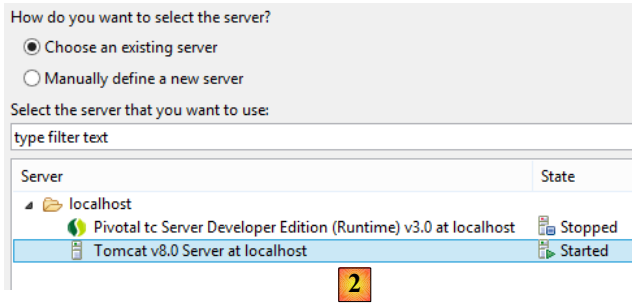
La classe [ApplicationInitializer] est la suivante :

```
1.  package hello;
2.
3.  import org.springframework.boot.builder.SpringApplicationBuilder;
4.  import org.springframework.boot.context.web.SpringBootServletInitializer;
5.
6.  public class ApplicationInitializer extends SpringBootServletInitializer {
7.
8.      @Override
9.      protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10.         return application.sources(Application.class);
11.     }
12.
13. }
```

- ligne 6 : la classe [ApplicationInitializer] étend la classe [SpringBootServletInitializer] ;
- ligne 9 : la méthode [configure] est redéfinie (ligne 8) ;
- ligne 10 : on fournit la classe qui configure le projet ;

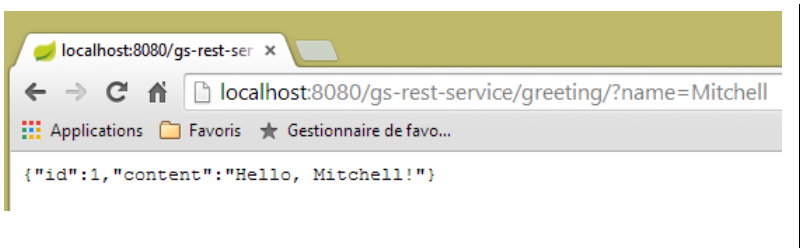
Pour exécuter le projet, on peut procéder ainsi :





- en [1], on exécute le projet sur l'un des serveurs enregistrés dans l'IDE Eclipse ;
- en [2], on choisit ci-dessus [Tomcat v8.0] ;

Ceci fait, on peut demander l'URL [http://localhost:8080/gs-rest-service/greeting/?name=Mitchell] dans un navigateur :

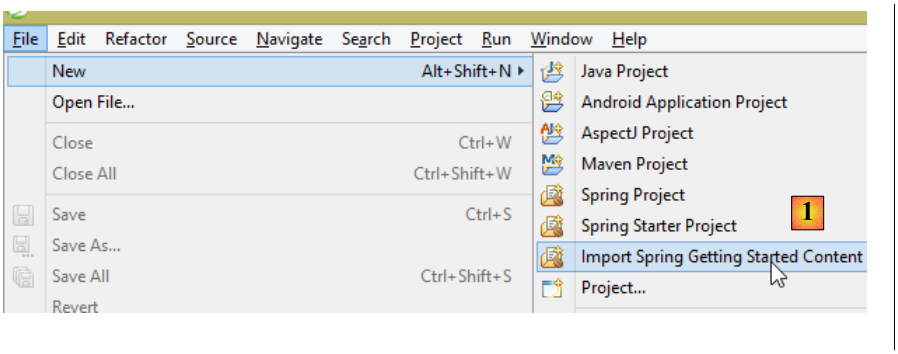


Note : selon les versions de [tomcat] et [tc Server Developer], cette exécution peut échouer. Cela a été le cas avec [Apache Tomcat 8.0.3 et 8.0.15] par exemple. Ci-dessus la version de Tomcat utilisée était la [8.0.9].

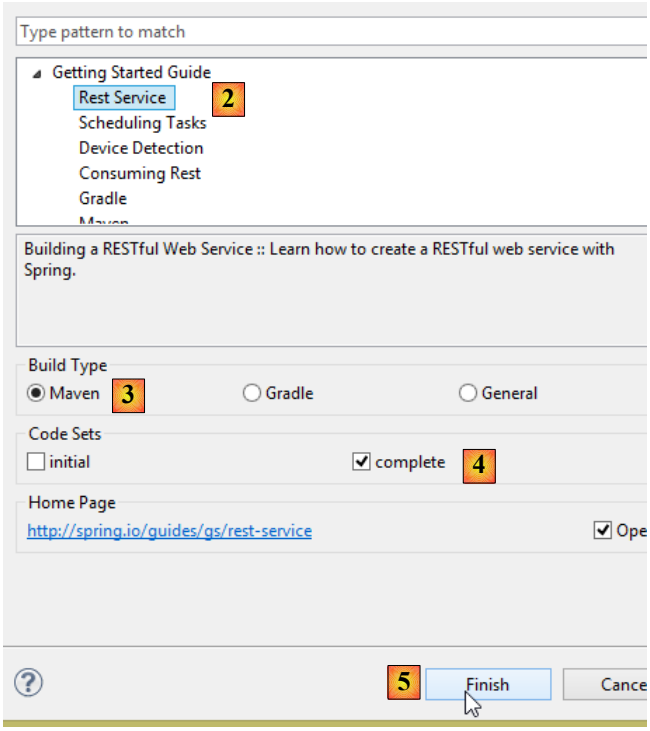
Nous savons désormais générer une archive war. Par la suite, nous continuerons à travailler avec Spring Boot et son archive jar exécutable.

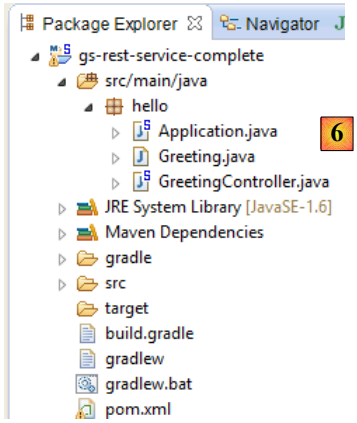
1.7 Un second projet Spring MVC

1.7.1 Le projet de démonstration



- en [1], nous importons l'un des guides Spring ;





- en [2], nous choisissons l'exemple [Rest Service] ;
- en [3], on choisit le projet Maven ;
- en [4], on prend la version finale du guide ;
- en [5], on valide ;
- en [6], le projet importé ;

Les services web accessibles via des URL standard et qui délivrent du texte jSON sont souvent appelés des services REST (REpresentational State Transfer). Dans ce document, je me contenterai d'appeler le service que nous allons construire, un service web / jSON. Un service est dit Restful s'il respecte certaines règles. Je n'ai pas cherché à respecter celles-ci.

Examinons maintenant le projet importé, d'abord sa configuration Maven.

1.7.2 Configuration Maven

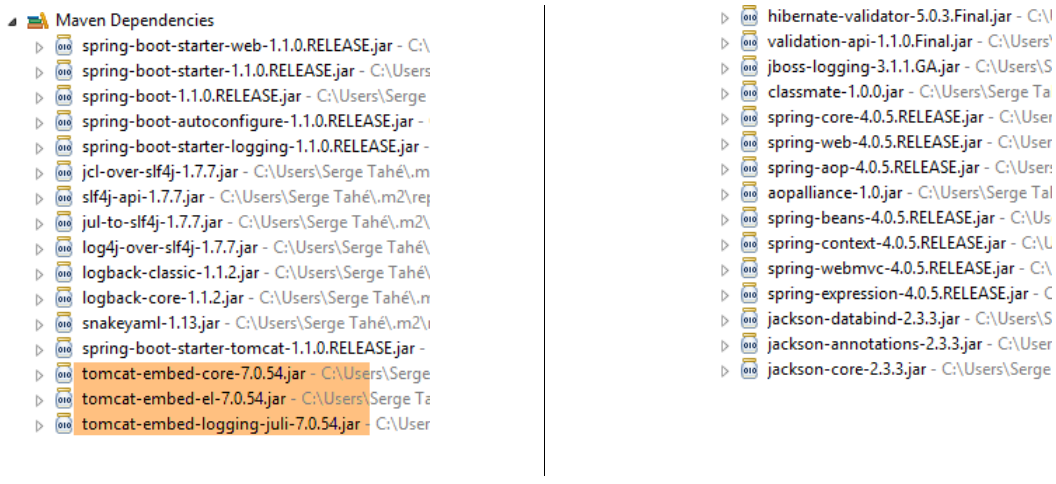
Le fichier [pom.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>org.springframework</groupId>
7.     <artifactId>gs-rest-service</artifactId>
8.     <version>0.1.0</version>
9.
10.    <parent>
11.       <groupId>org.springframework.boot</groupId>
12.       <artifactId>spring-boot-starter-parent</artifactId>
13.       <version>1.1.9.RELEASE</version>
14.    </parent>
15.
16.    <dependencies>
17.       <dependency>
18.          <groupId>org.springframework.boot</groupId>
19.          <artifactId>spring-boot-starter-web</artifactId>
20.       </dependency>
21.    </dependencies>
22.
23.    <properties>
```

```
24.         <start-class>hello.Application</start-class>
25.     </properties>
26.
27.     <build>
28.         <plugins>
29.             <plugin>
30.                 <groupId>org.springframework.boot</groupId>
31.                 <artifactId>spring-boot-maven-plugin</artifactId>
32.             </plugin>
33.         </plugins>
34.     </build>
35.
36.     <repositories>
37.         <repository>
38.             <id>spring-releases</id>
39.             <url>https://repo.spring.io/libs-release</url>
40.         </repository>
41.     </repositories>
42.     <pluginRepositories>
43.         <pluginRepository>
44.             <id>spring-releases</id>
45.             <url>https://repo.spring.io/libs-release</url>
46.         </pluginRepository>
47.     </pluginRepositories>
48. </project>
```

- lignes 6-8 : les propriétés du projet Maven. Manque une balise [<packaging>] indiquant le type du fichier produit par la compilation Maven. En l'absence de celle-ci, c'est le type [jar] qui est utilisé. L'application est donc une application exécutable de type console, et non une application web où le packaging serait alors [war] ;
- lignes 10-14 : le projet Maven a un projet parent [spring-boot-starter-parent]. C'est lui qui définit l'essentiel des dépendances du projet. Elles peuvent être suffisantes, auquel cas on n'en rajoute pas, ou pas, auquel cas on rajoute les dépendances manquantes ;
- lignes 17-20 : l'artifact [spring-boot-starter-web] amène avec lui les bibliothèques nécessaires à un projet Spring MVC de type service web où il n'y a pas de vues générées. Cet artifact amène avec lui un très grand de bibliothèques dont celles d'un serveur Tomcat embarqué. C'est sur ce serveur que l'application sera exécutée ;

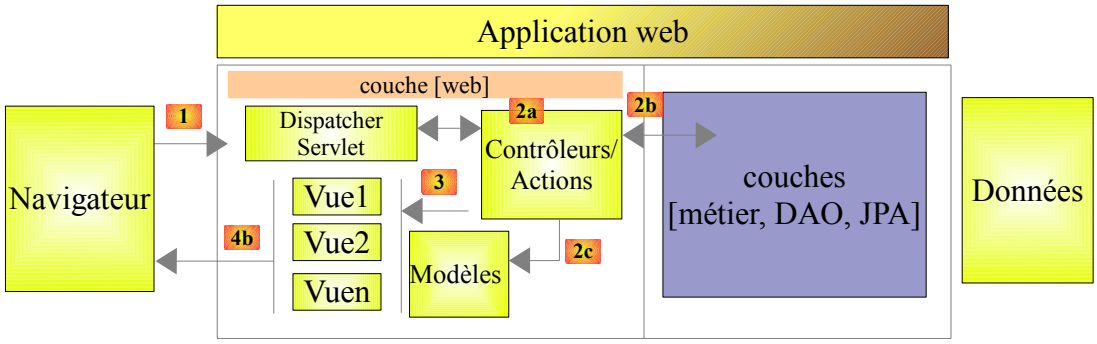
Les bibliothèques amenées par cette configuration sont très nombreuses :



Ci-dessus on voit les trois archives du serveur Tomcat.

1.7.3 L'architecture d'un service Spring [web / jSON]

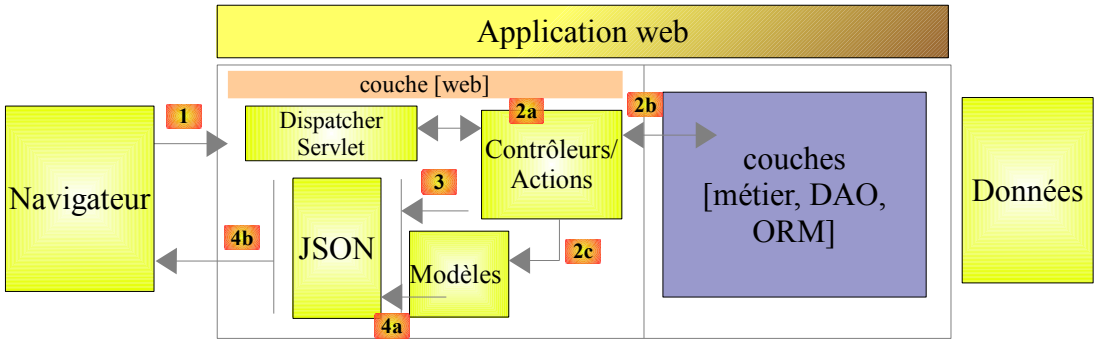
Rappelons comment Spring MVC implémente le modèle MVC :



Le traitement d'une demande d'un client se déroule de la façon suivante :

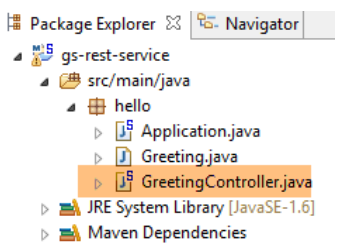
1. **demande** - les URL demandées sont de la forme *http://machine:port/contexte/Action/param1/param2/....?p1=v1&p2=v2&...* La [Dispatcher Servlet] est la classe de Spring qui traite les URL entrantes. Elle "route" l'URL vers l'action qui doit la traiter. Ces actions sont des méthodes de classes particulières appelées [Contrôleurs]. Le **C** de MVC est ici la chaîne [Dispatcher Servlet, Contrôleur, Action]. Si aucune action n'a été configurée pour traiter l'URL entrante, la servlet [Dispatcher Servlet] répondra que l'URL demandée n'a pas été trouvée (erreur 404 NOT FOUND) ;
2. **traitement**
   - l'action choisie peut exploiter les paramètres *parami* que la servlet [Dispatcher Servlet] lui a transmis. Ceux-ci peuvent provenir de plusieurs sources :
     - du chemin [*/param1/param2/...*] de l'URL,
     - des paramètres [*p1=v1&p2=v2*] de l'URL,
     - de paramètres postés par le navigateur avec sa demande ;
   - dans le traitement de la demande de l'utilisateur, l'action peut avoir besoin de la couche [metier] [2b]. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
     - une page d'erreur si la demande n'a pu être traitée correctement
     - une page de confirmation sinon
   - l'action demande à une certaine vue de s'afficher [3]. Cette vue va afficher des données qu'on appelle le **modèle de la vue**. C'est le **M** de MVC. L'action va créer ce modèle M [2c] et demander à une vue **V** de s'afficher [3] ;
3. **réponse** - la vue **V** choisie utilise le modèle **M** construit par l'action pour initialiser les parties dynamiques de la réponse HTML qu'elle doit envoyer au client puis envoie cette réponse.

Pour un service web / jSON, l'architecture précédente est légèrement modifiée :



- en [4a], le modèle qui est une classe Java est transformé en chaîne jSON par une bibliothèque jSON ;
- en [4b], cette chaîne jSON est envoyée au navigateur ;

1.7.4 Le contrôleur C



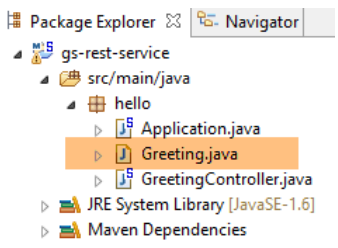
L'application importée a le contrôleur suivant :

```
1.  package hello;
2.
3.  import java.util.concurrent.atomic.AtomicLong;
4.  import org.springframework.web.bind.annotation.RequestMapping;
5.  import org.springframework.web.bind.annotation.RequestParam;
6.  import org.springframework.web.bind.annotation.RestController;
7.
8.  @RestController
9.  public class GreetingController {
10.
11.     private static final String template = "Hello, %s!";
12.     private final AtomicLong counter = new AtomicLong();
13.
14.     @RequestMapping("/greeting")
15.     public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {
16.         return new Greeting(counter.incrementAndGet(), String.format(template, name));
17.     }
18. }
```

- ligne 9 : l'annotation [**@RestController**] fait de la classe [GreetingController] un contrôleur Spring, ç-à-d que ses méthodes sont enregistrées pour traiter des URL. Nous avons vu l'annotation similaire [**@Controller**]. Le résultat des méthodes de ce contrôleur était un type [String] qui était le nom de la vue à afficher. Ici c'est différent. Les méthodes d'un contrôleur de type [**@RestController**] rendent des objets qui sont sérialisés pour être envoyés au navigateur. Le type de sérialisation opérée dépend de la configuration de Spring MVC. Ici, ils seront sérialisés en jSON. C'est la présence d'une bibliothèque jSON dans les dépendances du projet qui fait que Spring Boot va, par autoconfiguration, configurer le projet de cette façon ;
- ligne 14 : l'annotation [**@RequestMapping**] indique l'URL que traite la méthode, ici l'URL [/greeting] ;
- ligne 15 : nous avons déjà expliqué l'annotation [@RequestParam]. Le résultat rendu par la méthode est un objet de type [Greeting].
- ligne 12 : un entier long de type atomique. Cela signifie qu'il supporte la concurrence d'accès. Plusieurs threads peuvent vouloir incrémenter la variable [counter] en même temps. Cela se fera proprement. Un thread ne peut lire la valeur du compteur que si le thread en train de le modifier a terminé sa modification.

1.7.5 Le modèle M

Le modèle M produit par la méthode précédente est l'objet [Greeting] suivant :




```
1.  package hello;
2.
3.  public class Greeting {
4.
5.      private final long id;
```

```
6.      private final String content;
7.
8.      public Greeting(long id, String content) {
9.          this.id = id;
10.         this.content = content;
11.     }
12.
13.     public long getId() {
14.         return id;
15.     }
16.
17.     public String getContent() {
18.         return content;
19.     }
20. }
```

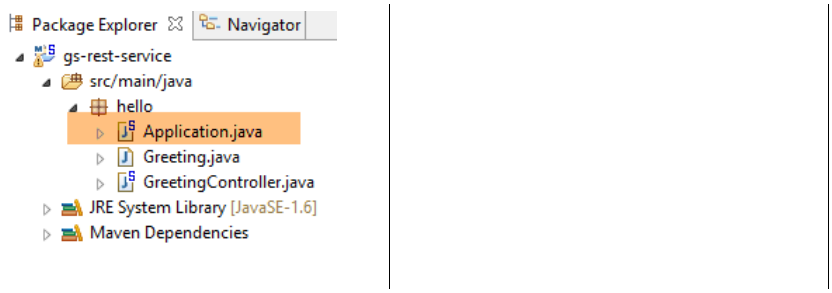
La transformation jSON de cet objet créera la chaîne de caractères **{"id":n,"content":"texte"}**. Au final, la chaîne jSON produite par la méthode du contrôleur sera de la forme :

{"id":2,"content":"Hello, World!"}

ou

{"id":2,"content":"Hello, John!"}

1.7.6 Exécution



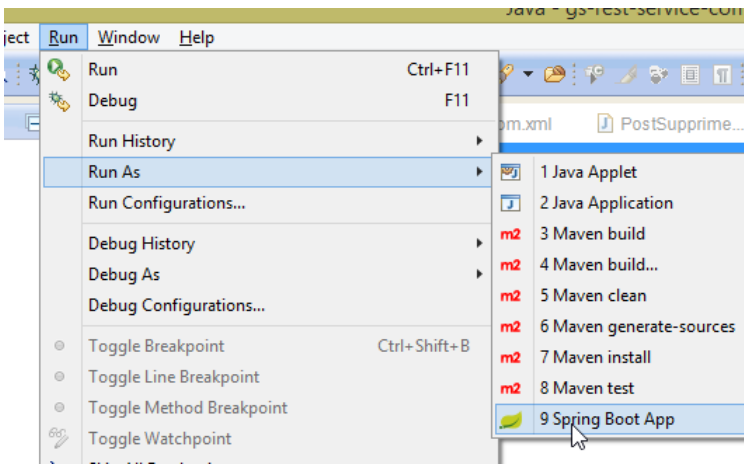
La classe [Application.java] est la classe exécutable du projet. Son code est le suivant :

```
16. package hello;
17.
18. import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
19. import org.springframework.boot.SpringApplication;
20. import org.springframework.context.annotation.ComponentScan;
21.
22. @ComponentScan
23. @EnableAutoConfiguration
24. public class Application {
25.
26.     public static void main(String[] args) {
27.         SpringApplication.run(Application.class, args);
28.     }
29.
30. }
```

Nous avons déjà rencontré et expliqué ce code dans l'exemple précédent.

1.7.7 Exécution du projet

Exécutons le projet :



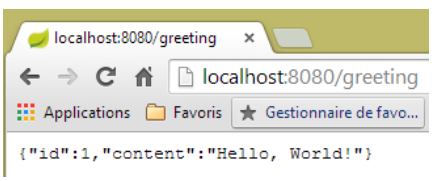
On obtient les logs console suivants :

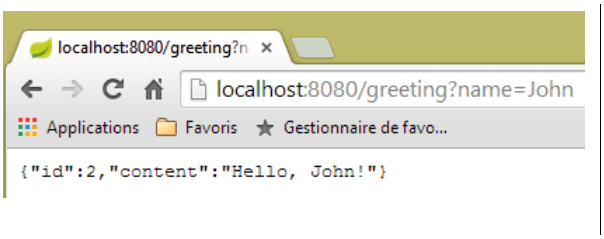
```
1.    .   ____          _            __ _ _
2.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
3.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
4.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
5.    '  |____| .__|_| |_|_| |_\__, | / / / /
6.   =========|_|==============|___/=/_/_/_/
7.   :: Spring Boot ::        (v1.1.9.RELEASE)
8.
9.  2014-11-28 15:22:55.005  INFO 3152 --- [           main] hello.Application                        :
    Starting Application on Gportpers3 with PID 3152 (started by ST in D:\data\istia-1415\spring mvc\dvp-
    final\gs-rest-service)
10. 2014-11-28 15:22:55.046  INFO 3152 --- [           main] ationConfigEmbeddedWebApplicationContext :
    Refreshing
    org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@62e136d3: startup
    date [Fri Nov 28 15:22:55 CET 2014]; root of context hierarchy
11. 2014-11-28 15:22:55.762  INFO 3152 --- [           main] o.s.b.f.s.DefaultListableBeanFactory     :
    Overriding bean definition for bean 'beanNameViewResolver': replacing [Root bean: class [null]; scope=;
    abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true; primary=false;
    factoryBeanName=org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewCo
    nfiguration; factoryMethodName=beanNameViewResolver; initMethodName=null; destroyMethodName=(inferred);
    defined in class path resource
    [org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration.cla
    ss]] with [Root bean: class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3;
    dependencyCheck=0; autowireCandidate=true; primary=false;
    factoryBeanName=org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfiguration
    Adapter; factoryMethodName=beanNameViewResolver; initMethodName=null; destroyMethodName=(inferred);
    defined in class path resource
    [org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter.class]]
12. 2014-11-28 15:22:56.567  INFO 3152 --- [           main] .t.TomcatEmbeddedServletContainerFactory : Server
    initialized with port: 8080
13. 2014-11-28 15:22:56.738  INFO 3152 --- [           main] o.apache.catalina.core.StandardService   :
    Starting service Tomcat
14. 2014-11-28 15:22:56.740  INFO 3152 --- [           main] org.apache.catalina.core.StandardEngine  :
    Starting Servlet Engine: Apache Tomcat/7.0.56
15. 2014-11-28 15:22:56.869  INFO 3152 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
    Initializing Spring embedded WebApplicationContext
16. 2014-11-28 15:22:56.870  INFO 3152 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root
    WebApplicationContext: initialization completed in 1827 ms
17. 2014-11-28 15:22:57.478  INFO 3152 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
    Mapping servlet: 'dispatcherServlet' to [/]
18. 2014-11-28 15:22:57.481  INFO 3152 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
    Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
19. 2014-11-28 15:22:57.685  INFO 3152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped
    URL path [/**/favicon.ico] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
20. 2014-11-28 15:22:57.879  INFO 3152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
    "{[/greeting],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    hello.Greeting hello.GreetingController.greeting(java.lang.String)
21. 2014-11-28 15:22:57.884  INFO 3152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
    "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
```

```
    org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletReques
    t)
22. 2014-11-28 15:22:57.885  INFO 3152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
    "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[text/html],custom=[]}" onto public
    org.springframework.web.servlet.ModelAndView
    org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRe
    quest)
23. 2014-11-28 15:22:57.906  INFO 3152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped
    URL path [/webjars/**] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
24. 2014-11-28 15:22:57.907  INFO 3152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped
    URL path [/**] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
25. 2014-11-28 15:22:58.231  INFO 3152 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
    Registering beans for JMX exposure on startup
26. 2014-11-28 15:22:58.318  INFO 3152 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat
    started on port(s): 8080/http
27. 2014-11-28 15:22:58.319  INFO 3152 --- [           main] hello.Application                        :
    Started Application in 3.788 seconds (JVM running for 4.424)
```

- ligne 13 : le serveur Tomcat démarre sur le port 8080 (ligne 12) ;
- ligne 17 : la servlet [DispatcherServlet] est présente ;
- ligne 20 : la méthode [GreetingController.greeting] a été découverte ;

Pour tester l'application web, on demande l'URL [http://localhost:8080/greeting] :

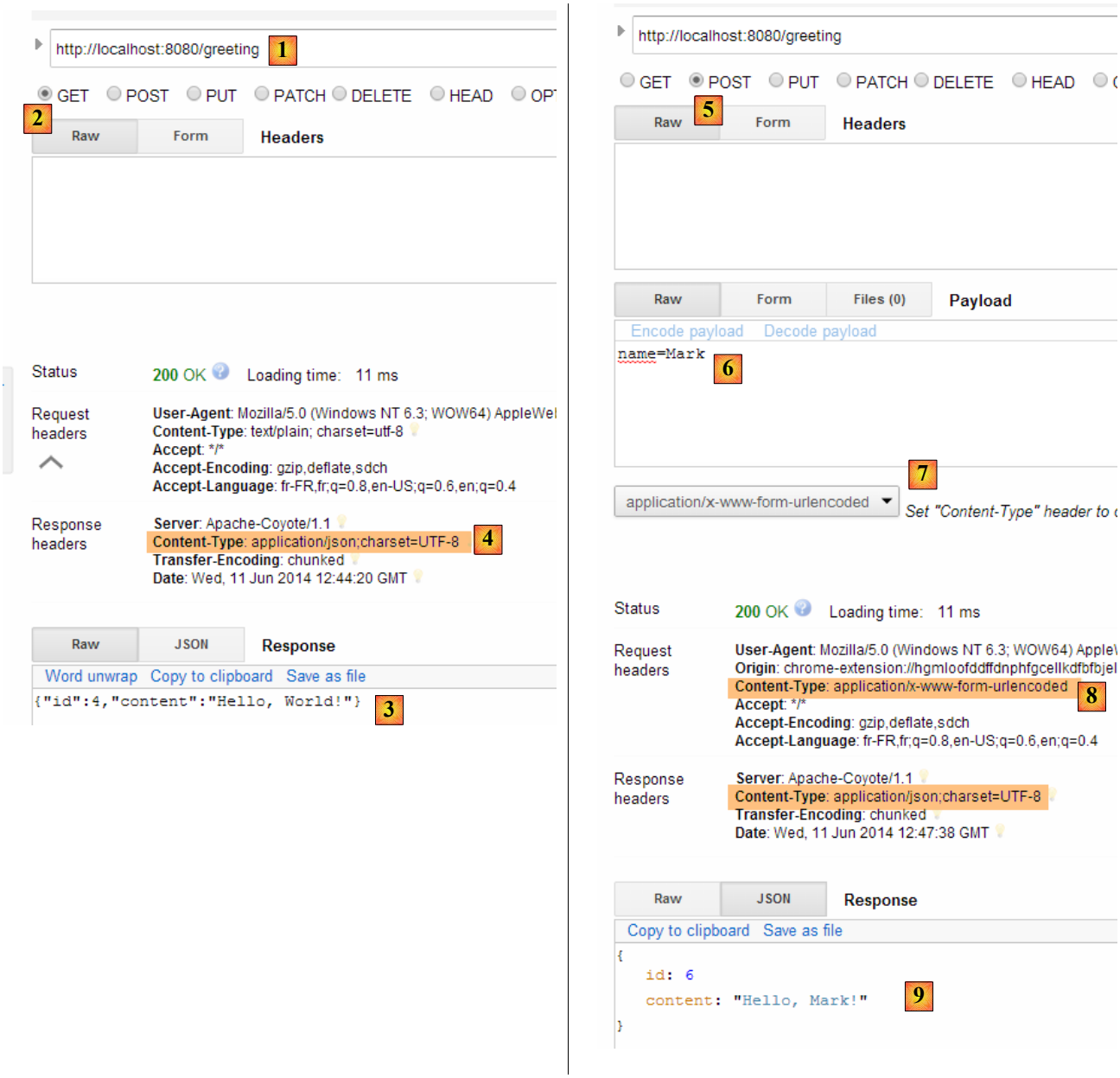




On reçoit bien la chaîne jSON attendue.

Note : cet exemple n'a pas fonctionné avec le navigateur intégré d'Eclipse.

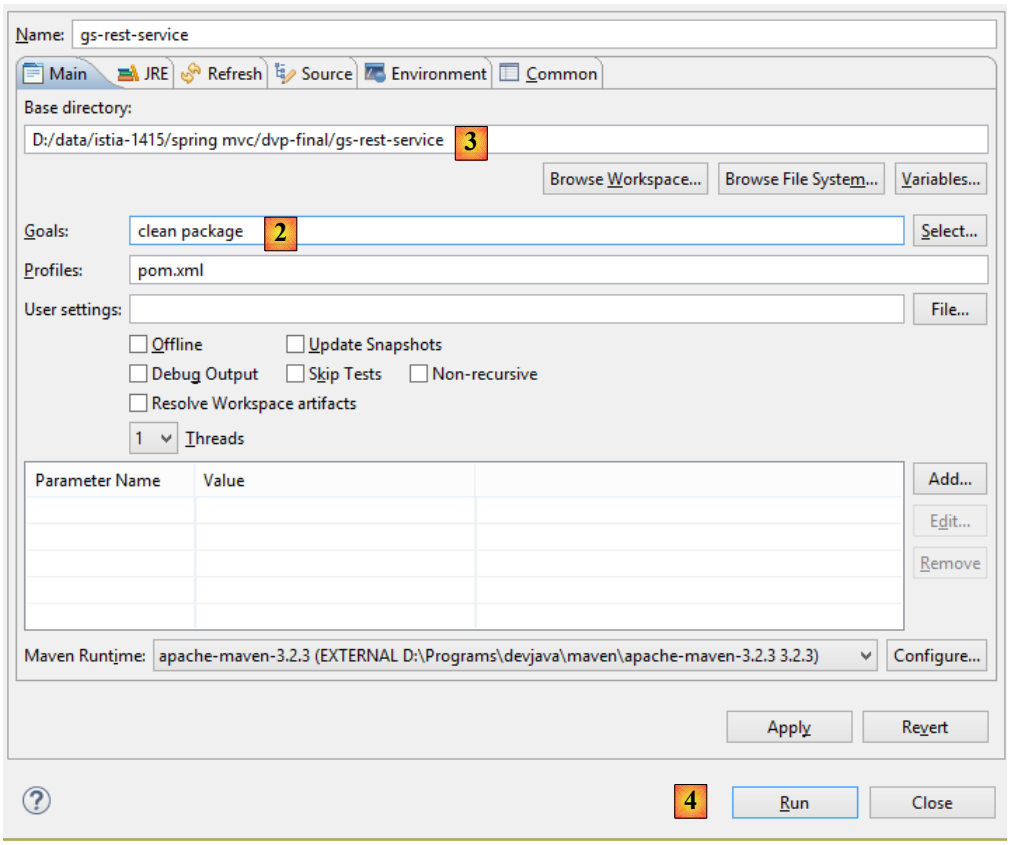
Il peut être intéressant de voir les entêtes HTTP envoyés par le serveur. Pour cela, on va utiliser le plugin de Chrome appelé [Advanced Rest Client] (cf Annexes) :



- en [1], l'URL demandée ;
- en [2], la méthode GET est utilisée ;
- en [3], la réponse jSON ;
- en [4], le serveur a indiqué qu'il envoyait une réponse au format jSON ;
- en [5], on demande la même URL mais cette fois-ci avec un POST ;
- en [7], les informations sont envoyées au serveur sous la forme [urlencoded] ;
- en [6], le paramètre **name** avec sa valeur ;
- en [8], le navigateur indique au serveur qu'il lui envoie des informations [urlencoded] ;
- en [9], la réponse jSON du serveur ;

1.7.8 Création d'une archive exécutable

Comme nous l'avons fait pour le projet précédent, nous créons une archive exécutable :



- en [1] : on exécute une cible Maven ;
- en [2] : il y a deux cibles (goals) : [clean] pour supprimer le dossier [target] du projet Maven, [package] pour le régénérer ;
- en [3] : le dossier [target] généré, le sera dans ce dossier ;
- en [4] : on génère la cible ;

Dans les logs qui apparaissent dans la console, il est important de voir apparaître le plugin [`spring-boot-maven-plugin`]. C'est lui qui génère l'archive exécutable.

```
[INFO] --- spring-boot-maven-plugin:1.1.0.RELEASE:repackage (default) @ gs-rest-service ---
```

Avec une console, on se place dans le dossier généré :

```
1.  D:\Temp\wksSTS\gs-rest-service\target>dir
2.   ...
3.  11/06/2014  15:30    <DIR>          classes
4.  11/06/2014  15:30    <DIR>          generated-sources
5.  11/06/2014  15:30        11 073 572 gs-rest-service-0.1.0.jar
6.  11/06/2014  15:30             3 690 gs-rest-service-0.1.0.jar.original
7.  11/06/2014  15:30    <DIR>          maven-archiver
8.  11/06/2014  15:30    <DIR>          maven-status
9.  ...
```

- ligne 5 : l'archive générée ;

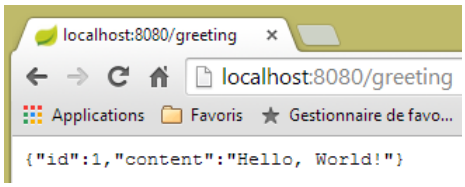
Cette archive est exécutée de la façon suivante :

```
1.  D:\Temp\wksSTS\gs-rest-service-complete\target>java -jar gs-rest-service-0.1.0.jar
2.
3.    .   ____          _            __ _ _
4.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
5.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
6.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
7.    '  |____| .__|_| |_|_| |_\__, | / / / /
8.   =========|_|==============|___/=/_/_/_/
```

```
 :: Spring Boot ::        (v1.1.0.RELEASE)

2014-06-11 15:32:47.088  INFO 4972 --- [           main] hello.Application
                  : Starting Application on Gportpers3 with PID 4972 (D:\Temp\wk
sSTS\gs-rest-service-complete\target\gs-rest-service-0.1.0.jar started by ST in
D:\Temp\wksSTS\gs-rest-service-complete\target)
...
```

Note : il faut auparavant arrêter le service web éventuellement lancé dans Eclipse (cf page 23).

Maintenant que l'application web est lancée, on peut l'interroger avec un navigateur :



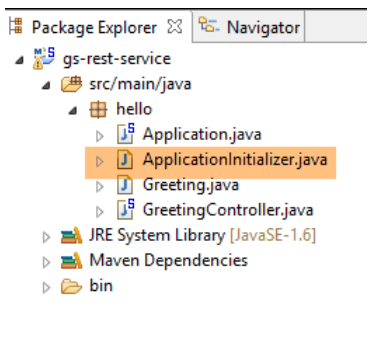
1.7.9 Déployer l'application sur un serveur Tomcat

Comme il a été fait pour le projet précédent, nous modifions le fichier [pom.xml] de la façon suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>org.springframework</groupId>
7.     <artifactId>gs-rest-service</artifactId>
8.     <version>0.1.0</version>
9.     <packaging>war</packaging>
10.
11.    ...
12. </project>
```

- ligne 9 : il faut indiquer qu'on va générer une archive war (Web ARchive) ;

Il faut par ailleurs configurer l'application web. En l'absence de fichier [web.xml], cela se fait avec une classe héritant de [`SpringBootServletInitializer`] :



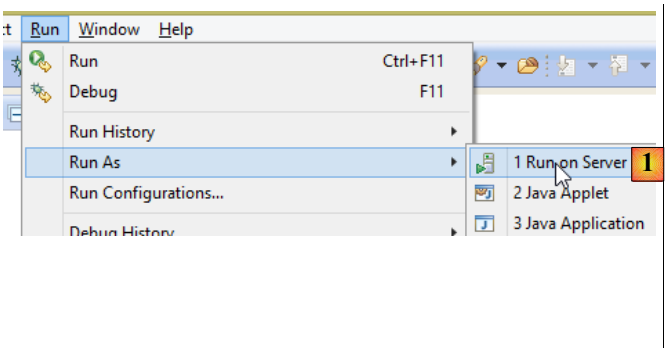
La classe [ApplicationInitializer] est la suivante :

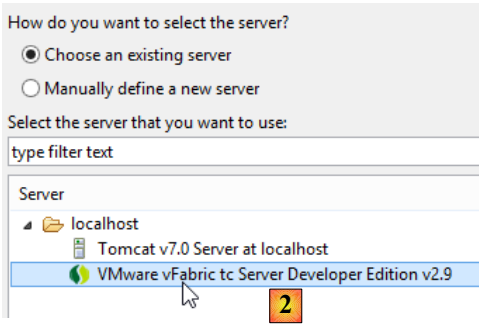
```
1.  package hello;
2.
3.  import org.springframework.boot.builder.SpringApplicationBuilder;
4.  import org.springframework.boot.context.web.SpringBootServletInitializer;
5.
6.  public class ApplicationInitializer extends SpringBootServletInitializer {
7.
```

```
8.      @Override
9.      protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10.         return application.sources(Application.class);
11.     }
12.
13. }
```

- ligne 6 : la classe [ApplicationInitializer] étend la classe [SpringBootServletInitializer] ;
- ligne 9 : la méthode [configure] est redéfinie (ligne 8) ;
- ligne 10 : on fournit la classe qui configure le projet ;

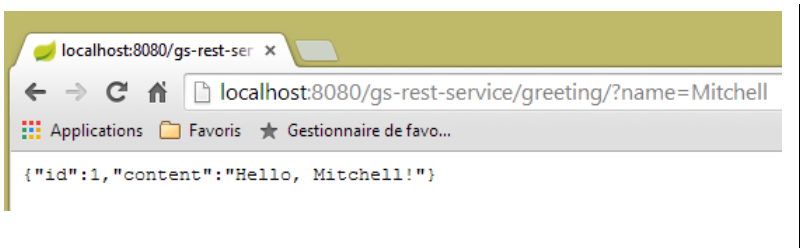
Pour exécuter le projet, on peut procéder ainsi :





- en [1-2], on exécute le projet sur l'un des serveurs enregistrés dans l'IDE Eclipse ;

Ceci fait, on peut demander l'URL [http://localhost:8080/gs-rest-service/greeting/?name=Mitchell] dans un navigateur :



1.8 Conclusion

Nous avons introduit deux types de projets Spring MVC :

- un projet où l'application web envoie un flux HTML au navigateur. Ce flux est généré par le moteur de vues [Thymeleaf] ;
- un projet où l'application web envoie un flux jSON au navigateur ;

Dans le premier cas, deux dépendances Maven sont nécessaires au projet :

```
1.      <parent>
2.         <groupId>org.springframework.boot</groupId>
3.         <artifactId>spring-boot-starter-parent</artifactId>
4.         <version>1.1.9.RELEASE</version>
5.      </parent>
6.
7.      <dependencies>
8.         <dependency>
9.            <groupId>org.springframework.boot</groupId>
10.           <artifactId>spring-boot-starter-thymeleaf</artifactId>
11.        </dependency>
12. </dependencies>
```

Dans le second cas, les dépendances Maven sont les suivantes :

```
1.      <parent>
```

```
2.          <groupId>org.springframework.boot</groupId>
3.          <artifactId>spring-boot-starter-parent</artifactId>
4.          <version>1.1.9.RELEASE</version>
5.      </parent>
6.
7.      <dependencies>
8.          <dependency>
9.              <groupId>org.springframework.boot</groupId>
10.             <artifactId>spring-boot-starter-web</artifactId>
11.         </dependency>
12. </dependencies>
```

Les dépendances amenées en cascade par ces configurations sont très nombreuses et beaucoup sont inutiles. Pour la mise en exploitation de l'application, on utilisera une configuration Maven manuelle où seront présentes les seules dépendances nécessaires au projet.

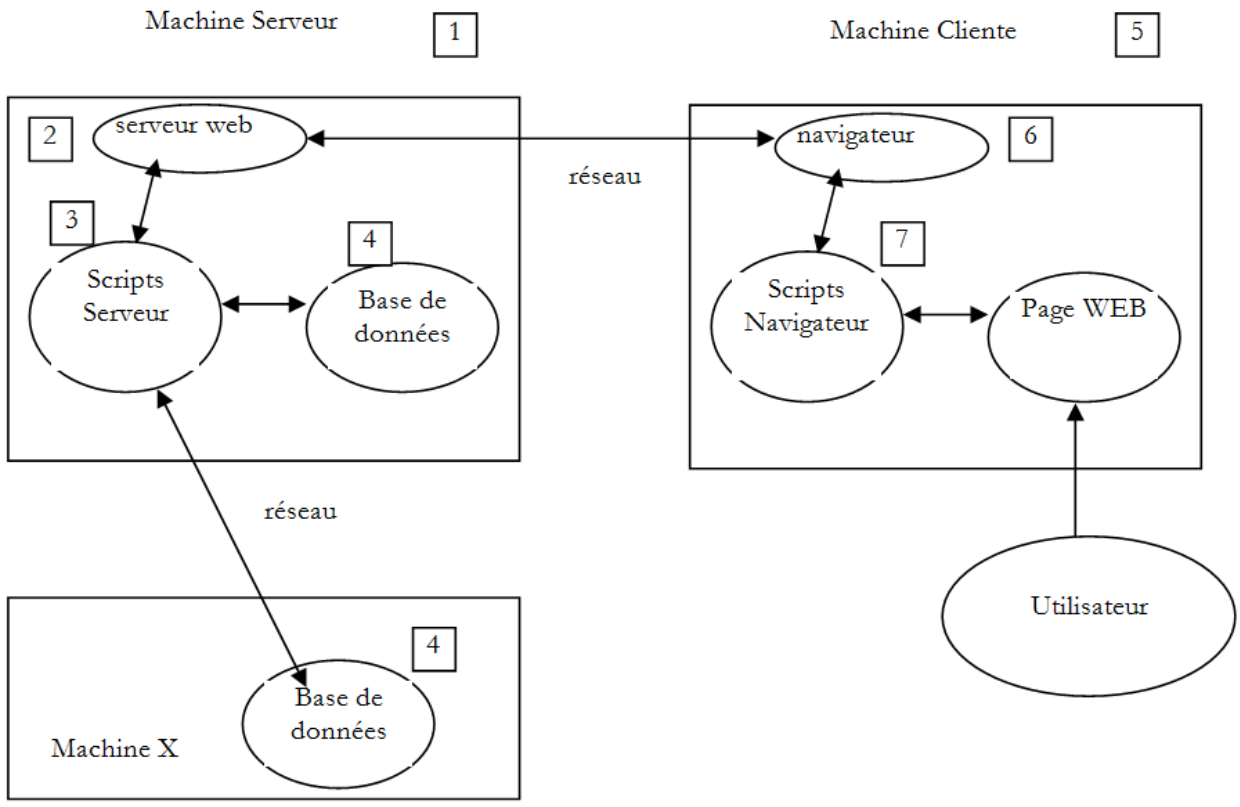
Nous allons maintenant revenir aux bases de la programmation web en présentant deux notions de base :

- le dialogue HTTP (HyperText Transfer Protocol) entre un navigateur et une application web ;
- le langage HTML (HyperText Markup Language) que le navigateur interprète pour afficher une page qu'il a reçue ;

2 Les bases de la programmation Web

Ce chapitre a pour but essentiel de faire découvrir les grands principes de la programmation Web qui sont indépendants de la technologie particulière utilisée pour les mettre en oeuvre. Il présente de nombreux exemples qu'il est conseillé de tester afin de "s'imprégner" peu à peu de la philosophie du développement Web. Le lecteur ayant déjà ces connaissances peut passer directement au chapitre suivant **page 78**.

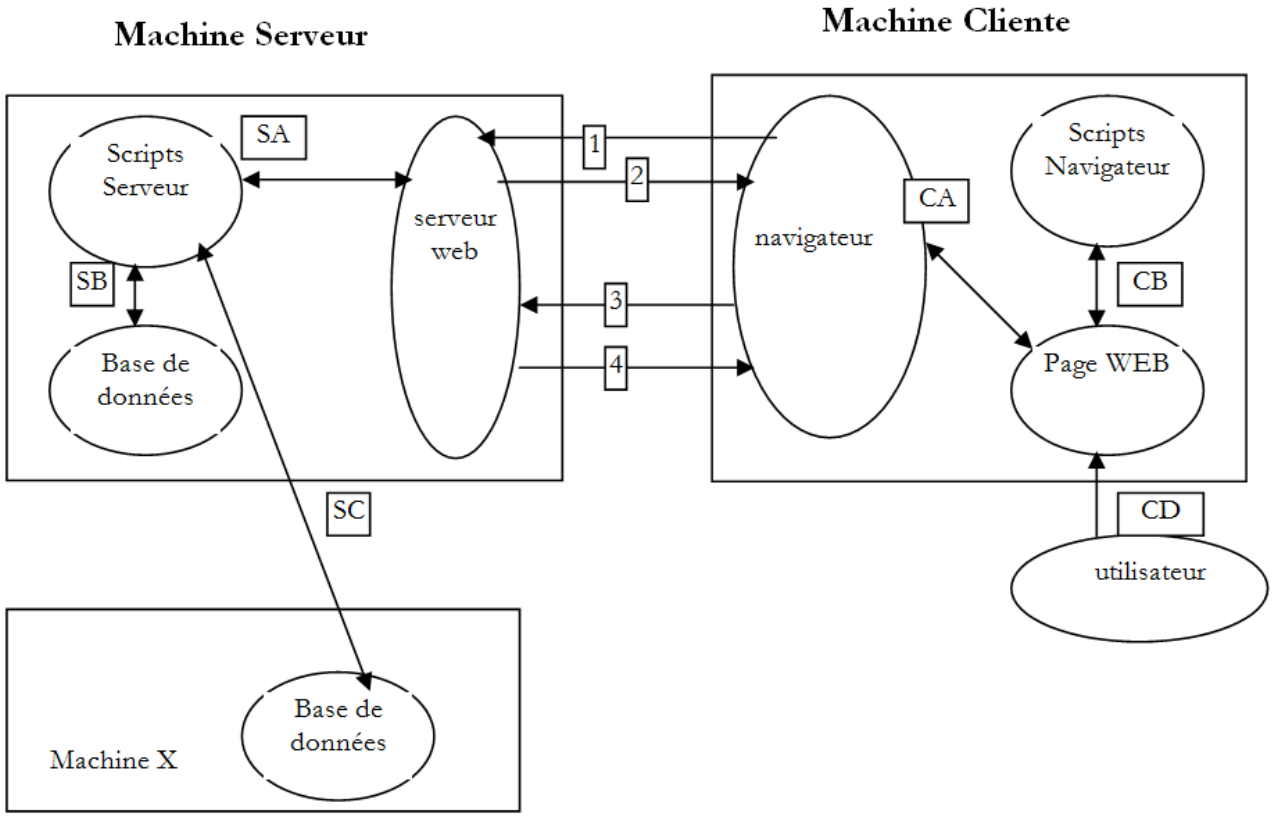
Les composantes d'une application Web sont les suivantes :



Numéro	Rôle	Exemples courants
1	OS Serveur	Unix, Linux, Windows
2	Serveur Web	Apache (Unix, Linux, Windows) IIS (Windows+plate-forme .NET) Node.js (Unix, Linux, Windows)
3	Codes exécutés côté serveur. Ils peuvent l'être par des modules du serveur ou par des programmes externes au serveur (CGI).	JAVASCRIPT (Node.js) PHP (Apache, IIS) JAVA (Tomcat, Websphere, JBoss, Weblogic, ...) C#, VB.NET (IIS)
4	Base de données - Celle-ci peut être sur la même machine que le programme qui l'exploite ou sur une autre via Internet.	Oracle (Linux, Windows) MySQL (Linux, Windows) Postgres (Linux, Windows) SQL Server (Windows)
5	OS Client	Unix, Linux, Windows
6	Navigateur Web	Chrome, Internet Explorer, Firefox, Opera, Safari, ...

7	Scripts exécutés côté client au sein du navigateur. **Ces scripts n'ont aucun accès aux disques du poste client**.	Javascript (tout navigateur)

2.1 Les échanges de données dans une application Web avec formulaire



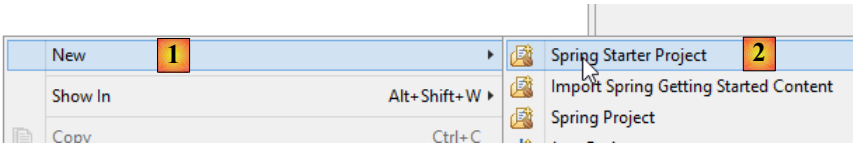
Numéro	Rôle
1	Le navigateur demande une URL pour la 1ère fois *(http://machine/url)*. Aucun paramètre n'est passé.
2	Le serveur Web lui envoie la page Web de cette URL. Elle peut être statique ou bien dynamiquement générée par un script serveur (SA) qui a pu utiliser le contenu de bases de données (SB, SC). Ici, le script détectera que l'URL a été demandée sans passage de paramètres et génèrera la page Web initiale. Le navigateur reçoit la page et l'affiche (CA). Des scripts côté navigateur (CB) ont pu modifier la page initiale envoyée par le serveur. Ensuite par des interactions entre l'utilisateur (CD) et les scripts (CB) la page Web va être modifiée. Les formulaires vont notamment être remplis.
3	L'utilisateur valide les données du formulaire qui doivent alors être envoyées au serveur Web. Le navigateur redemande l'URL initiale ou une autre selon les cas et transmet en même temps au serveur les valeurs du formulaire. Il peut utiliser pour ce faire deux méthodes appelées GET et POST. A réception de la demande du client, le serveur déclenche le script (SA) associé à l'URL demandée, script qui va détecter les paramètres et les traiter.
4	Le serveur délivre la page Web construite par programme (SA, SB, SC). Cette étape est identique à l'étape 2 précédente. Les échanges se font désormais selon les étapes 2 et 3.

2.2 Pages Web statiques, Pages Web dynamiques

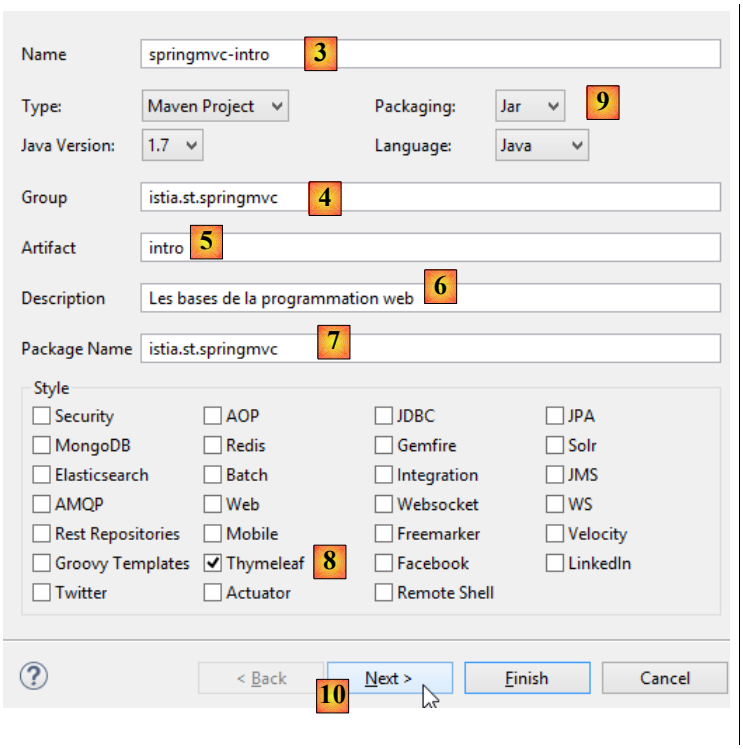
Une page statique est représentée par un fichier HTML. Une page dynamique est une page HTML générée "à la volée" par le serveur Web.

2.2.1 Page statique HTML (HyperText Markup Language)

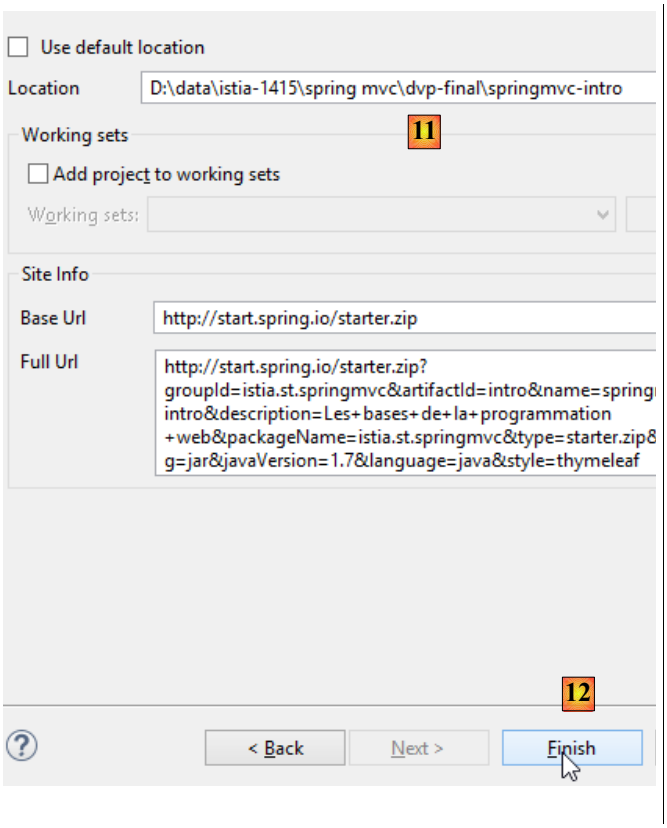
Construisons un premier projet Spring MVC [1-2] :

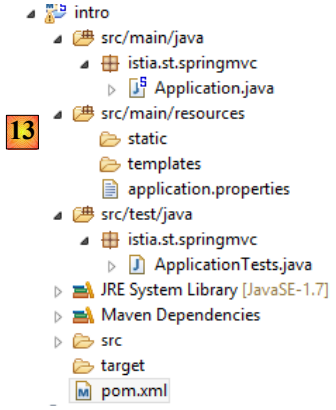


- en [1-2], nous créons un nouveau projet basé sur Spring Boot [http://projects.spring.io/spring-boot/] ;



- les informations [3-7] sont pour la configuration Maven du projet ;
- en [3], le nom du projet Maven ;
- en [4], le groupe Maven dans lequel sera placé le résultat de la compilation du projet ;
- en [5], le nom donné au produit de la compilation ;
- en [6], une description du projet ;
- en [7], le package dans lequel sera placée la classe exécutable du projet ;
- en [8], la nature du projet. C'est un projet web avec des vues Thymeleaf. On voit ici, toutes les dépendances Maven prêtes à l'emploi offertes par le projet Spring Boot ;
- en [9], on indique que le produit issu du build Maven sera packagé dans une archive **jar** et non **war**. Le projet va alors utiliser un serveur Tomcat embarqué qui se trouvera dans ses dépendances ;
- en [10], on passe à la suite de l'assistant ;





- en [11], on indique le dossier du projet ;
- en [12], on termine l'assistant ;
- en [13], le projet généré.

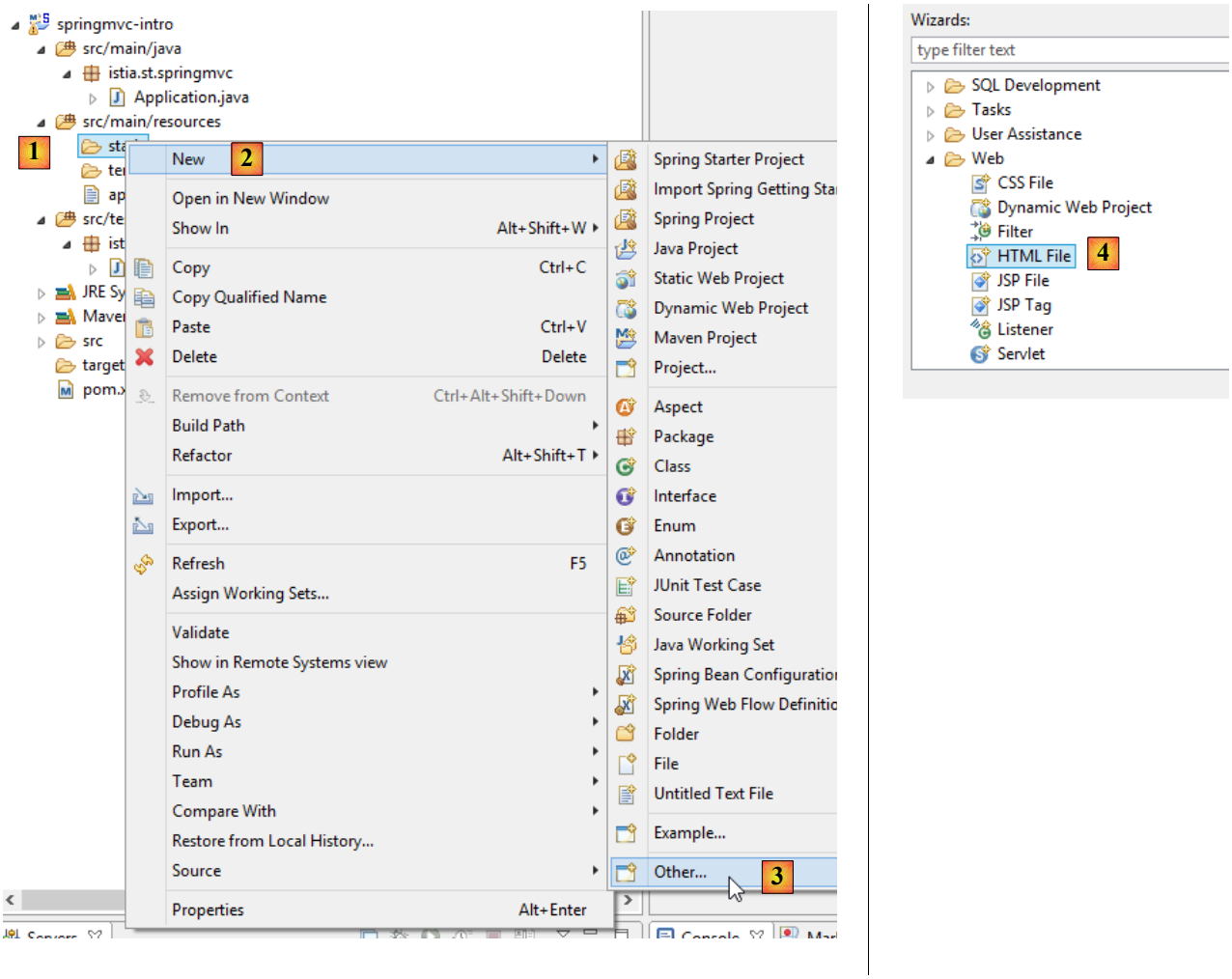
Examinons le fichier [pom.xml] généré :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>istia.st.springmvc</groupId>
7.     <artifactId>intro</artifactId>
8.     <version>0.0.1-SNAPSHOT</version>
9.     <packaging>jar</packaging>
10.
11.    <name>springmvc-intro</name>
12.    <description>Les bases de la programmation web</description>
13.
14.    <parent>
15.       <groupId>org.springframework.boot</groupId>
16.       <artifactId>spring-boot-starter-parent</artifactId>
17.       <version>1.1.9.RELEASE</version>
18.       <relativePath /> <!-- lookup parent from repository -->
19.    </parent>
20.
21.    <dependencies>
22.       <dependency>
23.          <groupId>org.springframework.boot</groupId>
24.          <artifactId>spring-boot-starter-web</artifactId>
25.       </dependency>
26.       <dependency>
27.          <groupId>org.springframework.boot</groupId>
28.          <artifactId>spring-boot-starter-test</artifactId>
29.          <scope>test</scope>
30.       </dependency>
31.    </dependencies>
32.
33.    <properties>
```

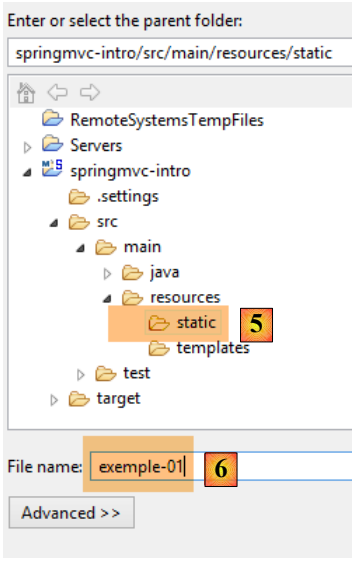
```
34.         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
35.         <start-class>istia.st.springmvc.Application</start-class>
36.         <java.version>1.7</java.version>
37.     </properties>
38.
39.     <build>
40.         <plugins>
41.             <plugin>
42.                 <groupId>org.springframework.boot</groupId>
43.                 <artifactId>spring-boot-maven-plugin</artifactId>
44.             </plugin>
45.         </plugins>
46.     </build>
47.
48. </project>
```

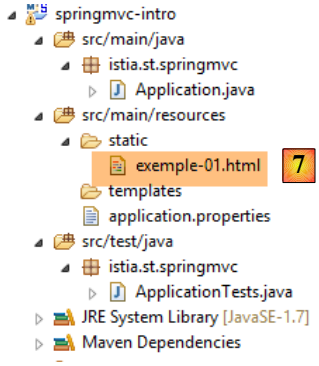
Il reprend toutes informations données dans l'assistant. Lignes 26-30, nous trouvons une dépendance que nous ne connaissions pas. Elle permet l'intégration des tests unitaires JUnit avec Spring.

Commençons par créer une page HTML statique dans ce projet. Elle doit être placée par défaut dans le dossier [src / main / resources / static] :



- en [1-4], on crée un fichier HTML dans le dossier [static] ;





- en [6], donner un nom à la page ;
- en [7], la page a été ajoutée.

Le contenu de la page créée est le suivant :

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.  <meta charset="ISO-8859-1">
5.  <title>Insert title here</title>
6.  </head>
7.  <body>
8.
9.  </body>
10. </html>
```

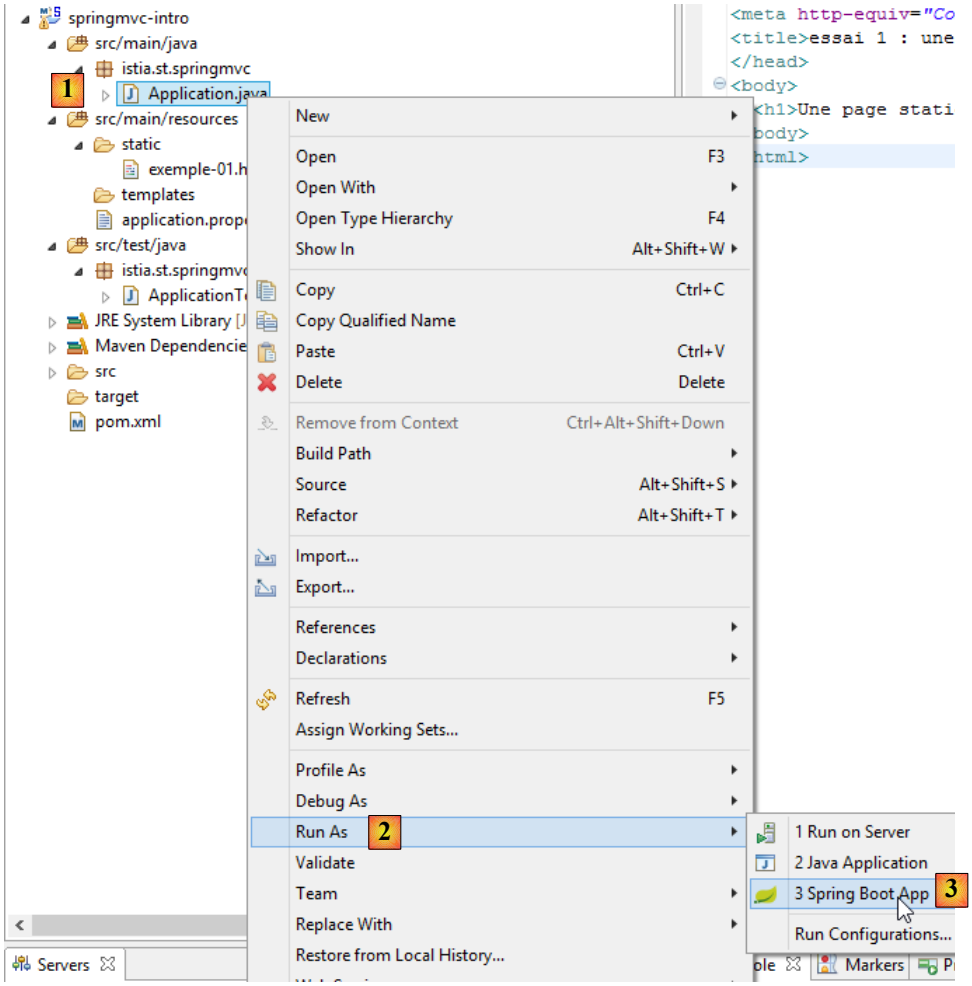
- lignes 2-10 : le code est délimité par la balise racine <**html**> ;
- lignes 3-6 : la balise <**head**> délimite ce qu'on appelle l'entête de la page ;
- lignes 7-9 : la balise <**body**> délimite ce qu'on appelle le corps de la page.

Modifions ce code de la façon suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>essai 1 : une page statique</title>
6.  </head>
7.  <body>
8.    <h1>Une page statique...</h1>
9.  </body>
10. </html>
```

- ligne 5 : définit le titre de la page – sera affiché comme titre de la fenêtre du navigateur affichant la page ;
- ligne 8 : un texte en gros caractères (<**h1**>).

Exécutons l'application [1-3] :



puis avec un navigateur, demandons l'URL [http://localhost:8080/exemple-01.html] :

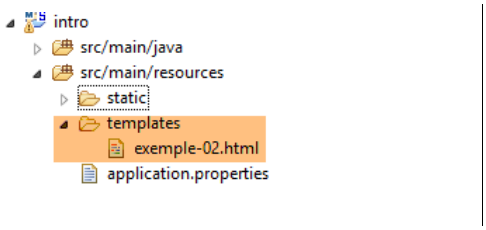
- en [1], l'URL de la page visualisée ;
- en [2], le titre de la fenêtre – a été fourni par la balise <**title**> de la page ;
- en [3], le corps de la page - a été fourni par la balise <**h1**>.

Regardons [4-5] le code HTML reçu par le navigateur :

- en [5], le navigateur a reçu la page HTML que nous avions construite. Il l'a interprétée et en a fait un affichage graphique.

2.2.2 Une page Thymeleaf dynamique

Créons maintenant une page Thymeleaf. C'est une page HTML classique avec des balises enrichies d'attributs [Thymeleaf] [http://www.thymeleaf.org/]. On suit une démarche analogue à celui de la création de la page HTML mais cette fois-ci c'est dans le dossier [templates] qu'il faut placer la nouvelle page HTML :

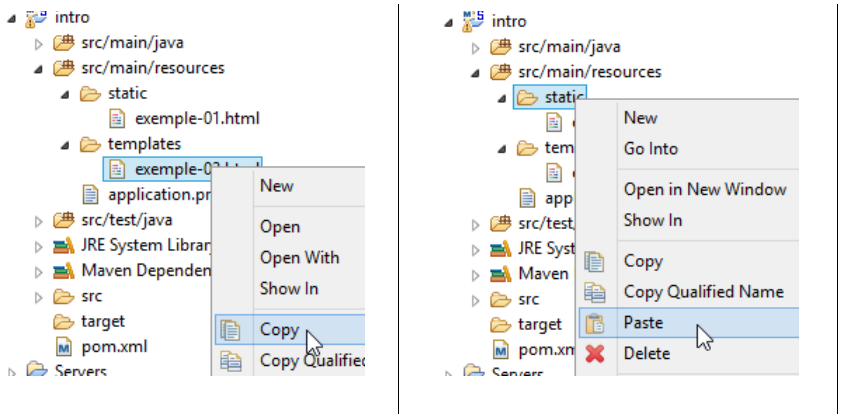


La page [exemple-02.html] sera la suivante :

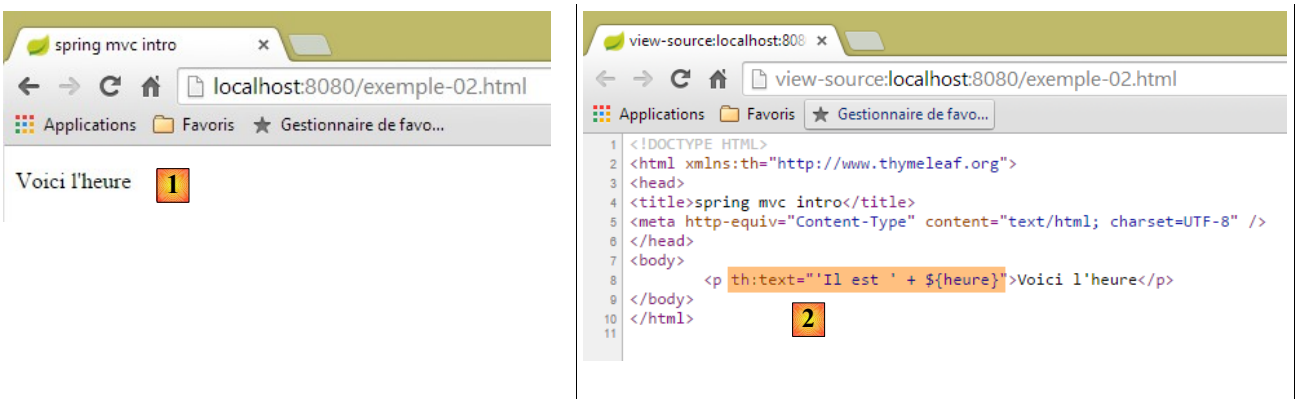
```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title>spring mvc intro</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.      <p th:text="'Il est ' + ${heure}">Voici l'heure</p>
9.  </body>
10. </html>
```

- ligne 8 : la balise <p> est une balise HTML qui introduit un paragraphe dans la page affichée. [th:text] est un attribut [Thymeleaf] qui a deux destinées différentes selon que [Thymeleaf] est à l'oeuvre ou non :
  - si [Thymeleaf] n'interprète pas la page HTML, l'attribut [th:text] sera ignoré car inconnu en HTML. Le texte affiché sera alors [Voici l'heure],
  - si [Thymeleaf] interprète la page HTML, l'attribut [th:text] sera évalué et sa valeur remplacera le texte [Voici l'heure]. Sa valeur sera du genre [Il est 17:11:06] ;

Voyons cela à l'oeuvre. Nous dupliquons la page [templates / exemple-02.html] dans le dossier [static]. Les pages HTML placées dans ce dossier ne sont pas interprétées par [Thymeleaf] :

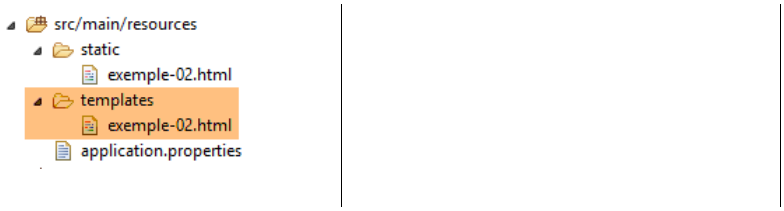


Nous exécutons l'application comme nous l'avons déjà fait plusieurs fois, puis nous demandons avec un navigateur l'URL [http://localhost:8080/exemple-02.html] :



Nous voyons en [1] que l'attribut [th:text] n'a pas été interprété et n'a pas provoqué non plus d'erreur. Le code source de la page reçue en [2] montre que le navigateur a bien reçu la page complète.

Revenons à la page [exemple-02.html] du dossier [templates] :



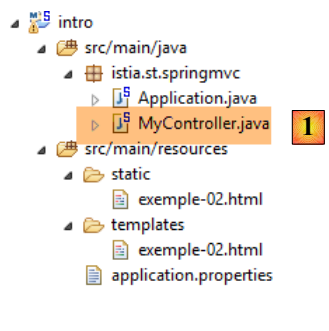
Les pages HTML placées dans le dossier [templates] sont interpétées par [Thymeleaf]. Revenons au code de la page :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title>spring mvc intro</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.      <p th:text="'Il est ' + ${heure}">Voici l'heure</p>
9.  </body>
10. </html>
```

- ligne 7 : [Thymeleaf] va interpréter l'attribut [th:text] et va remplacer [Voici l'heure] par la valeur de l'expression :

```
"'Il est ' + ${heure}"
```

Cette expression utilise la variable [${heure}] où [heure] appartient au modèle de la vue [exemple-02.html]. Il nous faut donc créer ce modèle. Pour cela, nous allons suivre l'exemple étudié au paragraphe 1.6, page 14. Nous faisons évoluer le projet de la façon suivante :

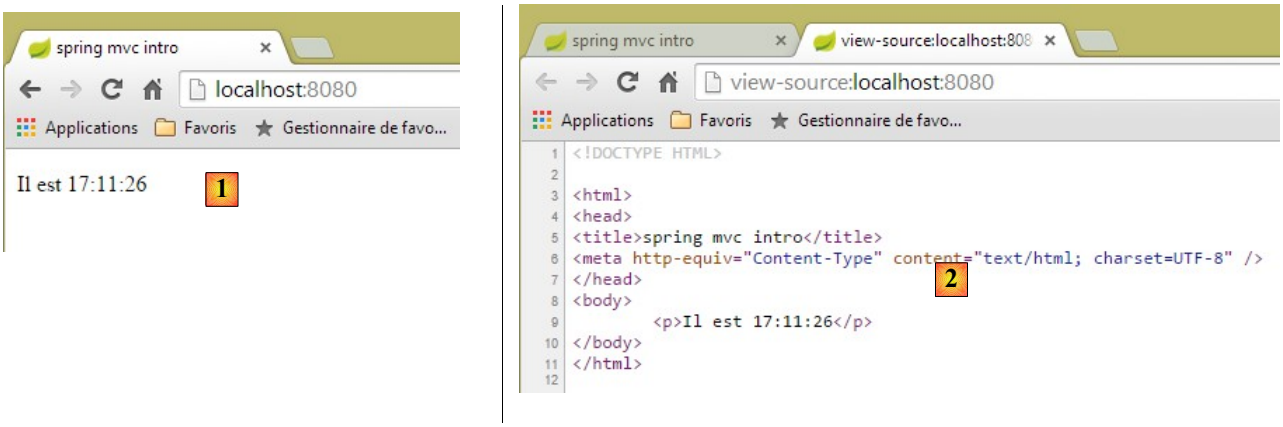


En [1], nous ajoutons le contrôleur suivant :

```
1.  package istia.st.springmvc;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.
6.  import org.springframework.stereotype.Controller;
7.  import org.springframework.ui.Model;
8.  import org.springframework.web.bind.annotation.RequestMapping;
9.
10. @Controller
11. public class MyController {
12.
13.     @RequestMapping("/")
14.     public String heure(Model model) {
15.         // format de l'heure
16.         SimpleDateFormat formater = new SimpleDateFormat("HH:MM:ss");
17.         // l'heure du moment
18.         String heure = formater.format(new Date());
19.         // on met l'heure dans le modèle de la vue
20.         model.addAttribute("heure", heure);
21.         // on fait afficher la vue [exemple-02.html]
22.         return "exemple-02";
23.     }
24.  }
```

- lignes 13-14 : la méthode [heure] traite l'URL [/] ;
- ligne 14 : [Model model] est un modèle vide. L'action [heure] doit y mettre les attributs qu'elle souhaite voir dans le modèle. On sait que la vue [exemple-02.html] attend un attribut nommé [heure] ;
- lignes 19-22 : réalisent ce qu'on vient d'expliquer. La vue [exemple-02.html] va être affichée (ligne 22) avec dans son modèle un attribut nommé [heure] (ligne 20) ;
- ligne 16 : on crée un formateur de date. Le format [HH:MM:ss] utilisé est un format [heures:minutes:secondes] où les heures sont dans l'intervalle [0-24] ;
- ligne 18 : avec ce formateur, on formate la date du jour ;
- ligne 20 : l'heure obtenue est associée à un attribut nommé [heure] ;

Nous lançons l'application et nous demandons l'URL [/] :



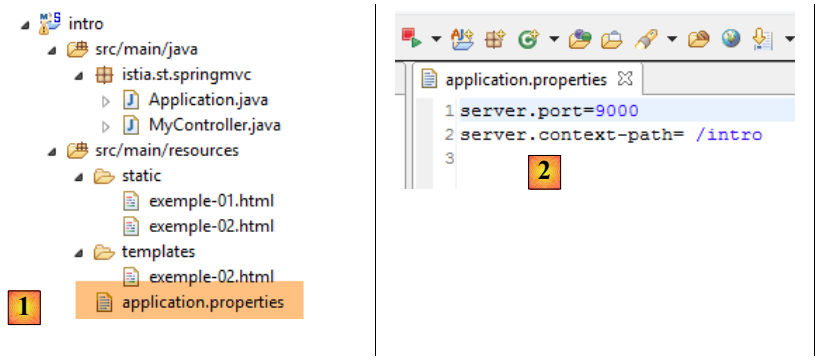
- en [1] la page obtenue et en [2] son contenu HTML. On peut constater que le texte initial [Voici l'heure] a complètement disparu ;

Si maintenant on rafraîchit la page [1] (F5), nous obtenons un autre affichage (nouvelle heure) alors que l'URL ne change pas. C'est l'aspect dynamique de la page : son contenu peut changer au fil du temps.

On retiendra de ce qui précède la nature fondamentalement différente des pages dynamiques et statiques.

2.2.3 Configuration de l'application Spring Boot

Revenons à l'architecture du projet Eclipse :



Le fichier [application.properties] permet de configurer l'application Spring Boot. Pour l'instant ce fichier est vide. On peut l'utiliser pour configurer l'application de multiples façons décrites à l'URL [http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html]. Nous allons utiliser le fichier [application.properties] suivant [2] :

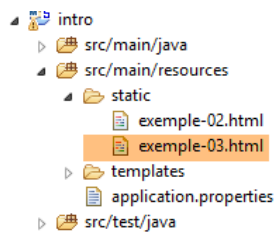
- ligne 1 : fixe le port de service de l'application web ;
- ligne 2 : fixe le contexte de l'application web ;

Avec cette configuration, la page statique [exemple-01.html] sera obtenue avec l'URL [http://localhost:9000/intro/exemple-01.html] :

2.3 Scripts côté navigateur

Une page HTML peut contenir des scripts qui seront exécutés par le navigateur. Le principal langage de script côté navigateur est actuellement (janv 2015) **Javascript**. Des centaines de bibliothèques ont été construites avec ce langage pour faciliter la vie du développeur.

Construisons une nouvelle page [exemple-03.html] dans le dossier [static] du projet existant :

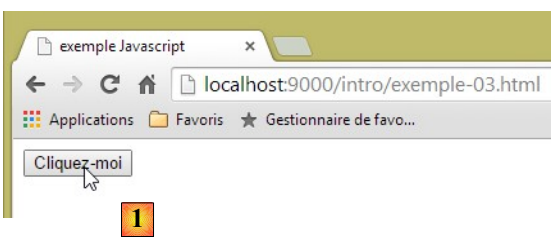


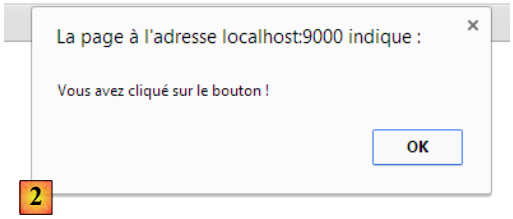
Editons le fichier [exemple-03.html] avec le contenu suivant :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>exemple Javascript</title>
6.    <script type="text/javascript">
7.      function réagir() {
8.        alert("Vous avez cliqué sur le bouton !");
9.      }
10.   </script>
11. </head>
12. <body>
13.   <input type="button" value="Cliquez-moi" onclick="réagir()" />
14. </body>
15. </html>
```

- ligne 13 : définit un bouton (attribut **type**) avec le texte " Cliquez-moi " (attribut **value**). Lorsqu'on clique dessus, la fonction Javascript [réagir] est exécutée (attribut **onclick**) ;
- lignes 6-10 : un script Javascript ;
- lignes 7-9 : la fonction [réagir] ;
- ligne 8 : affiche une boîte de dialogue avec le message [Vous avez cliqué sur le bouton].

Visualisons la page dans un navigateur :

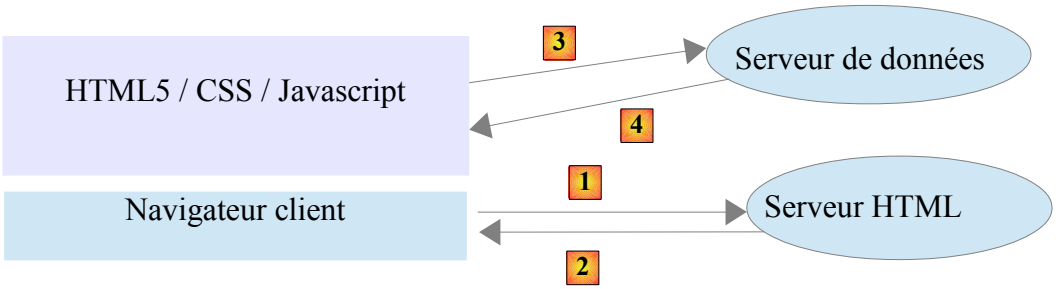




- en [1], la page affichée ;
- en [2], la boîte de dialogue lorsqu'on clique sur le bouton.

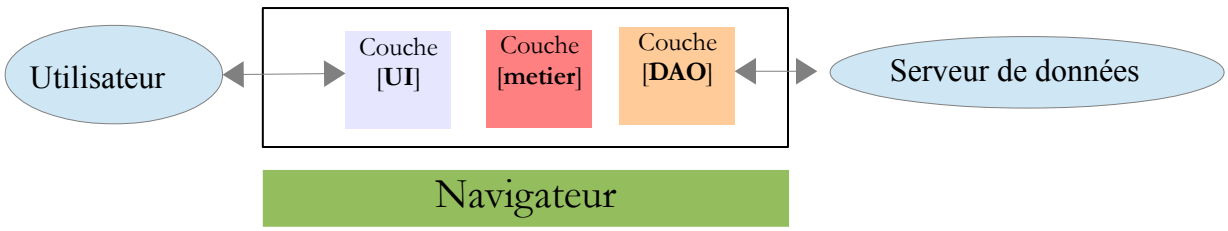
Lorsqu'on clique sur le bouton, il n'y a pas d'échanges avec le serveur. Le code Javascript est exécuté par le navigateur.

Avec les très nombreuses bibliothèques Javascript disponibles, on peut désormais embarquer de véritables applications sur le navigateur. On tend alors vers les architectures suivantes :



- 1-2 : le serveur HTML est un serveur de pages statiques HTML5 / CSS / Javascript ;
- 3-4 : les pages HTML5 / CSS / Javascript délivrées interagissent directement avec le serveur de données. Celui-ci délivre uniquement des données sans habillage HTML. C'est le Javascript qui les insère dans des pages HTML déjà présentes sur le navigateur.

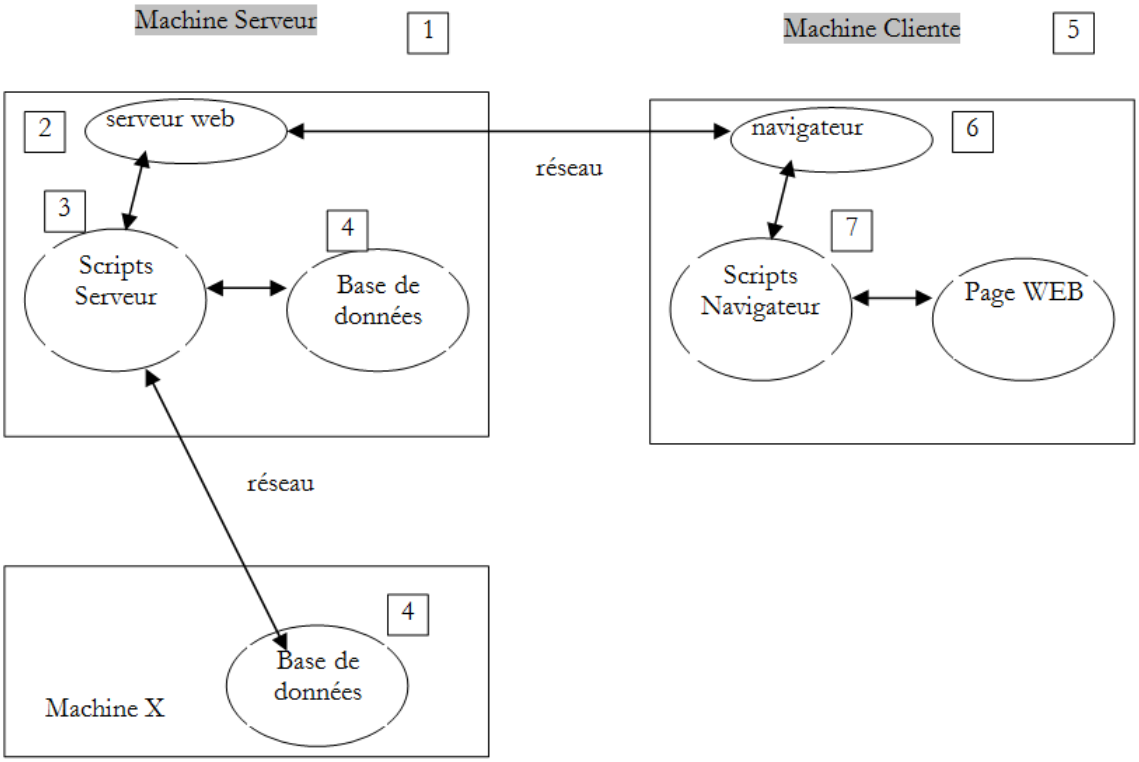
Dans cette architecture, le code Javascript peut devenir lourd. On cherche alors à le structurer en couches comme on le fait pour le code côté serveur :



- le couche [UI] est celle qui interagit avec l'utilisateur ;
- la couche [DAO] interagit avec le serveur de données ;
- la couche [métier] rassemble les procédures métier qui n'interagissent ni avec l'utilisateur, ni avec le serveur de données. Cette couche peut ne pas exister.

2.4 Les échanges client-serveur

Revenons à notre schéma de départ qui illustrait les acteurs d'une application Web :



Nous nous intéressons ici aux échanges entre la machine cliente et la machine serveur. Ceux-ci se font au travers d'un réseau et il est bon de rappeler la structure générale des échanges entre deux machines distantes.

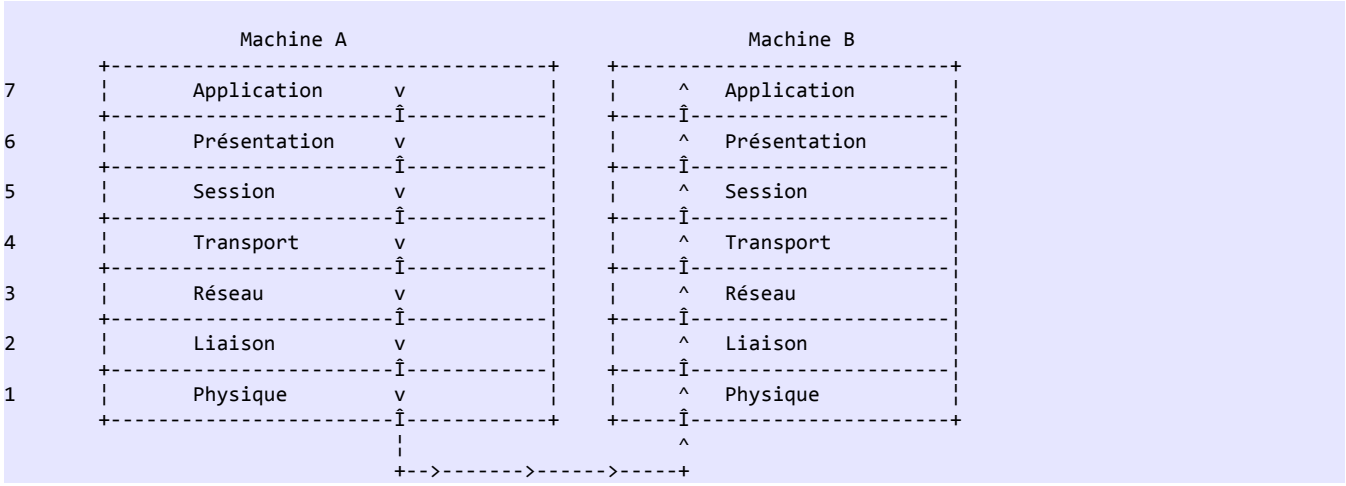
2.4.1 Le modèle OSI

Le modèle de réseau ouvert appelé **OSI** (**O**pen **S**ystems **I**nterconnection Reference Model) défini par l'**ISO** (**I**nternational **S**tandards **O**rganisation) décrit un réseau idéal où la communication entre machines peut être représentée par un modèle à sept couches :

```
        |-----------------------------------|
7       |       Application                 |
        |-----------------------------------|
6       |       Présentation                |
        |-----------------------------------|
5       |       Session                     |
        |-----------------------------------|
4       |       Transport                   |
        |-----------------------------------|
3       |       Réseau                      |
        |-----------------------------------|
2       |       Liaison                     |
        |-----------------------------------|
1       |       Physique                    |
        |-----------------------------------|
```

Chaque couche reçoit des services de la couche inférieure et offre les siens à la couche supérieure. Supposons que deux applications situées sur des machines A et B différentes veulent communiquer : elles le font au niveau de la couche *Application*. Elles n'ont pas besoin de connaître tous les détails du fonctionnement du réseau : chaque application remet l'information qu'elle souhaite transmettre à la couche du dessous : la couche *Présentation*. L'application n'a donc à connaître que les règles d'interfaçage avec la couche *Présentation*. Une fois l'information dans la couche *Présentation*, elle est passée selon d'autres règles à la couche *Session* et ainsi de suite, jusqu'à ce que l'information arrive sur le support physique et soit transmise physiquement à la machine destination. Là, elle subira le traitement inverse de celui qu'elle a subi sur la machine expéditeur.

A chaque couche, le processus expéditeur chargé d'envoyer l'information, l'envoie à un processus récepteur sur l'autre machine apartenant à la même couche que lui. Il le fait selon certaines règles que l'on appelle le **protocole** de la couche. On a donc le schéma de communication final suivant :



Le rôle des différentes couches est le suivant :

`Physique` Assure la transmission de bits sur un support physique. On trouve dans cette couche des équipements terminaux de traitement des données (E.T.T.D.) tels que terminal ou ordinateur, ainsi que des équipements de terminaison de circuits de données (E.T.C.D.) tels que modulateur/démodulateur, multiplexeur, concentrateur. Les points d'intérêt à ce niveau sont :

- le choix du codage de l'information (analogique ou numérique)
- le choix du mode de transmission (synchrone ou asynchrone).

`Liaison de données` Masque les particularités physiques de la couche Physique. Détecte et corrige les erreurs de transmission.

`Réseau` Gère le chemin que doivent suivre les informations envoyées sur le réseau. On appelle cela le *routage* : déterminer la route à suivre par une information pour qu'elle arrive à son destinataire.

`Transport` Permet la communication entre deux applications alors que les couches précédentes ne permettaient que la communication entre machines. Un service fourni par cette couche peut être le multiplexage : la couche transport pourra utiliser une même connexion réseau (de machine à machine) pour transmettre des informations appartenant à plusieurs applications.

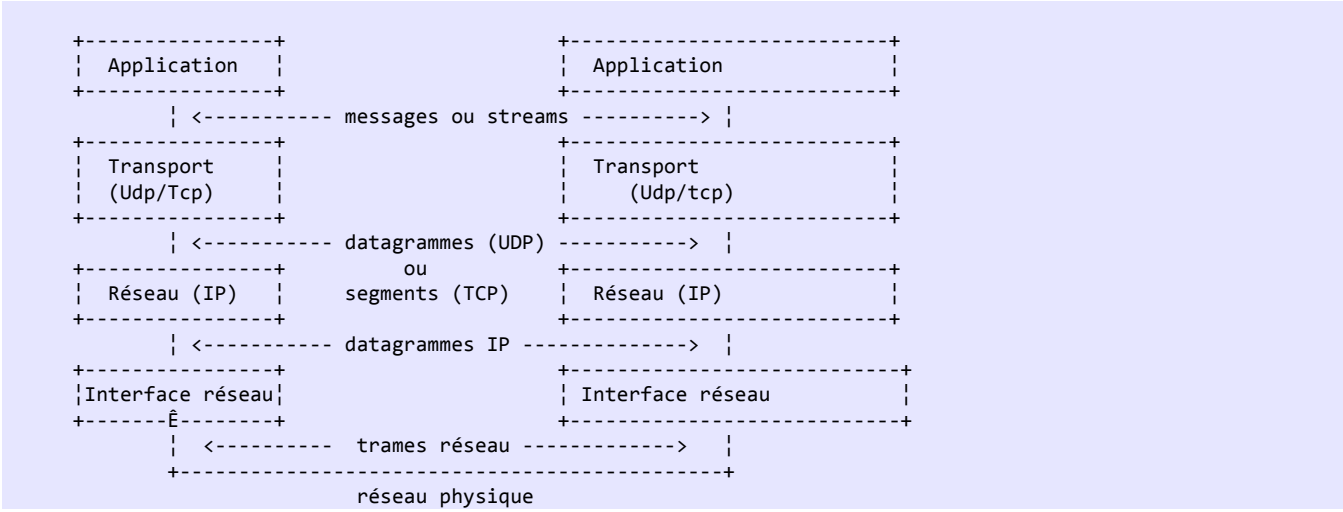
`Session` On va trouver dans cette couche des services permettant à une application d'ouvrir et de maintenir une session de travail sur une machine distante.

`Présentation` Elle vise à uniformiser la représentation des données sur les différentes machines. Ainsi des données provenant d'une machine A, vont être "habillées" par la couche *Présentation* de la machine A, selon un format standard avant d'être envoyées sur le réseau. Parvenues à la couche *Présentation* de la machine destinatrice B qui les reconnaîtra grâce à leur format standard, elles seront habillées d'une autre façon afin que l'application de la machine B les reconnaisse.

`Application` A ce niveau, on trouve les applications généralement proches de l'utilisateur telles que la messagerie électronique ou le transfert de fichiers.

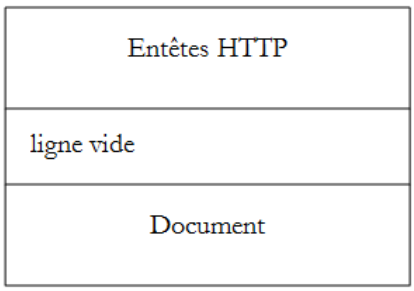
2.4.2 Le modèle TCP/IP

Le modèle OSI est un modèle idéal. La suite de protocoles TCP/IP s'en approche sous la forme suivante :



- l'**interface réseau** (la carte réseau de l'ordinateur) assure les fonctions des couches 1 et 2 du modèle OSI
- la couche **IP** (Internet Protocol) assure les fonctions de la couche 3 (réseau)
- la couche **TCP** (Transfer Control Protocol) ou **UDP** (User Datagram Protocol) assure les fonctions de la couche 4 (transport). Le protocole TCP s'assure que les paquets de données échangés par les machines arrivent bien à destination. Si ce n'est pas les cas, il renvoie les paquets qui se sont égarés. Le protocole UDP ne fait pas ce travail et c'est alors au développeur d'applications de le faire. C'est pourquoi sur l'internet qui n'est pas un réseau fiable à 100%, c'est le protocole TCP qui est le plus utilisé. On parle alors de réseau **TCP-IP**.
- la couche **Application** recouvre les fonctions des niveaux 5 à 7 du modèle OSI.

Les applications Web se trouvent dans la couche *Application* et s'appuient donc sur les protocoles TCP-IP. Les couches *Application* des machines clientes et serveur s'échangent des messages qui sont confiées aux couches 1 à 4 du modèle pour être acheminées à destination. Pour se comprendre, les couches application des deux machines doivent "parler" un même langage ou protocole. Celui des applications Web s'appelle **HTTP** (HyperText Transfer Protocol). C'est un protocole de type texte, c.a.d. que les machines échangent des lignes de texte sur le réseau pour se comprendre. Ces échanges sont normalisés, ç-à-d. que le client dispose d'un certain nombre de messages pour indiquer exactement ce qu'il veut au serveur et ce dernier dispose également d'un certain nombre de messages pour donner au client sa réponse. Cet échange de messages a la forme suivante :



Client --> Serveur
Lorsque le client fait sa demande au serveur Web, il envoie

1. des lignes de texte au format HTTP pour indiquer ce qu'il veut ;
2. une ligne vide ;
3. optionnellement un document.

Serveur --> Client
Lorsque le serveur fait sa réponse au client, il envoie

1. des lignes de texte au format HTTP pour indiquer ce qu'il envoie ;
2. une ligne vide ;
3. optionnellement un document.

Les échanges ont donc la même forme dans les deux sens. Dans les deux cas, il peut y avoir envoi d'un document même s'il est rare qu'un client envoie un document au serveur. Mais le protocole HTTP le prévoit. C'est ce qui permet par exemple aux abonnés d'un fournisseur d'accès de télécharger des documents divers sur leur site personnel hébergé chez ce fournisseur d'accès. Les documents échangés peuvent être quelconques. Prenons un navigateur demandant une page Web contenant des images :

1. le navigateur se connecte au serveur Web et demande la page qu'il souhaite. Les ressources demandées sont désignées de façon unique par des URL (Uniform Resource Locator). Le navigateur n'envoie que des entêtes HTTP et aucun document.
2. le serveur lui répond. Il envoie tout d'abord des entêtes HTTP indiquant quel type de réponse il envoie. Ce peut être une erreur si la page demandée n'existe pas. Si la page existe, le serveur dira dans les entêtes HTTP de sa réponse qu'après ceux-ci il va envoyer un document **HTML** (HyperText Markup Language). Ce document est une suite de lignes de texte au format HTML. Un texte HTML contient des balises (marqueurs) donnant au navigateur des indications sur la façon d'afficher le texte.
3. le client sait d'après les entêtes HTTP du serveur qu'il va recevoir un document HTML. Il va analyser celui-ci et s'apercevoir peut-être qu'il contient des références d'images. Ces dernières ne sont pas dans le document HTML. Il fait donc une nouvelle demande au même serveur Web pour demander la première image dont il a besoin. Cette demande est identique à celle faite en 1, si ce n'est que la resource demandée est différente. Le serveur va traiter cette demande en envoyant à son client l'image demandée. Cette fois-ci, dans sa réponse, les entêtes HTTP préciseront que le document envoyé est une image et non un document HTML.
4. le client récupère l'image envoyée. Les étapes 3 et 4 vont être répétées jusqu'à ce que le client (un navigateur en général) ait tous les documents lui permettant d'afficher l'intégralité de la page.

2.4.3 Le protocole HTTP

Découvrons le protocole HTTP sur des exemples. Que s'échangent un navigateur et un serveur Web ?

Le service Web ou service HTTP est un service TCP-IP qui travaille habituellement sur le port 80. Il pourrait travailler sur un autre port. Dans ce cas, le navigateur client serait obligé de préciser ce port dans l'URL qu'il demande. Une URL a la forme générale suivante :

protocole://machine[:port]/chemin/infos

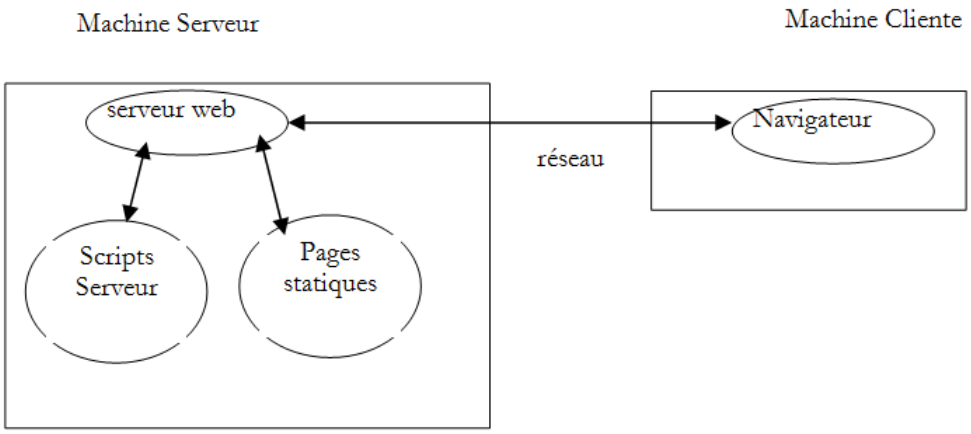
avec

`protocole`	http pour le service Web. Un navigateur peut également servir de client à des services ftp, news, telnet, ..
`machine`	nom de la machine où officie le service Web
`port`	port du service Web. Si c'est 80, on peut omettre le n° du port. C'est le cas le plus fréquent
`chemin`	chemin désignant la ressource demandée
`infos`	informations complémentaires données au serveur pour préciser la demande du client

Que fait un navigateur lorsqu'un utilisateur demande le chargement d'une URL ?

1. il ouvre une communication TCP-IP avec la machine et le port indiqués dans la partie **machine[:port]** de l'URL. Ouvrir une communication TCP-IP, c'est créer un "tuyau" de communication entre deux machines. Une fois ce tuyau créé, toutes les informations échangées entre les deux machines vont passer dedans. La création de ce tuyau TCP-IP n'implique pas encore le protocole HTTP du Web.
2. le tuyau TCP-IP créé, le client va faire sa demande au serveur Web et il va la faire en lui envoyant des lignes de texte (des commandes) au format HTTP. Il va envoyer au serveur la partie **chemin/infos** de l'URL
3. le serveur lui répondra de la même façon et dans le même tuyau
4. l'un des deux partenaires prendra la décision de fermer le tuyau. Cela dépend du protocole HTTP utilisé. Avec le protocole HTTP 1.0, le serveur ferme la connexion après chacune de ses réponses. Cela oblige un client qui doit faire plusieurs demandes pour obtenir les différents documents constituant une page Web à ouvrir une nouvelle connexion à chaque demande, ce qui a un coût. Avec le protocole HTTP/1.1, le client peut dire au serveur de garder la connexion ouverte jusqu'à ce qu'il lui dise de la fermer. Il peut donc récupérer tous les documents d'une page Web avec une seule connexion et fermer lui-même la connexion une fois le dernier document obtenu. Le serveur détectera cette fermeture et fermera lui aussi la connexion.

Pour découvrir les échanges entre un client et un serveur Web, nous allons utiliser l'extension [Advanced Rest Client] du navigateur Chrome que nous avons installée page Erreur : source de la référence non trouvée. Nous serons dans la situation suivante :

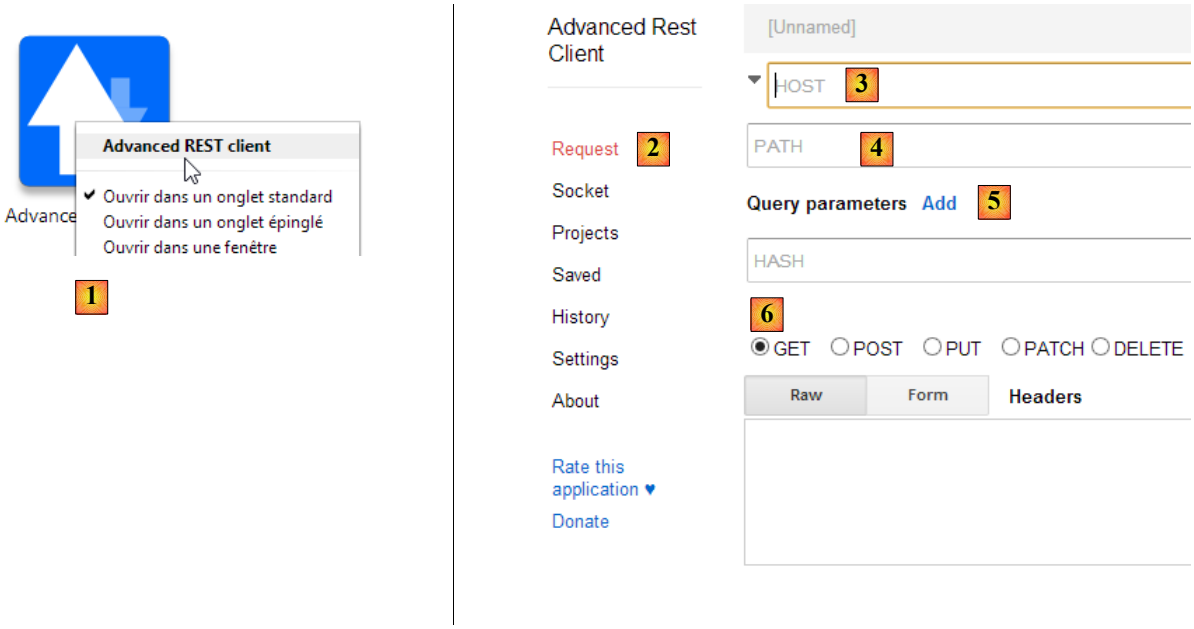


Le serveur Web pourra être quelconque. Nous cherchons ici à découvrir les échanges qui vont se produire entre navigateur et le serveur Web. Précédemment, nous avons créé la page HTML statique suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>essai 1 : une page statique</title>
6.  </head>
7.  <body>
8.    <h1>Une page statique...</h1>
9.  </body>
10. </html>
```

que nous visualisons dans un navigateur :

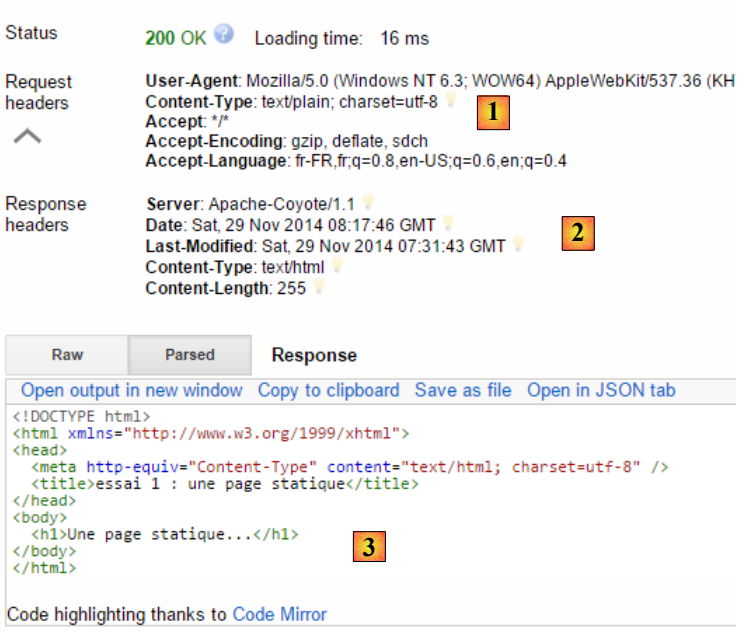
On voit que l'URL demandée est : [http://localhost:9000/intro/exemple-01.html]. La machine du service Web est donc *localhost* (=machine locale) et le port 9000. Utilisons l'application [Advanced Rest Client] pour demander la même URL :



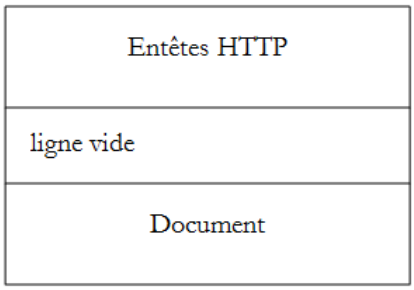
- en [1], on lance l'application (dans l'onglet [Applications] d'un nouvel onglet Chrome) ;
- en [2], on sélectionne l'option [Request] ;
- en [3], on précise le serveur interrogé : http://localhost:9000;
- en [4], on précise l'URL demandée : /intro/exemple-01.html ;
- en [5], on ajoute d'éventuels paramètres à l'URL. Aucun ici ;
- en [6], on précise la commande HTTP utilisée pour la requête, ici GET.

Cela donne la requête suivante :

La requête ainsi préparée [7] est envoyée au serveur par [8]. La réponse obtenue est alors la suivante :



Nous avons dit plus haut que les échanges client-serveur avaient la forme suivante :



- en [1], on voit les entêtes HTTP envoyés par le navigateur dans sa requête. Il n'avait pas de document à envoyer ;
- en [2], on voit les entêtes HTTP envoyés par le serveur en réponse. En [3], on voit le document qu'il a envoyé.

En [3], on reconnaît la page HTML statique que nous avons placée sur le serveur web.

Examinons la requête HTTP du navigateur :

```
1.  GET /intro/exemple-01.html HTTP/1.1
2.  Host: localhost:9000
3.  Connection: keep-alive
4.  Pragma: no-cache
5.  Cache-Control: no-cache
6.  User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71
    Safari/537.36
7.  Content-Type: text/plain; charset=utf-8
8.  Accept: */*
9.  Accept-Encoding: gzip, deflate, sdch
10. Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
```

- la ligne 1 n'a pas été affichée par l'application ;
- ligne 6 : le navigateur s'identifie avec l'entête [User-Agent] ;
- ligne 7 : le navigateur indique qu'il envoie au serveur un document texte (text/plain) au format UTF-8. En fait ici, le navigateur n'a envoyé aucun document ;
- ligne 8 : le navigateur indique qu'il accepte tout type de document en réponse ;
- ligne 9 : le navigateur précise les formats de document acceptés ;
- ligne 10 : le navigateur précise les langues qu'il souhaite par ordre de préférence.

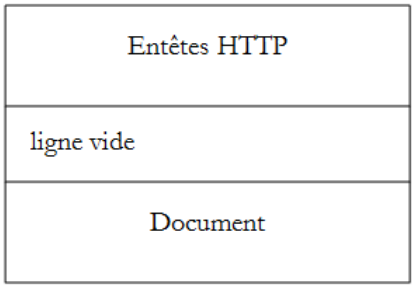
Le serveur lui a répondu en envoyant les entêtes HTTP suivants :

```
1.  HTTP/1.1 200 OK
2.  Server: Apache-Coyote/1.1
3.  Last-Modified: Sat, 29 Nov 2014 07:31:43 GMT
4.  Content-Type: text/html
5.  Content-Length: 255
6.  Date: Sat, 29 Nov 2014 08:20:52 GMT
```

- ligne 1 : n'a pas été affichée par l'application ;
- ligne 2 : le serveur s'identifie, ici un serveur Apache-Coyote ;
- ligne 3 : la date de dernière modification du document envoyé ;
- ligne 4 : la nature du document envoyé par le serveur. Ici un document HTML ;
- ligne 5 : la taille en octets du document HTML envoyé.
- ligne 6 : date et heure de la réponse ;

2.4.4 Conclusion

Nous avons découvert la structure de la demande d'un client Web et celle de la réponse qui lui est faite par le serveur Web sur quelques exemples. Le dialogue se fait à l'aide du protocole HTTP, un ensemble de commandes au format texte échangées par les deux partenaires. La requête du client et la réponse du serveur ont la même structure suivante :



Les deux commandes usuelles pour demander une ressource sont GET et POST. La commande GET n'est pas accompagnée d'un document. La commande POST elle, est accompagnée d'un document qui est le plus souvent une chaîne de caractères rassemblant l'ensemble des valeurs saisies dans un formulaire. La commande HEAD permet de demander seulement les entêtes HTTP et n'est pas accompagnée de document.

A la demande d'un client, le serveur envoie une réponse qui a la même structure. La ressource demandée est transmise dans la partie [Document] sauf si la commande du client était HEAD, auquel cas seuls les entêtes HTTP sont envoyés.

2.5 Les bases du langage HTML

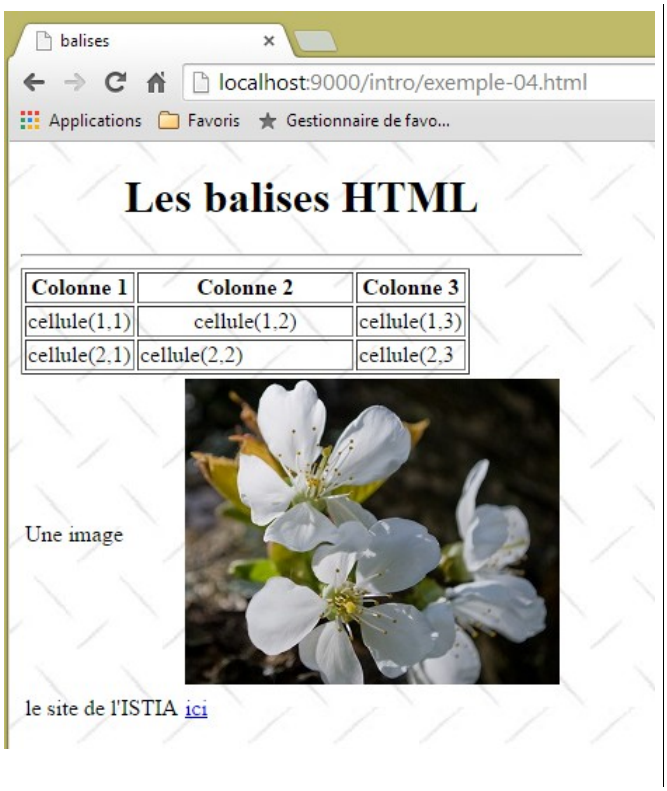
Un navigateur Web peut afficher divers documents, le plus courant étant le document HTML (HyperText Markup Language). Celui-ci est un texte formaté avec des balises de la forme <*balise*>*texte*</*balise*>. Ainsi le texte <*B*>*important*</*B*> affichera le texte important en gras. Il existe des balises seules telles que la balise <*hr*/> qui affiche une ligne horizontale. Nous ne passerons pas en revue les balises que l'on peut trouver dans un texte HTML. Il existe de nombreux logiciels WYSIWYG permettant de construire une page Web sans écrire une ligne de code HTML. Ces outils génèrent automatiquement le code HTML d'une mise en page faite à l'aide de la souris et de contrôles prédéfinis. On peut ainsi insérer (avec la souris) dans la page un tableau puis consulter le code HTML généré par le logiciel pour découvrir les balises à utiliser pour définir un tableau dans une page Web. Ce n'est pas plus compliqué que cela. Par ailleurs, la connaissance du langage HTML est indispensable puisque les applications Web dynamiques doivent générer elles-mêmes le code HTML à envoyer aux clients Web. Ce code est généré par programme et il faut bien sûr savoir ce qu'il faut générer pour que le client ait la page Web qu'il désire.

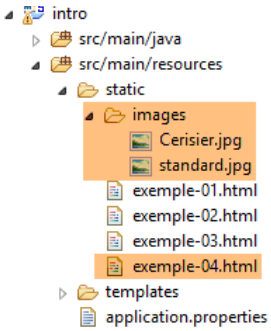
Pour résumer, il n'est nul besoin de connaître la totalité du langage HTML pour démarrer la programmation Web. Cependant cette connaissance est nécessaire et peut être acquise au travers de l'utilisation de logiciels WYSIWYG de construction de pages Web tels que DreamWeaver et des dizaines d'autres. Une autre façon de découvrir les subtilités du langage HTML est de parcourir le Web et d'afficher le code source des pages qui présentent des caractéristiques intéressantes et encore inconnues pour vous.

2.5.1 Un exemple

Considérons l'exemple suivant qui présente quelques éléments qu'on peut trouver dans un document Web tels que :

- un tableau ;
- une image ;
- un lien.





Un document HTML a la forme générale suivante :

```
<html>
   <head>
      <title>Un titre</title>
      ...
   </head>
   <body attributs>
      ...
   </body>
</html>
```

L'ensemble du document est encadré par les balises **<html>...</html>**. Il est formé de deux parties :

1. **<head>...</head>** : c'est la partie non affichable du document. Elle donne des renseignements au navigateur qui va afficher le document. On y trouve souvent la balise **<title>...</title>** qui fixe le texte qui sera affiché dans la barre de titre du navigateur. On peut y trouver d'autres balises notamment des balises définissant les mots clés du document, mot clés utilisés ensuite par les moteurs de recherche. On peut trouver également dans cette partie des scripts, écrits le plus souvent en javascript ou vbscript et qui seront exécutés par le navigateur.

2. **<body attributs>...</body>** : c'est la partie qui sera affichée par le navigateur. Les balises HTML contenues dans cette partie indiquent au navigateur la forme visuelle "souhaitée" pour le document. Chaque navigateur va interpréter ces balises à sa façon. Deux navigateurs peuvent alors visualiser différemment un même document Web. C'est généralement l'un des casse-têtes des concepteurs Web.

Le code HTML de notre document exemple est le suivant :

```
1. <!DOCTYPE html>
```

```
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.  <title>balises</title>
6.  </head>
7.
8.  <body style="height: 400px; width: 400px; background-image: url(images/standard.jpg)">
9.     <h1 style="text-align: center">Les balises HTML</h1>
10.    <hr />
11.    <table border="1">
12.       <thead>
13.          <tr>
14.             <th>Colonne 1</th>
15.             <th>Colonne 2</th>
16.             <th>Colonne 3</th>
17.          </tr>
18.       </thead>
19.       <tbody>
20.          <tr>
21.             <td>cellule(1,1)</td>
22.             <td style="width: 150px; text-align: center;">cellule(1,2)</td>
23.             <td>cellule(1,3)</td>
24.          </tr>
25.          <tr>
26.             <td>cellule(2,1)</td>
27.             <td>cellule(2,2)</td>
28.             <td>cellule(2,3</td>
29.          </tr>
30.       </tbody>
31.    </table>
32.
33.    <table>
34.       <tr>
35.          <td>Une image</td>
36.          <td><img border="0" src="images/cerisier.jpg" /></td>
37.       </tr>
38.       <tr>
39.          <td>le site de l'ISTIA</td>
40.          <td><a href="http://istia.univ-angers.fr">ici</a></td>
41.       </tr>
42.    </table>
43. </body>
44. </html>
```

`Elément`	balises et exemples HTML
`titre du document`	**<title>balises</title>** (ligne 5) le texte *balises* apparaîtra dans la barre de titre du navigateur qui affichera le document
`barre horizontale`	* * : affiche un trait horizontal (ligne 10)
`tableau`	*<table attributs>....</table>* : pour définir le tableau (lignes 11, 31) *<thead>...</thead>* : pour définir les entêtes des colonnes (lignes 12, 18) *<tbody>...</tbody>* : pour définir le contenu du tableau (ligne 19, 30) *<tr attributs>...</tr>* : pour définir une ligne (lignes 20, 24) *<td attributs>...</td>* : pour définir une cellule (ligne 21) **exemples** :

<table border="1">...</table> : l'attribut border définit l'épaisseur de la bordure du tableau
<td style="width: 150px; text-align: center;">cellule(1,2)</td> : définit une cellule dont le contenu sera **cellule(1,2)**. Ce contenu sera centré horizontalement (text-align :center). La cellule aura une largeur de 150 pixels (width :150px)

`image`

 (ligne 36) : définit une image sans bordure (border=0") dont le fichier source est */images/cerisier.jpg* sur le serveur Web (src="images/cerisier.jpg"). Ce lien se trouve sur un document Web obtenu avec l'URL *http://localhost:port/intro/exemple-04.html*. Aussi, le navigateur demandera-t-il l'URL *http://localhost:port/intro/images/cerisier.jpg* pour avoir l'image référencée ici.

`lien`

ici (ligne 40) : fait que le texte *ici* sert de lien vers l'URL *http://istia.univ-angers.fr*.

`fond de page`

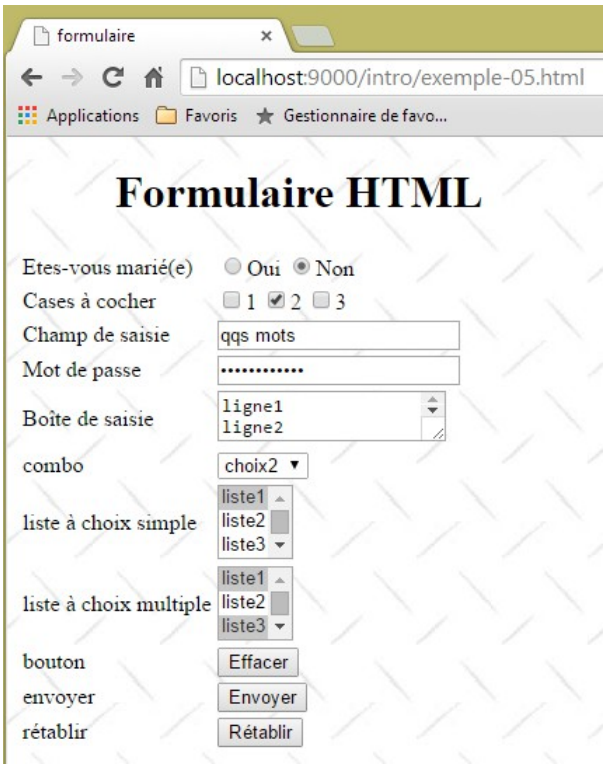
<body style="height:400px;width:400px;background-image:url(images/standard.jpg)"> *(ligne 8)* : indique que l'image qui doit servir de fond de page se trouve à l'URL [*images/standard.jpg*] du serveur Web. Dans le contexte de notre exemple, le navigateur demandera l'URL *http://localhost:port/intro/images/standard.jpg* pour obtenir cette image de fond. Par ailleurs, le corps du document sera affiché dans un rectangle de hauteur 400 pixels et de largeur 400 pixels.

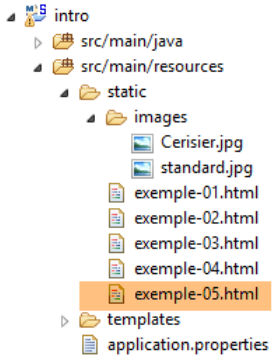
On voit dans ce simple exemple que pour construire l'intéralité du document, le navigateur doit faire trois requêtes au serveur :

1. *http://localhost:port/intro/exemple-04.html* pour avoir le source HTML du document
2. *http://localhost:port/intro/images/cerisier.jpg* pour avoir l'image *cerisier.jpg*
3. *http://localhost:port/intro/images/standard.jpg* pour obtenir l'image de fond *standard.jpg*

2.5.2 Un formulaire HTML

L'exemple suivant présente un formulaire :





Le code HTML produisant cet affichage est le suivant :

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>formulaire</title>
  <script type="text/javascript">
    function effacer() {
      alert("Vous avez cliqué sur le bouton Effacer");
    }
  </script>
</head>

<body style="height: 400px; width: 400px; background-image: url(images/standard.jpg)">
  <h1 style="text-align: center">Formulaire HTML</h1>
  <form method="post" action="postFormulaire">
    <table>
      <tr>
        <td>Etes-vous marié(e)</td>
        <td>
          <input type="radio" value="Oui" name="R1" />Oui
          <input type="radio" name="R1" value="non" checked="checked" />Non
        </td>
      </tr>
      <tr>
        <td>Cases à cocher</td>
        <td>
          <input type="checkbox" name="C1" value="un" />1
          <input type="checkbox" name="C2" value="deux" checked="checked" />2
          <input type="checkbox" name="C3" value="trois" />3
        </td>
      </tr>
      <tr>
        <td>Champ de saisie</td>
        <td>
          <input type="text" name="txtSaisie" size="20" value="qqs mots" />
        </td>
      </tr>
      <tr>
        <td>Mot de passe</td>
        <td>
          <input type="password" name="txtMdp" size="20" value="unMotDePasse" />
        </td>
      </tr>
      <tr>
        <td>Boîte de saisie</td>
        <td>
          <textarea rows="2" name="areaSaisie" cols="20">
ligne1
ligne2
ligne3
</textarea>
        </td>
      </tr>
      <tr>
        <td>combo</td>
        <td>
          <select size="1" name="cmbValeurs">
            <option value="1">choix1</option>
            <option selected="selected" value="2">choix2</option>
            <option value="3">choix3</option>
          </select>
        </td>
      </tr>
      <tr>
        <td>liste à choix simple</td>
        <td>
          <select size="3" name="lst1">
            <option selected="selected" value="1">liste1</option>
            <option value="2">liste2</option>
            <option value="3">liste3</option>
            <option value="4">liste4</option>
            <option value="5">liste5</option>
          </select>
```

```
        </td>
      </tr>
      <tr>
        <td>liste à choix multiple</td>
        <td>
          <select size="3" name="lst2" multiple="multiple">
            <option value="1" selected="selected">liste1</option>
            <option value="2">liste2</option>
            <option selected="selected" value="3">liste3</option>
            <option value="4">liste4</option>
            <option value="5">liste5</option>
          </select>
        </td>
      </tr>
      <tr>
        <td>bouton</td>
        <td>
          <input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()" />
        </td>
      </tr>
      <tr>
        <td>envoyer</td>
        <td>
          <input type="submit" value="Envoyer" name="cmdRenvoyer" />
        </td>
      </tr>
      <tr>
        <td>rétablir</td>
        <td>
          <input type="reset" value="Rétablir" name="cmdRétablir" />
        </td>
      </tr>
    </table>
    <input type="hidden" name="secret" value="uneValeur" />
  </form>
</body>
</html>
```

L'association contrôle visuel <--> balise HTML est le suivant :

	balise HTML
Contrôle formulaire	`<form method="post" action="...">`
champ de saisie	`<input type="text" name="txtSaisie" size="20" value="qqs mots" />`
champ de saisie cachée	`<input type="password" name="txtMdp" size="20" value="unMotDePasse" />`

champ de saisie multilignes	`<textarea rows="2" name="areaSaisie" cols="20">`
	ligne1
	ligne2
	ligne3
	`</textarea>`

boutons radio	`<input type="radio" value="Oui" name="R1" />Oui` `<input type="radio" name="R1" value="non" checked="checked" />Non`

cases à cocher	`<input type="checkbox" name="C1" value="un" />1`
	`<input type="checkbox" name="C2" value="deux" checked="checked" />2`
	`<input type="checkbox" name="C3" value="trois" />3`

Combo	`<select size="1" name="cmbValeurs">`
	`<option value="1">choix1</option>`
	`<option selected="selected" value="2">choix2</option>`
	`<option value="3">choix3</option>`
	`</select>`

liste à sélection unique	`<select size="3" name="lst1">`
	`<option selected="selected" value="1">liste1</option>`
	`<option value="2">liste2</option>`
	`<option value="3">liste3</option>`
	`<option value="4">liste4</option>`

```
                    <option value="5">liste5</option>
                  </select>
```

liste à sélection multiple

```
<select size="3" name="lst2" multiple="multiple">
                    <option value="1">liste1</option>
                    <option value="2">liste2</option>
                    <option selected="selected" value="3">liste3</option>
                    <option value="4">liste4</option>
                    <option value="5">liste5</option>
                  </select>
```

bouton de type submit

```
<input type="submit" value="Envoyer" name="cmdRenvoyer" />
```

bouton de type reset

```
<input type="reset" value="Rétablir" name="cmdRétablir" />
```

bouton de type button

```
<input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()" />
```

Passons en revue ces différentes balises :

2.5.2.1 *Le formulaire*

formulaire

```
<form method="post" action="postFormulaire">
```

balise HTML

<form name="..." method="..." action="...">...</form>

attributs

name="frmexemple" : nom du formulaire

method="..." : méthode utilisée par le navigateur pour envoyer au serveur Web les valeurs récoltées dans le formulaire

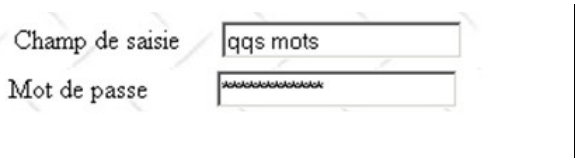
action="..." : URL à laquelle seront envoyées les valeurs récoltées dans le formulaire.

Un formulaire Web est entouré des balises *<form>...</form>*. Le formulaire peut avoir un nom (*name="xx"*). C'est le cas pour tous les contrôles qu'on peut trouver dans un formulaire. Le but d'un formulaire est de rassembler des informations données par l'utilisateur au clavier/souris et d'envoyer celles-ci à une URL de serveur Web. Laquelle ? Celle référencée dans l'attribut *action="URL"*. Si cet attribut est absent, les informations seront envoyées à l'URL du document dans lequel se trouve le formulaire. Un client Web peut utiliser deux méthodes différentes appelées POST et GET pour envoyer des données à un serveur web. L'attribut *method="méthode"*, avec *method* égal à GET ou POST, de la balise *<form>* indique au navigateur la méthode à utiliser pour envoyer les informations recueillies dans le formulaire à l'URL précisée par l'attribut *action="URL"*. Lorsque l'attribut *method* n'est pas précisé, c'est la méthode GET qui est prise par défaut.

2.5.2.2 *Les champs de saisie texte*

champ de saisie

```
<input type="text" name="txtSaisie" size="20" value="qqs mots" />
<input type="password" name="txtMdp" size="20" value="unMotDePasse" />
```



balise HTML

<input type="..." name="..." size=".." value=".."/>

La balise **input** existe pour divers contrôles. C'est l'attribut *type* qui permet de différencier ces différents contrôles entre eux.

attributs

type="text" : précise que c'est un champ de saisie

type="password" : les caractères présents dans le champ de saisie sont remplacés par des caractères *. C'est

la seule différence avec le champ de saisie normal. Ce type de contrôle convient pour la saisie des mots de passe.

size="20" : nombre de caractères visibles dans le champ - n'empêche pas la saisie de davantage de caractères

name="txtSaisie" : nom du contrôle

value="qqs mots" : texte qui sera affiché dans le champ de saisie.

2.5.2.3 *Les champs de saisie multilignes*

champ de saisie multilignes

```
<textarea rows="2" name="areaSaisie" cols="20">
ligne1
ligne2
ligne3
</textarea>
```

balise HTML

<textarea ...>texte</textarea>

attributs

affiche une zone de saisie multilignes avec au départ **texte** dedans
rows="2" : nombre de lignes

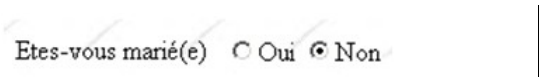
cols="'20" : nombre de colonnes

name="areaSaisie" : nom du contrôle

2.5.2.4 *Les boutons radio*

boutons radio

```
<input type="radio" value="Oui" name="R1" />Oui
<input type="radio" name="R1" value="non" checked="checked" />Non
```



balise HTML

<input type="radio" attribut2="valeur2" ..../>texte

attributs

affiche un bouton radio avec **texte** à côté.
name="radio" : nom du contrôle. Les boutons radio portant le même nom forment un groupe de boutons exclusifs les uns des autres : on ne peut cocher que l'un d'eux.

value="valeur" : valeur affectée au bouton radio. Il ne faut pas confondre cette valeur avec le texte affiché à côté du bouton radio. Celui-ci n'est destiné qu'à l'affichage.

checked= "checked" : si ce mot clé est présent, le bouton radio est coché, sinon il ne l'est pas.

2.5.2.5 *Les cases à cocher*

cases à cocher

```
<input type="checkbox" name="C1" value="un" />1
<input type="checkbox" name="C2" value="deux" checked="checked" />2
<input type="checkbox" name="C3" value="trois" />3
```

e

value="valeur" : valeur affectée à la case à cocher. Il ne faut pas confondre cette valeur avec le texte affiché à côté du bouton radio. Celui-ci n'est destiné qu'à l'affichage.

checked= "checked" : si ce mot clé est présent, le case à cocher est cochée, sinon elle ne l'est pas.

2.5.2.6 La liste déroulante (combo)

Combo

```
<select size="1" name="cmbValeurs">
            <option value="1">choix1</option>
            <option selected="selected" value="2">choix2</option>
            <option value="3">choix3</option>
</select>
```

balise HTML

<select size=".." name="..">

<option [selected="selected"] value="v">...</option>

...

</select>

affiche dans une liste les textes compris entre les balises <option>...</option>

attributs

name="cmbValeurs" : nom du contrôle.

size="1" : nombre d'éléments de liste visibles. *size="1"* fait de la liste l'équivalent d'un combobox.

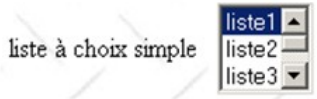
selected="selected" : si ce mot clé est présent pour un élément de liste, ce dernier apparaît sélectionné dans la liste. Dans notre exemple ci-dessus, l'élément de liste *choix2* apparaît comme l'élément sélectionné du combo lorsque celui-ci est affiché pour la première fois.

value="v" : si l'élément est sélectionné par l'utilisateur, c'est cette valeur [v] qui est postée au serveur. En l'absence de cet attribut, c'est le texte affiché et sélectionné qui est posté au serveur.

2.5.2.7 Liste à sélection unique

liste à sélection unique

```
<select size="3" name="lst1">
            <option selected="selected" value="1">liste1</option>
            <option value="2">liste2</option>
            <option value="3">liste3</option>
            <option value="4">liste4</option>
            <option value="5">liste5</option>
</select>
```



balise HTML

<select size=".." name="..">

<option [selected="selected"]>...</option>

...

</select>

affiche dans une liste les textes compris entre les balises *<option>...</option>*

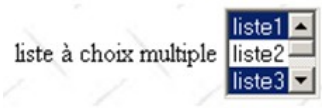
attributs

les mêmes que pour la liste déroulante n'affichant qu'un élément. Ce contrôle ne diffère de la liste déroulante précédente que par son attribut size>1.

2.5.2.8 *Liste à sélection multiple*

liste à sélection unique

```
<select size="3" name="lst2" multiple="multiple">
            <option value="1" selected="selected">liste1</option>
            <option value="2">liste2</option>
            <option selected="selected" value="3">liste3</option>
            <option value="4">liste4</option>
            <option value="5">liste5</option>
</select>
```



balise HTML

<select size=".." name=".." multiple="multiple">

<option [selected="selected"]>...</option>

...

</select>

affiche dans une liste les textes compris entre les balises *<option>...</option>*

attributs

multiple : permet la sélection de plusieurs éléments dans la liste. Dans l'exemple ci-dessus, les éléments *liste1* et *liste3* sont tous deux sélectionnés.

2.5.2.9 *Bouton de type button*

bouton de type button

```
<input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()" />
```

balise HTML

<input type="button" value="..." name="..." onclick="effacer()" ..../>

attributs

type="button" : définit un contrôle bouton. Il existe deux autres types de bouton, les types *submit* et *reset*.

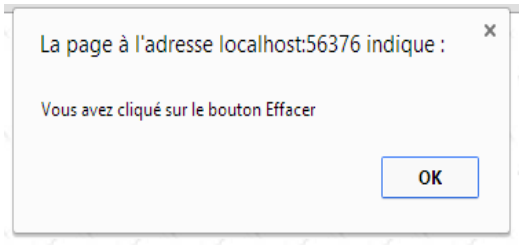
value="Effacer" : le texte affiché sur le bouton

onclick="fonction()" : permet de définir une fonction à exécuter lorsque l'utilisateur clique sur le bouton. Cette fonction fait partie des scripts définis dans le document Web affiché. La syntaxe précédente est une syntaxe *javascript*. Si les scripts sont écrits en vbscript, il faudrait écrire **onclick="fonction"** sans les parenthèses. La syntaxe devient identique s'il faut passer des paramètres à la fonction : **onclick="fonction(val1, val2,...)"**

Dans notre exemple, un clic sur le bouton *Effacer* appelle la fonction javascript *effacer* suivante :

```
<script type="text/javascript">
  function effacer() {
    alert("Vous avez cliqué sur le bouton Effacer");
  }
</script>
```

La fonction *effacer* affiche un message :



2.5.2.10 *Bouton de type submit*

bouton de type submit

```
<input type="submit" value="Envoyer" name="cmdRenvoyer" />
```

balise HTML

```
<input type="submit" value="Envoyer" name="cmdRenvoyer" />
```

attributs

type="submit" : définit le bouton comme un bouton d'envoi des données du formulaire au serveur Web. Lorsque le client va cliquer sur ce bouton, le navigateur va envoyer les données du formulaire à l'URL définie dans l'attribut **action** de la balise **<form>** selon la méthode définie par l'attribut **method** de cette même balise.
value="Envoyer" : le texte affiché sur le bouton

2.5.2.11 *Bouton de type reset*

bouton de type reset

```
<input type="reset" value="Rétablir" name="cmdRétablir" />
```

balise HTML

<input type="reset" value="Rétablir" name="cmdRétablir"/>

attributs

type="reset" : définit le bouton comme un bouton de réinitialisation du formulaire. Lorsque le client va cliquer sur ce bouton, le navigateur va remettre le formulaire dans l'état où il l'a reçu.
value="Rétablir" : le texte affiché sur le bouton

2.5.2.12 *Champ caché*

champ caché

```
<input type="hidden" name="secret" value="uneValeur" />
```

balise HTML

<input type="hidden" name="..." value="..."/>

attributs

type="hidden" : précise que c'est un champ caché. Un champ caché fait partie du formulaire mais n'est pas présenté à l'utilisateur. Cependant, si celui-ci demandait à son navigateur l'affichage du code source, il verrait la présence de la balise <input type="hidden" value="..."> et donc la valeur du champ caché.

value="uneValeur" : valeur du champ caché.

Quel est l'intérêt du champ caché ? Cela peut permettre au serveur Web de garder des informations au fil des requêtes d'un client. Considérons une application d'achats sur le Web. Le client achète un premier article *art1* en quantité *q1* sur une première page d'un catalogue puis passe à une nouvelle page du catalogue. Pour se souvenir que le client a acheté *q1* articles *art1*, le serveur peut mettre ces deux informations dans un champ caché du formulaire Web de la nouvelle page. Sur cette nouvelle page, le client achète *q2* articles *art2*. Lorsque les données de ce second formulaire vont être envoyées au serveur (submit), celui-ci va non seulement recevoir l'information (*q2*,*art2*) mais aussi (*q1*,*art1*) qui fait partie également partie du formulaire en tant que champ caché. Le serveur Web va alors mettre dans un nouveau champ caché les informations (*q1*,*art1*) et (*q2*,*art2*) et envoyer une nouvelle page de catalogue. Et ainsi de suite.

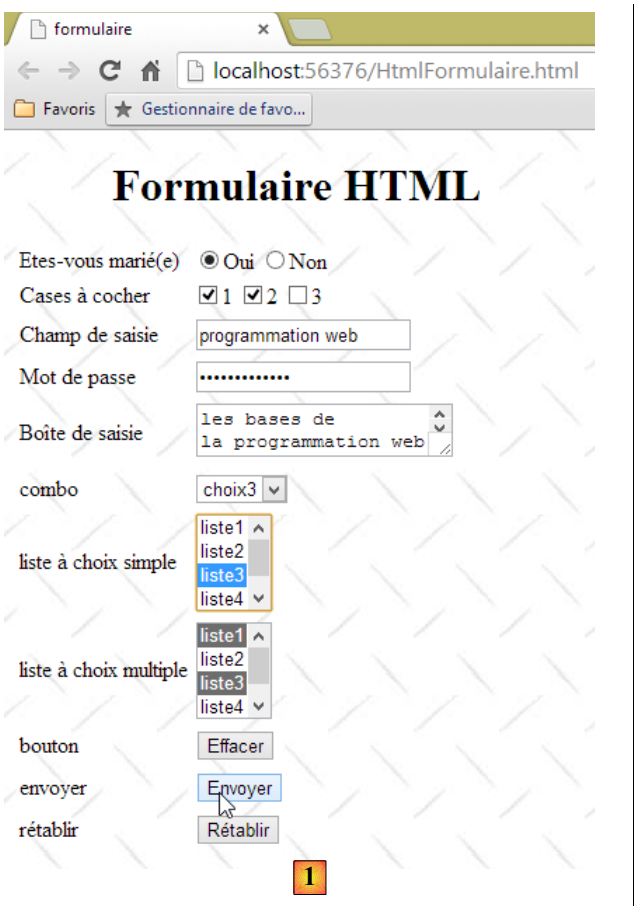
2.5.3 Envoi à un serveur Web par un client Web des valeurs d'un formulaire

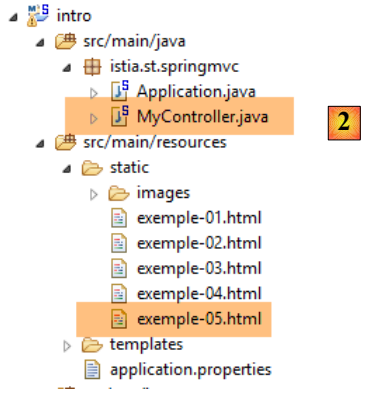
Nous avons dit dans l'étude précédente que le client Web disposait de deux méthodes pour envoyer à un serveur Web les valeurs d'un formulaire qu'il a affiché : les méthodes GET et POST. Voyons sur un exemple la différence entre les deux méthodes.

2.5.3.1 Méthode GET

Faisons un premier test, où dans le code HTML du document, la balise **<form>** est définie de la façon suivante :

```
<form method="get" action="doNothing">
```





Lorsque l'utilisateur va cliquer sur le bouton [1], les valeurs saisies dans le formulaire vont être envoyées au contrôleur Spring [2]. Nous avons vu que les valeurs du formulaire allaient être envoyées à l'URL [doNothing] :

```
<form method="get" action="doNothing">
```

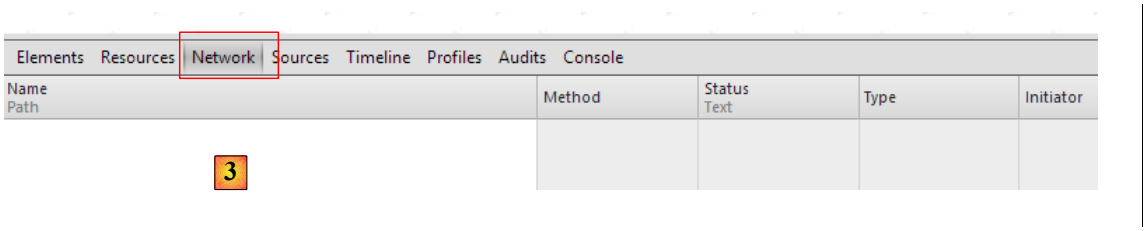
L'action [doNothing] est définie dans le contrôleur [MyController] [2] de la façon suivante :

```
1.      // ---------------------- rendre un flux vide [Content-Length=0]
2.      @RequestMapping(value = "/doNothing")
3.      @ResponseBody
```

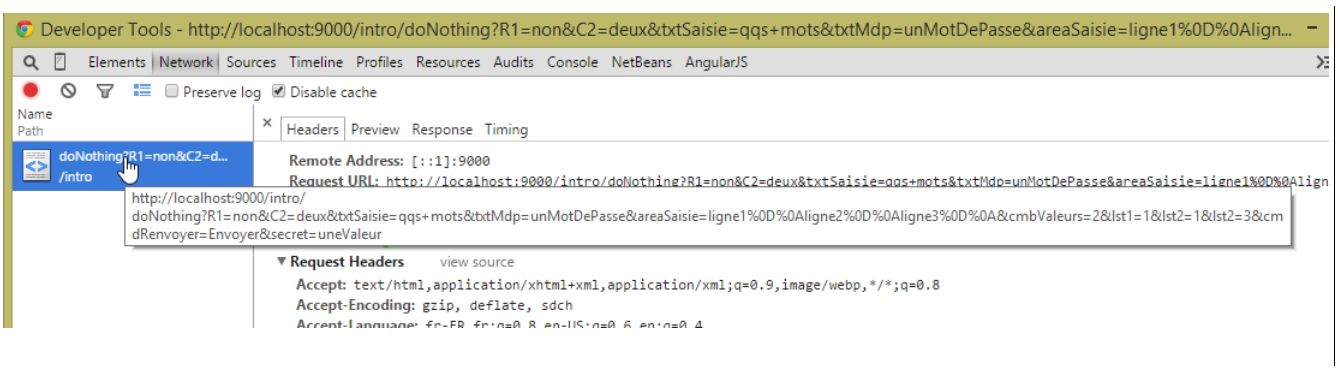
```
4.      public void doNothing() {
5.  }
```

- ligne 1 : l'action traite l'URL [/doNothing] donc en réalité [/context/doNothing] où [context] est le contexte ou nom de l'application web, ici [/intro] ;
- ligne 3 : l'annotation [@ResponseBody] indique que le résultat de la méthode annotée doit être envoyé directement au client ;
- ligne 4 : la méthode ne rend rien. Donc le client recevra une réponse vide du serveur.

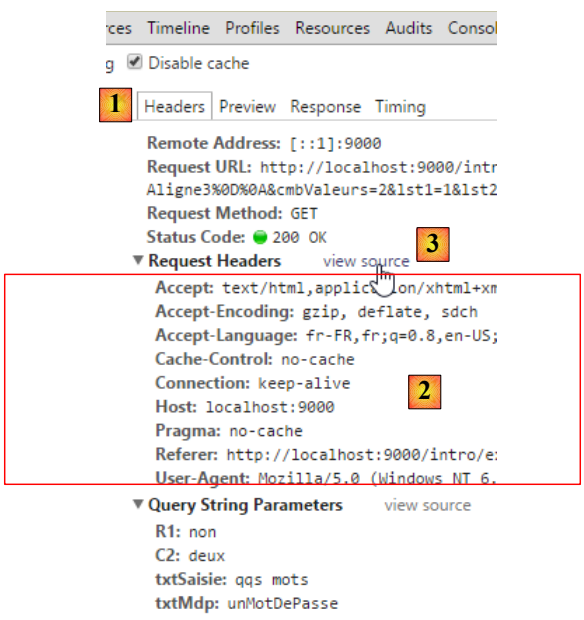
On veut seulement savoir comment le navigateur transmet les valeurs saisies au serveur web. Pour cela, nous allons utiliser un outil de débogage disponible dans Chrome. On l'active en tapant CTRL-Maj-I (majuscule) [3] :

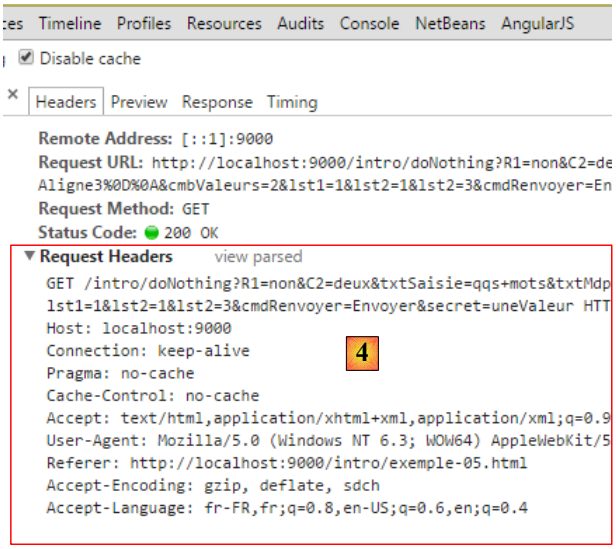


Comme nous nous intéressons aux échanges réseau entre le navigateur et le serveur web, nous activons ci-dessus l'onglet [Network] puis nous cliquons sur le bouton [Envoyer] du formulaire. Celui-ci est un bouton de type [submit] à l'intérieur d'une balise [form]. Le navigateur réagit au clic en demandant l'URL [/intro/doNothing] indiquée dans l'attribut [action] de la balise [form], avec la méthode GET indiquée dans l'attribut [method]. Nous obtenons alors les informations suivantes :



La copie d'écran ci-dessus nous montre l'URL demandée par le navigateur à l'issue du clic sur le bouton [envoyer]. Il demande bien l'URL prévue [/intro/doNothing] mais derrière il rajoute des informations qui sont les valeurs saisies dans le formulaire. Pour avoir plus d'informations, nous cliquons sur le lien ci-dessus :





Ci-dessus [1, 2], nous voyons les entêtes HTTP envoyés par le navigateur. Ils ont été ici mis en forme. Pour voir le texte brut de ces entêtes, nous suivons le lien [view source] [3, 4]. Le texte complet est le suivant :

```
1.  GET /intro/doNothing?R1=non&C2=deux&txtSaisie=qqs+mots&txtMdp=unMotDePasse&areaSaisie=ligne1%0D
    %0Aligne2%0D%0Aligne3%0D%0A&cmbValeurs=2&lst1=1&lst2=1&lst2=3&cmdRenvoyer=Envoyer&secret=uneValeur
    HTTP/1.1
2.  Host: localhost:9000
3.  Connection: keep-alive
4.  Pragma: no-cache
5.  Cache-Control: no-cache
6.  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
7.  User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71
    Safari/537.36
8.  Referer: http://localhost:9000/intro/exemple-05.html
9.  Accept-Encoding: gzip, deflate, sdch
10. Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
```

Nous retrouvons des éléments déjà rencontrés précédemment. D'autres apparaissent pour la première fois :

`Connection: keep-alive`	le client demande au serveur de ne pas fermer la connexion après sa réponse. Cela lui permettra d'utiliser la même connexion pour une demande ultérieure. La connexion ne reste pas ouverte indéfiniment. Le serveur la fermera après un trop long délai d'inutilisation.
`Referer`	l'URL qui était affichée dans le navigateur lorsque la nouvelle demande a été faite.

La nouveauté est ligne 1 dans les informations qui suivent l'URL. On constate que les choix faits dans le formulaire se retrouvent dans l'URL. Les valeurs saisies par l'utilisateur dans le formulaire ont été passées dans la commande *GET URL?param1=valeur1&param2=valeur2&... HTTP/1.1* où les *parami* sont les noms (attribut *name*) des contrôles du formulaire Web et *valeuri* les valeurs qui leur sont associées. Nous présentons ci-dessous un tableau à trois colonnes :

- colonne 1 : reprend la définition d'un contrôle HTML de l'exemple ;
- colonne 2 : donne l'affichage de ce contrôle dans un navigateur ;
- colonne 3 : donne la valeur envoyée au serveur par le navigateur pour le contrôle de la colonne 1 sous la forme qu'elle a dans la requête GET de l'exemple.

contrôle HTML	visuel	valeur(s) renvoyée(s)
`<input type="radio" value="Oui" name="R1"/>Oui` `<input type="radio" name="R1" value="non" checked="checked"/>Non`	Etes-vous marié(e) ◉ Oui ○ Non	`R1=Oui` - la valeur de l'attribut *value* du bouton radio coché par l'utilisateur.
`<input type="checkbox" name="C1" value="un"/>1`	Cases à cocher ☑ 1 ☑ 2 ☐ 3	**`C1=un`**

```
<input type="checkbox" name="C2"
value="deux" checked="checked"/>2

<input type="checkbox" name="C3"
value="trois"/>3
```

C2=deux

- valeurs des attributs *value* des cases cochées par l'utilisateur

```
<input type="text" name="txtSaisie"
size="20" value="qqs mots"/>
```

txtSaisie=programmation+Web

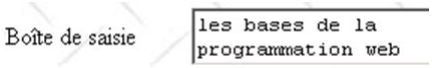
- texte tapé par l'utilisateur dans le champ de saisie. Les espaces ont été remplacés par le signe +

```
<input type="password" name="txtMdp"
size="20" value="unMotDePasse"/>
```

txtMdp=ceciestsecret

- texte tapé par l'utilisateur dans le champ de saisie

```
<textarea rows="2" name="areaSaisie"
cols="20">
ligne1
ligne2
ligne3
</textarea>
```



areaSaisie=les+bases+de+la%0D%0A

programmation+Web

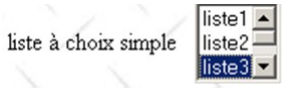
- texte tapé par l'utilisateur dans le champ de saisie. %OD%OA est la marque de fin de ligne. Les espaces ont été remplacés par le signe +

```
<select size="1" name="cmbValeurs">
<option value='1'>choix1</option>
<option selected="selected"
value='2'>choix2</option>
<option value='3'>choix3</option>
</select>
```

cmbValeurs=3

- attribut [value] de l'élément sélectionné par l'utilisateur

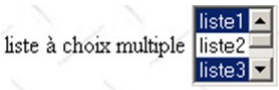
```
<select size="3" name="lst1">
<option selected="selected"
value='1'>liste1</option>
<option value='2'>liste2</option>
<option value='3'>liste3</option>
<option value='4'>liste4</option>
<option value='5'>liste5</option>
</select>
```



lst1=3

- attribut [value] de l'élément sélectionné par l'utilisateur

```
<select size="3" name="lst2"
multiple="multiple">
<option selected="selected"
value='1'>liste1</option>
<option value='2'>liste2</option>
<option selected="selected"
value='3'>liste3</option>
<option value='4'>liste4</option>
<option value='5'>liste5</option>
</select>
```



lst2=1

lst2=3

- attributs [value] des éléments sélectionnés par l'utilisateur

```
<input type="submit" value="Envoyer"
name="cmdRenvoyer"/>
```

cmdRenvoyer=Envoyer

- nom et attribut *value* du bouton qui a servi à envoyer les données du formulaire au serveur

```
<input type="hidden" name="secret"
value="uneValeur"/>
```

secret=uneValeur

- attribut *value* du champ caché

2.5.3.2 *Méthode POST*

Nous changeons le document HTML pour que le navigateur utilise maintenant la méthode POST pour envoyer les valeurs du formulaire au serveur Web :

```
<form method="post" action="doNothing">
```

Nous remplissons le formulaire tel que pour la méthode GET et nous transmettons les paramètres au serveur avec le bouton [Envoyer]. Comme il a été fait au paragraphe précédent page 72, nous avons accès dans Chrome aux entêtes HTTP de la requête envoyée par le navigateur :

```
1.  POST /intro/doNothing HTTP/1.1
2.  Host: localhost:9000
3.  Connection: keep-alive
4.  Content-Length: 172
5.  Pragma: no-cache
6.  Cache-Control: no-cache
7.  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
8.  Origin: http://localhost:9000
9.  User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71
    Safari/537.36
10. Content-Type: application/x-www-form-urlencoded
11. Referer: http://localhost:9000/intro/exemple-05.html
12. Accept-Encoding: gzip, deflate
13. Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
14.
15. R1=non&C2=deux&txtSaisie=qqs+mots&txtMdp=unMotDePasse&areaSaisie=ligne1%0D%0Aligne2%0D%0Aligne3%0D
    %0A&cmbValeurs=2&lst1=1&lst2=1&lst2=3&cmdRenvoyer=Envoyer&secret=uneValeur
```

Des nouveautés apparaissent dans la requête HTTP du client :

`POST URL HTTP/1.1`	la requête GET a laissé place à une requête POST. Les paramètres ne sont plus présents dans cette première ligne de la requête. On peut constater qu'ils sont maintenant placés (ligne 15) derrière la requête HTTP après une ligne vide. Leur encodage est identique à celui qu'ils avaient dans la requête GET.
`Content-Length`	nombre de caractères "postés", c.a.d. le nombre de caractères que devra lire le serveur Web après avoir reçu les entêtes HTTP pour récupérer le document que lui envoie le client. Le document en question est ici la liste des valeurs du formulaire.
`Content-type`	précise le type du document que le client enverra après les entêtes HTTP. Le type [application/x-www-form-urlencoded] indique que c'est un document contenant des valeurs de formulaire.

Il y a deux méthodes pour transmettre des données à un serveur Web : GET et POST. Y-a-t-il une méthode meilleure que l'autre ? Nous avons vu que si les valeurs d'un formulaire étaient envoyées par le navigateur avec la méthode GET, le navigateur affichait dans son champ *Adresse* l'URL demandée sous la forme *URL?param1=val1&param2=val2&...*. On peut voir cela comme un avantage ou un inconvénient :

- un avantage si on veut permettre à l'utilisateur de placer cette URL paramétrée dans ses liens favoris ;
- un inconvénient si on ne souhaite pas que l'utilisateur ait accès à certaines informations du formulaire tels, par exemple, les champs cachés.

Par la suite, nous utiliserons quasi exclusivement la méthode POST dans nos formulaires.

2.6 Conclusion

Ce chapitre a présenté différents concepts de base du développement Web :

- les échanges client-serveur via le protocole HTTP ;
- la conception d'un document à l'aide du langage HTML ;
- la conception de formulaires de saisie.

Nous avons pu voir sur un exemple comment un client pouvait envoyer des informations au serveur Web. Nous n'avons pas présenté comment le serveur pouvait

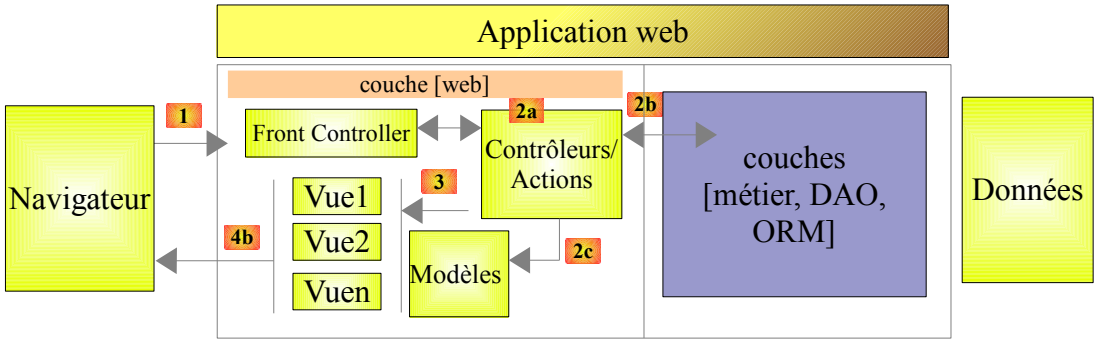
- récupérer ces informations ;

- les traiter ;
- envoyer au client une réponse dynamique dépendant du résultat du traitement.

C'est le domaine de la programmation Web, domaine que nous abordons dans le chapitre suivant avec la présentation de la technologie Spring MVC.

3 Actions : la réponse

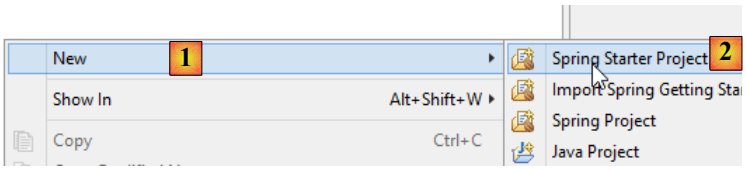
Considérons l'architecture d'une application Spring MVC :



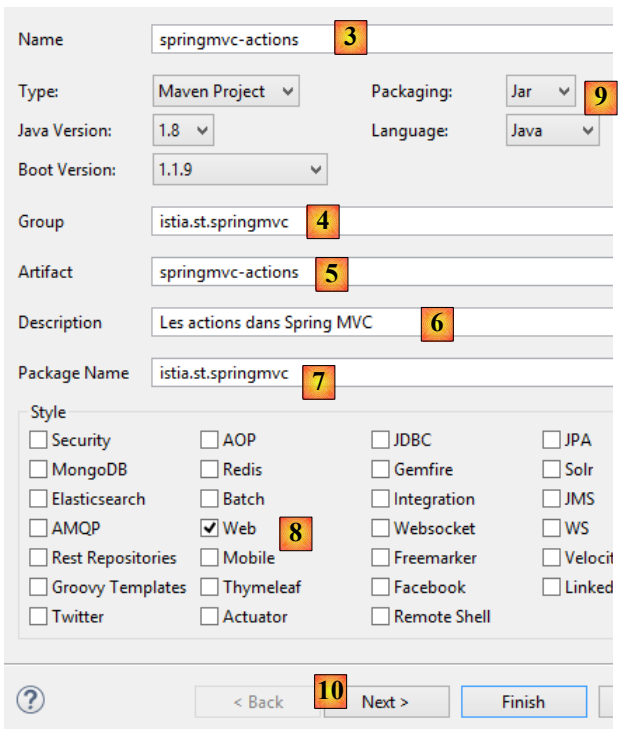
Dans ce chapitre, nous regardons le processus qui amène la requête [1] au contrôleur et à l'action [2a] qui vont la traiter, un mécanisme qu'on appelle le routage. Nous présentons par ailleurs les différentes réponses [3] que peut faire une action au navigateur. Ce peut être autre chose qu'une vue **V** [4b].

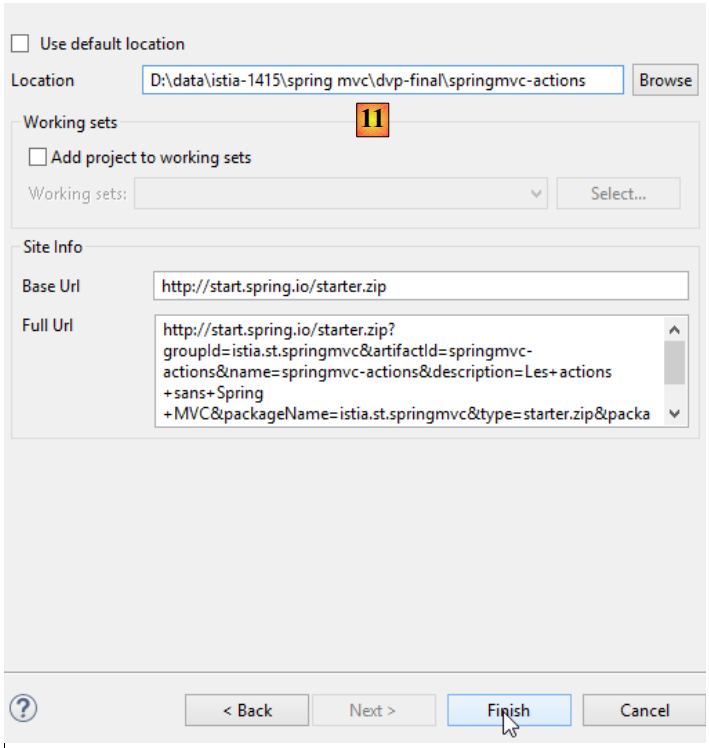
3.1 Le nouveau projet

Nous créons un nouveau projet Spring MVC :

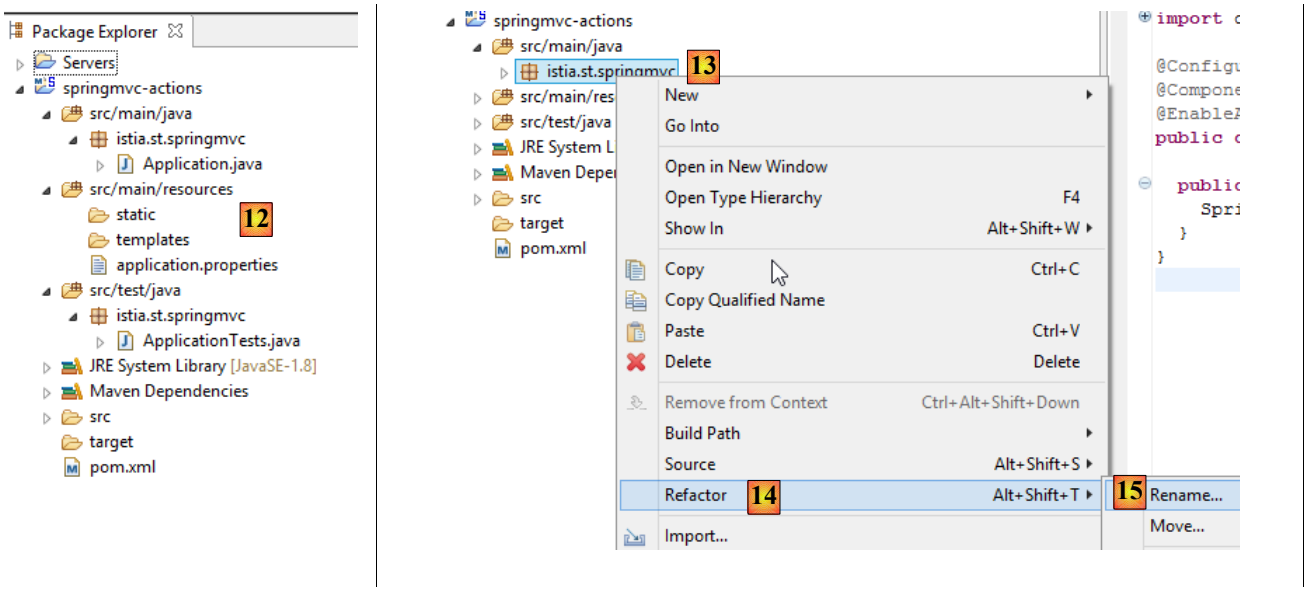


- en [1-2], nous créons un nouveau projet basé sur Spring Boot ;

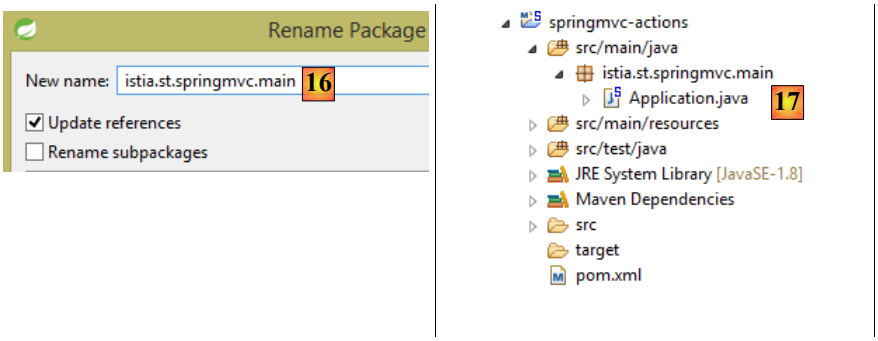




- en [3], le nom du projet Maven ;
- en [4], le groupe Maven dans lequel sera placé le résultat de la compilation du projet ;
- en [5], le nom donné au produit de la compilation ;
- en [6], une description du projet ;
- en [7], le package dans lequel sera placée la classe exécutable du projet ;
- en [8], la nature du projet. C'est un projet web avec des vues Thymeleaf. On voit ici, toutes les dépendances Maven prêtes à l'emploi offertes par le projet Spring Boot ;
- en [9], on indique que le produit issu du build Maven sera packagé dans une archive **jar** et non **war**. Le projet va alors utiliser un serveur Tomcat embarqué qui se trouvera dans ses dépendances ;
- en [10], on passe à la suite de l'assistant ;
- en [11], on indique le dossier du projet ;

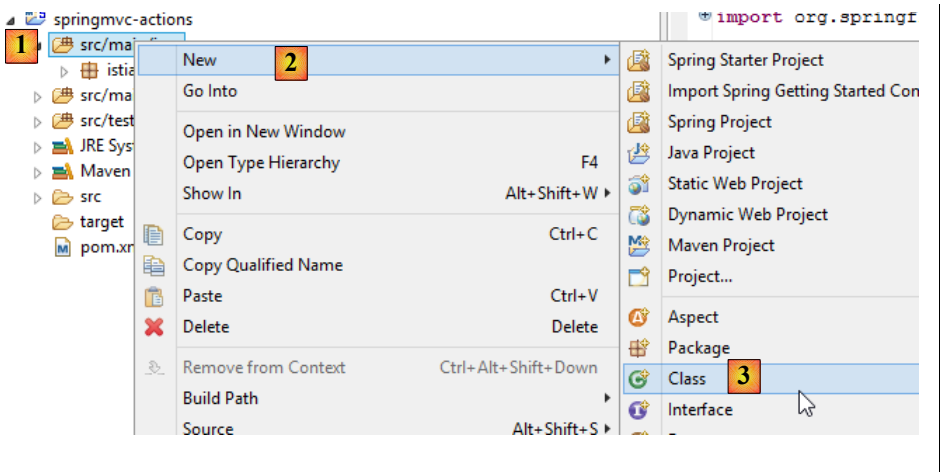


- en [12], le projet généré ;
- en [14-15], on renomme le package [istia.st.springmvc] ;

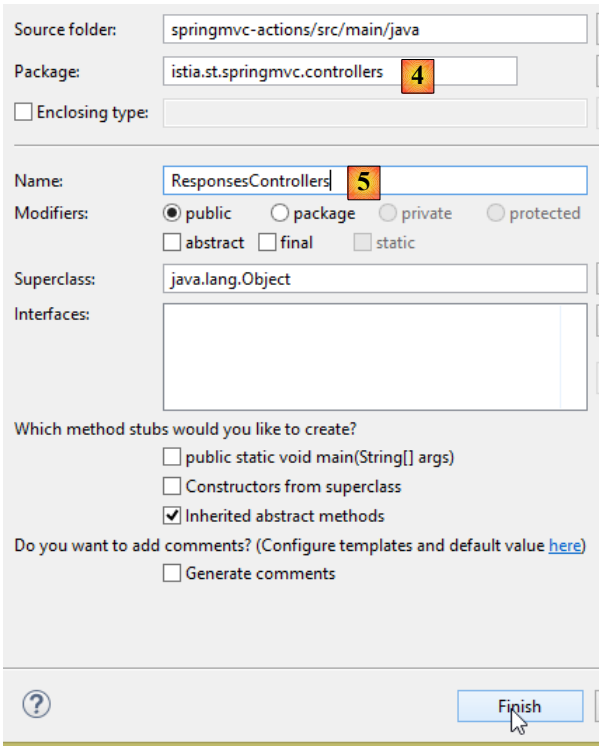


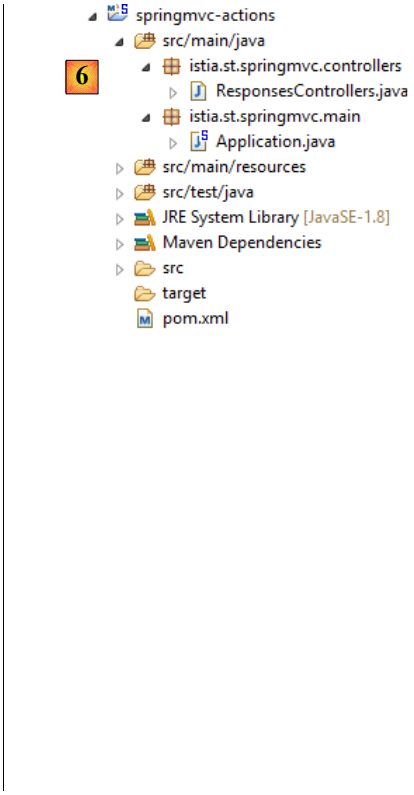
- en [16], le nouveau nom du package ;
- en [17], le nouveau projet ;

Nous créons maintenant une nouvelle classe ;



- en [1-3], nous créons une nouvelle classe ;





- en [5] nous lui donnons et en [4] nous précisons son package ;
- en [6] le nouveau projet ;

La classe est pour l'instant la suivante :

```
1.  package istia.st.springmvc;
2.
3.  public class ActionsController {
4.
5.  }
```

Nous faisons évoluer ce code de la façon suivante :

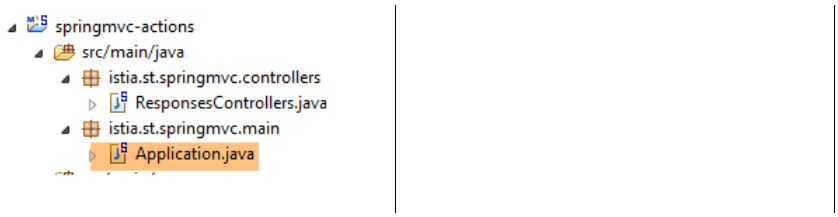
```
1.  package istia.st.springmvc;
2.
3.  import org.springframework.web.bind.annotation.RestController;
```

```
4.
5.  @RestController
6.  public class ActionsController {
7.
8.  }
```

- ligne 6 : l'annotation [@RestController] indique deux choses :
  - que la classe [ActionsController] ainsi annotée est un contrôleur Spring MVC, donc contient des actions qui traitent des URL de clients ;
  - que le résultat de ces actions est envoyé au client ;

L'autre annotation [@Controller] que nous avons rencontrée est différente : les actions d'un contrôleur ainsi annoté rendent le nom de la vue qui doit être affichée. C'est alors la combinaison de cette vue et du modèle construit par l'action pour cette vue qui fournit la réponse envoyée au client.

Le changement de structure de notre projet entraîne un changement de configuration de notre projet :



La classe [Application] évolue de la façon suivante :

```
1.  package istia.st.springmvc.main;
2.
3.  import org.springframework.boot.SpringApplication;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.  import org.springframework.context.annotation.ComponentScan;
6.  import org.springframework.context.annotation.Configuration;
7.
8.  @Configuration
9.  @ComponentScan({"istia.st.springmvc.controllers"})
10. @EnableAutoConfiguration
11. public class Application {
12.
13.     public static void main(String[] args) {
14.         SpringApplication.run(Application.class, args);
15.     }
16. }
```

- ligne 9 : l'annotation [ComponentScan] admet comme paramètre un tableau de noms de packages où Spring Boot doit chercher des composants Spring. Ici nous mettons dans ce tableau le package [istia.st.springmvc.controllers] afin que le contrôleur annoté par [@RestController] soit trouvé ;

Nous allons construire diverses actions dans le contrôleur pour illustrer leurs principales caractéristiques. Nous allons tout d'abord nous intéresser aux divers types de réponses possibles d'une action dans une application sans vues.

3.2 [/a01, /a02] - Hello world

Notre première action sera la suivante :

```
1.  @RestController
2.  public class ActionsController {
3.      // ---------------------- hello world ------------------------
4.      @RequestMapping(value = "/a01", method = RequestMethod.GET)
5.      public String a01() {
6.          return "Greetings from Spring Boot!";
7.      }
8.  }
```

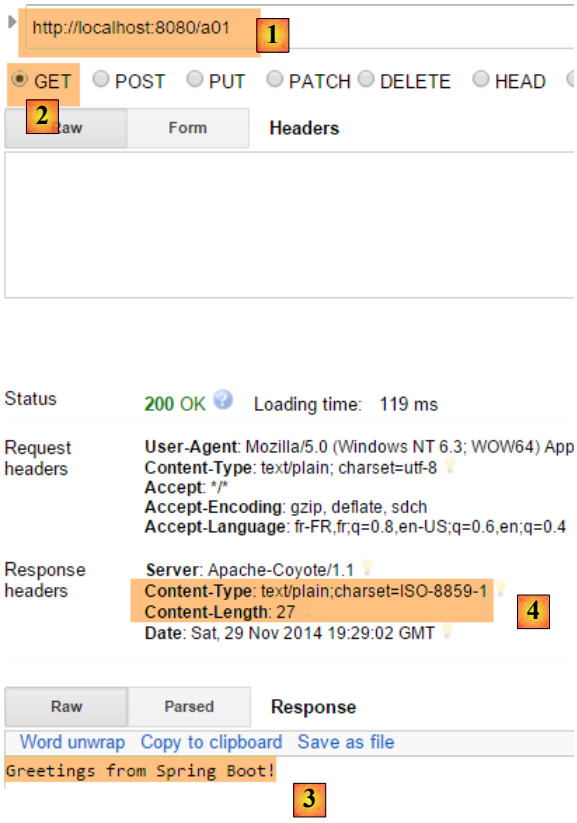
- ligne 4 : l'annotation [RequestMapping] qualifie la requête traitée par l'action annotée :
  - l'attribut [value] est l'URL traitée,

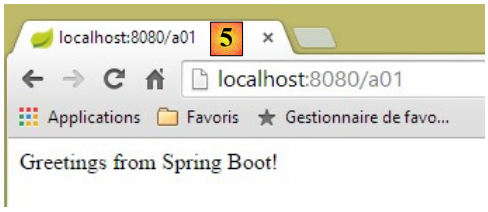
- l'attribut [method] fixe la méthode acceptée ;

Ainsi la méthode [a01] traite la requête HTTP [GET /a01].

- ligne 5 : la méthode [a01] rend un type [String] qui sera envoyé tel quel au client ;
- ligne 6 : la chaîne retournée ;

Lançons l'application comme nous l'avons fait déjà plusieurs fois puis avec le client [Advanced Rest Client], nous demandons l'URL [/a01] avec un GET [1-2] :





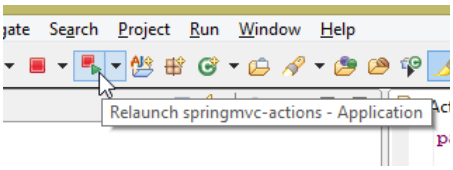
- en [3], la réponse du serveur ;
- en [4], les entêtes HTTP de la réponse. On voit que l'encodage utilisé est [ISO-8859-1]. On peut préférer l'encodage UTF-8. Cela peut se configurer ;
- en [5], on demande la même URL avec le navigateur Chrome ;

Nous ajoutons l'action [/a02] suivante dans le contrôleur [ActionsController] (on confondra ainsi parfois l'URL et la méthode qui la traite sous le nom d'action) :

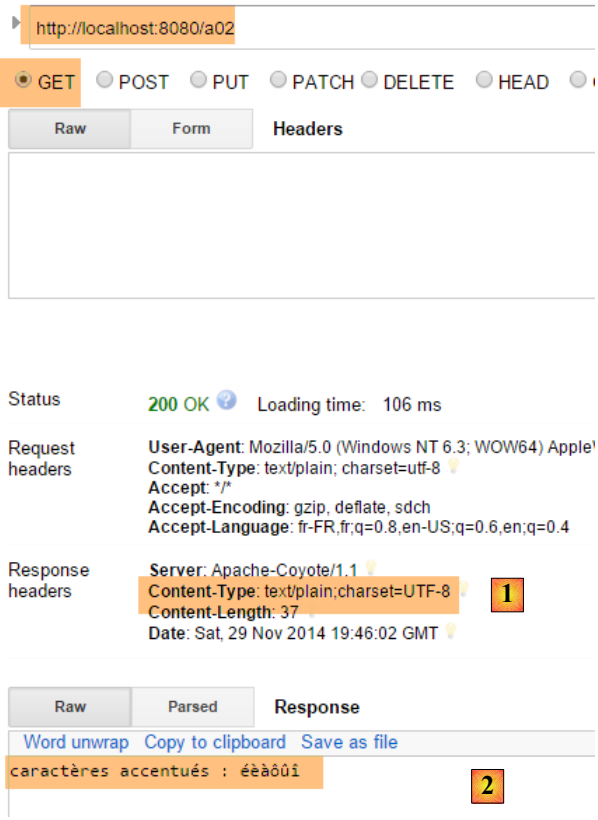
```
1.      // ---------------------- caractères accentués - UTF8 -----------------------
2.      @RequestMapping(value = "/a02", method = RequestMethod.GET, produces="text/plain;charset=UTF-8")
3.      public String a02() {
4.          return "caractères accentués : éèàôûî";
5.  }
```

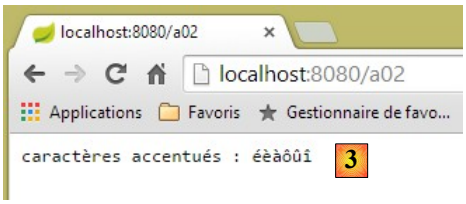
- ligne 2 : l'attribut [produces="text/plain;charset=UTF-8"] indique que l'action envoie un flux texte avec des caractères encodés au format [UTF-8]. Ce format permet notamment l'utilisation des caractères accentués ;

Pour prendre en compte cette nouvelle action, nous devons relancer l'application :



Le résultat est le suivant :





- en [1], on voit la nature du document envoyé par le serveur ;
- en [2-3], on a bien les caractères accentués ;

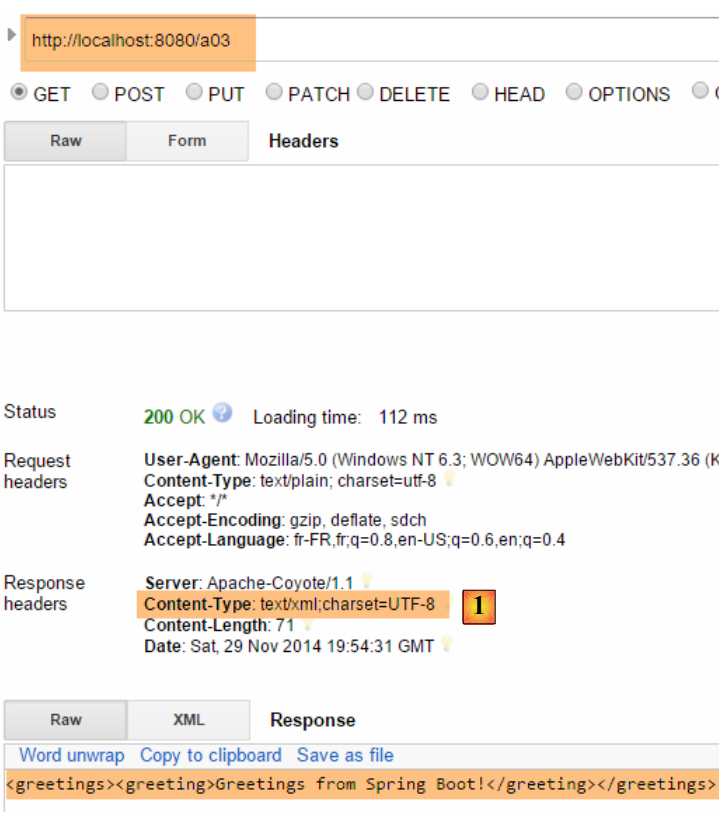
3.3 [/a03] : rendre un flux XML

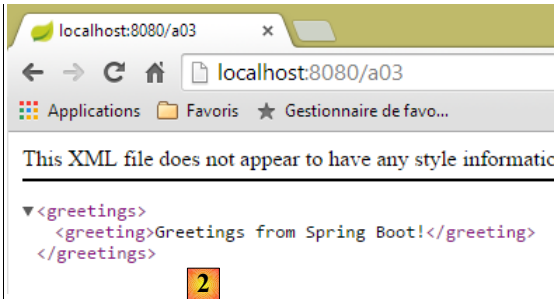
Nous ajoutons l'action [/a03] suivante :

```
1.     // ----------------------- text/xml ------------------------
2.     @RequestMapping(value = "/a03", method = RequestMethod.GET, produces = "text/xml;charset=UTF-8")
3.     public String a03() {
4.         String greeting = "<greetings><greeting>Greetings from Spring Boot!</greeting></greetings>";
5.         return greeting;
6.  }
```

- ligne 2 : l'attribut [produces="text/xml;charset=UTF-8"] indique que l'action envoie un flux XML avec des caractères encodés au format [UTF-8] ;

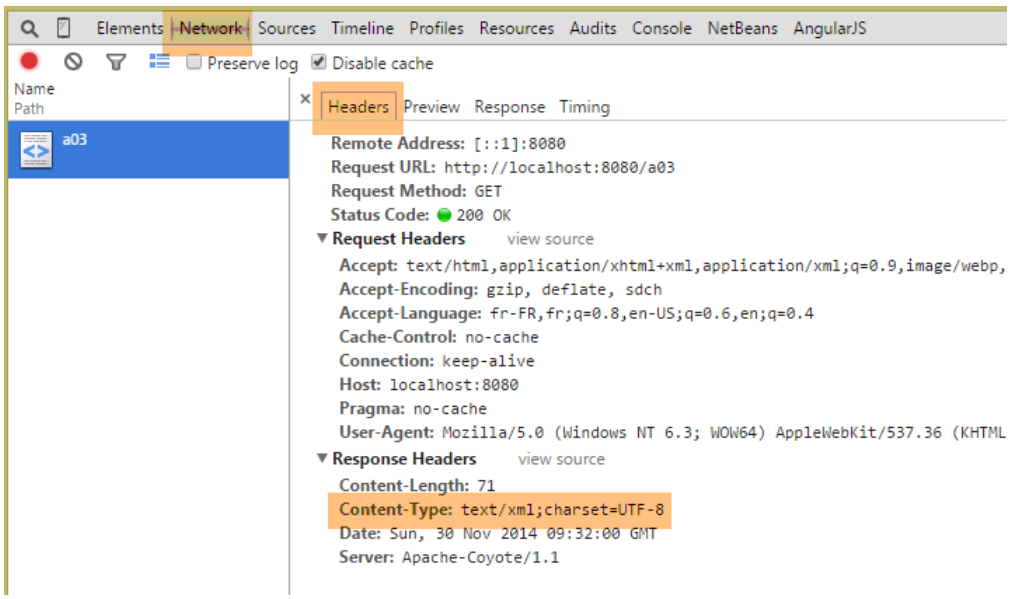
Son exécution donne la chose suivante :





- en [1], l'entête HTTP précise que le document envoyé est du HTML ;
- en [2], le navigateur Chrome utilise cette information pour formater le texte XML reçu ;

Rappelons qu'avec Chrome, on a accès aux échanges HTTP entre le client et le serveur dans la fenêtre de développement (Ctrl-Maj-I) :



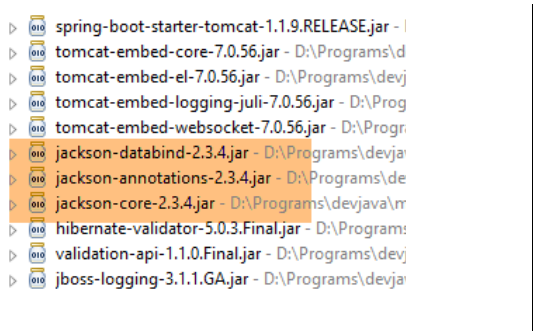
Dorénavant, on ne fera pas systématiquement des copies d'écran des échanges HTTP entre le client et le serveur. Parfois, on se contentera d'indiquer le texte de ces échanges.

3.4 [/a04, /a05] : rendre un flux jSON

Nous ajoutons l'action [/a04] suivante :

```
1.      // ---------------------- produire du jSON -----------------------
2.      @RequestMapping(value = "/a04", method = RequestMethod.GET)
3.      public Map<String, Object> a04() {
4.          Map<String, Object> map = new HashMap<String, Object>();
5.          map.put("1", "un");
6.          map.put("2", new int[] { 4, 5 });
7.          return map;
8.  }
```

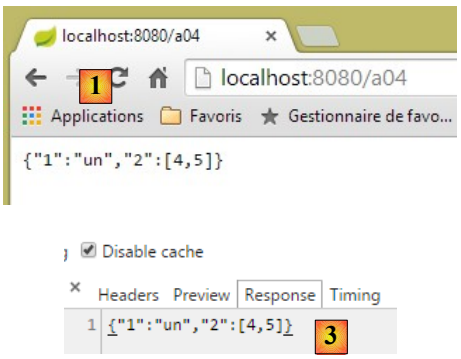
- ligne 3 : l'action rend un type [Map], un dictionnaire. On se rappelle qu'avec un contrôleur de type [@RestController], le résultat de l'action est la réponse envoyée au client. Le protocole HTTP étant un protocole d'échanges de lignes de texte, la réponse du client doit être sérialisée en une chaîne de caractères. Pour cela, Spring MVC utilise divers convertisseurs [Objet <---> chaîne de caractères]. L'association d'un objet particulier avec un convertisseur se fait par configuration. Ici l'autoconfiguration de Spring Boot va inspecter les dépendances du projet :

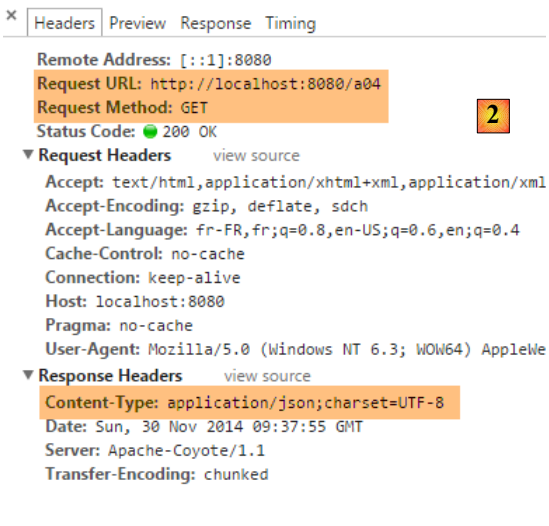


Les dépendances Jackson ci-dessus sont des bibliothèques de sérialisation / désérialisation d'objets en chaînes jSON. Spring Boot va alors utiliser ces bibliothèques pour sérialiser / désérialiser les objets rendus par les actions. On trouvera un exemple de code Java pour sérialiser / désérialiser des objets Java en jSON au paragraphe 9.7, page 607.

On notera en ligne 2 que nous n'avons pas mis le type de la réponse envoyée. Nous allons voir le type par défaut qui va être envoyé.

Les résultats sont les suivants dans Chrome [1-3] :

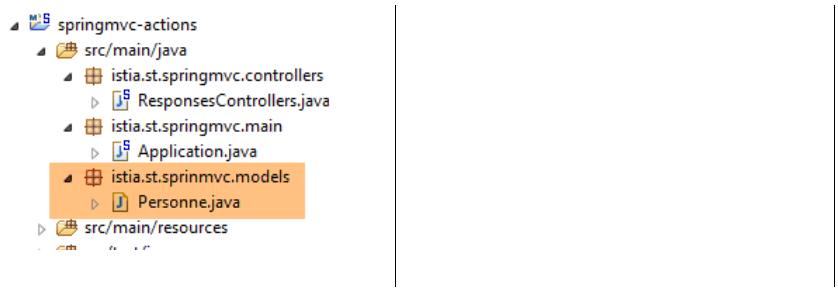




Nos ajoutons maintenant l'action [/a05] suivante :

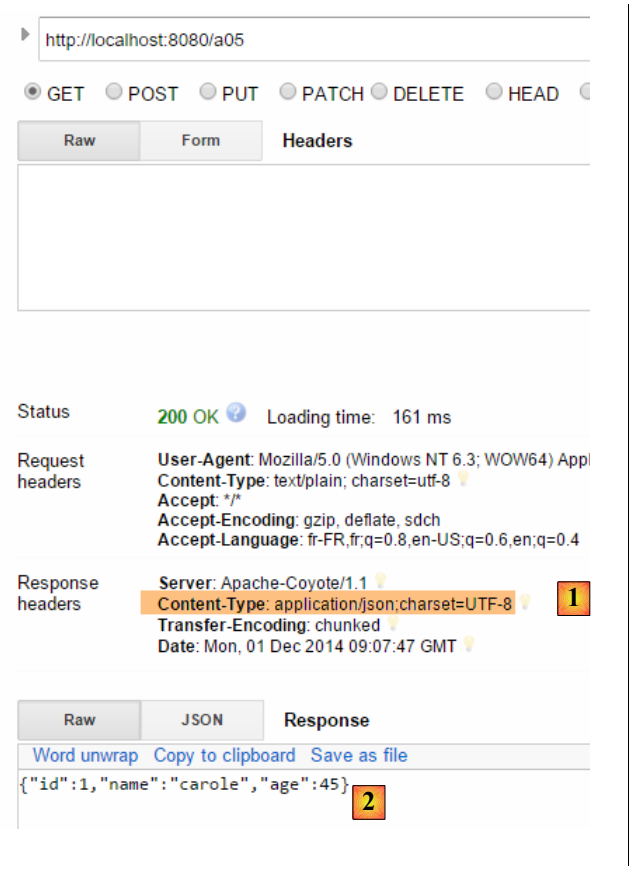
```
    // ---------------------- produire du jSON - 2 -----------------------
    @RequestMapping(value = "/a05", method = RequestMethod.GET)
    public Personne a05() {
       return new Personne(1,"carole",45);
    }
```

La classe [Personne] est la suivante :




```
package istia.st.sprinmvc.models;

public class Personne {

    // identifiant
    private Integer id;
    // nom
    private String nom;
    // âge
    private int age;

    // constructeurs
    public Personne() {

    }

    public Personne(String nom, int age) {
       this.nom = nom;
       this.age = age;
    }

    public Personne(Integer id, String nom, int age) {
       this(nom, age);
       this.id = id;
    }

    @Override
    public String toString() {
       return String.format("[id=%s, nom=%s,  age=%d]", id, nom, age);
    }

    // getters et setters
...
}
```

L'exécution donne les résultats suivants :



- en [1], le serveur indique que le document qu'il envoie est du jSON ;
- en [2], le document jSON reçu ;

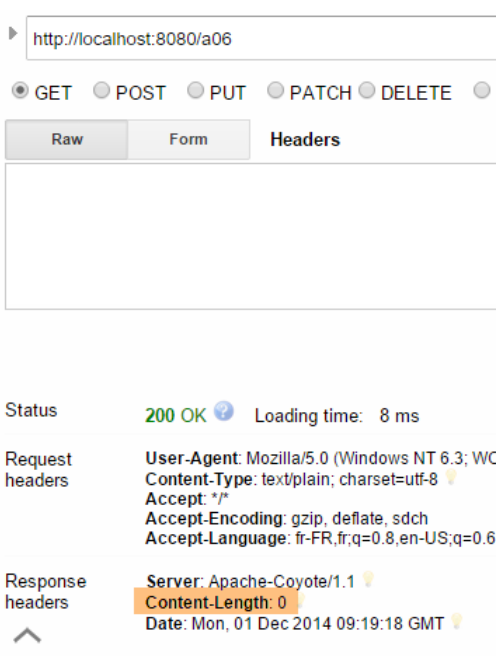
3.5 [/a06] : rendre un flux vide

Nous ajoutons l'action [/a06] suivante :

```
1.      // ---------------------- rendre un flux vide -----------------------
2.      @RequestMapping(value = "/a06")
3.      public void a06() {
4.  }
```

- ligne 3, l'action [/a06] ne rend rien. Spring MVC va alors générer une réponse vide au client ;

L'exécution donne les résultats suivants :



Ci-dessus, l'attribut HTTP [Content-Length] dans la réponse indique que le serveur envoie un document vide.

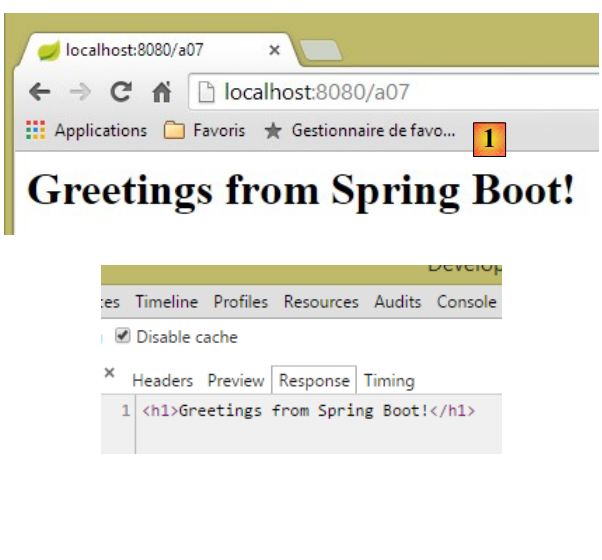
3.6 [/a07, /a08, /a09] : nature du flux avec [Content-Type]

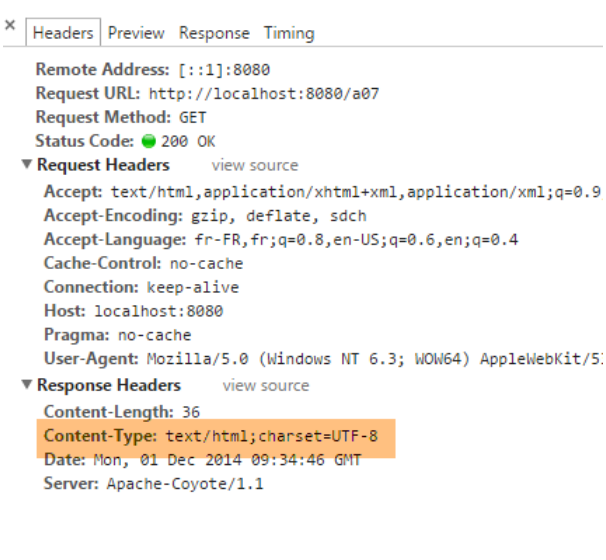
Nous ajoutons l'action [/a07] suivante :

```
1.     // ---------------------- text/html -----------------------
2.     @RequestMapping(value = "/a07", method = RequestMethod.GET, produces = "text/html;charset=UTF-8")
3.     public String a07() {
4.        String greeting = "<h1>Greetings from Spring Boot!</h1>";
5.        return greeting;
6.  }
```

- ligne 2, l'action [/a07] rend un flux HTML [text/html] ;
- ligne 4 : une chaîne HTML ;

L'exécution donne les résultats suivants :





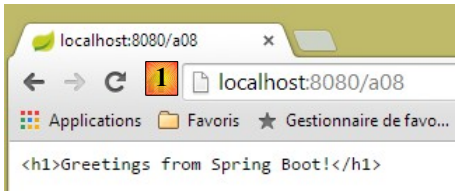
- en [1], on voit que Chrome a interprété la balise HTML <h1> qui affiche en gros caractères son contenu ;

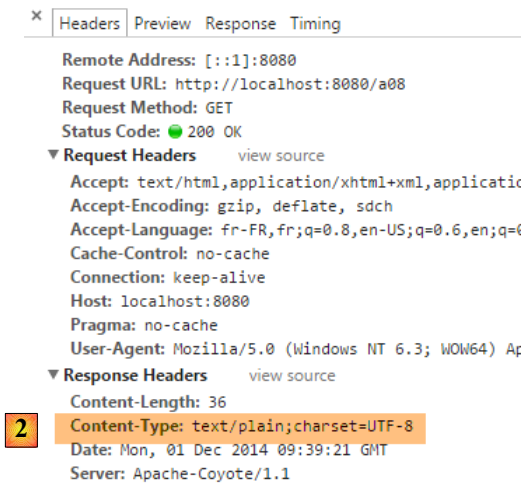
Maintenant faisons la même chose avec l'action [/a08] suivante :

```
1.      // ----------------------- résultat HTML en text/plain -----------------------
2.      @RequestMapping(value = "/a08", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
3.      public String a08() {
4.          String greeting = "<h1>Greetings from Spring Boot!</h1>";
5.          return greeting;
6.  }
```

- ligne 2 : la réponse de l'action est de type [text/plain] ;

Les résultats sont les suivants :





- en [1], Chrome n'a pas interprété la balise HTML <h1> parce que le serveur lui a dit qu'il lui envoyait un flux [text/plain] [2] ;

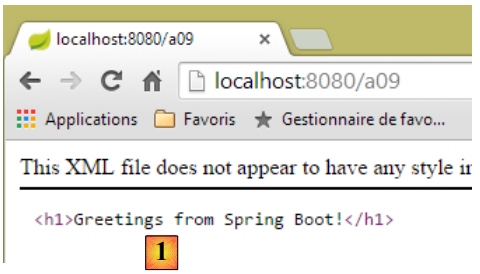
Recommençons quelque chose d'analogue avec l'action [/a09] suivante :

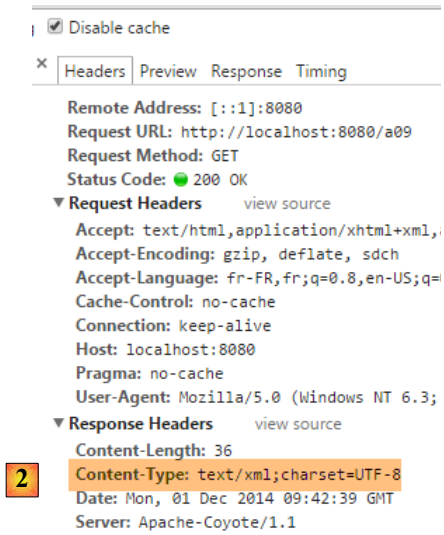
```
1.      // ----------------------- résultat HTML en text/xml -----------------------
```

```
2.     @RequestMapping(value = "/a09", method = RequestMethod.GET, produces = "text/xml;charset=UTF-8")
3.     public String a09() {
4.         String greeting = "<h1>Greetings from Spring Boot!</h1>";
5.         return greeting;
6.  }
```

- ligne 2 : on envoie un flux de type [text/xml] ;

Les résultats sont les suivants :



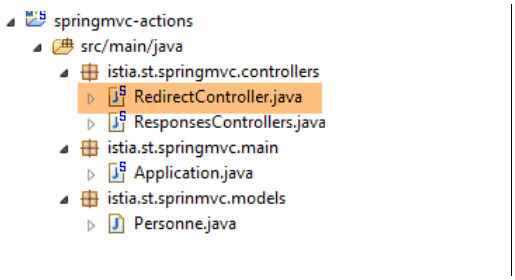


- en [1], Chrome n'a pas interprété la balise HTML <h1> parce que le serveur lui a dit qu'il lui envoyait un flux [text/xml] [2]. Il a alors géré la balise <h1> comme une balise XML ;

On retiendra de ces exemples l'importance de l'entête HTTP [Content-Type] dans la réponse du serveur. Le navigateur utilise cet entête pour savoir comment interpréter le document qu'il reçoit ;

3.7 [/a10, /a11, /a12] : rediriger le client

Nous créons un nouveau contrôleur [RedirectController] :



Le code de [RedirectCntroller] sera pour l'instant le suivant :

```
1.  package istia.st.springmvc.controllers;
2.
3.  import org.springframework.stereotype.Controller;
4.  import org.springframework.web.bind.annotation.RequestMapping;
5.  import org.springframework.web.bind.annotation.RequestMethod;
6.
7.  @Controller
8.  public class RedirectController {
9.  }
```

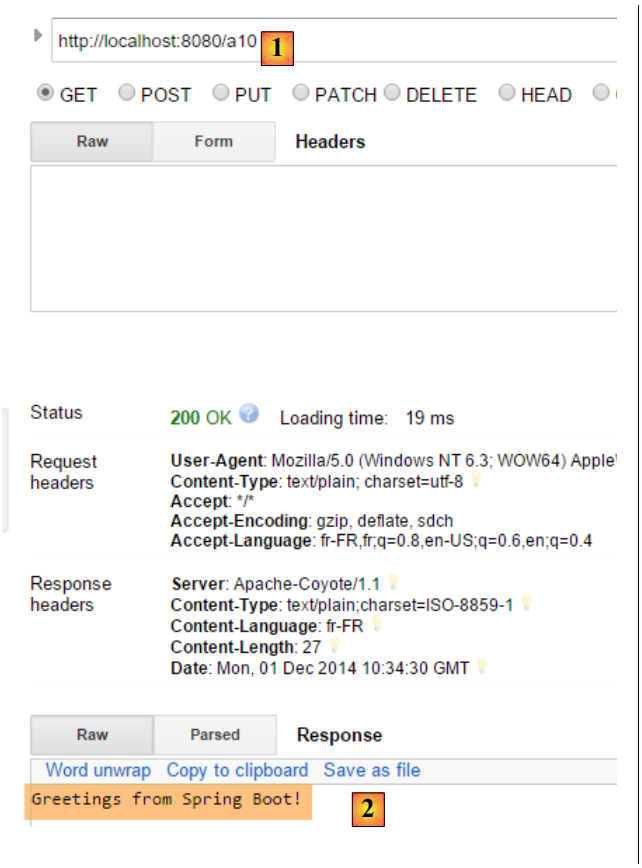
- ligne 7 : on utilise l'annotation [@Controller] ce qui fait que désormais par défaut le type [String] du résultat des actions désigne le nom d'une action ou d'une vue ;

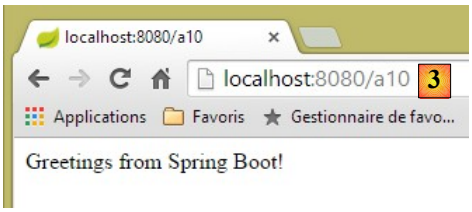
Nous créons l'action [/a10] suivante :

```
1.      // ------------ pont vers une action tierce ----------------------
2.      @RequestMapping(value = "/a10", method = RequestMethod.GET)
3.      public String a10() {
4.          return "a01";
5.  }
```

- ligne 4 : on rend comme résultat 'a01' qui est le nom d'une action. Ce sera alors elle qui va envoyer la réponse au client ;

Voici un exemple :



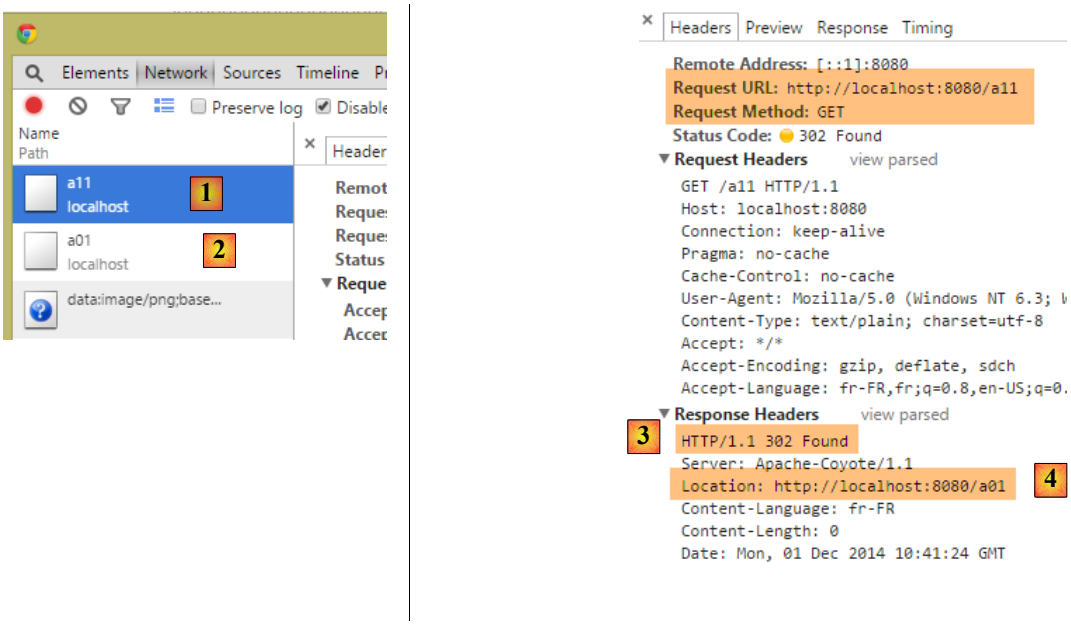


- en [2], on a reçu le flux de l'action [/a01] ;
- en [3], le navigateur affiche l'URL de l'action [/a10] ;

Nous créons maintenant l'action [/a11] suivante :

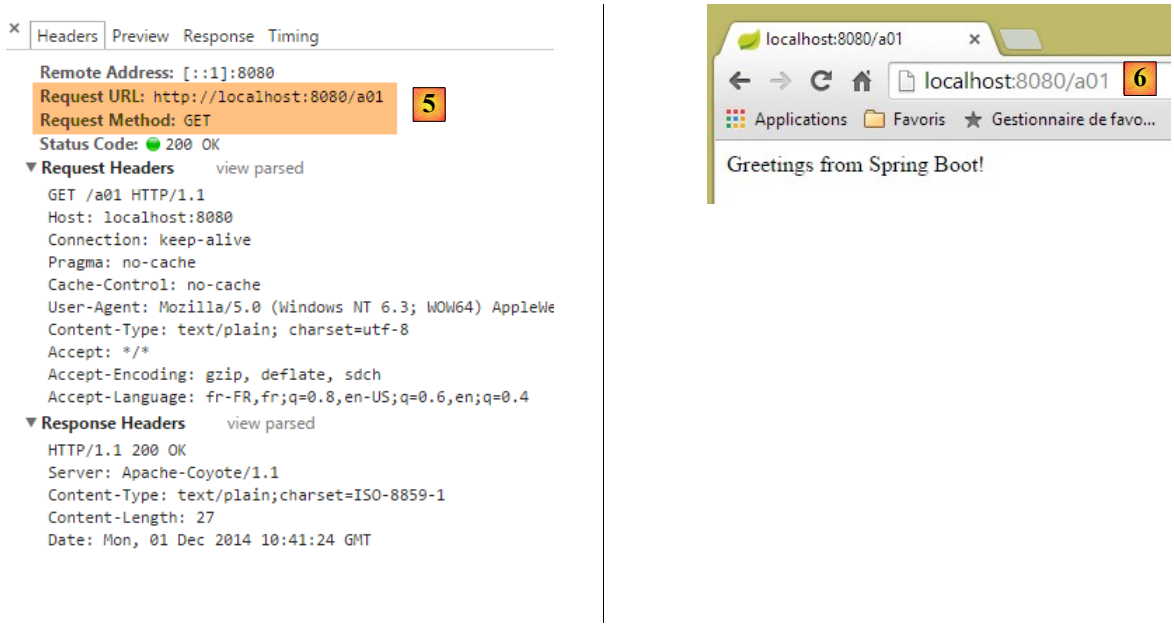
```
1.      // ------------ redirection temporaire 302 vers une action tierce ----------------------
2.      @RequestMapping(value = "/a11", method = RequestMethod.GET)
3.      public String a11() {
4.          return "redirect:/a01";
5.  }
```

Nous obtenons les résultats suivants :



- dans les logs de Chrome [1-2], on voit deux requêtes, l'une vers [/a11], l'autre vers [/a01] ;
- en [3], le serveur répond avec un code [302] qui demande au navigateur client de se rediriger vers l'URL indiquée par l'entête HTTP [Location:] [4]. Le code [302] est un code de redirection temporaire ;

Le navigateur fait alors la deuxième requête vers l'URL de redirection :



- en [5], la seconde requête du client ;
- en [6], le navigateur client affiche l'URL de la requête de direction ;

On peut vouloir indiquer une reirection permanente, auquel cas, il faut envoyer au client l'entête HTTP suivant :

```
HTTP/1.1 301 Moved Permanently
```

qui veut dire que la redirection est permanente. Cette différence entre redirection temporaire (302) et permanente (301) est prise en compte par certains moteurs de recherche.

Nous écrivons l'action [/a12] qui va opérer cette redirection permanente :

```
1.      // ------------ redirection permanente 301 vers une action tierce----------------
2.      @RequestMapping(value = "/a12", method = RequestMethod.GET)
3.      public void a12(HttpServletResponse response) {
4.         response.setStatus(301);
5.         response.addHeader("Location", "/a01");
6.  }
```

- ligne 3 : on demande à Spring MVC d'injecter l'objet [HttpServletResponse] qui encapsule la réponse envoyée au client ;
- ligne 4 : on fixe le [status] de la réponse, le [301] de l'entête HTTP :

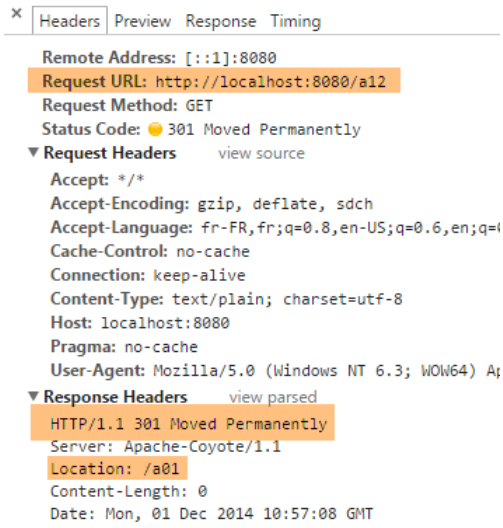
```
HTTP/1.1 301 Moved Permanently
```

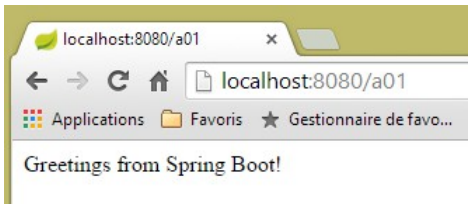
- ligne 5 : on crée à la main l'entête HTTP suivant :

```
Location: /a01
```

qui est l'URL de redirection.

L'exécution donne les résultats suivants :



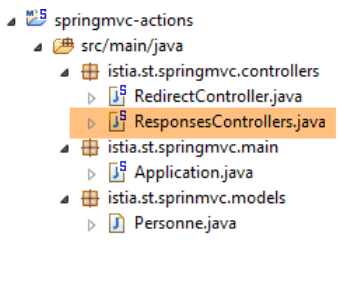


On retiendra de cet exemple la façon de :

- générer le statut de la réponse HTTP ;
- d'inclure un entête HTTP dans la réponse ;

3.8 [/a13] : générer la réponse complète

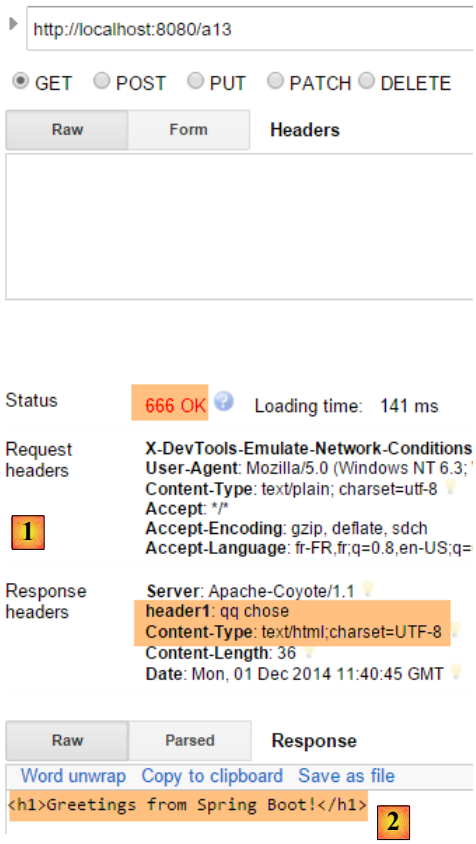
Il est possible de maîtriser totalement la réponse comme le montre l'action suivante de la classe [ResponsesController] :



```
1.      // ---------------------- génération complète de la réponse -----------------------
2.      @RequestMapping(value = "/a13")
3.      public void a13(HttpServletResponse response) throws IOException {
4.          response.setStatus(666);
5.          response.addHeader("header1", "qq chose");
6.          response.addHeader("Content-Type", "text/html;charset=UTF-8");
7.          String greeting = "<h1>Greetings from Spring Boot!</h1>";
8.          response.getWriter().write(greeting);
9.  }
```

- ligne 3 : le résultat de l'action est [void]. Dans ce cas, pour envoyer une réponse non vide au client, il faut utiliser l'objet [HttpServletResponse response] fourni par Spring MVC ;
- ligne 4 : on donne à la réponse un statut qui sera non reconnu par le client ;
- ligne 5 : on ajoute un entête HTTP qui sera non reconnu par le client ;
- ligne 6 : on ajoute un entête HTTP [Content-Type] pour préciser le type de flux qu'on va envoyer, ici du HTML ;
- lignes 7-8 : le document qui va suivre les entêtes HTTP dans la réponse ;

Les résultats sont les suivants :

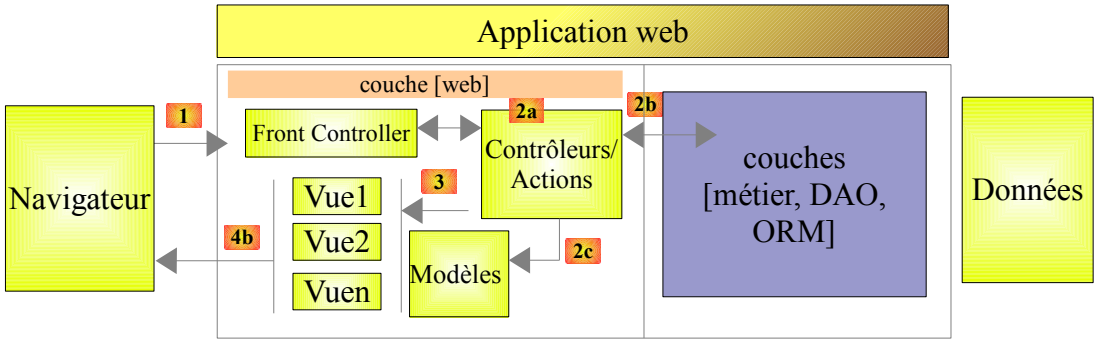


- en [1], on reconnaît les éléments de notre réponse ;
- en [2-3], on voit que Chrome a ignoré le fait que :
  - le statut HTTP de la réponse n'était pas un statut HTTP reconnu,
  - que l'entête [header1] n'était pas un entête HTTP reconnu ;

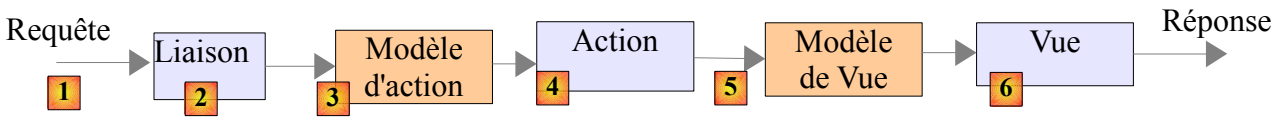
Si le client n'est pas un navigateur mais un client programmé, on est libre d'utiliser les statuts et les entêtes que l'on veut.

4 Actions : le modèle

Revenons à l'architecture d'une application Spring MVC :



Dans le chapitre précédent, nous avons regardé le processus qui amène la requête [1] au contrôleur et à l'action [2a] qui vont la traiter, un mécanisme qu'on appelle le routage. Nous avons présenté par ailleurs les différentes réponses que peut faire une action au navigateur. Nous avons pour l'instant présenté des actions qui n'exploitaient pas la requête qui leur était présentée. Une requête [1] transporte avec elle diverses informations que Spring MVC présente [2a] à l'action sous forme d'un **modèle**. On ne confondra pas ce terme avec le modèle **M** d'une vue **V** [2c] qui est produit par l'action :



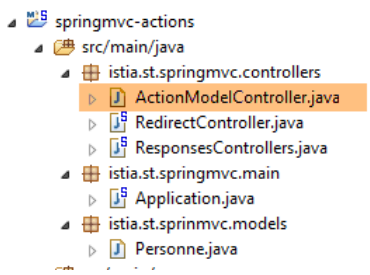
- la requête HTTP du client arrive en [1] ;
- en [2], les informations contenues dans la requête vont être transformées en modèle d'action [3], une classe souvent mais pas forcément, qui servira d'entrée à l'action [4] ;
- en [4], l'action, à partir de ce modèle, va générer une réponse. Celle-ci aura deux composantes : une vue V [6] et le modèle M de cette vue [5] ;
- la vue V [6] va utiliser son modèle M [5] pour générer la réponse HTTP destinée au client.

Dans le modèle **MVC**, l'action [4] fait partie du **C** (contrôleur), le modèle de la vue [5] est le **M** et la vue [6] est le **V**.

Ce chapitre étudie les mécanismes de liaison entre les informations transportées par la requête, qui sont par nature des chaînes de caractères et le modèle de l'action qui peut être une classe avec des propriétés de divers types.

Note : le terme [Modèle d'action] n'est pas un terme reconnu.

Nous créons un nouveau contrôleur pour ces nouvelles actions :



Le contrôleur [ActionModelController] sera pour l'instant le suivant :

```
1.  package istia.st.springmvc.controllers;
2.
3.  import org.springframework.web.bind.annotation.RestController;
4.
```

```
5.  @RestController
6.  public class ActionModelController {
7.
8.  }
```

- ligne 5 : on rappelle que l'annotation [@RestController] fait que la réponse envoyée au client est la sérialisation en chaîne de caractères du résultat des actions du contrôleur ;

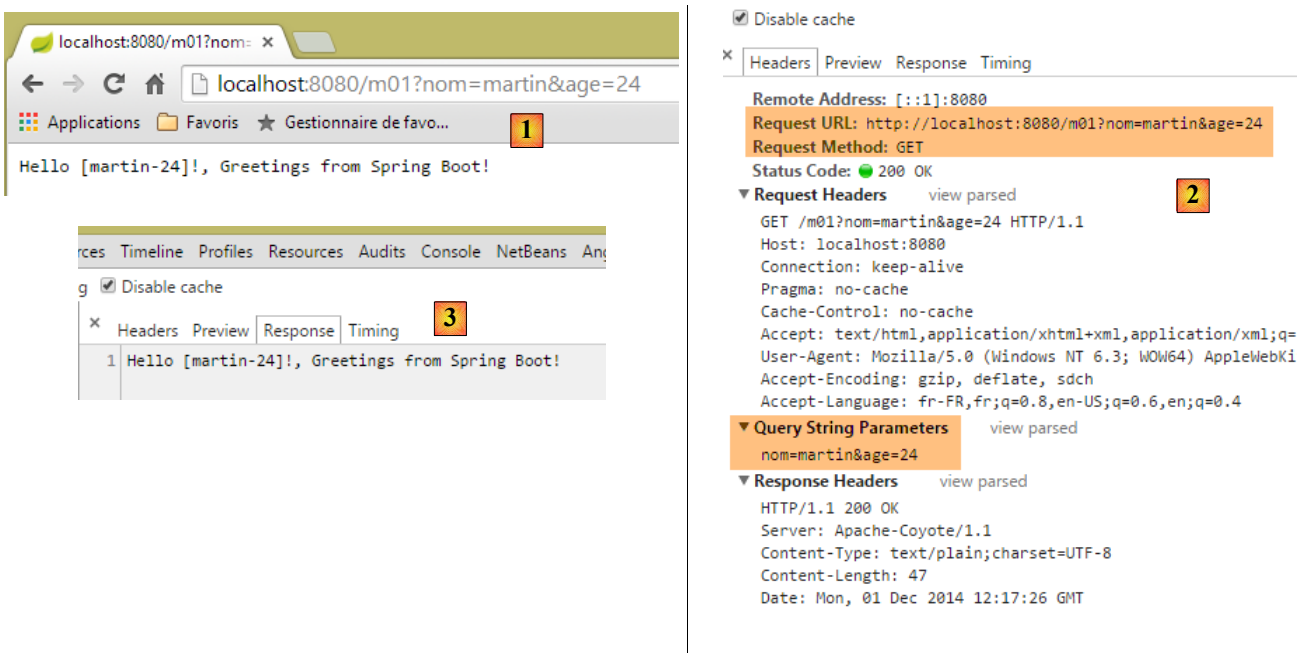
4.1 [/m01] : paramètres d'un GET

Nous ajoutons l'action [/m01] suivante :

```
1.
2.      // ---------------------- récupérer des paramètre avec GET-----------------------
3.      @RequestMapping(value = "/m01", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
4.      public String m01(String nom, String age) {
5.          return String.format("Hello [%s-%s]!, Greetings from Spring Boot!", nom, age);
6.  }
```

- ligne 4 : l'action admet deux paramètres nommé [nom] et [age]. Ils seront initialisés avec des paramètres portant ces mêmes noms dans la requête HTTP GET ;

Les résultats sont les suivants dans Chrome [1-3] :



- en [1], la requête GET avec les paramètres [nom] et [age] ;
- en [3], on voit que l'action [/m01] a bien récupéré ces paramètres ;

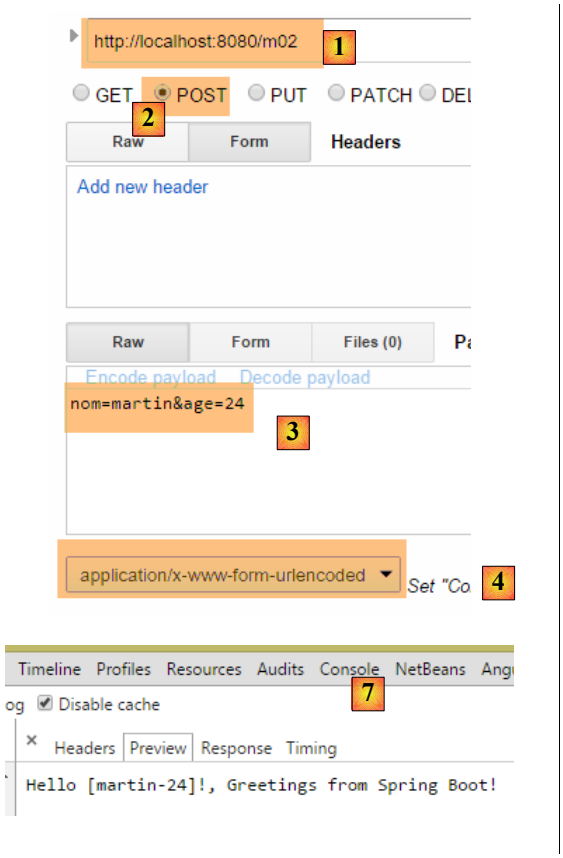
4.2 [/m02] : paramètres d'un POST

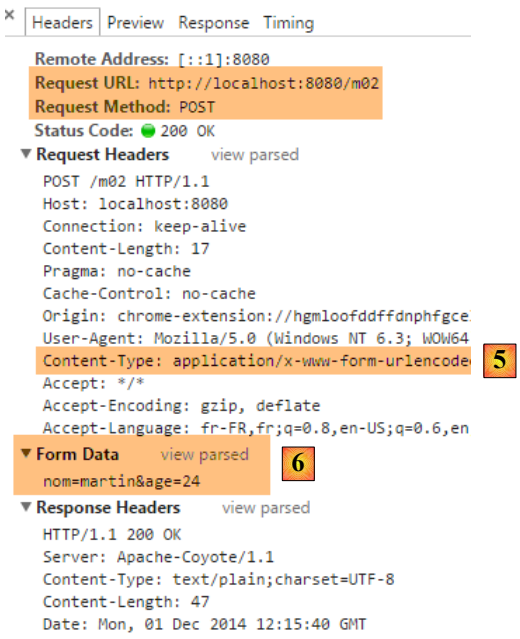
Nous ajoutons l'action [/m02] suivante :

```
1.
2.      // ---------------------- récupérer des paramètre avec POST-----------------------
3.      @RequestMapping(value = "/m02", method = RequestMethod.POST, produces = "text/plain;charset=UTF-8")
4.      public String m02(String nom, String age) {
5.          return String.format("Hello [%s-%s]!, Greetings from Spring Boot!", nom, age);
6.  }
```

- ligne 4 : l'action admet deux paramètres nommé [nom] et [age]. Ils seront initialisés avec des paramètres portant ces mêmes noms dans la requête HTTP POST ;

Les résultats avec [Advanced rest Client] sont les suivants :





- en [1-3], la requête POST avec les paramètres [nom] et [age] ;
- en [4-5], on fixe l'entête HTTP [Content-Type] de la requête POST. Il doit être [Content-Type: application/x-www-form-urlencoded] ;
- en [6], [Form Data] donne la liste des paramètres d'une opération POST. Ici on voit les paramètres [nom] et [age] ;
- en [7], la réponse du serveur qui montre que l'action [/m02] a bien récupéré les paramètres [nom] et [age] ; ;

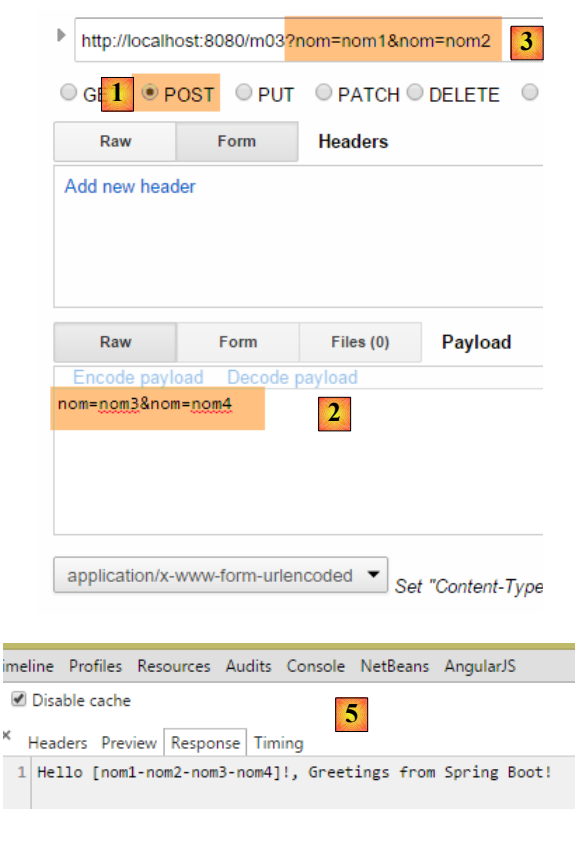
4.3 [/m03] : paramètres de mêmes noms

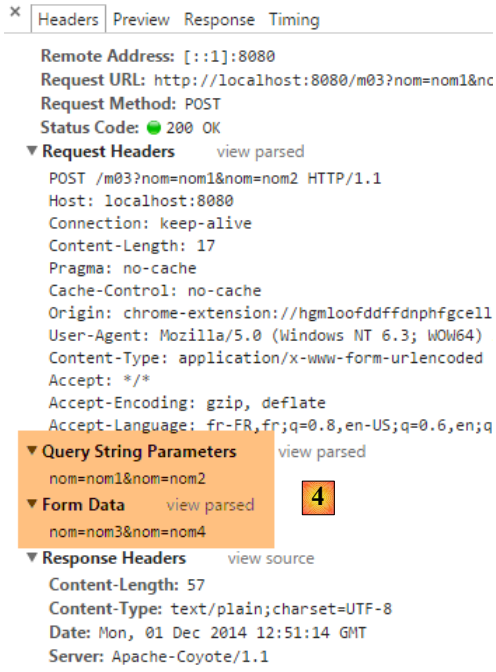
Nous avons vu au paragraphe 2.5.2.8, page 70, que la liste à sélection multiple pouvait envoyer au serveur des paramètres de mêmes noms. Voyons comment une action peut les récupérer. Nous ajoutons l'action [/m03] suivante :

```
1.     // ---------------------- récupérer des paramètres de mêmes noms-----------------
2.     @RequestMapping(value = "/m03", method = RequestMethod.POST, produces = "text/plain;charset=UTF-8")
3.     public String m03(String nom[]) {
4.        return String.format("Hello [%s]!, Greetings from Spring Boot!", String.join("-", nom));
5.  }
```

- ligne 2 : l'action admet un paramètre nommé [**nom[]**]. Il sera initialisé ici avec tous les paramètres portant ce nom que ce soit dans un GET ou un POST, puisqu'ici le type de la requête n'a pas été précisé ;

Les résultats sont les suivants :





- par un POST [1], on envoie les paramètres [2] ;
- on met également des paramètres dans l'URL [3] ;
- en [4], les quatre paramètres portant le même nom [nom] : [Query String parameters] sont les paramètres de l'URL, [Form Data] sont les paramètres postés ;
- en [5], on voit que l'action [/m03] a récupéré les quatre paramètres nommés [nom] ;

4.4 [/m04] : mapper les paramètres de l'action dans un objet Java

Soit la nouvelle action [/m04] suivante :

```
1.     // ------ mapper les paramètres dans un objet (Command Object) ---------------
2.     @RequestMapping(value = "/m04", method = RequestMethod.POST)
3.     public Personne m04(Personne personne) {
4.         return person;
5.  }
```

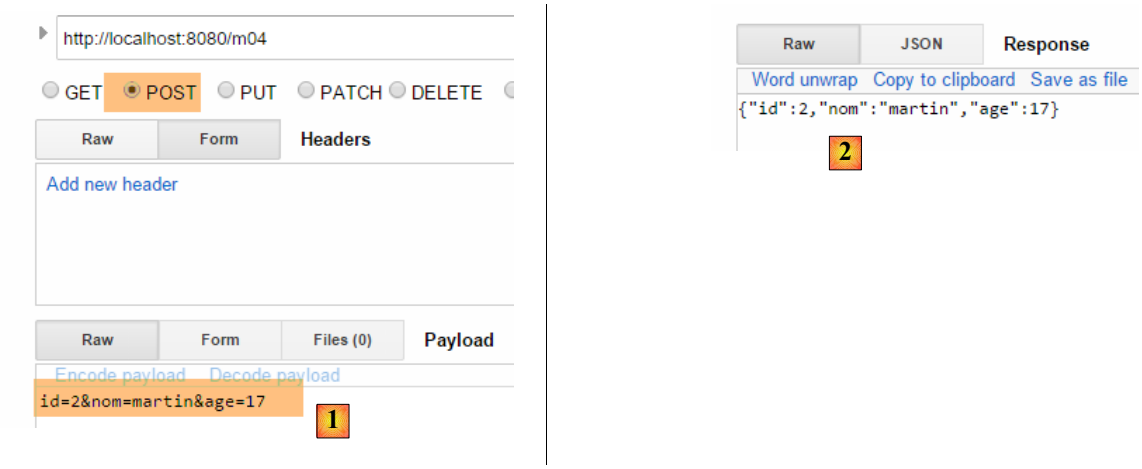
- ligne 3 : l'action a pour paramètre une personne de type suivant :

```
1.  public class Personne {
2.
3.     // identifiant
4.     private Integer id;
5.     // nom
6.     private String nom;
7.     // âge
8.     private int age;
9.  ....
10.    // getters et setters
11. ...
12. }
```

- pour créer le paramètre [Personne personne], Spring MVC fait un [new Personne()] ;
- puis s'il y a des paramètres portant le nom des champs [id, nom, age] de l'objet créé, il instancie avec les champs via leurs setters ;

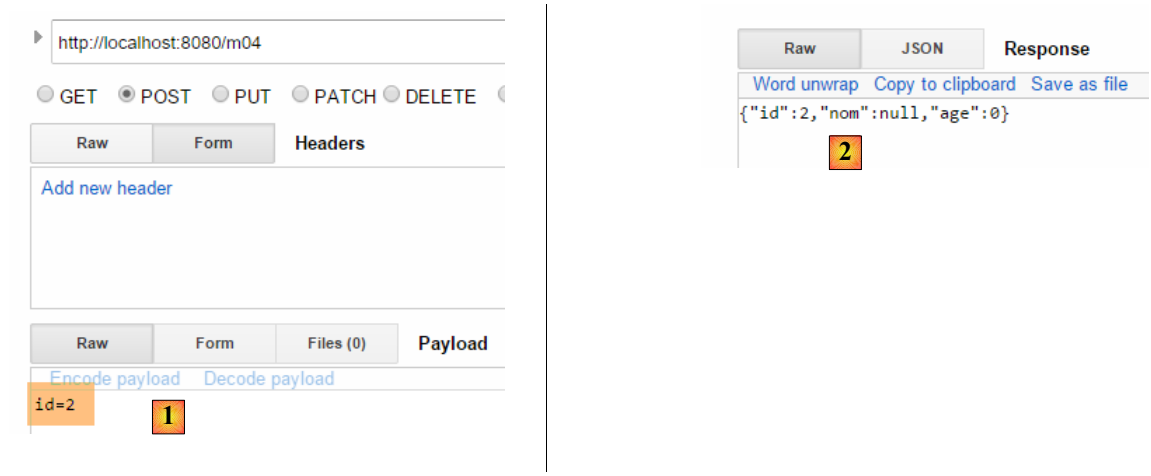
- ligne 4 : l'action rend un type [Personne] qui va donc être sérialisée en chaîne de caractères avant d'être envoyé au client. On a vu que par défaut, la sérialisation effectuée était une sérialisation jSON. Le client devrait donc recevoir la chaîne jSON d'une personne ;

Voici un exemple :



- en [1], les paramètres [id, nom, age] pour construire un objet [Personne] ;
- en [2], la chaîne jSON de cette personne ;

Que se passe-t-il si on n'envoie pas tous les champs d'une personne ? Essayons :



- en [2], seul le paramètre [id] a été initialisé ;

4.5 [/m05] : récupérer les éléments d'une URL

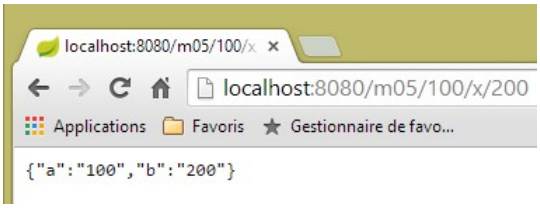
Soit la nouvelle action [/m05] suivante :

```
1.     // ---------------------- récupérer les éléments de l'URL ----------------------
2.     @RequestMapping(value = "/m05/{a}/x/{b}", method = RequestMethod.GET)
3.     public Map<String, String> m05(@PathVariable("a") String a, @PathVariable("b") String b) {
4.         Map<String, String> map = new HashMap<String, String>();
5.         map.put("a", a);
6.         map.put("b", b);
7.         return map;
8.     }
```

- ligne 2 : l'URL traitée est de la forme [/m05/{a}/x/{b}] où {param} est un élément paramètre de l'URL ;
- ligne 3 : les éléments paramètres de l'URL sont récupérés avec l'annotation [@PathVariable] ;

- lignes 4-6 : les éléments [a] et [b] récupérés sont mis dans un dictionnaire ;
- ligne 7 : la réponse sera la chaîne jSON de ce dictionnaire ;

Les résultats sont les suivants :



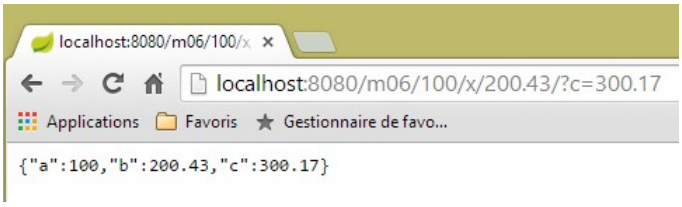
4.6 [/m06] : récupérer des éléments d'URL et des paramètres

Soit la nouvelle action [/m06] suivante :

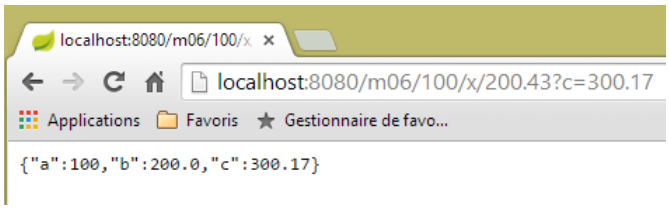
```
1.      // -------- récupérer des éléments de l'URL et des paramètres---------------
2.      @RequestMapping(value = "/m06/{a}/x/{b}", method = RequestMethod.GET)
3.      public Map<String, Object> m06(@PathVariable("a") Integer a, @PathVariable("b") Double b, Double c) {
4.          Map<String, Object> map = new HashMap<String, Object>();
5.          map.put("a", a);
6.          map.put("b", b);
7.          map.put("c", c);
8.          return map;
9.  }
```

- ligne 3 : on récupère à la fois des éléments d'URL [Integer a, Double b] et un paramètre (GET ou POST) [Double c] ;
- lignes 4-7 : ces éléments sont mis dans un dictionnaire ;
- ligne 8 : qui forme la réponse du client qui recevra donc la chaîne jSON de ce dictionnaire ;

Voici les résultats :



On notera le / à la fin du chemin [http://localhost:8080/m06/100/x/200.43/]. Sans lui, on obtient le résultat incorrect suivant :



4.7 [/m07] : accéder à la totalité de la requête

Soit la nouvelle action [/m07] suivante :

```
1.      // ------ accéder à la requête HttpServletRequest -----------------------
2.      @RequestMapping(value = "/m07", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
3.      public String m07(HttpServletRequest request) {
4.         // les entêtes HTTP
5.         Enumeration<String> headerNames = request.getHeaderNames();
6.         StringBuffer buffer = new StringBuffer();
7.         while (headerNames.hasMoreElements()) {
8.            String name = headerNames.nextElement();
9.            buffer.append(String.format("%s : %s\n", name, request.getHeader(name)));
10.        }
11.        return buffer.toString();
12. }
```

- ligne 3 : on demande à Spring MVC d'injecter l'objet [HttpServletRequest request] qui encapsule la totalité des informations qu'on peut obtenir sur la requête ;
- lignes 5-10 : on récupère tous les entêtes HTTP de la requête pour les assembler dans une chaîne de caractères qu'on envoie au client (ligne 11) ;

Les résultats sont les suivants :

- en [1], les entêtes HTTP de la requête ;

- en [2], la réponse. On y retrouve bien tous les entêtes HTTP de la requête.

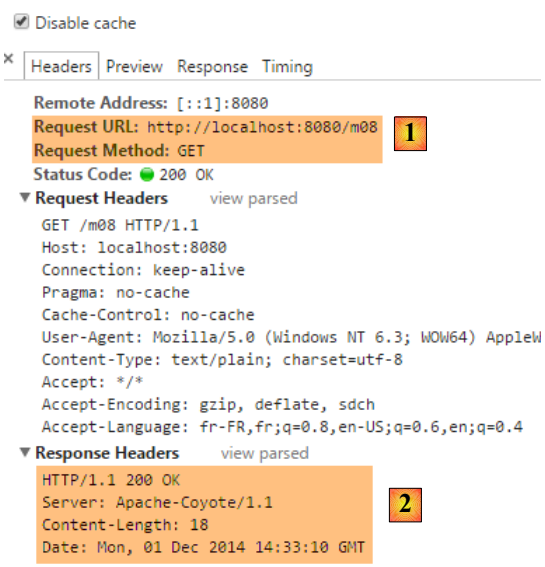
4.8 [/m08] : accès à l'objet [Writer]

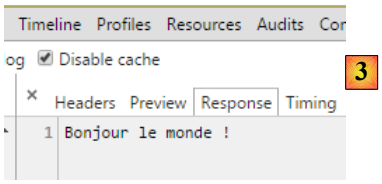
Considérons l'action suivante :

```
1.      // ----------------------- injection de writer -----------------------
2.      @RequestMapping(value = "/m08", method = RequestMethod.GET)
3.      public void m08(Writer writer) throws IOException {
4.         writer.write("Bonjour le monde !");
5.   }
```

- ligne 3 : Spring MVC injecte l'objet [Writer writer] qui permet d'écrire dans le flux de la réponse au client ;
- ligne 3 : l'action rend un type [void] ce qui indique qu'il doit construire lui-même la réponse au client ;
- ligne 4 : ajout d'un texte dans le flux de la réponse au client ;

Les résultats sont les suivants :





- en [2], on voit que l'entête HTTP [Content-Type] n'a pas été envoyé ;
- en [3], la réponse ;

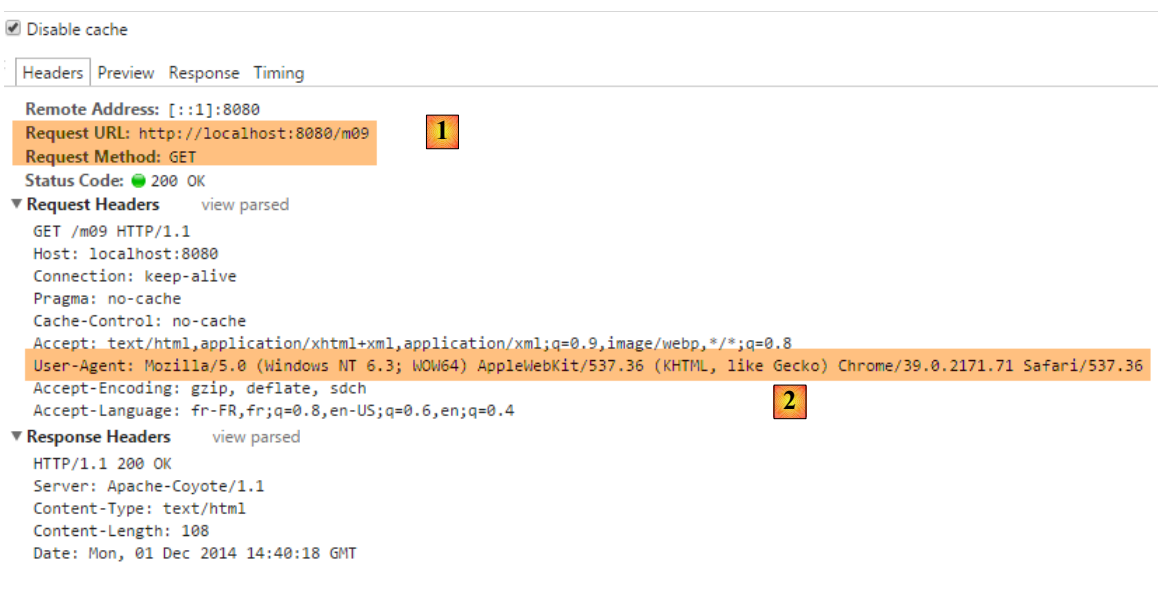
4.9 [/m09] : accéder à un entête HTTP

Considérons l'action suivante :

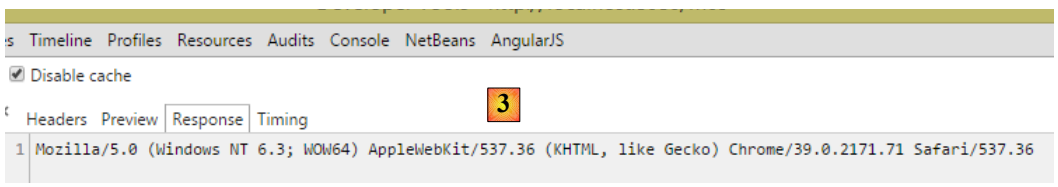
```
1.      // ---------------------- injection de RequestHeader ------------------------
2.      @RequestMapping(value = "/m09", method = RequestMethod.GET)
3.      public String m09(@RequestHeader("User-Agent") String userAgent) {
4.          return userAgent;
5.  }
```

- ligne 3 : l'annotation [@RequestHeader("User-Agent")] permet de récupérer l'entête HTTP [User-Agent] ;
- ligne 4 : on rend le texte de cet entête ;

Les résultats sont les suivants :



- en [2], l'entête HTTP [User-Agent] ;



- en [3], l'action [/m08] a correctement récupéré cet entête ;

4.10 [/m10, /m11] : accéder à un cookie

Un cookie est en général un entête HTTP que le :

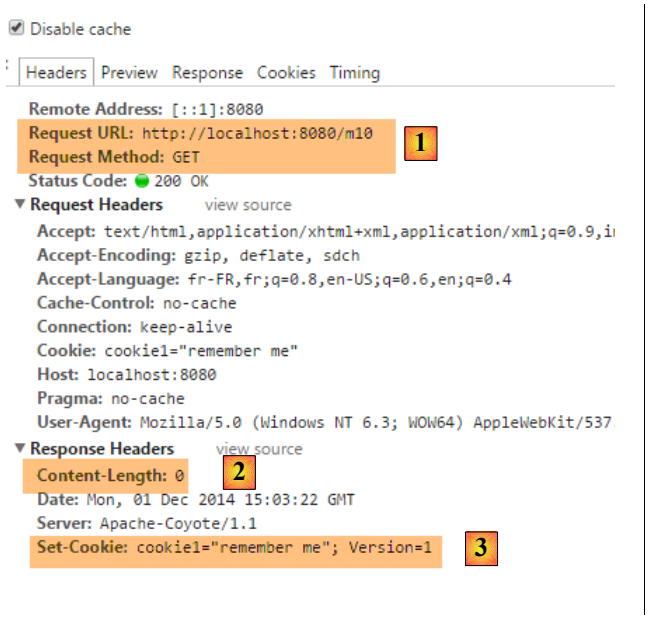
- serveur envoie une première fois au client ;
- client renvoie ensuite systématiquement au serveur ;

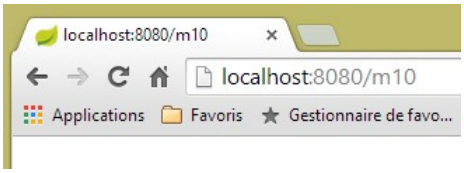
Créons d'abord une action qui crée le cookie :

```
1.      // ---------------------- création de Cookie -----------------------
2.      @RequestMapping(value = "/m10", method = RequestMethod.GET)
3.      public void m10(HttpServletResponse response) {
4.          response.addCookie(new Cookie("cookie1", "remember me"));
5.  }
```

- ligne 3 : on injecte l'objet [HttpServletResponse response] afin d'avoir le contrôle total sur la réponse ;
- ligne 4 : on crée un cookie avec une clé [cookie1] et une valeur [remember me] (**Note** : les caractères accentués dans la valeur d'un cookie provoquent des erreurs) ;
- ligne 3 : l'action ne rend rien. Par ailleurs, elle n'écrit rien dans le corps de la réponse. C'est donc un document vide que va recevoir le client. La réponse n'est utilisée que pour y ajouter l'entête HTTP d'un cookie ;

Voyons les résultats :





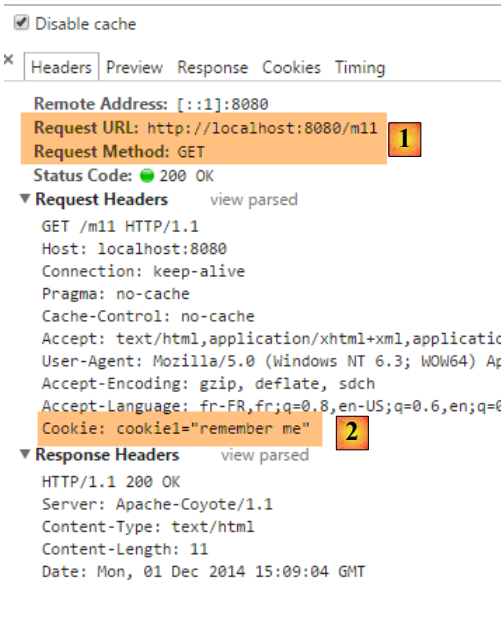
- en [1] : la requête ;
- en [2] : la réponse est vide ;
- en [3] : le cookie créé par l'action ;

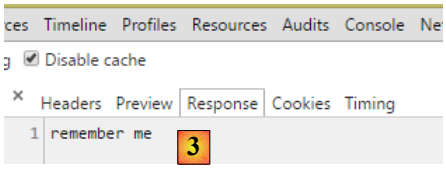
Maintenant créons une action pour récupérer ce cookie que le navigateur va désormais envoyer à chaque requête :

```
1.      // ----------------------- injection de Cookie ------------------------
2.      @RequestMapping(value = "/m11", method = RequestMethod.GET)
3.      public String m10(@CookieValue("cookie1") String cookie1) {
4.          return cookie1;
5.  }
```

- ligne 3 : l'annotation [@CookieValue("cookie1")] permet de récupérer le cookie de clé [cookie1] ;
- ligne 4 : cette valeur sera la réponse faite au client ;

Voyons les résultats :





- en [2], on voit que le navigateur renvoie le cookie ;

- en [3], l'action l'a bien récupéré ;

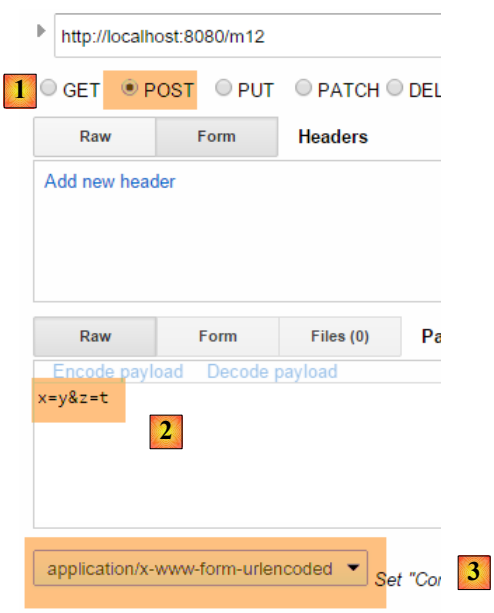
4.11 [/m12] : accéder au corps d'un POST

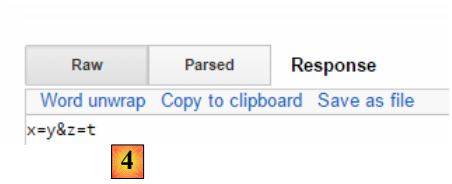
Les paramètres postés sont habituellement accompagnés de l'entête HTTP [Content-Type: application/x-www-form-urlencoded]. On peut accéder à la totalité de la chaîne postée. Nous créons l'action suivante :

```
1.     // ---------- récupérer le corps d'un POST de type String-----------------------
2.     @RequestMapping(value = "/m12", method = RequestMethod.POST)
3.     public String m12(@RequestBody String requestBody) {
4.        return requestBody;
5.  }
```

- ligne 3 : l'annotation [@RequestBody] permet de récupérer le corps du POST. Ici, on suppose que celui-ci est de type [String] ;
- ligne 4 : on renvoie ce corps au client ;

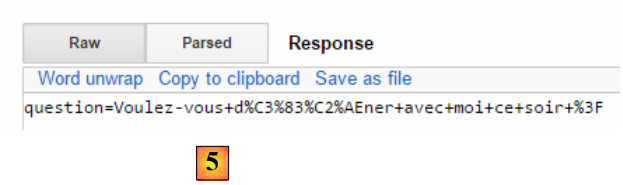
Voici un premier exemple :





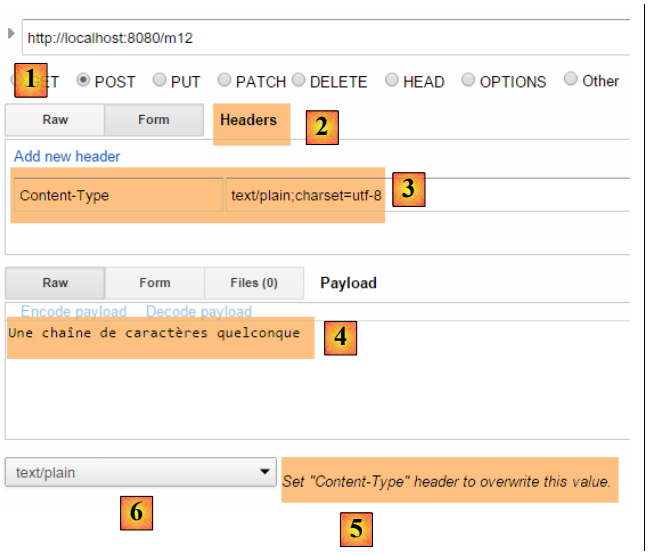
- en [2], les valeurs postées ;
- en [3], l'entête HTTP [Content-Type] de la requête ;
- en [4], la réponse du serveur ;

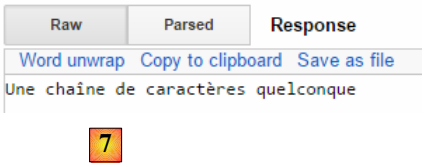
Les paramètres postés n'ont pas toujours la forme simple [p1=v1&p2=v2] qu'on a souvent utilisée jusqu'ici. Prenons un cas plus complexe :



- en [2-3] : on rentre les valeurs postées sous la forme [clé:value] ;
- en [5], la chaîne qui a été postée ;

Avec le type [Content-Type: application/x-www-form-urlencoded], la chaîne postée doit avoir la forme [p1=v1&p2=v2]. Si on veut poster n'importe quoi, on prendra le type [Content-Type: text/plain]. Voici un exemple :





- en [2-3], on crée l'entête HTTP [Content-Type]. Par défaut [5], c'est lui qui sera utilisé au lieu de celui défini en [6]. L'attribut [charset=utf-8] est important. Sans lui, on perd les caractères accentués de la chaîne postée ;
- en [4], la chaîne postée qu'on récupère correctement en [7] ;

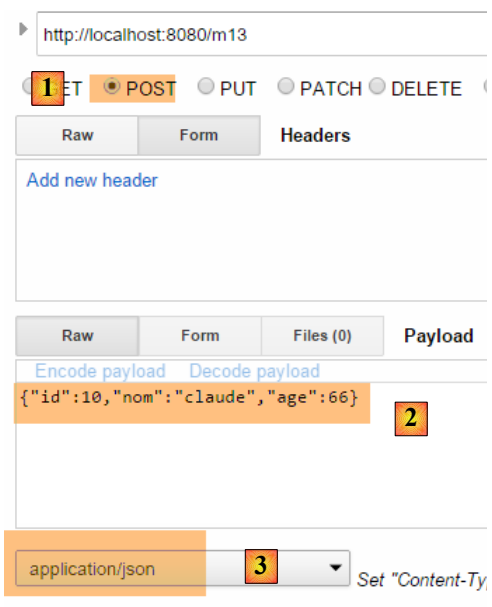
4.12 [/m13, /m14] : récupérer des valeurs postées en jSON

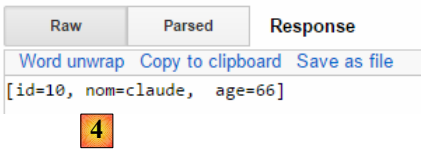
Il est possible de poster des paramètres avec l'entête HTTP [Content-Type: application/json]. Nous créons l'action suivante :

```
1.      // ---------------------- récupérer le corps jSON d'un POST
2.      @RequestMapping(value = "/m13", method = RequestMethod.POST, consumes = "application/json")
3.      public String m13(@RequestBody Personne personne) {
4.          return personne.toString();
5.  }
```

- ligne 2 : [consumes = "application/json"] précise que l'action attend un corps jSON ;
- ligne 3 : [@RequestBody] représente ce corps. Cette annotation a été associée à un objet de type [Personne]. Le corps jSON sera automatiquement désérialisé dans cet objet ;
- ligne 4 : on utilise la méthode [Personne].toString() pour retourner quelque chose qui ne soit pas la chaîne jSON envoyée ;

Voici un exemple :





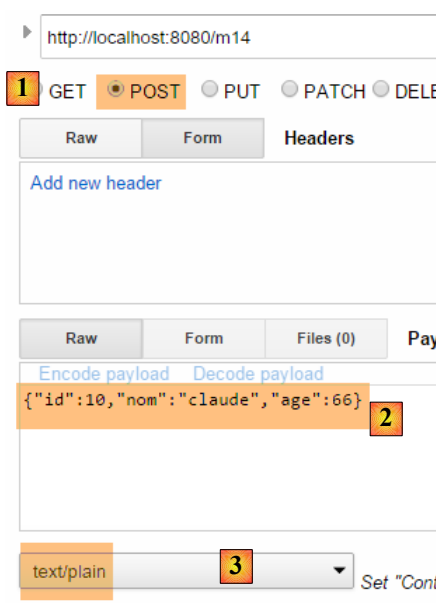
- en [2], la chaîne jSON postée ;
- en [3], le [Content-Type] de la requête ;
- en [4], la réponse du serveur ;

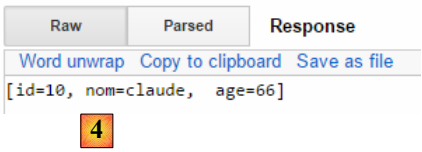
On peut faire la même chose différemment :

```
1.      // ---------------------- récupérer le corps jSON d'un POST 2 -------------------
2.      @RequestMapping(value = "/m14", method = RequestMethod.POST, consumes = "text/plain")
3.      public String m14(@RequestBody String requestBody) throws JsonParseException, JsonMappingException,
    IOException {
4.          Personne personne = new ObjectMapper().readValue(requestBody, Personne.class);
5.          return personne.toString();
6.  }
```

- ligne 2 : on a indiqué que la méthode attendait un flux de type [text/plain]. Spring MVC traitera alors le corps de la requête comme un type [String] (ligne 3) ;
- ligne 4 : on désérialise la chaîne jSON en un objet [Personne] (cf paragraphe 9.7, page 607) ;

Les résultats sont les suivants :

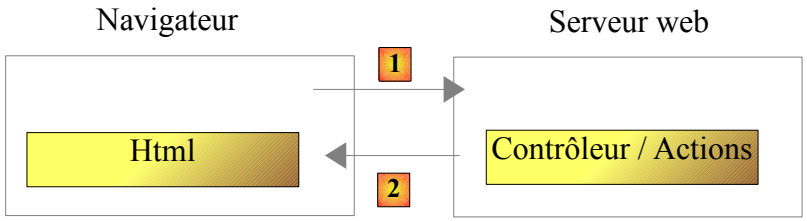




- en [3], bien mettre [text/plain] ;

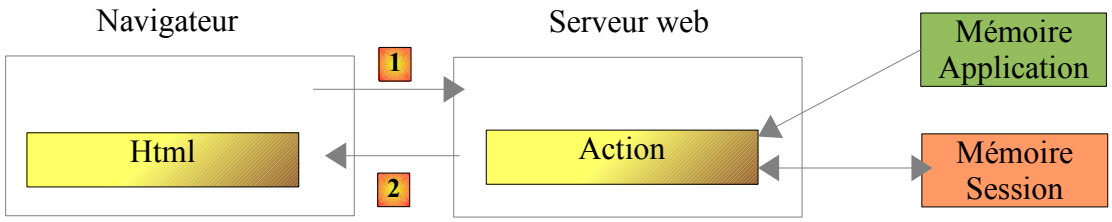
4.13 [/m15] : récupérer la session

Revenons sur l'architecture d'exécution d'une action :



La classe du contrôleur est instanciée au début de la requête du client et détruite à la la fin de celle-ci. Aussi ne peut-elle servir à mémoriser des données entre deux requêtes même si elle est appelée de façon répétée. On peut vouloir mémoriser deux types de données :

- des données **partagées par tous les utilisateurs** de l'application web. Ce sont en général des données en lecture seule ;
- des données partagées par les requêtes d'**un même client**. Ces données sont mémorisées dans un objet appelé **Session**. On parle alors de **session client** pour désigner la mémoire du client. Toutes les requêtes d'un client ont accès à cette session. Elles peuvent y stocker et y lire des informations.



Ci-dessus, nous montrons les types de mémoire auxquels a accès une action :

- la mémoire de l'application qui contient la plupart du temps des données en lecture seule et qui est accessible à tous les utilisateurs ;

- la mémoire d'un utilisateur particulier, ou session, qui contient des données en lecture / écriture et qui est accessible aux requêtes successives d'un même utilisateur ;
- non représentée ci-dessus, il existe une mémoire de requête, ou contexte de requête. La requête d'un utilisateur peut être traitée par plusieurs actions successives. Le contexte de la requête permet à une action 1 de transmettre de l'information à une action 2.

Regardons un premier exemple mettant en lumière ces différentes mémoires :

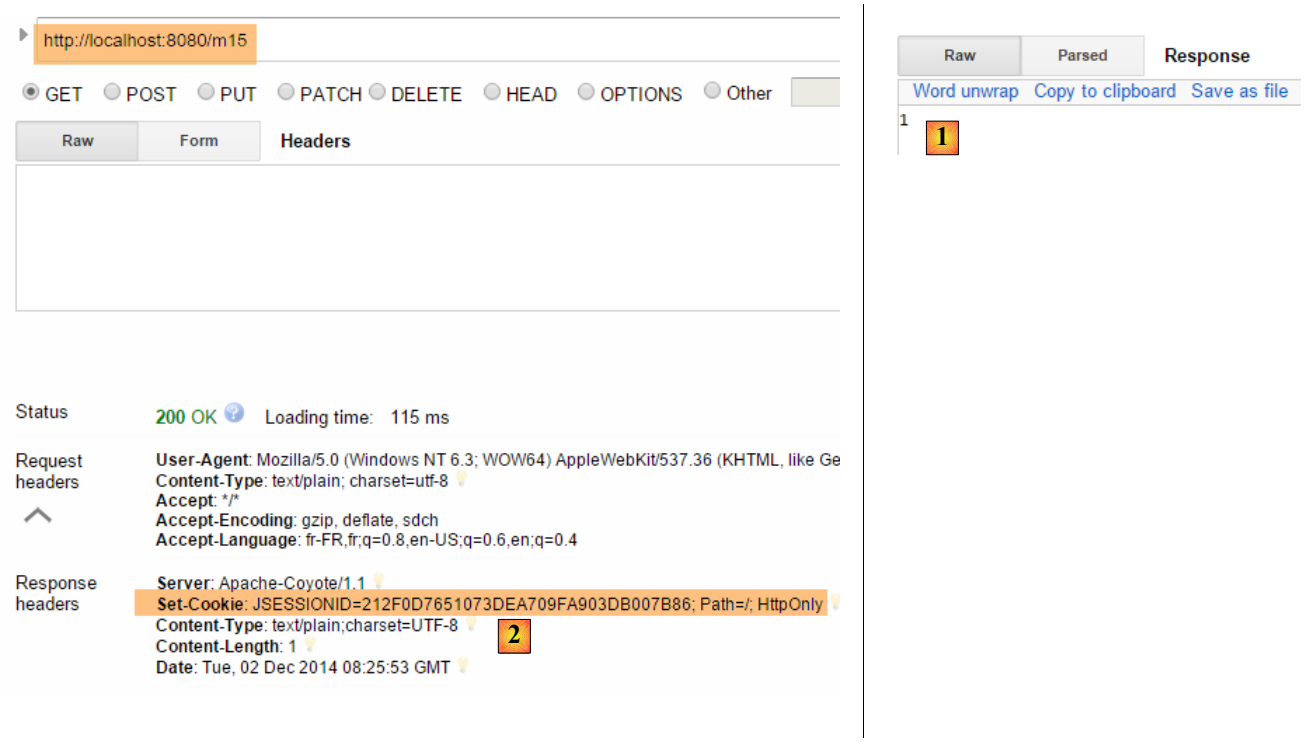
```
1.     // ---------------------- récupérer la session -----------------------
2.     @RequestMapping(value = "/m15", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
3.     public String m15(HttpSession session) {
4.        // on récupère l'objet de clé [compteur] dans la session
5.        Object objCompteur = session.getAttribute("compteur");
6.        // on le convertit en entier pour l'incrémenter
7.        int iCompteur = objCompteur == null ? 0 : (Integer) objCompteur;
8.        iCompteur++;
9.        // on le remet dans la session
10.       session.setAttribute("compteur", iCompteur);
11.       // on le rend comme résultat de l'action
12.       return String.valueOf(iCompteur);
13. }
```

Spring MVC maintient la session de l'utilisateur dans un objet de type [HttpSession].

- ligne 3 : on demande à Spring MVC d'injecter l'objet [HttpSession] dans les paramètres de l'action ;
- ligne 5 : on récupère dans celle-ci un attribut nommé [compteur]. Une session se comporte comme un dictionnaire, un ensemble de couples [clé, valeur]. Si la clé [compteur] n'existe pas dans la session, on récupère un pointeur *null* ;
- ligne 7 : la valeur associée à la clé [compteur] sera un type [Integer] ;
- ligne 8 : incrémentation du compteur ;
- ligne 10 : mise à jour du compteur dans la session ;
- ligne 12 : la valeur du compteur est envoyée au client ;

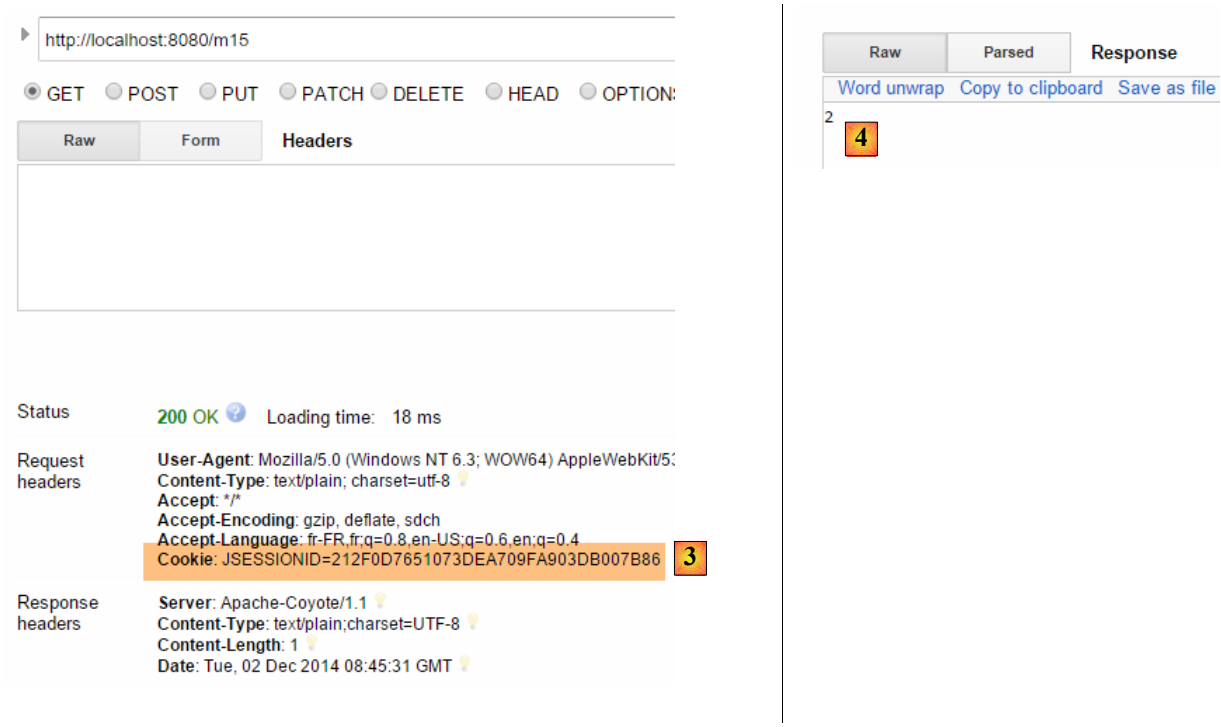
Lorsque [/m15] sera exécutée la :
- première fois, ligne 12 le compteur aura la valeur 1 ;
- seconde fois, ligne 5 on récupèrera cette valeur 1 pour la passer à 2 ;
- ...

Voici un exemple d'exécution :



- en [1], on obtient bien la 1ère valeur du compteur ;
- en [2], le serveur a envoyé un cookie de session. Il a la clé [JSESSIONID] et pour valeur une chaîne de caractères unique pour chaque utilisateur. On se rappelle que le navigateur renvoie systématiquement les cookies qu'il reçoit. Ainsi lorsqu'on va demander l'action [/m15] une seconde fois, le client va renvoyer ce cookie, ce qui va permettre au serveur de le reconnaître et de le rattacher à sa session. C'est de cette façon que la mémoire de l'utilisateur est maintenue ;

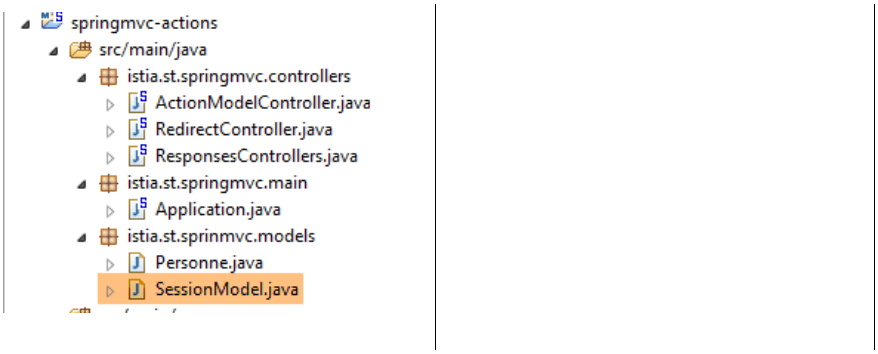
Voyons la seconde demande :



- en [3], on voit que le client renvoie le cookie de session. On peut remarquer que dans la réponse du serveur, il n'y a plus ce cookie de session. C'est désormais le client qui l'envoie pour se faire reconnaître ;
- en [4], la seconde valeur du compteur. Il a bien été incrémenté ;

4.14 [/m16] : récupérer un objet de portée [session]

On peut vouloir mettre toutes les données de la session d'un utilisateur dans un unique objet et mettre uniquement celui-ci dans la session. Nous suivons cette voie. Nous mettons le compteur dans l'objet [SessionModel] suivant :




```
1.  package istia.st.sprinmvc.models;
2.
3.  import org.springframework.context.annotation.Scope;
4.  import org.springframework.context.annotation.ScopedProxyMode;
5.  import org.springframework.stereotype.Component;
6.
7.  @Component
```

```
8.  @Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
9.  public class SessionModel {
10.
11.     private int compteur;
12.
13.     public int getCompteur() {
14.        return compteur;
15.     }
16.
17.     public void setCompteur(int compteur) {
18.        this.compteur = compteur;
19.     }
20.
21. }
```

- ligne 7 : l'annotation [@Component] est une annotation Spring (ligne 5) qui fait de la classe [SessionModel] un composant dont le cycle de vie est géré par Spring ;
- ligne 8 : l'annotation [@Scope(value = "session", proxyMode = ScopedProxyMode.***TARGET_CLASS***)] est également une annotation Spring (lignes 3-4). Lorsque Spring MVC la rencontre, la classe correspondante est créée et mise dans la session de l'utilisateur. L'attribut [proxyMode = ScopedProxyMode.***TARGET_CLASS***] est important. C'est grâce à lui que Spring MVC crée une instance par utilisateur et non une unique instance pour tous les utilisateurs (singleton) ;
- ligne 11 : le compteur ;

Pour que ce nouveau composant Spring soit reconnu, il faut vérifier la configuration de l'application dans la classe [Application] :

```
1.  package istia.st.springmvc.main;
2.
3.  import org.springframework.boot.SpringApplication;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.  import org.springframework.context.annotation.ComponentScan;
6.  import org.springframework.context.annotation.Configuration;
7.
8.  @Configuration
9.  @ComponentScan({"istia.st.springmvc.controllers"})
10. @EnableAutoConfiguration
11. public class Application {
12.
13.     public static void main(String[] args) {
14.        SpringApplication.run(Application.class, args);
15.     }
16. }
```

- ligne 9 : les composants Spring sont cherchés dans le package [istia.st.springmvc.controllers]. Ce n'est plus suffisant. Nous faisons évoluer cette ligne de la façon suivante :

```
@ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
```

Nous avons rajouté le package où se trouve la classe [SessionModel].

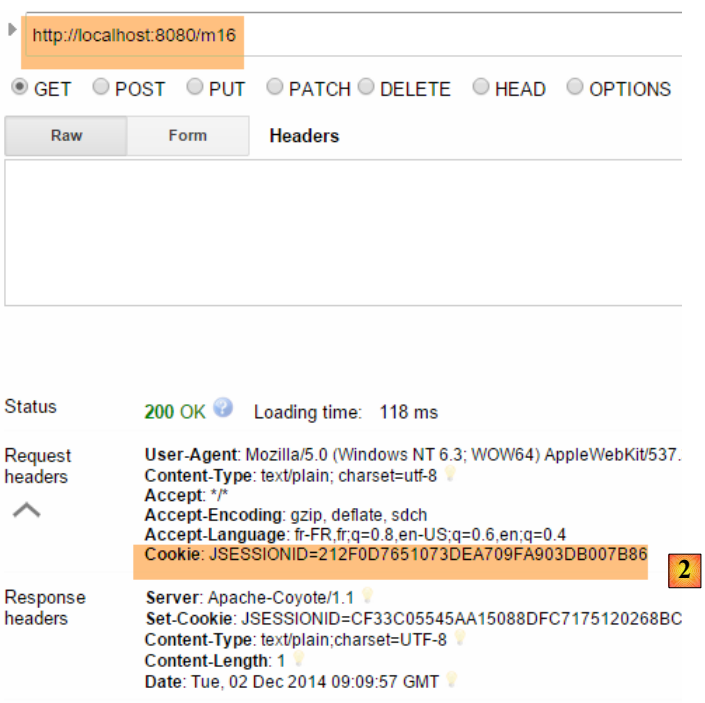
Maintenant, nous ajoutons l'action suivante :

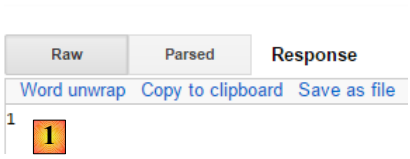
```
1.      @Autowired
2.      private SessionModel session;
3.
4.      // ------ gérer un objet de portée (scope) session [Autowired] -----------
5.      @RequestMapping(value = "/m16", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
6.      public String m16() {
7.         session.setCompteur(session.getCompteur() + 1);
8.         return String.valueOf(session.getCompteur());
9.  }
```

- lignes 1-2 : le composant Spring [SessionModel] est injecté [@Autowired] dans le contrôleur ;
- ligne 6 : on n'a plus besoin de l'objet [HttpSession] dans les paramètres de l'action ;
- ligne 7 : on récupère / incrémente le compteur ;
- ligne 8 : on rend sa valeur ;

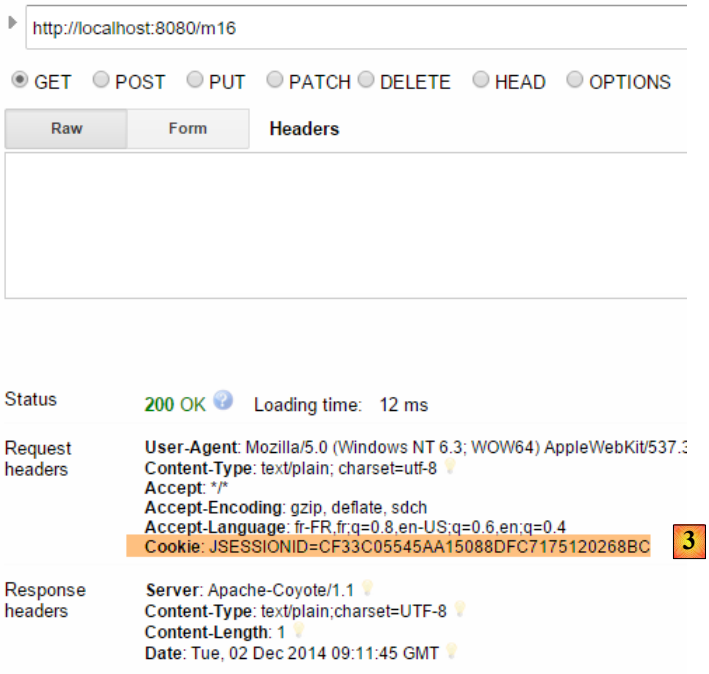
Voici un exemple d'exécution :

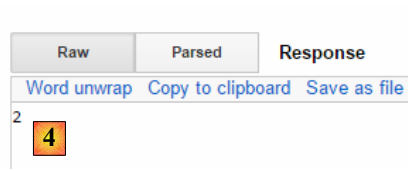
La 1ère fois



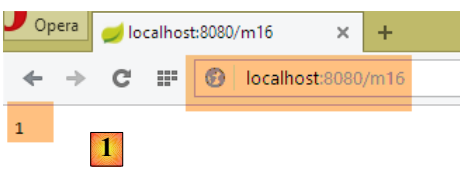


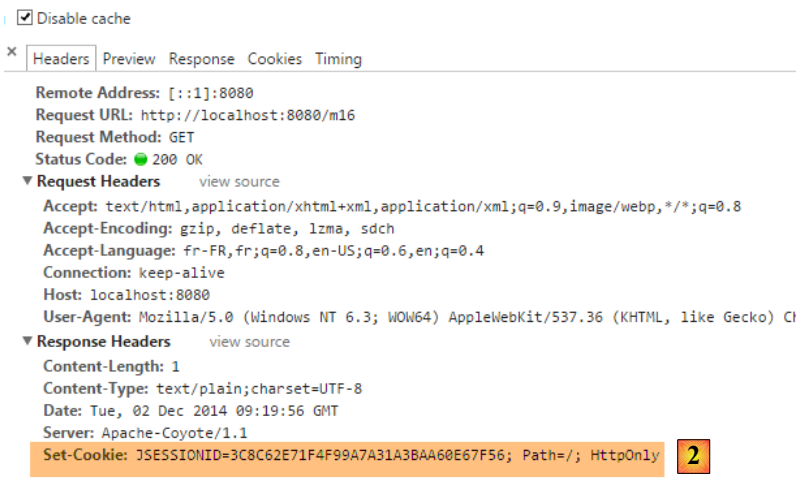
La seconde fois





Maintenant, prenons un autre navigateur qui va symboliser un deuxième utilisateur. Nous prenons ici un navigateur Opera :

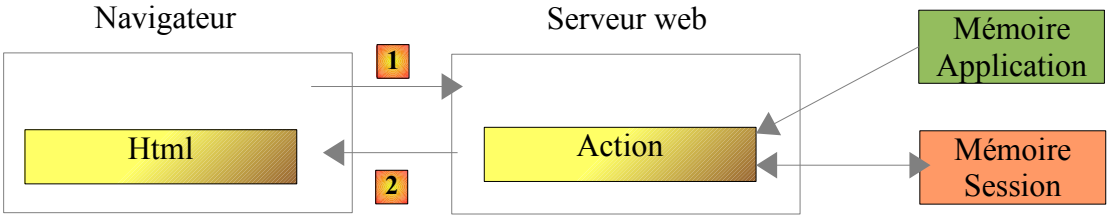




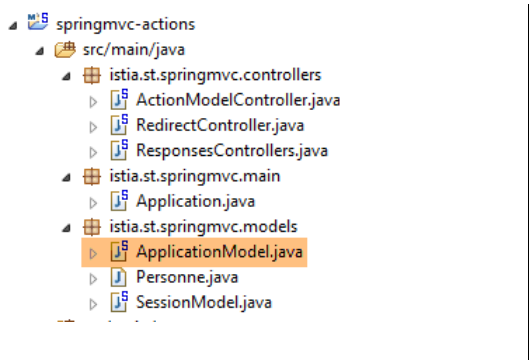
Ci-dessus en [1], ce deuxième utilisateur récupère une valeur de compteur à 1. Ce qui montre, que sa session et celle du premier utilisateur sont différentes. Si on regarde les échanges client / serveur (Ctrl-Maj-I pour Opera également), on voit en [2] que ce second utilisateur a un cookie de session différente de celui du 1er utilisateur. C'est ce qui assure l'indépendance des sessions.

4.15 [/m17] : récupérer un objet de portée [application]

Revenons sur l'architecture d'exécution d'une action :



Nous savons comment construire la session de l'utilisateur. Nous allons maintenant construire un objet de portée [application] dont le contenu sera en lecture seule et accessible à tous les utilisateurs. Nous introduisons la classe [ApplicationModel] qui sera l'objet de portée [application] :




```
1.  package istia.st.springmvc.models;
2.
3.  import java.util.concurrent.atomic.AtomicLong;
4.
5.  import org.springframework.stereotype.Component;
6.
7.  @Component
8.  public class ApplicationModel {
9.
10.     // compteur
```

```
11.     private AtomicLong compteur = new AtomicLong(0);
12.
13.     // getters et setters
14.     public AtomicLong getCompteur() {
15.         return compteur;
16.     }
17.
18.     public void setCompteur(AtomicLong compteur) {
19.         this.compteur = compteur;
20.     }
21.
22. }
```

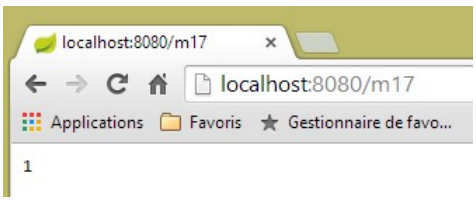
- ligne 5 : l'annotation [@Component] fait que la classe [ApplicationModel] sera un composant géré par Spring. La nature par défaut des composants Spring est le type [singleton] : le composant est créé en un unique exemplaire lorsque le conteneur Spring est instancié ç-à-d en général au démarrage de l'application. Nous pouvons utiliser ce cycle de vie pour stocker dans le singleton des informations de configuration qui seront accessibles à tous les utilisateurs ;
- ligne 11 : un compteur de type [AtomicLong]. Ce type a une méthode [incrementAndGet] dite atomique. Cela signifie qu'un thread qui exécute cette méthode est assuré qu'un autre thread ne lira pas la valeur du compteur (Get) entre sa lecture (Get) et son incrément (increment) par le 1er thread, ce qui provoquerait des erreurs puisque deux threads liraient la même valeur du compteur, et celui-ci au lieu d'être incrémenté de deux le serait de un ;

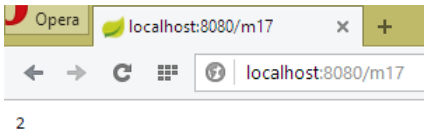
Nous créons la nouvelle action [/m17] suivante :

```
1.  @Autowired
2.      private ApplicationModel application;
3.
4.      // ----- gérer un objet de portée application [Autowired] ------------------------
5.      @RequestMapping(value = "/m17", method = RequestMethod.GET, produces = "text/plain;charset=UTF-8")
6.      public String m17() {
7.          return String.valueOf(application.getCompteur().incrementAndGet());
8.      }
```

- lignes 1-2 : on injecte le composant [ApplicationModel] dans le contrôleur. C'est un singleton. Donc chaque utilisateur aura une référence sur le même objet ;
- ligne 7 : on rend le compteur de portée [application] après l'avoir incrémenté ;

Voici deux exemples, l'un avec Chrome, l'autre avec Opera :





Ci-dessus, on voit que les deux navigateurs ont travaillé avec le même compteur, ce qui n'était pas le cas avec la session. Ces deux navigateurs symbolisent deux utilisateurs différents qui ont accès tous les deux aux données de portée [application]. De façon générale, on évitera de mettre dans les objets de portée [application] des informations en lecture / écriture comme il a été fait ci-dessus avec le compteur. En effet, les threads d'exécution des différents utilisateurs accèdent en même temps aux données de portée [application]. S'il y a des informations en écriture, il faut synchroniser les accès en écriture comme il a été fait ci-dessus avec le type [AtomicLong]. Les accès concurrents sont sources d'erreurs de programmation. Aussi préfèrera-t-on ne mettre que des informations en lecture seule dans les objets de portée [application].

4.16 [/m18] : récupérer un objet de portée [session] avec [@SessionAttributes]

Il existe une autre façon de récupérer des informations de portée [session]. Nous allons mettre en session l'objet suivant :

```
1.  package istia.st.springmvc.models;
2.
3.  public class Container {
4.      // le compteur
5.      public int compteur=10;
```

```
6.
7.      // les getters et setters
8.      public int getCompteur() {
9.         return compteur;
10.     }
11.
12.     public void setCompteur(int compteur) {
13.        this.compteur = compteur;
14.     }
15. }
```

Nous allons utiliser cet objet avec les deux actions suivantes :

```
1.      // utilisation de [@SessionAttribute] ----------------------
2.      @RequestMapping(value = "/m18", method = RequestMethod.GET)
3.      public void m18(HttpSession session) {
4.         // ici on met la clé [container] dans la session
5.         session.setAttribute("container", new Container());
6.      }
7.
8.      // utilisation de [@ModelAttribute] ----------------------
9.      // la clé [container] de la session sera ici injectée
10.     @RequestMapping(value = "/m19", method = RequestMethod.GET)
11.     public String m19(@ModelAttribute("container") Container container) {
12.        container.setCompteur(1 + container.getCompteur());
13.        return String.valueOf(container.getCompteur());
14.     }
```

- lignes 3-6 : l'action [/m18] ne rend aucun résultat. Elle ne sert qu'à créer un objet dans la session avec la clé [container] ;
- ligne 11 : dans l'action [/m19], on utilise l'annotation [@ModelAttribute]. Le comportement de cette annotation est assez complexe. Le paramètre [container] de cette annotation peut désigner diverses choses et en particulier un objet de la session. Il faut pour cela que celui-ci ait été déclaré avec une annotation [@SessionAttributes] sur la classe elle-même :

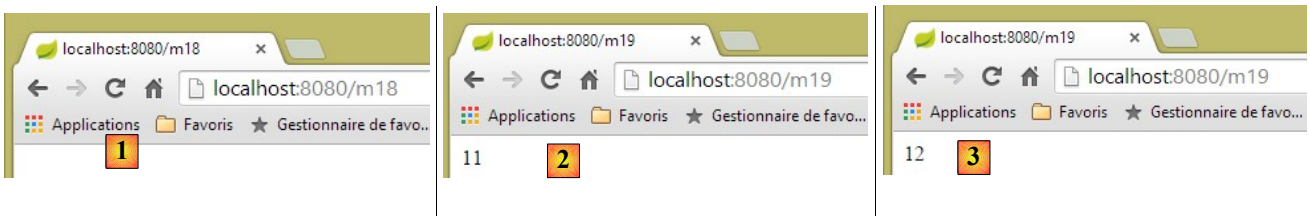
```
1.  @RestController
2.  @SessionAttributes({"container"})
3.  public class ActionModelController {
```

- la ligne 2 ci-dessus, désigne la clé [container] comme faisant partie des attributs de la session ;

Résumons :

- en [/m18], la clé [container] est mise en session ;
- l'annotation [@SessionAttributes({"container"})] fait que cette clé peut être injectée dans un paramètre annoté avec [@ModelAttribute("container")] ;
- pas visible dans l'exemple d'exécution qui va suivre, mais une information annotée avec [@ModelAttribute] fait automatiquement partie du modèle M transmis à la vue V ;

Voici un exemple d'exécution. Tout d'abord, on met la clé [container] dans la session avec l'action [/m18] [1]. Ensuite, on appelle deux fois l'action [/m19] pour voir le compteur s'incrémenter.



4.17 [/m20-/m23] : injection d'informations avec [@ModelAttribute]

Considérons la nouvelle action suivante :

```
1.      // l'attribut p fera partie de tous les modèles [Model] de vue ----------------
2.      @ModelAttribute("p")
3.      public Personne getPersonne() {
4.         return new Personne(7,"abcd", 14);
```

```
5.      }
6.
7.      // ---------------instanciation de @ModelAttribute -------------------------
8.      // sera injecté s'il est dans la session
9.      // sera injecté si le contrôleur a défini une méthode pour cet attribut
10.     // peut provenir des champs de l'URL s'il existe un convertisseur String --> type de l'attribut
11.     // sinon est construit avec le constructeur par défaut
12.     // ensuite les attributs du modèle sont initialisés avec les paramètres du GET ou du POST
13.     // le résultat final fera partie du modèle produit par l'action
14.
15.     // l'attribut p est injecté dans les arguments-----------------------
16.     @RequestMapping(value = "/m20", method = RequestMethod.GET)
17.     public Personne m20(@ModelAttribute("p") Personne personne) {
18.        return personne;
19. }
```

- ligne 2-5 : définissent un attribut de modèle nommé [p]. Il s'agit du modèle M d'une vue V, modèle représenté par un type [Model] dans Spring MVC. Un modèle se comporte comme un dictionnaire de couples [clé, valeur]. Ici, la clé [p] est associée à l'objet [Personne] construit par la méthode [getPersonne]. Le nom de la méthode peut être quelconque ;
- ligne 17 : l'attribut de modèle de clé [p] est injecté dans les paramètres de l'action. Cette injection se fait selon les règles des lignes 8-12. Ici, on sera dans le cas défini ligne 9. Donc ligne 17 le paramètre [Personne personne] sera l'objet [Personne(7,'abcd',14)] ;
- ligne 18 : on rend l'objet [personne] pour vérification. Celui-ci sera sérialisé en jSON avant d'être envoyé au client.

Voici un exemple :

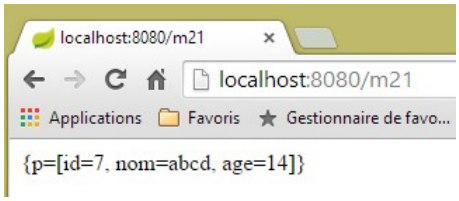
Maintenant, examinons l'action suivante :

```
1.      // --------- l'attribut p fait automatiquement partie du modèle M de la vue V
2.      @RequestMapping(value = "/m21", method = RequestMethod.GET)
3.      public String m21(Model model) {
4.         return model.toString();
5.  }
```

Une action qui veut faire afficher une vue V doit construire le modèle M de celle-ci. Spring MVC gère celui-ci avec un type [Model] qui peut être injecté dans les paramètres de l'action. Au départ ce modèle est vide ou contient les informations taguées avec l'annotation [@ModelAttribute]. L'action enrichit ou non ce modèle avant de le transmettre à une vue.

- ligne 3 : injection du modèle M ;
- ligne 4 : on veut voir ce qu'il y a dedans. On le sérialise en chaîne de caractères pour l'envoyer au client. Ici, la méthode [Personne.toString] va être utilisée. Il faut donc qu'elle existe ;

Voici une exécution :



Ci-dessus, on voit que les instructions :

```
1.      @ModelAttribute("p")
2.      public Personne getPersonne() {
3.         return new Personne(7,"abcd", 14);
4.   }
```

ont créé une entrée [p, Personne(7,'abcd',14)] dans le modèle. C'est toujours ainsi.

On considère maintenant le cas suivant :

```
1.      // sinon est construit avec le constructeur par défaut
2.      // ensuite les attributs du modèle sont initialisés avec les paramètres du GET ou du POST
```

avec l'action suivante :

```
1.      // --------- l'attribut de modèle [param1] fait partie du modèle mais est non initialisé
2.      @RequestMapping(value = "/m22", method = RequestMethod.GET)
3.      public String m22(@ModelAttribute("param1") String p1, Model model) {
4.         return model.toString();
5.   }
```

- ligne 3 : l'attribut de modèle de clé [param1] n'existe pas. Dans ce cas, le type associé doit avoir un constructeur par défaut. C'est le cas ici du type [String] mais on ne peut écrire [@ModelAttribute("param1") Integer p1] car la classe [Integer] n'a pas de constructeur par défaut ;
- ligne 4 : on retourne le modèle pour voir si l'attribut de modèle de clé [param1] en fait partie ;

Voici un exemple d'exécution :

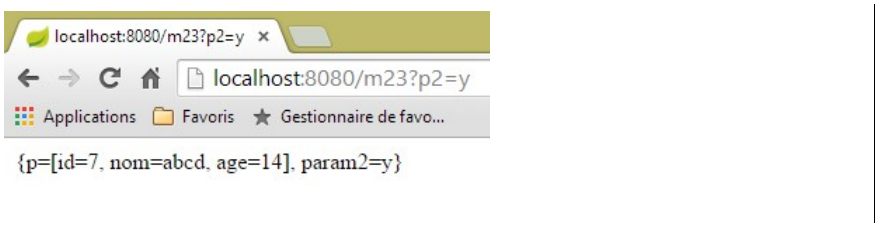
L'attribut de modèle [param1] est bien présent dans le modèle mais la méthode [toString] de la valeur associée ne donne pas d'indication sur cette valeur.

Considérons maintenant l'action suivante, où nous mettons explicitement une information dans le modèle :

```
1.      // --------- l'attribut de modèle [param2] est mis explicitement dans le modèle
2.      @RequestMapping(value = "/m23", method = RequestMethod.GET)
3.      public String m23(String p2, Model model) {
4.         model.addAttribute("param2",p2);
5.         return model.toString();
6.   }
```

- ligne 4 : la valeur [p2] récupérée ligne 3 est mise dans le modèle associée à la clé [param2] :

Voici un exemple d'exécution :

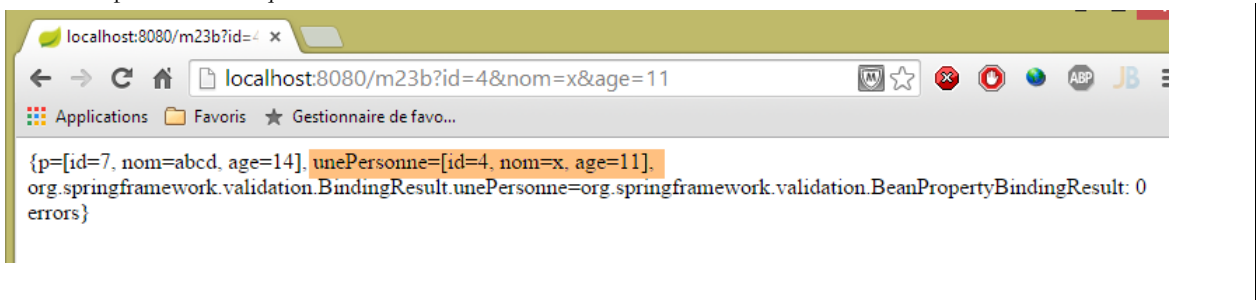


Les règles changent si le paramètre de l'action est un objet. Voici un premier exemple :

```
1.      // ------ l'attribut de modèle [unePersonne] est automatiquement mis dans le modèle
```

```
2.      @RequestMapping(value = "/m23b", method = RequestMethod.GET)
3.      public String m23b(@ModelAttribute("unePersonne") Personne p1, Model model) {
4.         return model.toString();
5.  }
```

L'action ne modifie pas le modèle qu'on lui a donné. Le résultat est le suivant :



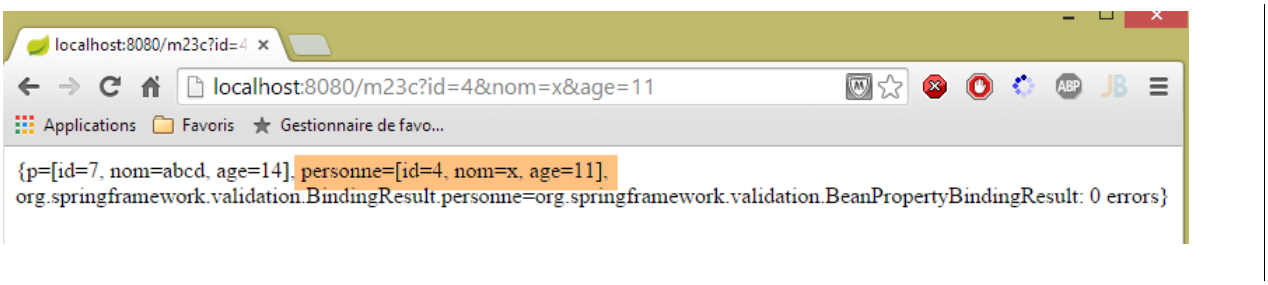
On constate que l'annotation [@ModelAttribute("unePersonne") Personne p1] a mis la personne [p1] dans le modèle, associée à la clé [unePersonne].

Considérons maintenant l'action suivante :

```
1.      // --------- la personne p1 est automatiquement mise dans le modèle
2.      // -------- avec pour clé le nom de sa classe avec le 1er caractère en minuscule
3.      @RequestMapping(value = "/m23c", method = RequestMethod.GET)
4.      public String m23c(Personne p1, Model model) {
5.         return model.toString();
6.  }
```

- ligne 4 : on n'a pas mis l'annotation [@ModelAttribute] ;

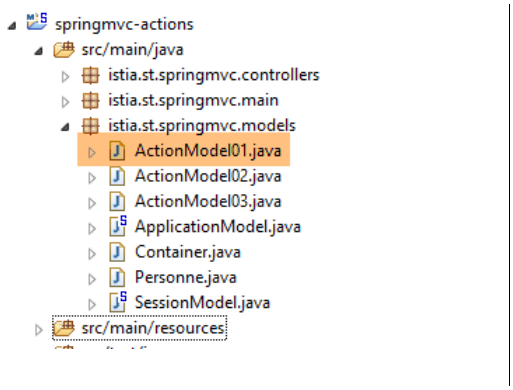
Le résultat est le suivant :



On constate que la présence du paramètre [Personne p1] a mis la personne [p1] dans le modèle, associée à la clé [personne] qui est le nom de la classe [Personne] avec le 1er caractère en minuscule.

4.18 [/m24] : validation du modèle de l'action

Considérons le modèle d'action [ActionModel01] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import javax.validation.constraints.NotNull;
4.
5.  public class ActionModel01 {
6.
7.      // data
8.      @NotNull
9.      private Integer a;
10.     @NotNull
11.     private Double b;
12.
13.     // getters et setters
14. ...
15.     }
```

- lignes 8 et 9 : l'annotation [@NotNull] est une contrainte de validation qui indique que la donnée annotée ne peut avoir la valeur *null* ;

Examinons maintenant l'action suivante :

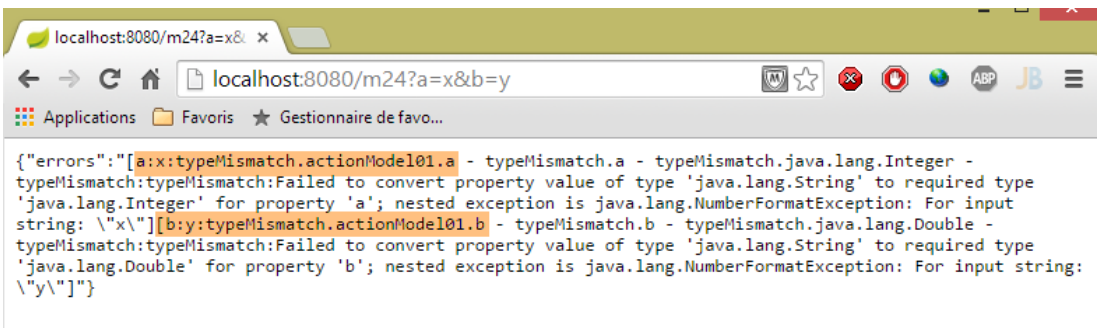
```
1.      // ----------------------- validation d'un modèle -----------------------
2.      @RequestMapping(value = "/m24", method = RequestMethod.GET)
3.      public Map<String, Object> m24(@Valid ActionModel01 data, BindingResult result) {
4.          Map<String, Object> map = new HashMap<String, Object>();
5.          // des erreurs ?
6.          if (result.hasErrors()) {
7.              StringBuffer buffer = new StringBuffer();
8.              // parcours de la liste des erreurs
9.              for (FieldError error : result.getFieldErrors()) {
10.                 buffer.append(String.format("[%s:%s:%s:%s:%s]", error.getField(), error.getRejectedValue(),
11.                         String.join(" - ", error.getCodes()), error.getCode(),error.getDefaultMessage()));
12.             }
13.             map.put("errors", buffer.toString());
14.         } else {
15.             // pas d'erreurs
16.             Map<String, Object> mapData = new HashMap<String, Object>();
17.             mapData.put("a", data.getA());
18.             mapData.put("b", data.getB());
19.             map.put("data", mapData);
20.         }
21.         return map;
22. }
```

- ligne 3 : un objet [ActionModel01] va être instancié et ses champs [a, b] initialisés avec des paramètres de mêmes noms. L'annotation [@Valid] indique que les contraintes de validité doivent être vérifiées. Les résultats de cette vérification seront placés dans le paramètre de type [BindingResult] (second paramètre). Les vérifications suivantes auront lieu :
  - à cause des annotations [@NotNull], les paramètres [a] et [b] doivent être présents ;
  - à cause du type [Integer a], le paramètre [a] qui par nature est de type [String] doit être convertible en un type [Integer] ;
  - à cause du type [Double b], le paramètre [b] qui par nature est de type [String] doit être convertible en un type [Double] ;

Avec l'annotation [@Valid], les erreurs de validation vont être reportées dans le paramètre [BindingResult result]. Sans l'annotation [@Valid], les erreurs de validation provoquent un plantage de l'action et le serveur envoie au client une réponse HTTP avec un statut 500 (Internal server error).

- ligne 3 : le résultat de l'action est de type [Map]. Ce sera la chaîne jSON de ce résultat qui sera envoyée au client. On construit deux sortes de dictionnaire :
  - en cas d'échec, un dictionnaire avec une entrée ['errors', value] où [value] est une chaîne de caractères décrivant toutes les erreurs (ligne 13) ;
  - en cas de réussite, un dictionnaire à une entrée ['data',value] où [value] est lui-même un dictionnaire à deux entrées : ['a', value], ['b', value] (ligne 19) ;
- lignes 9-12 : pour chaque erreur [error] détectée, on construit la chaîne [error.getField(), error.getRejectedValue(), error.Codes, error.getDefaultMessage()] :
  - le 1er élément est le champ erroné, [a] ou [b],
  - le second élément est la valeur refusée, [x] par exemple,
  - le troisième élément est une liste de codes d'erreur. Nous allons voir leurs rôles prochainement ;
  - le quatrième élément est le code de l'erreur. il fait partie de la liste précédente ;
  - le dernier élément est le message d'erreur par défaut. On peut en effet avoir plusieurs messages d'erreur ;

Voici quelques exemples d'exécution :

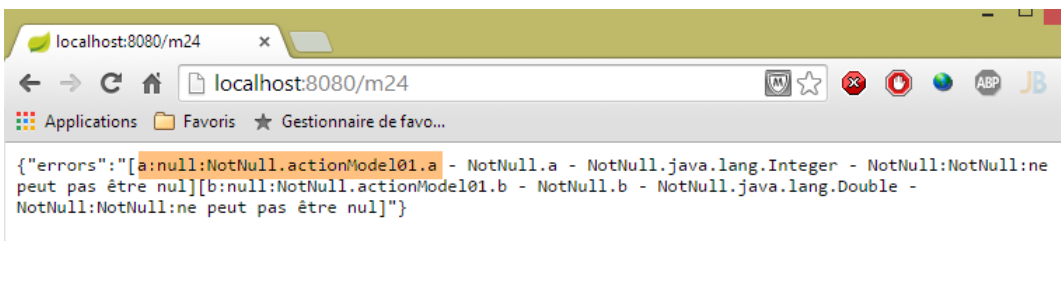


Ci-dessus, on voit que :

- l'affectation de 'x' au champ [ActionModel01.a] a échoué et le message d'erreur dit pourquoi ;
- l'affectation de 'y' au champ [ActionModel01.b] a échoué et le message d'erreur dit pourquoi ;

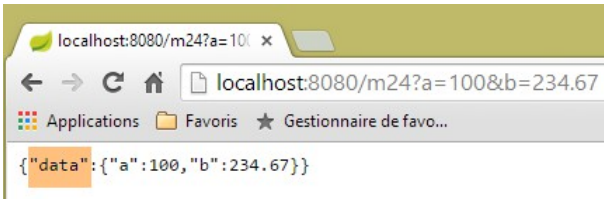
On notera les codes de l'erreur sur le champ [a] : [typeMismatch.actionModel01.a - typeMismatch.a - typeMismatch.java.lang.Integer - typeMismatch]. Nous reviendrons sur ces codes d'erreur lorsqu'il faudra personnaliser le message de l'erreur. On notera que le code de l'erreur est [typeMismatch].

Un autre exemple :



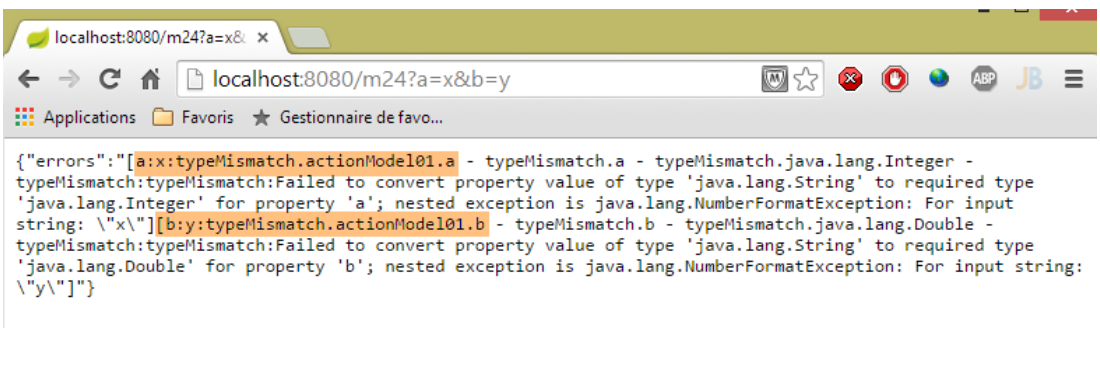
Ici, on n'a pas passé les paramètres [a] et [b]. Les validateurs [@NotNull] du modèle d'action [ActionModel01] ont alors joué leur rôle ;

Enfin, des valeurs correctes :



4.19 [m/24] : personnalisation des messages d'erreur

Revenons à une copie d'écran de l'exemple précédent :

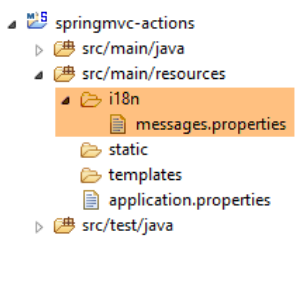


Nous voyons ci-dessus les messages d'erreur par défaut. Il est clair que nous ne pouvons les garder dans une application réelle. Il est possible de définir ces messages d'erreur. Pour cela, nous allons nous aider des codes de l'erreur. Ci-dessus, nous voyons que l'erreur pour le champ [a] a les codes suivants : [typeMismatch.actionModel01.a - typeMismatch.a - typeMismatch.java.lang.Integer - typeMismatch]. Ces codes d'erreur vont du plus précis au moins précis :

- [typeMismatch.actionModel01.a] : erreur de type sur le champ [a] du type [ActionModel01] ;
- [typeMismatch.a] : erreur de type sur un champ nommé [a] ;
- [typeMismatch.java.lang.Integer] : erreur de type sur un type Integer ;
- [typeMismatch] : erreur de type ;

On remarque également que le code d'erreur sur le champ [a] obtenu par [error.getCode()] est [typeMismatch] (cf copie d'écran ci-dessus).

Nous allons placer les messages d'erreur dans un fichier de propriétés :



Le fichier [messages.properties] ci-dessus sera le suivant :

```
1. NotNull=Le champ ne peut être vide
2. typeMismatch=Format invalide
3. typeMismatch.model01.a=Le paramètre [a] doit être entier
```

Chaque ligne a la forme suivante :

```
clé=message
```

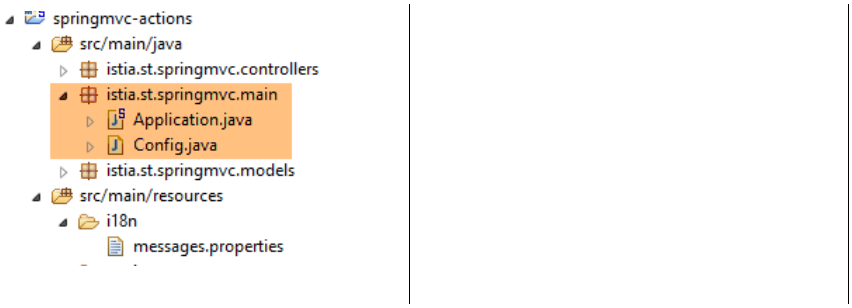
Ici, la clé sera un code d'erreur et le message, le message d'erreur associé à ce code.

Rappelons les codes d'erreur pour les deux champs :

- [typeMismatch.actionModel01.a - typeMismatch.a - typeMismatch.java.lang.Integer - typeMismatch], lorsque le paramètre [a] est invalide ;
- [typeMismatch.actionModel01.b - typeMismatch.b - typeMismatch.java.lang.Double - typeMismatch:typeMismatch] lorsque le paramètre [b] est invalide ;
- [NotNull.actionModel01.a - NotNull.a - NotNull.java.lang.Integer - NotNull] lorsque le paramètre [a] est absent ;
- [NotNull.actionModel01.b - NotNull.b - NotNull.java.lang.Double - NotNull] lorsque le paramètre [b] est absent ;

Le fichier [messages.properties] doit comporter un message d'erreur pour tous les cas d'erreur possibles. Pour le cas :
- des paramètres [a] et [b] absents, c'est le code [NotNull] qui sera utilisé ;
- du paramètre [a] incorrect, nous avons mis des messages pour deux codes [typeMismatch.actionModel01.a, typeMismatch]. Nous verrons lequel est utilisé ;
- du paramètre [b] incorrect, c'est le code [typeMismatch] qui sera utilisé ;

Pour que le fichier [messages.properties] soit utilisé, il faut configurer Spring :



Nous enlevons les annotations de configuration de la classe [Application] :

```
1.  package istia.st.springmvc.main;
2.
3.  import org.springframework.boot.SpringApplication;
4.
5.  public class Application {
6.
7.      public static void main(String[] args) {
8.          SpringApplication.run(Config.class, args);
9.      }
10. }
```

- ligne 8 : l'application Spring Boot est lancée. Le premier paramètre de la méthode statique [SpringApplication.*run*] est la classe qui configure désormais l'application ;

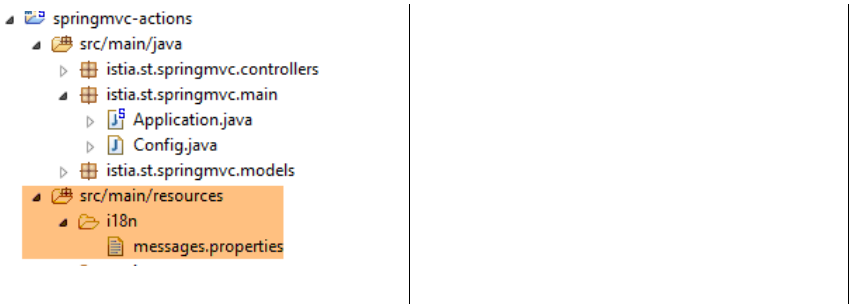
La classe [Config] est la suivante :

```
1.  package istia.st.springmvc.main;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.context.MessageSource;
5.  import org.springframework.context.annotation.Bean;
6.  import org.springframework.context.annotation.ComponentScan;
7.  import org.springframework.context.annotation.Configuration;
8.  import org.springframework.context.support.ResourceBundleMessageSource;
9.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
10.
11. @Configuration
12. @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
13. @EnableAutoConfiguration
14. public class Config extends WebMvcConfigurerAdapter {
15.     @Bean
```

```
16.     public MessageSource messageSource() {
17.         ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
18.         messageSource.setBasename("i18n/messages");
19.         return messageSource;
20.     }
21. }
```

- lignes 11-13 : on retrouve les annotations de configuration qui étaient auparavant dans la classe [Application] ;
- ligne 14 : pour configurer une application Spring MVC, il faut étendre la classe [WebMvcConfigurerAdapter] ;
- ligne 15 : l'annotation [@Bean] introduit un composant Spring, un singleton ;
- ligne 16 : on définit un bean nommé [messageSource] (le nom de la méthode). Ce bean sert à definir les fichiers de messages de l'application et il doit avoir obligatoirement ce nom ;
- lignes 17-19 : indique à Spring que le fichier des messages :
  - est dans le dossier [i18n] dans le Classpath du projet (ligne 18),
  - s'appelle [messages.properties] (ligne 18). En fait le terme [messages] est la racine des noms des fichiers de messages plutôt que le nom lui-même. Nous allons voir que dans le cadre de l'internationalisation, on peut trouver plusieurs fichiers de messages, un par culture gérée. Ainsi peut-on avoir [messages_fr.properties] pour la langue française et [messages_en.properties] pour la langue anglaise. Les suffixes ajoutés à la racine [messages] sont normalisés. On ne peut pas mettre n'importe quoi ;

Dans le projet STS, il faut mettre le dossier [i18n] dans le dossier des ressources car celui-ci est mis dans le Classpath du projet :



Pour exploiter ce fichier, nous créons la nouvelle action suivante :

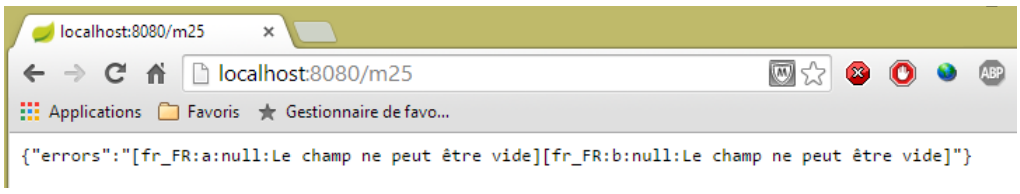
```
1.  // validation d'un modèle, gestion des messages d'erreur ------------------------
2.     @RequestMapping(value = "/m25", method = RequestMethod.GET)
3.     public Map<String, Object> m25(@Valid ActionModel01 data, BindingResult result, HttpServletRequest
    request)
4.             throws Exception {
5.         // le dictionnaire des résultats
6.         Map<String, Object> map = new HashMap<String, Object>();
7.         // le contexte de l'application Spring
8.         WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(request.getServletContext());
9.         // locale
10.        Locale locale = RequestContextUtils.getLocale(request);
11.        // des erreurs ?
12.        if (result.hasErrors()) {
13.            StringBuffer buffer = new StringBuffer();
14.            for (FieldError error : result.getFieldErrors()) {
15.                // recherche du msg d'erreur à parir des codes d'erreur
16.                // le msg est cherché dans les fichiers de messages
17.                // les codes d'erreur sous forme de tableau
18.                String[] codes = error.getCodes();
19.                // sous forme de chaîne
20.                String listCodes = String.join(" - ", codes);
21.                // recherche
22.                String msg = null;
23.                int i = 0;
24.                while (msg == null && i < codes.length) {
25.                    try {
26.                        msg = ctx.getMessage(codes[i], null, locale);
27.                    } catch (Exception e) {
28.
29.                    }
30.                    i++;
31.                }
```

```
32.             // a-t-on trouvé ?
33.             if (msg == null) {
34.                throw new Exception(String.format("Indiquez un message pour l'un des codes [%s]",
    listCodes));
35.             }
36.             // on a trouvé - on ajoute le msg d'erreur à la chaîne des msg d'erreur
37.             buffer.append(String.format("[%s:%s:%s:%s]", locale.toString(), error.getField(),
    error.getRejectedValue(),
38.                   String.join(" - ", msg)));
39.          }
40.          map.put("errors", buffer.toString());
41.       } else {
42.          // ok
43.          Map<String, Object> mapData = new HashMap<String, Object>();
44.          mapData.put("a", data.getA());
45.          mapData.put("b", data.getB());
46.          map.put("data", mapData);
47.       }
48.       return map;
49.    }
```

Ce code est analogue à celui de l'action [/m24]. Nous expliquons les différences :

- ligne 3 : on injecte la requête [HttpServletRequest request] dans les paramètres de l'action. Nous allons en avoir besoin ;
- lignes 7-8 : nous récupérons le contexte de Spring. Ce contexte contient tous les beans Spring de l'application. Il permet également d'accéder aux fichiers de messages ;
- ligne 10 : on récupère la locale de l'application. Ce terme est explicité un peu plus loin ;
- lignes 15-31 : pour chaque erreur, on cherche un message correspondant à l'un de ces codes d'erreur. Ils sont cherchés dans l'ordre des codes trouvés dans [error.getCodes()]. Dès qu'un message est trouvé, on s'arrête ;
- ligne 26 : la façon de récupérer un message dans [messages.properties] :
  - le premier paramètre est le code cherché dans [messages.properties],
  - le second est un tableau de paramètres car parfois les messages sont paramétrés. Ce n'est pas le cas ici,
  - le troisième est la locale utilisée (obtenue ligne 10). La locale désigne la langue utilisée, [fr_FR] pour le français de France, [en_US] pour l'anglais des USA. Le message est cherché dans messages_[locale].properties donc par exemple [messages_fr_FR.properties]. Si ce fichier n'existe pas, le message est cherché dans [messages_fr.properties]. Si ce fichier n'existe pas, le message est cherché dans [messages.properties]. C'est ce dernier cas qui fonctionnera pour nous ;
- lignes 25-29 : de façon un peu inattendue, lorsqu'on cherche un code inexistant dans un fichier de messages, on a une exception plutôt qu'un pointeur *null* ;
- ligne 33-35 : on traite le cas de l'absence de message d'erreur ;
- lignes 37-38 : on construit la chaîne d'erreur. Dans celle-ci, on inclut la locale et le message d'erreur trouvé ;

Voici des exemples d'exécution :

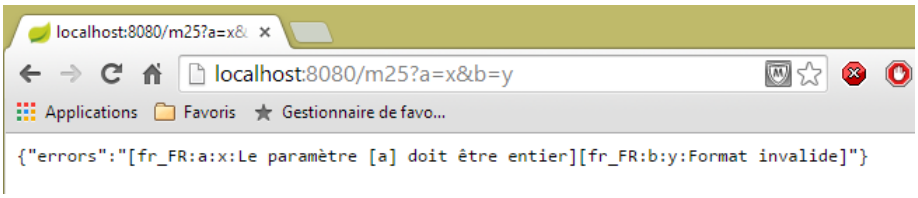


On voit que :

- la locale de l'application est [fr_FR]. C'est une valeur par défaut puisque nous n'avons rien fait pour l'initialiser ;
- que le message utilisé pour les deux champs est le suivant :

```
NotNull=Le champ ne peut être vide
```

Un autre exemple :



On voit que :

- le message d'erreur utilisé pour le paramètre [a] est le suivant :

```
typeMismatch.actionModel01.a=Le paramètre [a] doit être entier
```

- le message d'erreur utilisé pour le paramètre [b] est le suivant :

```
typeMismatch=Format invalide
```

Pourquoi deux messages différents ? Pour le paramètre [a], il y avait deux messages possibles :

```
1. typeMismatch=Format invalide
2. typeMismatch.actionModel01.a=Le paramètre [a] doit être entier
```

Les codes d'erreur ont été explorés dans l'ordre du tableau [error.getCodes()]. Il se trouve que cet ordre va du code le plus précis au code le plus général. C'est pourquoi le code [typeMismatch.model01.a] a été trouvé le premier.

4.20 [/m25] : internationalisation d'une application Spring MVC

Maintenant que nous savons personnaliser les messages d'erreur en français, nous voudrions les avoir également en anglais, ce qui nous amène à l'internationalisation d'une application Spring MVC. Pour gérer celle-ci, nous allons étoffer la classe de configuration [Config] qui devient la suivante :

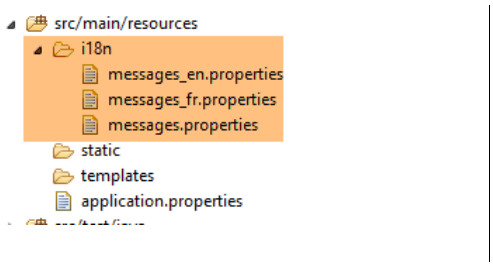
```
1.  package istia.st.springmvc.main;
2.
3.  import java.util.Locale;
4.
5.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
6.  import org.springframework.context.MessageSource;
7.  import org.springframework.context.annotation.Bean;
8.  import org.springframework.context.annotation.ComponentScan;
9.  import org.springframework.context.annotation.Configuration;
10. import org.springframework.context.support.ResourceBundleMessageSource;
11. import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
12. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
13. import org.springframework.web.servlet.i18n.CookieLocaleResolver;
14. import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
15.
16. @Configuration
17. @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
18. @EnableAutoConfiguration
19. public class Config extends WebMvcConfigurerAdapter {
20.     @Bean
21.     public MessageSource messageSource() {
22.         ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
23.         messageSource.setBasename("i18n/messages");
24.         return messageSource;
25.     }
26.
27.     @Bean
28.     public LocaleChangeInterceptor localeChangeInterceptor() {
29.         LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
30.         localeChangeInterceptor.setParamName("lang");
31.         return localeChangeInterceptor;
32.     }
33.
34.     @Override
```

```
35.     public void addInterceptors(InterceptorRegistry registry) {
36.         registry.addInterceptor(localeChangeInterceptor());
37.     }
38.
39.     @Bean
40.     public CookieLocaleResolver localeResolver() {
41.         CookieLocaleResolver localeResolver = new CookieLocaleResolver();
42.         localeResolver.setCookieName("lang");
43.         localeResolver.setDefaultLocale(new Locale("fr"));
44.         return localeResolver;
45.     }
46. }
```

- lignes 28-32 : on crée un intercepteur de requête. Un intercepteur de requête étend l'interface [HandlerInterceptor]. Une telle classe inspecte la requête entrante avant qu'elle ne soit traitée par une action. Ici l'intercepteur [localeChangeInterceptor] va rechercher un paramètre nommé [lang] dans la requête entrante, GET ou POST et va changer la locale de l'application en fonction de ce paramètre. Ainsi si le paramètre est [lang=en_US], la locale de l'application deviendra l'anglais des USA ;
- lignes 34-37 : on redéfinit la méthode [WebMvcConfigurerAdapter.addInterceptors] pour ajouter l'intercepteur précédent ;
- lignes 39-45 : servent à paramétrer la façon dont la locale va être encapsulée dans un cookie. On sait qu'un cookie peut servir de mémoire de l'utilisateur, puisque le navigateur client le renvoie systématiquement au serveur. L'intercepteur [localeChangeInterceptor] précédent crée un cookie encapsulant la locale. La ligne 42 donne le nom [lang] à ce cookie. Le cookie est également utilisé pour changer la locale ;
- ligne 43 : indique qu'en l'absence du cookie [lang], la locale sera [fr] ;

En résumé, la locale d'une requête peut être fixée de deux façons :
- en passant un paramètre nommé [lang] ;
- en envoyant un cookie nommé [lang]. Ce cookie est automatiquement créé à l'issue de la méthode précédente ;

Pour exploiter cette locale, nous allons créer des fichiers de messages pour les locales [fr] et [en] :



Le fichier [messages_fr.properties] est le suivant :

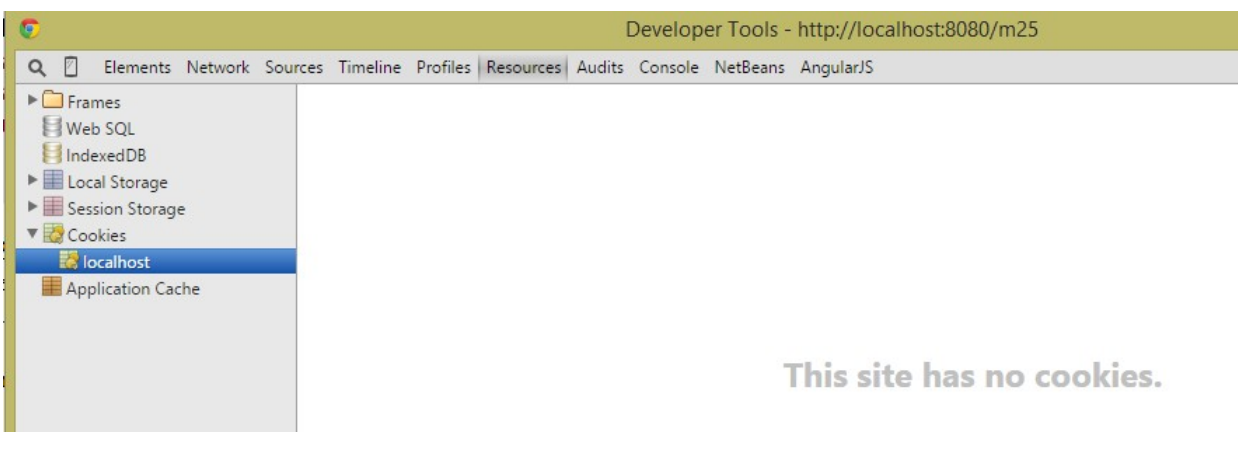
```
1. NotNull=Le champ ne peut être vide
2. typeMismatch=Format invalide
3. typeMismatch.actionModel01.a=Le paramètre [a] doit être entier
```

Le fichier [messages_en.properties] est le suivant :

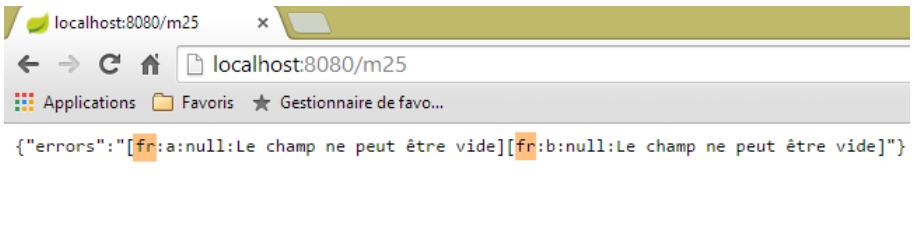
```
1. NotNull=The field can't be empty
2. typeMismatch=Invalid format
3. typeMismatch.actionModel01.a=Parameter [a] must be an integer
```

Le fichier [messages.properties] est une recopie du fichier [messages_en.properties]. On rappelle que le fichier [messages.properties] est utilisé lorsqu'aucun fichier correspondant à la locale de la requête n'est trouvé. Dans notre cas, si l'utilisateur envoie un paramètre [lang=en], comme le fichier [messages_en.properties] n'existe pas, c'est le fichier [messages.properties] qui sera utilisé. L'utilisateur aura donc des messages en anglais.

Essayons. Tout d'abord, dans l'environnement de déceloppement de Chrome (Ctrl-Maj-I), vérifiez vos cookies :



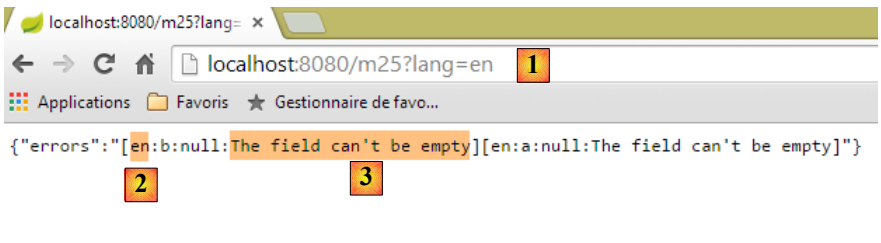
Si vous avez un cookie nommé [lang], supprimez-le. Puis avec Chrome, demandez l'URL [http://localhost:8080/m25] :



Le navigateur a envoyé les entêtes HTTP suivants :

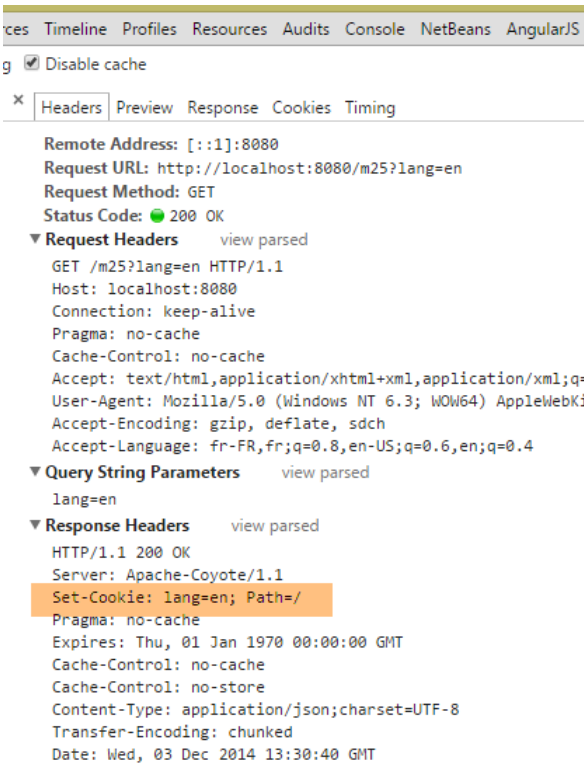
```
1.  GET /m25 HTTP/1.1
2.  Host: localhost:8080
3.  Connection: keep-alive
4.  Pragma: no-cache
5.  Cache-Control: no-cache
6.  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
7.  User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71
    Safari/537.36
8.  Referer: http://localhost:8080/m25
9.  Accept-Encoding: gzip, deflate, sdch
10. Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
```

On voie que dans ces entêtes, il n'y a pas de cookie [lang]. Notre code dans ce cas, utilise la locale [fr]. C'est ce que montre la copie d'écran. Essayons un autre cas :

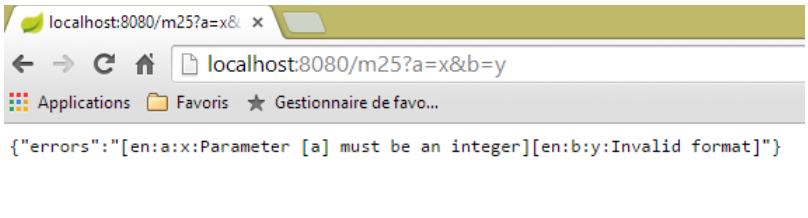


- en [1], on a passé le paramètre [lang=en] pour passer la locale à [en] ;
- en [2], on voit la nouvelle locale ;
- en [3], le message est passé en anglais ;

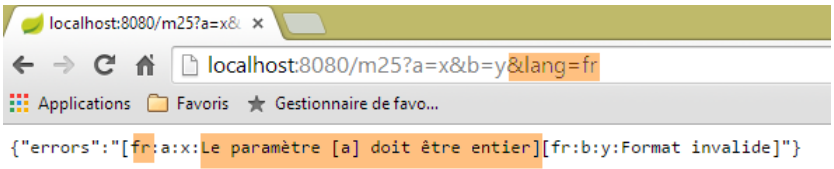
Regardons maintenant les échanges HTTP :



On voit ci-dessus que le serveur a renvoyé un cookie [lang]. Cela a une conséquence importante : la locale de la prochaine requête sera [en] de nouveau à cause du cookie [lang] qui va être renvoyé par le navigateur. On devrait donc garder les messages en anglais. Vérifions-le :



Ci-dessus, on voit que la locale est restée à [en]. A cause du cookie qu'envoie systématiquement le navigateur, elle le restera tant que l'utilisateur ne la changera pas en envoyant le paramètre [lang] comme suit :



4.21 [/m26] : injection de la locale dans le modèle de l'action

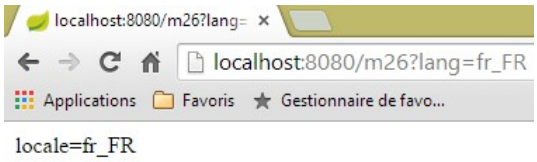
Dans l'exemple précédent, nous avons vu une façon de récupérer la locale de la requête :

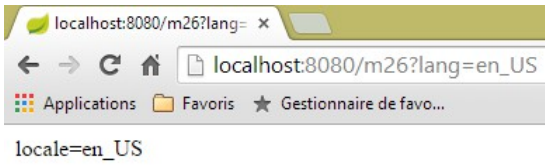
```
1.      @RequestMapping(value = "/m25", method = RequestMethod.GET)
2.      public Map<String, Object> m25(@Valid ActionModel01 data, BindingResult result, HttpServletRequest
    request)
```

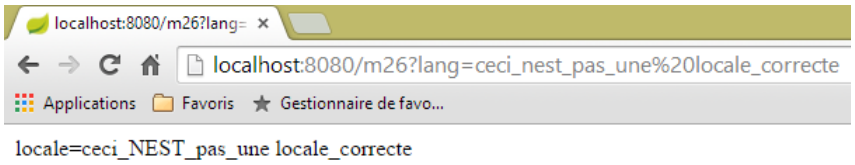
```
3.           throws Exception {
4.  ...
5.        // locale
6.        Locale locale = RequestContextUtils.getLocale(request);
7.  // des erreurs ?
```

La locale peut être directement injectée dans les paramètres de l'action. Voici un exemple :

```
1.     @RequestMapping(value = "/m26", method = RequestMethod.GET)
2.     public String m26(Locale locale) {
3.        return String.format("locale=%s", locale.toString());
4.  }
```







On voit ci-dessus qu'il n'y a pas de vérification de la validité de la locale demandée. Mais néanmoins, la requête suivante du navigateur provoque une exception côté serveur car le cookie de locale qu'il reçoit est incorrect.

4.22 [/m27] : vérifier la validité d'un modèle avec Hibernate Validator

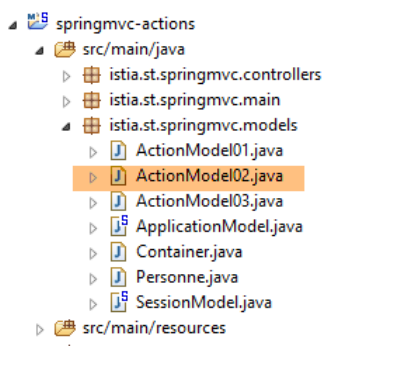
Considérons la nouvelle action suivante :

```
1.     //validation d'un modèle avec Hibernate Validator ------------------------
2.     @RequestMapping(value = "/m27", method = RequestMethod.POST)
3.     public Map<String, Object> m27(@Valid ActionModel02 data, BindingResult result) {
4.        Map<String, Object> map = new HashMap<String, Object>();
5.        // des erreurs ?
6.        if (result.hasErrors()) {
7.           // parcours de la liste des erreurs
8.           for (FieldError error : result.getFieldErrors()) {
9.              map.put(error.getField(),
10.                    String.format("[message=%s, codes=%s]", error.getDefaultMessage(), String.join("|",
    error.getCodes())));
11.          }
12.       } else {
13.          // pas d'erreurs
14.          map.put("data", data);
15.       }
16.       return map;
17. }
```

On a là du code vu maintenant plusieurs fois :

- ligne 3 : l'action [/m27] est demandée via un POST ;
- lignes 8-11, chaque erreur sera caractérisée par [champ, message] avec :
  - champ : le champ erroné,
  - message : le message d'erreur associé ainsi que la liste des codes d'erreur ;
- ligne 14 : s'il n'y a pas d'erreurs, on rend la chaîne jSON des valeurs postées ;

Ligne 3, on utilise le modèle d'action [ActionModel02] suivant :



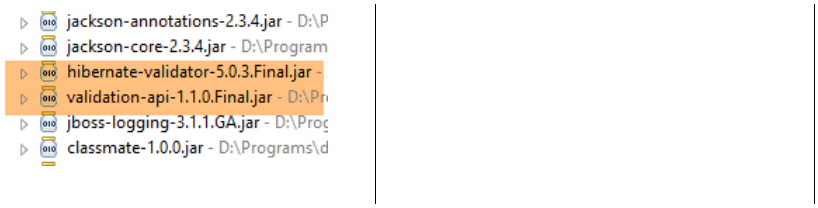

```java
package istia.st.springmvc.models;

import java.util.Date;

import javax.validation.constraints.AssertFalse;
import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.Future;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.NotBlank;
import org.hibernate.validator.constraints.Range;
import org.hibernate.validator.constraints.URL;

public class ActionModel02 {

    @NotNull(message = "La donnée est obligatoire")
    @AssertFalse(message = "Seule la valeur [false] est acceptée")
    private Boolean assertFalse;

    @NotNull(message = "La donnée est obligatoire")
    @AssertTrue(message = "Seule la valeur [true] est acceptée")
    private Boolean assertTrue;

    @NotNull(message = "La donnée est obligatoire")
    @Future(message = "Il faut une date postérieure à aujourd'hui")
    private Date dateInFuture;

    @NotNull(message = "La donnée est obligatoire")
    @Past(message = "Il faut une date antérieure à aujourd'hui")
    private Date dateInPast;

    @NotNull(message = "La donnée est obligatoire")
    @Max(value = 100, message = "Maximum 100")
    private Integer intMax100;

    @NotNull(message = "La donnée est obligatoire")
    @Min(value = 10, message = "Minimum 10")
    private Integer intMin10;

    @NotNull(message = "La donnée est obligatoire")
    @NotBlank(message = "La chaîne doit être non blanche")
    private String strNotBlank;

    @NotNull(message = "La donnée est obligatoire")
    @Size(min = 4, max = 6, message = "La chaîne doit avoir entre 4 et 6 caractères")
    private String strBetween4and6;

    @NotNull(message = "La donnée est obligatoire")
    @Pattern(regexp = "^\\d{2}:\\d{2}:\\d{2}$", message = "Le format doit être hh:mm:ss")
```

```
57.     private String hhmmss;
58.
59.     @NotNull(message = "La donnée est obligatoire")
60.     @Email(message = "Adresse invalide")
61.     private String email;
62.
63.     @NotNull(message = "La donnée est obligatoire")
64.     @Length(max = 4, min = 4, message = "La chaîne doit avoir 4 caractères exactement")
65.     private String str4;
66.
67.     @Range(min = 10, max = 14, message = "La valeur doit être dans l'intervalle [10,14]")
68.     @NotNull(message = "La donnée est obligatoire")
69.     private Integer int1014;
70.
71.     @URL(message = "URL invalide")
72.     private String url;
73.
74.     // getters et setters
75.
76. ...
77. }
```

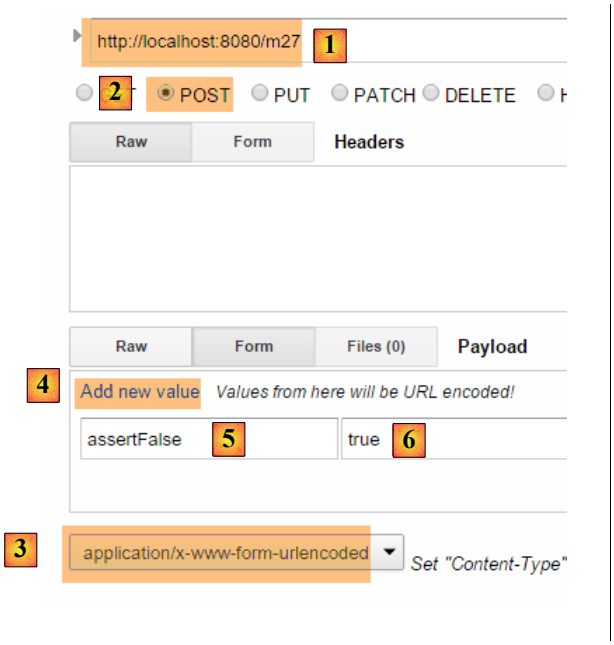
La classe utilise des contraintes de validation issues de deux packages :

- [javax.validation.constraints] aux lignes 5-13 ;
- [org.hibernate.validator.constraints] aux lignes 15-19 ;

Les dépendances Maven de ces deux packages sont présentes dans le projet :



Ici, nous n'allons pas utiliser de messages internationalisés mais des messages définis à l'intérieur de la contrainte avec l'attribut [message]. Pour tester cette action, nous allons utiliser [Advanced Rest Client] :



- en [1-2], la requête POST ;
- en [3], l'entête HTTP [Content-Type] à utiliser ;
- en [4], le lien [Add new value] permet d'ajouter un couple [paramètre, value] ;

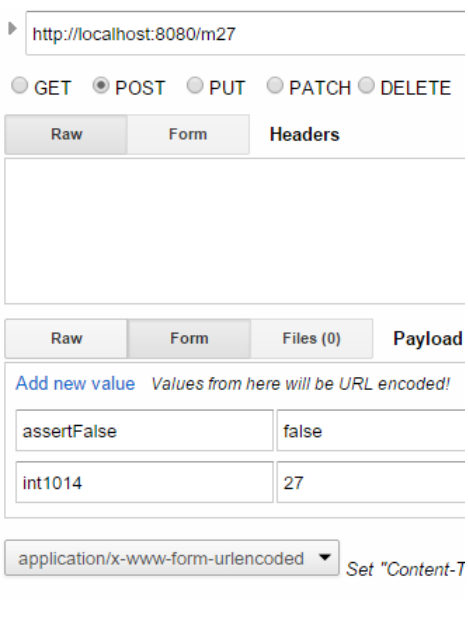
- en [5], mettre un champ de [ActionModel02], ici le champ [assertFalse] :

```
1.      @NotNull(message = "La donnée est obligatoire")
2.      @AssertFalse(message = "Seule la valeur [false] est acceptée")
3.  private Boolean assertFalse;
```

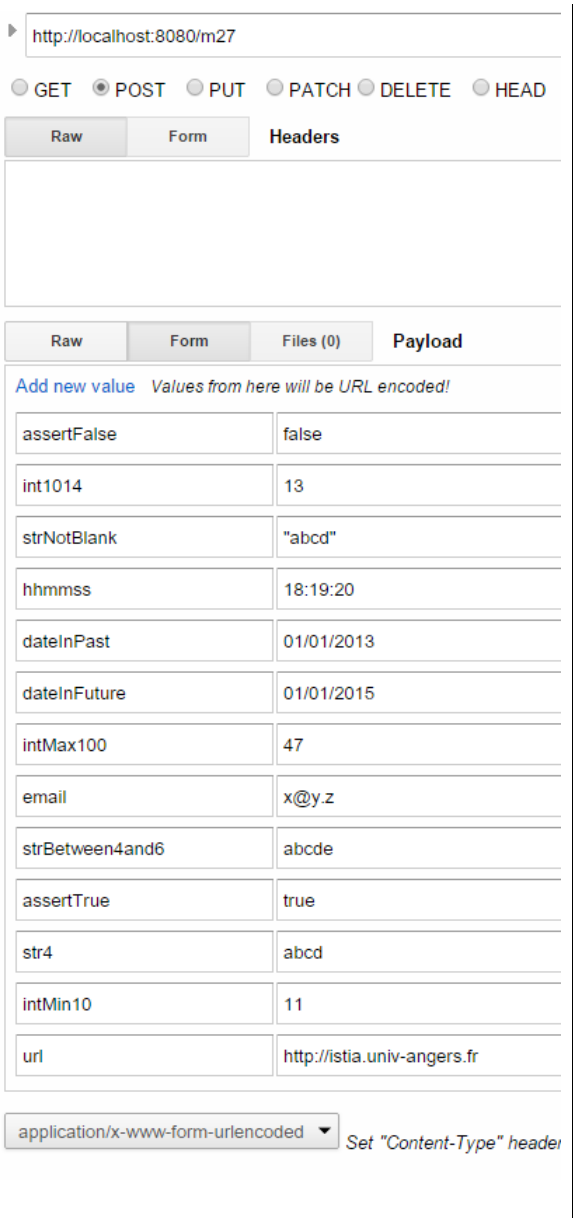
- en [6], mettre une valeur erronée pour voir un message d'erreur. Ci-dessus, la contrainte [@AssertFalse] exige que le champ [assertFalse] ait la valeur [false] ;

- en [7], la réponse du serveur : la contrainte [@NotNull] des champs vides a été déclenchée et le message d'erreur associé, rendu ;
- en [8], le message du champ [assertFalse] pour lequel la contrainte [@AssertFalse] n'était pas vérifiée ainsi que les codes de cette erreur. On rappelle que ces codes peuvent être associés à des messages internationalisés ;

Voici un autre exemple :



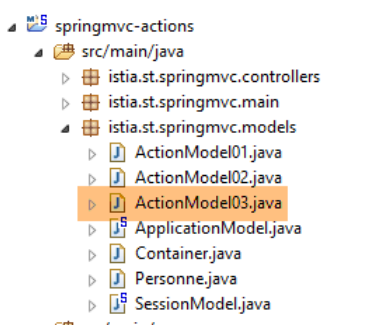
Le lecteur est invité à tester les différents cas d'erreur jusqu'au POST de données toutes valides :



Note : le format des dates est le format anglo-saxon : **mm/jj/aaaa**.

4.23 [/m28] : externalisation des messages d'erreur

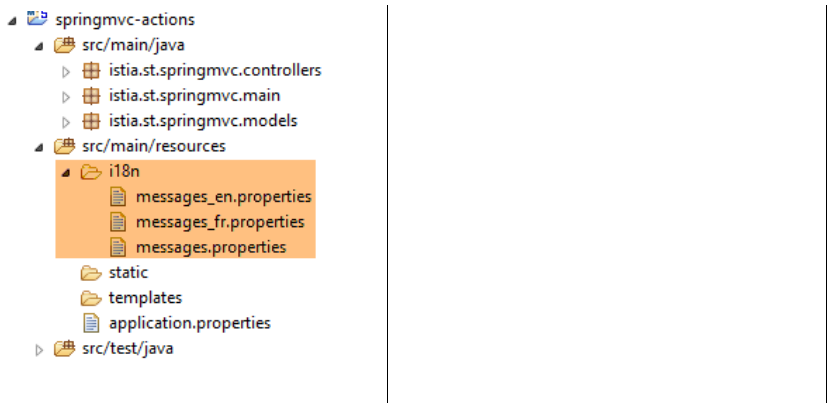
Dans la classe [ActionModel02], nous avons mis les messages en 'dur'. Il est préférable de les externaliser dans des fichiers de messages. Nous suivons l'exemple de l'action [/m25]. Nous créons le nouveau modèle d'action [ActionModel03] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import java.util.Date;
4.
5.  import javax.validation.constraints.AssertFalse;
6.  import javax.validation.constraints.AssertTrue;
7.  import javax.validation.constraints.Future;
8.  import javax.validation.constraints.Max;
9.  import javax.validation.constraints.Min;
10. import javax.validation.constraints.NotNull;
11. import javax.validation.constraints.Past;
12. import javax.validation.constraints.Pattern;
13. import javax.validation.constraints.Size;
14.
15. import org.hibernate.validator.constraints.Email;
16. import org.hibernate.validator.constraints.Length;
17. import org.hibernate.validator.constraints.NotBlank;
18. import org.hibernate.validator.constraints.Range;
19. import org.hibernate.validator.constraints.URL;
20.
21. public class ActionModel03 {
22.
23.     @NotNull
24.     @AssertFalse
25.     private Boolean assertFalse;
26.
27.     @NotNull
28.     @AssertTrue
29.     private Boolean assertTrue;
30.
31.     @NotNull
32.     @Future
33.     private Date dateInFuture;
34.
35.     @NotNull
36.     @Past
37.     private Date dateInPast;
38.
39.     @NotNull
40.     @Max(value = 100)
41.     private Integer intMax100;
42.
43.     @NotNull
44.     @Min(value = 10)
45.     private Integer intMin10;
46.
47.     @NotNull
48.     @NotBlank
49.     private String strNotBlank;
50.
51.     @NotNull
```

```
52.     @Size(min = 4, max = 6)
53.     private String strBetween4and6;
54.
55.     @NotNull
56.     @Pattern(regexp = "^\\d{2}:\\d{2}:\\d{2}$")
57.     private String hhmmss;
58.
59.     @NotNull
60.     @Email
61.     private String email;
62.
63.     @NotNull
64.     @Length(max = 4, min = 4)
65.     private String str4;
66.
67.     @Range(min = 10, max = 14)
68.     @NotNull
69.     private Integer int1014;
70.
71.     @URL
72.     private String url;
73.
74.     // getters et setters
75.         ...
76. }
```

Les messages d'erreur sont externalisés dans les fichiers [messages.properties] :



Le fichier [messages_fr.properties] est le suivant :

```
1.  NotNull=Le champ ne peut être vide
2.  typeMismatch=Format invalide
3.  typeMismatch.actionModel01.a=Le paramètre [a] doit être entier
4.  Range.actionModel03.int1014=La valeur doit être dans l'intervalle [10,14]
5.  NotBlank.actionModel03.strNotBlank=La chaîne doit être non blanche
6.  AssertFalse.actionModel03.assertFalse=Seule la valeur [false] est acceptée
7.  Pattern.actionModel03.hhmmss=Le format doit être hh:mm:ss
8.  Past.actionModel03.dateInPast=Il faut une date antérieure ou égale à celle d'aujourd'hui
9.  Future.actionModel03.dateInFuture=Il faut une date postérieure à celle d'aujourd'hui
10. Length.actionModel03.str4=La chaîne doit avoir 4 caractères exactement
11. Min.actionModel03.intMin10=Minimum 10
12. Max.actionModel03.intMax100=Maximum 100
13. AssertTrue.actionModel03.assertTrue=Seule la valeur [true] est acceptée
14. Email.actionModel03.email=Adresse invalide
15. Size.actionModel03.strBetween4and6=La chaîne doit avoir entre 4 et 6 caractères
16. URL.actionModel03.url=URL invalide
```

Les messages d'erreur ont été ajoutés aux lignes 4-16. Ils sont sous la forme :

```
code=message
```

Les codes ne peuvent être quelconques. Ce sont ceux affichés dans l'action [/m27] précédente. Par exemple :

```
int1014: "[message=La valeur doit être dans l'intervalle [10,14], codes=Range.actionModel02.int1014|Range.int1014|Range.java.lang.Integer|Range]"
```

Dans les fichiers de messages, il faut pour le champ [int1014] utiliser l'un des quatre codes ci-dessus.

Le fichier [messages_en.properties] est le suivant :

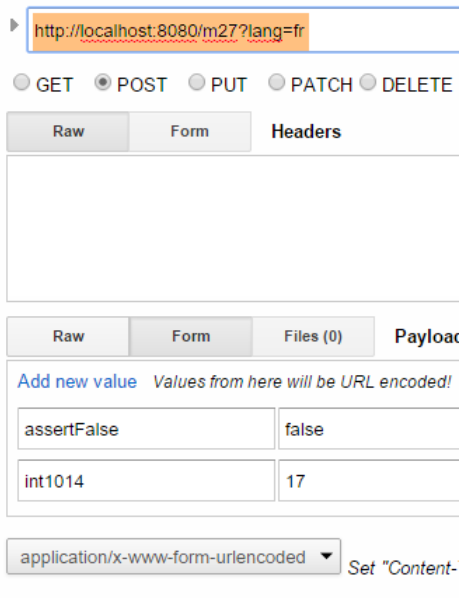
```
1.  NotNull=The field can't be empty
2.  typeMismatch=Invalid format
3.  typeMismatch.actionModel01.a=Parameter [a] must be an integer
4.  Range.actionModel03.int1014=Value must be in [10,14] interval
5.  NotBlank.actionModel03.strNotBlank=String can't be empty
6.  AssertFalse.actionModel03.assertFalse=Only boolean [false] is allowed
7.  Pattern.actionModel03.hhmmss=String format is hh:mm:ss
8.  Past.actionModel03.dateInPast=Date must be before or equal to today's date
9.  Future.actionModel03.dateInFuture=Date must be after today's date
10. Length.actionModel03.str4=String must be four characters long
11. Min.actionModel03.intMin10=Minimum 10
12. Max.actionModel03.intMax100=Maximum 100
13. AssertTrue.actionModel03.assertTrue=Only boolean [true] is allowed
14. Email.actionModel03.email=Invalid email
15. Size.actionModel03.strBetween4and6=String must be between four and six characters long
16. URL.actionModel03.url=Invalid URL
```

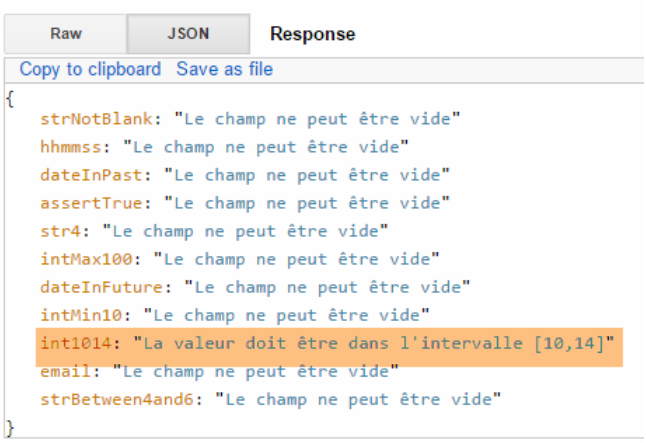
Le modèle d'action [ActionModel03] est exploité par l'action suivante :

```
1.  // ----------------------- externalisation des messages d'erreur ------------------------
2.      @RequestMapping(value = "/m28", method = RequestMethod.POST)
3.      public Map<String, Object> m28(@Valid ActionModel03 data, BindingResult result, HttpServletRequest
    request) {
4.          Map<String, Object> map = new HashMap<String, Object>();
5.          // le contexte de l'application Spring
6.          WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(request.getServletContext());
7.          // locale
8.          Locale locale = RequestContextUtils.getLocale(request);
9.          // des erreurs ?
10.         if (result.hasErrors()) {
11.             for (FieldError error : result.getFieldErrors()) {
12.                 // recherche du msg d'erreur à parir des codes d'erreur
13.                 // le msg est cherché dans les fichiers de messages
14.                 // les codes d'erreur sous forme de tableau
15.                 String[] codes = error.getCodes();
16.                 // sous forme de chaîne
17.                 String listCodes = String.join(" - ", codes);
18.                 // recherche
19.                 String msg = null;
20.                 int i = 0;
21.                 while (msg == null && i < codes.length) {
22.                     try {
23.                         msg = ctx.getMessage(codes[i], null, locale);
24.                     } catch (Exception e) {
25.
26.                     }
27.                     i++;
28.                 }
29.                 // a-t-on trouvé ?
30.                 if (msg == null) {
31.                     msg = String.format("Indiquez un message pour l'un des codes [%s]", listCodes);
32.                 }
33.                 // on a trouvé - on ajoute l'erreur au dictionnaire
34.                 map.put(error.getField(), msg);
35.             }
36.         } else {
37.             // pas d'erreurs
38.             map.put("data", data);
39.         }
40.         return map;
41.     }
```

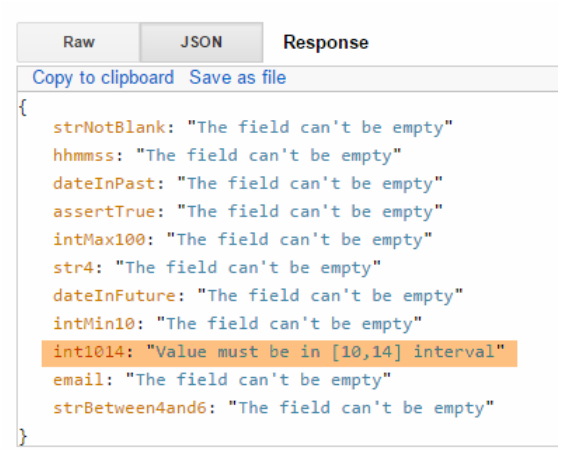
On a déjà commenté ce type de code. La seule chose réellement importante est la ligne 23 : le message d'erreur récupéré dépend de la locale de la requête.

Voici un exemple en français :



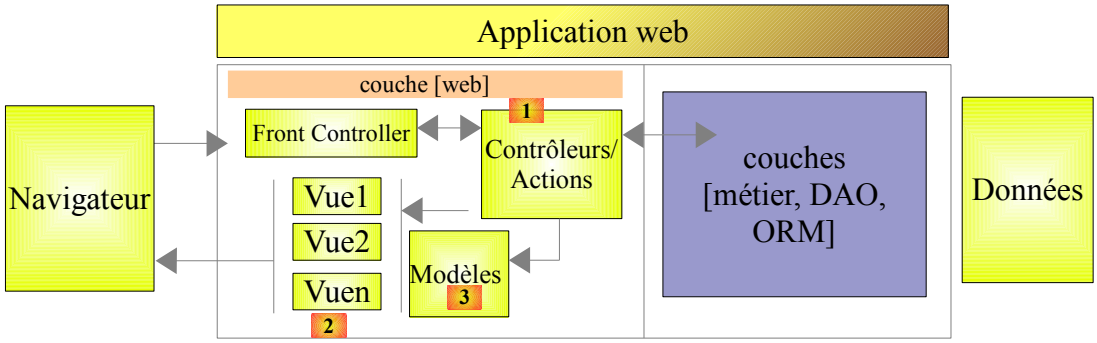


et maintenant en anglais :



5 Les vues Thymeleaf

Revenons à l'architecture d'une application Spring MVC.



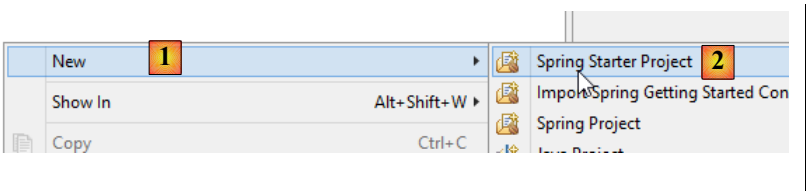
Les deux chapitres précédents ont décrit divers aspects du bloc [1], les **actions**. Nous abordons maintenant :

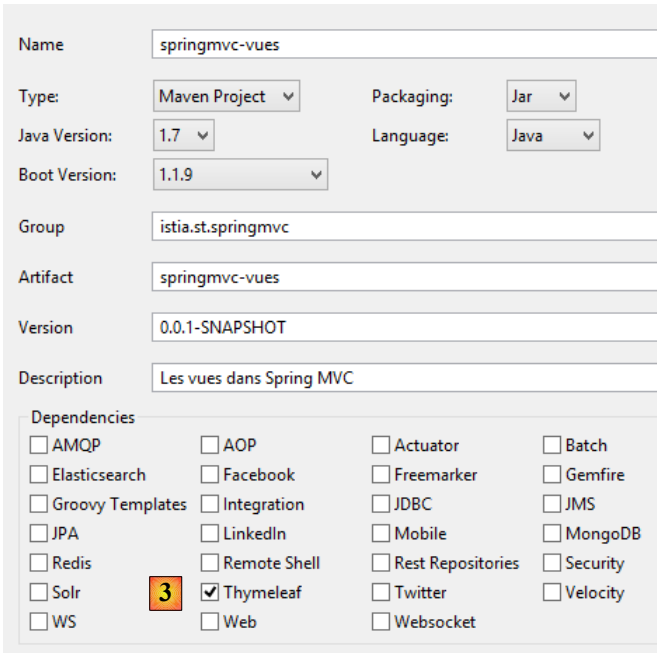
- le bloc [2] des vues V ;
- le bloc [3] du modèle M affiché par ces vues ;

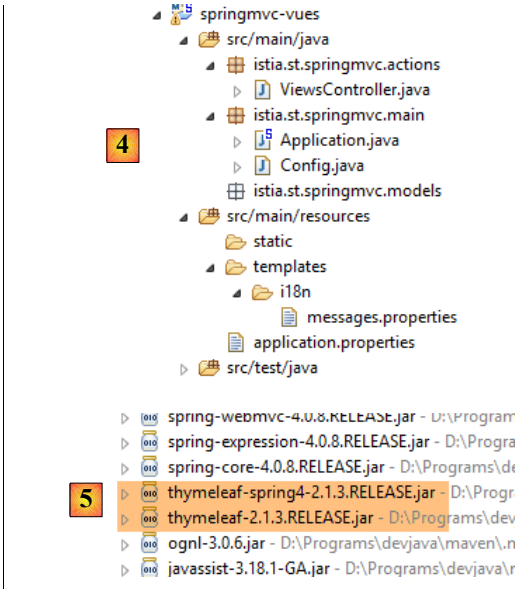
Depuis la création de Spring MVC, la technologie de génération des pages HTML envoyées aux navigateurs client était celle des pages JSP (Java Server Pages). Depuis quelques années, la technologie [Thymeleaf] [http://www.thymeleaf.org/] peut être également utilisée. C'est elle que nous présentons maintenant.

5.1 Le projet STS

Nous créons un nouveau projet :

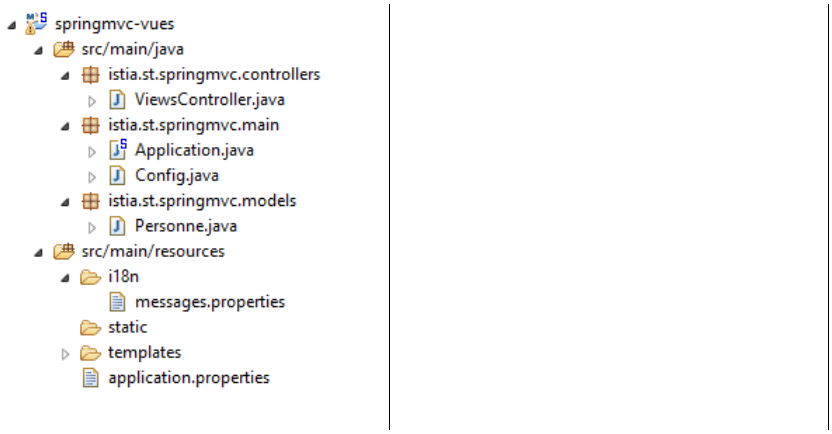






- en [3], indiquer que le projet a besoin des dépendances [Thymeleaf]. Cela amènera en plus des dépendances [Spring MVC] du projet précédent, celles du framework [Thymeleaf] [5] ;

Maintenant, faisons évoluer ce projet de la façon suivante :



Nous nous inspirons du projet précédent :

- [istia.st.springmvc.controllers] contiendra les contrôleurs ;
- [istia.st.springmvc.models] contiendra les modèles des actions et des vues ;
- [istia.st.springmvc.main] est le package de la classe exécutable Spring Boot ;
- [templates] contiendra les vues Thymeleaf ;
- [i18n] contiendra les messages internationalisés affichés par les vues ;

La classe [Application] est la suivante :

```
1.  package istia.st.springmvc.main;
2.
3.  import org.springframework.boot.SpringApplication;
4.
5.  public class Application {
6.
7.     public static void main(String[] args) {
8.        SpringApplication.run(Config.class, args);
9.     }
10. }
```

La classe [Config] est la suivante :

```
1.  package istia.st.springmvc.main;
2.
3.  import java.util.Locale;
4.
5.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
6.  import org.springframework.context.MessageSource;
7.  import org.springframework.context.annotation.Bean;
8.  import org.springframework.context.annotation.ComponentScan;
9.  import org.springframework.context.annotation.Configuration;
10. import org.springframework.context.support.ResourceBundleMessageSource;
11. import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
12. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
13. import org.springframework.web.servlet.i18n.CookieLocaleResolver;
14. import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
15.
16. @Configuration
17. @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
18. @EnableAutoConfiguration
19. public class Config extends WebMvcConfigurerAdapter {
20.    @Bean
21.    public MessageSource messageSource() {
22.       ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
23.       messageSource.setBasename("i18n/messages");
24.       return messageSource;
25.    }
```

```
26.
27.     @Bean
28.     public LocaleChangeInterceptor localeChangeInterceptor() {
29.         LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
30.         localeChangeInterceptor.setParamName("lang");
31.         return localeChangeInterceptor;
32.     }
33.
34.     @Override
35.     public void addInterceptors(InterceptorRegistry registry) {
36.         registry.addInterceptor(localeChangeInterceptor());
37.     }
38.
39.     @Bean
40.     public CookieLocaleResolver localeResolver() {
41.         CookieLocaleResolver localeResolver = new CookieLocaleResolver();
42.         localeResolver.setCookieName("lang");
43.         localeResolver.setDefaultLocale(new Locale("fr"));
44.         return localeResolver;
45.     }
46. }
```

Cette configuration permet pour l'instant la gestion des locales.

Le contrôleur [ViewController] est le suivant :

```
1.  package istia.st.springmvc.actions;
2.
3.  import org.springframework.stereotype.Controller;
4.
5.  @Controller
6.  public class ViewsController {
7.
8.  }
```

- ligne 5, l'annotation [@Controller] a remplacé l'annotation [@RestController] car désormais, les actions ne vont pas générer la réponse au client. Elles vont :
  - construire un modèle M
  - rendre un type [String] qui sera le nom de la vue [Thymeleaf] chargée d'afficher ce modèle. C'est la combinaison de cette vue V et de ce modèle M qui va générer le flux HTML envoyé au client ;

Le fichier [messages.properties] est pour l'instant vide.

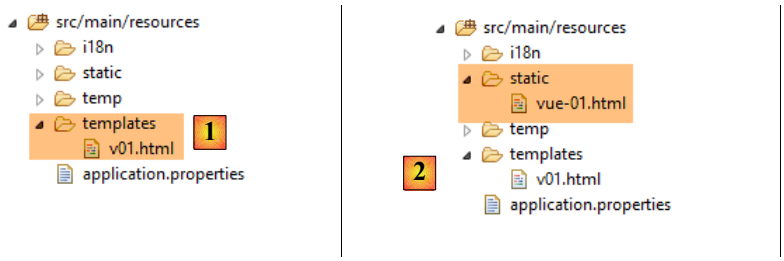
5.2 [/v01] : les bases de Thymeleaf

Nous considérons la première action suivante dans [ViewsController] :

```
1.      // les bases de Thymeleaf - 1
2.      @RequestMapping(value = "/v01", method = RequestMethod.GET)
3.      public String v01() {
4.          return "v01";
5.  }
```

- ligne 3 : l'action rend un type [String]. Ce sera le nom de l'action ;
- ligne 4 : cette vue sera [v01]. Par défaut, elle doit se trouver dans le dossier [templates] et s'appeler [v01.html] ;

La vue [v01.html] est la suivante :

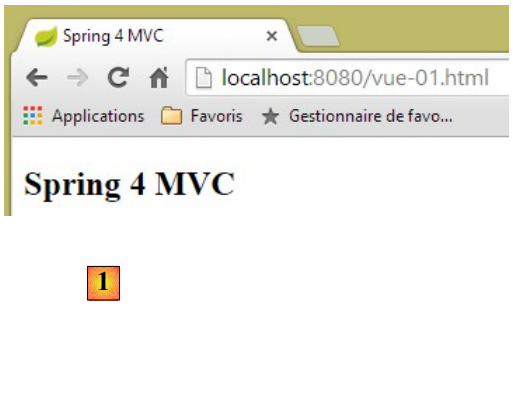


```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title th:text="'Les vues'">Spring 4 MVC</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.     <h2 th:text="'Les vues dans Spring MVC'">Spring 4 MVC</h2>
9.  </body>
10. </html>
```

C'est un fichier HTML. La présence de Thymeleaf se voit :

- à l'espace de noms [th] de la ligne 2 ;
- aux attributs [th:text] des lignes 4 et 8 ;

On a là un fichier HTML valide qui peut être visualisé. Nous le mettons dans le dossier [static] [2] sous le nom [vue-01.html] et nous le demandons directement avec un navigateur :



Si nous examinons le code source de la page en [2], nous pouvons constater que les attributs [th:text] ont été envoyés par le serveur et été ignorés par le navigateur. Lorsqu'une vue est le résultat d'une action, Thymeleaf entre en oeuvre et interprète les attributs [th] avant l'envoi de la réponse au client.

La balise HTML :

```
<title th:text="'Les vues'">Spring 4 MVC</title>
```

est traitée de la façon suivante par Thymeleaf :

- **th:text** a la syntaxe **th:text="expression"** où expression est une expression à évaluer. Lorsque cette expression est une chaîne de caractères comme ici, il faut entourer celle-ci par des apostrophes ;
- la valeur de [expression] remplace le texte de la balise HTML, ici le texte de la balise [title] ;

Après traitement, la balise ci-dessus est devenue :

```
<title>Les vues</title>
```

Demandons l'action [/v01] :

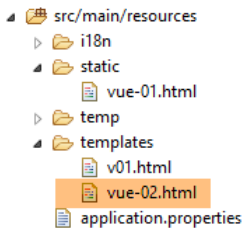
- en [2], on voit le travail de remplacement fait par Thymeleaf ;

Maintenant demandons l'URL [http://localhost:8080/v01.html] :

Comment faut-il interpréter cela ? La vue [templates/v01.html] a-t-elle été servie directement sans passer par une action ? Pour éclaircir les choses, nous créons l'action [/v02] suivante :

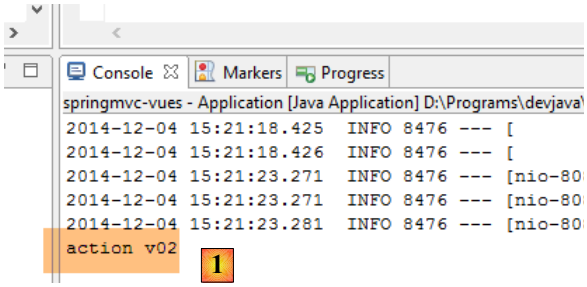
```
1.      // les bases de Thymeleaf - 2
2.      @RequestMapping(value = "/v02", method = RequestMethod.GET)
3.      public String v02() {
4.          System.out.println("action v02");
5.          return "vue-02";
6.  }
```

La vue [vue-02.html] est une copie de [v01.html] :



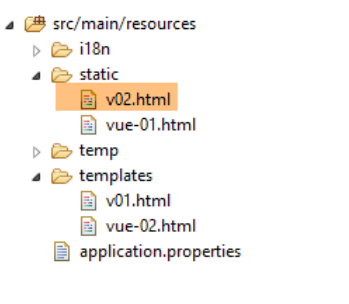
Maintenant demandons l'URL [http://localhost:8080/vue-02.html] :

L'URl n'a pas été trouvée. Maintenant demandons l'URL [http://localhost:8080/v02.html]



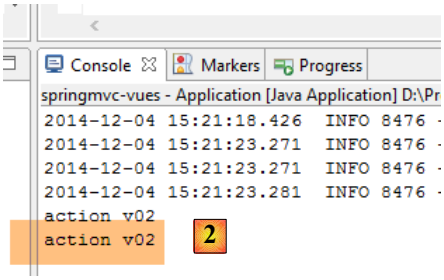
- dans les logs console en [1], on voit que l'action [/v02] a été appelée, et celle-ci a fait afficher la vue [vue-02.html] en [2] ;

Maintenant on sait que l'URL [http://localhost:8080/v02.html] peut désigner également un fichier [/v02.html] dans le dossier [static]. Que se passe-t-il si ce fichier existe ? Nous essayons. Nous créons dans le dossier [static] le fichier [v02.html] suivant :




```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title>Spring 4 MVC</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.     <h2>Spring 4 MVC</h2>
9.  </body>
10. </html>
```

puis nous demandons l'URL [http://localhost:8080/v02.html] :



[1] et [2] montrent que c'est l'action [/v02] qui a été appelée. On retiendra donc que lorsque l'URL demandée est de la forme [**/x.html**], Spring / Thymeleaf :

- exécute l'action [/x] si elle existe ;
- sert la page [/static/x.html] si elle existe ;
- lance une exception 404 Not found sinon ;

Pour éviter des confusions, à partir de maintenant, les actions et les vues n'auront pas les mêmes noms.

5.3 [/v03] : internationalisation des vues

L'intégration Spring / Thymeleaf permet à Thymeleaf d'utiliser les fichiers de messages de Spring. Considérons la nouvelle action [/v03] suivante :

```
1.     // internationalisation des vues
2.     @RequestMapping(value = "/v03", method = RequestMethod.GET)
3.     public String v03() {
4.        return "vue-03";
5.  }
```

Elle fait afficher la vue [vue-03.html] suivante :


```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.  <head>
4.  <title th:text="#{title}">Spring 4 MVC</title>
5.  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.  </head>
7.  <body>
8.     <h2 th:text="#{title}">Spring 4 MVC</h2>
9.  </body>
10. </html>
```

Aux lignes 4 et 8, l'expression de l'attribut [th:text] est **#{title}** dont la valeur est le message de clé [title]. Nous créons les fichiers [messages_fr.properties] et [messages_en.properties] suivants :

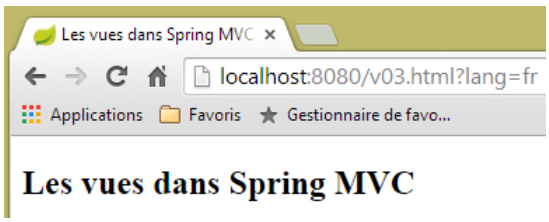
[messages_fr.properties]

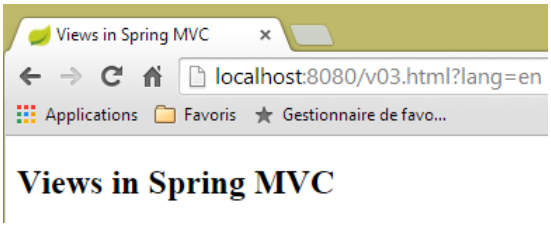
```
title=Les vues dans Spring MVC
```

[messages_en.properties]

```
title=Views in Spring MVC
```

Demandons les URL [http://localhost:8080/v03.html?lang=fr] et [http://localhost:8080/v03.html?lang=en] :





Remarquons que nous avons utilisé ce que nous avons appris récemment. Plutôt que de désigner l'action [v03] par [/v03], nous l'avons désigné par [/v03.html].

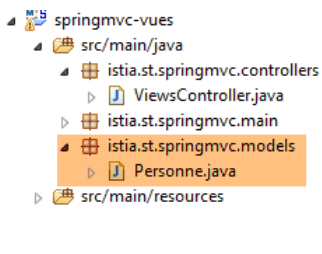
5.4 [/v04] : création du modèle M d'une vue V

Considérons la nouvelle action [/v04] suivante :

```
1.      // création du modèle M d'une vue V
2.      @RequestMapping(value = "/v04", method = RequestMethod.GET)
3.      public String v04(Model model) {
4.          model.addAttribute("personne", new Personne(7, "martin", 17));
5.          System.out.println(String.format("Modèle=%s", model));
6.          return "vue-04";
7.  }
```

- ligne 4 : le modèle de la vue est injecté dans les paramètres de l'action. Par défaut, ce modèle initial est vide. On verra qu'il est possible de le pré-remplir ;
- ligne 4 : un modèle de type [Model] est une sorte de dictionnaire d'éléments de type <String, Object>. Ligne 4, nous ajoutons une entrée dans ce dictionnaire avec la clé [personne] associée à une valeur de type [Personne] ;
- ligne 5 : on affiche sur la console le modèle pour voir à quoi il ressemble ;
- ligne 6 : on fait afficher la vue [vue-04.html] ;

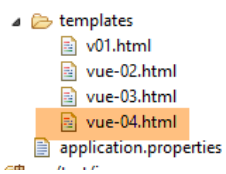
La classe [Personne] est celle utilisée dans le chapitre précédent :




```
1.  package istia.st.springmvc.models;
2.
3.  public class Personne {
4.
5.      // identifiant
6.      private Integer id;
7.      // nom
8.      private String nom;
9.      // âge
10.     private int age;
11.
12.     // constructeurs
13.     public Personne() {
14.
15.     }
```

```
16.
17.     public Personne(String nom, int age) {
18.         this.nom = nom;
19.         this.age = age;
20.     }
21.
22.     public Personne(Integer id, String nom, int age) {
23.         this(nom, age);
24.         this.id = id;
25.     }
26.
27.     @Override
28.     public String toString() {
29.         return String.format("[id=%s, nom=%s,  age=%d]", id, nom, age);
30.     }
31.
32.     // getters et setters
33. ...
34. }
```

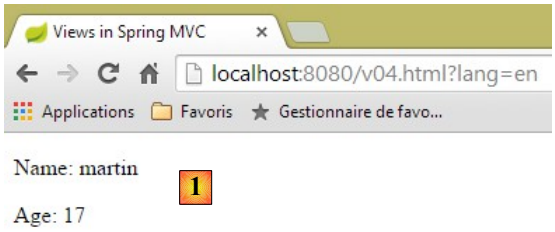
La vue [vue-04.html] est la suivante :

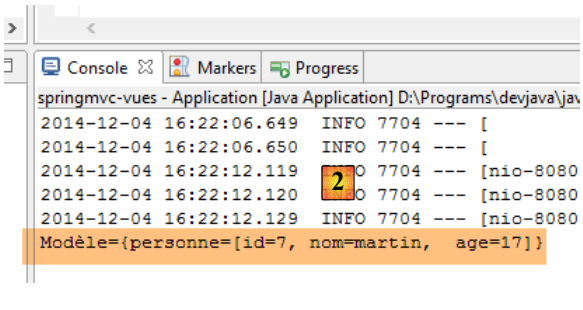



```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title th:text="#{title}">Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.      </head>
7.      <body>
8.          <p>
9.              <span th:text="#{personne.nom}">Nom :</span>
10.             <span th:text="${personne.nom}">Bill</span>
11.         </p>
12.         <p>
13.             <span th:text="#{personne.age}">Age :</span>
14.             <span th:text="${personne.age}">56</span>
15.         </p>
16.     </body>
17. </html>
```

- la ligne 10, introduit un nouveau type d'expression Thymeleaf **${var}** où **var** est une **clé** du modèle M de la vue. On se rappelle que l'action [/v04] a mis dans le modèle une clé [personne] associée à un type Personne[id, nom, age] ;
- ligne 10 : affiche le nom de la personne présente dans le modèle ;
- ligne 14 : affiche son âge ;

Les fichiers de messages sont modifiés pour ajouter les clés [personne.nom] et [personne.age] des lignes 9 et 13. Le résultat est le suivant :



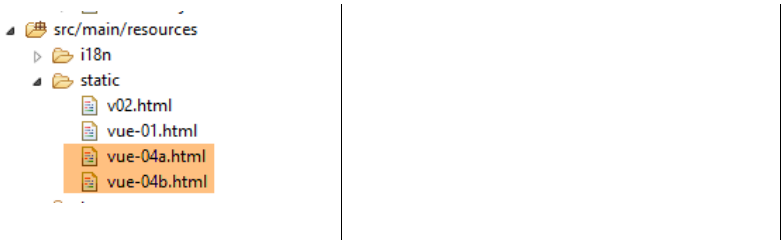


et on trouve la nature du modèle M dans les logs de la console [2].

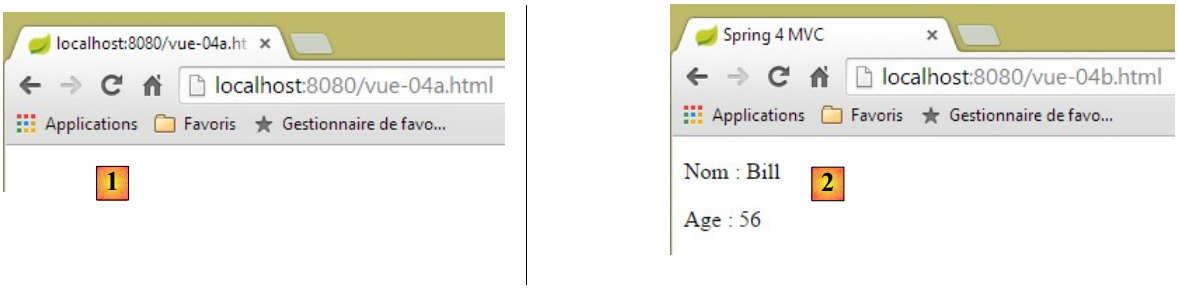
On peut se demander pourquoi on n'écrit pas la vue [vue-04] de la façon suivante :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
   <head>
      <title th:text="#{title}"></title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
   </head>
   <body>
      <p>
         <span th:text="#{personne.nom}" /></span>
         <span th:text="${personne.nom}"></span>
      </p>
      <p>
         <span th:text="#{personne.age}"></span>
         <span th:text="${personne.age}"></span>
      </p>
   </body>
</html>
```

Cette vue est parfaitement licite et donnera le même résultat que précédemment. L'un des objectifs de Thymeleaf est que la page Thymeleaf puisse être affichée même si elle ne passe pas dans les mains de Thymeleaf. Ainsi, créons deux nouvelles pages statiques :



La vue [vue-04b.html] est une copie de la vue [vue-04.html]. Il en est de même pour la vue [vue-04a.html] mais on a enlevé les textes statiques de la page. Si nous visualisons les deux pages, on a les résultats suivants :



Dans le cas [1], la structure de la page n'apparaît pas alors que dans le cas [2] elle est bien visible. Voilà l'intérêt de mettre des textes statiques dans une vue Thymeleaf même si à l'exécution ils vont être remplacés par d'autres textes.

Maintenant, regardons un détail technique. Dans la vue [vue-04.html], nous mettons le code en forme par [ctrl-Maj-F]. Nous obtenons le résultat suivant :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<title th:text="#{title}">Spring 4 MVC</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
   <p>
      <span th:text="#{personne.nom}">Nom :</span> <span
         th:text="${personne.nom}">Bill</span>
   </p>
```

```
12.     <p>
13.        <span th:text="#{personne.age}">Age :</span> <span
14.           th:text="${personne.age}">56</span>
15.     </p>
16. </body>
17. </html>
```

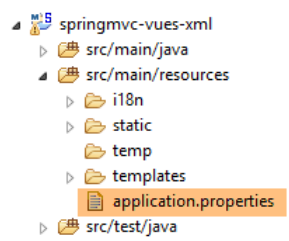
Les balises sont mal alignées et le code devient plus difficile à lire. Si nous renommons [vue-04.html] en [vue-04.xml] et que nous reformatons le code, alors les balises redeviennent alignées. Donc le suffixe [xml] serait plus pratique. Il est possible de travailler avec ce suffixe. Il faut pour cela configurer Thymeleaf. Pour ne pas défaire ce que nous avons fait, nous dupliquons le projet [springmvc-vues] étudié en un projet [springmvc-vues-xml]

Nous modifions le fichier [pom.xml] de la façon suivante :

```
1.     <groupId>istia.st.springmvc</groupId>
2.     <artifactId>springmvc-vues-xml</artifactId>
3.     <version>0.0.1-SNAPSHOT</version>
4.     <packaging>jar</packaging>
5.
6.     <name>springmvc-vues-xml</name>
7.  <description>Les vues dans Spring MVC</description>
```

Le nom du projet est changé aux lignes 2 et 6. Par ailleurs, nous changeons le suffixe des vues présentes dans le dossier [templates] :

Le document [http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html] liste les propriétés de configuration de Spring Boot utilisables dans le fichier [application.properties] :



Ce document donne les propriétés que Spring Boot utilise lorsqu'il fait de l'autoconfiguration et qu'on peut modifier en faisant une configuration différente dans [application.properties]. Pour Thymeleaf, les propriétés d'autoconfiguration sont les suivantes :

```
# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html # ;charset=<encoding> is added
spring.thymeleaf.cache=true # set to false for hot refresh
```

On pourrait donc se contenter de mette la ligne

```
spring.thymeleaf.suffix=.xml
```

dans [application.properties]. Nous allons suivre une autre voie, celle de la configuration par programmation. Nous allons configurer Thymeleaf dans la classe [Config] :

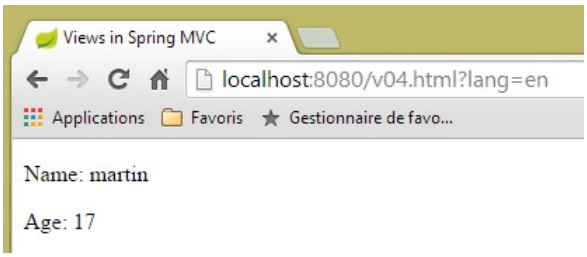
```
package istia.st.springmvc.main;

import java.util.Locale;

...
import org.thymeleaf.spring4.SpringTemplateEngine;
import org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver;

@Configuration
@ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
@EnableAutoConfiguration
public class Config extends WebMvcConfigurerAdapter {
   ...

   @Bean
   public SpringResourceTemplateResolver templateResolver() {
      SpringResourceTemplateResolver templateResolver = new
SpringResourceTemplateResolver();
      templateResolver.setPrefix("classpath:/templates/");
      templateResolver.setSuffix(".xml");
      templateResolver.setTemplateMode("HTML5");
      templateResolver.setCharacterEncoding("UTF-8");
      templateResolver.setCacheable(true);
      return templateResolver;
   }

   @Bean
   SpringTemplateEngine templateEngine(SpringResourceTemplateResolver templateResolver) {
      SpringTemplateEngine templateEngine = new SpringTemplateEngine();
      templateEngine.setTemplateResolver(templateResolver);
      return templateEngine;
   }

}
```

- les lignes 16-24 configurent un [TemplateResolver] pour Thymeleaf. C'est cet objet qui est chargé à partir d'un nom de vue délivré par une action, de trouver le fichier correspondant ;
- lignes 18 et 19 fixent le préfixe et le suffixe à ajouter au nom de la vue pour trouver le fichier. Ainsi si le nom de la vue est [vue04], le fichier cherché sera [classpath:/templates/vue04.xml]. [classpath:/templates] est une syntaxe Spring qui désigne un dossier [/templates] placé à la racine du Classpath du projet ;
- ligne 21 : pour que dans la réponse faite au client on ait l'entête HTTP :

```
Content-Type:text/html;charset=UTF-8
```

- ligne 20 : indique que la vue respecte la norme HTML5 ;
- ligne 22 : indique que les vues Thymeleaf peuvent être mises en cache ;
- lignes 26-31 : fixe le moteur de résolution des vues du couple Spring / Thymeleaf avec le moteur de résolution précédent ;

Lançons l'exécutable de ce nouveau projet et demandons l'URL [http://localhost:8080/v04.html?lang=en] :



On remarque que dans l'URL, l'action [/v04] a pu être remplacée là encore par [v04.html].

5.5 [/v05] : factorisation d'un objet dans une vue Thymeleaf

Nous créons l'action [/v05] suivante :

```
1.     // création du modèle M d'une vue V - 2
2.     @RequestMapping(value = "/v05", method = RequestMethod.GET)
3.     public String v05(Model model) {
4.         model.addAttribute("personne", new Personne(7, "martin", 17));
5.         return "vue-05";
6.  }
```

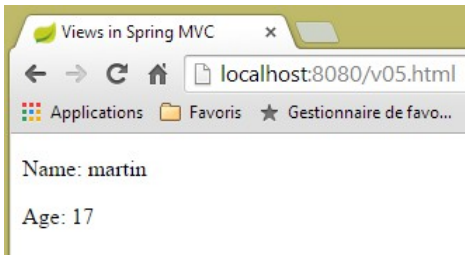
Elle est identique à l'action [/v04]. La vue [vue-05.xml] est la suivante :


```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${personne}">
9.           <p>
10.             <span th:text="#{personne.nom}">Nom :</span>
```

```
11.             <span th:text="*{nom}">Bill</span>
12.          </p>
13.          <p>
14.             <span th:text="#{personne.age}">Age :</span>
15.             <span th:text="*{age}">56</span>
16.          </p>
17.       </div>
18.    </body>
19. </html>
```

- lignes 8-17 : à l'intérieur de ces lignes un objet Thymeleaf est défini par l'attribut [th:object="${personne}"] (ligne 8). Cet objet est ici l'objet de clé [personne] qui est dans le modèle :
- ligne 11 : l'expression Thymeleaf [*{nom}] est équivalente à [${objet.nom}] où [objet] est l'objet Thymeleaf courant. Donc ici l'expression [*{nom}] est équivalente à [${personne.nom}] ;
- ligne 15 : idem ;

Le résultat :



5.6 [/v06] : les tests dans une vue Thymeleaf

Considérons l'action [/v06] suivante :

```
1.     // création du modèle M d'une vue V - 3
2.     @RequestMapping(value = "/v06", method = RequestMethod.GET)
3.     public String v06(Model model) {
4.        model.addAttribute("personne", new Personne(7, "martin", 17));
5.        return "vue-06";
6.  }
```

Elle est identique aux deux précédentes actions. Elle affiche la vue [vue-06.xml] suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${personne}">
9.           <p>
10.             <span th:text="#{personne.nom}">Nom :</span>
11.             <span th:text="*{nom}">Bill</span>
12.          </p>
13.          <p>
14.             <span th:text="#{personne.age}">Age :</span>
15.             <span th:text="*{age}">56</span>
16.          </p>
17.          <p th:if="*{age} >= 18" th:text="#{personne.majeure}">Vous êtes majeur</p>
18.          <p th:if="*{age} &lt; 18" th:text="#{personne.mineure}">Vous êtes mineur</p>
19.       </div>
20.    </body>
21. </html>
```

- ligne 17 : l'attribut [th:if] évalue une expression booléenne. Si cette expression est vraie, la balise est affichée sinon elle ne l'est pas. Donc ici si ${personne.age}>=18, le texte [#{*personne.majeure*}] sera affiché, ç-à-d le message de clé [personne.majeure] dans les fichiers de messages ;
- ligne 18 : on ne peut pas écrire [*{*age*} < *18*] car le signe < est un caractère réservé. Il faut donc utiliser son équivalent HTML [<] appelé également entité HTML [http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references];

Les fichiers de messages sont modifiés :

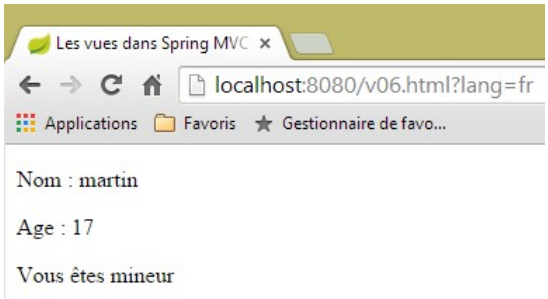
[messages_fr.properties]

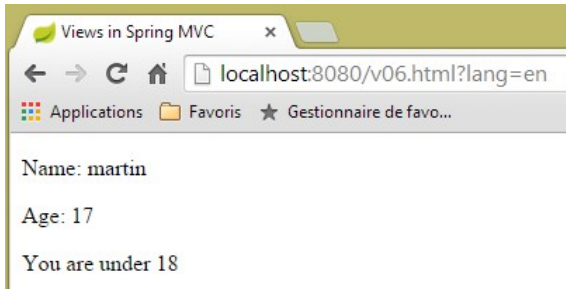
```
title=Les vues dans Spring MVC
personne.nom=Nom :
personne.age=Age :
personne.mineure=Vous êtes mineur
personne.majeure=Vous êtes majeur
```

[messages_en.properties]

```
title=Views in Spring MVC
personne.nom=Name:
personne.age=Age:
personne.mineure=You are under 18
personne.majeure=You are over 18
```

Le résultat est le suivant :





5.7 [/v07] : itération dans une vue Thymeleaf

Considérons l'action [/v07] suivante :

```
1.      // création du modèle M d'une vue V - 4
2.      @RequestMapping(value = "/v07", method = RequestMethod.GET)
3.      public String v07(Model model) {
4.          model.addAttribute("liste", new Personne[] { new Personne(7, "martin", 17), new Personne(8,
    "lucie", 32),
5.                  new Personne(9, "paul", 7) });
6.          return "vue-07";
7.  }
```

- l'action crée une liste de trois personnes, la met dans le modèle associée à la clé [liste] et fait afficher la vue [vue-07] ;

La vue [vue-07.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title th:text="#{title}">Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.      </head>
7.      <body>
8.          <h3 th:text="#{liste.personnes}">Liste de personnes</h3>
```

```
9.          <ul>
10.            <li th:each="element : ${liste}" th:text="'['+ ${element.id} + ', ' +${element.nom}+ ', ' + ${element.age} + ']'">[id,nom,age]</li>
11.         </ul>
12.      </body>
13. </html>
```

- ligne 10 : l'attribut [th:each] répète la balise dans laquelle elle se trouve, ici une balise <li>. Elle a ici deux paramètres [element : collection] où [collection] est une collection d'objets, ici une liste de personnes. Thymeleaf va parcourir la collection et générer autant de balises <li> qu'il y a d'éléments dans la collection. Pour chaque balise <li> [element] va représenter l'élément de la collection attaché à la balise. Pour cet élément, l'attribut [th:text] va être évalué. Son expression est ici une concaténation de chaînes pour avoir le résultat [id, nom, age] ;
- ligne 8 : on ajoute la clé [*liste.personnes*] dans les fichiers de messages ;

Voici le résultat :

5.8 [/v08-/v10] : @ModelAttribute

Nous revenons sur quelque chose que nous avons vu lors de l'étude des actions, le rôle de l'annotation [@ModelAttribute]. Nous ajoutons la nouvelle action suivante :

```
1.     // --------------- Binding et ModelAttribute ---------------------------------
2.
3.     // si le paramètre est un objet, il est instancié et éventuellement modifié par les paramètres de la requête
4.     // il fera automatiquement partie du modèle de la vue avec la clé [key]
5.     // pour @ModelAttribute("xx") paramètre, key sera égal à xx
6.     // pour @ModelAttribute paramètre, key sera égal au nom de la classe du paramètre commençant avec une minuscule
7.     // si @ModelAttribute est absent, alors tout se passe comme s'il était présent sans clé
8.     // on notera que cette présence automatique dans le modèle n'est pas effectuée si le paramètre n'est pas un objet
9.
10.    @RequestMapping(value = "/v08", method = RequestMethod.GET)
11.    public String v08(@ModelAttribute("someone") Personne p, Model model) {
12.       System.out.println(String.format("Modèle=%s", model));
13.       return "vue-08";
14. }
```

- ligne 11 : l'annotation [@ModelAttribute("someone")] va automatiquement ajouter l'objet [Personne p] dans le modèle, associé à la clé [someone] ;
- ligne 12 : pour vérifier le modèle ;
- ligne 13 : affiche la vue [vue-08.xml] ;

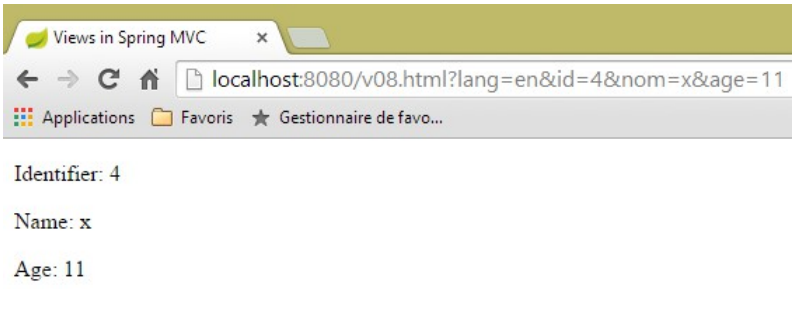
La vue [vue-08.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```
6.      </head>
7.      <body>
8.         <div th:object="${someone}">
9.            <p>
10.              <span th:text="#{personne.id}">Id :</span>
11.              <span th:text="*{id}">14</span>
12.           </p>
13.           <p>
14.              <span th:text="#{personne.nom}">Nom :</span>
15.              <span th:text="*{nom}">Bill</span>
16.           </p>
17.           <p>
18.              <span th:text="#{personne.age}">Age :</span>
19.              <span th:text="*{age}">56</span>
20.           </p>
21.        </div>
22.     </body>
23. </html>
```

- ligne 8 : l'objet Thymeleaf est initialisé avec l'objet de clé [someone] ;

Le résultat est le suivant :



et dans la console, on a le log suivant :

```
Modèle={someone=[id=4, nom=x,  age=11],
org.springframework.validation.BindingResult.someone=org.springframework.validation.BeanPropertyBindingResult: 0
errors}
```

Considérons maintenant l'action [/v09] suivante :

```
1.      @RequestMapping(value = "/v09", method = RequestMethod.GET)
2.      public String v09(Personne p, Model model) {
3.         System.out.println(String.format("Modèle=%s", model));
4.         return "vue-09";
5.   }
```

- ligne 1 : la présence du paramètre [Personne p] va automatiquement mettre la personne [p] dans le modèle. Comme il n'est pas précisé de clé, la clé utilisée est le nom de la classe avec son premier caractère en minuscule. Donc [Personne p] est équivalent à [@ModelAttribute("personne") Personne p] ;

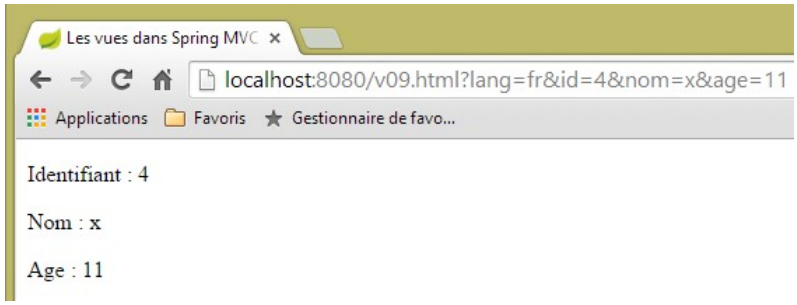
La vue [vue.09.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${personne}">
9.           <p>
10.             <span th:text="#{personne.id}">Id :</span>
11.             <span th:text="*{id}">14</span>
12.          </p>
13.          <p>
```

```
14.             <span th:text="#{personne.nom}">Nom :</span>
15.             <span th:text="*{nom}">Bill</span>
16.          </p>
17.          <p>
18.             <span th:text="#{personne.age}">Age :</span>
19.             <span th:text="*{age}">56</span>
20.          </p>
21.       </div>
22.    </body>
23. </html>
```

- ligne 8 : la clé de modèle utilisée est [personne] ;

Voici un résultat :



et le log dans la console du serveur :

```
Modèle={personne=[id=4, nom=x,  age=11],
org.springframework.validation.BindingResult.personne=org.springframework.validation.BeanPropertyBindingResult: 0
errors}
```

Maintenant, considérons la nouvelle action [/v10] suivante :

```
1.     @ModelAttribute("uneAutrePersonne")
2.     private Personne getPersonne(){
3.        return new Personne(24,"pauline",55);
4.     }
5.
6.     @RequestMapping(value = "/v10", method = RequestMethod.GET)
7.     public String v10(Model model) {
8.        System.out.println(String.format("Modèle=%s", model));
9.        return "vue-10";
10. }
```

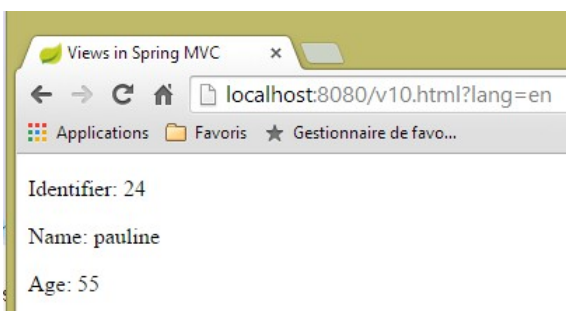
- lignes 1-4 : définissent une méthode créant dans le modèle **de chaque requête** un élément de clé [uneAutrePersonne] associé à l'objet [**new** Personne(24,"pauline",55)] ;
- lignes 6-10 : l'action [/v10] ne fait rien si ce n'est de passer le modèle qu'elle reçoit à la vue [vue-10.xml]. A Noter que le paramètre [Model model] n'a besoin d'être présent que pour l'instruction de la ligne 8. Sans elle, il est inutile ;

La vue [vue-10.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${uneAutrePersonne}">
9.           <p>
10.             <span th:text="#{personne.id}">Id :</span>
11.             <span th:text="*{id}">14</span>
12.          </p>
13.          <p>
14.             <span th:text="#{personne.nom}">Nom :</span>
```

```
15.             <span th:text="*{nom}">Bill</span>
16.          </p>
17.          <p>
18.             <span th:text="#{personne.age}">Age :</span>
19.             <span th:text="*{age}">56</span>
20.          </p>
21.       </div>
22.    </body>
23. </html>
```

Le résultat est le suivant :



et le log console le suivant :

```
Modèle={uneAutrePersonne=[id=24, nom=pauline,  age=55]}
```

5.9 [/v11] : @SessionAttributes

Nous revenons sur quelque chose que nous avons vu lors de l'étude des actions, le rôle de l'annotation [@SessionAttributes]. Nous ajoutons la nouvelle action [/v11] suivante :

```
1.     @ModelAttribute("jean")
2.     private Personne getJean(){
3.        return new Personne(33,"jean",10);
4.     }
5.
6.     @RequestMapping(value = "/v11", method = RequestMethod.GET)
7.     public String v11(Model model, HttpSession session) {
8.        System.out.println(String.format("Modèle=%s, Session[jean]=%s", model,
   session.getAttribute("jean")));
9.        return "vue-11";
10. }
```

Nous avons quelque chose d'analogue à ce qui vient d'être étudié. La différence réside en une annotation [@SessionAttributes] placée sur la classe elle-même :

```
1.  @Controller
2.  @SessionAttributes("jean")
3.  public class ViewsController {
```

- ligne 2 : on indique que la clé [jean] du modèle doit être placé dans la session ;

C'est pourquoi en ligne 7 de l'action, on a injecté la session. Ligne 8, on affiche la valeur de la session associée à la clé [jean].

La vue [vue-11.xml] est la suivante :

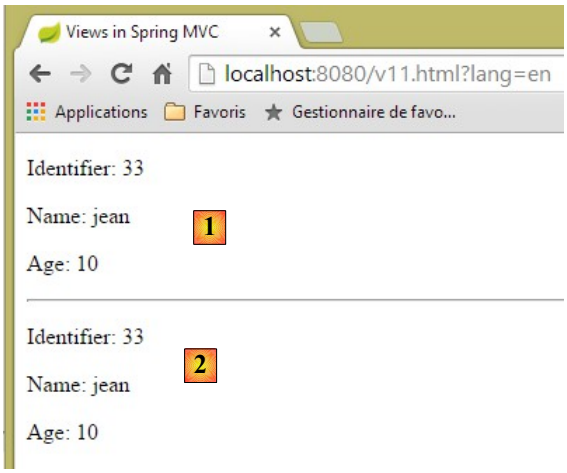
```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${jean}">
```

```
9.            <p>
10.               <span th:text="#{personne.id}">Id :</span>
11.               <span th:text="*{id}">14</span>
12.           </p>
13.           <p>
14.               <span th:text="#{personne.nom}">Nom :</span>
15.               <span th:text="*{nom}">Bill</span>
16.           </p>
17.           <p>
18.               <span th:text="#{personne.age}">Age :</span>
19.               <span th:text="*{age}">56</span>
20.           </p>
21.       </div>
22.       <hr />
23.       <div th:object="${session.jean}">
24.           <p>
25.               <span th:text="#{personne.id}">Id :</span>
26.               <span th:text="*{id}">14</span>
27.           </p>
28.           <p>
29.               <span th:text="#{personne.nom}">Nom :</span>
30.               <span th:text="*{nom}">Bill</span>
31.           </p>
32.           <p>
33.               <span th:text="#{personne.age}">Age :</span>
34.               <span th:text="*{age}">56</span>
35.           </p>
36.       </div>
37.   </body>
38. </html>
```

On affiche deux personnes :

- lignes 8-21 : la personne de clé [jean] dans le modèle ;
- lignes 23-36 : la personne de clé [jean] dans la session ;

Les résultats sont les suivants :



- en [1], la personne de clé [jean] dans le modèle ;
- en [2], la personne de clé [jean] dans la session ;

Le log console est lui le suivant :

```
Modèle={uneAutrePersonne=[id=24, nom=pauline,  age=55], jean=[id=33, nom=jean,  age=10]}, Session[jean]=null
```

Ci-dessus, on voit que la clé [jean] n'est pas dans la session que reçoit l'action. On en déduit, que la clé [jean] a été mise dans la session **après l'exécution de l'action** et avant l'affichage de la vue.

Maintenant, considérons le cas où une clé est à la fois référencée par [@ModelAttribute] et [@SessionAttributes]. Nous construisons les deux actions suivantes :

```
1.      @RequestMapping(value = "/v12a", method = RequestMethod.GET)
2.      @ResponseBody
3.      public void v12a(HttpSession session) {
4.         session.setAttribute("paul", new Personne(51, "paul", 33));
5.      }
6.
7.      // cas où la clé de [@ModelAttribute] est également une clé de [@SessionAttributes]
8.      // dans ce cas, le paramètre correspondant est initialisé avec la valeur de la session
9.      @RequestMapping(value = "/v12b", method = RequestMethod.GET)
10.     public String v12b(Model model, @ModelAttribute("paul") Personne p) {
11.        System.out.println(String.format("Modèle=%s", model));
12.        return "vue-12";
13. }
```

L'action [/v12a] ne sert qu'à mettre dans la session l'élément ['paul',**new** Personne(51, "paul", 33)]. Elle ne fait rien d'autre. Le fait qu'elle soit taguée par [@ResponseBody] indique que c'est elle qui génère la réponse au client. Comme son type est [void], aucune réponse n'est générée.

L'action [/v12b] admet comme paramètre [@ModelAttribute("paul") Personne p]. Si on ne fait rien d'autre, un objet [Personne] est instancié puis initialisé avec les paramètres de la requête et cet objet n'a rien à voir avec l'objet de clé [paul] mis dans la session par l'action [/v12a]. Nous allons ajouter la clé [paul] aux attributs de session de la classe :

```
1.  @Controller
2.  @SessionAttributes({ "jean", "paul" })
3.  public class ViewsController {
```

- ligne 2, il y a désormais deux attributs de session ;

Revenons aux paramètres de l'action [/v12b] :

```
public String v12b(Model model, @ModelAttribute("paul") Personne p) {
```

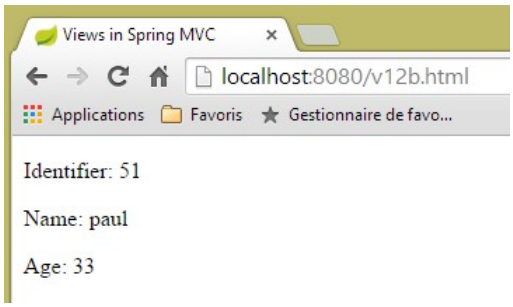
Maintenant, l'objet [Personne p] ne va pas être instancié mais va référencer l'objet de clé [paul] dans la session. Ensuite la procédure reste la même. L'objet de clé [paul] va notamment se retrouver dans le modèle de la vue qui sera affichée. C'est ce qu'on veut voir ligne 11 de l'action [/v12b].

La vue [vue-12.xml] sera la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <div th:object="${paul}">
9.           <p>
10.             <span th:text="#{personne.id}">Id :</span>
11.             <span th:text="*{id}">14</span>
12.          </p>
13.          <p>
14.             <span th:text="#{personne.nom}">Nom :</span>
15.             <span th:text="*{nom}">Bill</span>
16.          </p>
17.          <p>
18.             <span th:text="#{personne.age}">Age :</span>
19.             <span th:text="*{age}">56</span>
20.          </p>
21.       </div>
22.    </body>
23. </html>
```

- ligne 8 : on référence la clé [paul] du modèle de la vue ;

Cela donne le résultat suivant (après avoir exécuté l'action [/v12a] qui met la clé [paul] dans la session) :



Le log console est le suivant :

```
Modèle={jean=[id=33, nom=jean,  age=10], uneAutrePersonne=[id=24, nom=pauline,  age=55], paul=[id=51, nom=paul,  age=33],
org.springframework.validation.BindingResult.paul=org.springframework.validation.BeanPropertyBindingResult: 0 errors}
```

La clé [paul] a bien été mise dans le modèle avec pour valeur, la valeur associée à la clé [paul] dans la session.

5.10 [/v13] : générer un formulaire de saisie

Nous abordons maintenant la saisie des formulaires et leur validation. Nous construisons un premier formulaire avec l'action [/v13] suivante :

```
1.    // génère un formulaire pour saisir une personne
2.    @RequestMapping(value = "/v13", method = RequestMethod.GET)
3.    public String v13() {
4.      return "vue-13";
5.  }
```

qui se contente d'afficher la vue [vue-13.xml] suivante :

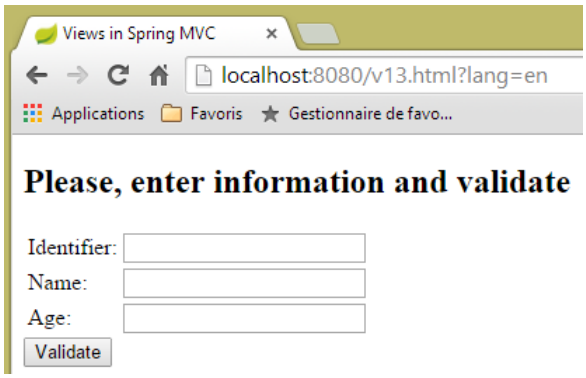
```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <form action="/someURL" th:action="@{/v14.html}" method="post">
9.           <h2 th:text="#{personne.formulaire.titre}">Entrez les informations suivantes</h2>
10.          <div th:object="${personne}">
11.             <table>
12.                <thead></thead>
13.                <tbody>
14.                   <tr>
15.                      <td th:text="#{personne.id}">Id :</td>
16.                      <td>
17.                         <input type="text" name="id" value="11" th:value="''" />
18.                      </td>
19.                   </tr>
20.                   <tr>
21.                      <td th:text="#{personne.nom}">Nom :</td>
22.                      <td>
23.                         <input type="text" name="nom" value="Tintin" th:value="''" />
24.                      </td>
25.                   </tr>
26.                   <tr>
27.                      <td th:text="#{personne.age}">Age :</td>
28.                      <td>
29.                         <input type="text" name="age" value="17" th:value="''" />
30.                      </td>
31.                   </tr>
32.                </tbody>
33.             </table>
34.          </div>
35.          <input type="submit" value="Valider" th:value="#{personne.formulaire.valider}" />
```

```
36.         </form>
37.     </body>
38. </html>
```

Si nous mettons cette vue dans le dossier [static] sous le nom [vue-13.html] et que nous demandons l'URL [http://localhost:8080/vue-13.html], nous obtenons la page suivante :

- ligne 8 du formulaire, on trouve la balise <form> avec l'attribut [th:action]. Cet attribut va être évalué par Thymeleaf et sa valeur remplacer le valeur actuelle de l'attribut [action] qui n'est donc là que pour décorer. Ici la valeur de l'attribut [th:action] sera [/v14.html] ;
- lignes 17, 23 et 29, la valeur de l'attribut [th:value] va remplacer celle de l'attribut [value]. Ici cette valeur sera la chaîne vide ;

Lorsqu'on demande l'URL [/v13.html], on obtient le résultat suivant :



Regardons le code source généré par Thymeleaf :

```
1.  <!DOCTYPE html>
2.
3.  <html>
4.      <head>
5.          <title>Views in Spring MVC</title>
6.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.      </head>
8.      <body>
9.          <form action="/v14.html" method="post">
10.             <h2>Please, enter information and validate</h2>
11.             <div>
12.                 <table>
13.                     <thead></thead>
14.                     <tbody>
15.                         <tr>
16.                             <td>Identifier:</td>
```

```
17.                         <td>
18.                             <input type="text" name="id" value="" />
19.                         </td>
20.                     </tr>
21.                     <tr>
22.                         <td>Name:</td>
23.                         <td>
24.                             <input type="text" name="nom" value="" />
25.                         </td>
26.                     </tr>
27.                     <tr>
28.                         <td>Age:</td>
29.                         <td>
30.                             <input type="text" name="age" value="" />
31.                         </td>
32.                     </tr>
33.                 </tbody>
34.             </table>
35.         </div>
36.         <input type="submit" value="Validate" />
37.     </form>
38. </body>
39. </html>
```

Lignes 9, 18, 24 et 30, on voit l'évaluation des attributs [th:action] et [th:value] faite par Thymeleaf.

5.11 [/v14] : gérer les valeurs postées par un formulaire

L'action [/v14] est l'action qui reçoit les valeurs postées. C'est la suivante :

```
1.    // traite les valeurs du formulaire
2.    @RequestMapping(value = "/v14", method = RequestMethod.POST)
3.    public String v14(Personne p) {
4.      return "vue-14";
5.  }
```

- ligne 3 : les valeurs postées sont encapsulées dans un objet [Personne p]. On sait que cet objet fait automatiquement partie du modèle M de la vue V qui sera affichée par l'action, associé à la clé [personne] ;
- ligne 4, la vue affichée est la vue [vue-14.xml] ;

La vue [vue-14.xml] est la suivante :

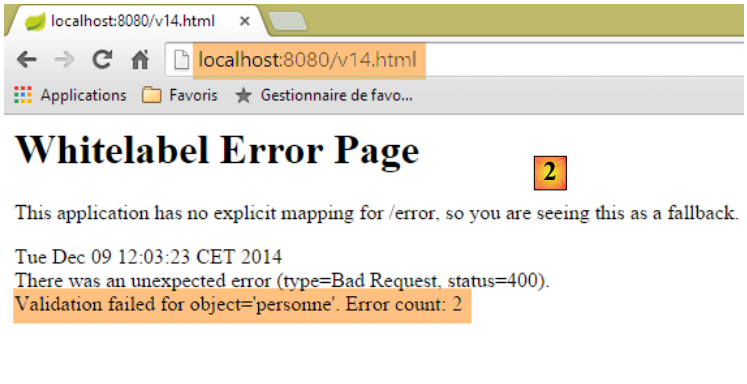
```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title th:text="#{title}">Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.      </head>
7.      <body>
8.      <h2 th:text="#{personne.formulaire.saisies}">Voici vos saisies</h2>
9.          <div th:object="${personne}">
10.             <p>
11.                 <span th:text="#{personne.id}">Id :</span>
12.                 <span th:text="*{id}">14</span>
13.             </p>
14.             <p>
15.                 <span th:text="#{personne.nom}">Nom :</span>
16.                 <span th:text="*{nom}">Bill</span>
17.             </p>
18.             <p>
19.                 <span th:text="#{personne.age}">Age :</span>
20.                 <span th:text="*{age}">56</span>
21.             </p>
22.         </div>
23.     </body>
24. </html>
```

- ligne 9 : on récupère dans le modèle l'objet associé à la clé [personne] ;
- lignes 12, 16 et 20 : on affiche les caractéristiques de cet objet ;

Cela donne le résultat suivant :

5.12 [/v15-/v16] : validation d'un modèle

Avec l'exemple précédent, regardons la séquence suivante :



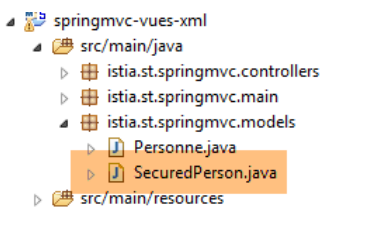
- en [1], on rentre des valeurs erronées pour les champs [id] et [age] de type [int] ;
- en [2], la réponse du serveur nous indique qu'il y a eu deux erreurs ;

Nous allons utiliser le même formulaire mais en cas d'erreurs de validation, nous allons renvoyer une page signalant ces erreurs afin que l'utilisateur puisse les corriger.

L'action [/v15] est la suivante :

```
1.      // ---------------------- affichage d'un formulaire
2.      @RequestMapping(value = "/v15", method = RequestMethod.GET)
3.      public String v15(SecuredPerson p) {
4.          return "vue-15";
5.  }
```

Elle reçoit en paramètre un type [SecuredPerson] suivant :



```
1.  package istia.st.springmvc.models;
2.
3.  import javax.validation.constraints.NotNull;
4.
5.  import org.hibernate.validator.constraints.Length;
6.  import org.hibernate.validator.constraints.Range;
7.
8.  public class SecuredPerson {
9.
10.    @Range(min = 1)
11.    private int id;
12.
13.    @Length(min = 4, max = 10)
14.    private String nom;
15.
16.    @Range(min = 8, max = 14)
17.    private int age;
18.
19.    // constructeurs
20.    public SecuredPerson() {
21.
22.    }
23.
24.    public SecuredPerson(int id, String nom, int age) {
25.       this.id=id;
26.       this.nom = nom;
27.       this.age = age;
28.    }
29.
30.    // getters et setters
31. ...
32. }
```

Les champs [id, nom, age] ont été annotés avec des contraintes de validation. La vue [vue-15.xml] affichée par l'action [/v15] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <form action="/someURL" th:action="@{/v16.html}" method="post">
9.           <h2 th:text="#{personne.formulaire.titre}">Entrez les informations suivantes</h2>
10.          <div th:object="${securedPerson}">
11.             <table>
12.                <thead></thead>
13.                <tbody>
14.                   <tr>
15.                      <td th:text="#{personne.id}">Id :</td>
16.                      <td>
17.                         <input type="text" name="id" value="11" th:value="*{id}" />
18.                      </td>
19.                      <td>
20.                         <span th:if="${#fields.hasErrors('id')}" th:errors="*{id}" style="color:
    red">Identifiant erroné</span>
21.                      </td>
22.                   </tr>
23.                   <tr>
24.                      <td th:text="#{personne.nom}">Nom :</td>
25.                      <td>
26.                         <input type="text" name="nom" value="Tintin" th:value="*{nom}" />
27.                      </td>
28.                      <td>
29.                         <span th:if="${#fields.hasErrors('nom')}" th:errors="*{nom}" style="color:
    red">Nom erroné</span>
30.                      </td>
31.                   </tr>
32.                   <tr>
33.                      <td th:text="#{personne.age}">Age :</td>
34.                      <td>
35.                         <input type="text" name="age" value="17" th:value="*{age}" />
36.                      </td>
```

```
37.                         <td>
38.                            <span th:if="${#fields.hasErrors('age')}" th:errors="*{age}" style="color:
    red">Âge erroné</span>
39.                         </td>
40.                      </tr>
41.                   </tbody>
42.                </table>
43.                <input type="submit" value="Valider" th:value="#{personne.formulaire.valider}" />
44.                <ul>
45.                   <li th:each="err : ${#fields.errors('*')}" th:text="${err}" style="color: red" />
46.                </ul>
47.             </div>
48.          </form>
49.       </body>
50. </html>
```

- lignes 10-47 : l'objet du modèle de la page attaché à la clé [securedPerson] est récupéré. A l'issue du GET, on a un objet avec sa valeur d'instanciation [id=0, nom=null, age=0] ;
- ligne 17 : la valeur du champ [securedPerson.id] ;
- ligne 20 : l'expression [*${#fields.hasErrors('id')}*] permet de savoir s'il y a eu des erreurs de validation sur le champ [securedPerson.id]. Si c'est le cas, l'attribut [th:errors="*{id}"] affiche le message d'erreur associé ;
- ce scénario se répète aux lignes 29 pour le champ [nom] et 38 pour le champ [age] ;
- ligne 45 : l'expression [*${#fields.errors('*')}*] désigne l'ensemble des erreurs sur les champs de l'objet [securedPerson]. Ainsi, c'est l'ensemble de ces erreurs qui va être affiché par les lignes 44-46 ;
- ligne 16 : on voit que les valeurs du formulaire vont être postées à l'action [/v16]. Celle-ci est la suivante :

```
1.      // -------------------- validation d'un modèle-----------------
2.      @RequestMapping(value = "/v16", method = RequestMethod.POST)
3.      public String v16(@Valid SecuredPerson p, BindingResult result) {
4.         // erreurs ?
5.         if (result.hasErrors()) {
6.            return "vue-15";
7.         } else {
8.            return "vue-16";
9.         }
10. }
```

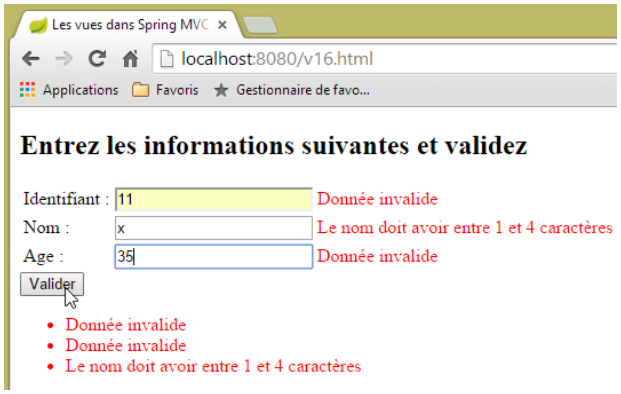
- ligne 3, l'annotation [@Valid SecuredPerson p] force la validation des valeurs postées ;
- ligne 5 : teste si le modèle de l'action est erroné ou non ;
- ligne 6 : s'il est erroné, on retourne le formulaire [vue-15.xml]. Comme celui-ci affiche les messages d'erreur, on va voir ceux-ci ;
- ligne 8 : si le modèle de l'action est validé, alors on affiche la vue [vue-16.xml] suivante :

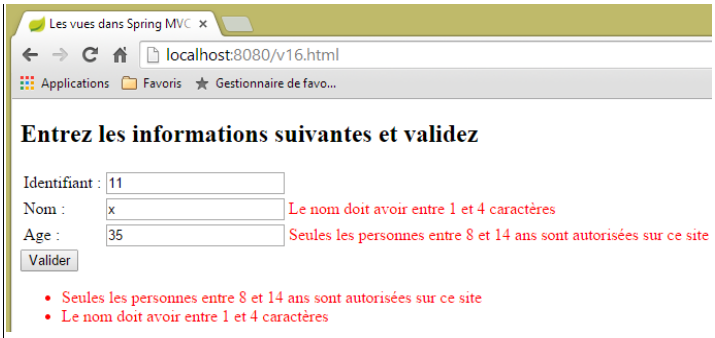
```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.     <h2 th:text="#{personne.formulaire.saisies}">Voici vos saisies</h2>
9.        <div th:object="${securedPerson}">
10.          <p>
11.             <span th:text="#{personne.id}">Id :</span>
12.             <span th:text="*{id}">14</span>
13.          </p>
14.          <p>
15.             <span th:text="#{personne.nom}">Nom :</span>
16.             <span th:text="*{nom}">Bill</span>
17.          </p>
18.          <p>
19.             <span th:text="#{personne.age}">Age :</span>
20.             <span th:text="*{age}">56</span>
21.          </p>
22.       </div>
23.    </body>
24. </html>
```

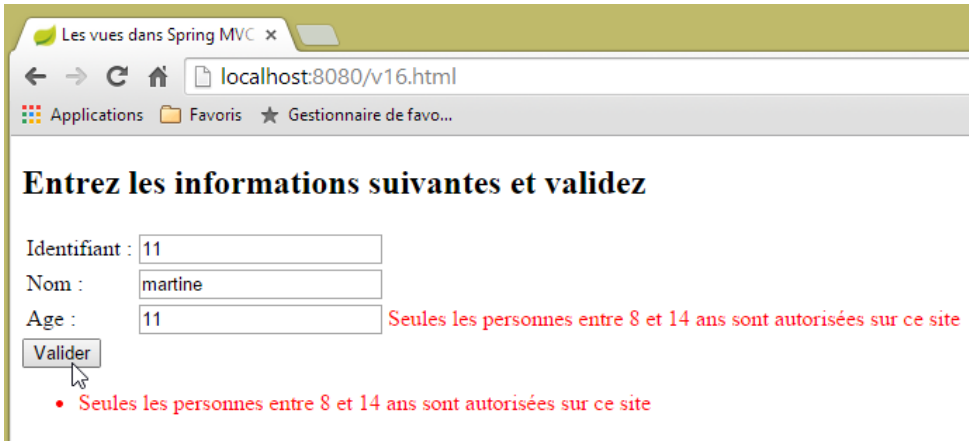
Voici des exemples d'exécution :


Les vues dans Spring MVC
localhost:8080/v15.html?lang=fr
Applications
Favoris
Gestionnaire de favo...
Entrez les informations suivantes et validez
Identifiant : x
Nom : x
Age : x
Valider


Les vues dans Spring MVC
localhost:8080/v16.html
Applications
Favoris
Gestionnaire de favo...
Entrez les informations suivantes et validez
Identifiant : 0 Donnée invalide
Nom : x Le nom doit avoir entre 1 et 4 caractères
Age : 0 Donnée invalide
Valider
Donnée invalide
Donnée invalide
Le nom doit avoir entre 1 et 4 caractères


Les vues dans Spring MVC
localhost:8080/v16.html
Applications
Favoris
Gestionnaire de favo...
Entrez les informations suivantes et validez
Identifiant : 11 Donnée invalide
Nom : x Le nom doit avoir entre 1 et 4 caractères
Age : 35 Donnée invalide
Valider
Donnée invalide
Donnée invalide
Le nom doit avoir entre 1 et 4 caractères


Les vues dans Spring MVC
localhost:8080/v16.html
Applications
Favoris
Gestionnaire de favo...
Entrez les informations suivantes et validez
Identifiant : 11
Nom : x Le nom doit avoir entre 1 et 4 caractères
Age : 35 Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
Valider
Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
Le nom doit avoir entre 1 et 4 caractères


Les vues dans Spring MVC
localhost:8080/v16.html
Applications
Favoris
Gestionnaire de favo...
Entrez les informations suivantes et validez
Identifiant : 11
Nom : martine
Age : 11 Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
Valider
Seules les personnes entre 8 et 14 ans sont autorisées sur ce site

5.13 [/v17-/v18] : contrôle des messages d'erreur

Lorsqu'on demande la première fois l'action [/v15], on obtient le résultat suivant :

On pourrait vouloir un formulaire vide plutôt que des zéros dans les champs [Identifiant, Age]. Pour obtenir cela, nous faisons évoluer le modèle de l'action de la façon suivante :

```
1.  package istia.st.springmvc.models;
2.
3.  import javax.validation.constraints.Digits;
4.
5.  import org.hibernate.validator.constraints.Length;
6.  import org.hibernate.validator.constraints.Range;
7.
8.  public class StringSecuredPerson {
9.
10.     @Range(min = 1)
11.     @Digits(fraction = 0, integer = 4)
12.     private String id;
13.
14.     @Length(min = 4, max = 10)
15.     private String nom;
16.
17.     @Range(min = 8, max = 14)
18.     @Digits(fraction = 0, integer = 2)
19.     private String age;
20.
21.     // constructeurs
22.     public StringSecuredPerson() {
23.
24.     }
25.
26.     public StringSecuredPerson(String id, String nom, String age) {
27.        this.id = id;
```

```
28.         this.nom = nom;
29.         this.age = age;
30.     }
31.
32.     // getters et setters
33. ...
34.
35. }
```

- lignes 12 et 19 : les champs [id] et [age] sont passés en type [String] ;
- ligne 11 : on indique que le champ [id] doit être un nombre d'au plus quatre chiffres, sans décimales ;
- ligne 18 : idem pour le champ [age] qui doit être un nombre entier d'au plus deux chiffres ;

L'action [/v17] devient la suivante :

```
1.      // ---------------------- affichage d'un formulaire
2.      @RequestMapping(value = "/v17", method = RequestMethod.GET)
3.      public String v17(StringSecuredPerson p) {
4.         return "vue-17";
5.  }
```

La vue [vue-17.xml] affichée par l'action [/v17] est la suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{title}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <form action="/someURL" th:action="@{/v18.html}" method="post">
9.           <h2 th:text="#{personne.formulaire.titre}">Entrez les informations suivantes</h2>
10.          <div th:object="${stringSecuredPerson}">
11.             <table>
12.                <thead></thead>
13.                <tbody>
14.                   <tr>
15.                      <td th:text="#{personne.id}">Id :</td>
16.                      <td>
17.                         <input type="text" name="id" value="11" th:value="*{id}" />
18.                      </td>
19.                      <td>
20.                         <span th:each="err,status : ${#fields.errors('id')}" th:if="${status.index}==0"
    th:text="${err}" style="color: red">
21.                            Identifiant erroné
22.                         </span>
23.                      </td>
24.                   </tr>
25.                   <tr>
26.                      <td th:text="#{personne.nom}">Nom :</td>
27.                      <td>
28.                         <input type="text" name="nom" value="Tintin" th:value="*{nom}" />
29.                      </td>
30.                      <td>
31.                         <span th:if="${#fields.hasErrors('nom')}" th:errors="*{nom}" style="color:
    red">Nom erroné</span>
32.                      </td>
33.                   </tr>
34.                   <tr>
35.                      <td th:text="#{personne.age}">Age :</td>
36.                      <td>
37.                         <input type="text" name="age" value="17" th:value="*{age}" />
38.                      </td>
39.                      <td>
40.                         <span th:if="${#fields.hasErrors('age')}" th:errors="*{age}" style="color:
    red">Âge erroné</span>
41.                      </td>
42.                   </tr>
43.                </tbody>
44.             </table>
45.             <input type="submit" value="Valider" th:value="#{personne.formulaire.valider}" />
46.             <ul>
47.                <li th:each="err : ${#fields.errors('*')}" th:text="${err}" style="color: red" />
```

```
48.             </ul>
49.         </div>
50.       </form>
51.    </body>
52. </html>
```

Les changements ont lieu aux lignes suivantes :

- ligne 10 : on travaille désormais avec l'objet du modèle de clé [*stringSecuredPerson*] ;
- ligne 20 : on parcourt la liste des erreurs du champ [id]. Dans la syntaxe [th:each=*"err,status : ${#fields.errors('id')}"*], c'est la variable [err] qui parcourt la liste. La variable [status] donne des informations sur chaque itération. C'est un objet [index, count, size, current] où :
  - index : est le n° de l'élément courant,
  - current : la valeur de cet élément courant,
  - count, size : la taille de la liste parcourue ;
- ligne 20 : on n'affiche que le 1er élément de la liste [th:if=*"${status.index}==0"*] ;

L'action [/v18] qui traite le POST de l'action [/v17] est la suivante :

```
1.     // -------------------- validation d'un modèle-----------------
2.     @RequestMapping(value = "/v18", method = RequestMethod.POST)
3.     public String v18(@Valid StringSecuredPerson p, BindingResult result) {
4.        // erreurs ?
5.        if (result.hasErrors()) {
6.           return "vue-17";
7.        } else {
8.           return "vue-18";
9.        }
10. }
```

Les fichiers de messages évoluent de la façon suivante :

[messages_fr.properties]

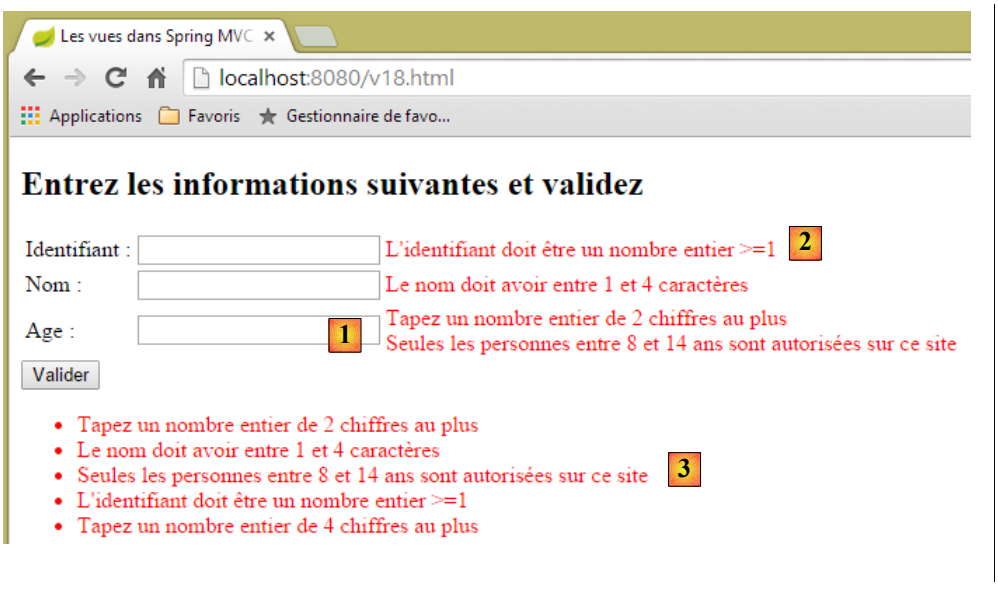
```
1.  title=Les vues dans Spring MVC
2.  personne.nom=Nom :
3.  personne.age=Age :
4.  personne.id=Identifiant :
5.  personne.mineure=Vous êtes mineur
6.  personne.majeure=Vous êtes majeur
7.  liste.personnes=Liste de personnes
8.  personne.formulaire.titre=Entrez les informations suivantes et validez
9.  personne.formulaire.valider=Valider
10. personne.formulaire.saisies=Voici vos saisies
11. notNull=La donnée est obligatoire
12. Range.securedPerson.id=L''identifiant doit être un nombre entier >=1
13. Range.securedPerson.age=Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
14. Length.securedPerson.nom=Le nom doit avoir entre 1 et 4 caractères
15. typeMismatch=Donnée invalide
16. Range.stringSecuredPerson.id=L''identifiant doit être un nombre entier >=1
17. Range.stringSecuredPerson.age=Seules les personnes entre 8 et 14 ans sont autorisées sur ce
    site
18. Length.stringSecuredPerson.nom=Le nom doit avoir entre 1 et 4 caractères
19. Digits.stringSecuredPerson.id=Tapez un nombre entier de 4 chiffres au plus
20. Digits.stringSecuredPerson.age=Tapez un nombre entier de 2 chiffres au plus
```

[messages_en.properties]

```
1.  title=Views in Spring MVC
2.  personne.nom=Name:
3.  personne.age=Age:
4.  personne.id=Identifier:
5.  personne.mineure=You are under 18
6.  personne.majeure=You are over 18
7.  liste.personnes=Persons' list
```

```
personne.formulaire.titre=Please, enter information and validate
personne.formulaire.valider=Validate
personne.formulaire.saisies=Here are your inputs
NotNull=Data is required
Range.securedPerson.id=Identifier must be an integer >=1
Range.securedPerson.age=Only kids who are 8 to 14 years old are allowed on this site
Length.securedPerson.nom=Name must be 4 to 10 characters long
typeMismatch=Invalid format
Range.stringSecuredPerson.id=Identifier must be an integer >=1
Range.stringSecuredPerson.age=Only kids who are 8 to 14 years old are allowed on this site
Length.stringSecuredPerson.nom=Name must be 4 to 10 characters long
Digits.stringSecuredPerson.id=Should be an integer with at most four digits
Digits.stringSecuredPerson.age=Should be an integer with at most two digits
```

Voyons quelques exemples :



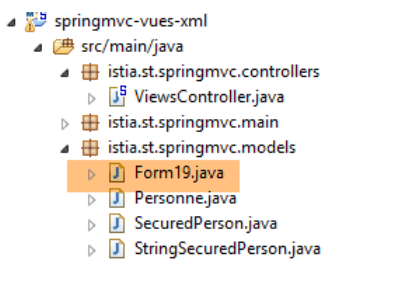
On voit en [1], que les deux validateurs du champ [age] ont été exécutés :

```
    @Range(min = 8, max = 14)
    @Digits(fraction = 0, integer = 2)
    private String age;
```

Y-a-t-il un ordre des messages d'erreur ? Pour le champ [age], il semble que les validateurs se soient exécutés dans l'ordre [Digits, Range]. Mais si on fait plusieurs requêtes, on peut constater que cet ordre peut changer. Donc, on ne peut se fier à l'ordre des validateurs. En [2], on n'affiche qu'un message sur les deux du champ [id]. En [3], on voit l'ensemble des messages d'erreur.

5.14 [/v19-/v20] : usage de différents validateurs

Considérons le nouveau modèle d'action suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import java.util.Date;
4.
5.  import javax.validation.constraints.AssertFalse;
6.  import javax.validation.constraints.AssertTrue;
7.  import javax.validation.constraints.Future;
8.  import javax.validation.constraints.Max;
9.  import javax.validation.constraints.Min;
10. import javax.validation.constraints.NotNull;
11. import javax.validation.constraints.Past;
12. import javax.validation.constraints.Pattern;
13. import javax.validation.constraints.Size;
14.
15. import org.hibernate.validator.constraints.Email;
16. import org.hibernate.validator.constraints.Length;
17. import org.hibernate.validator.constraints.NotBlank;
18. import org.hibernate.validator.constraints.NotEmpty;
19. import org.hibernate.validator.constraints.Range;
20. import org.hibernate.validator.constraints.URL;
21. import org.springframework.format.annotation.DateTimeFormat;
22.
23. public class Form19 {
24.
25.     @NotNull
26.     @AssertFalse
27.     private Boolean assertFalse;
28.
29.     @NotNull
30.     @AssertTrue
31.     private Boolean assertTrue;
32.
33.     @NotNull
34.     @Future
35.     @DateTimeFormat(pattern = "yyyy-MM-dd")
36.     private Date dateInFuture;
37.
38.     @NotNull
39.     @Past
40.     @DateTimeFormat(pattern = "yyyy-MM-dd")
41.     private Date dateInPast;
42.
43.     @NotNull
44.     @Max(value = 100)
45.     private Integer intMax100;
46.
47.     @NotNull
48.     @Min(value = 10)
49.     private Integer intMin10;
50.
```

```
51.     @NotNull
52.     @NotEmpty
53.     private String strNotEmpty;
54.
55.     @NotNull
56.     @NotBlank
57.     private String strNotBlank;
58.
59.     @NotNull
60.     @Size(min = 4, max = 6)
61.     private String strBetween4and6;
62.
63.     @NotNull
64.     @Pattern(regexp = "^\\d{2}:\\d{2}:\\d{2}$")
65.     private String hhmmss;
66.
67.     @NotNull
68.     @Email
69.     @NotBlank
70.     private String email;
71.
72.     @NotNull
73.     @Length(max = 4, min = 4)
74.     private String str4;
75.
76.     @Range(min = 10, max = 14)
77.     @NotNull
78.     private Integer int1014;
79.
80.     @URL
81.     @NotBlank
82.     private String url;
83.
84.     // getters et setters
85. ...
86. }
```

Il sera affiché par l'action [/v19] suivante :

```
1.     // ------------------ affichage d'un formulaire
2.     @RequestMapping(value = "/v19", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
3.     public String v19(Form19 formulaire) {
4.         return "vue-19";
5.  }
```

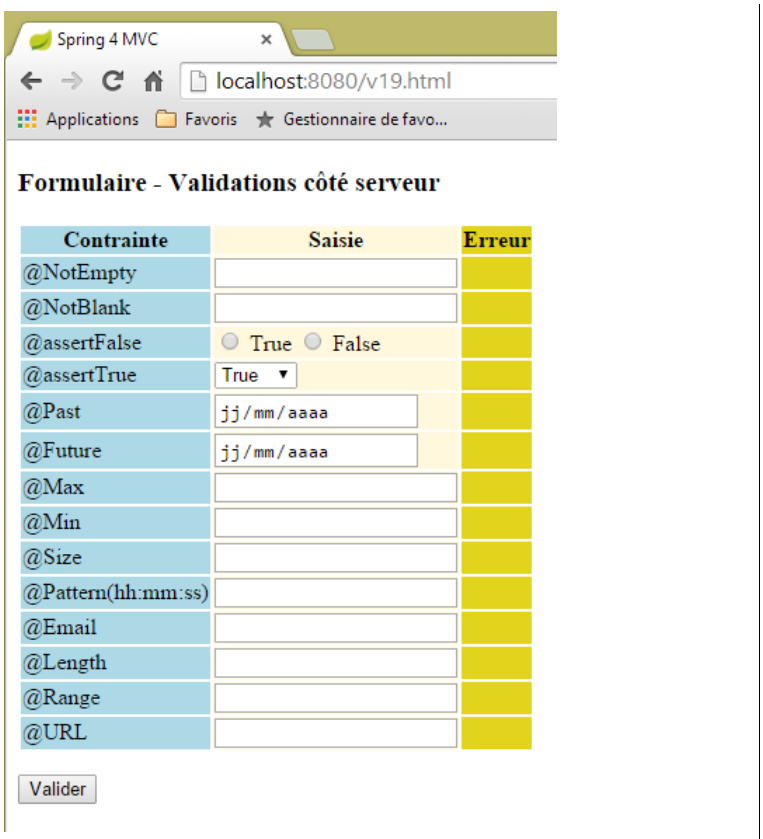
- ligne 3 : l'action reçoit comme paramètre un objet [Form19 formulaire]. Si le GET ne reçoit pas de paramètres, cet objet sera initialisé avec les valeurs par défaut du Java ;
- ligne 4 : la vue [vue-19.xml] est affichée. Celle-ci est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title>Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.        <link rel="stylesheet" href="/css/form19.css" />
7.     </head>
8.     <body>
9.        <h3>Formulaire - Validations côté serveur</h3>
10.       <form action="/someURL" th:action="@{/v20.html}" method="post" th:object="${form19}">
11.          <table>
12.             <thead>
13.                <tr>
14.                   <th class="col1">Contrainte</th>
15.                   <th class="col2">Saisie</th>
16.                   <th class="col3">Erreur</th>
17.                </tr>
18.             </thead>
19.             <tbody>
20.                <tr>
21.                   <td class="col1">@NotEmpty</td>
22.                   <td class="col2">
23.                      <input type="text" th:field="*{strNotEmpty}" />
24.                   </td>
```

```
25.                     <td class="col3">
26.                         <span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}"
    class="error">Donnée erronée</span>
27.                     </td>
28.                 </tr>
29.                 <tr>
30.                     <td class="col1">@NotBlank</td>
31.                     <td class="col2">
32.                         <input type="text" th:field="*{strNotBlank}" />
33.                     </td>
34.                     <td class="col3">
35.                         <span th:if="${#fields.hasErrors('strNotBlank')}" th:errors="*{strNotBlank}"
    class="error">Donnée erronée</span>
36.                     </td>
37.                 </tr>
38.                 <tr>
39.                     <td class="col1">@assertFalse</td>
40.                     <td class="col2">
41.                         <input type="radio" th:field="*{assertFalse}" value="true" />
42.                         <label th:for="${#ids.prev('assertFalse')}">True</label>
43.                         <input type="radio" th:field="*{assertFalse}" value="false" />
44.                         <label th:for="${#ids.prev('assertFalse')}">False</label>
45.                     </td>
46.                     <td class="col3">
47.                         <span th:if="${#fields.hasErrors('assertFalse')}" th:errors="*{assertFalse}"
    class="error">Donnée erronée</span>
48.                     </td>
49.                 </tr>
50.                 <tr>
51.                     <td class="col1">@assertTrue</td>
52.                     <td class="col2">
53.                         <select th:field="*{assertTrue}">
54.                             <option value="true">True</option>
55.                             <option value="false">False</option>
56.                         </select>
57.                     </td>
58.                     <td class="col3">
59.                         <span th:if="${#fields.hasErrors('assertTrue')}" th:errors="*{assertTrue}"
    class="error">Donnée erronée</span>
60.                     </td>
61.                 </tr>
62.                 <tr>
63.                     <td class="col1">@Past</td>
64.                     <td class="col2">
65.                         <input type="date" th:field="*{dateInPast}" th:value="*{dateInPast}" />
66.                     </td>
67.                     <td class="col3">
68.                         <span th:if="${#fields.hasErrors('dateInPast')}" th:errors="*{dateInPast}"
    class="error">Donnée erronée</span>
69.                     </td>
70.                 </tr>
71.                 <tr>
72.                     <td class="col1">@Future</td>
73.                     <td class="col2">
74.                         <input type="date" th:field="*{dateInFuture}" th:value="*{dateInFuture}" />
75.                     </td>
76.                     <td class="col3">
77.                         <span th:if="${#fields.hasErrors('dateInFuture')}" th:errors="*{dateInFuture}"
    class="error">Donnée erronée</span>
78.                     </td>
79.                 </tr>
80.                 <tr>
81.                     <td class="col1">@Max</td>
82.                     <td class="col2">
83.                         <input type="text" th:field="*{intMax100}" th:value="*{intMax100}" />
84.                     </td>
85.                     <td class="col3">
86.                         <span th:if="${#fields.hasErrors('intMax100')}" th:errors="*{intMax100}"
    class="error">Donnée erronée</span>
87.                     </td>
88.                 </tr>
89.                 <tr>
90.                     <td class="col1">@Min</td>
91.                     <td class="col2">
92.                         <input type="text" th:field="*{intMin10}" th:value="*{intMin10}" />
```

```
93.                     </td>
94.                     <td class="col3">
95.                         <span th:if="${#fields.hasErrors('intMin10')}" th:errors="*{intMin10}"
    class="error">Donnée erronée</span>
96.                     </td>
97.                 </tr>
98.                 <tr>
99.                     <td class="col1">@Size</td>
100.                    <td class="col2">
101.                        <input type="text" th:field="*{strBetween4and6}" th:value="*{strBetween4and6}" />
102.                    </td>
103.                    <td class="col3">
104.                        <span th:if="${#fields.hasErrors('strBetween4and6')}" th:errors="*{strBetween4and6}"
    class="error">Donnée erronée</span>
105.                    </td>
106.                </tr>
107.                <tr>
108.                    <td class="col1">@Pattern(hh:mm:ss)</td>
109.                    <td class="col2">
110.                        <input type="text" th:field="*{hhmmss}" th:value="*{hhmmss}" />
111.                    </td>
112.                    <td class="col3">
113.                        <span th:if="${#fields.hasErrors('hhmmss')}" th:errors="*{hhmmss}"
    class="error">Donnée erronée</span>
114.                    </td>
115.                </tr>
116.                <tr>
117.                    <td class="col1">@Email</td>
118.                    <td class="col2">
119.                        <input type="text" th:field="*{email}" th:value="*{email}" />
120.                    </td>
121.                    <td class="col3">
122.                        <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}"
    class="error">Donnée erronée</span>
123.                    </td>
124.                </tr>
125.                <tr>
126.                    <td class="col1">@Length</td>
127.                    <td class="col2">
128.                        <input type="text" th:field="*{str4}" th:value="*{str4}" />
129.                    </td>
130.                    <td class="col3">
131.                        <span th:if="${#fields.hasErrors('str4')}" th:errors="*{str4}" class="error">Donnée
    erronée</span>
132.                    </td>
133.                </tr>
134.                <tr>
135.                    <td class="col1">@Range</td>
136.                    <td class="col2">
137.                        <input type="text" th:field="*{int1014}" th:value="*{int1014}" />
138.                    </td>
139.                    <td class="col3">
140.                        <span th:if="${#fields.hasErrors('int1014')}" th:errors="*{int1014}"
    class="error">Donnée erronée</span>
141.                    </td>
142.                </tr>
143.                <tr>
144.                    <td class="col1">@URL</td>
145.                    <td class="col2">
146.                        <input type="text" th:field="*{url}" th:value="*{url}" />
147.                    </td>
148.                    <td class="col3">
149.                        <span th:if="${#fields.hasErrors('url')}" th:errors="*{url}" class="error">Donnée
    erronée</span>
150.                    </td>
151.                </tr>
152.            </tbody>
153.        </table>
154.        <p>
155.            <input type="submit" value="Valider" />
156.        </p>
157.    </form>
158.  </body>
159.</html>
```

Ce code affiche la vue suivante :



La page présente un tableau à trois colonnes :

- colonne 1 : le validateur du champ de saisie ;
- colonne 2 : le champ de saisie ;
- colonne 3 : les messages d'erreur sur le champ de saisie ;

Examinons par exemple le code de la vue [/v19.html] pour le validateur [@Pattern] :

```
1.                  <tr>
2.                     <td class="col1">@Pattern(hh:mm:ss)</td>
3.                     <td class="col2">
4.                        <input type="text" th:field="*{hhmmss}" th:value="*{hhmmss}" />
5.                     </td>
6.                     <td class="col3">
7.                        <span th:if="${#fields.hasErrors('hhmmss')}" th:errors="*{hhmmss}"
    class="error">Donnée erronée</span>
8.                     </td>
9.  </tr>
```

On retrouve du code que nous venons d'étudier avec les formulaires de type [Personne] :

- ligne 2 : la 1ère colonne : le nom du validateur testé ;
- ligne 4 : l'attribut Thymeleaf [th:field="*{hhmmss}] va générer les attributs HTML [id="hhmmss"] et [name="hhmmss"]. L'attribut Thymeleaf [th:value="*{hhmmss}"] va générer l'attribut HTML [value="valeur de [form19.hhmmss]]" ;
- ligne 7 : si la valeur saisie pour le champ [form19.hhmmss] est erroné, alors la ligne 7 affiche les messages d'erreur associés à ce champ ;

Les valeurs postées sont traitées par l'action [/v20] suivante :

```
1.      // ---------------- validation du modèle du formulaire
2.      @RequestMapping(value = "/v20", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
3.      public String v20(@Valid Form19 formulaire, BindingResult result, RedirectAttributes
    redirectAttributes) {
```

```
4.         if (result.hasErrors()) {
5.            return "vue-19";
6.         } else {
7.            // redirection vers [vue-19]
8.            redirectAttributes.addFlashAttribute("form19", formulaire);
9.            return "redirect:/v19.html";
10.        }
11. }
```

- ligne 3 : les valeurs postées vont remplir les champs de l'objet [Form19 formulaire] si elles sont valides ;
- ligne 4-6 : si les valeurs postées ne sont pas valides, alors on réaffiche le formulaire [vue-19] avec les messages d'erreur ;
- lignes 6-10 : si les valeurs postées sont valides, alors l'objet [Form19 formulaire] construit avec ces valeurs est mis à la disposition de la requête suivante, ici celle de la redirection. Il est détruit ensuite ;
- ligne 9 : on redirige le client vers l'action [/v19.html]. Celle-ci va réafficher le formulaire [vue-19] qui contient du code tel que :

```
<form action="/someURL" th:action="@{/v20.html}" method="post" th:object="${form19}">
```

L'attribut [th:object="${form19}"] va alors récupérer l'objet associé à l'attribut Flash [form19] et ainsi réafficher le formulaire **tel qu'il a été saisi**.

Le code du formulaire mérite encore quelques explications. Considérons le code suivant :

```
1.                  <tr>
2.                     <td class="col1">@assertFalse</td>
3.                     <td class="col2">
4.                        <input type="radio" th:field="*{assertFalse}" value="true" />
5.                        <label th:for="${#ids.prev('assertFalse')}">True</label>
6.                        <input type="radio" th:field="*{assertFalse}" value="false" />
7.                        <label th:for="${#ids.prev('assertFalse')}">False</label>
8.                     </td>
9.                     <td class="col3">
10.                       <span th:if="${#fields.hasErrors('assertFalse')}" th:errors="*{assertFalse}"
    class="error">Donnée erronée</span>
11.                    </td>
12. </tr>
```

Cela génère le code HTML suivant :

```
1.  <tr>
2.    <td class="col1">@assertFalse</td>
3.    <td class="col2">
4.      <input type="radio" value="true" id="assertFalse1" name="assertFalse" />
5.      <label for="assertFalse1">True</label>
6.      <input type="radio" value="false" id="assertFalse2" name="assertFalse" />
7.      <label for="assertFalse2">False</label>
8.    </td>
9.    <td class="col3">
10.   </td>
11. </tr>
```

Dans le code

```
1.  <input type="radio" th:field="*{assertFalse}" value="true" />
2.  <label th:for="${#ids.prev('assertFalse')}">True</label>
3.  <input type="radio" th:field="*{assertFalse}" value="false" />
4.  <label th:for="${#ids.prev('assertFalse')}">False</label>
```

les attributs Thymeleaf des lignes 1 et 3 [th:field="*{assertFalse}"] posent un problème. On a dit que cet attribut générait les attributs HTML [id=assertFalse] et [name=assertFalse]. La difficulté vient du fait que cela étant généré aux lignes 1 et 3 on a deux attributs [name] identiques et deux attributs [id] identiques. Si c'est possible avec l'attribut [name], cela ne l'est pas avec l'attribut [id]. Comme on le voit dans le code HTML généré, Thymeleaf a généré deux attributs [id] différents [id=asserFalse1] et [id=assertFalse2]. Ce qui est une bonne chose. Le problème est qu'on ne connaît pas ces identifiants et qu'on peut en avoir besoin. C'est le cas pour la balise [label] de la ligne 2. L'attribut [for] d'une balise HTML [label] doit référencer un attribut [id], en l'occurrence celui généré pour la balise [input] de la ligne 1. La documentation Thymeleaf indique que l'expression [*${#ids.prev('assertFalse')}*] permet d'obtenir le dernier attribut [id] généré pour le champ [assertFalse].

Maintenant considérons le code de la liste déroulante du formulaire :

```
1.  <select th:field="*{assertTrue}">
2.     <option value="true">True</option>
3.     <option value="false">False</option>
4.  </select>
```

Ce code génère le code HTML d'une liste déroulante :

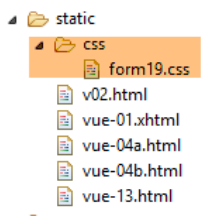
```
1.  <select id="assertTrue" name="assertTrue">
2.    <option value="true">True</option>
3.    <option value="false">False</option>
4.  </select>
```

La valeur postée le sera avec le nom [name="assertTrue"].

La vue [vue-19.xml] utilise une feuille de style :

```
1.     <head>
2.        <title>Spring 4 MVC</title>
3.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
4.        <link rel="stylesheet" href="/css/form19.css" />
5.  </head>
```

Ligne 4, la feuille de style utilisée doit être placée dans le dossier [static] du projet :



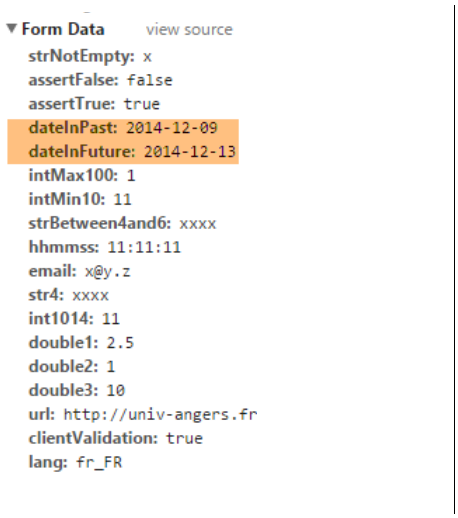
Son contenu est le suivant :

```
1.  @CHARSET "UTF-8";
2.
3.  .col1 {
4.     background: lightblue;
5.  }
6.
7.  .col2 {
8.     background: Cornsilk;
9.  }
10.
11. .col3 {
12.    background: #e2d31d;
13. }
14.
15. .error {
16.    color: red;
17.    }
```

Maintenant, examinons les dates :

```
1.     @NotNull
2.     @Future
3.     @DateTimeFormat(pattern = "yyyy-MM-dd")
4.     private Date dateInFuture;
5.
6.     @NotNull
7.     @Past
8.     @DateTimeFormat(pattern = "yyyy-MM-dd")
9.     private Date dateInPast;
```

L'examen des échanges réseau dans l'outil de développement de Chrome (Ctrl-Maj-I) montrent que les dates sont postées au format (aaaa-mm-dd) :



C'est la raison pour laquelle les dates ont été annotées avec le validateur :

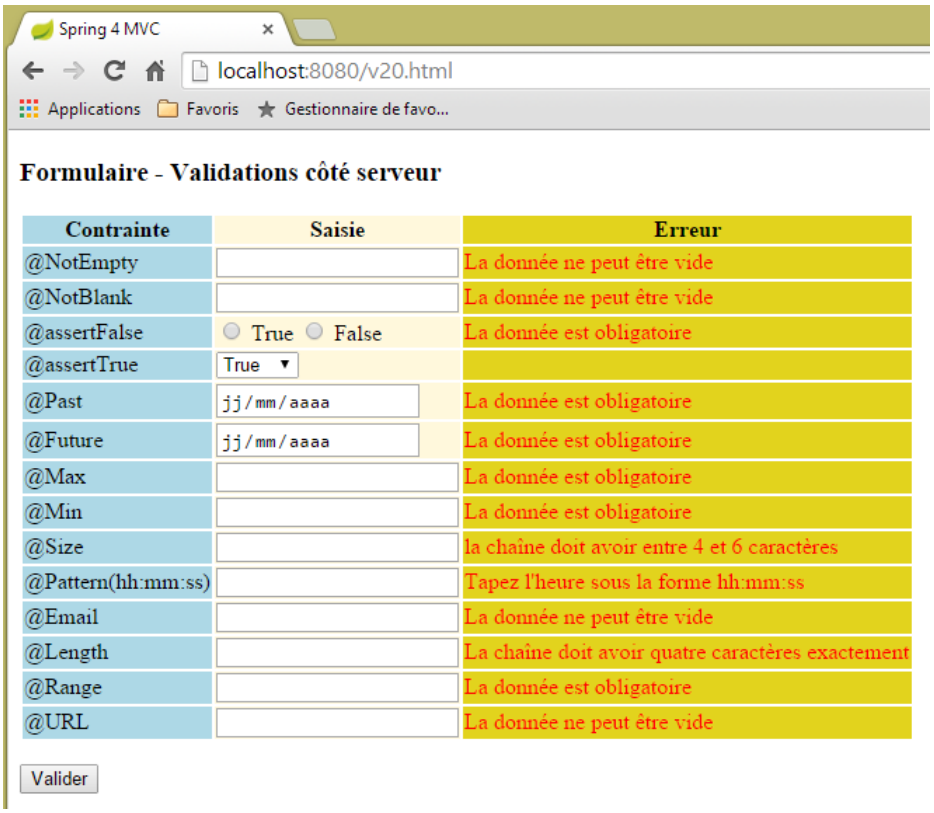
```
@DateTimeFormat(pattern = "yyyy-MM-dd")
```

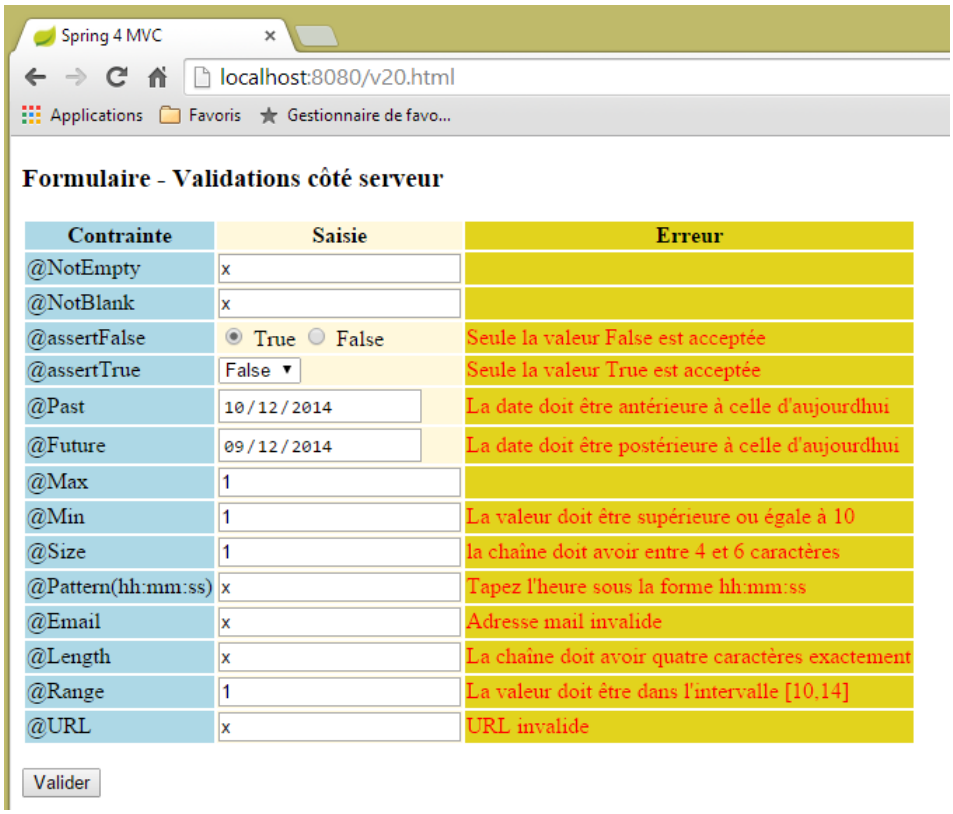
qui fixe le format attendu pour la valeur postée des dates.

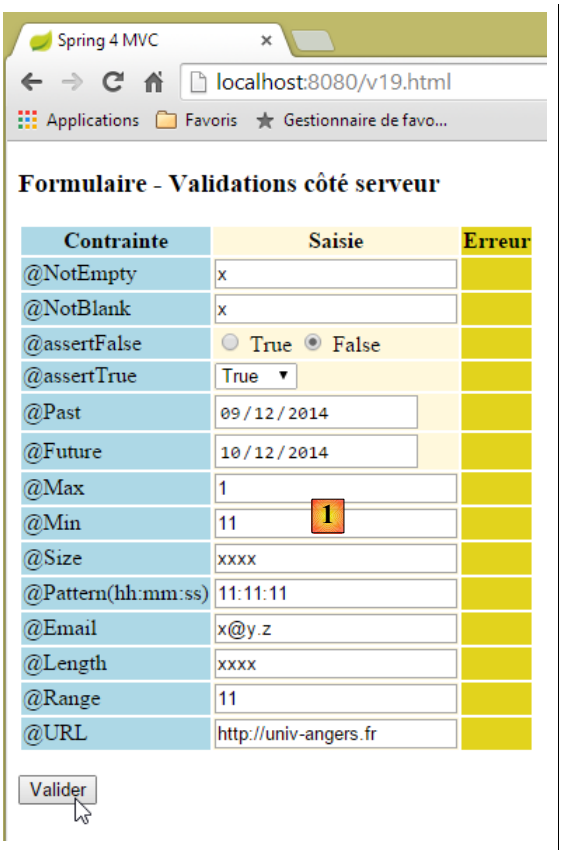
Pour terminer, le fichier des messages français [messages_fr.properties] :

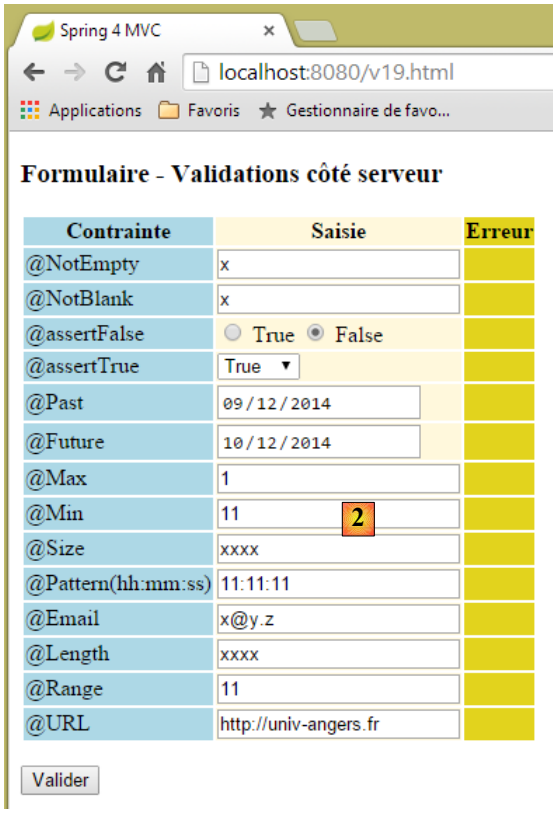
```
1.  title=Les vues dans Spring MVC
2.  personne.nom=Nom :
3.  personne.age=Age :
4.  personne.id=Identifiant :
5.  personne.mineure=Vous êtes mineur
6.  personne.majeure=Vous êtes majeur
7.  liste.personnes=Liste de personnes
8.  personne.formulaire.titre=Entrez les informations suivantes et validez
9.  personne.formulaire.valider=Valider
10. personne.formulaire.saisies=Voici vos saisies
11. NotNull=La donnée est obligatoire
12. Range.securedPerson.id=L''identifiant doit être un nombre entier >=1
13. Range.securedPerson.age=Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
14. Length.securedPerson.nom=Le nom doit avoir entre 1 et 4 caractères
15. typeMismatch=Donnée invalide
16. Range.stringSecuredPerson.id=L''identifiant doit être un nombre entier >=1
17. Range.stringSecuredPerson.age=Seules les personnes entre 8 et 14 ans sont autorisées sur ce site
18. Length.stringSecuredPerson.nom=Le nom doit avoir entre 1 et 4 caractères
19. Digits.stringSecuredPerson.id=Tapez un nombre entier de 4 chiffres au plus
20. Digits.stringSecuredPerson.age=Tapez un nombre entier de 2 chiffres au plus
21. Future.form19.dateInFuture=La date doit être postérieure à celle d''aujourd'hui
22. Past.form19.dateInPast=La date doit être antérieure à celle d''aujourd'hui
23. Size.form19.strBetween4and6=la chaîne doit avoir entre 4 et 6 caractères
24. Min.form19.intMin10=La valeur doit être supérieure ou égale à 10
25. Max.form19.intMax100=La valeur doit être inférieure ou égale à 100
26. Length.form19.str4=La chaîne doit avoir quatre caractères exactement
27. Email.form19.email=Adresse mail invalide
28. URL.form19.url=URL invalide
29. Range.form19.int1014=La valeur doit être dans l''intervalle [10,14]
30. AssertTrue=Seule la valeur True est acceptée
31. AssertFalse=Seule la valeur False est acceptée
32. Pattern.form19.hhmmss=Tapez l''heure sous la forme hh:mm:ss
33. NotEmpty=La donnée ne peut être vide
34. NotBlank=La donnée ne peut être vide
```

Voyons quelques exemples d'exécution :

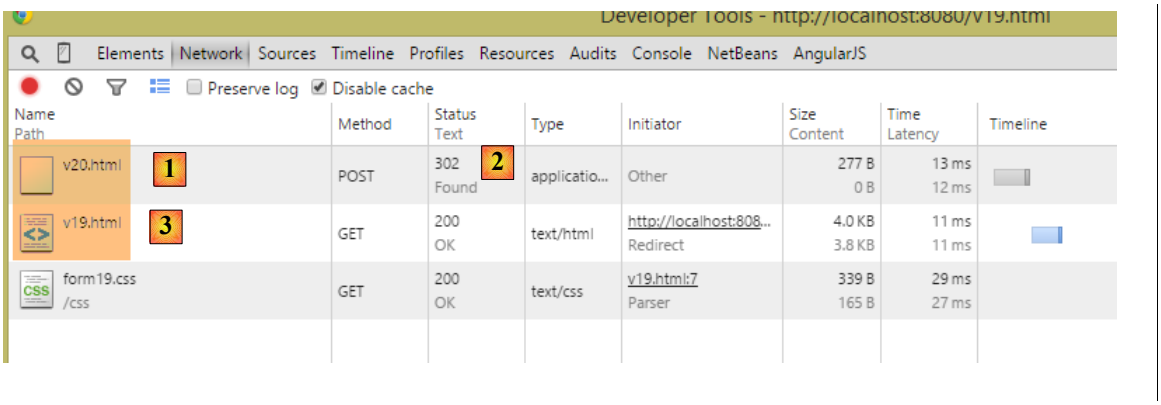

Spring 4 MVC
localhost:8080/v20.html
Applications
Favoris
Gestionnaire de favo...
Formulaire - Validations côté serveur
Contrainte
Saisie
Erreur
@NotEmpty
La donnée ne peut être vide
@NotBlank
La donnée ne peut être vide
@assertFalse
True
False
La donnée est obligatoire
@assertTrue
True
@Past
jj/mm/aaaa
La donnée est obligatoire
@Future
jj/mm/aaaa
La donnée est obligatoire
@Max
La donnée est obligatoire
@Min
La donnée est obligatoire
@Size
la chaîne doit avoir entre 4 et 6 caractères
@Pattern(hh:mm:ss)
Tapez l'heure sous la forme hh:mm:ss
@Email
La donnée ne peut être vide
@Length
La chaîne doit avoir quatre caractères exactement
@Range
La donnée est obligatoire
@URL
La donnée ne peut être vide
Valider


Spring 4 MVC
localhost:8080/v20.html
Applications
Favoris
Gestionnaire de favo...
Formulaire - Validations côté serveur
Contrainte
Saisie
Erreur
@NotEmpty
x
@NotBlank
x
@assertFalse
True
False
Seule la valeur False est acceptée
@assertTrue
False
Seule la valeur True est acceptée
@Past
10/12/2014
La date doit être antérieure à celle d'aujourdhui
@Future
09/12/2014
La date doit être postérieure à celle d'aujourdhui
@Max
1
@Min
1
La valeur doit être supérieure ou égale à 10
@Size
1
la chaîne doit avoir entre 4 et 6 caractères
@Pattern(hh:mm:ss)
x
Tapez l'heure sous la forme hh:mm:ss
@Email
x
Adresse mail invalide
@Length
x
La chaîne doit avoir quatre caractères exactement
@Range
1
La valeur doit être dans l'intervalle [10,14]
@URL
x
URL invalide
Valider





Ci-dessus, entre [1] et [2], on a l'impression qu'il ne s'est rien passé. Si on regarde les échanges réseau (Ctrl-Maj-I), on voit pourtant qu'il y a eu deux échanges réseau avec le serveur :



- en [1], le POST initial vers [/v20] ;
- en [2], la réponse de cette action est une redirection ;
- en [3], la seconde requête vers [/v19] cette fois ;

L'action [/v19] est alors exécutée :

```
1.      // ------------------ affichage d'un formulaire
2.      @RequestMapping(value = "/v19", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
3.      public String v19(Form19 formulaire) {
4.          return "vue-19";
5.  }
```

- ligne 3, le paramètre [Form19 formulaire] est initialisé avec l'attribut Flash de clé [form19] qui avait été créé par l'action précédente [/v19] et qui était un objet de type [Form19] avec pour valeurs, les valeurs postées à l'action [/v19] ;

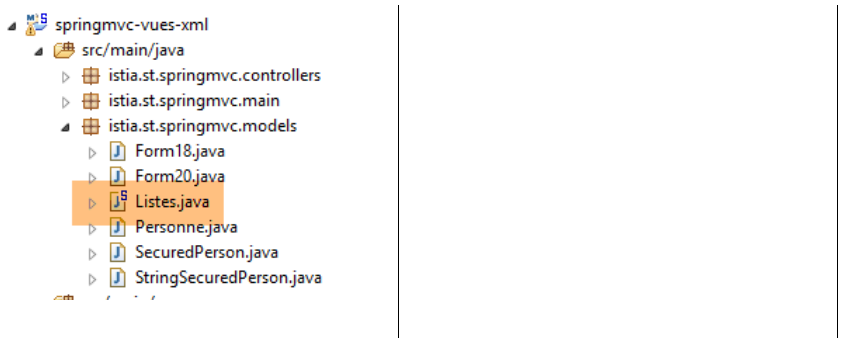
- ligne 4 : la vue [vue-19.xml] va être affichée avec dans son modèle un objet [Form19 formulaire] initialisé avec les valeurs postées. C'est pourquoi, l'utilisateur retrouve le formulaire tel qu'il l'a posté ;

Pourquoi une redirection ? Pourquoi n'a-t-on pas posté simplement à l'action [/v19] ci-dessus ? On aurait eu le même le résultat. A quelques différences près :

- le navigateur aurait mis dans son champ d'adresse [http://localhost:8080/v20.html] au lieu de [http://localhost:8080/v19.html] comme il l'a fait ici, car il affiche la dernière URL appelée ;
- si l'utilisateur fait un rafraîchissement de la page (F5), on n'a pas du tout le même résultat :
  - dans le cas de la redirection, l'URL affichée est [http://localhost:8080/v19.html] obtenue avec un GET. Le navigateur rejouera cette dernière commande et il obtiendra alors un formulaire tout neuf (l'attribut Flash n'est utilisé qu'une fois),
  - dans le cas de la non redirection, l'URL affichée est [http://localhost:8080/v20.html] obtenue avec un POST. Le navigateur rejouera cette dernière commande et donc fera de nouveau un POST avec les mêmes valeurs postées que précédemment. Ici ça ne porte pas à conséquence mais c'est souvent indésirable et donc on préfèrera en général la redirection ;

5.15 [/v21-/v22] : gérer des boutons radio

Considérons le composant Spring [Listes] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import org.springframework.stereotype.Component;
4.
5.  @Component
6.  public class Listes {
7.
8.      private String[] deplacements = new String[] { "0", "1", "2", "3", "4" };
9.      private String[] libellesDeplacements = new String[] { "vélo", "marche", "train", "avion", "autre" };
10.     private String[] libellesBijoux = new String[] { "émeraude", "rubis", "diamant", "opaline" };
11.
12.     // getters et setters
13.    ...
14.
15. }
```

- ligne 5 : la classe [Listes] sera un composant Spring ;
- lignes 8-10 : des listes utilisées pour alimenter des boutons radio, des cases à cocher et des listes déroulantes ;

Dans la classe de configuration [Config], il est écrit :

```
1.  @Configuration
2.  @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
3.  @EnableAutoConfiguration
4.  public class Config extends WebMvcConfigurerAdapter {
```

- ligne 2 : le package [models] où se trouve le composant [Listes] sera bien exploré par Spring ;

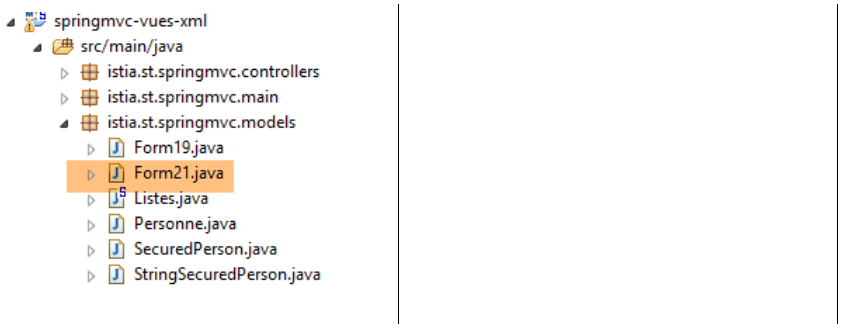
Nous créons les nouvelles actions suivantes :

```
1.      // ------------------ formulaire avec boutons radio
2.      @Autowired
```

```
3.      private Listes listes;
4.
5.      @RequestMapping(value = "/v21", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
6.      public String v21(@ModelAttribute("form") Form21 formulaire, Model model) {
7.          model.addAttribute("listes", listes);
8.          return "vue-21";
9.      }
10.
11.     @RequestMapping(value = "/v22", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
12.     public String v22(@ModelAttribute("form") Form21 formulaire, RedirectAttributes redirectAttributes) {
13.         redirectAttributes.addFlashAttribute("form", formulaire);
14.         return "redirect:/v21.html";
15. }
```

- lignes 2-3 : le composant [Listes] est injecté dans le contrôleur ;
- ligne 6 : nous gérons un formulaire de type [Form21] que nous allons décrire. A noter qu'on a précisé sa clé [form] dans le modèle de la vue. On rappelle que par défaut, cela aurait été [form21] ;
- ligne 7 : on injecte le composant [Listes] dans le modèle. La vue va en avoir besoin ;
- ligne 8 : on affiche la vue [vue-21.xml]. Cette vue va afficher le formulaire [Form21] et les valeurs postées le seront à l'action [/v22] des lignes 12-15 ;
- lignes 12-15 : l'action [/v22] se contente d'une redirection vers l'action [/v21] en mettant les valeurs postées qu'elle a reçues dans un attribut Flash de clé [form]. Il est important que cette clé soit la même que celle utilisée ligne 6 ;

Le modèle [Form21] est le suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  public class Form21 {
4.
5.      // valeurs postées
6.      private String marie = "non";
7.      private String deplacement = "4";
8.      private String[] couleurs;
9.      private String strCouleurs;
10.     private String[] bijoux;
11.     private String strBijoux;
12.     private int couleur2;
13.     private int[] bijoux2;
14.     private String strBijoux2;
15.
16.     // getters et setters
17.     ...
18. }
```

La vue [vue-21.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title>Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.          <link rel="stylesheet" href="/css/form19.css" />
7.      </head>
8.      <body>
9.
10.         <h3>Formulaire - Boutons radio</h3>
11.         <form action="/someURL" th:action="@{/v22.html}" method="post" th:object="${form}">
```

```
12.          <table>
13.             <thead>
14.                <tr>
15.                   <th class="col1">Texte</th>
16.                   <th class="col2">Saisie</th>
17.                   <th class="col3">Valeur</th>
18.                </tr>
19.             </thead>
20.             <tbody>
21.                <tr>
22.                   <td class="col1">Etes-vous marié(e)</td>
23.                   <td class="col2">
24.                      <input type="radio" th:field="*{marie}" value="oui" />
25.                      <label th:for="${#ids.prev('marie')}">Oui</label>
26.                      <input type="radio" th:field="*{marie}" value="non" />
27.                      <label th:for="${#ids.prev('marie')}">Non</label>
28.                   </td>
29.                   <td class="col3">
30.                      <span th:text="*{marie}"></span>
31.                   </td>
32.                </tr>
33.                <tr>
34.                   <td class="col1">Mode de déplacement</td>
35.                   <td class="col2">
36.                      <span th:each="mode, status : ${listes.deplacements}">
37.                         <input type="radio" th:field="*{deplacement}" th:value="${mode}" />
38.                         <label th:for="${#ids.prev('deplacement')}" th:text="$
    {listes.libellesDeplacements[status.index]}">Autre</label>
39.                      </span>
40.                   </td>
41.                   <td class="col3">
42.                      <span th:text="*{deplacement}"></span>
43.                   </td>
44.                </tr>
45.             </tbody>
46.          </table>
47.          <p>
48.             <input type="submit" value="Valider" />
49.          </p>
50.       </form>
51.    </body>
52. </html>
```

- lignes 36-40 : on notera l'exploitation du composant [Listes] mis dans le modèle, pour générer les libellés des cases à cocher ;
- la colonne 3 permet de connaître la valeur postée pour un POST, ou la valeur initiale du formulaire lors du GET initial ;

Ce code affiche la page suivante :

correspondant au code HTML suivant :

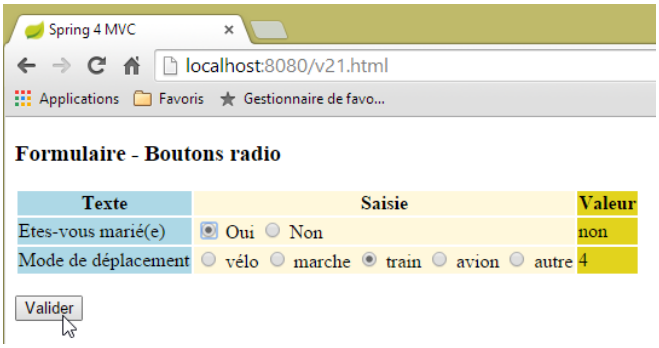
```
1.  <!DOCTYPE HTML>
2.
```

```
3.  <html>
4.     <head>
5.        <title>Spring 4 MVC</title>
6.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.        <link rel="stylesheet" href="/css/form19.css" />
8.     </head>
9.     <body>
10.
11.       <h3>Formulaire - Boutons radio</h3>
12.       <form action="/v22.html" method="post">
13.          <table>
14.             <thead>
15.                <tr>
16.                   <th class="col1">Texte</th>
17.                   <th class="col2">Saisie</th>
18.                   <th class="col3">Valeur</th>
19.                </tr>
20.             </thead>
21.             <tbody>
22.                <tr>
23.                   <td class="col1">Etes-vous marié(e)</td>
24.                   <td class="col2">
25.                      <input type="radio" value="oui" id="marie1" name="marie" />
26.                      <label for="marie1">Oui</label>
27.                      <input type="radio" value="non" id="marie2" name="marie" checked="checked" />
28.                      <label for="marie2">Non</label>
29.                   </td>
30.                   <td class="col3">
31.                      <span>non</span>
32.                   </td>
33.                </tr>
34.                <tr>
35.                   <td class="col1">Mode de déplacement</td>
36.                   <td class="col2">
37.                      <span>
38.                         <input type="radio" value="0" id="deplacement1" name="deplacement" />
39.                         <label for="deplacement1">vélo</label>
40.                      </span>
41.                      <span>
42.                         <input type="radio" value="1" id="deplacement2" name="deplacement" />
43.                         <label for="deplacement2">marche</label>
44.                      </span>
45.                      <span>
46.                         <input type="radio" value="2" id="deplacement3" name="deplacement" />
47.                         <label for="deplacement3">train</label>
48.                      </span>
49.                      <span>
50.                         <input type="radio" value="3" id="deplacement4" name="deplacement" />
51.                         <label for="deplacement4">avion</label>
52.                      </span>
53.                      <span>
54.                         <input type="radio" value="4" id="deplacement5" name="deplacement"
    checked="checked" />
55.                         <label for="deplacement5">autre</label>
56.                      </span>
57.                   </td>
58.                   <td class="col3">
59.                      <span>4</span>
60.                   </td>
61.                </tr>
62.             </tbody>
63.          </table>
64.          <p>
65.             <input type="submit" value="Valider" />
66.          </p>
67.       </form>
68.    </body>
69. </html>
```

On voit que les valeurs postées (attributs name) le sont dans les champs suivants du modèle [Form21] :

```
   private String marie = "non";
   private String deplacement = "4";
```

Le lecteur est invité à faire des tests. On notera bien que c'est l'attribut [value] des boutons radio qui est posté.



5.16 [/v23-/v24] : gérer des cases à cocher

Nous ajoutons la nouvelle action suivante :

```
1.      // ------------------ formulaire avec cases à cocher
2.      @RequestMapping(value = "/v23", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
3.      public String av20(@ModelAttribute("form") Form21 formulaire, Model model) {
4.          model.addAttribute("listes", listes);
5.          return "vue-23";
6.  }
```

- ligne 3 : nous continuons à utiliser le modèle [Form21] ;

La vue [vue-23.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title>Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.          <link rel="stylesheet" href="/css/form19.css" />
7.      </head>
8.      <body>
9.          <h3>Formulaire - Cases à cocher</h3>
10.         <form action="/someURL" th:action="@{/v24.html}" method="post" th:object="${form}">
11.             <table>
12.                 <thead>
13.                     <tr>
14.                         <th class="col1">Texte</th>
15.                         <th class="col2">Saisie</th>
16.                         <th class="col3">Valeur</th>
17.                     </tr>
18.                 </thead>
19.                 <tbody>
20.                     <tr>
21.                         <td class="col1">Vos couleurs préférées</td>
22.                         <td class="col2">
23.                             <input type="checkbox" th:field="*{couleurs}" value="0" />
24.                             <label th:for="${#ids.prev('couleurs')}">rouge</label>
25.                             <input type="checkbox" th:field="*{couleurs}" value="1" />
26.                             <label th:for="${#ids.prev('couleurs')}">vert</label>
27.                             <input type="checkbox" th:field="*{couleurs}" value="2" />
28.                             <label th:for="${#ids.prev('couleurs')}">bleu</label>
29.                         </td>
30.                         <td class="col3">
31.                             <span th:text="*{strCouleurs}"></span>
32.                         </td>
33.                     </tr>
34.                     <tr>
35.                         <td class="col1">Pierres préférées</td>
36.                         <td class="col2">
```

```
37.                         <span th:each="label, status : ${listes.libellesBijoux}">
38.                             <input type="checkbox" th:field="*{bijoux}" th:value="${status.index}" />
39.                             <label th:for="${#ids.prev('bijoux')}" th:text="${label}">Autre</label>
40.                         </span>
41.                     </td>
42.                     <td class="col3">
43.                         <span th:text="*{strBijoux}"></span>
44.                     </td>
45.                 </tr>
46.             </tbody>
47.         </table>
48.         <p>
49.             <input type="submit" value="Valider" />
50.         </p>
51.     </form>
52. </body>
53. </html>
```

- lignes 37-41 : on notera l'utilisation du composant [Listes] pour générer les libellés des cases à cocher ;

Ce code affiche la page suivante :

issue du code HTML suivant :

```
1.  <!DOCTYPE HTML>
2.
3.  <html>
4.      <head>
5.          <title>Spring 4 MVC</title>
6.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.          <link rel="stylesheet" href="/css/form19.css" />
8.      </head>
9.      <body>
10.         <h3>Formulaire - Cases à cocher</h3>
11.         <form action="/v24.html" method="post">
12.             <table>
13.                 <thead>
14.                     <tr>
15.                         <th class="col1">Texte</th>
16.                         <th class="col2">Saisie</th>
17.                         <th class="col3">Valeur</th>
18.                     </tr>
19.                 </thead>
20.                 <tbody>
21.                     <tr>
22.                         <td class="col1">Vos couleurs préférées</td>
23.                         <td class="col2">
24.                             <input type="checkbox" value="0" id="couleurs1" name="couleurs" /><input
    type="hidden" name="_couleurs" value="on" />
25.                             <label for="couleurs1">rouge</label>
26.                             <input type="checkbox" value="1" id="couleurs2" name="couleurs" /><input
    type="hidden" name="_couleurs" value="on" />
27.                             <label for="couleurs2">vert</label>
```

```
28.                         <input type="checkbox" value="2" id="couleurs3" name="couleurs" /><input
    type="hidden" name="_couleurs" value="on" />
29.                         <label for="couleurs3">bleu</label>
30.                      </td>
31.                      <td class="col3">
32.                         <span></span>
33.                      </td>
34.                   </tr>
35.                   <tr>
36.                      <td class="col1">Pierres préférées</td>
37.                      <td class="col2">
38.                         <span>
39.                            <input type="checkbox" value="0" id="bijoux1" name="bijoux" /><input
    type="hidden" name="_bijoux" value="on" />
40.                            <label for="bijoux1">émeraude</label>
41.                         </span>
42.                         <span>
43.                            <input type="checkbox" value="1" id="bijoux2" name="bijoux" /><input
    type="hidden" name="_bijoux" value="on" />
44.                            <label for="bijoux2">rubis</label>
45.                         </span>
46.                         <span>
47.                            <input type="checkbox" value="2" id="bijoux3" name="bijoux" /><input
    type="hidden" name="_bijoux" value="on" />
48.                            <label for="bijoux3">diamant</label>
49.                         </span>
50.                         <span>
51.                            <input type="checkbox" value="3" id="bijoux4" name="bijoux" /><input
    type="hidden" name="_bijoux" value="on" />
52.                            <label for="bijoux4">opaline</label>
53.                         </span>
54.                      </td>
55.                      <td class="col3">
56.                         <span></span>
57.                      </td>
58.                   </tr>
59.                </tbody>
60.             </table>
61.             <p>
62.                <input type="submit" value="Valider" />
63.             </p>
64.          </form>
65.       </body>
66. </html>
```

On notera que les valeurs postées (attributs *name*) le sont dans les champs suivants de [Form21] :

```
   private String[] couleurs;
   private String[] bijoux;
```

Ce sont des tableaux car pour chaque champ, il existe plusieurs cases à cocher portant le nom du champ. Il est donc possible que plusieurs valeurs postées arrivent avec le même nom (attribut *name* du formulaire). Il faut donc un tableau pour les récupérer.

Revenons au code Thymeleaf de la colonne 3 de la page :

```
1.    <td class="col3">
2.      <span th:text="*{strCouleurs}"></span>
3.    </td>
4.  </tr>
5.  <tr>
6.    <td class="col1">Pierres préférées</td>
7.    <td class="col2">
8.      <span th:each="label, status : ${listes.libellesBijoux}">
9.        <input type="checkbox" th:field="*{bijoux}" th:value="${status.index}" />
10.       <label th:for="${#ids.prev('bijoux')}" th:text="${label}">Autre</label>
11.     </span>
12.   </td>
13.   <td class="col3">
14.     <span th:text="*{strBijoux}"></span>
15.   </td>
16. </tr>
```

Les champs référencés lignes 2 et 14 sont les suivants :

```
    private String strCouleurs;
    private String strBijoux;
```

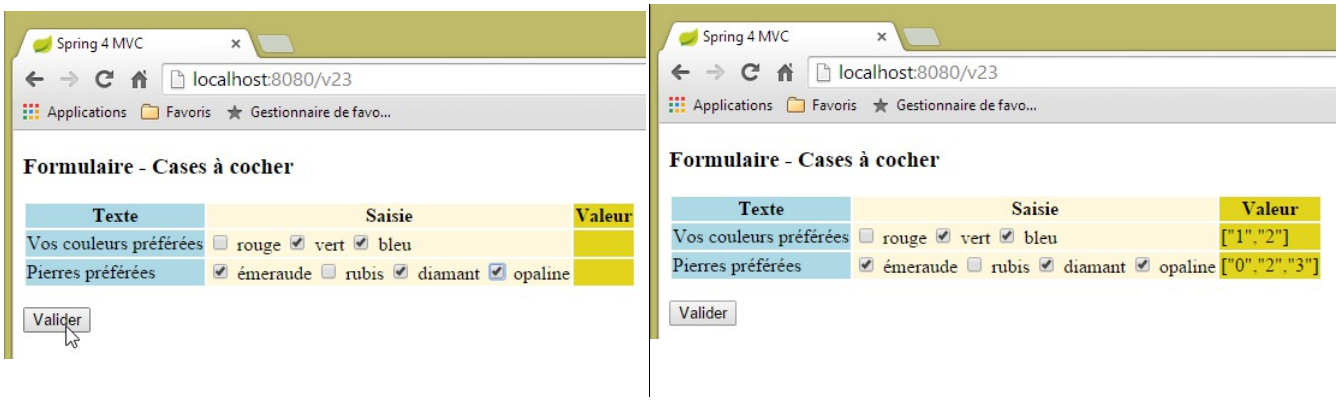
Ils sont calculés par l'action [/v24] qui gère le POST :

```
1.      // mappeur Jackson / jSON
2.      private ObjectMapper mapper = new ObjectMapper();
3.
4.      @RequestMapping(value = "/v24", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
5.      public String av21(@ModelAttribute("form") Form21 formulaire, RedirectAttributes redirectAttributes)
    throws JsonProcessingException {
6.          redirectAttributes.addFlashAttribute("form", formulaire);
7.          formulaire.setStrCouleurs(mapper.writeValueAsString(formulaire.getCouleurs()));
8.          formulaire.setStrBijoux(mapper.writeValueAsString(formulaire.getBijoux()));
9.          return "redirect:/v23.html";
10. }
```

Il faut se rappeler ici que la bibliothèque jackson / jSON est dans les dépendances du projet.

- ligne 2 : on crée un type [ObjectMapper] qui permet de sérialiser / désérialiser des objets en jSON,
- ligne 7 : on sérialise en jSON le tableau des couleurs. Le résultat est placé dans le champ [strCouleurs] ;
- ligne 8 : on sérialise en jSON le tableau des bijoux. Le résultat est placé dans le champ [strBijoux] ;

Voici un exemple d'exécution :



On notera bien que c'est l'attribut [value] des cases à cocher qui est posté.

5.17 [/25-/v26] : gérer des listes

Nous ajoutons l'action suivante [/v25] :

```
1.    // ------------------ formulaire avec listes
2.    @RequestMapping(value = "/v25", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
3.    public String v25(@ModelAttribute("form") Form21 formulaire, Model model) {
4.        model.addAttribute("listes", listes);
5.        return "vue-25";
6.  }
```

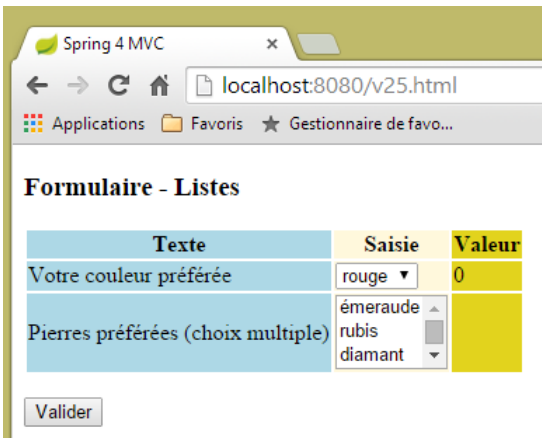
La vue [vue-25.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.      <head>
4.          <title>Spring 4 MVC</title>
5.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.          <link rel="stylesheet" href="/css/form19.css" />
7.      </head>
8.      <body>
9.
10.         <h3>Formulaire - Listes</h3>
```

```
11.         <form action="/someURL" th:action="@{/v26.html}" method="post"
12.            th:object="${form}">
13.            <table>
14.               <thead>
15.                  <tr>
16.                     <th class="col1">Texte</th>
17.                     <th class="col2">Saisie</th>
18.                     <th class="col3">Valeur</th>
19.                  </tr>
20.               </thead>
21.               <tbody>
22.                  <tr>
23.                     <td class="col1">Votre couleur préférée</td>
24.                     <td class="col2">
25.                        <select th:field="*{couleur2}">
26.                           <option value="0">rouge</option>
27.                           <option value="1">bleu</option>
28.                           <option value="2">vert</option>
29.                        </select>
30.                     </td>
31.                     <td class="col3">
32.                        <span th:text="*{couleur2}"></span>
33.                     </td>
34.                  </tr>
35.                  <tr>
36.                     <td class="col1">Pierres préférées (choix multiple)</td>
37.                     <td class="col2">
38.                        <select th:field="*{bijoux2}" multiple="multiple" size="3">
39.                           <option th:each="label, status : ${listes.libellesBijoux}"
40.                              th:text="${label}" th:value="${status.index}">
41.                           </option>
42.                        </select>
43.                     </td>
44.                     <td class="col3">
45.                        <span th:text="*{strBijoux2}"></span>
46.                     </td>
47.                  </tr>
48.
49.               </tbody>
50.            </table>
51.            <input type="submit" value="Valider" />
52.         </form>
53.      </body>
54. </html>
```

- lignes 38-42 : génération d'une liste à choix multiple où les libellés sont pris dans le composant [Listes] que nous avons déjà utilisé ;

La page affichée est la suivante :



générée par le code HTML suivant :

```
1.  <!DOCTYPE HTML>
2.
3.  <html>
4.     <head>
5.        <title>Spring 4 MVC</title>
6.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.        <link rel="stylesheet" href="/css/form19.css" />
8.     </head>
9.     <body>
10.
11.       <h3>Formulaire - Listes</h3>
12.       <form action="/v26.html" method="post">
13.          <table>
14.             <thead>
15.                <tr>
16.                   <th class="col1">Texte</th>
17.                   <th class="col2">Saisie</th>
18.                   <th class="col3">Valeur</th>
19.                </tr>
20.             </thead>
21.             <tbody>
22.                <tr>
23.                   <td class="col1">Votre couleur préférée</td>
24.                   <td class="col2">
25.                      <select id="couleur2" name="couleur2">
26.                         <option value="0" selected="selected">rouge</option>
27.                         <option value="1">bleu</option>
28.                         <option value="2">vert</option>
29.                      </select>
30.                   </td>
31.                   <td class="col3">
32.                      <span>0</span>
33.                   </td>
34.                </tr>
35.                <tr>
36.                   <td class="col1">Pierres préférées (choix multiple)</td>
37.                   <td class="col2">
38.                      <select multiple="multiple" size="3" id="bijoux2" name="bijoux2">
39.                         <option value="0">émeraude</option>
40.                         <option value="1">rubis</option>
41.                         <option value="2">diamant</option>
42.                         <option value="3">opaline</option>
43.                      </select>
44.                      <input type="hidden" name="_bijoux2" value="1" />
45.                   </td>
46.                   <td class="col3">
47.                      <span></span>
48.                   </td>
49.                </tr>
50.             </tbody>
51.          </table>
52.          <p>
53.             <input type="submit" value="Valider" />
54.          </p>
55.       </form>
56.    </body>
57. </html>
```

- ligne 44 : on peut remarquer que Thymeleaf a créé un champ caché. Je n'ai pas compris son rôle :
- les valeurs postées (attributs *value* des balises *option*) le seront dans les champs suivants (attributs *name*) de [Form21] :

```
   private int couleur2;
   private int[] bijoux2;
```

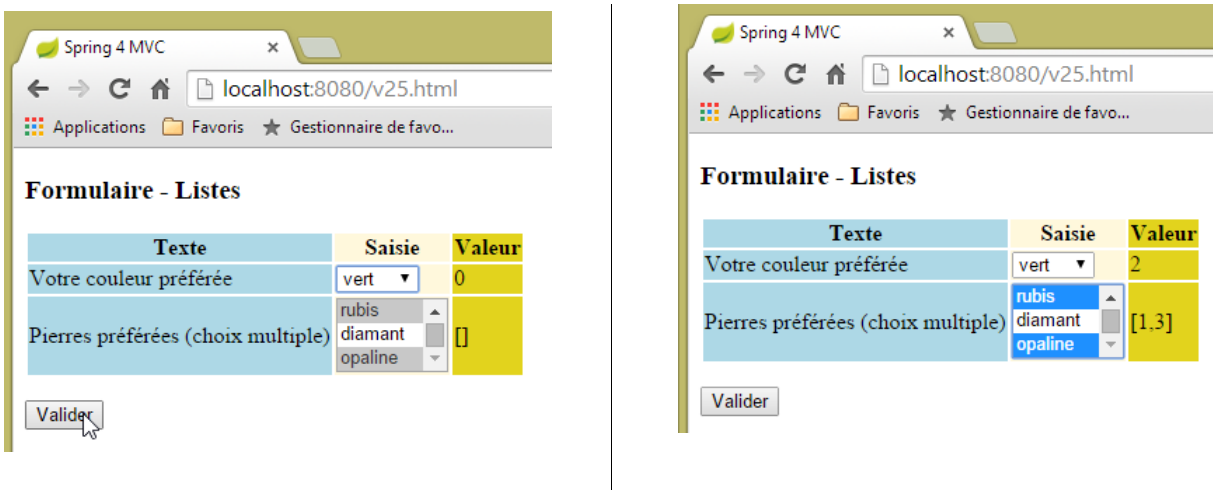
- ligne 38 : la liste [bijoux2] est à choix multiple. Donc plusieurs valeurs peuvent être postées associées au nom [bijoux2]. Pour les récupérer, le champ [bijoux2] doit être un tableau. On remarquera que c'est un tableau d'entiers. C'est possible puisque les valeurs postées peuvent être converties dans ce type ;

Les valeurs sont postées à l'action [/v26] suivante :

```
1.    @RequestMapping(value = "/v26", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
```

```
2.    public String v26(@ModelAttribute("form") Form21 formulaire, RedirectAttributes redirectAttributes)
    throws JsonProcessingException {
3.       redirectAttributes.addFlashAttribute("form", formulaire);
4.       formulaire.setStrBijoux2(mapper.writeValueAsString(formulaire.getBijoux2()));
5.       return "redirect:/v25.html";
6.    }
```

Il n'y a là rien qu'on n'ait déjà vu. Voici un exemple d'exécution :



5.18 [/v27] : paramétrage des messages

Considérons l'action [/v27] suivante :

```
1.    // ------------------ messages paramétrés
2.    @RequestMapping(value = "/v27", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
3.    public String v27(Model model) {
4.        model.addAttribute("param1","paramètre un");
5.        model.addAttribute("param2","paramètre deux");
6.        model.addAttribute("param3","paramètre trois");
7.        model.addAttribute("param4","messages.param4");
8.        return "vue-27";
9.    }
```

L'action se contente de mettre quatre valeurs dans le modèle et fait afficher la vue [vue-27.xml] suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title th:text="#{messages.titre}">Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <h2 th:text="#{messages.titre}">Spring 4 MVC</h2>
9.        <p th:text="#{messages.msg1(${param1})}"></p>
10.       <p th:text="#{messages.msg2(${param2},${param3})}"></p>
11.       <p th:text="#{messages.msg3(#{${param4}})}"></p>
12.    </body>
13. </html>
```

- ligne 8 : un message sans paramètres ;
- ligne 9 : un message avec un paramètre [$param1] pris dans le modèle ;
- ligne 10 : un message avec deux paramètres [$param2, $param3] pris dans le modèle ;
- ligne 11 : un message avec un paramètre. Ce paramètre est lui-même une clé de message (présence de #). La clé est fournie par [$param4] ;

Le fichier des messages français est le suivant :

[messages_fr.properties]

```
1. messages.titre=Messages paramétrés
2. messages.msg1=Un message avec un paramètre : {0}
3. messages.msg2=Un message avec deux paramètres : {0}, {1}
4. messages.msg3=Un message avec une clé de message comme paramètre : {0}
5. messages.param4=paramètre quatre
```

Pour indiquer la présence de paramètres dans le message, on utilise les symboles {0}, {1}, ...

La fusion du modèle construit par l'action [/v27] avec la vue [vue-27] va produire le code HTML suivant :

```
1.  <!DOCTYPE html>
2.
3.  <html>
4.     <head>
5.        <title>Messages paramétrés</title>
6.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.     </head>
8.     <body>
9.        <h2>Messages paramétrés</h2>
10.       <p>Un message avec un paramètre : paramètre un</p>
11.       <p>Un message avec deux paramètre : paramètre deux, paramètre trois</p>
12.       <p>Un message avec une clé de message comme paramètre : paramètre quatre</p>
13.    </body>
14. </html>
```

ce qui donne la vue suivante :

Le fichier des messages anglais est le suivant :

[messages_fr.properties]

```
1. messages.titre=Parameterized messages
2. messages.msg1=Message with one parameter: {0}
3. messages.msg2=Message with two parameters: {0}, {1}
4. messages.msg3=Message with a message key as a parameter: {0}
5. messages.param4=parameter four
```

La fusion du modèle construit par l'action [/v27] avec la vue [vue-27] va produire le code HTML suivant :

```
1.  <!DOCTYPE html>
2.
3.  <html>
4.     <head>
5.        <title>Parameterized messages</title>
6.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
7.     </head>
8.     <body>
9.        <h2>Parameterized messages</h2>
10.       <p>Message with one parameter: paramètre un</p>
11.       <p>Message with two parameters: paramètre deux, paramètre trois</p>
```

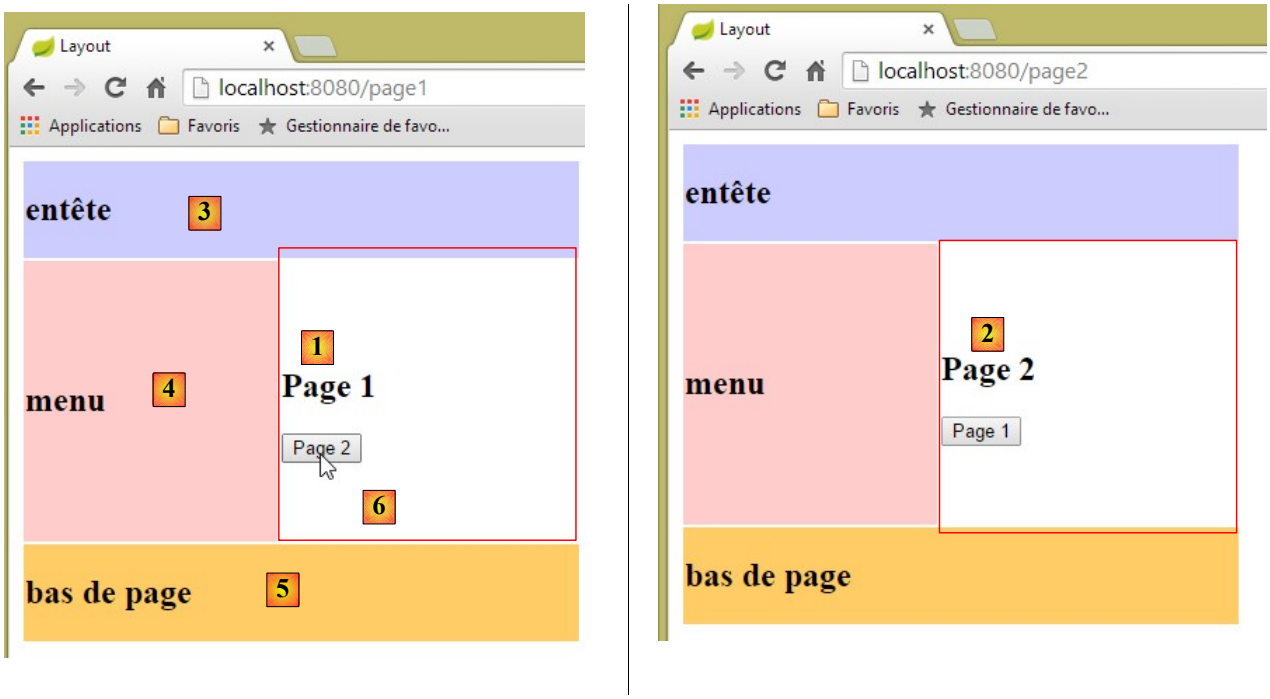
```
12.         <p>Message with a message key as a parameter: parameter four</p>
13.     </body>
14. </html>
```

ce qui donne la vue suivante :

On voit que le dernier message a été internationalisé de bout en bout, ce qui n'est pas le cas des deux précédents.

5.19 Utilisation d'une page maître

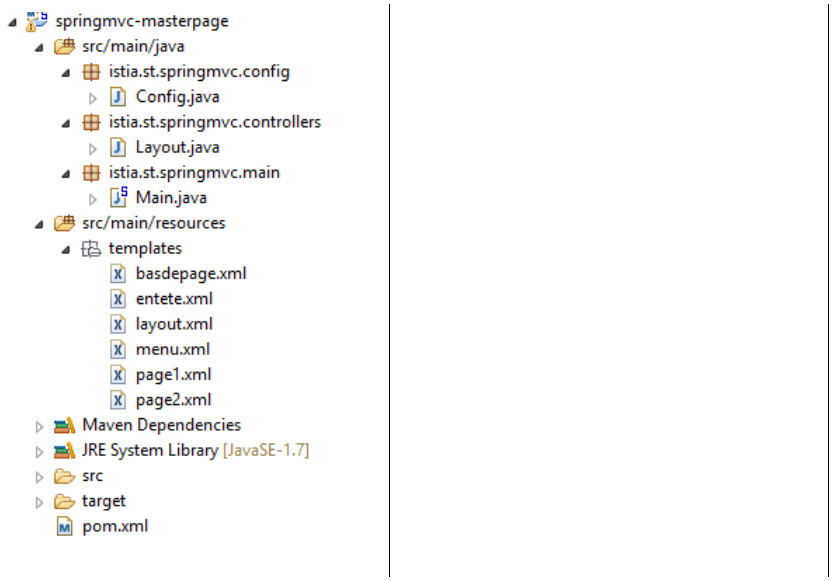
Dans une application web, il est fréquent que les vues partagent un certain nombre d'éléments qu'on peut factoriser dans une page maître. Voici un exemple :



Ci-dessus, on a deux pages semblables où le fragment [1] a été remplacé par le fragment [2]. La vue est celle d'une page maître ayant trois fragments fixes [3-5] et un fragment variable [6].

5.19.1 Le projet

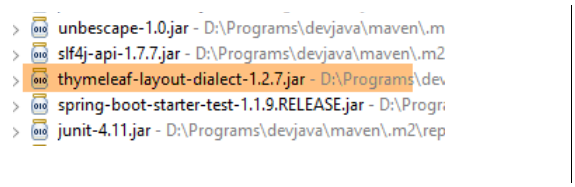
Nous construisons un projet [springmvc-masterpage] en suivant la démarche du paragraphe 5.1, page 138.



Le fichier [pom.xml] est le suivant :

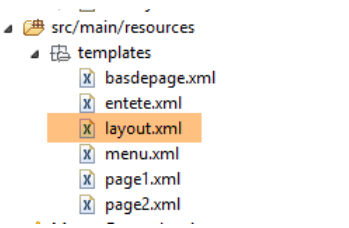
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>istia.st.springmvc</groupId>
7.     <artifactId>springmvc-masterpage</artifactId>
8.     <version>0.0.1-SNAPSHOT</version>
9.     <packaging>jar</packaging>
10.
11.    <name>springmvc-masterpage</name>
12.    <description>Page maître</description>
13.
14.    <parent>
15.       <groupId>org.springframework.boot</groupId>
16.       <artifactId>spring-boot-starter-parent</artifactId>
17.       <version>1.1.9.RELEASE</version>
18.       <relativePath/> <!-- lookup parent from repository -->
19.    </parent>
20.
21.    <dependencies>
22.       <dependency>
23.          <groupId>org.springframework.boot</groupId>
24.          <artifactId>spring-boot-starter-thymeleaf</artifactId>
25.       </dependency>
26.       <dependency>
27.          <groupId>org.springframework.boot</groupId>
28.          <artifactId>spring-boot-starter-test</artifactId>
29.          <scope>test</scope>
30.       </dependency>
31.    </dependencies>
32.
33.    <properties>
34.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
35.       <start-class>istia.st.springmvc.main.Main</start-class>
36.       <java.version>1.7</java.version>
37.    </properties>
38.
39.    <build>
40.       <plugins>
41.          <plugin>
42.             <groupId>org.springframework.boot</groupId>
43.             <artifactId>spring-boot-maven-plugin</artifactId>
44.          </plugin>
45.       </plugins>
46.    </build>
47.
48. </project>
```

L'une des dépendances amenées par ce fichier est nécessaire pour la page maître :



Les packages [config] et [main] sont identique à ceux de mêmes noms du projet précédent.

5.19.2 La page maître

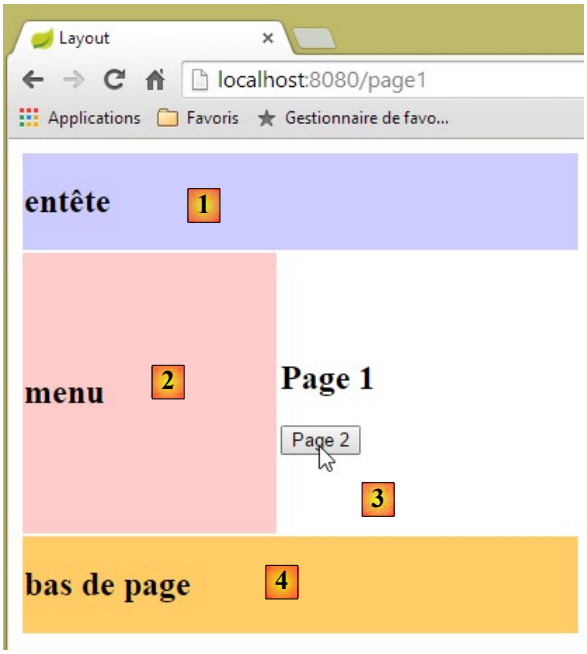


La page maître est la vue [layout.xml] suivante :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org" xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <title>Layout</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.     </head>
7.     <body>
8.        <table style="width: 400px">
9.           <tr>
10.             <td colspan="2" bgcolor="#ccccff">
11.                <div th:include="entete" />
12.             </td>
13.          </tr>
14.          <tr style="height: 200px">
15.             <td bgcolor="#ffcccc">
16.                <div th:include="menu" />
17.             </td>
18.             <td>
19.                <section layout:fragment="contenu">
20.                   <h2>Contenu</h2>
21.                </section>
22.             </td>
23.          </tr>
24.          <tr bgcolor="#ffcc66">
25.             <td colspan="2">
26.                <div th:include="basdepage" />
27.             </td>
28.          </tr>
29.       </table>
30.    </body>
31. </html>
```

- ligne 2 : la page maître doit définir l'espace de noms [xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"] dont un élément est utilisé ligne 19 ;
- lignes 10-12 : génèrent la zone [1] ci-dessous. La balise Thymeleaf [th:include] permet d'inclure dans la vue courante un fragment défini dans un autre fichier. Cela permet de factoriser les fragments utilisés dans plusieurs vues ;
- lignes 15-17 : génèrent la zone [2] ci-dessous ;
- lignes 19-20 : génèrent la zone [3] ci-dessous. L'attribut [layout:fragment] est un attribut de l'espace de noms [xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"]. Il indique une zone qui à l'exécution peut être remplacée par une autre ;

- lignes 24-28 : génèrent la zone [4] ci-dessous ;



5.19.3 Les fragments

Les fragments [*entete.xml*], [*menu.xml*] et [*basdepage.xml*] sont les suivants :

[**entete.xml**]

```
<!DOCTYPE html>
<html>
   <h2>entête</h2>
</html>
```

[**menu.xml**]

```
<!DOCTYPE html>
<html>
   <h2>menu</h2>
</html>
```

[**basdepage.xml**]

```
<!DOCTYPE html>
<html>
   <h2>bas de page</h2>
</html>
```

Le fragment [**page1.xml**] est le suivant :

```
1. <!DOCTYPE html>
2. <html xmlns:th="http://www.thymeleaf.org"
   xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout" layout:decorator="layout">
3.    <section layout:fragment="contenu">
4.       <h2>Page 1</h2>
5.       <form action="/someURL" th:action="@{/page2.html}" method="post">
6.          <input type="submit" value="Page 2" />
7.       </form>
8.    </section>
```

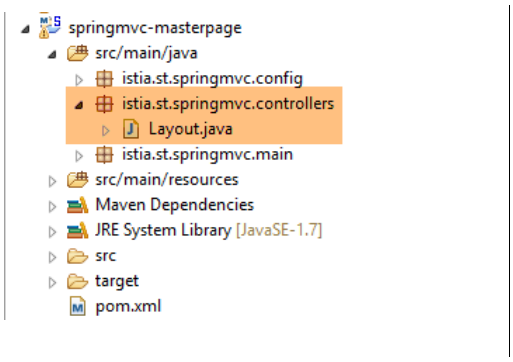
```
9.  </html>
```

- ligne 2 : l'attribut [layout:decorator="layout"] indique que la page courante [page1.xml] est 'décorée', ç-à-d. qu'elle appartient à une page maître. Celle-ci est la valeur de l'attribut, ici la vue [layout.xml] ;
- ligne 3 : on indique dans quel fragment de la page maître va venir s'insérer [page1.xml]. L'attribut [layout:fragment="contenu"] indique que [page1.xml] va s'insérer dans le fragment appelé [contenu], ç-à-d. la zone [3] de la page maître ;
- lignes 5-7 : le contenu du fragment est un formulaire qui offre un bouton de POST vers l'action [/page2.html] ;

Le fragment [**page2.xml**] est analogue :

```
1.  <!DOCTYPE html>
2.  <html xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
3.     layout:decorator="layout">
4.     <section layout:fragment="contenu">
5.        <h2>Page 2</h2>
6.        <form action="/someURL" th:action="@{/page1.html}" method="post">
7.           <input type="submit" value="Page 1" />
8.        </form>
9.     </section>
10. </html>
```

5.19.4 Les actions



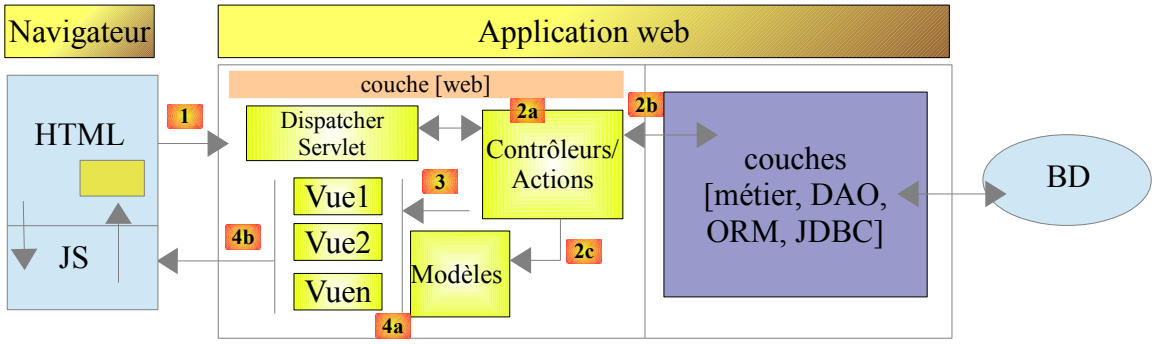
Le contrôleur [Layout.java] est le suivant :

```
1.  package istia.st.springmvc.controllers;
2.
3.  import org.springframework.stereotype.Controller;
4.  import org.springframework.web.bind.annotation.RequestMapping;
5.  import org.springframework.web.bind.annotation.RequestMethod;
6.
7.  @Controller
8.  public class Layout {
9.     @RequestMapping(value = "/page1")
10.    public String page1() {
11.       return "page1";
12.    }
13.
14.    @RequestMapping(value = "/page2", method=RequestMethod.POST)
15.    public String page2() {
16.       return "page2";
17.    }
18. }
```

- lignes 10-12 : l'action [/page1] se contente de faire afficher la vue [page1.xml] ;
- lignes 15-17 : idem pour l'action [/page2] qui fait afficher la vue [page2.xml] ;

6 Validation Javascript côté client

Dans le chapitre précédent nous nous sommes intéressés à la validation côté serveur. Revenons à l'architecture d'une application Spring MVC :



Pour l'instant, les pages envoyées au client ne contenaient pas de Javascript. Nous abordons maintenant cette technologie qui va nous permettre dans un premier temps de faire des validations côté client. Le principe est le suivant :

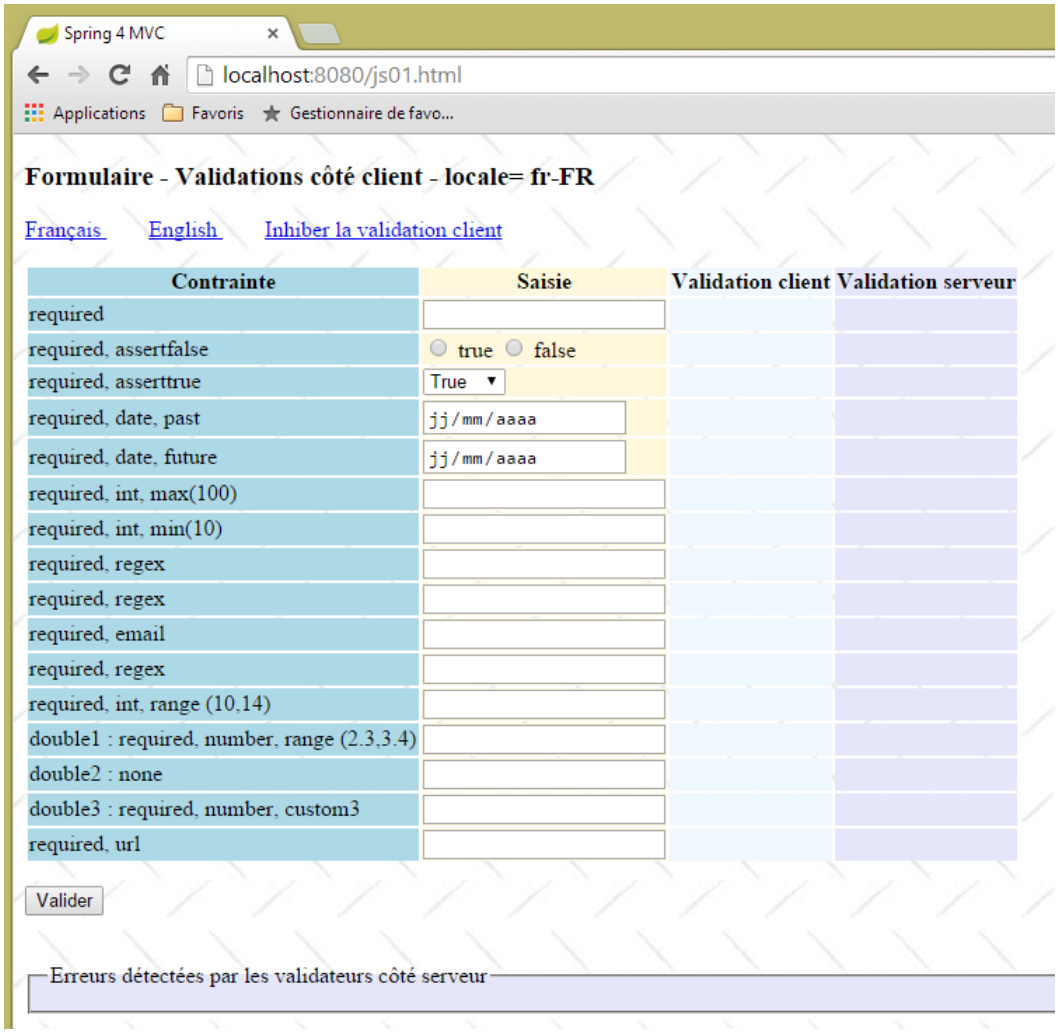
- c'est le Javascript qui poste les valeurs au serveur web ;
- et donc avant ce POST, il peut vérifier la validité des données et empêcher le POST si celles-ci sont invalides ;

Nous allons utiliser le formulaire que nous avons validé côté serveur. Nous allons maintenant offrir la possibilité de le valider à la fois côté client et côté serveur.

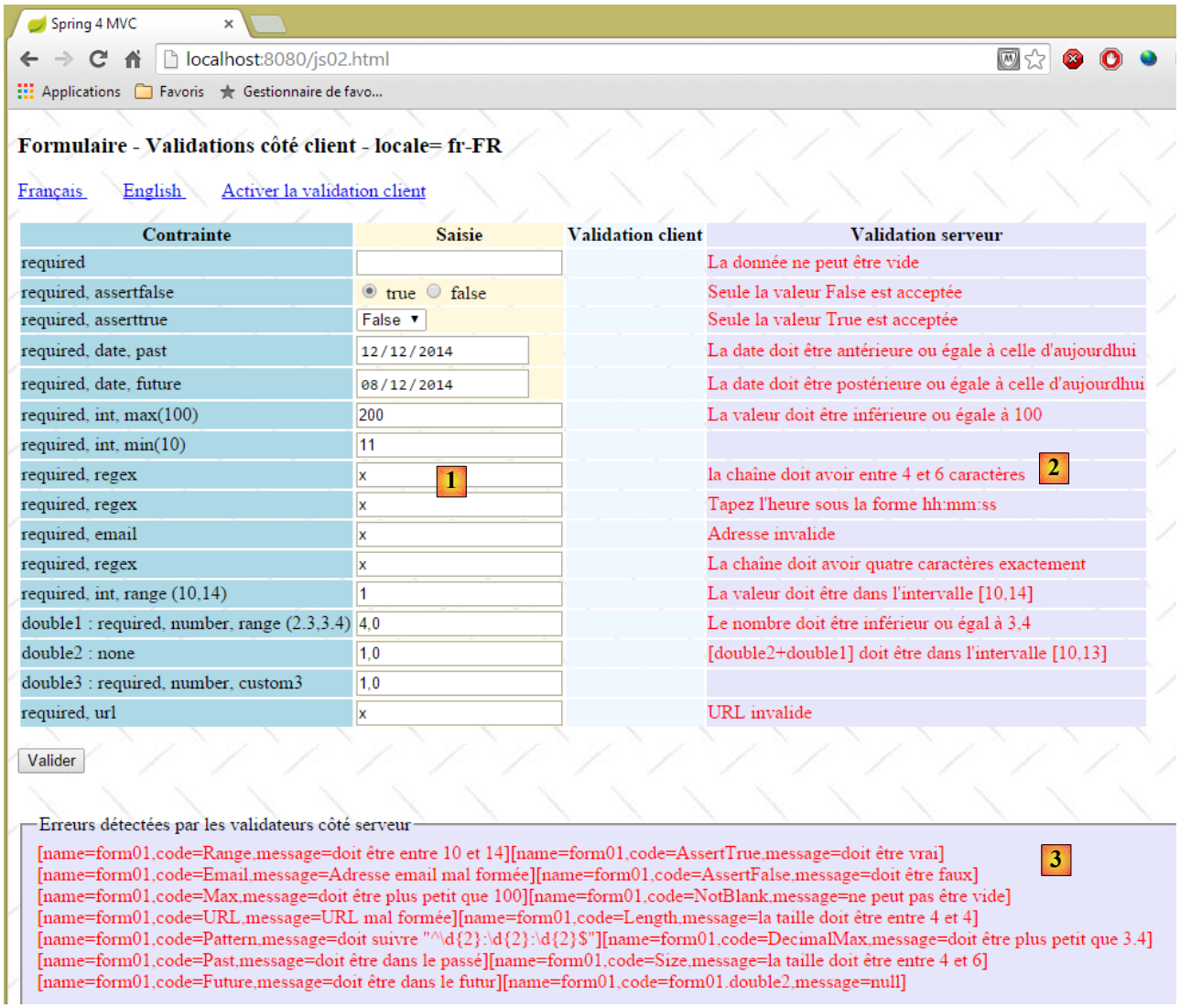
Note : le sujet est complexe. Le lecteur non intéressé par ce thème peut passer directement au paragraphe 7, page 253.

6.1 Les fonctionnalités du projet

Nous présentons quelques vues du projet pour présenter ses fonctionnalités. La page initiale est obtenue avec l'URL [http://localhost:8080/js01.html]

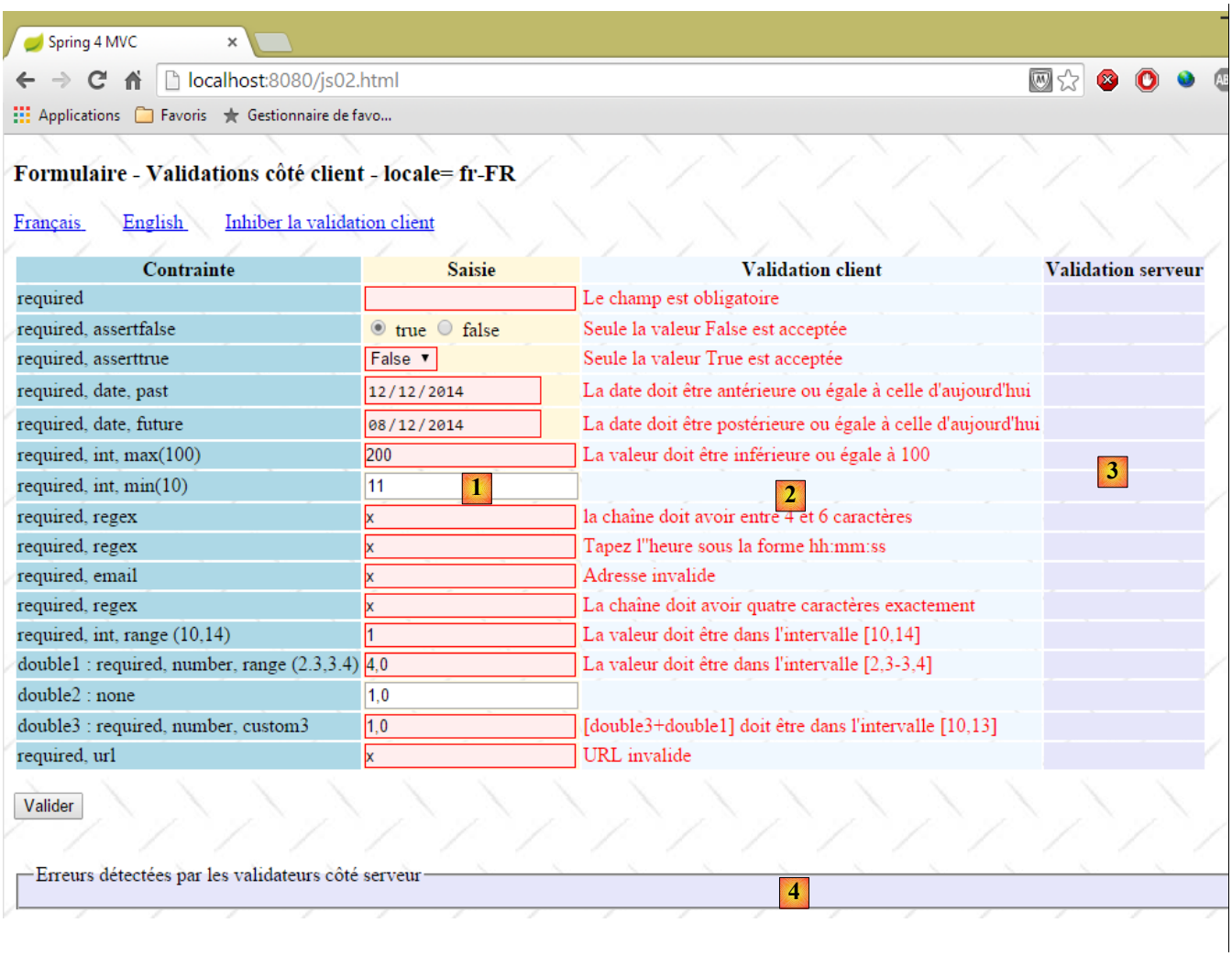


Les validations ont été mises en place des deux côtés : client et serveur. Comme le POST n'a lieu que si les valeurs ont été considérées comme valides côté client, les validations côté serveur réussissent tout le temps. On a donc offert un lien pour désactiver les validations côté client. Lorsqu'on est dans ce mode, on retrouve le mode de fonctionnement que nous avons déjà étudié. Voici un exemple :



- en [1], les valeurs saisies ;
- en [2], les messages d'erreur liées aux saisies ;
- en [3], un récapitulatif des erreurs avec pour chacune d'elles :
  - le nom du champ validé,
  - le code d'erreur,
  - le message par défaut de ce code d'erreur ;

Maintenant, autorisons la validation côté client :

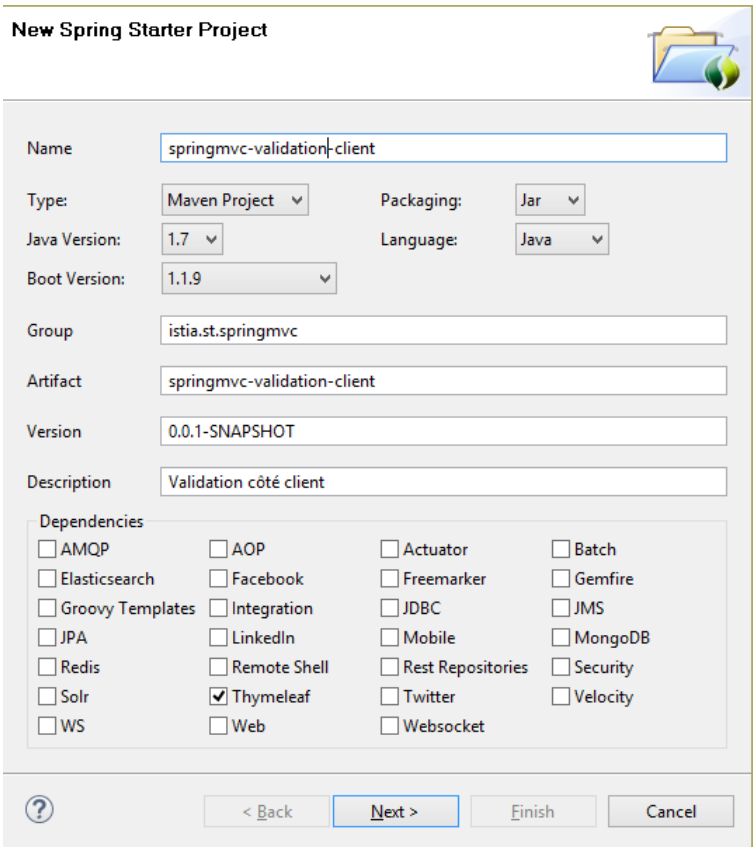


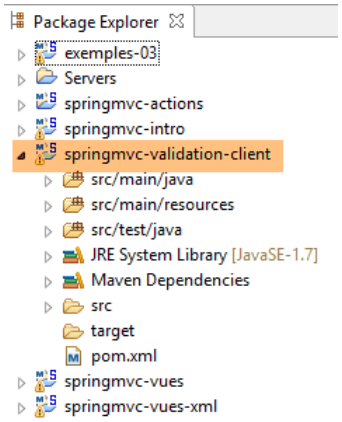
- en [1], les valeurs saisies. On peut remarquer que les saisies erronées ont un style particulier ;
- en [2], les messages d'erreur associés aux saisies erronées. Ils sont identiques à ceux générés par le serveur ;
- en [3-4], il n'y a plus rien car tant qu'il y a des saisies erronées, le POST vers le serveur n'a pas lieu ;

6.2 Validation côté serveur

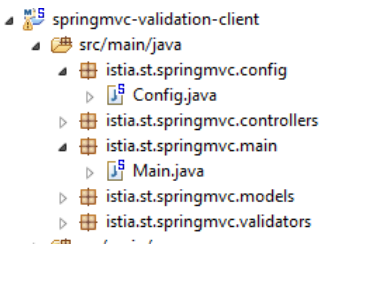
6.2.1 Configuration

Nous commençons par créer un nouveau projet Maven [springmvc-validation-client] :





Nous faisons évoluer le projet de la façon suivante :



La classe [Config] configure le projet. Elle est identique à ce qu'elle était dans les projets précédents :

```
1.  package istia.st.springmvc.config;
2.
3.
4.  import java.util.Locale;
5.
6.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
7.  import org.springframework.context.MessageSource;
8.  import org.springframework.context.annotation.Bean;
9.  import org.springframework.context.annotation.ComponentScan;
10. import org.springframework.context.annotation.Configuration;
11. import org.springframework.context.support.ResourceBundleMessageSource;
12. import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
13. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
14. import org.springframework.web.servlet.i18n.CookieLocaleResolver;
15. import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
16. import org.thymeleaf.spring4.SpringTemplateEngine;
17. import org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver;
18.
19. @Configuration
20. @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
```

```
@EnableAutoConfiguration
public class Config extends WebMvcConfigurerAdapter {
    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("i18n/messages");
        return messageSource;
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("lang");
        return localeChangeInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }

    @Bean
    public CookieLocaleResolver localeResolver() {
        CookieLocaleResolver localeResolver = new CookieLocaleResolver();
        localeResolver.setCookieName("lang");
        localeResolver.setDefaultLocale(new Locale("fr"));
        return localeResolver;
    }

    @Bean
    public SpringResourceTemplateResolver templateResolver() {
        SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
        templateResolver.setPrefix("classpath:/templates/");
        templateResolver.setSuffix(".xml");
        templateResolver.setTemplateMode("HTML5");
        templateResolver.setCacheable(true);
        templateResolver.setCharacterEncoding("UTF-8");
        return templateResolver;
    }

    @Bean
    SpringTemplateEngine templateEngine(SpringResourceTemplateResolver templateResolver) {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);
        return templateEngine;
    }

}
```

La classe [Main] est la classe exécutable du projet :

```
package istia.st.springmvc.main;

import istia.st.springmvc.config.Config;

import java.util.Arrays;

import org.springframework.boot.SpringApplication;
import org.springframework.context.ApplicationContext;

public class Main {
    public static void main(String[] args) {
        // on lance l'application
        ApplicationContext context = SpringApplication.run(Config.class, args);
        // on affiche la liste des beans trouvés par Spring
        System.out.println("Liste des beans Spring");
        String[] beanNames = context.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
            System.out.println(beanName);
        }
    }
}
```

- ligne 13, Spring Boot est lancé avec le fichier de configuration [Config] ;
- lignes 15-20 : pour l'exemple, nous montrons comment afficher la liste des objets gérée par Spring. Cela peut être utile si parfois on a l'impression que Spring ne gère pas l'un de nos composants. C'est un moyen de le vérifier. C'est aussi un moyen de vérifier l'autoconfiguration faite par Spring Boot. Sur la console, on obtient une liste analogue à la suivante :

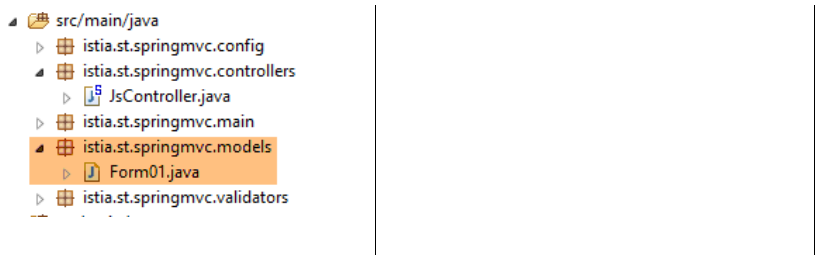
```
1.  Liste des beans Spring
2.  basicErrorController
3.  beanNameHandlerMapping
4.  beanNameViewResolver
5.  config
6.  defaultServletHandlerMapping
7.  defaultTemplateResolver
8.  defaultViewResolver
9.  dispatcherServlet
10. dispatcherServletRegistration
11. embeddedServletContainerCustomizerBeanPostProcessor
12. error
13. errorAttributes
14. faviconHandlerMapping
15. faviconRequestHandler
16. handlerExceptionResolver
17. hiddenHttpMethodFilter
18. http.mappers.CONFIGURATION_PROPERTIES
19. httpRequestHandlerAdapter
20. jacksonObjectMapper
21. jsController
22. layoutDialect
23. localeChangeInterceptor
24. localeResolver
25. mappingJackson2HttpMessageConverter
26. mbeanExporter
27. mbeanServer
28. messageConverters
29. messageSource
30. multipart.CONFIGURATION_PROPERTIES
31. multipartConfigElement
32. multipartResolver
33. mvcContentNegotiationManager
34. mvcConversionService
35. mvcUriComponentsContributor
36. mvcValidator
37. objectNamingStrategy
38. org.springframework.boot.autoconfigure.AutoConfigurationPackages
39. org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration
40. org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
41. org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperAutoConfigurati
    on
42. org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration
43. org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration$Empty
44. org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration
45. org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration$DefaultTemplateResolverConfigu
    ration
46. org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration$ThymeleafViewResolverConfigura
    tion
47. org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration$ThymeleafWebLayoutConfiguratio
    n
48. org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration
49. org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletConfigurati
    on
50. org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration
51. org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat
52. org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration
53. org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration
54. org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration
55. org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration$ObjectMappers
56. org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration
57. org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration
58. org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration
59. org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter
60. org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter$FaviconC
    onfiguration
61. org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor
62. org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor.store
63. org.springframework.context.annotation.ConfigurationClassPostProcessor.enhancedConfigurationProcessor
```

```
org.springframework.context.annotation.ConfigurationClassPostProcessor.importAwareProcessor
org.springframework.context.annotation.MBeanExportConfiguration
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalRequiredAnnotationProcessor
org.springframework.web.servlet.config.annotation.DelegatingWebMvcConfiguration
propertySourcesPlaceholderConfigurer
requestContextListener
requestMappingHandlerAdapter
requestMappingHandlerMapping
resourceHandlerMapping
serverProperties
simpleControllerHandlerAdapter
spring.mvc.CONFIGURATION_PROPERTIES
spring.resources.CONFIGURATION_PROPERTIES
templateEngine
templateResolver
thymeleafResourceResolver
thymeleafViewResolver
tomcatEmbeddedServletContainerFactory
viewControllerHandlerMapping
  viewResolver
```

Nous avons surligné les objets définis dans la classe [Config].

6.2.2 Le modèle du formulaire

Continuons l'exploration du projet :



La classe [Form01] est la classe qui va réceptionner les valeurs postées. Elle est la suivante :

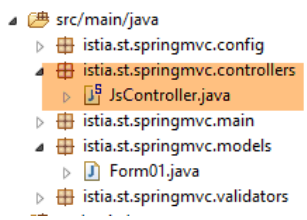
```
package istia.st.springmvc.models;

import java.util.Date;

import javax.validation.constraints.AssertFalse;
import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.DecimalMax;
import javax.validation.constraints.DecimalMin;
import javax.validation.constraints.Future;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.NotBlank;
import org.hibernate.validator.constraints.Range;
import org.hibernate.validator.constraints.URL;
import org.springframework.format.annotation.DateTimeFormat;

public class Form01 {

    // valeurs postées
    @NotNull
    @AssertFalse
```

```
29.     private Boolean assertFalse;
30.
31.     @NotNull
32.     @AssertTrue
33.     private Boolean assertTrue;
34.
35.     @NotNull
36.     @Future
37.     @DateTimeFormat(pattern = "yyyy-MM-dd")
38.     private Date dateInFuture;
39.
40.     @NotNull
41.     @Past
42.     @DateTimeFormat(pattern = "yyyy-MM-dd")
43.     private Date dateInPast;
44.
45.     @NotNull
46.     @Max(value = 100)
47.     private Integer intMax100;
48.
49.     @NotNull
50.     @Min(value = 10)
51.     private Integer intMin10;
52.
53.     @NotNull
54.     @NotBlank
55.     private String strNotEmpty;
56.
57.     @NotNull
58.     @Size(min = 4, max = 6)
59.     private String strBetween4and6;
60.
61.     @NotNull
62.     @Pattern(regexp = "^\\d{2}:\\d{2}:\\d{2}$")
63.     private String hhmmss;
64.
65.     @NotNull
66.     @Email
67.     @NotBlank
68.     private String email;
69.
70.     @NotNull
71.     @Length(max = 4, min = 4)
72.     private String str4;
73.
74.     @Range(min = 10, max = 14)
75.     @NotNull
76.     private Integer int1014;
77.
78.     @NotNull
79.     @DecimalMax(value = "3.4")
80.     @DecimalMin(value = "2.3")
81.     private Double double1;
82.
83.     @NotNull
84.     private Double double2;
85.
86.     @NotNull
87.     private Double double3;
88.
89.     @URL
90.     @NotBlank
91.     private String url;
92.
93.     // validation client
94.     private boolean clientValidation = true;
95.     // locale
96.     private String lang;
97.     ...
98. }
```

Nous retrouvons des validateurs déjà rencontrés. Nous allons de plus introduire la notion de validation spécifique. C'est une validation qui ne peut être formalisée avec un validateur prédéfini. On va ici demander à ce que [double1+double2] soit dans l'intervalle [10,13].

6.2.3 Le contrôleur

Le contrôleur [JsController] est le suivant :




```
1.  package istia.st.springmvc.controllers;
2.
3.  import istia.st.springmvc.models.Form01;
4.  ...
5.
6.  @Controller
7.  public class JsController {
8.
9.      @RequestMapping(value = "/js01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
10.     public String js01(Form01 formulaire, Locale locale, Model model) {
11.         setModel(formulaire, model, locale, null);
12.         return "vue-01";
13.     }
14. ...
15.
16.     // préparation du modèle de la vue vue-01
17.     private void setModel(Form01 formulaire, Model model, Locale locale, String message) {
18. ...
19.     }
20. }
```

- ligne 9, l'action [/js01] ;
- ligne 10 : un objet de type [Form01] est instancié et mis automatiquement dans le modèle, associé à la clé [form01] ;
- ligne 10 : la locale et le modèle sont injectés dans les paramètres ;
- ligne 11 : avec ces informations, on prépare le modèle ;
- ligne 12 : on affiche la vue [vue-01.xml] ;

La méthode [setModel] est la suivante :

```
1.      // préparation du modèle de la vue vue-01
2.      private void setModel(Form01 formulaire, Model model, Locale locale, String message) {
3.          // on ne gère que les locales fr-FR, en-US
4.          String language = locale.getLanguage();
5.          String country = null;
6.          if (language.equals("fr")) {
7.              country = "FR";
8.              formulaire.setLang("fr_FR");
9.          }
10.         if (language.equals("en")) {
11.             country = "US";
12.             formulaire.setLang("en_US");
13.         }
14.         model.addAttribute("locale", String.format("%s-%s", language, country));
15.         // le message éventuel
16.         if (message != null) {
17.             model.addAttribute("message", message);
18.         }
19. }
```

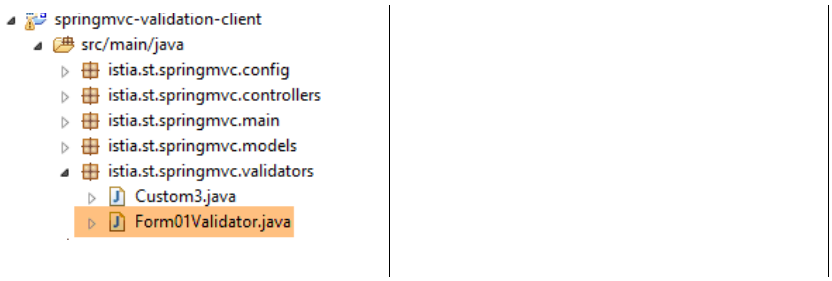
- le but de la méthode [setModel] est de mettre dans le modèle :
  - des informations sur la locale,
  - le message passé en dernier paramètre ;
- ligne 14 : on met dans le modèle des informations sur la locale (langue, pays) ;
- lignes 16-18 : on met dans la locale l'éventuel message passé en paramètre ;

- lignes 8, 12 : les informations sur la locale sont également stockées dans le formulaire [Form01]. Le Javascript va utiliser cette information ;

Les valeurs saisies dans le formulaire [vue-01.xml] vont être postées à l'action [/js02] suivante :

```
1.     @RequestMapping(value = "/js02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String js02(@Valid Form01 formulaire, BindingResult result, RedirectAttributes
   redirectAttributes, Locale locale, Model model) {
3.         Form01Validator validator = new Form01Validator(10, 13);
4.         validator.validate(formulaire, result);
5.         ...
6.  }
```

- ligne 2 : l'annotation [@Valid Form01 formulaire] fait que les valeurs postées vont être soumises aux validateurs de la classe [Form01]. Nous savons qu'il existe une validation spécifique [double1+double2] dans l'intervalle [10,13]. Lorsqu'on arrive à la ligne 3, cette validation n'a pas été faite ;
- ligne 3 : on crée l'objet [Form01Validator] suivant :




```
1.  package istia.st.springmvc.validators;
2.
3.  import istia.st.springmvc.models.Form01;
4.
5.  import org.springframework.validation.Errors;
6.  import org.springframework.validation.Validator;
7.
8.  public class Form01Validator implements Validator {
9.
10.    // l'intervalle de validation
11.    private double min;
12.    private double max;
13.
14.    // constructeur
15.    public Form01Validator(double min, double max) {
16.       this.min = min;
17.       this.max = max;
18.    }
19.
20.    @Override
21.    public boolean supports(Class<?> classe) {
22.       return Form01.class.equals(classe);
23.    }
24.
25.    @Override
26.    public void validate(Object form, Errors errors) {
27.       // objet validé
28.       Form01 form01 = (Form01) form;
29.       // la valeur de [double1]
30.       Double double1 = form01.getDouble1();
31.       if (double1 == null) {
32.          return;
33.       }
34.       // la valeur de [double2]
35.       Double double2 = form01.getDouble2();
36.       if (double2 == null) {
37.          return;
38.       }
39.       // [double1+double2]
40.       double somme = double1 + double2;
41.       // validation
```

```
42.         if (somme < min || somme > max) {
43.             errors.rejectValue("double2", "form01.double2", new Double[] { min, max }, null);
44.         }
45.     }
46.
47. }
```

- ligne 8 : pour implémenter une validation spécifique, nous créons une classe implémentant l'interface Spring [Validator]. Cette interface a deux méthodes : [supports] ligne 21 et [validate] ligne 26 ;
- lignes 21-23 : la méthode [supports] reçoit un objet de type [Class]. Elle doit rendre *true* pour dire qu'elle supporte cette classe, *false* sinon ;
- ligne 22 : nous disons que la classe [Form01Validator] ne valide que des objets de type [Form01] ;
- lignes 15-18 : rappelons que nous voulons implémenter la contrainte [double1+double2] dans l'intervalle [10,13]. plutôt que de s'en tenir à cet intervalle, nous allons vérifier la contrainte [double1+double2] dans l'intervalle [min, max]. C'est pourquoi nous avons un constructeur avec ces deux paramètres ;
- ligne 26 : la méthode [validate] est appelée avec une instance de l'objet validé, donc ici une instance de [Form01] et avec la collection des erreurs actuellement connues [Errors errors]. Si la validation faite par la méthode [validate] échoue, elle doit créer un nouvel élément dans la collection [Errors errors] ;
- ligne 43 : la validation a échoué. On ajoute un élément à la collection [Errors errors] avec la méthode [Errors.rejectValue] dont les paramètres sont les suivants :
  - paramètre 1 : habituellement le nom du champ erroné. Ici on a testé les champs [double1, double2]. On peut mettre l'un des deux,
  - le message d'erreur associé ou plus exactement **sa clé** dans les fichiers de messages externalisés :

    [messages_fr.properties]

    ```
    form01.double2=[double2+double1] doit être dans l''intervalle [{0},{1}]
    ```

    [messages_en.properties]

    ```
    form01.double2=[double2+double1] must be in [{0},{1}
    ```

    On a là des messages paramétrés par {0} et {1}. Il faut donc fournir deux valeurs à ce message. C'est ce que fait le troisième paramètre de la méthode [Errors.rejectValue].
  - le quatrième paramètre est un message par défaut pour l'erreur ;

Revenons à l'action [/js02] :

```
1.     @RequestMapping(value = "/js02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String js02(@Valid Form01 formulaire, BindingResult result, RedirectAttributes
   redirectAttributes, Locale locale, Model model) {
3.         Form01Validator validator = new Form01Validator(10, 13);
4.         validator.validate(formulaire, result);
5.         if (result.hasErrors()) {
6.             StringBuffer buffer = new StringBuffer();
7.             for (ObjectError error : result.getAllErrors()) {
8.                 buffer.append(String.format("[name=%s,code=%s,message=%s]", error.getObjectName(),
   error.getCode(), error.getDefaultMessage()));
9.             }
10.            setModel(formulaire, model, locale, buffer.toString());
11.            return "vue-01";
12.        } else {
13.            redirectAttributes.addFlashAttribute("form01", formulaire);
14.            return "redirect:/js01.html";
15.        }
16. }
```

- ligne 4 : le validateur [Form01Validator] est exécuté avec les paramètres :
  - paramètre 1 : l'objet en cours de validation,
  - paramètre 2 : la liste des erreurs de cet objet. Celle-ci est l'objet [BindingResult result] passé en paramètres de l'action. Si la validation échoue, cet objet aura une erreur de plus ;
- ligne 5 : on teste s'il y a des erreurs de validation ;
- lignes 7-10 : on parcourt la liste des erreurs pour mémoriser pour chacune d'elles :
  - le nom de l'objet validé,
  - son code d'erreur,
  - son message d'erreur par défaut ;

- ligne 10 : avec ces informations, on construit le modèle de la vue [vue-01.xml]. Cette fois-ci, il y a un message, la version concaténée et abrégée des différents messages d'erreur ;
- lignes 12-15 : si toutes les valeurs postées sont valides, on redirige le client vers l'action [/js01] en mettant les valeurs postées en attribut Flash ;

6.2.4 La vue

La vue [vue-01.xml] est complexe. Nous n'allons en présenter qu'une petite partie :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <title>Spring 4 MVC</title>
5.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6.        <link rel="stylesheet" href="/css/form01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-1.10.2.min.js"></script>
8.        ...
9.     </head>
10.    <body>
11.       <!-- titre -->
12.       <h3>
13.          <span th:text="#{form01.title}"></span>
14.          <span th:text="${locale}"></span>
15.       </h3>
16.       <!-- menu -->
17.       <p>
18. ...
19.       </p>
20.       <!-- formulaire -->
21.       <form action="/someURL" th:action="@{/js02.html}" method="post" th:object="${form01}" name="form"
    id="form">
22.          <table>
23.             <thead>
24.                <tr>
25.                   <th class="col1" th:text="#{form01.col1}">Contrainte</th>
26.                   <th class="col2" th:text="#{form01.col2}">Saisie</th>
27.                   <th class="col3" th:text="#{form01.col3}">Validation client</th>
28.                   <th class="col4" th:text="#{form01.col4}">Validation serveur</th>
29.                </tr>
30.             </thead>
31.             <tbody>
32.                <!-- required -->
33.                <tr>
34.                   <td class="col1">required</td>
35.                   <td class="col2">
36.                      <input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-
    required=#{NotNull}" />
37.                   </td>
38.                   <td class="col3">
39.                      <span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-
    replace="true"></span>
40.                   </td>
41.                   <td class="col4">
42.                      <span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}"
    class="error">Donnée erronée</span>
43.                   </td>
44.                </tr>
45. ...
46.             </tbody>
47.          </table>
48.          <p>
49.          <!-- bouton de validation -->
50.          <input type="submit" th:value="#{form01.valider}" value="Valider"
    onclick="javascript:postForm01()" />
51.          </p>
52.       </form>
53.       <!-- message des validateurs côté serveur -->
54.       <br/>
55.       <fieldset class="fieldset">
56.          <legend>
57.             <span th:text="#{server.error.message}"></span>
58.          </legend>
59.          <span th:text="${message}" class="error"></span>
```

```
60.       </fieldset>
61.    </body>
62. </html>
```

Cette page utilise un certain nombre de messages trouvés dans les fichiers de messages externalisés :

[messages_fr.properties]

```
1. form01.title=Formulaire - Validations côté client - locale=
2. form01.col1=Contrainte
3. form01.col2=Saisie
4. form01.col3=Validation client
5. form01.col4=Validation serveur
6. form01.valider=Valider
7. server.error.message=Erreurs détectées par les validateurs côté serveur
```

[messages_en.properties]

```
1. form01.title=Form - Client side validation - locale=
2. form01.col1=Constraint
3. form01.col2=Input
4. form01.col3=Client validation
5. form01.col4=Server validation
6. form01.valider=Validate
7. server.error.message=Errors detected by the validators on the server side
```

Revenons au code de la page :

- ligne 8 : un grand nombre d'imports de bibliothèques Javascript que nous pouvons ignorer ici ;
- ligne 14 : affiche la locale mise dans le modèle par le serveur ;
- ligne 59 : affiche le message mis dans le modèle par le serveur ;

Le code des lignes 33-44 est nouveau. Etudions-le :

```
63. <!-- required -->
64. <tr>
65.   <td class="col1">required</td>
66.   <td class="col2">
67.     <input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-required=#{NotNull}" />
68.   </td>
69.   <td class="col3">
70.     <span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-replace="true"></span>
71.   </td>
72.   <td class="col4">
73.     <span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}" class="error">Donnée
    erronée</span>
74.   </td>
75. </tr>
```

Le plus simple est peut-être de regarder le code HTML généré par ce segment Thymeleaf :

```
1. <!-- required -->
2. <tr>
3.   <td class="col1">required</td>
4.   <td class="col2">
5.     <input type="text" data-val="true" data-val-required="Le champ est obligatoire"
   id="strNotEmpty" name="strNotEmpty" value="" />
6.   </td>
7.   <td class="col3">
8.     <span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-
   replace="true"></span>
9.   </td>
10.  <td class="col4">
11.
12.  </td>
13. </tr>
```

Nous allons utiliser, côté client une bibliothèque de validation appelée [jquery.validate]. Tous les attributs [data-x] sont pour elle. Lorsque la validation côté client sera inhibée, ces attributs ne seront pas exploités. Donc pour l'instant, il est inutile de les comprendre. On peut simplement s'attarder sur la ligne Thymeleaf suivante :

```
<input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-required=#{NotNull}" />
```

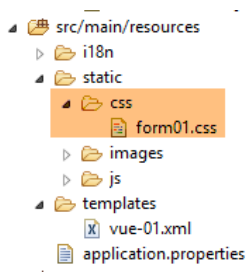
qui génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-required="Le champ est obligatoire" id="strNotEmpty"
name="strNotEmpty" value="" />
```

Ci-dessus, il y a une difficulté pour générer l'attribut [data-val-required="Le champ est obligatoire"]. En effet, la valeur associée à l'attribut provient des fichiers de messages externalisés. On est alors obligé de passer par une expression Thymeleaf pour l'obtenir. C'est l'expression suivante : [th:attr="data-val-required=#{NotNull}"]. Cette expression est évaluée et sa valeur mise telle quelle dans la balise HTML générée. Elle s'appelle [th:attr] car on l'utilise pour générer des attributs nons prédéfinis dans Thymeleaf Nous avons rencontré des attributs prédéfinis [th:text, th:value, th:class, ...] mais il n'existe pas d'attribut [th:data-val-required].

6.2.5 La feuille de style

Ci-dessus, on rencontre des classes CSS telles que [class="field-validation-valid"]. Certaine de ces classes sont utilisées par la bibliothèque Javascript de validation. Elles sont définies dans le fichier [form01.css] suivant :




```
1.  @CHARSET "UTF-8";
2.
3.  /*styles perso*/
4.  body {
5.      background-image: url("/images/standard.jpg");
6.  }
7.
8.  .col1 {
9.      background: lightblue;
10. }
11.
12. .col2 {
13.     background: Cornsilk;
14. }
15.
16. .col3 {
17.     background: AliceBlue;
18. }
19.
20. .col4 {
21.     background: Lavender;
22. }
23.
24. .error {
25.     color: red;
26. }
27.
28. .fieldset{
29.     background: Lavender;
30. }
31. /* Styles for validation helpers
32. -----------------------------------------------------------*/
```

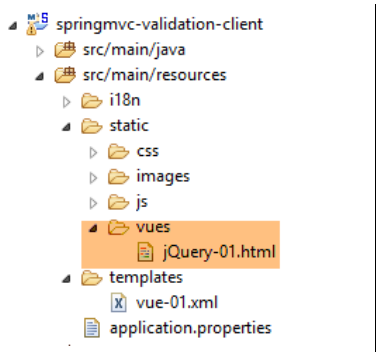
```
33. .field-validation-error {
34.     color: #f00;
35. }
36.
37. .field-validation-valid {
38.     display: none;
39. }
40.
41. .input-validation-error {
42.     border: 1px solid #f00;
43.     background-color: #fee;
44. }
45.
46. .validation-summary-errors {
47.     font-weight: bold;
48.     color: #f00;
49. }
50.
51. .validation-summary-valid {
52.     display: none;
53.     }
```

6.3 Validation côté client

6.3.1 Rudiments de jQuery et de Javascript

La validation côté client se fait avec du Javascript. Nous allons nous aider du framework jQuery qui apporte de nombreuses fonctions facilitant le développement Javascript. Nous présentons les rudiments de jQuery à connaître pour comprendre les scripts de ce chapitre et des suivants.

Nous créons un fichier statique HTML [JQuery-01.html] que l'on place dans un dossier [static / vues] :

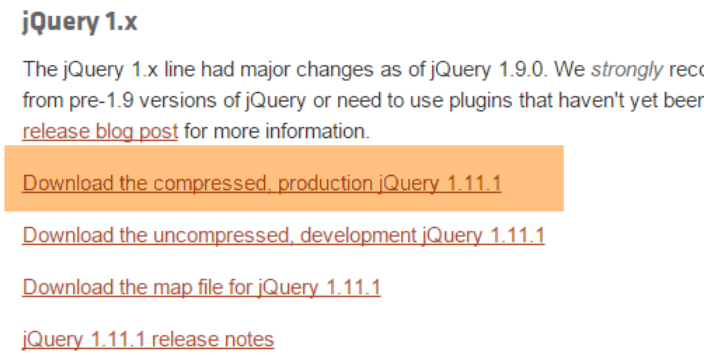


Ce fichier aura le contenu suivant :

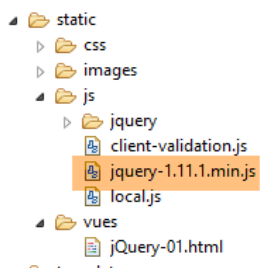
```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>JQuery-01</title>
6.    <script type="text/javascript" src="/js/jquery-1.11.1.min.js"></script>
7.  </head>
8.  <body>
9.    <h3>Rudiments de JQuery</h3>
10.   <div id="element1">
11.     Elément 1
12.   </div>
13. </body>
14. </html>
```

- ligne 6 : importation de jQuery ;
- lignes 10-12 : un élément de la page d'**id** [element1]. Nous allons jouer avec cet élément.

Il nous faut télécharger le fichier [*jquery-1.11.1.min.js*]. On le trouvera la dernière version de jQuery à l'URL [http://jquery.com/download/] :

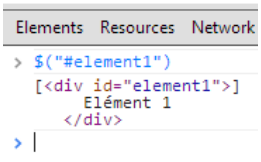
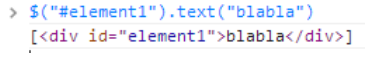


On placera le fichier téléchargé dans le dossier [static / js] :



Ceci fait, on demande la vue statique [jQuery-01.html] avec Chrome [1-2] :

Avec Google Chrome, faire [Ctrl-Maj-I] pour faire apparaître les outils de développement [3]. L'onglet [Console] [4] permet d'exécuter du code Javascript. Nous donnons dans ce qui suit des commandes Javascript à taper et nous en donnons une explication.

JS	résultat
`$("#element1")` : rend la collection de tous les éléments d'**id** [element1], donc normalement une collection de 0 ou 1 élément parce qu'on ne peut avoir deux **id** identiques dans une page HTML.	
`$("#element1").text("blabla")` : affecte le texte [blabla] à tous les éléments de la collection. Ceci a pour effet de changer le	

contenu affiché par la page

`$("#element1").hide()`
cache les éléments de la collection. Le texte [blabla] n'est plus affiché.

```
> $("#element1").hide()
  [<div id="element1">blabla</div>]
```

`$("#element1")`
: affiche de nouveau la collection. Cela nous permet de voir que l'élément d'id [element1] a l'attribut CSS **style='display : none;'** qui fait que l'élément est caché.

```
> $("#element1")
  [<div id="element1" style="display: none;">blabla</div>]
```

`$("#element1").show()`
: affiche les éléments de la collection. Le texte [blabla] apparaît de nouveau. C'est l'attribut CSS **style='display : block;'** qui assure cet affichage.

```
> $("#element1").show()
  [<div id="element1" style="display: block;">blabla</div>]
```

`$("#element1").attr('style','color: red')`
: fixe un attribut à tous les éléments de la collection. L'attribut est ici [style] et sa valeur [color: red]. Le texte [blabla] passe en rouge.

```
> $("#element1").attr('style','color: red')
  [<div id="element1" style="color: red">blabla</div>]
```

Tableau

```
> var data=["zéro",1,"deux"]
undefined
> data[0]
"zéro"
> data[2]
"deux"
> data.length
3
> data[3]="trois"
"trois"
> data
["zéro", 1, "deux", "trois"]
> data[1]="un"
"un"
> data
["zéro", "un", "deux", "trois"]
```

Dictionnaire

```
> data={"zéro":0,"un":1,"erreur":"msg"}
Object {zéro: 0, un: 1, erreur: "msg"}
> data["zéro"]
0
> data.zéro
0
> data.msg="msg2"
"msg2"
> data
Object {zéro: 0, un: 1, erreur: "msg", msg: "msg2"}
> data.erreur="autre msg"
"autre msg"
> data
Object {zéro: 0, un: 1, erreur: "autre msg", msg: "msg2"}
```

On notera que l'URL du navigateur n'a pas changé pendant toutes ces manipulations. Il n'y a pas eu d'échanges avec le serveur web. Tout se passe à l'intérieur du navigateur. Maintenant, visualisons le code source de la page :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml">
3.  <head>
4.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
5.    <title>JQuery-01</title>
6.    <script type="text/javascript" src="/js/jquery-1.11.1.min.js"></script>
7.  </head>
8.  <body>
9.    <h3>Rudiments de JQuery</h3>
10.   <div id="element1">
11.     Elément 1
12.   </div>
13. </body>
14. </html>
```

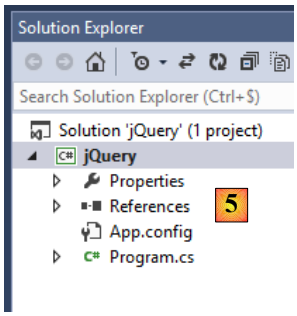
C'est le texte initial. Il ne reflète en rien les manipulations que l'on a faites sur l'élément des lignes 10-12. Il est important de s'en souvenir lorsqu'on fait du débogage Javascript. Il est alors souvent inutile de visualiser le code source de la page affichée.

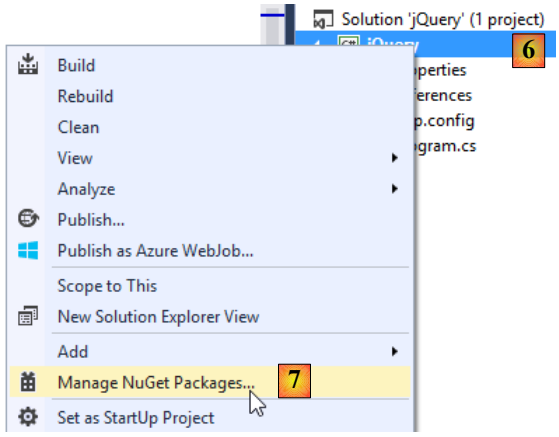
Nous en savons assez pour comprendre les scripts jS qui vont suivre.

6.3.2 Les bibliothèques jS de validation

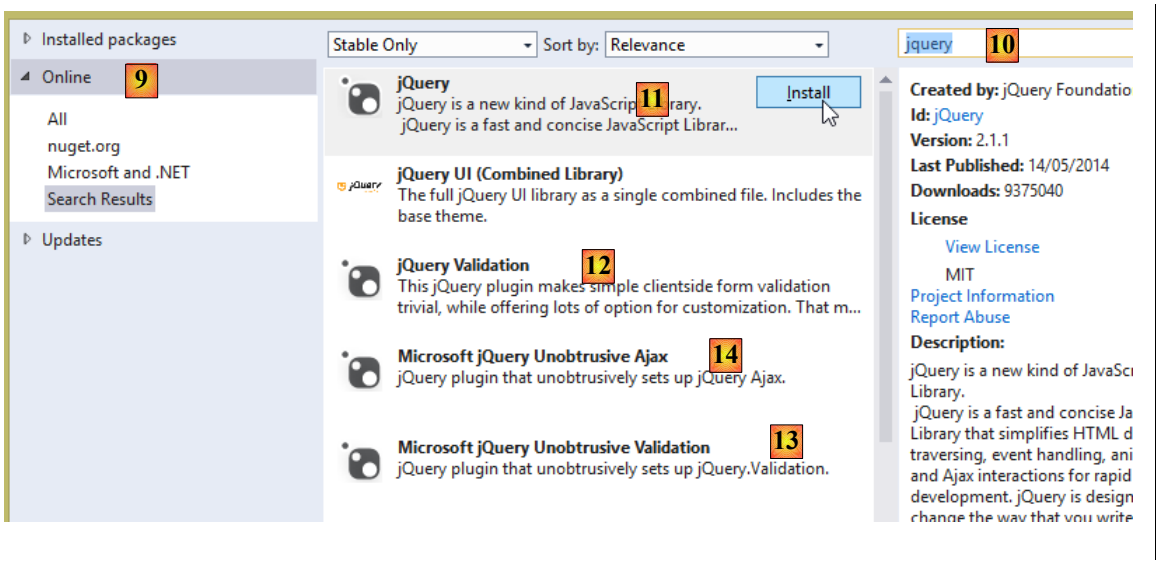
Nous allons utiliser des bibliothèques de l'écosystème jQuery. Gravite autour de jQuery, un certain nombre de projets qui donnent naissance à leur tour à des bibliothèques. Nous allons utiliser la bibliothèque de validation [jquery.validate.unobstrusive] créée par Microsoft et donnée à la fondation jQuery. Nous la désignerons par la suite par bibliothèque MS de validation ou plus simplement bibliothèque MS. Pour l'obtenir, il faut un environnement Microsoft Visual Studio. Je n'ai pas vu comment l'obtenir autrement. On peut utiliser une version gratuite de type [Visual Studio Community] [http://www.visualstudio.com/en-us/news/vs2013-community-vs.aspx] (déc 2014). Le lecteur pas intéressé à suivre la démarche qui suit peut récupérer cette bibliothèque et celles sur lesquelles elle s'appuie dans les exemples présents sur le site de ce document.

On crée un projet console avec Visual Studio [1-4] :

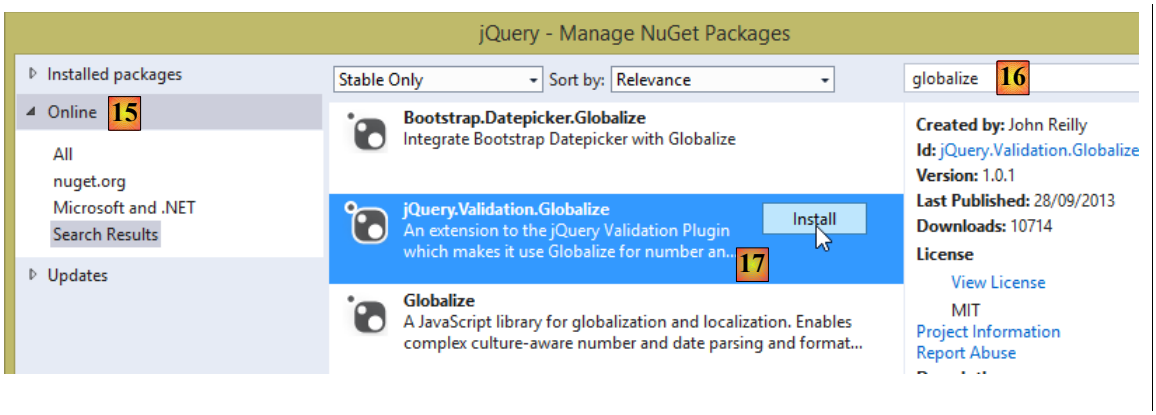




- en [5], le projet console ;
- en [6-7] : on va ajouter des packages [NuGet] au projet. [NuGet] est une fonction de Visual Studio permettant de télécharger des bibliothèques sous formes de DLL mais également des bibliothèques jS.

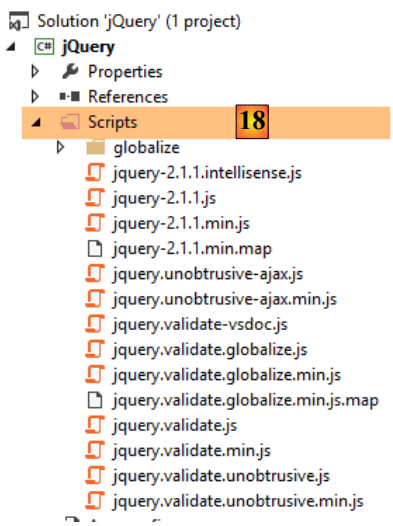


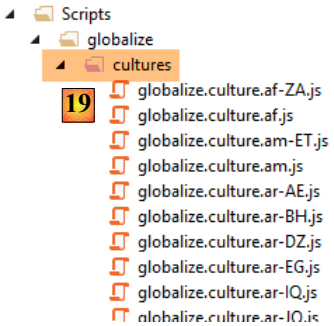
- en [9-10], faites une recherche avec le mot clé [jQuery] ;
- en [11-13], téléchargez dans l'ordre indiqué les bibliothèques jS nécessaires à la validation côté client ;
- en [14], téléchargez également la bibliothèque [Microsoft jQuery Unobtrusive Ajax] que nous allons utiliser prochainement ;



- en [15-16], faites une recherche de packages avec le mot clé [globalize] ;

- en [17], téléchargez la bibliothèque [jQuery.Validation.Globalize] ;



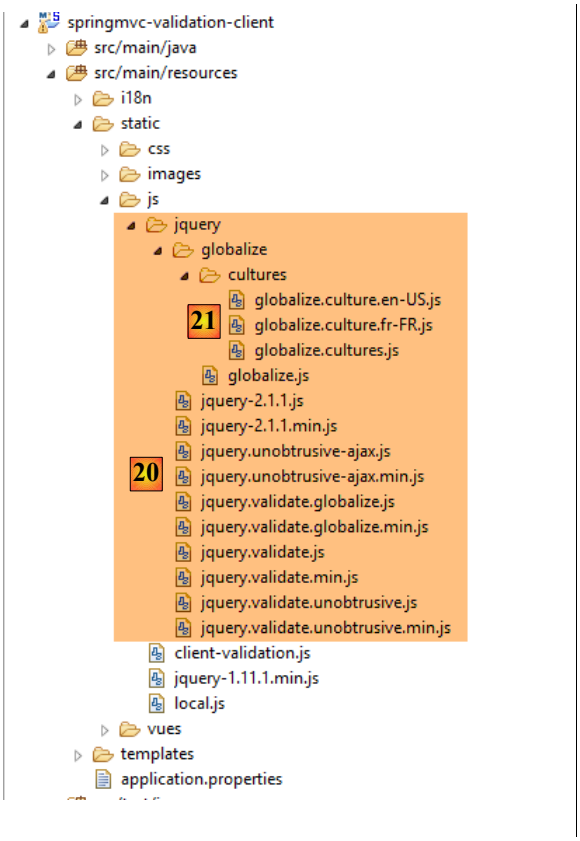


Ces divers téléchargements ont installé un certain nombre de bibliothèques jS dans le dossier [Scripts] du projet [18]. Ils ne sont pas tous utiles. Chaque fichier vient en deux exemplaires :

- [js] : la version lisible de la bibliothèque ;
- [min.js] : la version illisible dite minifiée 'minified' de la bibliothèque. Elle n'est pas vraiment illisible. C'est du texte. Mais elle n'est pas compréhensible. C'est la version à utiliser en production car ce fichier est plus petit que la version correspondante [js] et donc améliore la rapidité des échanges client / serveur ;

Les version [min.map] ne sont pas indispensables. Dans le dossier [cultures], on peut ne conserver que les cultures gérées par l'application.

Avec l'explorateur Windows, on copie ces fichiers dans le dossier [static / js / jquery] du projet [springmvc-validation-client] et on ne garde que les fichiers utiles [20] :



En [21], on ne garde que deux cultures :

- [fr-FR] : le français de France ;
- [en-US] : l'anglais des USA ;

6.3.3 Import des bibliothèques jS de validation

Pour être exploitées, ces bibliothèques doivent être importées par la vue [vue-01.xml] :

```
1.  <head>
2.        <title>Spring 4 MVC</title>
3.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
4.        <link rel="stylesheet" href="/css/form01.css" />
5.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
6.        <script type="text/javascript" src="/js/jquery/jquery.validate.min.js"></script>
7.        <script type="text/javascript" src="/js/jquery/jquery.validate.unobtrusive.min.js"></script>
8.        <script type="text/javascript" src="/js/jquery/globalize/globalize.js"></script>
9.        <script type="text/javascript" src="/js/jquery/globalize/cultures/globalize.culture.fr-
    FR.js"></script>
10.       <script type="text/javascript" src="/js/jquery/globalize/cultures/globalize.culture.en-
    US.js"></script>
11.       <script type="text/javascript" src="/js/client-validation.js"></script>
12.       <script type="text/javascript" src="/js/local.js"></script>
13.       <script th:inline="javascript">
14.          /*<![CDATA[*/
15.                var culture = [[${locale}]];
16.                Globalize.culture(culture);
17.                /*]]>*/
18.       </script>
19.    </head>
```

- ligne 11 : l'import d'un fichier jS dont nous n'avons pas encore parlé ;
- lignes 13-18 : un script jS interprété par Thymelaf. Il gère la locale côté client ;

6.3.4 Gestion de la locale côté client

La localisation côté client est faite par le script jS suivant :

```
1.  <script th:inline="javascript">
2.          /*<![CDATA[*/
3.                var culture = [[${locale}]];
4.                Globalize.culture(culture);
5.                /*]]>*/
6.  </script>
```

- lignes 3-4 : du code jS dans lequel on trouve l'expression Thymeleaf [[${locale}]]. Notez la syntaxe particulière de cette expression. Ceci parce qu'elle est dans du javascript. L'expression [[${locale}]] va être remplacée par la valeur de la clé [locale] du modèle de la vue ;

Le résultat dans le flux HTML généré de ces lignes est le suivant :

```
1.  <script>
2.          /*<![CDATA[*/
3.                var culture = 'en-US';
4.                Globalize.culture(culture);
5.                /*]]>*/
6.  </script>
```

Les lignes 3-4 fixent la culture côté client. On n'en gère que deux, [fr-FR] et [en-US]. C'est la raison pour laquelle nous n'avons importé que deux fichiers de culture :

```
1.      <script type="text/javascript"
    src="/js/jquery/globalize/cultures/globalize.culture.fr-FR.js"></script>
2.  <script type="text/javascript" src="/js/jquery/globalize/cultures/globalize.culture.en-
    US.js"></script>
```

La culture à utiliser côté client est fixée côté serveur. Revenons sur le code côté serveur :

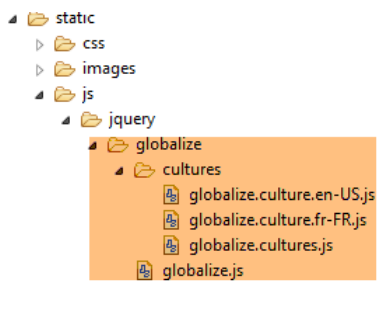
```
1.      @RequestMapping(value = "/js01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.      public String js01(Form01 formulaire, Locale locale, Model model) {
3.         setModel(formulaire, model, locale, null);
4.         return "vue-01";
5.      }
6.
7.      // préparation du modèle de la vue vue-01
8.      private void setModel(Form01 formulaire, Model model, Locale locale, String message) {
9.         // on ne gère que les locales fr-FR, en-US
10.        String language = locale.getLanguage();
11.        String country = null;
12.        if (language.equals("fr")) {
13.           country = "FR";
14.           formulaire.setLang("fr_FR");
15.        }
16.        if (language.equals("en")) {
17.           country = "US";
18.           formulaire.setLang("en_US");
19.        }
20.        model.addAttribute("locale", String.format("%s-%s", language, country));
21. ...
22. }
```

- ligne 20 : la locale [fr-FR] ou [en-US] est mise dans le modèle de la vue [vue-01.xml] (ligne 4). On notera une source de complications. Alors qu'une locale française est notée [fr-FR] côté client, elle est notée [fr_FR] côté serveur. C'est la raison pour laquelle, lignes 14 et 18, elle est stockée sous cette forme dans l'objet [Form01 formulaire] qui réceptionne les valeurs postées ;

On notera le point important suivant. Le script

```
1.  <script>
2.          /*<![CDATA[*/
3.                var culture = 'en-US';
4.                Globalize.culture(culture);
5.                /*]]>*/
6.  </script>
```

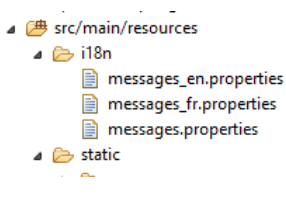
change la culture du client à partir de la locale transmise par le serveur. Cela n'internationalise pas les messages affichés par la page. Cela change seulement la façon d'interpréter certaines informations qui dépendent de la culture d'un pays. Avec la culture [fr_FR], le nombre réel [12,78] est valide alors qu'il est invalide avec le culture [en-US]. Il faut alors écrire [12.78]. De même la date [12/01/2014] est une date valide dans la culture [fr-FR] alors que dans la culture [en-US] il faut écrire [01/12/2014]. Les fichiers du dossier [jquery / globalize] gèrent ce genre de problèmes :



L'internationalisation des messages d'erreur est gérée uniquement côté serveur. Nous allons voir que la page HTML / jS transporte avec elle des messages d'erreur correspondant à la locale gérée par le serveur : en français pour la locale [fr_FR] et en anglais pour la locale [en_US].

6.3.5 Les fichiers de messages

La vue [vue-01.xml] utilise les messages internationalisés suivants :



[messages_fr.properties]

```
1. NotNull=Le champ est obligatoire
2. NotEmpty=La donnée ne peut être vide
3. NotBlank=La donnée ne peut être vide
4. typeMismatch=Format invalide
5. Future.form01.dateInFuture=La date doit être postérieure ou égale à celle d''aujourd'hui
6. Past.form01.dateInPast=La date doit être antérieure ou égale à celle d''aujourd'hui
7. Min.form01.intMin10=La valeur doit être supérieure ou égale à 10
8. Max.form01.intMax100=La valeur doit être inférieure ou égale à 100
9. Size.form01.strBetween4and6=La chaîne doit avoir entre 4 et 6 caractères
10. Length.form01.str4=La chaîne doit avoir quatre caractères exactement
11. Email.form01.email=Adresse mail invalide
12. URL.form01.url=URL invalide
13. Range.form01.int1014=La valeur doit être dans l''intervalle [10,14]
14. AssertTrue=Seule la valeur True est acceptée
15. AssertFalse=Seule la valeur False est acceptée
16. Pattern.form01.hhmmss=Tapez l''heure sous la forme hh:mm:ss
17. form01.hhmmss.pattern=^\\d{2}:\\d{2}:\\d{2}$
18. DateInvalide.form01=Date invalide
19. form01.str4.pattern=^.{4,4}$
20. form01.int1014.max=14
21. form01.int1014.min=10
22. form01.strBetween4and6.pattern=^.{4,6}$
23. form01.intMax100.value=100
24. form01.intMin10.value=10
25. form01.double1.min=2.3
26. form01.double1.max=3.4
```

```
Range.form01.double1=La valeur doit être dans l'intervalle [2,3-3,4]
form01.title=Formulaire - Validations côté client - locale=
form01.col1=Contrainte
form01.col2=Saisie
form01.col3=Validation client
form01.col4=Validation serveur
form01.valider=Valider
form01.double2=[double2+double1] doit être dans l''intervalle [{0},{1}]
form01.double3=[double3+double1] doit être dans l''intervalle [{0},{1}]
locale.fr=Français
locale.en=English
client.validation.true=Activer la validation client
client.validation.false=Inhiber la validation client
DecimalMin.form01.double1=Le nombre doit être supérieur ou égal à 2,3
DecimalMax.form01.double1=Le nombre doit être inférieur ou égal à 3,4
server.error.message=Erreurs détectées par les validateurs côté serveur
```

[messages_en.properties]

```
NotNull=Field is required
NotEmpty=Field can''t be empty
NotBlank=Field can''t be empty
typeMismatch=Invalid format
Future.form01.dateInFuture=Date must be greater or equal to today''s date
Past.form01.dateInPast=Date must be lower or equal today''s date
Min.form01.intMin10=Value must be higher or equal to 10
Max.form01.intMax100=Value must be lower or equal to 100
Size.form01.strBetween4and6=String must have between 4 and 6 characters
Length.form01.str4=String must be exactly 4 characters long
Email.form01.email=Invalid mail address
URL.form01.url=Invalid URL
Range.form01.int1014=Value must be in [10,14]
AssertTrue=Only value True is allowed
AssertFalse=Only value False is allowed
Pattern.form01.hhmmss=Time must follow the format hh:mm:ss
form01.hhmmss.pattern=^\\d{2}:\\d{2}:\\d{2}$
DateInvalide.form01=Invalid Date
form01.str4.pattern=^.{4,4}$
form01.int1014.max=14
form01.int1014.min=10
form01.strBetween4and6.pattern=^.{4,6}$
form01.intMax100.value=100
form01.intMin10.value=10
form01.double1.min=2.3
form01.double1.max=3.4
Range.form01.double1=Value must be in [2.3,3.4]
form01.title=Form - Client side validation - locale=
form01.col1=Constraint
form01.col2=Input
form01.col3=Client validation
form01.col4=Server validation
form01.valider=Validate
form01.double2=[double2+double1] must be in [{0},{1}]
form01.double3=[double3+double1] must be in [{0},{1}]
locale.fr=Français
locale.en=English
client.validation.true=Activate client validation
client.validation.false=Inhibate client validation
DecimalMin.form01.double1=Value must be greater or equal to 2.3
DecimalMax.form01.double1=Value must be lower or equal to 3.4
server.error.message=Errors detected by the validators on the server side
```

Le fichier [messages.properties] est une copie du fichier des messages anglais. Au final, toute locale différente de [fr] utilisera des messages anglais. On rappelle que le fichier [messages_fr.properties] est utilisé pour toute locale [fr_XX] telle que [fr_CA] ou [fr_FR].

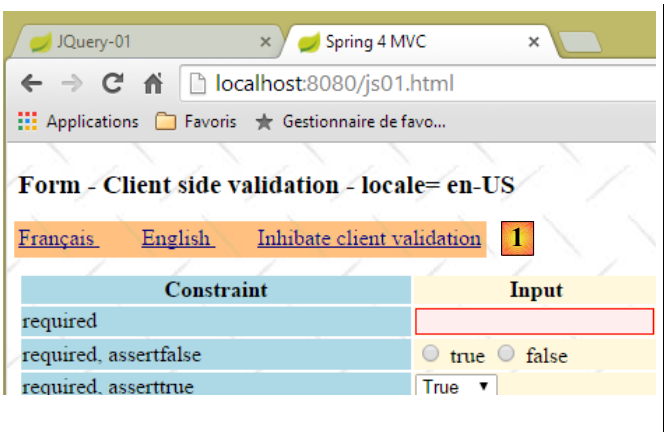
La vue [vue-01.xml] utilise les clés de ces messages. S'il souhaite connaître la valeur associée à ces clés, le lecteur est invité à revenir à ce paragraphe pour la découvrir.

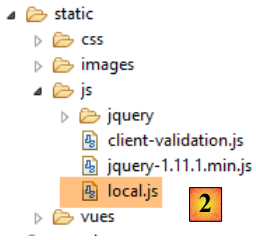
6.3.6 Changement de locale

La vue [vue-01.xml] présente quatre liens :

```
1.  <body>
2.        <!-- titre -->
3.        <h3>
4.           <span th:text="#{form01.title}"></span>
5.           <span th:text="${locale}"></span>
6.        </h3>
7.        <!-- menu -->
8.        <p>
9.           <a id="locale_fr" href="javascript:setLocale('fr_FR')">
10.             <span th:text="#{locale.fr}"></span>
11.          </a>
12.          <a id="locale_en" href="javascript:setLocale('en_US')">
13.             <span style="margin-left:30px" th:text="#{locale.en}"></span>
14.          </a>
15.          <a id="clientValidationTrue" href="javascript:setClientValidation(true)">
16.             <span style="margin-left:30px" th:text="#{client.validation.true}"></span>
17.          </a>
18.          <a id="clientValidationFalse" href="javascript:setClientValidation(false)">
19.             <span style="margin-left:30px" th:text="#{client.validation.false}"></span>
20.          </a>
21.       </p>
22.       <!-- formulaire -->
23.       <form action="/someURL" th:action="@{/js02.html}" method="post" th:object="${form01}" name="form"
    id="form">
24.          ...
```

dont certains sont représentés ci-dessous [1] :





Examinons les deux liens qui permettent de changer la locale en français ou en anglais :

```
1.          <a id="locale_fr" href="javascript:setLocale('fr_FR')">
2.             <span th:text="#{locale.fr}"></span>
3.          </a>
4.          <a id="locale_en" href="javascript:setLocale('en_US')">
5.             <span style="margin-left:30px" th:text="#{locale.en}"></span>
6.  </a>
```

Un clic sur ces liens provoque l'exécution d'un script jS présent dans le fichier [local.js] [2]. Dans les deux cas, c'est une fonction jS [setLocale] qui est appelée :

```
1.  // locale
```

```
2.  function setLocale(locale) {
3.     // on met à jour la locale
4.     lang.val(locale);
5.     // on soumet le formulaire - cela ne déclenche pas les validateurs du client - c'est pourquoi on n'a
    pas inhibé la validation côté client
6.     document.form.submit();
7.  }
```

La compréhension de la ligne 4 nécessite un préambule. La vue [vue-01.xml] embarque un champ caché nommé [lang] :

```
<input type="hidden" th:field="*{lang}" th:value="*{lang}" value="true" />
```

qui correspond à un champ [lang] dans [Form01] :

```
   // locale
   private String lang;
```

Les champs cachés sont pratiques lorsqu'on veut enrichir les valeurs postées. Le javascript permet de leur donner une valeur et cette valeur est postée comme une saisie normale faite par l'utilisateur. Le code HTML généré par Thymeleaf est le suivant :

```
<input type="hidden" value="en_US" id="lang" name="lang" />
```

La valeur du paramètre [value] est celle du champ [Form01.lang] au moment de la génération du HTML. Ce qu'il est important de noter c'est l'identifiant jS du noeud [id="*lang*"]. Cet identifiant est exploité par la fonction [] suivante :

```
1.  // variables globales
2.  var lang;
3.
4.  // document ready
5.  $(document).ready(function() {
6.     // références globales
7.     lang = $("#lang");
8.  });
9.
10. // locale
11. function setLocale(locale) {
12.    // on met à jour la locale
13.    lang.val(locale);
14.    // on soumet le formulaire - pour une raison ignorée cela ne déclenche pas les validateurs du client
15.    // c'est pourquoi on n'a pas inhibé la validation
16.    document.form.submit();
17. }
```

- lignes 5-8 : la fonction jS [$(document).ready(f)] est une fonction qui est exécutée lorsque le navigateur a chargé la totalité du document envoyé par le serveur. Son paramètre est une fonction. On utilise la fonction jS [$(document).ready(f)] pour initialiser l'environnement jS du document chargé ;
- ligne 7 : l'expression [$("#lang")] est une expression jQuery. Sa valeur est une référence sur le noeud du DOM d'attribut [id='lang'] ;
- ligne 2 : les variables déclarées en-dehors d'une fonction sont globales aux fonctions. Ici, cela signifie que la variable [lang] initialisée dans [$(document).ready()] est également connue dans la fonction [setLocale] de la ligne 11 ;
- ligne 13 : modifie l'attribut [value] du noeud identifié par [lang]. Si *lang* vaut [xx_XX] alors la balise HTML du noeud devient :

```
<input type="hidden" value="xx_XX" id="lang" name="lang" />
```

  Le javascript permet de modifier la valeur des éléments du DOM (Document Object Model).
- ligne 16 : [document] désigne le DOM. [document.form] désigne le 1er formulaire trouvé dans ce document. Un Document HTML peut avoir plusieurs balises <form> et donc plusieurs formulaires. Ici nous n'en avons qu'un. [document.form.submit] poste ce formulaire comme si l'utilisateur avait cliqué sur un bouton ayant l'attribut [type='submit']. A quelle action les valeurs du formulaire sont-elles postées ? Pour le savoir, il faut regarder la balise [form] du formulaire dans [vue-01.xml] :

```
<!-- formulaire -->
<form action="/someURL" th:action="@{/js02.html}" method="post" th:object="${form01}" name="form" id="form">
```

  L'action qui va recevoir les valeurs postées est celle désignée par l'attribut [th:action]. Ce sera donc l'action [/js02.html]. On rappelle que dans ce nom, le suffixe [.html] va être enlevé et c'est au final l'action [/js02] qui va être exécutée. Ce qu'il

est important de comprendre c'est que la nouvelle valeur [xx_XX] du noeud [lang] va être postée sous la forme [lang=xx_XX]. Or on a configuré notre application pour intercepter le paramètre [lang] et l'interpréter comme un changement de locale. Donc côté serveur, la locale va devenir [xx_XX]. Regardons l'action [/js02] qui va être exécutée :

```
1.     @RequestMapping(value = "/js02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String js02(@Valid Form01 formulaire, BindingResult result, RedirectAttributes
   redirectAttributes, Locale locale, Model model) {
3.         Form01Validator validator = new Form01Validator(10, 13);
4.         validator.validate(formulaire, result);
5.         if (result.hasErrors()) {
6.            StringBuffer buffer = new StringBuffer();
7.            for (ObjectError error : result.getAllErrors()) {
8.               buffer.append(String.format("[name=%s,code=%s,message=%s]", error.getObjectName(),
   error.getCode(),
9.                     error.getDefaultMessage()));
10.           }
11.           setModel(formulaire, model, locale, buffer.toString());
12.           return "vue-01";
13.        } else {
14.           redirectAttributes.addFlashAttribute("form01", formulaire);
15.           return "redirect:/js01.html";
16.        }
17.    }
18.
19.    // préparation du modèle de la vue vue-01
20.    private void setModel(Form01 formulaire, Model model, Locale locale, String message) {
21.       // on ne gère que les locales fr-FR, en-US
22.       String language = locale.getLanguage();
23.       String country = null;
24.       if (language.equals("fr")) {
25.          country = "FR";
26.          formulaire.setLang("fr_FR");
27.       }
28.       if (language.equals("en")) {
29.          country = "US";
30.          formulaire.setLang("en_US");
31.       }
32.       model.addAttribute("locale", String.format("%s-%s", language, country));
33.       ...
34. }
```

- ligne 2 : l'action [/js02] va recevoir la nouvelle locale [xx_XX] encapsulée dans le paramètre [Locale locale] :
- lignes 5-12 : si certaines des valeurs postées sont invalides, la vue [vue-01.xml] va être affichée avec des messages d'erreur utilisant la nouvelle locale [xx_XX]. Par ailleurs, la ligne 11 fait que la variable [locale=xx-XX] est mise dans le modèle. Côté client, cette valeur va être utilisée pour mettre à jour la locale côté client. Nous en avons décrit le processus ;
- lignes 14-15 : si les valeurs postées sont toutes valides, alors il y a une redirection vers l'action [/js01] suivante :

```
1.     @RequestMapping(value = "/js01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.     public String js01(Form01 formulaire, Locale locale, Model model) {
3.        setModel(formulaire, model, locale, null);
4.        return "vue-01";
5.  }
```

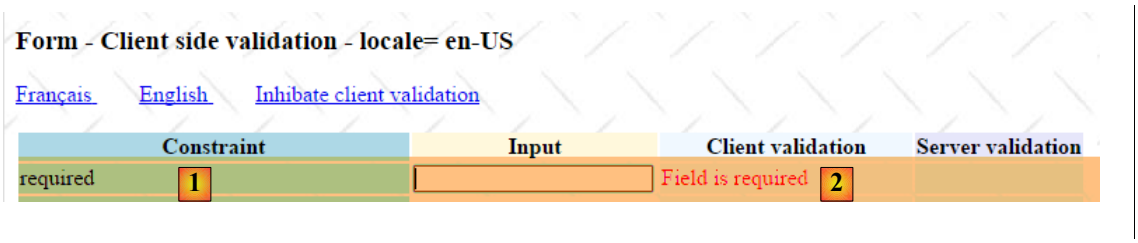
- ligne 2, la nouvelle locale [xx_XX] est injectée ;
- ligne 3 : la méthode [setModel] va alors mettre la culture du client à [xx-XX] ;

Maintenant regardons l'influence de la locale dans la vue [vue-01.xml]. Pour l'instant nous n'avons pas présenté celle-ci dans sa totalité car elle compte plus de 300 lignes. Néanmoins l'essentiel des lignes consiste en la répétition d'une séquence analogue à la suivante :

```
1.  <!-- required -->
2.  <tr>
3.    <td class="col1">required</td>
4.    <td class="col2">
5.      <input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-
   required=#{NotNull}" />
6.    </td>
7.     <td class="col3">
```

```
8.       <span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-
   replace="true"></span>
9.     </td>
10.   <td class="col4">
11.      <span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}"
   class="error">Donnée erronée</span>
12.   </td>
13. </tr>
```

Ce code affiche le fragment [1] suivant :



Le message d'erreur [2] provient de l'attribut [th:attr="data-val-required=#{NotNull}"] de la ligne 5. [#{NotNull}] est un message localisé. Selon la locale côté serveur, la ligne 5 génère la balise :

```
<input type="text" data-val="true" data-val-required="Field is required" id="strNotEmpty"
name="strNotEmpty" />
```

ou bien la balise :

```
<input type="text" data-val="true" data-val-required="Le champ est obligatoire" id="strNotEmpty"
name="strNotEmpty" />
```

Les attributs [data-x] sont exploités par la bibliothèque jS de validation.

Au final, on retiendra que les deux liens de changement de locale :

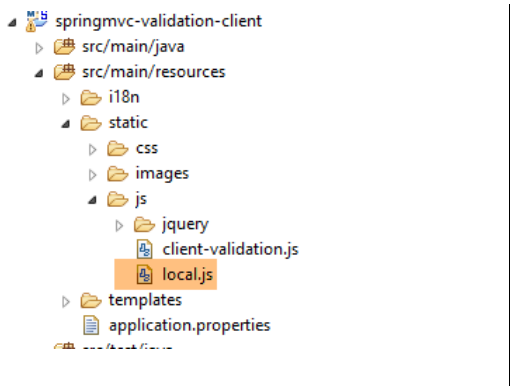
- provoquent un POST des valeurs saisies ;
- changent la locale à la fois côté serveur et côté client ;
- génèrent une page HTML qui emportent avec elle les messages d'erreur destinés à la bibliothèque jS de validation et que ces messages sont dans la langue de la locale choisie ;

6.3.7 Le POST des valeurs saisies

Etudions le bouton [Valider] qui poste les valeurs saisies de la vue [vue-01.xml]. Son code HTML est le suivant :

```
<!-- bouton de validation -->
<input type="submit" value="Valider" onclick="javascript:postForm01()" />
```

Si le Javascript est actif sur le navigateur, le clic sur le bouton va déclencher l'exécution de la méthode [postForm01]. Si cette fonction rend le booléen [False] alors le *submit* n'aura pas lieu. Si elle rend autre chose, alors il aura lieu. Cette fonction se trouve dans le fichier [local.js] :



Il est importé par la vue [vue-01.xml] par la ligne 6 ci-dessous :

```
1.     <head>
2.        <title>Spring 4 MVC</title>
3.        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
4.        <link rel="stylesheet" href="/css/form01.css" />
5.  ...
6.        <script type="text/javascript" src="/js/local.js"></script>
7.  </head>
```

Dans ce fichier, on trouve le code suivant :

```
1.  // variables globales
2.  var formulaire;
3.  var clientValidation;
4.  var double1;
5.  var double2;
6.  var double3;
7.  ...
8.  $(document).ready(function() {
9.     // références globales
10.    formulaire = $("#form");
11.    clientValidation = $("#clientValidation");
12.    double1 = $("#double1");
13.    double2 = $("#double2");
14.    double3 = $("#double3");
15. ...
16. });
17. ....
18. // post formulaire
19. function postForm01() {
20. ...
21. }
```

- lignes 8-16 : la fonction jS [$(document).ready(f)] est une fonction qui est exécutée lorsque le navigateur a chargé la totalité du document envoyé par le serveur. Son paramètre est une fonction. On utilise la fonction jS [$(document).ready(f)] pour initialiser l'environnement jS du document chargé ;
- ligne 10-14 : pour comprendre ces lignes, il faut à la fois regarder le code Thymeleaf et le code HTML généré ;

Le code Thymeleaf concerné est le suivant :

```
1.  <form action="/someURL" th:action="@{/js02.html}" method="post" th:object="${form01}" name="form"
    id="form">
2.  ...
3.  <input type="text" th:field="*{double1}" th:value="*{double1}" ... />
4.  ...
5.  <input type="text" th:field="*{double2}" th:value="*{double2}" />
6.  ...
7.  <input type="text" th:field="*{double3}" th:value="*{double3}" ... />
8.  ...
9.  <input type="hidden" th:field="*{clientValidation}" th:value="*{clientValidation}" value="true" />
```

qui génère le code HTML suivant :

```
1.  <form action="/js02.html" method="post" name="form" id="form">
```

```
2.  ...
3.  <input type="text" id="double1" name="double1" .../>
4.  ....
5.  <input type="text" value="" id="double2" name="double2" />
6.  ...
7.  <input value="" id="double3" name="double3" .../>
8.  ...
9.  <input type="hidden" value="false" id="clientValidation" name="clientValidation" />
```

Chaque attribut [th:field='x'] génère deux attributs HTML [name='x'] et [id='x']. L'attribut [name] est le nom des valeurs postées. Ainsi la présence des attributs [name='x'] et [value='y'] pour une balise HTML *<input type='text'>* va mettre la chaîne x=y dans les valeurs postées *name1=val1&name2=val2&...* L'attribut [id='x'] est lui utilisé par le Javascript. Il sert à identifier un élément du DOM (Document Object Model). Le document HTML chargé est en effet transformé en arbre Javascript appelé DOM où chaque noeud est repéré par son attribut [id].

Revenons au code de la fonction [$(document).ready()] :

```
1.  // variables globales
2.  var formulaire;
3.  var clientValidation;
4.  var double1;
5.  var double2;
6.  var double3;
7.  ...
8.  $(document).ready(function() {
9.      // références globales
10.     formulaire = $("#form");
11.     clientValidation = $("#clientValidation");
12.     double1 = $("#double1");
13.     double2 = $("#double2");
14.     double3 = $("#double3");
15. ...
16. });
17. ....
18. // post formulaire
19. function postForm01() {
20. ...
21. }
```

- ligne 10 : l'expression [$("#form")] est une expression jQuery. Sa valeur est une référence sur le noeud du DOM d'attribut [id='form '] ;
- lignes 10-14 : on récupère les références sur cinq noeuds du DOM ;
- lignes 2-6 : les variables déclarées en-dehors d'une fonction sont globales aux fonctions. Ici, cela signifie que les variables [formulaire, clientValidation , double1, double2, double3] initialisées dans [$(document).ready()] seront connues également dans la fonction [postForm01] de la ligne 19 ;

Maintenant, étudions la fonction [postForm01] :

```
1.  // post formulaire
2.  function postForm01() {
3.      // mode de validation côté client
4.      var validationActive = clientValidation.val() === "true";
5.      if (validationActive) {
6.          // on efface les erreurs du serveur
7.          clearServerErrors();
8.          // validation du formulaire
9.          if (!formulaire.validate().form()) {
10.             // pas de submit
11.             return false;
12.         }
13.     }
14.     // réels au format anglo-saxon
15.     var value1 = double1.val().replace(",", ".");
16.     double1.val(value1);
17.     var value2 = double2.val().replace(",", ".");
18.     double2.val(value2);
19.     var value3 = double3.val().replace(",", ".");
20.     double3.val(value3);
21.     // on laisse le submit se faire
22.     return true;
```

```
23. }
```

Rappelons que cette fonction jS est exécutée avant le [submit] du formulaire. Si elle rend le booléen [false] (ligne 11) alors le submit n'aura pas lieu. Si elle rend autre chose (ligne 22), alors il aura lieu.

- le code important est lignes 4-12 ;
- ligne 4 : on récupère la valeur du champ caché [clientValidation]. Cette valeur est 'true' si la validation client doit être activée, 'false' sinon ;
- ligne 6 : en cas de validation côté client, on efface les messages d'erreur du serveur qui peuvent être présents parce que l'utilisateur vient de changer de locale ;
- ligne 9 : rappelons que la variable [formulaire] représente le noeud de la balise HTML <form>, donc le formulaire. Celui-ci présente des validateurs jS que nous n'avons pas encore présentés et qui vont faire l'objet des paragraphes suivants. L'expression [formulaire.validate().form()] force l'exécution de tous les validateurs jS présents dans le formulaire. Sa valeur est [true] si les valeurs testées sont toutes valides, [false] sinon ;
- ligne 11 : on rend la valeur [false] si au moins l'une des valeurs testées est invalide. Cela empêchera le [submit] du formulaire au serveur ;
- lignes 15-20 : les identifiants [double1, double2, double3] représentent les trois nombres réels du formulaire. Selon la culture, la valeur saisie est différente. Avec la culture [fr-FR], on écrit [10,37] alors qu'avec la culture [en-US] on écrit [10.37]. Ca c'est pour la saisie. Avec la culture [fr-FR], la valeur postée pour [double1] ressemblera à [double1=10,37]. Arrivée côté serveur, la valeur [10,37] sera refusée car celui-ci attend [10.37], le format par défaut des nombres réels en Java. Aussi, les lignes 15-20, remplacent dans la valeur saisie pour ces nombres, la virgule par le point ;
- ligne 15 : l'expression [double1.val()] rend la chaîne de caractères saisie pour le noeud [double1]. L'expression [double1.val().replace(",", ".")] remplace dans cette chaîne, les virgules par des points. Le résultat est une chaîne [value1] ;
- ligne 16 : l'instruction [double1.val(value1)] affecte cette valeur [value1] au noeud [double1].

  Techniquement si l'utilisateur a saisi [10,37] pour le réel [double1], après les instruction précédentes le noeud [double1] a la valeur [10.37] et la valeur qui sera postée sera [param1=val1&**double1=10.37**&param2=val2], valeur qui sera acceptée par le serveur ;

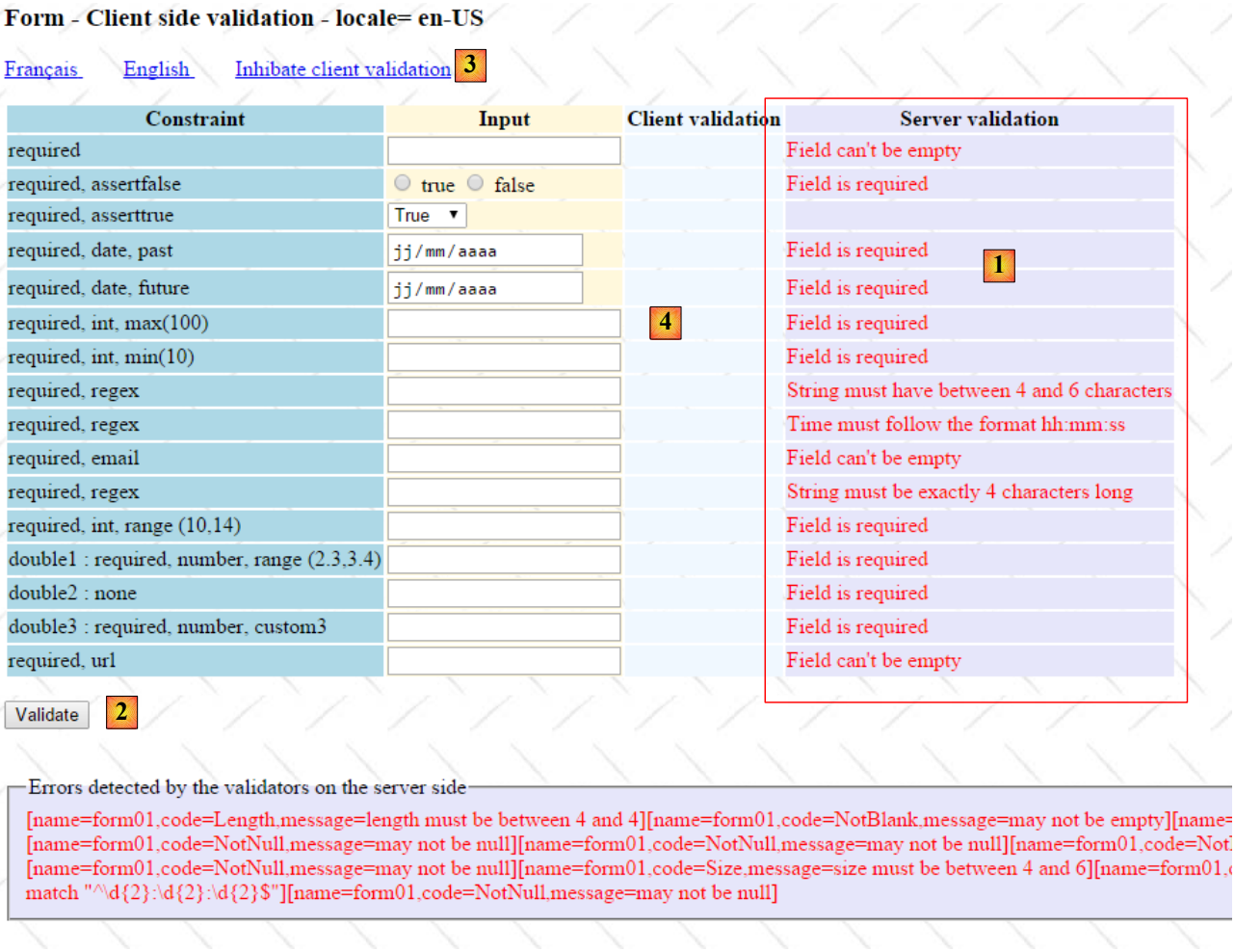
- ligne 22 : on rend la valeur [true] pour que le [submit] du formulaire s'exécute ;

On retiendra que la fonction jS [postForm01] :
- exécute tous les validateurs jS du formulaire si la validation côté client est activée et empêche le [submit] du formulaire au serveur si l'une des valeurs saisies a été déclarée invalide ;
- laisse faire le [submit] soit parce que la validation côté client n'est pas activée, soit parce qu'elle est activée et que toutes les valeurs saisies sont valides ;

Reste l'instruction de la ligne [3] :

```
1.     // on efface les erreurs du serveur
2.  clearServerErrors();
```

La fonction [`clearServerErrors`] a pour but l'effacement des messages présents dans la colonne 4 de la vue [vue-01.xml] :



Dans la copie d'écran ci-dessus, on a cliqué sur le lien [English]. Nous avons vu que cela provoquait un POST des valeurs saisies sans que les validateurs jS soient déclenchés. Au retour du POST, la colonne [Server Validation] se remplit des éventuels messages d'erreur. Si maintenant on clique sur le bouton [Validate] [2] avec les validateurs jS activés [3], alors la colonne [Client Validation] [4] va se remplir de messages. Si on ne fait rien, ceux qui étaient présents dans la colonne [Server Validation] vont rester ce qui va créer de la confusion puisque dans le cas d'erreurs détectées par les validateurs jS, le serveur n'est pas sollicité. Pour éviter cela, on efface la colonne [Server Validation] dans la fonction [postForm01]. C'est la fonction [] qui fait ce travail :

```
1.  function clearServerErrors() {
2.     // on efface les msg d'erreur du serveur
3.     $(".error").each(function(index) {
4.        $(this).text("");
5.     });
6.  }
```

Une particularité des messages d'erreur est qu'ils ont tous la classe [error]. Par exemple, pour la première ligne du tableau dans [vue-01.html] :

```
<span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}" class="error">Donnée erronée</span>
```

Et ce sont les seuls noeuds du DOM ayant cette classe. Nous utilisons cette propriété dans la fonction [clearServerErrors] :

```
1.  function clearServerErrors() {
2.     // on efface les msg d'erreur du serveur
3.     $(".error").each(function(index) {
4.        $(this).text("");
5.     });
6.  }
```

- ligne 3 : l'expression [$(".error")] ramène la collection des noeuds du DOM ayant la classe [error] ;
- ligne 3 : l'expression [$(".error").each(**function**(index){f}] exécute la fonction [f] pour chacun des noeuds de la collection. Elle reçoit un paramètre [index] qui n'est pas utilisé ici, qui est le n° du noeud dans la collection ;

- ligne 4 : l'expression [$(**this**)] désigne le noeud courant dans l'itération. Celui-ci est une balise HTML <span>. L'expression [$(**this**).text("")] attribue la chaîne vide au texte affiché par la balise ;

Nous allons examiner maintenant différents validateurs jS.

6.3.8 Validateur [required]

Examinons le premier élément du formulaire :

La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required -->
2.  <tr>
3.    <td class="col1">required</td>
4.    <td class="col2">
5.      <input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-required=#{NotNull}" />
6.    </td>
7.    <td class="col3">
8.      <span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-replace="true"></span>
9.    </td>
10.   <td class="col4">
11.     <span th:if="${#fields.hasErrors('strNotEmpty')}" th:errors="*{strNotEmpty}" class="error">Donnée
    erronée</span>
12.   </td>
13. </tr>
```

Ces lignes concernent le champ [strNotEmpty] du formulaire [Form01] :

```
1.      @NotNull
2.      @NotBlank
3.   private String strNotEmpty;
```

Les contraintes [1-2] font que le champ [strNotEmpty] doit être une chaîne existante [NotNull] et non vide et non constituée uniquement d'espaces [NotBlank]. On veut reproduire cette contrainte côté client avec du Javascript.

Etudions les lignes 5 et 8 . La ligne 11 ne pose pas de problème. Elle affiche le message d'erreur lié au champ [strNotEmpty]. Commençons par la ligne 5 :

```
<input type="text" th:field="*{strNotEmpty}" data-val="true" th:attr="data-val-
required=#{NotNull}" />
```

A partir de ce code, Thymeleaf va générer la balise suivante :

```
<input type="text" data-val="true" data-val-required="Field is required" id="strNotEmpty" name="strNotEmpty"
value="x" />
```

- l'attribut [data-val='true'] est utilisée par les bibliothèques jQuery de validation. Sa présence indique que la valeur du noeud fait l'objet d'une validation ;
- l'attribut [data-val-X='msg'] donne deux informations. [X] est le nom du validateur, [msg] est le message d'erreur associé à une valeur invalide du noeud sur lequel s'exerce le validateur. Ce n'est qu'une information. Cela ne provoque pas l'affichage du message d'erreur ;
- [required] est un validateur reconnu par la bibliothèque de validation [jquery.validate.unobstrusive] de Microsoft. Il n'y a pas besoin de le définir. Ce ne sera pas toujours le cas dans la suite ;
- les balises [data-x] sont ignorées par HTML5. Elles ne sont utiles que s'il y a du javascript pour les exploiter ;

Examinons la ligne 8 maintenant :

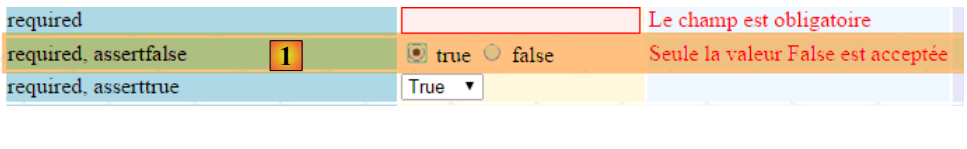
```
<span class="field-validation-valid" data-valmsg-for="strNotEmpty" data-valmsg-
replace="true"></span>
```

Elle sert à afficher le message d'ereur du validateur [required]. S'il y a erreur, la bibliothèque jS de validation va remplacer dynamiquement la ligne HTML du tableau par le code suivant :

```
1.  <tr>
2.     <td class="col1">required</td>
3.     <td class="col2">
4.        <input type="text" data-val="true" data-val-required="Le champ est obligatoire" id="strNotEmpty"
    name="strNotEmpty" value="" aria-required="true" aria-invalid="true" aria-describedby="strNotEmpty-error"
    class="input-validation-error">
5.     </td>
6.     <td class="col3">
7.        <span class="field-validation-error" data-valmsg-for="strNotEmpty" data-valmsg-replace="true">
8.           <span id="strNotEmpty-error" class="">Le champ est obligatoire</span>
9.        </span>
10.    </td>
11.    <td class="col4">
12.       <span class="error"></span>
13.    </td>
14. </tr>
15. </tr>
```

- ligne 4 : la classe du noeud [*strNotEmpty*] a changé. Elle est devenue [*input-validation-error*] qui fait que le champ erroné est coloré en rouge ;
- ligne 7 : la classe du [span] a changé. Elle est devenue [*field-validation-error*] qui va faire afficher le texte du [span] en rouge ;
- ligne 8 : le [span] qui était auparavant vide a maintenant un texte [Le champ est obligatoire]. Ce texte provient de la balise [data-val-required=*"Le champ est obligatoire"*] de la ligne 4 ;
- ligne 7 : pour afficher le message d'erreur du noeud [strNotEmpty] de la ligne 4, il faut utiliser ligne 7 les attributs [data-valmsg-for=*"strNotEmpty"*] et [data-valmsg-replace=*"true"*] ;

6.3.9 Validateur [assertfalse]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, assertfalse -->
2.  <tr>
3.     <td class="col1">required, assertfalse</td>
4.     <td class="col2">
5.        <input type="radio" th:field="*{assertFalse}" value="true" data-val="true"
6.           th:attr="data-val-required=#{NotNull},data-val-assertfalse=#{AssertFalse}" />
7.        <label th:for="${#ids.prev('assertFalse')}">true</label>
8.        <input type="radio" th:field="*{assertFalse}" value="false" data-val="true"
9.           th:attr="data-val-required=#{NotNull},data-val-assertfalse=#{AssertFalse}" />
10.       <label th:for="${#ids.prev('assertFalse')}">false</label>
11.    </td>
12.    <td class="col3">
13.       <span class="field-validation-valid" data-valmsg-for="assertFalse" data-valmsg-
    replace="true"></span>
14.    </td>
15.    <td class="col4">
16.       <span th:if="${#fields.hasErrors('assertFalse')}" th:errors="*{assertFalse}" class="error">Donnée
    erronée</span>
17.    </td>
18. </tr>
```

Ces lignes concernent le champ [*assertFalse*] du formulaire [Form01] :

```
1.      @NotNull
2.      @AssertFalse
```

```
3.  private Boolean assertFalse;
```

On veut reproduire cette contrainte côté client avec du Javascript. Les lignes 12-17 sont désormais classiques :

- lignes 12-14 : affichent en cas d'erreur sur le champ [assertFalse], le message transporté par l'attribut [data-val-assertfalse] de la ligne 6 ou celui transporté par l'attribut [data-val-required] de la même ligne. On rappelle que ces messages sont localisés, ç-à-d dans la langue choisie précédemment par l'utilisateur ou en français s'il n'a pas fait de choix ;
- lignes 5-10 : affichent les boutons radio avec des validateurs js qui sont déclenchés dès que l'utilisateur clique l'un d'eux.

Les deux boutons sont construits de la même façon. On va examiner le premier :

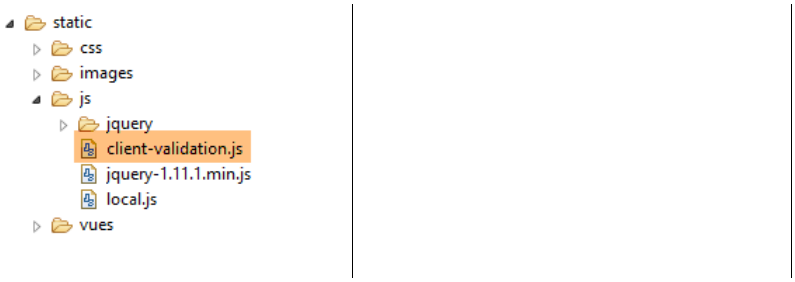
```
<input type="radio" th:field="*{assertFalse}" value="true" data-val="true" th:attr="data-val-required=#{NotNull},data-val-assertfalse=#{AssertFalse}" />
```

Une fois traitée par Thymeleaf cette ligne devient la suivante :

```
<input type="radio" value="true" data-val="true" data-val-required="Le champ est obligatoire" data-val-assertfalse="Seule la valeur False est acceptée" id="assertFalse1" name="assertFalse" />
```

On a des validateurs [data-val="true"]. On en a deux. Un validateur nommé [required] [data-val-required="Le champ est obligatoire"] et un autre nommé [assertfalse] [data-val-assertfalse="Seule la valeur False est acceptée"]. On rappelle que la valeur de l'attribut [data-val-X] est le message d'erreur du validateur X.

Nous avons vu le validateur [required]. La nouveauté ici est qu'on peut attacher plusieurs validateurs à une valeur saisie. Si le validateur [required] est connu de la bibliothèque MS (Microsoft) de validation, ce n'est pas le cas du validateur [assertFalse]. Nous allons donc apprendre à créer un nouveau validateur. Nous allons en créer plusieurs et ils seront placés dans un fichier [client-validation.js] :



Ce fichier, comme les autres, est importé par la vue [vue-01.xml] (ligne 6 ci-dessous) :

```
1.      <head>
2.          <title>Spring 4 MVC</title>
3.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
4.          <link rel="stylesheet" href="/css/form01.css" />
5.          ...
6.          <script type="text/javascript" src="/js/client-validation.js"></script>
7.  ...
8.  </head>
```

L'ajout du validateur [assertfalse] se résume à la création des deux fonctions jS suivantes :

```
1.  // -------------- assertfalse
2.  $.validator.addMethod("assertfalse", function(value, element, param) {
3.      return value === "false";
4.  });
5.
6.  $.validator.unobtrusive.adapters.add("assertfalse", [], function(options) {
7.      options.rules["assertfalse"] = options.params;
8.      options.messages["assertfalse"] = options.message.replace("'", "'");
9.  });
```

Très honnêtement, je ne suis pas un spécialiste de javascript, un langage qui garde encore pour moi toute son obscurité. Ses bases sont simples mais les bibliothèques posées sur ces bases sont souvent très complexes. Pour écrire les lignes de code ci-dessus, je me suis inspiré de codes trouvés sur Internet. C'est le lien [http://jsfiddle.net/LDDrk/] qui m'a donné la voie à suivre. S'il existe

encore, le lecteur est invité à le parcourir car il est complet avec un exemple fonctionnel à la clé. Il montre comment créer un nouveau validateur et il m'a permis de créer tous ceux de ce chapitre. Revenons au code :

- lignes 2-4 : définissent le nouveau validateur. La fonction [$.validator.addMethod] attend comme 1er paramètre, le nom du validateur, comme second paramètre une fonction définissant celui-ci ;
- ligne 2 : la fonction a trois paramètres :
  - [value] : la valeur à valider. La fonction doit rendre [true] si la valeur est valide, [false] sinon,
  - [element] : élément HTML à laquelle appartient la valeur à valider,
  - [param] : un objet contenant les valeurs associées aux paramètres d'un validateur. Nous n'avons pas encore introduit cette notion. Ici le validateur [assertFalse] n'a pas de paramètres. On peut dire si la valeur [value] est valide sans l'aide d'informations supplémentaires. Ce ne serait pas pareil s'il fallait vérifier que la valeur [value] était un nombre réel dans l'intervalle [min, max]. Alors là, il nous faudrait connaître [min] et [max]. On appelle ces deux valeurs les paramètres du validateur ;
- lignes 6-9 : une fonction nécessaire à la bibliothèque MS de validation. La fonction [$.validator.unobtrusive.adapters.add] attend comme 1er paramètre, le nom du validateur, comme second paramètre le tableau de paramètres du validateur, comme troisième paramètre une fonction ;
- le validateur [assertFalse] n'a pas de paramètres. C'est pourquoi le second paramètre est un tableau vide ;
- la fonction n'a qu'un paramètre, un objet [options] qui contient des informations sur l'élément à valider et pour lequel il faut définir deux nouvelles propriétés [rules] et [messages] ;
  - ligne 7 : on définit les règles [rules] pour le validateur [assertFalse]. Ces règles sont les paramètres du validateur [assertFalse], les mêmes que celles du paramètre [param] de la ligne 2. Ces paramètres sont trouvés dans [options.params] ;
  - ligne 8 : définissent le message d'erreur du validateur [assertFalse]. Celui-ci est trouvé dans [options.message]. On a la difficulté suivante avec les messages d'erreur. Dans les fichiers de messages, on va trouver le message suivant :

```
Range.form01.int1014=La valeur doit être dans l''intervalle [10,14]
```

    La double apostrophe est nécessaire pour Thymeleaf. Il l'interprète comme une apostrophe simple. Si on met une simple apostrophe, elle n'est pas affichée par Thymeleaf. Maintenant ces messages vont également servir de messages d'erreur pour la bibliothèque MS de validation. Or le javascript va lui afficher les deux apostrophes. Ligne 8, on remplace donc la double apostrophe du message d'erreur par une seule.

Pour voir un peu ce qui se passe, nous pouvons ajouter du code jS de log :

```
1.  // logs
2.  var logs = {
3.     assertfalse : true
4.  }
5.
6.
7.  // -------------- assertfalse
8.  $.validator.addMethod("assertfalse", function(value, element, param) {
9.     // logs
10.    if (logs.assertfalse) {
11.       console.log(jSON.stringify({
12.          "[assertfalse] value" : value
13.       }));
14.       console.log("[assertfalse] element");
15.       console.log(element);
16.       console.log(jSON.stringify({
17.          "[assertfalse] param" : param
18.       }));
19.    }
20.    // test validité
21.    return value === "false";
22. });
23.
24. $.validator.unobtrusive.adapters.add("assertfalse", [], function(options) {
25.    // logs
26.    if (logs.assertfalse) {
27.       console.log(jSON.stringify({
28.          "[assertfalse] options.params" : options.params
29.       }));
30.       console.log(jSON.stringify({
31.          "[assertfalse] options.message" : options.message
32.       }));
33.       console.log(jSON.stringify({
34.          "[assertfalse] options.messages" : options.messages
```

```
35.         }));
36.     }
37.     // code
38.     options.rules["assertfalse"] = options.params;
39.     options.messages["assertfalse"] = options.message.replace("'", "'");
40. });
```

Ce code utilise la bibliothèque jSON JSON3 [http://bestiejs.github.io/json3/]. Si on active les logs (ligne 3), on obtient les affichages suivants dans la console :

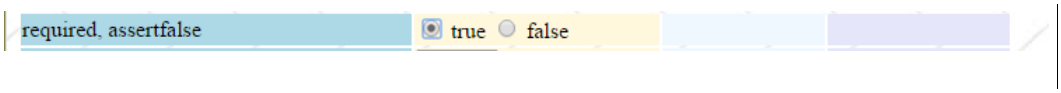
Au chargement initial de la page, on a les logs suivants :

```
{"[assertfalse] options.params":{}}
{"[assertfalse] options.message":"Seule la valeur False est acceptée"}
{"[assertfalse] options.messages":{"required":"Le champ est obligatoire"}}
```

La fonction jS [$.validator.unobtrusive.adapters.add] a été exécutée. On apprend les choses suivantes :

- [options.params] est un objet vide car le validateur [assertFalse] n'a pas de paramètres ;
- [options.message] est le message d'erreur qu'on a construit pour le validateur [assertFalse] dans l'attribut [data-val-assertFalse] ;
- [options.messages] est un objet qui contient les autres messages d'erreur de l'élément validé. Ici on retrouve le message d'erreur que nous avons mis dans l'attribut [data-val-required] ;

Maintenant donnons une valeur erronée au champ [assertFalse] et validons :



On obtient alors les logs suivants :

```
{"[assertfalse] value":"true"}                                   client-validation.js:118
[assertfalse] element                                            client-validation.js:121
                                                                 client-validation.js:122
<input type="radio" value="true" data-val="true" data-val-required="Le champ est
obligatoire" data-val-assertfalse="Seule la valeur False est acceptée" id="assertFalse1"
name="assertFalse" aria-required="true" class="input-validation-error" aria-describedby=
"assertFalse-error">
{"[assertfalse] param":{}}                                       client-validation.js:123
```

On y voit les choses suivantes :

- la valeur testée est la valeur [true] (ligne 118) ;
- l'élément HTML testé est le bouton radio d'id [assertFalse1] (ligne 122) ;
- le validateur [assertFalse] n'a pas de paramètre (ligne 123) ;

Voilà. Que retenir de tout cela ?

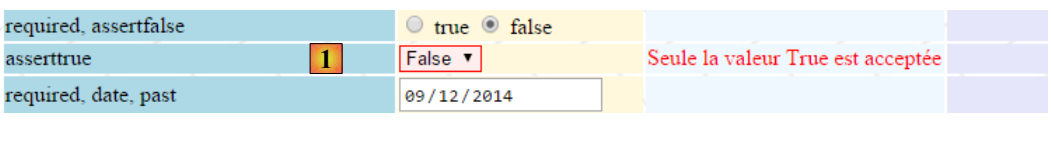
Pour un validateur X jS, nous devons définir :

- dans la balise HTML à valider, l'attribut [data-val-X='msg'] qui définit à la fois le validateur X et son message d'erreur ;
- deux fonctions jS à mettre dans le fichier [client-validation.js] :
  - [$.validator.addMethod("X", **function**(value, element, param)],
  - [$.validator.unobtrusive.adapters.add("X", [param1, param2], **function**(options)] ;

Par la suite, nous allons nous appuyer sur ce qui a été fait pour ce premier validateur et simplement présenter ce qui est nouveau.

6.3.10 Validateur [asserttrue]

Ce validateur est bien évidemment analogue au validateur [assertFalse].



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, asserttrue -->
2.  <tr>
3.     <td class="col1">asserttrue</td>
4.     <td class="col2">
5.        <select th:field="*{assertTrue}" data-val="true" th:attr="data-val-asserttrue=#{AssertTrue}">
6.           <option value="true">True</option>
7.           <option value="false">False</option>
8.        </select>
9.     </td>
10.    <td class="col3">
11.       <span class="field-validation-valid" data-valmsg-for="assertTrue" data-valmsg-replace="true"></span>
12.    </td>
13.    <td class="col4">
14.       <span th:if="${#fields.hasErrors('assertTrue')}" th:errors="*{assertTrue}" class="error">Donnée
   erronée</span>
15.    </td>
16. </tr>
```

Ces lignes concernent le champ [*assertTrue*] du formulaire [Form01] :

```
1.      @NotNull
2.      @AssertTrue
3.   private Boolean assertTrue;
```

Il n'y a rien de nouveau dans les lignes 1-16. Elles utilisent un validateur [asserrtrue] qu'il faut définir dans le fichier [client-validation.js] :

```
1.  // -------------- asserttrue
2.  $.validator.addMethod("asserttrue", function(value, element, param) {
3.     return value === "true";
4.  });
5.
6.  $.validator.unobtrusive.adapters.add("asserttrue", [], function(options) {
7.     options.rules["asserttrue"] = options.params;
8.     options.messages["asserttrue"] = options.message.replace("'", "'");
9.  });
```

6.3.11 Validateurs [date] et [past]

La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, date, past -->
2.  <tr>
3.     <td class="col1">required, date, past</td>
4.     <td class="col2">
5.        <input type="date" th:field="*{dateInPast}" th:value="*{dateInPast}" data-val="true"
6.           th:attr="data-val-required=#{NotNull},data-val-date=#{DateInvalide.form01},data-val-
   past=#{Past.form01.dateInPast}" />
7.     </td>
8.     <td class="col3">
9.        <span class="field-validation-valid" data-valmsg-for="dateInPast" data-valmsg-replace="true"></span>
10.    </td>
```

```
11.     <td class="col4">
12.         <span th:if="${#fields.hasErrors('dateInPast')}" th:errors="*{dateInPast}" class="error">Donnée
    erronée</span>
13.     </td>
14. </tr>
```

Ces lignes concernent le champ [*dateInPast*] du formulaire [Form01] :

```
1.      @NotNull
2.      @Past
3.      @DateTimeFormat(pattern = "yyyy-MM-dd")
4.  private Date dateInPast;
```

La ligne des validateurs de la date est la suivante :

```
<input type="date" th:field="*{dateInPast}" th:value="*{dateInPast}" data-val="true"
   th:attr="data-val-required=#{NotNull},data-val-date=#{DateInvalide.form01},data-val-
past=#{Past.form01.dateInPast}" />
```

On y trouve trois validateurs [data-val-X] : required, date, past. Il nous faut définir dans [client-validation.js] les fonctions associées à ces deux nouveaux validateurs :

```
1.  logs.date = true;
2.  // -------------- date
3.  $.validator.addMethod("date", function(value, element, param) {
4.      // validité
5.      var valide = Globalize.parseDate(value, "yyyy-MM-dd") != null;
6.      // logs
7.      if (logs.date) {
8.          console.log(jSON.stringify({
9.              "[date] value" : value,
10.             "[date] valide" : valide
11.         }));
12.     }
13.     // résultat
14.     return valide;
15. });
16.
17. $.validator.unobtrusive.adapters.add("date", [], function(options) {
18.     options.rules["date"] = options.params;
19.     options.messages["date"] = options.message.replace("'", "");
20. });
```

et

```
1.  logs.past = true;
2.  // -------------- past
3.  $.validator.addMethod("past", function(value, element, param) {
4.      // validité
5.      var valide = value <= new Date().toISOString().substring(0, 10);
6.      // logs
7.      if (logs.past) {
8.          console.log(jSON.stringify({
9.              "[past] value" : value,
10.             "[past] valide" : valide
11.         }));
12.     }
13.     // résultat
14.     return valide;
15. });
16.
17. $.validator.unobtrusive.adapters.add("past", [], function(options) {
18.     options.rules["past"] = options.params;
19.     options.messages["past"] = options.message.replace("'", "");
20. });
```

Avant d'expliquer le code, regardons les logs lorsqu'on saisit une date postérieure à celle d'aujourd'hui :

```
{"[date] value":"2014-12-13","[date] valide":true}
{"[past] value":"2014-12-13","[past] valide":false}
```

La première chose à remarquer est que la date à valider arrive comme une chaîne de caractères de format [aaaa-mm-jj]. Ce qui explique les lignes suivantes :

```
var valide = Globalize.parseDate(value, "yyyy-MM-dd") != null;
```

La bibliothèque [globalize.js] amène la fonction [Globalize.parseDate] ci-dessus. Le 1er paramètre est la date en tant que chaîne de caractères et le second son format. Le résultat est un pointeur *null* si la date est invalide, la date résultante sinon.

La validité du validateur [past] est vérifiée par le code suivant :

```
var valide = value <= new Date().toISOString().substring(0, 10);
```

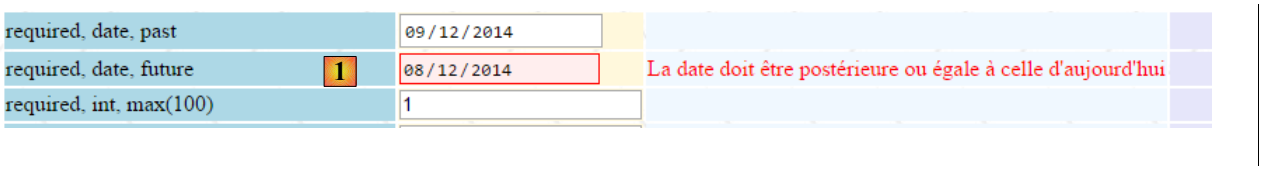
Voici sur une console l'évaluation de l'expression [**new** Date().toISOString().substring(0, 10)] :

```
> new Date().toISOString()
"2014-12-12T12:54:03.509Z"
> new Date().toISOString().substring(0, 10)
"2014-12-12"
>
```

La chaîne de caractères [value] doit précéder **alphabétiquement** la chaine [**new** Date().toISOString().substring(0, 10)] pour être valide.

On notera que la version de Chrome utilisée fournit la date au format [yyyy-mm-dd]. Pour un navigateur où ce ne serait pas le cas, il faudrait indiquer explicitement à l'utilisateur d'utiliser ce format de saisie.

6.3.12 Validateur [future]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, date, future -->
2.  <tr>
3.      <td class="col1">required, date, future</td>
4.      <td class="col2">
5.          <input type="date" th:field="*{dateInFuture}" th:value="*{dateInFuture}" data-val="true"
    th:attr="data-val-required=#{NotNull},data-val-date=#{DateInvalide.form01},data-val-
    future=#{Future.form01.dateInFuture}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="dateInFuture" data-valmsg-
    replace="true"></span>
9.      </td>
10.     <td class="col4">
11.         <span th:if="${#fields.hasErrors('dateInFuture')}" th:errors="*{dateInFuture}" class="error">Donnée
    erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [*dateInFuture*] du formulaire [Form01] :

```
@NotNull
@Future
```

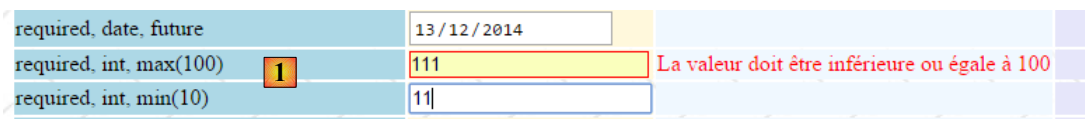
```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date dateInFuture;
```

- ligne 5, apparaît un nouveau validateur [*data-val-future*] ;

Ce validateur est bien sûr très analogue au validateur [past]. Les deux fonctions à ajouter dans [client-validation.js] sont les suivantes :

```
1.  // -------------- future
2.  $.validator.addMethod("future", function(value, element, param) {
3.      var now = new Date().toISOString().substring(0, 10);
4.      return value > now;
5.  });
6.
7.  $.validator.unobtrusive.adapters.add("future", [], function(options) {
8.      options.rules["future"] = options.params;
9.      options.messages["future"] = options.message.replace("'", "'");
10. });
```

6.3.13 Validateurs [int] et [max]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, int, max(100) -->
2.  <tr>
3.      <td class="col1">required, int, max(100)</td>
4.      <td class="col2">
5.          <input type="text" th:field="*{intMax100}" th:value="*{intMax100}" data-val="true"
        th:attr="data-val-required=#{NotNull},data-val-int=#{typeMismatch},data-val-
    max=#{Max.form01.intMax100},data-val-max-value=#{form01.intMax100.value}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="intMax100" data-valmsg-replace="true"></span>
9.      </td>
10.     <td class="col4">
11.         <span th:if="${#fields.hasErrors('intMax100')}" th:errors="*{intMax100}" class="error">Donnée
    erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [*intMax100*] du formulaire [Form01] :

```
@NotNull
@Max(value = 100)
private Integer intMax100;
```

Ligne 5, il y a deux nouveaux validateurs : [int] et [max]. Ce dernier a un paramètre : la valeur du maximum. Examinons le code HTML généré par la ligne 5 :

```
1.  <!-- required, int, max(100) -->
2.  <tr>
3.      <td class="col1">required, int, max(100)</td>
4.      <td class="col2">
5.          <input type="text" data-val="true" data-val-int="Format invalide" data-val-max-value="100" data-val-
    required="Le champ est obligatoire" data-val-max="La valeur doit être inférieure ou égale à 100" value=""
    id="intMax100" name="intMax100" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="intMax100" data-valmsg-replace="true"></span>
9.      </td>
10.     <td class="col4">
11.
```

```
12.     </td>
13. </tr>
```

Rappelons la signification des différents attributs [data-X] :

- [data-val="*true*"] indique que des validateurs sont associés à l'élément HTML ;
- [data-val-required] introduit le validateur [required] avec son message ;
- [data-val-int] introduit le validateur [int] avec son message ;
- [data-val-max] introduit le validateur [max] avec son message ;
- [data-val-max-value="*100*"] introduit un paramètre nommé [value] pour le validateur [max]. [100] est la valeur de ce paramètre. C'est la première fois que nous rencontrons la notion de paramètres d'un validateur.

Le fichier [client-validation.js] est enrichi du validateur [int] suivant :

```
1.  logs.int = true;
2.  // -------------- int
3.  $.validator.addMethod("int", function(value, element, param) {
4.      // validité
5.      valide = /^\s*[-\+]?\s*\d+\s*$/.test(value);
6.      // logs
7.      if (logs.int) {
8.          console.log(jSON.stringify({
9.              "[int] value" : value,
10.             "[int] valide" : valide,
11.         }));
12.     }
13.     // résultat
14.     return valide;
15. });
16.
17. $.validator.unobtrusive.adapters.add("int", [], function(options) {
18.     options.rules["int"] = options.params;
19.     options.messages["int"] = options.message.replace("'", "'");
20. });
```

- ligne 5 : on utilise une expression régulière pour vérifier que la chaîne [value] représente bien un entier. Celui-ci peut être signé ;

Voici quelques exemples de logs :

```
1.  {"[int] value":"x","[int] valide":false}
2.  {"[int] value":"11","[int] valide":true}
3.  {"[int] value":"11x","[int] valide":false}
```

Le validateur [max] est ajouté de la façon suivante dans [client-validation.js]

```
1.  // -------------- max à utiliser conjointement avec [int] ou [number]
2.  logs.max = true;
3.  $.validator.addMethod("max", function(value, element, param) {
4.      // logs
5.      if (logs.max) {
6.          console.log(jSON.stringify({
7.              "[max] value" : value,
8.              "[max] param" : param
9.          }));
10.     }
11.     // validité
12.     var val = Globalize.parseFloat(value);
13.     if (isNaN(val)) {
14.         // logs
15.         if (logs.max) {
16.             console.log(jSON.stringify({
17.                 "[max] valide" : true
18.             }));
19.         }
20.         // résultat
21.         return true;
22.     }
23.     var max = Globalize.parseFloat(param.value);
24.     var valide = val <= max;
25.     // logs
26.     if (logs.max) {
```

```
27.         console.log(jSON.stringify({
28.            "[max] valide" : valide
29.         }));
30.     }
31.     // résultat
32.     return valide;
33. });
34.
35. $.validator.unobtrusive.adapters.add("max", [ "value" ], function(options) {
36.     options.rules["max"] = options.params;
37.     options.messages["max"] = options.message.replace("'{'", "'");
38. });
```

Nous allons traiter tout de suite le cas du paramètre [value] du validateur [max] introduit par l'attribut [data-val-max-value="100"].

- ligne 35, le paramètre [value] est intégré dans le second paramètre de la fonction [$.validator.unobtrusive.adapters.add] ;
- ligne 3, l'objet [param] ne va plus être vide, mais contenir {"value":100} ;

Pour comprendre le code des lignes 3-33, il faut savoir que lorsqu'il y a plusieurs validateurs sur un même élément HTML :

- on ne connaît pas l'ordre d'exécution des validateurs ;
- l'exécution des validateurs s'arrête dès qu'un validateur déclare l'élément invalide. C'est alors le message d'erreur de ce dernier qui est associé à l'élément invalide ;

Etudions le code :

- ligne 12 : on vérifie qu'on a un nombre. Si le validateur [int] a été exécuté avant le validateur [max], c'est forcément vrai puisque une valeur invalide arrête l'exécution des validateurs ;
- lignes 13-22 : si on n'a pas un nombre, cela veut dire que le validateur [int] n'a pas encore été exécuté. On indique alors que la valeur testée est valide pour laisser le validateur [int] faire son travail et déclarer l'élément invalide avec son propre message d'erreur ;
- lignes 23-24 : calcule la validité de [value] ;

Voici quelques logs :

Valeur saisie	logs
x	{"[max] value":"x","[max] param":{"value":"100"}} {"[max] valide":true} {"[int] value":"x","[int] valide":false}
111	{"[max] value":"111","[max] param":{"value":"100"}} {"[max] valide":false}
111x	{"[max] value":"111x","[max] param":{"value":"100"}} {"[max] valide":true} {"[int] value":"111x","[int] valide":false}

6.3.14 Validateur [min]

La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, int, min(10) -->
2.  <tr>
3.      <td class="col1">required, int, min(10)</td>
4.      <td class="col2">
5.          <input type="text" th:field="*{intMin10}" th:value="*{intMin10}" data-val="true"
        th:attr="data-val-required=#{NotNull},data-val-int=#{typeMismatch},data-val-
    min=#{Min.form01.intMin10},data-val-min-value=#{form01.intMin10.value}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="intMin10" data-valmsg-replace="true"></span>
```

```
9.      </td>
10.     <td class="col4">
11.        <span th:if="${#fields.hasErrors('intMin10')}" th:errors="*{intMin10}" class="error">Donnée
    erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [*intMin10*] du formulaire [Form01] :

```
1.      @NotNull
2.      @Min(value = 10)
3.  private Integer intMin10;
```

La ligne 5 introduit un nouveau validateur [min] [*data-val-int=#{typeMismatch}*] avec un paramètre [value] [*data-val-min-value=#{form01.intMin10.value}*"]. On a là un cas analogue au validateur [max]. On ajoute dans [client-validation.js] le code suivant :

```
1.  logs.min = true;
2.  //-------------- min à utiliser conjointement avec [int] ou [number]
3.  $.validator.addMethod("min", function(value, element, param) {
4.     // logs
5.     if (logs.min) {
6.        console.log(jSON.stringify({
7.           "[min] value" : value,
8.           "[min] param" : param
9.        }));
10.    }
11.    // validité
12.    var val = Globalize.parseFloat(value);
13.    if (isNaN(val)) {
14.       // logs
15.       if (logs.min) {
16.          console.log(jSON.stringify({
17.             "[min] valide" : true
18.          }));
19.       }
20.       // résultat
21.       return true;
22.    }
23.    var min = Globalize.parseFloat(param.value);
24.    var valide = val >= min;
25.    // logs
26.    if (logs.min) {
27.       console.log(jSON.stringify({
28.          "[min] valide" : valide
29.       }));
30.    }
31.    // résultat
32.    return valide;
33. });
34.
35. $.validator.unobtrusive.adapters.add("min", [ "value" ], function(options) {
36.    options.rules["min"] = options.params;
37.    options.messages["min"] = options.message.replace("'{'", "'");
38. });
```

Voici quelques logs d'exécution :

Valeur saisie	logs
x	{"[min] value":"x","[min] param":{"value":"10"}} {"[min] valide":true} {"[int] value":"x","[int] valide":false}
11	{"[min] value":"11","[min] param":{"value":"10"}} {"[min] valide":true} {"[int] value":"11","[int] valide":true}
8x	{"[min] value":"8x","[min] param":{"value":"10"}} {"[min] valide":true} {"[int] value":"8x","[int] valide":false}

6.3.15 Validateur [regex]

La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, regex -->
2.  <tr>
3.      <td class="col1">required, regex</td>
4.      <td class="col2">
5.          <input type="text" th:field="*{strBetween4and6}" th:value="*{strBetween4and6}" data-val="true"
        th:attr="data-val-required=#{NotNull},data-val-regex=#{Size.form01.strBetween4and6}, data-val-regex-
    pattern=#{form01.strBetween4and6.pattern}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="strBetween4and6" data-valmsg-
    replace="true"></span>
9.      </td>
10.     <td class="col4">
11.         <span th:if="${#fields.hasErrors('strBetween4and6')}" th:errors="*{strBetween4and6}"
    class="error">Donnée erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [*strBetween4and6*] du formulaire [Form01] :

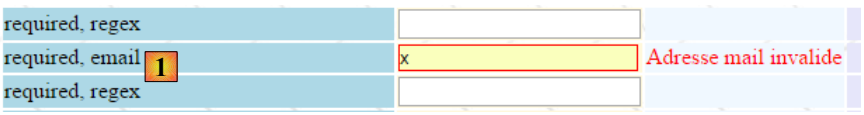
```
    @NotNull
    @Size(min = 4, max = 6)
    private String strBetween4and6;
```

La ligne 5 génère le HTML suivant :

```
<input type="text" data-val="true" data-val-required="Le champ est obligatoire" data-val-regex="La chaîne doit
avoir entre 4 et 6 caractères" data-val-regex-pattern="^.{4,6}$" value="" id="strBetween4and6"
name="strBetween4and6" />
```

Cette balise introduit le validateur [regex] [data-val-regex=*"La chaîne doit avoir entre 4 et 6 caractères"*] avec son paramètre [pattern] [data-val-regex-pattern=*"^.{4,6}$"*]. Le paramètre [pattern] est l'expression régulière que doit vérifier la valeur à valider. Ici l'expression régulière vérifie que la chaîne a entre 4 et 6 caractères quelconques. Le validateur [regex] est prédéfini dans la bibliothèque de validation MS. Il n'y a donc rien à ajouter dans le fichier [client-validation.js].

6.3.16 Validateur [email]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, email -->
2.  <tr>
3.      <td class="col1">required, email</td>
4.      <td class="col2">
5.          <input type="text" th:field="*{email}" th:value="*{email}" data-val="true"
        th:attr="data-val-required=#{NotNull},data-val-email=#{Email.form01.email}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="email" data-valmsg-replace="true"></span>
```

```
9.      </td>
10.     <td class="col4">
11.        <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}" class="error">Donnée erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [email] du formulaire [Form01] :

```
@NotNull
@Email
@NotBlank
private String email;
```

La ligne 5 génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-required="Le champ est obligatoire" data-val-email="Adresse mail invalide" value="" id="email" name="email" />
```

Cette balise introduit le validateur [email] [data-val-email="Adresse mail invalide"]. Le validateur [email] est prédéfini dans la bibliothèque de validation MS. Il n'y a donc rien à ajouter dans le fichier [client-validation.js].

6.3.17 Validateur [range]

La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, int, range (10,14) -->
2.  <tr>
3.     <td class="col1">required, int, range (10,14)</td>
4.     <td class="col2">
5.        <input type="text" th:field="*{int1014}" th:value="*{int1014}" data-val="true"
       th:attr="data-val-required=#{NotNull},data-val-int=#{typeMismatch}, data-val-
    range=#{Range.form01.int1014},data-val-range-max=#{form01.int1014.max},data-val-range-
    min=#{form01.int1014.min}" />
6.     </td>
7.     <td class="col3">
8.        <span class="field-validation-valid" data-valmsg-for="int1014" data-valmsg-replace="true"></span>
9.     </td>
10.    <td class="col4">
11.       <span th:if="${#fields.hasErrors('int1014')}" th:errors="*{int1014}" class="error">Donnée
    erronée</span>
12.    </td>
13. </tr>
```

Ces lignes concernent le champ [int1014] du formulaire [Form01] :

```
@Range(min = 10, max = 14)
@NotNull
private Integer int1014;
```

La ligne 5 génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-range-max="14" data-val-range="La valeur doit être dans l&#39;&#39;intervalle [10,14]" data-val-int="Format invalide" data-val-required="Le champ est obligatoire" data-val-range-min="10" value="" id="int1014" name="int1014" />
```

Cette balise introduit un nouveau validateur [range] [data-val-range="La valeur doit être dans l''intervalle [10,14]"] qui a deux paramètres [min] [data-val-range-min="10"] et [max] [data-val-range-max="14"].

Dans le fichier [client-validation.js], nous définissons le validateur [range] de la façon suivante :

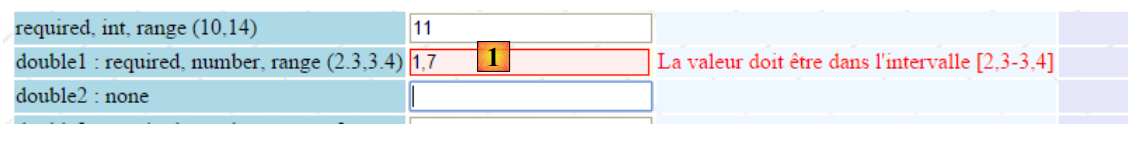
```
1.  // -------------- range à utiliser conjointement avec [int] ou [number]
2.  logs.range=true
3.  $.validator.addMethod("range", function(value, element, param) {
4.     // logs
5.     if (logs.range) {
6.        console.log(jSON.stringify({
7.           "[range] value" : value,
8.           "[range] param" : param
9.        }));
10.    }
11.    // validité
12.    var val = Globalize.parseFloat(value);
13.    if (isNaN(val)) {
14.       // logs
15.       if (logs.min) {
16.          console.log(jSON.stringify({
17.             "[range] valide" : true
18.          }));
19.       }
20.       // terminé
21.       return true;
22.    }
23.    var min = Globalize.parseFloat(param.min);
24.    var max = Globalize.parseFloat(param.max);
25.    var valide = val >= min && val <= max;
26.    // logs
27.    if (logs.range) {
28.       console.log(jSON.stringify({
29.          "[range] valide" : valide
30.       }));
31.    }
32.    // terminé
33.    return valide;
34. });
35.
36. $.validator.unobtrusive.adapters.add("range", [ "min", "max" ], function(options) {
37.    options.rules["range"] = options.params;
38.    options.messages["range"] = options.message.replace("'", "'");
39. });
```

Il est très semblable aux validateurs [min] et [max] déjà étudiés.

Voici quelques exemples de logs :

Valeur saisie	logs
x	{"[range] value":"x","[range] param":{"min":"10","max":"14"}} {"[int] value":"x","[int] valide":false}
8	{"[range] value":"8","[range] param":{"min":"10","max":"14"}} {"[range] valide":false}
11	{"[range] valide":true} {"[int] value":"11","[int] valide":true}

6.3.18 Validateur [number]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- double1 : required, number, range (2.3,3.4) -->
2.  <tr>
3.     <td class="col1">double1 : required, number, range (2.3,3.4)</td>
```

```
4.      <td class="col2">
5.         <input type="text" th:field="*{double1}" th:value="*{double1}" data-val="true"
6.            th:attr="data-val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-
    range=#{Range.form01.double1},data-val-range-max=#{form01.double1.max},data-val-range-
    min=#{form01.double1.min}" />
7.      </td>
8.      <td class="col3">
9.         <span class="field-validation-valid" data-valmsg-for="double1" data-valmsg-replace="true"></span>
10.     </td>
11.     <td class="col4">
12.        <span th:if="${#fields.hasErrors('double1')}" th:errors="*{double1}" class="error">Donnée
    erronée</span>
13.     </td>
14. </tr>
```

Ces lignes concernent le champ [*double1*] du formulaire [Form01] :

```
1.      @NotNull
2.      @DecimalMax(value = "3.4")
3.      @DecimalMin(value = "2.3")
4.  private Double double1;
```

La ligne 5 génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-number="Format invalide" data-val-range-max="3.4"
   data-val-range="La valeur doit être dans l&#39;intervalle [2,3-3,4]" data-val-required="Le
champ est obligatoire" data-val-range-min="2.3" value="" id="double1" name="double1" />
```

La balise introduit un nouveau validateur [number] avec l'attribut [`data-val-number="Format invalide"`]. Ce validateur est défini de la façon suivante dans le fichier [client-validation.js] :

```
1.  // -------------- number
2.  logs.number = true;
3.  $.validator.addMethod("number", function(value, element, param) {
4.      var valide = !isNaN(Globalize.parseFloat(value));
5.      // logs
6.      if (logs.number) {
7.          console.log(jSON.stringify({
8.              "[number] value" : value,
9.              "[number] valide" : valide
10.         }));
11.     }
12.     // résultat
13.     return valide;
14. });
15.
16. $.validator.unobtrusive.adapters.add("number", [], function(options) {
17.     options.rules["number"] = options.params;
18.     options.messages["number"] = options.message.replace("'", "'");
19. });
```

Voici quelques exemples de logs :

Valeur saisie	logs
x	{"[number] value":"x","[number] valide":false}
-2,5	{"[number] value":"-2,5","[number] valide":true} {"[range] value":"-2,5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":false}
2,5	{"[number] value":"+2,5","[number] valide":true} {"[range] value":"+2,5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":true}
+2.5	{"[number] value":"+2.5","[number] valide":true} {"[range] value":"+2.5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":true}

On sait que les nombres réels sont sensibles à la culture. Ci-dessus, on est dans la culture [fr-FR]. Lorsqu'on saisit [2.5] (notation anglo-saxonne), le nombre est accepté. C'est la faute à [Globalize.parseFloat] qui accepte les deux notations :

```
Globalize.parseFloat("3.3")
3.3
Globalize.parseFloat("3,3")
3.3
```

Passons en anglais et faisons les saisies [+2,5] et [+2.5]. Les logs sont les suivants :

Valeur saisie	logs
x	{"[number] value":"x","[number] valide":false}
2,5	{"[number] value":"+2,5","[number] valide":true} {"[range] value":"+2,5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":false}
+2.5	{"[number] value":"+2.5","[number] valide":true} {"[range] value":"+2.5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":true}

Il y a un problème avec [2,5]. Il a été déclaré comme un réel valide alors qu'il faut écrire [2.5]. C'est la faute à [Globalize.parseFloat] :

```
Globalize.parseFloat("2,5")
25
```

Ci-dessus, [Globalize.parseFloat] ignore la virgule et considère que le nombre est 25. Dans la culture [en-US], un nombre réel peut comporter un point décimal et des virgules qui sont utilisées parfois pour séparer les milliers.

On peut améliorer les choses de la façon suivante :

```
1.  // -------------- number
2.  logs.number = true;
3.  $.validator.addMethod("number", function(value, element, param) {
4.     // on gère les cultures [fr-FR] et [en-US] uniquement
5.     var pattern_fr_FR = /^\s*[-+]?[0-9]*\,?[0-9]+\s*$/;
6.     var pattern_en_US = /^\s*[-+]?[0-9]*\.?[0-9]+\s*$/;
7.     var culture = Globalize.culture().name;
8.     // test de validité
9.     var valide;
10.    if (culture === "fr-FR") {
11.       valide = pattern_fr_FR.test(value);
12.    } else if (culture === "en-US") {
13.       valide = pattern_en_US.test(value);
14.    } else {
15.       valide = !isNaN(Globalize.parseFloat(value));
16.    }
17.    // logs
18.    if (logs.number) {
19.       console.log(jSON.stringify({
20.          "[number] value" : value,
21.          "[number] culture" : culture,
22.          "[number] valide" : valide
23.       }));
24.    }
25.    // résultat
26.    return valide;
27. });
```

- ligne 5 : l'expression régulière d'un nombre réel pour la culture [fr-FR] ;
- ligne 6 : l'expression régulière d'un nombre réel pour la culture [en-US] ;
- ligne 7 : le nom de la culture du moment. Dans notre exemple, ce sera l'une de deux cultures ci-dessus ;
- lignes 9-16 : le test de validité de la valeur saisie ;
- ligne 15 : on a prévu le cas où la culture ne serait ni [fr-FR], ni [en-US] ;

Les logs donnent maintenant la chose suivante :

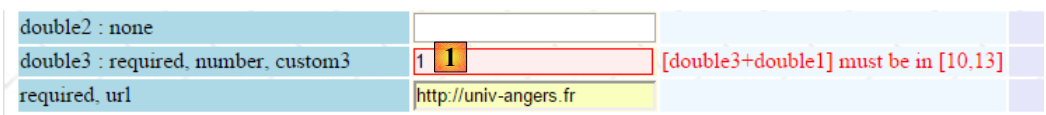
Culture [fr-FR]

Valeur saisie	logs
x	{"[number] value":"x","[number] culture":"fr-FR","[number] valide":false}
-2,5	{"[number] value":"-2,5","[number] culture":"fr-FR","[number] valide":true} {"[range] value":"-2,5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":false}
2,5	{"[number] value":"+2,5","[number] culture":"fr-FR","[number] valide":true} {"[range] value":"+2,5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":true}
+2.5	{"[number] value":"+2.5","[number] culture":"fr-FR","[number] valide":false}

Culture [en-US]

Valeur saisie	logs
x	{"[number] value":"x","[number] culture":"en-US","[number] valide":false}
2,5	{"[number] value":"+2,5","[number] culture":"en-US","[number] valide":false}
+2.5	{"[number] value":"+2.5","[number] culture":"en-US","[number] valide":true} {"[range] value":"+2.5","[range] param":{"min":"2.3","max":"3.4"}} {"[range] valide":true}

6.3.19 Validateur [custom3]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- double3 : required, number, custom3 -->
2.  <tr>
3.     <td class="col1">double3 : required, number, custom3</td>
4.     <td class="col2">
5.        <input type="text" th:field="*{double3}" th:value="*{double3}" data-val="true"
       th:attr="data-val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-custom3=$
    {custom3.message},data-val-custom3-field=${custom3.otherFieldName},data-val-custom3-max=$
    {custom3.max},data-val-custom3-min=${custom3.min}" />
6.     </td>
7.     <td class="col3">
8.        <span class="field-validation-valid" data-valmsg-for="double3" data-valmsg-replace="true"></span>
9.     </td>
10.    <td class="col4">
11.       <span th:if="${#fields.hasErrors('double3')}" th:errors="*{double3}" class="error">Donnée
    erronée</span>
12.    </td>
13. </tr>
```

Ces lignes concernent le champ [*double3*] du formulaire [Form01] :

```
@NotNull
private Double double3;
```

On veut étudier ici un validateur qui valide non plus une valeur saisie mais une relation entre deux valeurs saisies. Ici, on veut que [double1+double3] soit dans l'intervalle [10,13].

La ligne 5 génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-custom3-min="10.0" data-val-number="Invalid format"
```

```
  data-val-custom3="[double3+double1] must be in [10,13]" data-val-custom3-max="13.0" data-val-custom3-field="double1"  data-val-required="Field is required" value="" id="double3" name="double3" />
```

Cette ligne introduit le nouveau validateur [custom3] déclaré par l'attribut [data-val-custom3="[double3+double1] must be in [10,13]"]. Ce validateur a les paramètres suivants :

- [field] déclaré par l'attribut [data-val-custom3-field="double1"]. Ce paramètre désigne le champ dont la valeur participe au calcul de validité de [double3] ;
- [min] déclaré par l'attribut [data-val-custom3-min="10.0"]. Ce paramètre est le min de l'intervalle [min, max] dans laquelle doit se trouver [double1+double3] ;
- [max] déclaré par l'attribut [data-val-custom3-max="13.0"]. Ce paramètre est le max de l'intervalle [min, max] dans laquelle doit se trouver [double1+double3] ;

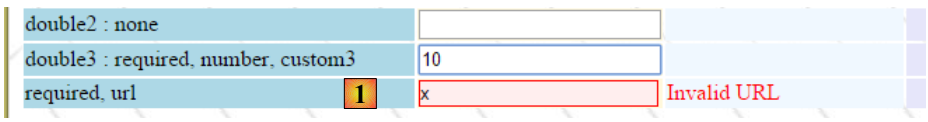
Ce validateur est géré de la façon suivante dans [client-validation.js] :

```
1.  // -------------- custom3 utilisé conjointement avec [number]
2.  logs.custom3 = true;
3.  $.validator.addMethod("custom3", function(value1, element, param) {
4.      // seconde valeur
5.      var value2 = $("#" + param.field).val();
6.      // logs
7.      if (logs.custom3) {
8.          console.log(jSON.stringify({
9.              "[custom3] value1" : value1,
10.             "[custom3] param" : param,
11.             "[custom3] value2" : value2
12.         }))
13.     }
14.     // première valeur
15.     var valeur1 = Globalize.parseFloat(value1);
16.     if (isNaN(valeur1)) {
17.         // on laisse le validateur [number] faire le travail
18.         if (logs.custom3) {
19.             console.log(jSON.stringify({
20.                 "[custom3] valide" : true
21.             }))
22.         }
23.         return true;
24.     }
25.     // seconde valeur
26.     var valeur2 = Globalize.parseFloat(value2);
27.     if (isNaN(valeur2)) {
28.         // on ne peut faire le calcul de validité
29.         if (logs.custom3) {
30.             console.log(jSON.stringify({
31.                 "[custom3] valide" : false
32.             }))
33.         }
34.         return false;
35.     }
36.     // calcul de validité
37.     var min = Globalize.parseFloat(param.min);
38.     var max = Globalize.parseFloat(param.max);
39.     var somme = valeur1 + valeur2;
40.     var valide = somme >= min && somme <= max;
41.     // logs
42.     if (logs.custom3) {
43.         console.log(jSON.stringify({
44.             "[custom3] valide" : valide
45.         }))
46.     }
47.     // résultat
48.     return valide;
49. });
50.
51. $.validator.unobtrusive.adapters.add("custom3", [ "field", "max", "min" ], function(options) {
52.     options.rules["custom3"] = options.params;
53.     options.messages["custom3"] = options.message.replace("'", "'");
54. });
```

Voici quelques exemples de logs :

Valeurs saisies [double1,double3]	logs
[x,1]	{"[custom3] value1":"1","[custom3] param": {"field":"double1","max":"13.0","min":"10.0"},"[custom3] value2":"x"} {"[custom3] valide":false}
[1,x]	{"[number] value":"x","[number] culture":"en-US","[number] valide":false}
[1,20]	{"[custom3] value1":"20","[custom3] param": {"field":"double1","max":"13.0","min":"10.0"},"[custom3] value2":"1"} {"[custom3] valide":false}
[1,10]	{"[number] value":"10","[number] culture":"en-US","[number] valide":true} {"[custom3] value1":"10","[custom3] param": {"field":"double1","max":"13.0","min":"10.0"},"[custom3] value2":"1"} {"[custom3] valide":true}

6.3.20 Validateur [url]



La ligne [1] est générée par la séquence suivante de la vue [vue-01.xml] :

```
1.  <!-- required, url -->
2.  <tr>
3.      <td class="col1">required, url</td>
4.      <td class="col2">
5.          <input type="text" th:field="*{url}" th:value="*{url}" data-val="true"
        th:attr="data-val-required=#{NotNull},data-val-url=#{URL.form01.url}" />
6.      </td>
7.      <td class="col3">
8.          <span class="field-validation-valid" data-valmsg-for="url" data-valmsg-replace="true"></span>
9.      </td>
10.     <td class="col4">
11.         <span th:if="${#fields.hasErrors('url')}" th:errors="*{url}" class="error">Donnée erronée</span>
12.     </td>
13. </tr>
```

Ces lignes concernent le champ [*url*] du formulaire [Form01] :

```
    @URL
    @NotBlank
    private String url;
```

La ligne 5 génère la ligne HTML suivante :

```
<input type="text" data-val="true" data-val-url="Invalid URL" data-val-required="Field is
required" value="" id="url" name="url" />
```

Elle introduit la validateur [url] avec l'attribut [data-val-url]. Ce validateur est prédéfini dans la bibliothèque jQuery de validation. Il n'y a rien à ajouter dans [client-validation.js].

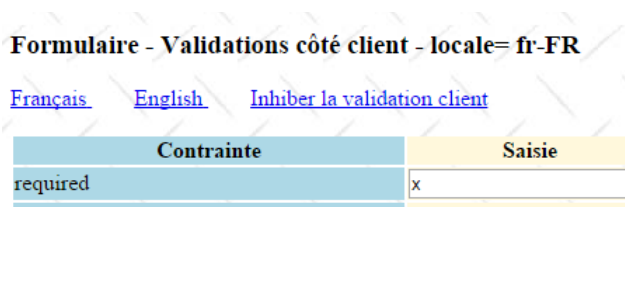
6.3.21 Activation / Désactivation de la validation côté client

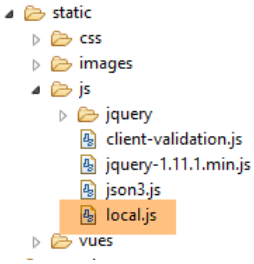
Tant que la validation côté client est active, on ne voit jamais la validation côté serveur car les valeurs postées n'arrivent au serveur que si elles ont été déclarées valides côté client. Pour voir la validation côté serveur oeuvrer il faut désactiver la validation côté client. La vue [vue-01.xml] offre deux liens pour gérer cette activation / désactivation :

```
1.  <a id="clientValidationTrue" href="javascript:setClientValidation(true)">
2.      <span style="margin-left:30px" th:text="#{client.validation.true}"></span>
```

```
3.  </a>
4.  <a id="clientValidationFalse" href="javascript:setClientValidation(false)">
5.     <span style="margin-left:30px" th:text="#{client.validation.false}"></span>
6.  </a>
```

Ces deux liens ne sont pas visibles en même temps :





La traduction HTML de ces liens est la suivante :

```
<a id="clientValidationTrue" href="javascript:setClientValidation(true)">
   <span style="margin-left:30px">Activer la validation client</span>
</a>
<a id="clientValidationFalse" href="javascript:setClientValidation(false)">
   <span style="margin-left:30px">Inhiber la validation client</span>
</a>
```

Le script jS [setClientValidation] est défini dans le fichier [local.js] (cf ci-dessus). Dans la fonction [$(document).ready] de ce fichier, les liens de validation sont exploités :

```
1.  // document ready
2.  $(document).ready(function() {
3.     // références globales
4.  ...
5.     activateValidationTrue = $("#clientValidationTrue");
6.     activateValidationFalse = $("#clientValidationFalse");
7.     clientValidation = $("#clientValidation");
8.  ...
9.     // liens de validation
10.    // clientValidation est un champ caché positionné par le serveur
11.    var validate = clientValidation.val();
12.    setClientValidation2(validate === "true");
13. });
```

- ligne 5 : une référence sur le lien d'activation de la validation côté client ;
- ligne 6 : une référence sur le lien de désactivation de la validation côté client ;
- ligne 7 : une référence sur un champ caché du formulaire qui mémorise le dernier état de l'activation sous la forme d'un booléen [true : validation client activée, false : validation client désactivée]. Ce champ se trouve dans la vue [vue-01.xml] sous la forme suivante :

```
<input type="hidden" th:field="*{clientValidation}" th:value="*{clientValidation}" value="true" />
```

et correspond au champ [clientValidation] du formulaire [Form01] :

```
1.  // validation client
2.  private boolean clientValidation = true;
```

- ligne 11 : on récupère la valeur du champ caché ;
- ligne 12 : on appelle la fonction [setClientValidation2] suivante :

```
1. function setClientValidation2(activate) {
2.     // liens
3.     if (activate) {
4.       // la validation client est active
5.       activateValidationTrue.hide();
6.       activateValidationFalse.show();
```

```
7.      // on parse les validateurs du formulaire
8.      $.validator.unobtrusive.parse(formulaire);
9.    } else {
10.     // la validation client est inactive
11.     activateValidationFalse.hide();
12.     activateValidationTrue.show();
13.     // on désactive les validateurs du formulaire
14.     formulaire.data('validator', null);
15.   }
16. }
```

- ligne 1 : le paramètre [activate] vaut [true] s'il faut activer la validation côté client, false sinon ;
- lignes 5-6 : le lien de désactivation est montré, le lien d'activation caché ;
- ligne 8 : pour que la validation côté client soit fonctionnelle, il faut parser (analyser) le document à la recherche de validateurs [data-val-X]. Le paramètre de la fonction [$.validator.unobtrusive.parse] est l'identifiant jS du formulaire à parser ;
- lignes 11-12 : le lien de d'activation est montré, le lien de désactivation caché ;
- lignes 14 : les validateurs du formulaire sont désactivés. A partir de maintenant, c'est comme s'il n'y avait pas de validateurs jS dans le formulaire ;

A quoi sert cette fonction [`setClientValidation2`] ? Elle sert à gérer les POST. Comme le champ [clientValidation] est un champ caché, il est posté et revient avec le formulaire renvoyé par le serveur. On se sert alors de sa valeur pour remettre la validation côté client comme elle était avant le POST. En effet, il n'y a pas de mémoire jS entre les requêtes. Il faut donc que le serveur transmette dans la nouvelle vue les informations qui permettent d'initialiser le jS de celle-ci. Cela se fait habituellement dans la fonction [$(document).ready].

Revenons à la fonction [setClientValidation] qui gère le clic sur les liens d'activation / désactivation de la validation côté client :

```
1.  // validation côté client
2.  function setClientValidation(activate) {
3.     // on gère l'activation / désactivation de la validation client
4.     setClientValidation2(activate);
5.     // on mémorise le choix de l'utilisateur dans le champ caché
6.     clientValidation.val(activate ? "true" : "false");
7.     // ajustements supplémentaires
8.     if (activate) {
9.        // la validation client est active
10.       // on efface tous les messages d'erreur du serveur
11.       clearServerErrors();
12.       // on valide le formulaire
13.       formulaire.validate().form();
14.    } else {
15.       // la validation client est inactive
16.       // on efface tous les messages d'erreur du client
17.       clearClientErrors();
18.    }
19. }
```

- ligne 4 : on utilise la fonction [setClientValidation2] que nous venons de voir ;
- ligne 6 : on mémorise le choix de l'utilisateur dans le champ caché pour le récupérer au retour du prochain POST ;
- ligne 11 : si la validation client est active, on efface les messages d'erreur de la colonne [serveur] de la vue. Nous avons décrit la fonction [clearServerErrors] page 229 ;
- ligne 13 : les validateurs jS sont exécutés pour faire apparaître d'éventuels messages d'erreur dans la colonne [client] de la vue ;
- ligne 17 : si la validation client est désactivée alors on efface les messages d'erreur de la colonne [client] de la vue. Examinons dans la console de développement de Chrome le code HTML d'un élément erroné :

```
1.  <td class="col2">
2.     <input type="text" data-val="true" data-val-int="Format invalide" data-val-max-value="100" data-val-
    required="Le champ est obligatoire" data-val-max="La valeur doit être inférieure ou égale à 100" value=""
    id="intMax100" name="intMax100" aria-required="true"           class="input-validation-error" aria-
    describedby="intMax100-error">
3.  </td>
4.  <td class="col3">
5.     <span class="field-validation-error" data-valmsg-for="intMax100" data-valmsg-replace="true">
6.        <span id="intMax100-error" class="">Le champ est obligatoire</span>
7.     </span>
8.  </td>
```

- ligne 2, on voit que dans la colonne 2 du tableau, l'élément erroné a le style [class="input-validation-error"] ;
- ligne 5, on voit que dans la colonne 3 du tableau, le message d'erreur a le style [class="field-validation-error"] ;

C'est vrai pour tous les éléments erronés. On utilise ces deux information dans la fonction [clearClientErrors] suivante :

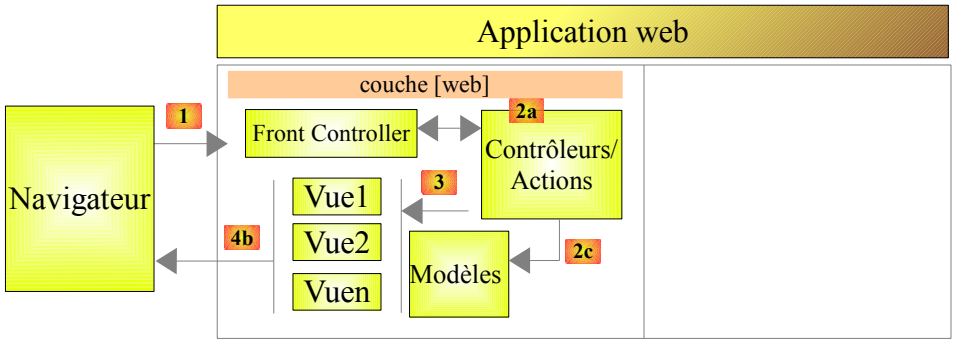
```
1.  // clear client errors
2.  function clearClientErrors() {
3.     // on efface les msg d'erreur du client
4.     $(".field-validation-error").each(function(index) {
5.        $(this).text("");
6.     });
7.     // on change la classe CSS des saisies erronées
8.     $(".input-validation-error").each(function(index) {
9.        $(this).removeClass("input-validation-error");
10.    });
11. }
```

- lignes 4-6 : on recherche tous les éléments du DOM ayant la classe [field-validation-error] et on efface le texte qu'ils affichent. C'est ainsi que les messages d'erreur sont effacés ;
- lignes 8-10 : on recherche tous les éléments du DOM ayant la classe [input-validation-error] et on leur enlève cette classe. Ainsi l'élément erroné qui avait été coloré en rouge retrouve son style primitif ;

7 Ajaxification d'une application Spring MVC

7.1 La place d'AJAX dans une application web

Pour l'instant, les exemples d'apprentissage étudiés avaient l'architecture suivante :



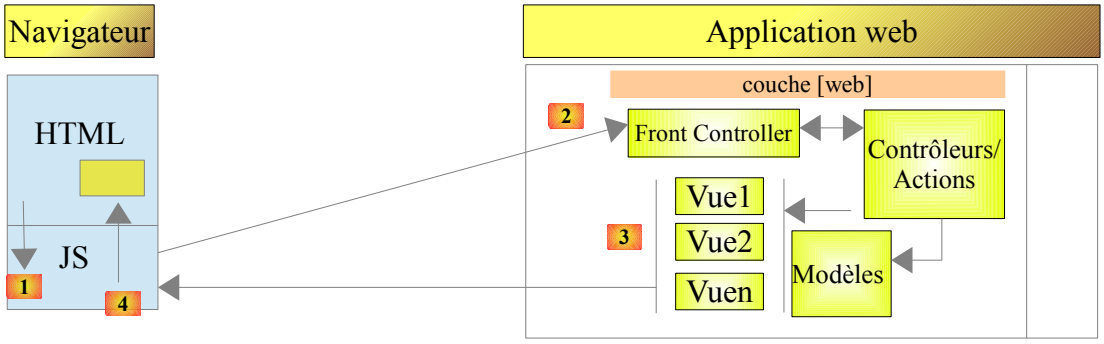
Pour passer d'une vue [Vue1] à une vue [Vue2], le navigateur :

- émet une requête vers l'application web ;
- reçoit la vue [Vue2] et l'affiche à la place de la vue [Vue1].

C'est le schéma classique :

- demande du navigateur ;
- élaboration d'une vue en réponse au client par le serveur web ;
- affichage de cette nouvelle vue par le navigateur.

Il existe depuis quelques années un autre mode d'interaction entre le navigateur et le serveur web : **AJAX** (**A**synchronous **J**avascript **A**nd **X**ml). Il s'agit en fait d'interactions entre la vue affichée par le navigateur et le serveur web. Le navigateur continue à faire ce qu'il sait faire, afficher une vue HTML mais il est désormais manipulé par du Javascript embarqué dans la vue HTML affichée. Le schéma est le suivant :



- en [1], un événement se produit dans la page affichée dans le navigateur (clic sur un bouton, changement d'un texte, ...). Cet événement est intercepté par du Javascript (jS) embarqué dans la page ;
- en [2], le code Javascript fait une requête HTTP comme l'aurait fait le navigateur. La requête est **asynchrone** : l'utilisateur peut continuer à interagir avec la page sans être bloqué par l'attente de la réponse à la requête HTTP. La requête suit le processus classique de traitement. Rien (ou peu) ne la distingue d'une requête classique ;
- en [3], une réponse est envoyée au client jS. Plutôt qu'une vue HTML complète, c'est plutôt une vue HTML partielle, un flux XML ou jSON (**J**ava**S**cript **O**bject **N**otation) qui est envoyé ;
- en [4], le Javascript récupère cette réponse et l'utilise pour mettre à jour une région de la page HTML affichée.

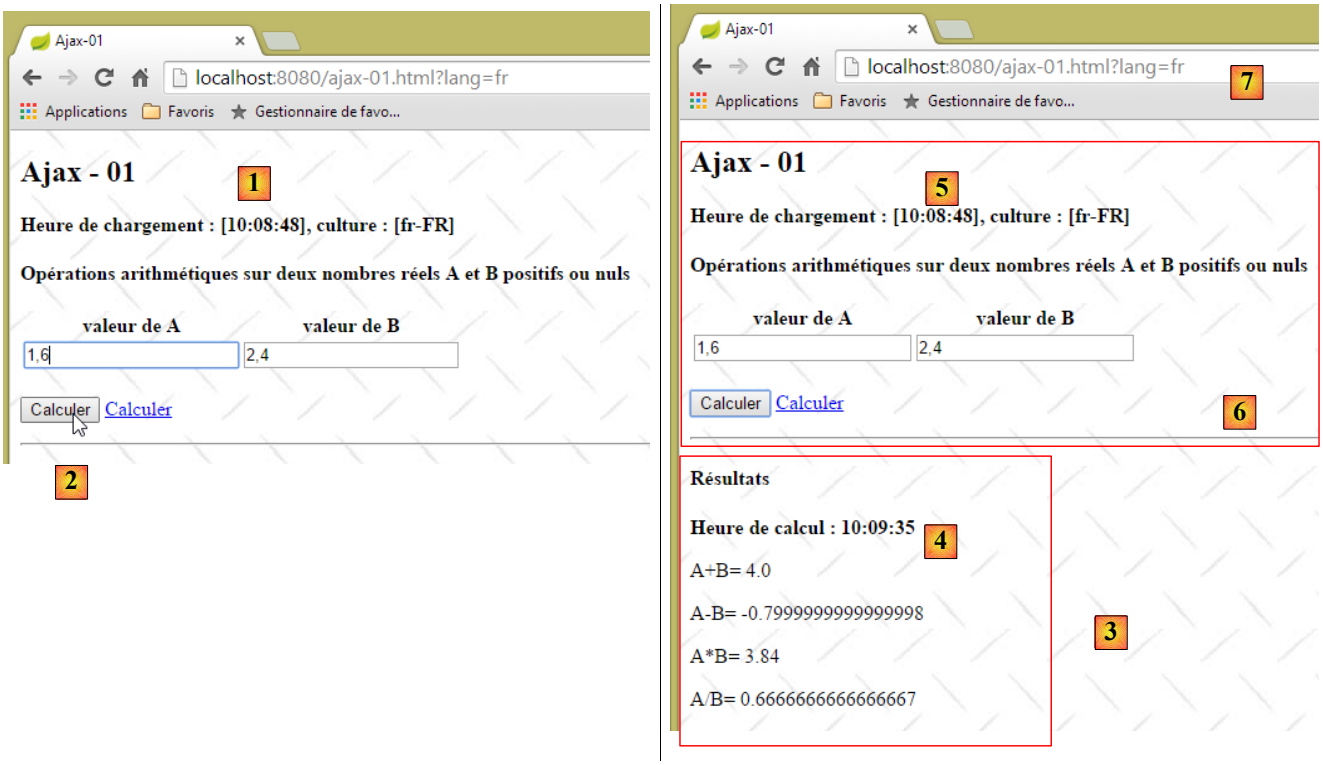
Pour l'utilisateur, il y a changement de vue car ce qu'il voit a changé. Il n'y a cependant pas rechargement total d'une page mais simplement modification partielle de la page affichée. Cela contribue à donner de la fluidité et de l'interactivité à la page : parce qu'il n'y a pas de rechargement total de la page, on peut se permettre de gérer des événements qu'auparavant on ne gérait pas. Par exemple, proposer à l'utilisateur une liste d'options au fur et à mesure qu'il saisit des caractères dans une boîte de saisie. A chaque

nouveau caractère tapé, une requête AJAX est faite vers le serveur qui renvoie alors d'autres propositions. Sans Ajax, ce genre d'aide à la saisie était auparavant impossible. On ne pouvait pas recharger une nouvelle page à chaque caractère tapé.

7.2 Mise à jour d'une page avec un flux HTML

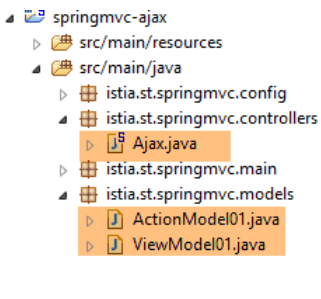
7.2.1 Les vues

On se propose d'étudier l'application suivante :



- en [1], l'heure de chargement de la page ;
- en [2], on fait les quatre opérations arithmétiques sur deux nombres réels A et B ;
- en [3], la réponse du serveur vient s'inscrire dans une région de la page ;
- en [4], l'heure du calcul. Celle-ci est différente de l'heure de chargement de la page [5]. Cette dernière est égale à [1] montrant que la région [6] n'a pas été rechargée. Par ailleurs l'URL [7] de la page n'a pas changé.

7.2.2 L'action [/ajax-01]



Le contrôleur [Ajax.java] définit l'action [/ajax-01] suivante :

```
1.      @RequestMapping(value = "/ajax-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-
    8")
2.      public String ajax01(Locale locale, Model modèle, HttpSession session, String tempo) {
3.          // tempo valide ?
```

```
4.         if (tempo != null) {
5.            boolean valide = false;
6.            int valueTempo = 0;
7.            try {
8.               valueTempo = Integer.parseInt(tempo);
9.               valide = valueTempo >= 0;
10.           } catch (NumberFormatException e) {
11.
12.           }
13.           if (valide) {
14.              session.setAttribute("tempo", new Integer(valueTempo));
15.           }
16.        }
17.        // on prépare le modèle de la vue [vue-01]
18.        ...
19. }
```

- ligne 2 : l'action [/ajax-01] n'accepte qu'un seul paramètre [tempo]. C'est la durée en millisecondes pendant laquelle le serveur devra attendre avant d'envoyer les résultats des opérations arithmétiques ;
- ligne 4 : le paramètre [tempo] est facultatif ;
- lignes 5-12 : on vérifie que la valeur du paramètre [tempo] est acceptable ;
- lignes 13-15 : si c'est le cas, la valeur de la temporisation est mise en session. Cela veut dire qu'elle sera en vigueur tant qu'on ne la changera pas ;

Le code de l'action [/ajax-01] se poursuit ainsi :

```
1.     @RequestMapping(value = "/ajax-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-
   8")
2.     public String ajax01(Locale locale, Model modèle, HttpSession session, String tempo) {
3.        // tempo valide ?
4.  ...
5.        // on prépare le modèle de la vue [vue-01]
6.        modèle.addAttribute("actionModel01", new ActionModel01());
7.  ...
8.        // vue
9.        return "vue-01";
10. }
```

La classe [ActionModel01] sert principalement à encapsuler les valeurs postées par l'action [/ajax-01]. Ici, il n'y a rien de posté. On crée une classe vide qu'on met dans le modèle car la vue [vue-01.xml] l'utilise. La classe [ActionModel01] est la suivante :

```
1.  package istia.st.springmvc.models;
2.
3.  import javax.validation.constraints.DecimalMin;
4.  import javax.validation.constraints.NotNull;
5.
6.  public class ActionModel01 {
7.
8.     // données postées
9.     @NotNull
10.    @DecimalMin(value = "0.0")
11.    private Double a;
12.
13.    @NotNull
14.    @DecimalMin(value = "0.0")
15.    private Double b;
16.
17.    // getters et setters
18.    ...
19. }
```

- lignes 11 et 15 : deux réels [a,b] qui vont être postés par un formulaire ;

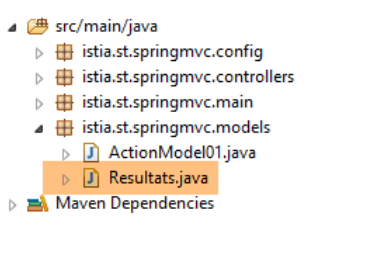
Revenons au code de l'action :

```
1.     @RequestMapping(value = "/ajax-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-
   8")
2.     public String ajax01(Locale locale, Model modèle, HttpSession session, String tempo) {
3.  ...
4.        // on prépare le modèle de la vue [vue-01]
5.        modèle.addAttribute("actionModel01", new ActionModel01());
```

```
6.          Resultats résultats = new Resultats();
7.          modèle.addAttribute("resultats", résultats);
8.    ...
9.          // vue
10.         return "vue-01";
11.   }
```

- lignes 6-7 : on met une instance de type [Resultats] dans le modèle ;

Le type [Resultats] mis dans le modèle est le suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  public class Resultats {
4.
5.      // données
6.      private String aplusb;
7.      private String amoinsb;
8.      private String amultiplieparb;
9.      private String adiviseparb;
10.     private String heureGet;
11.     private String heurePost;
12.     private String erreur;
13.     private String vue;
14.     private String culture;
15.
16.     // getters et setters
17.     ...
18. }
```

- lignes 6-9 : le résultat des quatre opération arithmétiques sur les nombres [a,b] ;
- ligne 10 : l'heure du chargement initial de la page ;
- ligne 11 : l'heure d'exécution des quatre opérations arithmétiques ;
- ligne 12 : un éventuel message d'erreur ;
- ligne 13 : l'éventuelle vue qui doit être affichée ;
- ligne 14 : la culture de la vue, [fr-FR] ou [en-US] ;

Le code de l'action [/ajax-01] se poursuit ainsi :

```
1.      @RequestMapping(value = "/ajax-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.      public String ajax01(ActionModel01 formulaire, Locale locale, Model modèle, HttpSession session) {
3.          ...
4.          // locale
5.          setLocale(locale, modèle, résultats);
6.  ...
7.  }
```

- ligne 5 : la méthode [setLocale] sert à mettre dans le modèle de la vue la culture à utiliser, [fr-FR] ou [en-US]. Cette culture est à destination du Javascript embarqué dans la vue ;

La méthode [setLocale] est la suivante :

```
1.      private void setLocale(Locale locale, Model modèle, Resultats résultats) {
2.          // on ne gère que les locales fr-FR, en-US
3.          String language = locale.getLanguage();
4.          String country = null;
5.          switch (language) {
6.          case "fr":
```

```
7.            country = "FR";
8.            break;
9.         default:
10.           language = "en";
11.           country = "US";
12.           break;
13.        }
14.        // culture
15.        résultats.setCulture(String.format("%s-%s", language, country));
16. }
```

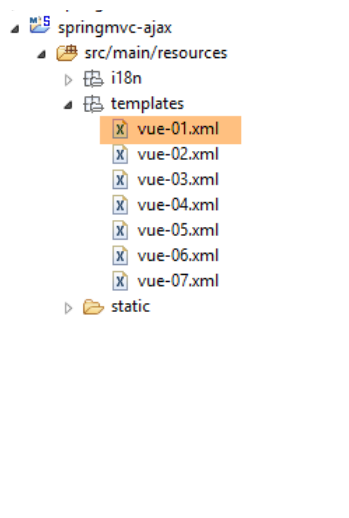
Dans le modèle on aura la chaîne [${resultats.culture}] égale à 'fr-FR' ou 'en-US'.

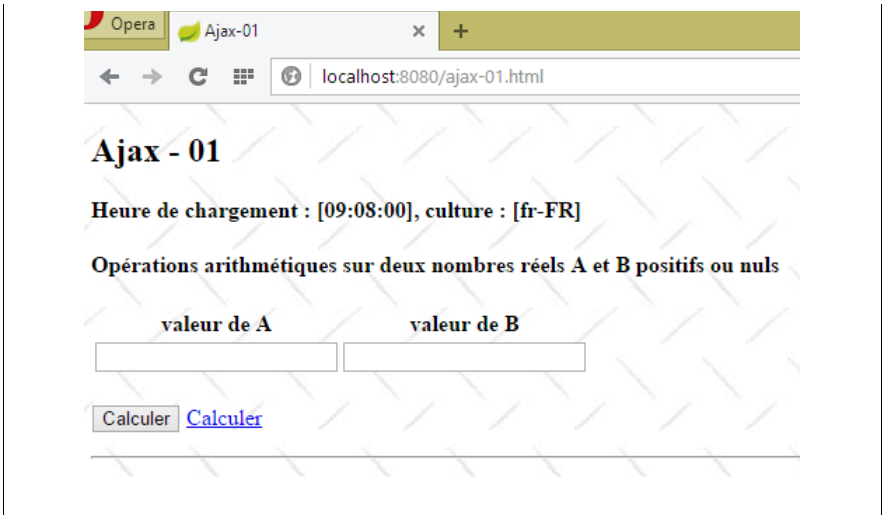
Revenons à l'action [/ajax-01] :

```
1.  @RequestMapping(value = "/ajax-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.     public String ajax01(ActionModel01 formulaire, Locale locale, Model modèle, HttpSession session) {
3.  ...
4.        // locale
5.        setLocale(locale, modèle, résultats);
6.        // heure
7.        résultats.setHeureGet(new SimpleDateFormat("hh:mm:ss").format(new Date()));
8.        // vue
9.        return "vue-01";
10.    }
```

- ligne 7 : on met l'heure du GET dans le modèle ;
- lignes 9 : on affiche la vue [vue-01.xml] :

7.2.3 La vue [vue-01.xml]





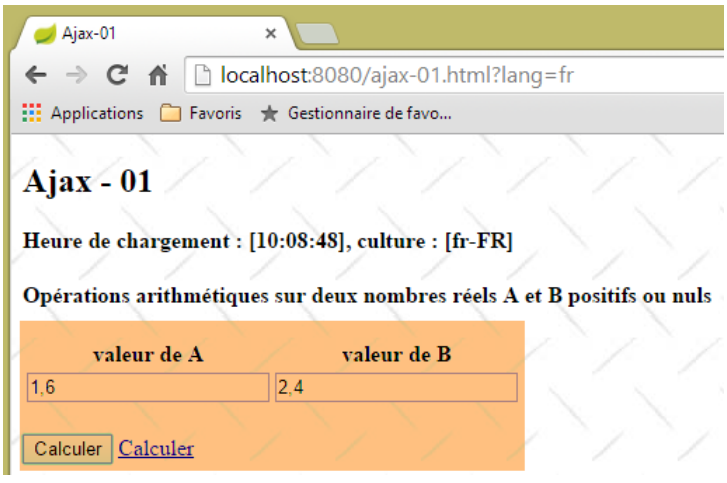
La vue [vue-01.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>Ajax-01</title>
6.        <link rel="stylesheet" href="/css/ajax01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
8.        <script type="text/javascript" src="/js/jquery/jquery.validate.min.js"></script>
9.        <script type="text/javascript" src="/js/jquery/jquery.validate.unobtrusive.min.js"></script>
10.       <script type="text/javascript" src="/js/jquery/globalize/globalize.js"></script>
11.       <script type="text/javascript" src="/js/jquery/globalize/cultures/globalize.culture.fr-
    FR.js"></script>
12.       <script type="text/javascript" src="/js/jquery/globalize/cultures/globalize.culture.en-
    US.js"></script>
```

```
13.       <script type="text/javascript" src="/js/jquery/jquery.unobtrusive-ajax.js"></script>
14.       <script type="text/javascript" src="/js/json3.js"></script>
15.       <script type="text/javascript" src="/js/client-validation.js"></script>
16.       <script type="text/javascript" src="/js/local1.js"></script>
17.       <script th:inline="javascript">
18.          /*<![CDATA[*/
19.                var culture = [[${resultats.culture}]];
20.                Globalize.culture(culture);
21.                /*]]>*/
22.       </script>
23.    </head>
24.    <body>
25.       <h2>Ajax - 01</h2>
26.       <p>
27.          <strong th:text="#{labelHeureGetCulture(${resultats.heureGet},${resultats.culture})}">
28.             Heure de chargement :
29.          </strong>
30.       </p>
31.       <h4>
32.          <p th:text="#{titre.part1}">
33.             Opérations arithmétiques sur deux nombres réels A et B positifs ou nuls
34.          </p>
35.       </h4>
36.       <form id="formulaire" name="formulaire" ... ">
37. ...
38.       </form>
39.       <hr />
40.       <div id="resultats" />
41.    </body>
42. </html>
```

- lignes 7-12 : les bibliothèques jQuery de validation et d'internationalisation (cultures) ;
- ligne 15 : la bibliothèque [client-validation] construite au paragraphe 6.3, page 212 ;
- ligne 14 : la bibliothèque jSON utilisée par la bibliothèque [client-validation]. Elle est facultative si les logs de validation ont été désactivés ;
- ligne 13 : la bibliothèque [Unobtrusive Ajax] de Microsoft. Cette bibliothèque permet parfois de s'affranchir d'écrire du Javascript ;
- ligne 16 : un fichier jS pour nos propres besoins ;
- lignes 17-22 : pour gérer côté client les cultures [fr-FR] et [en-US]. Nous avons déjà rencontré ce code ;
- ligne 27 : un message paramétré. Nous les avons étudiés au paragraphe 5.18, page 190 ;
- lignes 36-38 : le formulaire sur lequel nous allons revenir ;
- ligne 40 : la zone du document dans lequel le Javascript placera la réponse du serveur ;

7.2.4 Le formulaire



Dans la vue [vue-01.xml], le formulaire est le suivant :

```
<form id="formulaire" name="formulaire" th:action="@{/ajax-02.html}" method="post" th:object="${actionModel01}" th:attr="data-ajax='true',data-ajax-loading='#loading',data-ajax-loading-duration='0',data-ajax-method='post',data-ajax-mode='replace',data-ajax-update='#resultats', data-ajax-begin='beforeSend',data-ajax-complete='afterComplete' ">
   <table>
      <thead>
         <tr>
            <th>
               <span th:text="#{valeur.a}"></span>
            </th>
            <th>
               <span th:text="#{valeur.b}"></span>
            </th>
         </tr>
      </thead>
      <tbody>
         <tr>
            <td>
               <input type="text" th:field="*{a}" th:value="*{a}" data-val="true"
                  th:attr="data-val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-min=#{actionModel01.a.min},data-val-min-value=#{actionModel01.a.min.value}" />
            </td>
            <td>
               <input type="text" th:field="*{b}" th:value="*{b}" data-val="true"
                  th:attr="data-val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-min=#{actionModel01.b.min},data-val-min-value=#{actionModel01.b.min.value}" />
            </td>
         </tr>
         <tr>
            <td>
               <span class="field-validation-valid" data-valmsg-for="a" data-valmsg-replace="true"></span>
               <span th:if="${#fields.hasErrors('a')}" th:errors="*{a}" class="error">Donnée
                  erronée
               </span>
            </td>
            <td>
               <span class="field-validation-valid" data-valmsg-for="b" data-valmsg-replace="true"></span>
               <span th:if="${#fields.hasErrors('b')}" th:errors="*{b}" class="error">Donnée
                  erronée
               </span>
            </td>
         </tr>
      </tbody>
   </table>
   <p>
      <input type="submit" th:value="#{action.calculer}" value="Calculer"></input>
      <img id="loading" style="display: none" src="/images/loading.gif" />
      <a href="javascript:postForm()" th:text="#{action.calculer}">Calculer</a>
   </p>
</form>
```

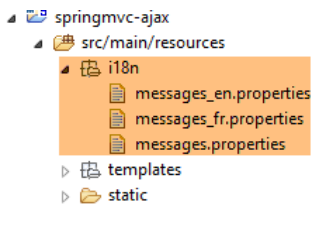
qui produit le HTML suivant :

```
<form id="formulaire" name="formulaire" method="post" data-ajax-update="#resultats" data-ajax-complete="afterComplete"  data-ajax-begin="beforeSend" data-ajax-loading-duration="0" data-ajax-mode="replace" data-ajax="true" data-ajax-method="post" data-ajax-loading="#loading" action="/ajax-02.html">
   <table>
      <thead>
         <tr>
            <th>
               <span>valeur de A</span>
            </th>
            <th>
               <span>valeur de B</span>
            </th>
         </tr>
      </thead>
      <tbody>
         <tr>
            <td>
```

```
16.                 <input type="text" data-val="true" data-val-min="Le nombre doit être supérieur ou égal à
    0" data-val-number="Format invalide" data-val-min-value="0" data-val-required="Le champ est obligatoire"
    value="" id="a" name="a" />
17.             </td>
18.             <td>
19.                 <input type="text" data-val="true" data-val-min="Le nombre doit être supérieur ou égal à
    0" data-val-number="Format invalide" data-val-min-value="0" data-val-required="Le champ est obligatoire"
    value="" id="b" name="b" />
20.             </td>
21.          </tr>
22.          <tr>
23.             <td>
24.                 <span class="field-validation-valid" data-valmsg-for="a" data-valmsg-
    replace="true"></span>
25.
26.             </td>
27.             <td>
28.                 <span class="field-validation-valid" data-valmsg-for="b" data-valmsg-
    replace="true"></span>
29.
30.             </td>
31.          </tr>
32.       </tbody>
33.    </table>
34.    <p>
35.       <input type="submit" value="Calculer" />
36.       <img id="loading" style="display: none" src="/images/loading.gif" />
37.       <a href="javascript:postForm()">Calculer</a>
38.    </p>
39. </form>
```

- ligne 16 : au champ [a] sont associés les validateurs [required], [number] et [min] ;
- ligne 19 : idem pour le champ [b] ;

Les divers messages sont trouvés dans les fichiers [messages.properties] du projet :



[messages_fr.properties]

```
1.  NotNull=Le champ est obligatoire
2.  typeMismatch=Format invalide
3.  actionModel01.a.min=Le nombre doit être supérieur ou égal à 0
4.  DecimalMin.actionModel01.a=Le nombre doit être supérieur ou égal à 0
5.  DecimalMax.actionModel01.b=Le nombre doit être supérieur ou égal à 0
6.  actionModel01.b.min=Le nombre doit être supérieur ou égal à 0
7.  valeur.a=valeur de A
8.  valeur.b=valeur de B
9.  actionModel01.a.min.value=0
10. actionModel01.b.min.value=0
11. labelHeureCalcul=Heure de calcul :
12. LabelErreur=Une erreur s''est produite : [{0}]
13. labelAplusB=A+B=
14. labelAmoinsB=A-B=
15. labelAfoisB=A*B=
16. labelAdivB=A/B=
17. titre.part1=Opérations arithmétiques sur deux nombres réels A et B positifs ou nuls
18. labelHeureGetCulture=Heure de chargement : [{0}], culture : [{1}]
19. action.calculer=Calculer
20. erreur.aleatoire=erreur aléatoire
21. resultats=Résultats
22. resultats.erreur=Une erreur s''est produite : [{0}]
23. resultats.titre=Résultats
```

```
24. message.zone=Nombre d'accès :
```

[messages_en.properties]

```
1.  NotNull=Required field
2.  typeMismatch=Invalid format
3.  actionModel01.a.min=The number must be greater or equal to 0
4.  DecimalMin.actionModel01.a=The number must be greater or equal to 0
5.  DecimalMax.actionModel01.b=The number must be greater or equal to 0
6.  actionModel01.b.min=The number must be greater or equal to 0
7.  valeur.a=A value
8.  valeur.b=B value
9.  actionModel01.a.min.value=0
10. actionModel01.b.min.value=0
11. labelHeureCalcul=Computing hour:
12. LabelErreur=There was an error: [{0}]
13. labelAplusB=A+B=
14. labelAmoinsB=A-B=
15. labelAfoisB=A*B=
16. labelAdivB=A/B=
17. titre.part1=Arithmetic operations on two positive or equal to zero real numbers
18. labelHeureGetCulture=Loading hour: [{0}], culture: [{1}]
19. action.calculer=Calculate
20. erreur.aleatoire=randomly generated error
21. resultats=Results
22. resultats.erreur=Some error occurred : [{0}]
23. resultats.titre=Results
24. message.zone=Number of hits:
```

Maintenant, étudions les attributs de la balise [form] :

```
<form id="formulaire" name="formulaire" method="post" data-ajax-update="#resultats" data-ajax-
complete="afterComplete" data-ajax-begin="beforeSend" data-ajax-loading-duration="0" data-ajax-mode="replace"
data-ajax="true" data-ajax-method="post" data-ajax-loading="#loading" action="/ajax-02.html">
```

On reconnaît les attributs classiques de la balise [form] :

```
<form id="formulaire" name="formulaire" method="post" action="/ajax-02.html">
```

On peut noter tout de suite que si sur le navigateur qui affiche la page, le Javascript est désactivé, alors le formulaire sera posté à l'URL [*/ajax-02.html*]. Maintenant, analysons les autres attributs :

```
<form ... data-ajax-update="#resultats" data-ajax-complete="afterComplete" data-ajax-
begin="beforeSend" data-ajax-loading-duration="0" data-ajax-mode="replace" data-ajax="true"
data-ajax-method="post" data-ajax-loading="#loading">
```

Les attributs [data-ajax-xxx] sont gérés par la bibliothèque jS [*unobtrusive-ajax*] qui a été importée par la vue [vue-01.xml] :

```
<script type="text/javascript" src="/js/jquery/jquery.unobtrusive-ajax.js"></script>
```

Lorsque les attributs [data-ajax-xxx] sont présents, le [submit] du formulaire va être exécuté par un appel Ajax de la bibliothèque [*unobtrusive-ajax*]. La signification des paramètres est la suivante :

- [data-ajax=*"true"*] : c'est la présence de cet attribut qui fait que le [submit] du formulaire va être ajaxifié ;
- [data-ajax-method=*"post"*] : la méthode du [submit]. L'URL du post sera celle de l'attibut [action=*"/ajax-02.html"*] ;
- [data-ajax-loading=*"#loading"*] : l'id d'une zone à afficher en attendant la réponse du serveur. La zone identifiée par [loading] dans la vue [vue-01.xml] est la suivante :

  ```
  <img id="loading" style="display: none" src="/images/loading.gif" />
  ```

  C'est une image animée d'attente qui sera affichée tant que la réponse du serveur n'aura pas été reçue ;
- [data-ajax-loading-duration=*"0"*] : le temps d'attente en millisecondes avant que la zone [data-ajax-loading=*"#loading"*] soit affichée. Ici, elle sera affichée dès que l'attente commencera ;
- [data-ajax-begin=*"beforeSend"*] : la fonction jS à exécuter avant de faire le [submit] ;
- [data-ajax-complete=*"afterComplete"*] : la fonction jS à exécuter lorsque la réponse a été reçue ;
- [data-ajax-update=*"#resultats"*] : l'identifiant de la zone où le résultat envoyé par le serveur sera placé. La vue [vue-01.xml] possède la zone suivante :

```
<div id="resultats" />
```

- [data-ajax-mode="replace"] : le mode d'insertion du résultat dans la zone précédente. Le mode [replace] fera que le résultat 'écrasera' ce qu'il y avait avant dans la zone d'id [resultats] ;

Il faut noter que le [submit] Javascript n'aura lieu que si les validateurs ont déclaré valides les valeurs testées.

La bibliothèque jS [*unobtrusive-ajax*] a deux objectifs :
- faire en sorte que le formulaire s'adapte correctement aux deux possibilités : activation ou non du Javascript sur le navigateur ;
- éviter d'écrire du Javascript. Nous verrons qu'ici, cela n'a pas pu être évité.

7.2.5 L'action [/ajax-02]

Nous avons vu que les valeurs postées étaient envoyées à l'action [/ajax-02]. Celle-ci est la suivante :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.      public String ajax02(ActionModel01 formulaire, Locale locale, Model modèle, HttpSession session)
    throws InterruptedException {
3.          // tempo ?
4.          Integer tempo = (Integer) session.getAttribute("tempo");
5.          if (tempo != null && tempo > 0) {
6.              Thread.sleep(tempo);
7.          }
8.          // on prépare le modèle de la prochaine vue
9.          Resultats résultats = new Resultats();
10.         modèle.addAttribute("resultats", résultats);
11.         // on fixe la locale
12.         setLocale(locale, modèle, résultats);
13.         // heure
14.         résultats.setHeurePost(new SimpleDateFormat("hh:mm:ss").format(new Date()));
15.         ...
16. }
```

- nous allons simplifier dans un premier temps : nous supposons que le POST qui a lieu a bien été fait par le Javascript de la vue [vue-01.xml]. Nous reviendrons sur cette hypothèse un peu plus tard ;
- ligne 2 : les valeurs [a,b] postées sont mises dans le modèle [ActionModel01] ;
- lignes 4-7 : si l'utilisateur avait fixé une temporisation lors d'un précédent GET, celle-ci est récupérée dans la session et on fait la temporisation (ligne 6). Le but de celle-ci est de permettre à l'utilisateur de voir l'effet de l'attribut [data-ajax-loading="#loading"] dans le formulaire ;
- lignes 9-10 : on met un attribut [resultats] dans le modèle ;
- ligne 12 : on met la culture [fr-FR] ou [en-US] dans le modèle ;
- ligne 14 : on met l'heure du POST dans le modèle ;

Rappelons le type [Resultats] mis dans le modèle :

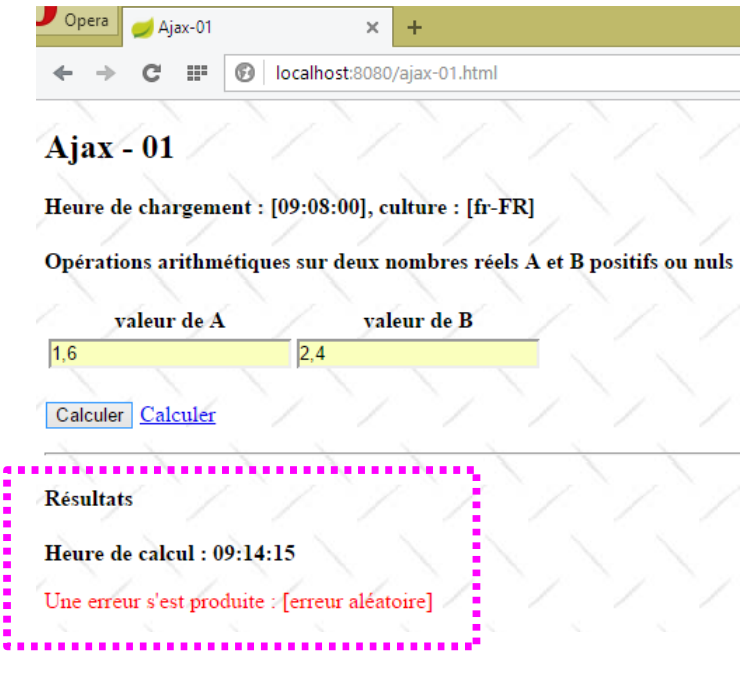
```
1.  public class Resultats {
2.
3.      // données
4.      private String aplusb;
5.      private String amoinsb;
6.      private String amultiplieparb;
7.      private String adiviseparb;
8.      private String heureGet;
9.      private String heurePost;
10.     private String erreur;
11.     private String vue;
12.     private String culture;
13.
14.     // getters et setters
15. ...
16. }
```

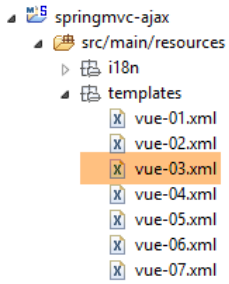
Le code de l'action [/ajax-02] se poursuit ainsi :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html;
    charset=UTF-8")
```

```
2.     public String ajax02(ActionModel01 formulaire, BindingResult result, Locale locale,
   Model modèle, HttpSession session) throws InterruptedException {
3.  ...
4.        résultats.setHeurePost(new SimpleDateFormat("hh:mm:ss").format(new Date()));
5.        // on génère une erreur une fois sur deux
6.        int val = new Random().nextInt(2);
7.        if (val == 0) {
8.           // on renvoie un message d'erreur
9.           résultats.setErreur("erreur.aleatoire");
10.          return "vue-03";
11.       }
12. ...
13.    }
```

- lignes 6-11 : pour l'exemple on montre comment renvoyer une page d'erreur au client jS. Une fois sur deux, on renvoie la vue [vue-03.xml] suivante :





On notera ligne 9, que ce n'est pas un message qu'on met dans le modèle, mais une clé de message :

[messages_fr.properties]

```
erreur.aleatoire=erreur aléatoire
```

[messages_fr.properties]

```
erreur.aleatoire=randomly generated error
```

Le code de la vue [vue-03.xml] est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h4>Résultats</h4>
5.        <p>
6.           <strong>
7.              <span th:text="#{labelHeureCalcul}">Heure de calcul :</span>
8.              <span id="heureCalcul" th:text="${resultats.heurePost}"></span>
9.           </strong>
10.       </p>
11.       <p style="color: red;">
12.          <span th:text="#{labelErreur(#{${resultats.erreur}})}">Une erreur s'est produite :</span>
```

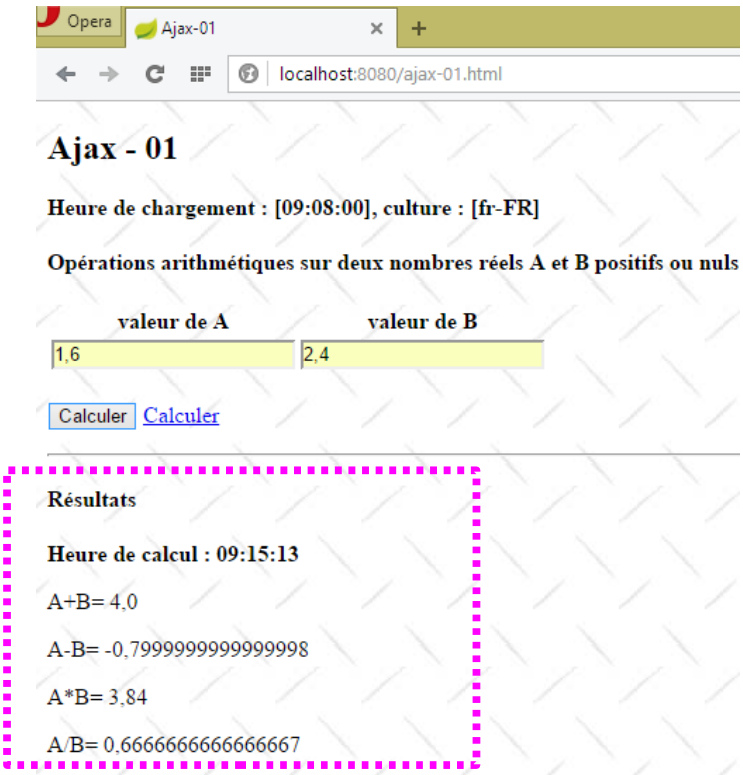
```
13.           <!-- <span id="erreur" th:text="${resultats.erreur}"></span> -->
14.       </p>
15.    </body>
16. </html>
17.
```

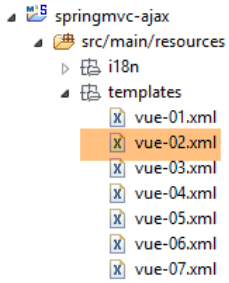
- ligne 12, on notera un message paramétré par une clé de message qui est elle-même calculée. Nous avons introduit cette notion au paragraphe 5.18, page 190.

Le code de l'action [/ajax-02] se poursuit ainsi :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String ajax02(ActionModel01 formulaire, BindingResult result, Locale locale, Model modèle,
       HttpSession session) throws InterruptedException {
3.  ...
4.        // on récupère les valeurs postées
5.        double a = formulaire.getA();
6.        double b = formulaire.getB();
7.        // on construit le modèle
8.        résultats.setAplusb(String.valueOf(a + b));
9.        résultats.setAmoinsb(String.valueOf(a - b));
10.       résultats.setAmultiplieparb(String.valueOf(a * b));
11.       try {
12.          résultats.setAdiviseparb(String.valueOf(a / b));
13.       } catch (RuntimeException e) {
14.          résultats.setAdiviseparb("NaN");
15.       }
16.       // on affiche la vue
17.       return "vue-02";
18.    }
```

- lignes 5-15 : les quatre opération arithmétiques sont faites sur les nombres [a,b] et encapsulées dans l'instance [Resultats] du modèle ;
- ligne 17 : on renvoie la vue [vue-02.xml] suivante :





La vue [vue-02.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h4>Résultats</h4>
5.        <p>
6.           <strong>
7.              <span th:text="#{labelHeureCalcul}">Heure de calcul :</span>
8.              <span id="heureCalcul" th:text="${resultats.heurePost}"></span>
9.           </strong>
10.       </p>
11.       <p>
12.          <span th:text="#{labelAplusB}">A+B=</span>
13.          <span id="aplusb" th:text="${resultats.aplusb}"></span>
14.       </p>
15.       <p>
16.          <span th:text="#{labelAmoinsB}">A-B=</span>
17.          <span id="amoinsb" th:text="${resultats.amoinsb}"></span>
18.       </p>
19.       <p>
20.          <span th:text="#{labelAfoisB}">A*B=</span>
21.          <span id="amultiplieparb" th:text="${resultats.amultiplieparb}"></span>
22.       </p>
23.       <p>
24.          <span th:text="#{labelAdivB}">A/B=</span>
25.          <span id="adiviseparb" th:text="${resultats.adiviseparb}"></span>
26.       </p>
27.    </body>
28. </html>
```

Que le résultat soit la vue [vue-02.xml] ou la vue [vue-03.xml], ce résultat HTML est placé dans la zone identifiée par [resultats] dans la vue [vue-01.xml], ceci à cause de l'attribut [data-ajax-update="*#resultats*"] du formulaire.

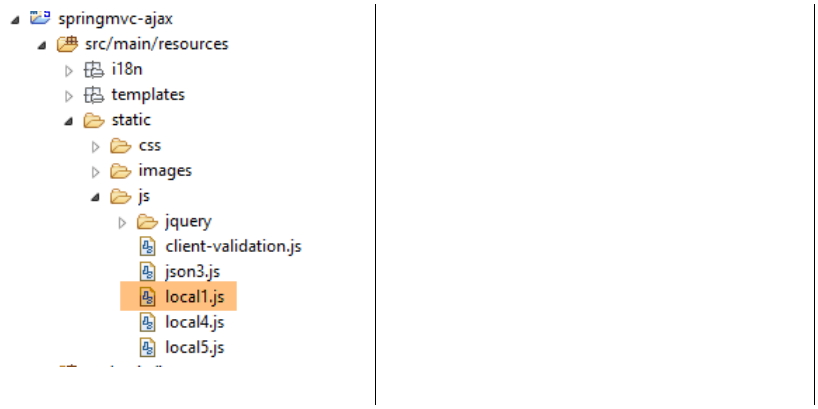
7.2.6 Le POST des valeurs saisies

On a une difficulté ici avec les valeurs postées. On travaille avec deux cultures [fr-FR] et [en-US] qui écrivent différemment les nombres réels. Nous nous étions attaqués à cette difficulté lorsqu'il avait fallu dans le pararaphe 6.3, page 212, poster des réels dans deux cultures différentes. Nous allons reprendre ici des outils utilisés alors. Mais on a une difficulté supplémentaire : on n'a pas accès à la méthode qui opère le POST des valeurs saisies. C'est la raison pour laquelle, nous avons ajouté les attributs suivants à la balise du formulaire :

- [data-ajax-begin="*beforeSend*"] : la fonction jS à exécuter avant de faire le [submit] ;
- [data-ajax-complete="*afterComplete*"] : la fonction jS à exécuter lorsque la réponse a été reçue ;

Nous n'avons pas accès à la fonction jS qui va poster les valeurs saisies, mais nous pouvons écrire deux fonctions jS :
- [beforeSend] : une fonction jS exécutée avant le POST ;
- [afterComplete] : une fonction jS exécutée à réception de la réponse au POST ;

Ces deux fonctions sont placées dans un fichier [local1.js] :



Le fichier [local1.js] initialise l'environnement jS de la vue [vue-01.xml] de la façon suivante :

```
1.  // données globales
2.  var loading;
3.  var formulaire;
4.  var résultats;
5.  var a, b;
6.
7.  // au chargement du document
8.  $(document).ready(function() {
9.     // on récupère les références des différents composants de la page
10.    loading = $("#loading");
11.    formulaire = $("#formulaire");
12.    resultats = $('#resultats');
13.    a = $("#a");
14.    b = $("#b");
15.    // on cache certains éléments
16.    loading.hide();
17.    // on parse les validateurs du formulaire
18.    $.validator.unobtrusive.parse(formulaire);
19.    // on gère deux locales [fr_FR, en_US]
20.    // les réels [a,b] sont envoyés par le serveur au format anglo-saxon
21.    // on les met au format français si nécessaire
22.    checkCulture(2);
23. });
```

- ligne 22 : la fonction [checkCulture] est présentée un peu plus loin ;

La fonction jS [beforeSend] sera la suivante :

```
1.  function beforeSend(jqXHR, settings) {
2.     // avant le POST
3.     // les nombres doivent être postés au format anglo-saxon
4.     var culture = Globalize.culture().name;
5.     if (culture === 'fr-FR') {
6.        checkCulture(1);
7.        settings.data = formulaire.serialize();
8.     }
9.  }
10.
11. function afterComplete(jqXHR, settings) {
12.    ...
13. }
14.
15. function checkCulture(mode) {
16.    if (mode == 1) {
17.       // on met les nombres [a,b] au format anglo-saxon
18.       var value1 = a.val().replace(",", ".");
19.       a.val(value1);
20.       var value2 = b.val().replace(",", ".");
21.       b.val(value2);
22.    }
23.    if (mode == 2) {
24. ...
25.    }
26. }
```

- lignes 4-6 : on vérifie si la culture de la vue est [fr-FR]. Dans ce cas, il faut changer les valeurs postées. En effet, si l'utilisateur a saisi [1,6], il faut poster la valeur [1.6] sinon la valeur [1,6] sera refusée côté serveur. Il suffit pour cela de changer la virgule des valeurs postées en point décimal (lignes 18-21) ;
- on ne peut s'en tenir là. En effet, lorsque la fonction [beforeSend] est appelée, la chaîne des valeurs postées [a=val1&b=valB] a déjà été construite. Il nous faut donc la modifier. Cela se fait à l'aide du second paramètre [settings] de la fonction ;
- ligne 7 : [settings.data] (*settings* est un paramètre de la fonction) représente la chaîne postée. On recrée cette chaîne avec l'expression [formulaire.serialize()]. Cette expression parcourt le formulaire à la recherche des valeurs à poster et construit la chaîne du POST. Elle va alors prendre les nouvelles valeurs de [a,b] avec des points décimaux ;

Si on ne fait rien de plus, le serveur va envoyer sa réponse qui va être correctement affichée. Seulement maintenant les valeurs de [a,b] sont avec le point décimal alors qu'on est toujours dans la culture [fr-FR]. Si donc l'utilisateur ne s'en aperçoit pas et reclique

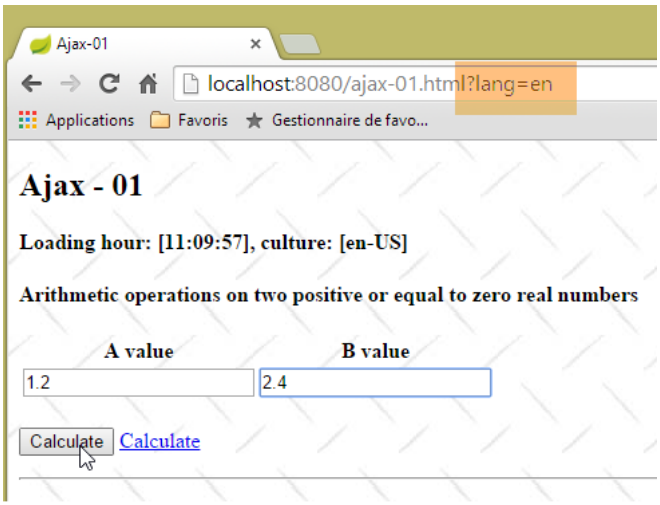
sur [Calculer], les validateurs lui répondent que les valeurs [a,b] sont invalides. Ce qui est juste. C'est là qu'intervient la fonction [afterComplete] exécutée à la réception du résultat :

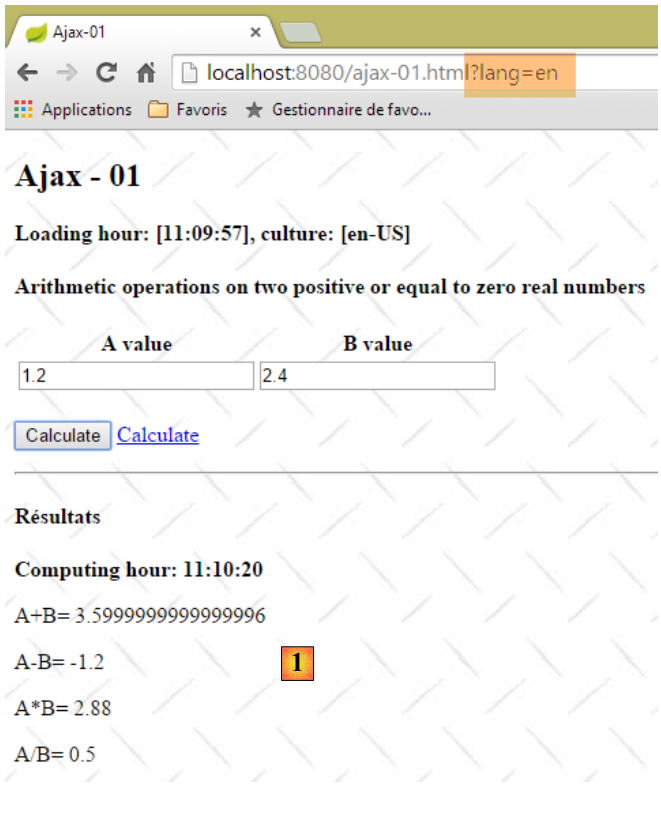
```
1.  function beforeSend(jqXHR, settings) {
2.     // avant le POST
3.  ...
4.  }
5.
6.  function afterComplete(jqXHR, settings) {
7.     // après le POST
8.     // les nombres doivent être remis au format français si nécessaire
9.     var culture = Globalize.culture().name;
10.    if (culture === 'fr-FR') {
11.       checkCulture(2);
12.    }
13. }
14.
15. function checkCulture(mode) {
16.    if (mode == 1) {
17. ...
18.    }
19.    if (mode == 2) {
20.       // on met les nombres au format français
21.       var value1 = a.val().replace(".", ",");
22.       a.val(value1);
23.       var value2 = b.val().replace(".", ",");
24.       b.val(value2);
25.    }
26. }
```

- lignes 9-12 : si la culture de la vue est [fr-FR], on remet les nombres [a,b] au format français.

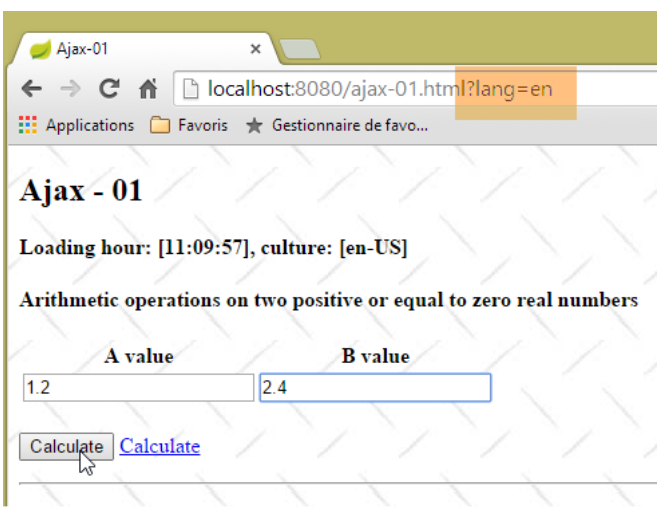
7.2.7 Tests

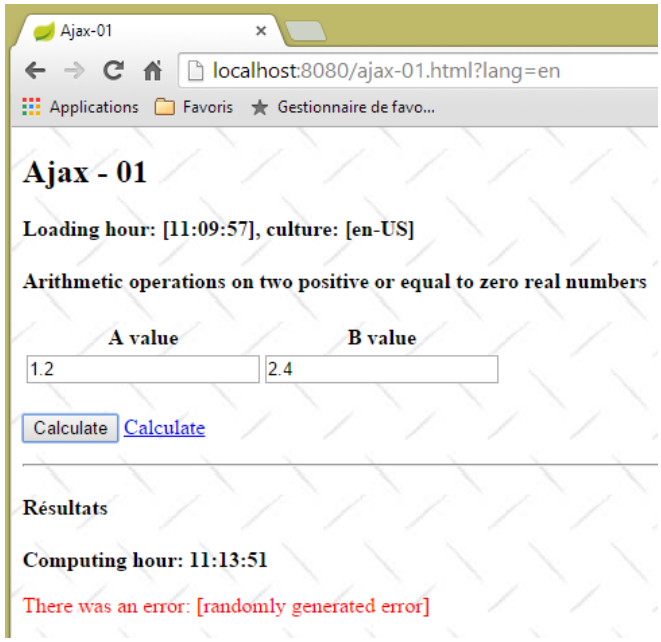
Voici quelques copies d'écran de tests :



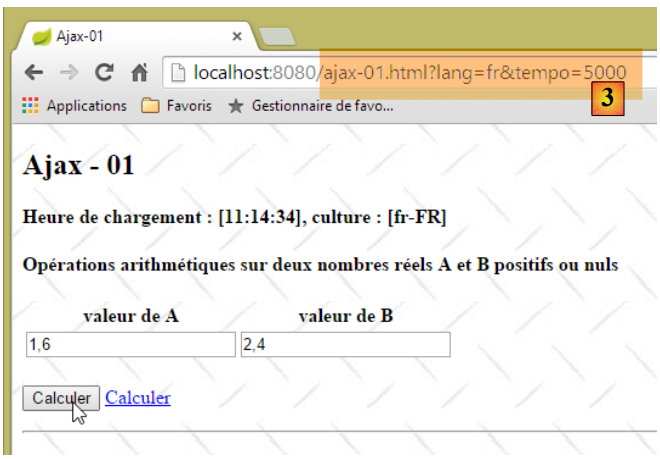


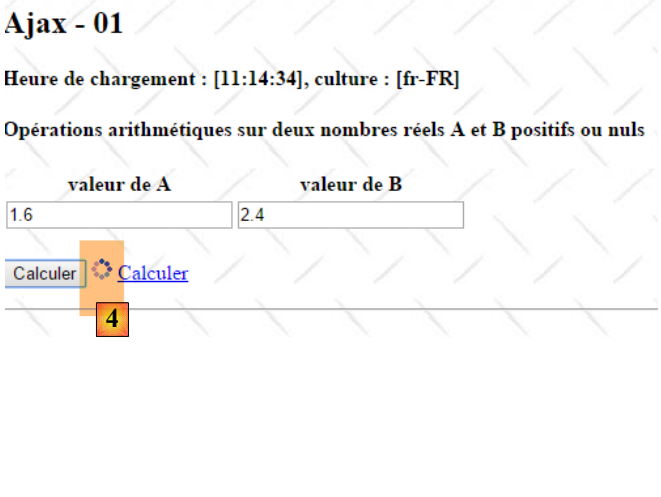
- en [1], la réponse du serveur ;





- en [2], la réponse du serveur avec un message d'erreur ;





- en [3], on fixe une temporisation de 5 secondes. Cela veut dire que le serveur attendra 5 secondes avant d'envoyer sa réponse. Dans la balise [form], nous avons utilisé l'attribut [`data-ajax-loading='#loading'`]. Le paramètre [loading] est l'identifiant d'une zone qui est :
  - affichée pendant toute la durée de l'attente ;
  - cachée après réception de la réponse du serveur ;

  Ici [loading] est l'identifiant d'une image animée qu'on voit en [4].

7.2.8 Désactivation du Javascript avec la culture [en-US]

Que se passe-t-il si on désactive le Javascript du navigateur ?

Le POST des valeurs saisies va se faire selon la balise [form] dont les attributs [data-ajax-attr] ne vont pas être utilisés. Tout se passe comme si on avait la balise [form] suivante :

```
<form id="formulaire" name="formulaire" method="post" action="/ajax-02.html">
```

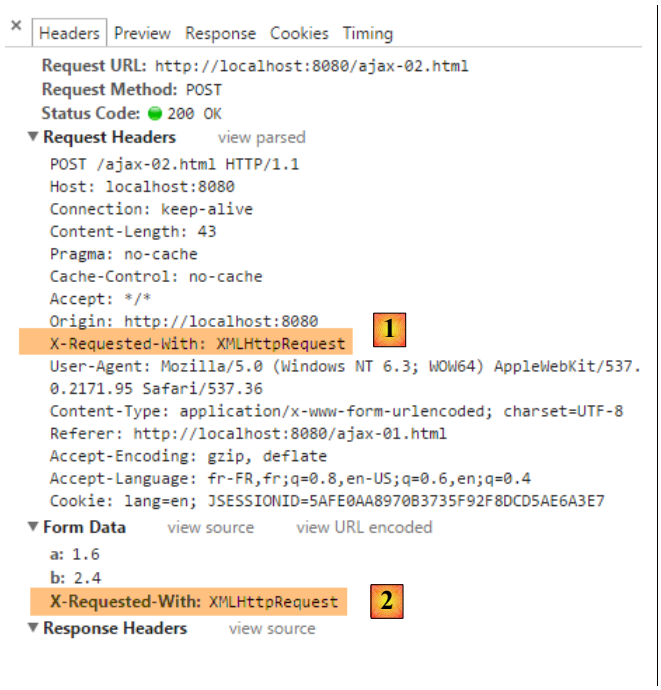
Les valeurs saisies vont donc être postées à l'action [/ajax-02]. Elles n'auront pas été vérifiées côté client. Ce sont donc les validateurs côté serveur qui vont intervenir. Ils intervenaient déjà auparavant mais sur des valeurs déjà validées côté client, donc correctes. Ce n'est plus le cas.

Nous modifions l'action [/ajax-02] de la façon suivante :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html;
    charset=UTF-8")
2.     public String ajax02(@Valid ActionModel01 formulaire, BindingResult result, Locale
    locale, Model modèle, HttpSession session, HttpServletRequest request) throws
    InterruptedException {
3.         // requête Ajax ?
4.         boolean isAjax = "XMLHttpRequest".equals(request.getHeader("X-Requested-With"));
5.         ...
6.     }
```

- ligne 4 : l'action [/ajax-02] peut donc désormais être appelé via un POST Ajax ou via un POST classique. Il nous faut savoir différencier ces deux cas. On le fait avec les entêtes HTTP envoyés par le navigateur client ;

Lorsqu'on regarde les échanges réseau dans la console de développement de Chrome (Ctrl-Maj-I) alors que le Javascript est activé, on voit que le client envoie les entêtes suivants au moment du POST :



On voit ci-dessus que :

- un entête [X-Requested-With] a été envoyé [1] ;
- un paramètre [X-Requested-With] a été ajouté aux valeurs postées [2] ;

Ceci n'est pas fait dans le cas d'un POST classique. On a donc deux possibilités pour récupérer l'information : la récupérer dans les entêtes HTTP ou dans les valeurs postées. La ligne 4 de l'action [/ajax-02] a choisi la première solution.

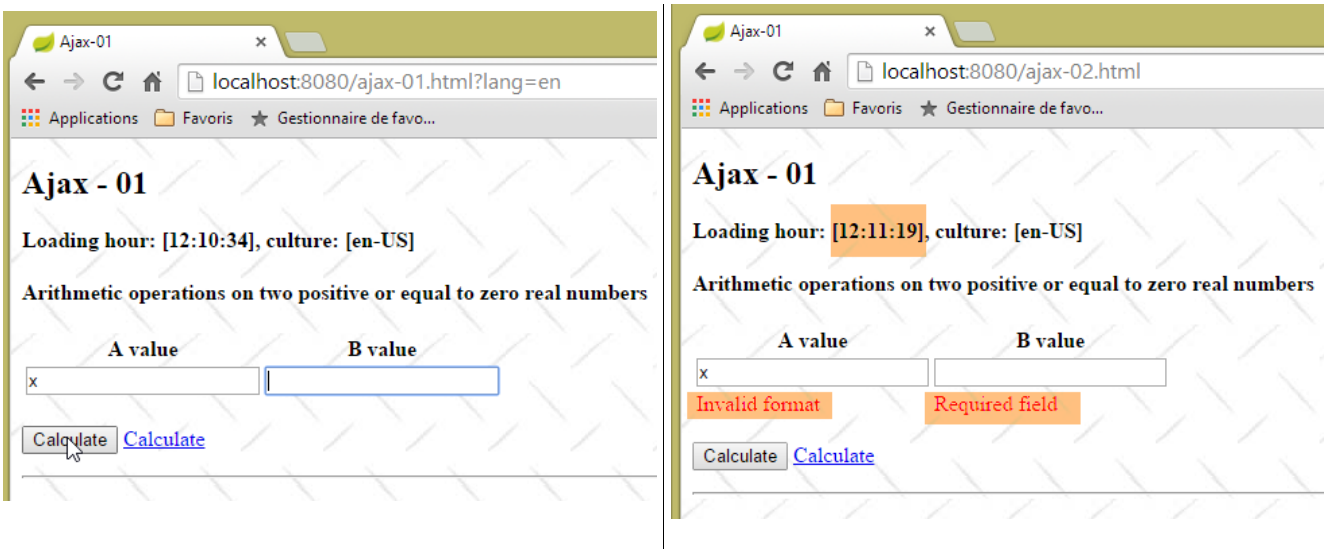
Continuons avec le code de cette action :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String ajax02(@Valid ActionModel01 formulaire, BindingResult result, Locale locale, Model
    modèle, HttpSession session, HttpServletRequest request) throws InterruptedException {
3.         // requête Ajax ?
4.         boolean isAjax = "XMLHttpRequest".equals(request.getHeader("X-Requested-With"));
```

```
5.        // tempo ?
6.        Integer tempo = (Integer) session.getAttribute("tempo");
7.        if (tempo != null && tempo > 0) {
8.           Thread.sleep(tempo);
9.        }
10.       // on prépare le modèle de la prochaine vue
11.       Resultats résultats = new Resultats();
12.       modèle.addAttribute("resultats", résultats);
13.       // on fixe la locale
14.       setLocale(locale, modèle, résultats);
15.       // heure
16.       String heure = new SimpleDateFormat("hh:mm:ss").format(new Date());
17.       résultats.setHeurePost(heure);
18.       résultats.setHeureGet(heure);
19.       // requête valide ?
20.       if (!isAjax && result.hasErrors()) {
21.          return "vue-01";
22.       }
23. ...
```

- ligne 2 : le paramètre [@Valid ActionModel01 formulaire] actionne les validateurs côté serveur ;
- lignes 20-22 : si l'appel n'est pas un appel Ajax et que la validation a échoué, alors on renvoie la vue [vue-01.xml] avec les messages d'erreur.

Voici un exemple :



Continuons l'étude de l'action [/ajax-02] :

```
1.  @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
2.     public String ajax02(@Valid ActionModel01 formulaire, BindingResult result, Locale locale, Model
    modèle,   HttpSession session, HttpServletRequest request) throws InterruptedException {
3.        // requête Ajax ?
4.        boolean isAjax = "XMLHttpRequest".equals(request.getHeader("X-Requested-With"));
5.  ...
6.        // requête valide ?
7.        if (!isAjax && result.hasErrors()) {
8.           return "vue-01";
9.        }
10.       // on génère une erreur une fois sur deux
11.       int val = new Random().nextInt(2);
12.       if (val == 0) {
13.          // on renvoie un message d'erreur
14.          résultats.setErreur("erreur.aleatoire");
15.          if (isAjax) {
16.             return "vue-03";
17.          } else {
18.             résultats.setVue("vue-03");
19.             return "vue-01";
20.          }
```

```
21.         }
22. ...
```

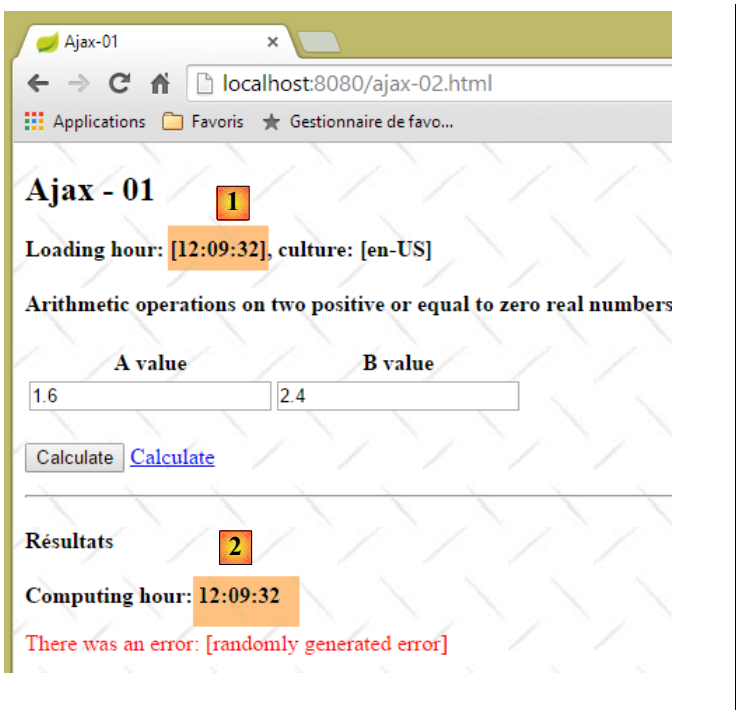
- ligne 14 : on génère une erreur aléatoire ;
- ligne 16 : dans le cas d'un appel Ajax, on retourne la vue [vue-03.xml] qui sera placée dans la zone identifiée par [resultats] ;
- ligne 18 : dans le cas d'un appel non Ajax, on met la vue à afficher dans le modèle de type [Resultats] ;
- ligne 19 : on rend de nouveau la vue [vue-01.xml] ;

La vue [vue-01.xml] est modifiée de la façon suivante :

```
1.  <div id="resultats" />
2.  <div th:if="${resultats.vue}=='vue-02'" th:include="vue-02" />
3.  <div th:if="${resultats.vue}=='vue-03'" th:include="vue-03" />
```

- ligne 3 : la vue [vue-03.xml] va être insérée sous la zone [resultats] ;

Voici un exemple :



On notera que désormais les heures [1] et [2] sont identiques.

Continuons l'étude de l'action [/ajax-02] :

```
1.     @RequestMapping(value = "/ajax-02", method = RequestMethod.POST, produces = "text/html; charset=UTF-
   8")
2.     public String ajax02(@Valid ActionModel01 formulaire, BindingResult result, Locale locale, Model
   modèle, HttpSession session, HttpServletRequest request) throws InterruptedException {
3.         // requête Ajax ?
4.         boolean isAjax = "XMLHttpRequest".equals(request.getHeader("X-Requested-With"));
5.  ...
6.         // on récupère les valeurs postées
7.         double a = formulaire.getA();
8.         double b = formulaire.getB();
9.         // on construit le modèle
10.        résultats.setAplusb(String.valueOf(a + b));
11.        résultats.setAmoinsb(String.valueOf(a - b));
12.        résultats.setAmultiplieparb(String.valueOf(a * b));
13.        try {
14.           résultats.setAdiviseparb(String.valueOf(a / b));
15.        } catch (RuntimeException e) {
16.           résultats.setAdiviseparb("NaN");
```

```
17.         }
18.         // on affiche la vue
19.         if (isAjax) {
20.            return "vue-02";
21.         } else {
22.            résultats.setVue("vue-02");
23.            return "vue-01";
24.         }
25.  }
```

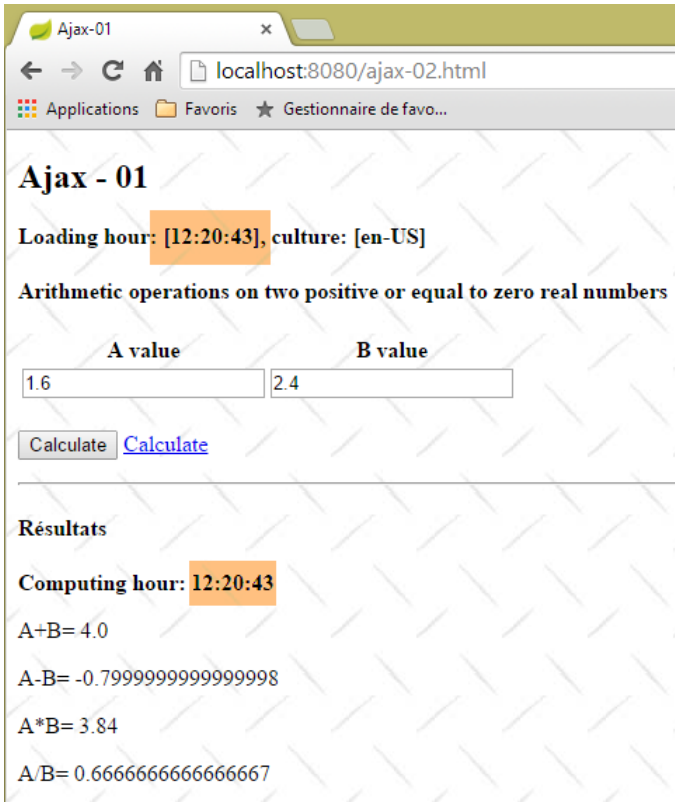
- lignes 7-17 : les résultats des quatre opération arithmétiques sont mises dans le modèle ;
- lignes 22-23 : on rend la vue [vue-01.xml] (ligne 22) en lui insérant la vue [vue-02.xml] (ligne 22) ;

Cette insertion de fait de la façon suivante dans [vue-01.xml] :

```
1.  <div id="resultats" />
2.  <div th:if="${resultats.vue}=='vue-02'" th:include="vue-02" />
3.  <div th:if="${resultats.vue}=='vue-03'" th:include="vue-03" />
```

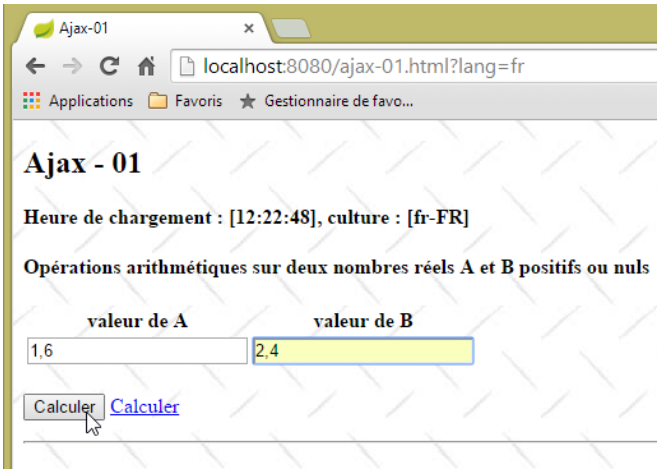
- ligne 2 : la vue [vue-02.xml] va être insérée sous la zone [resultats] ;

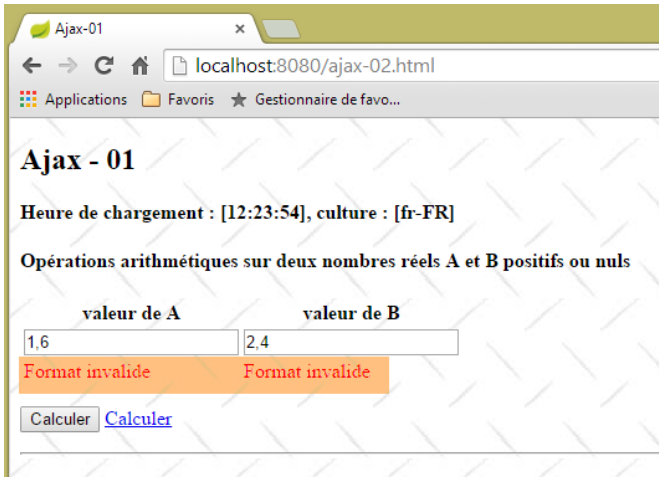
Voici un exemple d'exécution :



7.2.9 Désactivation du Javascript avec la culture [fr-FR]

Avec la culture [fr-FR] on a le problème suivant :





Les valeurs saisies au format français ont été déclarées invalides. En effet, le serveur attend des réels au format anglo-saxon. La solution est assez complexe. Nous allons créer un filtre qui va :

- intercepter la requête ;
- changer les virgules dans les valeurs postées [a] et [b] en point décimal ;
- puis passer la nouvelle requête à l'action qui doit la traiter ;

Tout d'abord, nous introduisons un champ caché dans la vue [vue-01.xml] :

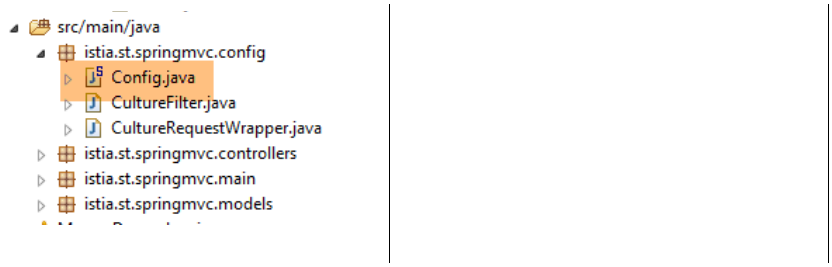
```
1.  <form ...>
2.  ...
3.  </p>
4.     <!-- champs cachés -->
5.     <input type="hidden" id="culture" name="culture" th:value="${resultats.culture}"></input>
6.  </form>
```

- ligne 5 : la culture [fr-FR] ou [en-US] est mise dans le champ d'attribut [name=culture]. Comme la balise [input] est dans le formulaire, sa valeur va être postée avec les valeurs de [a] et [b]. On aura alors une chaîne postée de la forme :

```
culture=fr-FR&a=12,7&b=20,78
```

  Il est important de comprendre ce point.

Ensuite nous incluons un filtre dans la configuration de l'application :

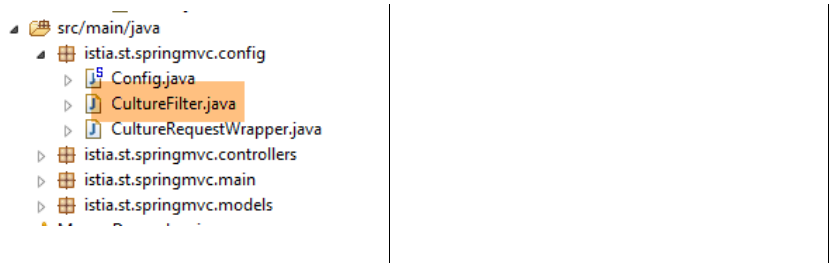


Le fichier [Config] est modifié de la façon suivante :

```
1.  @Configuration
2.  @ComponentScan({ "istia.st.springmvc.controllers", "istia.st.springmvc.models" })
3.  @EnableAutoConfiguration
4.  public class Config extends WebMvcConfigurerAdapter {
5.  ...
6.     @Bean
7.     public Filter cultureFilter() {
8.        return new CultureFilter();
9.     }
10. }
```

- ligne 7 : le fait que le bean [cultureFilter] rende un type [Filter] fait de lui un filtre. Le bean, lui, peut porter un nom quelconque ;

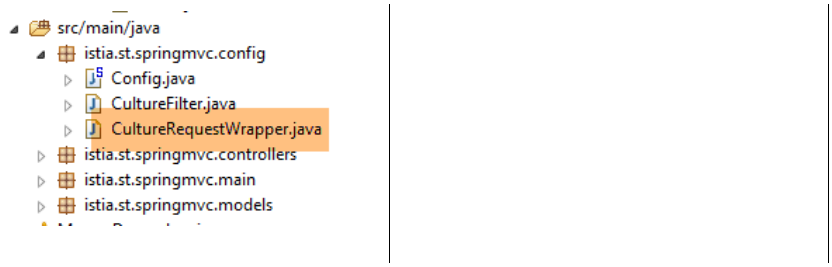
L'étape suivante est de créer le filtre lui-même :




```
1.  package istia.st.springmvc.config;
2.
3.  import java.io.IOException;
4.
5.  import javax.servlet.FilterChain;
6.  import javax.servlet.ServletException;
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.filter.OncePerRequestFilter;
11.
12. public class CultureFilter extends OncePerRequestFilter {
13.
14.     @Override
15.     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
    filterChain)
16.             throws ServletException, IOException {
17.         // handler suivant
18.         filterChain.doFilter(new CultureRequestWrapper(request), response);
19.     }
20. }
```

- ligne 12 : nous étendons la classe [OncePerRequestFilter] qui est une classe Spring et ce que nous devons faire est de rédéfinir la méthode [doFilterInternal] de cette classe ;
- ligne 15 : la méthode [doFilterInternal] reçoit trois informations :
  - [HttpServletRequest request] : la requête à filtrer. Celle-ci ne peut être modifiée,
  - [HttpServletResponse response] : la réponse qui sera faite au serveur. Le filtre peut décider de la faire lui-même,
  - [FilterChain filterChain] : la chaîne des filtres. Une fois que la méthode [doFilterInternal] a fini son travail, elle doit passer la requête au filtre suivant de la chaîne des filtres ;
- ligne 18 : on crée une nouvelle requête à partir de celle qu'on a reçue [**new** CultureRequestWrapper(request)] et on la passe au filtre suivant. Parce qu'on ne peut modifier la requête initiale [HttpServletRequest request], on en crée une nouvelle ;

La classe [CultureRequestWrapper] est la suivante :

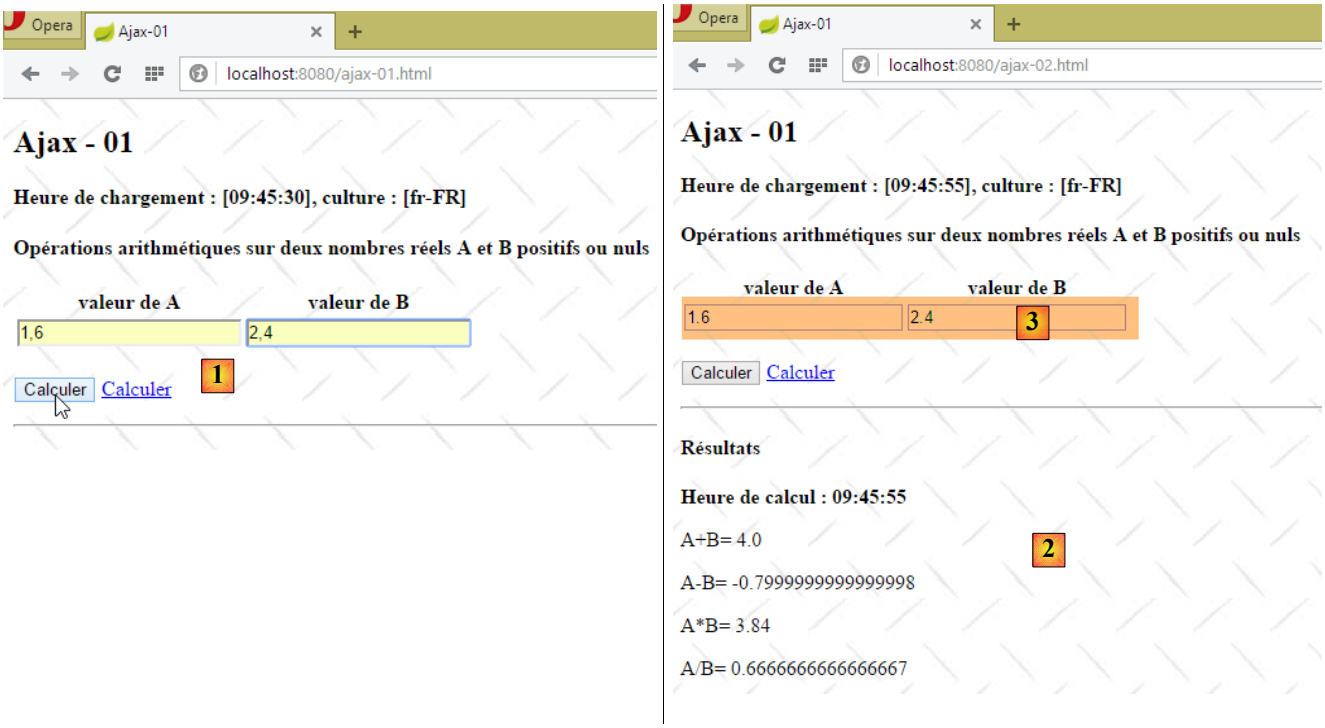



```
1.  package istia.st.springmvc.config;
2.
3.  import javax.servlet.http.HttpServletRequest;
4.  import javax.servlet.http.HttpServletRequestWrapper;
5.
6.  public class CultureRequestWrapper extends HttpServletRequestWrapper {
```

```
7.
8.      public CultureRequestWrapper(HttpServletRequest request) {
9.         super(request);
10.     }
11.
12.     @Override
13.     public String[] getParameterValues(String name) {
14.        // valeurs postées a et b
15.        if (name != null && (name.equals("a") || name.equals("b"))) {
16.           String[] values = super.getParameterValues(name);
17.           String[] newValues = values.clone();
18.           newValues[0] = newValues[0].replace(",", ".");
19.           return newValues;
20.        }
21.        // autres cas
22.        return super.getParameterValues(name);
23.     }
24.
25. }
```

- ligne 6 : la classe [CultureRequestWrapper] étend la classe [HttpServletRequestWrapper] et va redéfinir certaines de ces méthodes ;
- lignes 8-10 : le constructeur qui reçoit la requête à filtrer et la passe à la classe parent ;
- il faut comprendre ici que la requête filtrée va au final aboutir comme paramètre d'entrée d'une classe appelée une servlet. Avec Spring MVC, cette servlet est de type [DispatcherServlet]. Cette classe dispose de diverses méthodes pour récupérer les paramètres de la requête : [getParameter, getParameterMap, getParameterNames, getParameterValues, ...]. Il faut redéfinir la méthode utilisée par la servlet. Il faudrait lire pour cela le code de la classe [DispatcherServlet]. Je ne l'ai pas fait et j'ai redéfini diverses méthodes. C'est finalement la méthode [getParameterValues] qui a été redéfinie ;
- ligne 13 : la méthode [getParameterValues] reçoit en paramètre, le nom d'un des paramètres rendus par la méthode [getParameterNames] et doit rendre le tableau de ses valeurs. En effet, on sait qu'un paramètre peut être présent en plusieurs exemplaires dans une requête ;
- ligne 18 : on remplace la virgule par un point décimal ;

Voici un exemple d'exécution :



- en [1], les valeurs [a,b] sont saisies au format français ;
- en [2], les résultats ;
- en [3], le serveur a renvoyé une page avec des nombres au format anglo-saxon.

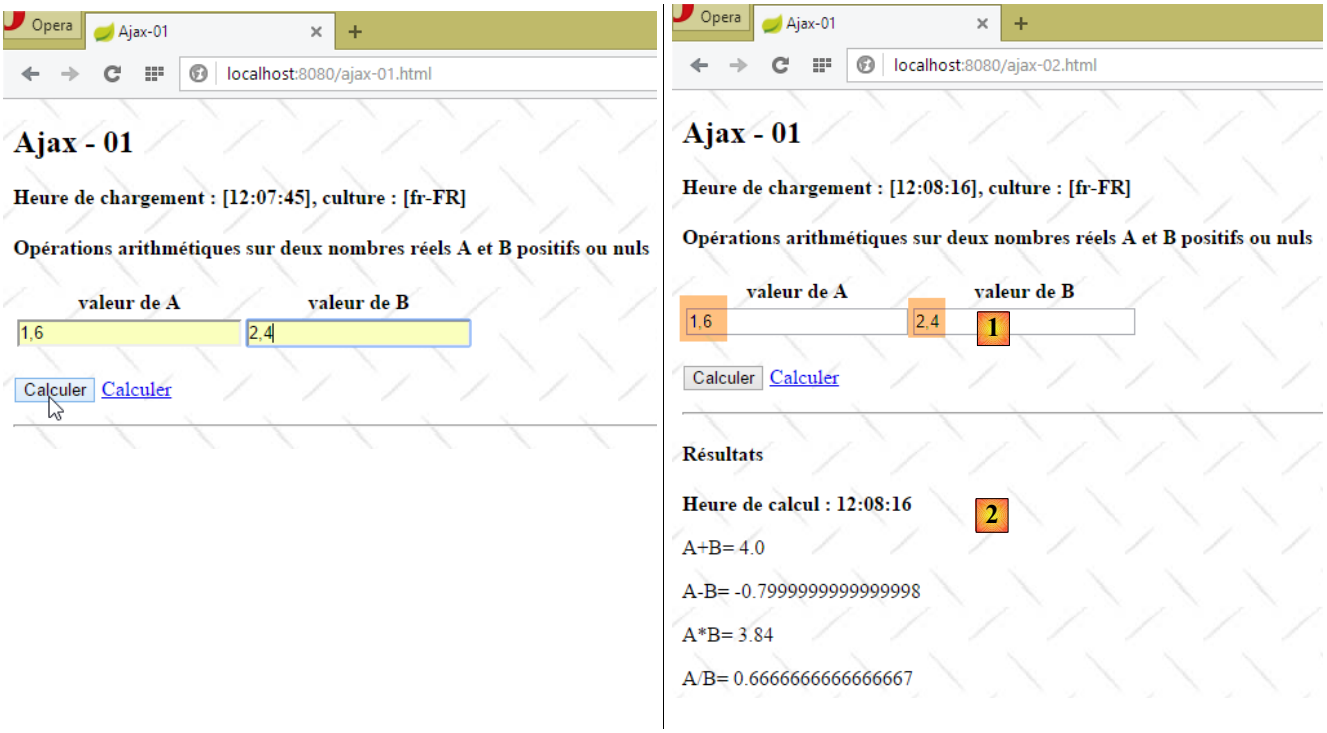
Ce dernier problème peut-être résolu avec Thymeleaf de la façon suivante dans la vue [vue-01.xml]

```
1.  <tr>
2.     <td>
3.        <input type="text" id="a" name="a" th:value="${resultats.culture}=='fr-FR' and ${actionModel01.a}!
    =null? ${#strings.replace(actionModel01.a,'.',',')} : ${actionModel01.a}" data-val="true" th:attr="data-
    val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-min=#{actionModel01.a.min},data-val-min-
    value=#{actionModel01.a.min.value}" />
4.     </td>
5.     <td>
6.        <input type="text" id="b" name="b" th:value="${resultats.culture}=='fr-FR' and ${actionModel01.b}!
    =null? ${#strings.replace(actionModel01.b,'.',',')} : ${actionModel01.b}" data-val="true" th:attr="data-
    val-required=#{NotNull},data-val-number=#{typeMismatch},data-val-min=#{actionModel01.b.min},data-val-min-
    value=#{actionModel01.b.min.value}" />
7.     </td>
8.  </tr>
```

Il y a plusieurs modification à faire lignes 3 et 6. Nous allons raisonner sur la ligne 3 :

- on avait écrit [th:field="*{a}"]. Le paramètre [th:field] fixe les attributs [id, name, value] de la balise HTML [input] générée. Ici, on veut gérer l'attribut [value] nous-mêmes. On fixe donc aussi les attributs [id, name] nous-mêmes ;
- l'attribut [th:value] évalue une expression utilisant l'opérateur ternaire ?. On teste l'expression [*${resultats.culture}=='fr-FR' and ${actionModel01.b}!=null*]. Si elle est vraie on donne à l'attribut [value] la valeur de a [*actionModel01.a*] où le point décimal est remplacé par la virgule. Si elle est fausse, on donne à l'attribut [value] la valeur de a [*actionModel01.a*] sans modifications ;
- ligne 6 : on refait la même chose pour le champ [b] ;

Voici un exemple d'exécution :



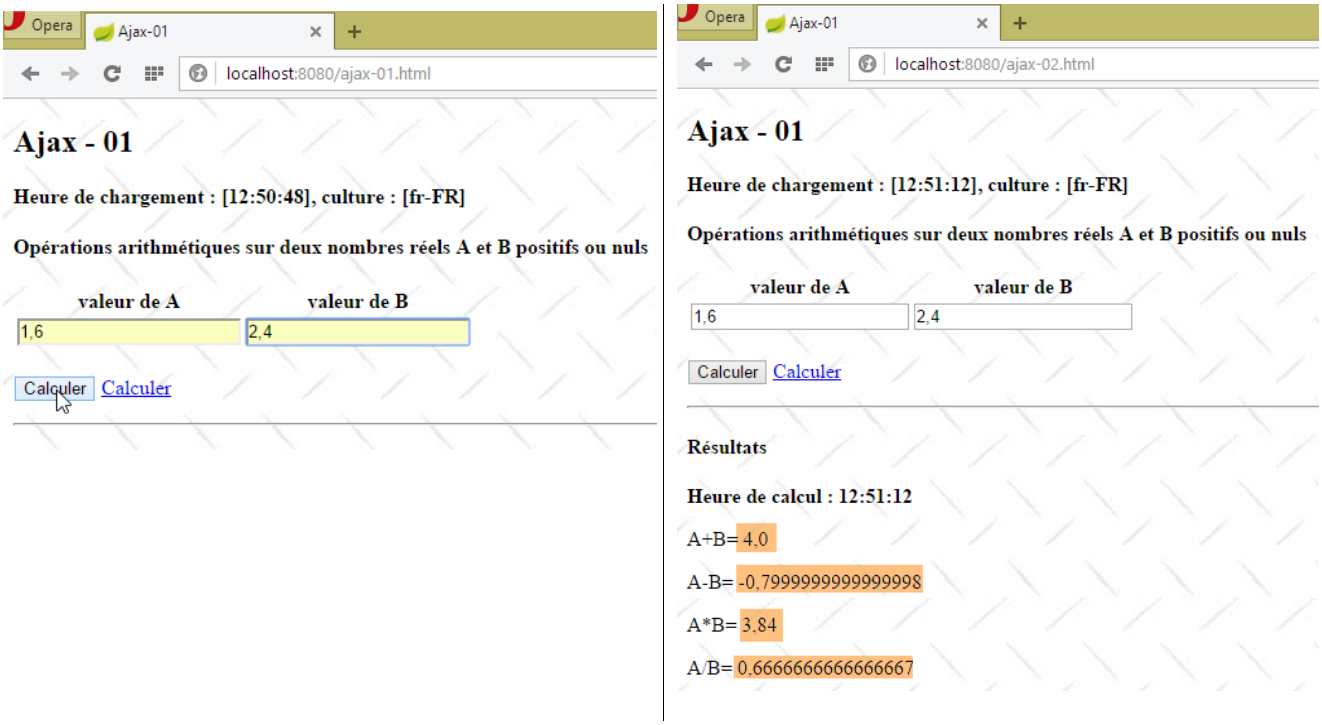
- en [1], les nombres [a,b] ont gardé la notation française. Ce n'est pas le cas en [2] ;

Ce nouveau problème se gère de la même façon que le précédent. On modifie la vue [vue-03.xml] de la façon suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h4 th:text="#{resultats}">Résultats</h4>
5.        <p>
6.           <strong>
7.              <span th:text="#{labelHeureCalcul}">Heure de calcul :</span>
8.              <span id="heureCalcul" th:text="${resultats.heurePost}"></span>
9.           </strong>
10.       </p>
11.       <p>
```

```
12.          <span th:text="#{labelAplusB}">A+B=</span>
13.          <span id="aplusb" th:text="${resultats.culture}=='fr-FR' and ${resultats.aplusb}!=null? ${#strings.replace(resultats.aplusb,'.',',')} : ${resultats.aplusb}"></span>
14.       </p>
15.       <p>
16.          <span th:text="#{labelAmoinsB}">A-B=</span>
17.          <span id="amoinsb" th:text="${resultats.culture}=='fr-FR' and ${resultats.amoinsb}!=null? ${#strings.replace(resultats.amoinsb,'.',',')} : ${resultats.amoinsb}"></span>
18.       </p>
19.       <p>
20.          <span th:text="#{labelAfoisB}">A*B=</span>
21.          <span id="amultiplieparb" th:text="${resultats.culture}=='fr-FR' and ${resultats.amultiplieparb}!=null? ${#strings.replace(resultats.amultiplieparb,'.',',')} : ${resultats.amultiplieparb}"></span>
22.       </p>
23.       <p>
24.          <span th:text="#{labelAdivB}">A/B=</span>
25.          <span id="adiviseparb" th:text="${resultats.culture}=='fr-FR' and ${resultats.adiviseparb}!=null? ${#strings.replace(resultats.adiviseparb,'.',',')} : ${resultats.adiviseparb}"></span>
26.       </p>
27.    </body>
28. </html>
```

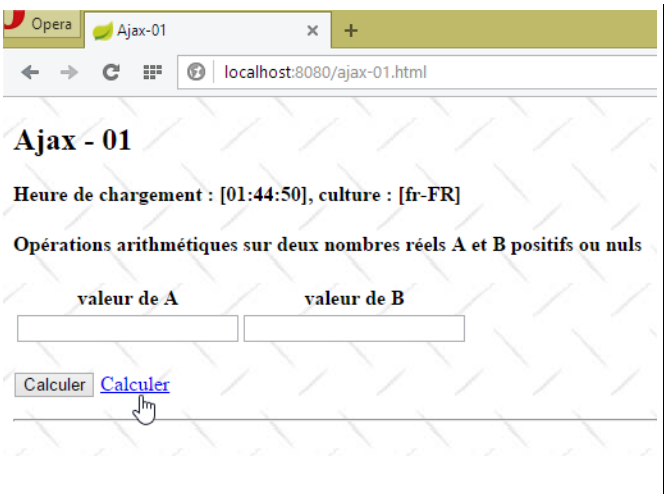
Voici un exemple :

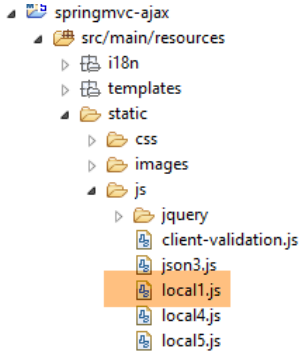


On a désormais une application qui gère correctement deux cultures dans un environnement utilisant ou non du Javascript. Il a fallu pour cela complexifier de façon importante le code côté serveur. Par la suite, nous supposerons toujours que le Javascript du navigateur est activé. Cela permet des choses impossibles en mode serveur uniquement.

7.2.10 Gestion du lien [Calculer]

Examinons le lien [Calculer] de la page principale [vue-01.xml] :





Le code du lien [Calculer] dans la vue [vue-01.xml] est le suivant :

```
<a href="javascript:postForm()" th:text="#{action.calculer}">Calculer</a>
```

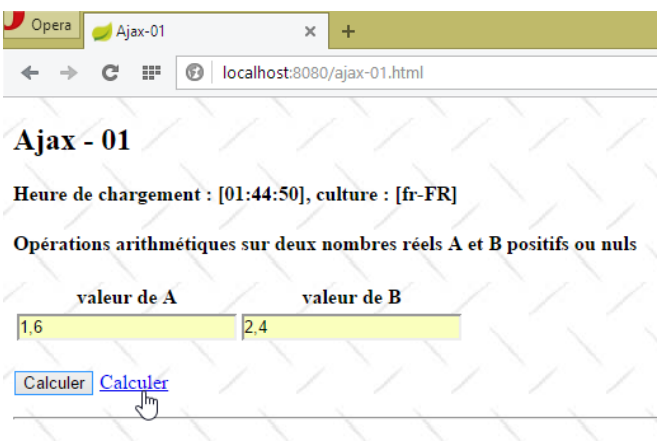
La fonction jS [postForm] est définie dans le fichier [local1.js] de la façon suivante :

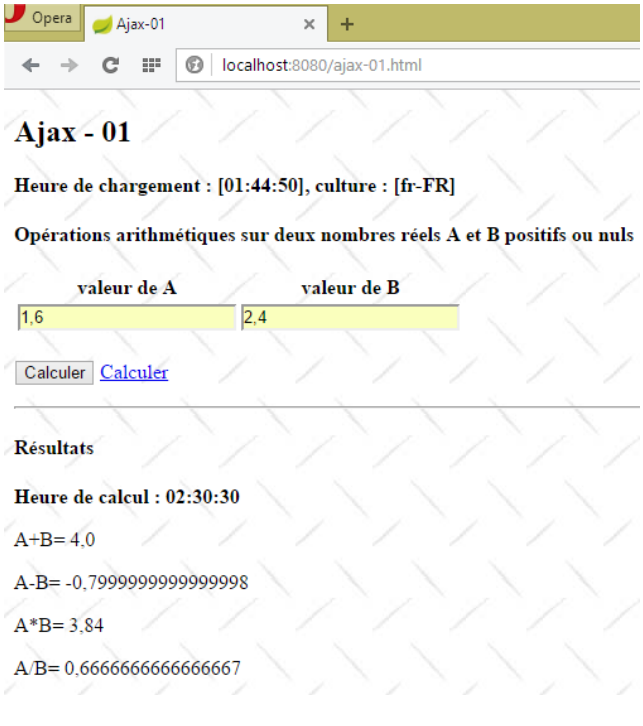
```
1.  // données globales
2.  var loading;
3.  var formulaire;
4.  var résultats;
5.  var a, b;
6.
7.  function postForm() {
8.     // formulaire valide ?
9.     if (!formulaire.validate().form()) {
10.       // formulaire invalide - terminé
11.       return;
12.    }
13.    // on gère deux locales [fr_FR, en_US]
14.    // les réels [a,b] doivent être postés au format anglo-saxon dans tous les cas
15.    // ils le seront par le filtre [CultureFilter]
16.
17.    // on fait un appel Ajax à la main
18.    $.ajax({
19.       url : '/ajax-02',
20.       headers : {
21.          'X-Requested-With' : 'XMLHttpRequest'
22.       },
23.       type : 'POST',
24.       data : formulaire.serialize(),
25.       dataType : 'html',
26.       beforeSend : function() {
27.          loading.show();
28.       },
29.       success : function(data) {
30.          resultats.html(data);
31.       },
32.       complete : function() {
33.          loading.hide();
34.       },
35.       error : function(jqXHR) {
36.          résultats.html(jqXHR.responseText);
37.       }
38.    })
39. }
```

- lignes 2-5 : rappelons que ces éléments ont été initialisés par la fonction [`$(document).ready`] ;
- lignes 9-12 : on exécute les validateurs jS du formulaire. Si l'une des valeurs est invalide, l'expression [formulaire.validate().form()] rend la valeur *false*. Dans ce cas, le [submit] du formulaire est annulé ;
- lignes 18-38 : on fait un appel Ajax à la main ;
- ligne 19 : l'URL cible de l'appel Ajax ;

- lignes 20-22 : un tableau d'entêtes HTTP à ajouter à ceux présents par défaut dans la requête HTTP. Ici, on ajoute l'entête HTTP qui va indiquer au serveur qu'on fait un appel Ajax ;
- ligne 23 : la méthode HTTP utilisée ;
- ligne 24 : les données postées. [formulaire.serialize] crée la chaîne à poster [culture=fr-FR&a=12,7&b=20,89] du formulaire d'**id** [formulaire]. On va retrouver ici le problème étudié précédemment : il faut que les valeurs [a,b] soient postées au format anglo-saxon. On sait que ce problème a été désormais réglé avec la création du filtre [cultureFilter] ;
- ligne 25 : le type de données attendu en retour. On sait que le serveur va renvoyer un flux HTML ;
- ligne 26 : la méthode à exécuter lorsque la requête démarre. Ici, on indique qu'il faut afficher le composant d'**id** [loading]. C'est l'image animée d'attente ;
- ligne 29 : la méthode à exécuter en cas de succès de la requête Ajax. Le paramètre [data] est la réponse complète du serveur. On sait que c'est un flux HTML ;
- ligne 30: on met à jour le composant d'**id** [résultats] avec le HTML du paramètre [data].
- ligne 33 : on cache le signal d'attente ;
- ligne 35 : fonction exécutée lorsque la réponse du serveur a été reçue, quelle que soit celle-ci, succès ou erreur ;
- lignes 35-37 : en cas d'erreur (le serveur a renvoyé une réponse HTTP avec un statut indiquant qu'il y a eu erreur côté serveur), on affiche la réponse HTML du serveur dans la zone [resultats] ;

Voici un exemple d'exécution :





7.3 Mise à jour d'une page HTML avec un flux jSON

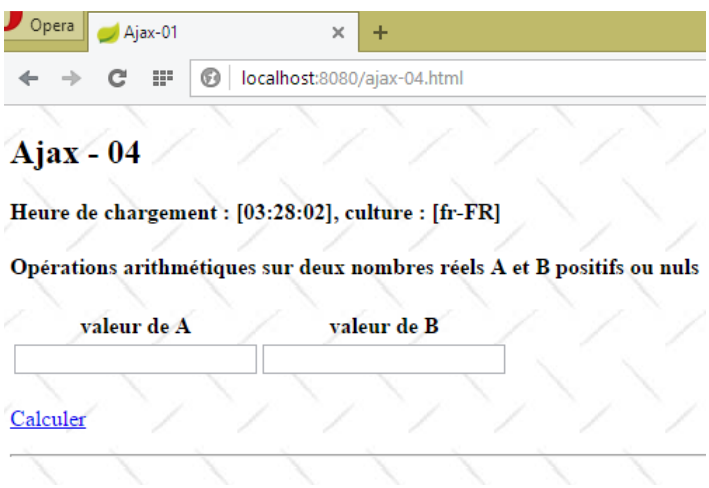
Dans l'exemple précédent, le serveur web répondait à la requête HTTP Ajax par un flux HTML. Dans ce flux, il y avait des données accompagnées par du formatage HTML. On se propose de reprendre l'exemple précédent avec cette fois-ci des réponses jSON (**J**ava**S**cript **O**bject **N**otation) ne contenant que les données. L'intérêt est qu'on transmet ainsi moins d'octets. On suppose que le Javascript est activé sur le navigateur.

7.3.1 L'action [/ajax-04]

L'action [/ajax-04] est identique à l'action [/ajax-01], si ce n'est qu'on affiche la vue [vue-04.xml] au lieu de la vue [vue-01.xml] :

```
1.  @RequestMapping(value = "/ajax-04", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.      public String ajax04(Locale locale, Model modèle, HttpSession session, String tempo) {
3.          ...
4.          // vue
5.          return "vue-04";
6.      }
```

7.3.2 La vue [vue-04.xml]



La vue [vue-04.xml] reprend le corps de la vue [vue-01.xml] avec les différences suivantes :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        ...
5.        <script type="text/javascript" src="/js/local4.js"></script>
6.        <script th:inline="javascript">
7.           /*<![CDATA[*/
8.                 var culture = [[${resultats.culture}]];
9.                 Globalize.culture(culture);
10.                /*]]>*/
11.       </script>
12.    </head>
13.    <body>
14.       <h2>Ajax - 04</h2>
15.    ...
16.       <form id="formulaire" name="formulaire" th:object="${actionModel01}">
17. ...
18.          <p>
19.             <img id="loading" style="display: none" src="/images/loading.gif" />
20.             <a href="javascript:postForm()" th:text="#{action.calculer}">Calculer</a>
21.          </p>
22.          <!-- champs cachés -->
23.          <input type="hidden" id="culture" name="culture" th:value="$
    {resultats.culture}"></input>
24. </form>
25.       <hr />
26.       <div id="entete">
27.          <h4 id="titre">Résultats</h4>
28.          <p>
29.             <strong>
30.                <span id="labelHeureCalcul">Heure de calcul :</span>
31.                <span id="heureCalcul">12:10:87</span>
32.             </strong>
33.          </p>
34.       </div>
35.       <div id="résultats">
36.          <p>
37.             A+B=
```

```
38.            <span id="aplusb">16,7</span>
39.         </p>
40.         <p>
41.            A-B=
42.            <span id="amoinsb">16,7</span>
43.         </p>
44.         <p>
45.            A*B=
46.            <span id="afoisb">16,7</span>
47.         </p>
48.         <p>
49.            A/B=
50.            <span id="adivb">16,7</span>
51.         </p>
52.      </div>
53.      <div id="erreur">
54.         <p style="color: red;">
55.            <span id="msgErreur">xx</span>
56.         </p>
57.      </div>
58.   </body>
59. </html>
```

- ligne 5 : le Javascript de la vue est désormais dans le fichier [local4.js] ;
- ligne 16 : la balise [form] n'a plus les paramètres [data-ajax-attr] de la bibliothèque [Unobtrusive Ajax]. Nous n'allons pas l'utiliser ici. La balise [form] n'a pas non plus les attributs [method] et [action] qui indiquent comment et où poster les valeurs saisies dans le formulaire. Ceci parce que celui-ci va être posté par une fonction jS (ligne 20) ;
- lignes 26-57 : la zone d'id [resultats] qui auparavant était une zone vide contient désormais du code HTML pour afficher les résultats ;
- lignes 26-34 : l'entête des résultat où l'heure de calcul est affichée ;
- lignes 35-52 : les résultats des quatre opérations arithmétiques ;
- lignes 53-57 : un éventuel message d'erreur envoyé par le serveur ;

Le code jS exécuté au chargement de la vue [vue-04.xm] est dans le fichier [local4.js]. C'est le suivant :

```
1.  // données globales
2.     var loading;
3.     var formulaire;
4.     var résultats;
5.     var titre;
6.     var labelHeureCalcul;
7.     var heureCalcul;
8.     var aplusb;
9.     var amoinsb;
10.    var afoisb;
11.    var adivb;
12.    var msgErreur;
13.
14. // au chargement du document
15. $(document).ready(function() {
16.    // on récupère les références des différents composants de la page
17.    loading = $("#loading");
18.    formulaire = $("#formulaire");
19.    résultats = $('#résultats');
20.    titre=$("#titre");
21.    labelHeureCalcul=$("#labelHeureCalcul");
22.    heureCalcul=$("#heureCalcul");
23.    aplusb=$("#aplusb");
24.    amoinsb=$("#amoinsb");
25.    afoisb=$("#afoisb");
26.    adivb=$("#adivb");
27.    msgErreur=$("#msgErreur");
28.    // on cache certains éléments
29.    résultats.hide();
30.    erreur.hide();
31.    loading.hide();
32. });
```

- lignes 17-27 : on récupère les références jQuery de tous les éléments de la page ;
- ligne 29 : la zone des résultats est cachée ;
- ligne 30 : ainsi que la zone de l'erreur ;
- ligne 31 : ainsi que l'image animée d'attente ;
- lignes 2-12 : les références récupérées sont faites globales afin que les autres fonctions puissent en disposer ;

7.3.3 La fonction jS [postForm]

Le lien [Calculer] est le suivant :

```
1.  <p>
2.     <img id="loading" style="display: none" src="/images/loading.gif" />
3.     <a href="javascript:postForm()" th:text="#{action.calculer}">Calculer</a>
4.  </p>
```

La fonction jS [postForm] est définie dans le fichier [local.js] de la façon suivante :

```
1.  function postForm() {
2.     // formulaire valide ?
3.     if (!formulaire.validate().form()) {
4.        // formulaire invalide - terminé
5.        return;
6.     }
7.     // on fait un appel Ajax à la main
8.     $.ajax({
9.        url : '/ajax-05',
10.       headers : {
11.          'Accept' : 'application/json'
12.       },
13.       type : 'POST',
14.       data : formulaire.serialize(),
15.       dataType : 'json',
16.       beforeSend : onBegin,
17.       success : onSuccess,
18.       error : onError,
19.       complete : onComplete
20.    })
21. }
22.
23. // avant l'appel Ajax
24. function onBegin() {
25. ...
26. }
27.
28. // à réception de la réponse du serveur
29. // en cas de succès
30. function onSuccess(data) {
31. ...
32. }
33.
34. // à réception de la réponse du serveur
35. // en cas d'échec
36. function onError(jqXHR) {
37. ...
38. }
39.
40. // après [onSuccess, onError]
41. function onComplete() {
42. ...
43. }
```

- lignes 3-6 : avant de poster les valeurs saisies, on les vérifie. Si elles sont incorrectes, on ne fait pas le POST du formulaire ;
- ligne 9 : les valeurs saisies sont envoyées à l'action [/ajax-05] que nous détaillons un peu plus loin ;
- lignes 10-12 : un entête HTTP pour dire au serveur qu'on attend une réponse au format jSON ;
- ligne 13 : les valeurs saisies vont être postées ;
- ligne 14 : sérialisation des valeurs saisies en une chaîne prête à être postée [a=1,6&b=2,4&culture=fr-FR] ;
- ligne 15 : le type de la réponse envoyée par le serveur. Ce sera du jSON ;
- ligne 16 : la fonction à exécuter avant le POST ;
- ligne 17 : la fonction à exécuter à réception de la réponse du serveur si celle-ci est un succès. Le 'succès' d'une requête HTTP est mesuré à l'aune du statut de la réponse HTTP du serveur. Une réponse [`HTTP/1.1 200 OK`] est une réponse

de succès. Une réponse [HTTP/1.1 500 Internal Server Error] est une réponse d'échec. Ce qu'on appelle le statut d'une réponse HTTP est le code [200] ou [500]. Un certain nombre de ces codes sont reliés au 'succès' alors que d'autres codes sont reliés à 'l'échec' ;

- ligne 18 : la fonction à exécuter à réception de la réponse du serveur lorsque le statut HTTP de cette de cette réponse est un statut d'échec ;
- ligne 18 : la fonction à exécuter en dernier lieu, après les fonctions [onSuccess, onError] précédentes ;

La fonction [onBegin] est la suivante :

```
1.  // avant l'appel Ajax
2.  function onBegin() {
3.      console.log("onBegin");
4.      // on montre l'image animée
5.      loading.show();
6.      // on cache certains éléments de la vue
7.      entete.hide();
8.      résultats.hide();
9.      erreur.hide();
10. }
```

Avant d'étudier les autres fonctions jS de l'appel Ajax, nous avons besoin de connaître la réponse envoyée par l'action [/ajax-05].

7.3.4 L'action [/ajax-05]

L'action [/ajax-05] est la suivante :

```
1.      @RequestMapping(value = "/ajax-05", method = RequestMethod.POST)
2.      @ResponseBody()
3.      // traite le POST de la vue [vue-04]
4.      public JsonResults ajax05(@Valid ActionModel01 formulaire, BindingResult result, Locale locale,
        HttpServletRequest request, HttpSession session) throws InterruptedException {
5.          if(result.hasErrors()){
6.              // cas anormal - on ne rend rien
7.              return null;
8.          }
9.          ...
10. }
```

- ligne 2 : l'attribut [ResponseBody] indique que l'action [/ajax-05] rend elle-même la réponse au client. Parce qu'une bibliothèque jSON est dans les dépendances du projet, Spring Boot autoconfigure ce type d'actions pour qu'elles rendent du jSON. C'est donc la chaîne jSON d'un type [JsonResults] (ligne 4) qui va être envoyée au client ;
- ligne 2 : les valeurs postées [a, b, culture] vont être encapsulées dans un type [ActionModel01] dont on demande la validation [@Valid ActionModel01]. C'est pour la forme. On est parti sur l'hypothèse que le Javascrit était activé sur le navigateur client et donc lorsqu'elles arrivent, les valeurs postées ont déjà été vérifiées côté client. Néanmoins, on peut prévoir le cas d'un POST sauvage qui n'utiliserait pas notre client jS. Dans ce cas, la validation peut échouer ;
- lignes 5-7 : en cas d'erreur, on rend un flux jSON vide ;

Continuons l'étude de l'action [/ajax-05] :

```
1.      @RequestMapping(value = "/ajax-05", method = RequestMethod.POST)
2.      @ResponseBody()
3.      // traite le POST de la vue [vue-04]
4.      public JsonResults ajax05(@Valid ActionModel01 formulaire, BindingResult result, Locale locale,
5.              HttpServletRequest request, HttpSession session) throws InterruptedException {
6.  ...
7.          // le contexte de l'application Spring
8.          WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(request.getServletContext());
9.          // tempo ?
10.         Integer tempo = (Integer) session.getAttribute("tempo");
11.         if (tempo != null && tempo > 0) {
12.             Thread.sleep(tempo);
13.         }
14.     ...
15.         // on rend le résultat
16.         return résultats;
17. }
```

- ligne 8 : on récupère le contexte [ctx] de l'application Spring. On en a besoin pour récupérer les messages des fichiers [messages.properties] à partir d'une clé de message et d'une locale. Cela se fait avec la syntaxe suivante :

```
ctx.getMessage(clé_message, tableau_de_paramètres, locale)
```

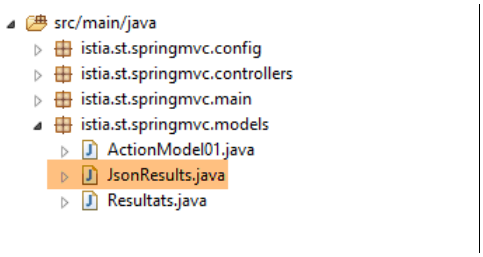
  [**clé_message**] : la clé du message recherché ;
  [**locale**] : la locale utilisée. Ainsi si cette locale est [en_US], c'est le fichier [messages_en.properties] qui sera exploité ;
  [**tableau_de_paramètres**] : le message obtenu peut être paramétré tel que dans [clé=message {0} {1}]. Il y a dans ce message deux paramètres [{0} {1}]. Il faudra fournir comme second paramètre de [ctx.getMessage] un tableau de deux valeurs ;
- lignes 10-13 : s'il y a une temporisation dans la session, on arrête le thread courant le temps de celle-ci ;

L'action [/ajax-05] se poursuit de la façon suivante :

```
1.         // on prépare le modèle de la prochaine vue
2.         JsonResults résultats = new JsonResults();
3.         ...
4.   }
```

- ligne 2 : création du modèle de la chaîne jSON envoyée au client ;

Le modèle [JsonResults] est le suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  public class JsonResults {
4.
5.     // data
6.     private String titre;
7.     private String labelHeureCalcul;
8.     private String heureCalcul;
9.     private String aplusb;
10.    private String amoinsb;
11.    private String afoisb;
12.    private String adivb;
13.    private String msgErreur;
14.
15.    // getters et setters
16. ...
17.
18. }
```

- lignes 6-13 : chacun des champs de la classe [JsonResult] correspond à un champ de même [id] dans la vue [vue-04.xml] :

L'action [/ajax-05] se poursuit de la façon suivante :

```
1.         // on prépare le modèle de la prochaine vue
2.         JsonResults résultats = new JsonResults();
3.         // entête
4.         résultats.setTitre(ctx.getMessage("resultats.titre", null, locale));
5.         résultats.setLabelHeureCalcul(ctx.getMessage("labelHeureCalcul", null, locale));
6.         résultats.setHeureCalcul(new SimpleDateFormat("hh:mm:ss").format(new Date()));
7.         // on génère une erreur une fois sur deux
8.         int val = new Random().nextInt(2);
9.         if (val == 0) {
10.           // on renvoie un message d'erreur
11.           résultats.setMsgErreur(ctx.getMessage("resultats.erreur",
12.                 new Object[] { ctx.getMessage("erreur.aleatoire", null, locale) }, locale));
```

```
13.           return résultats;
14. }
```

- ligne 2 : création du modèle de la chaîne jSON envoyée au client ;
- lignes 4-6 : on crée les messages de l'entête des résultats ;
- lignes 8-14 : une fois sur deux en moyenne, on génère un message d'erreur. Dans ce cas, on ne va pas plus loin et on rend la chaîne jSON au client (ligne 13) ;
- ligne 11 : on a ici un exemple de message paramétré :

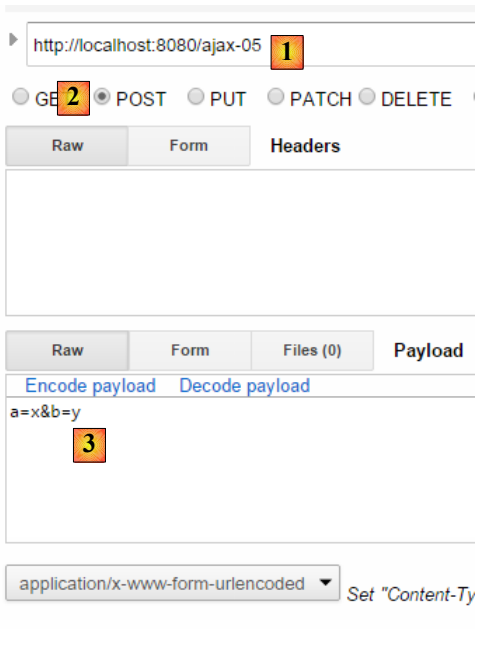
```
1. erreur.aleatoire=erreur aléatoire
2. resultats.erreur=Une erreur s''est produite : [{0}]
```

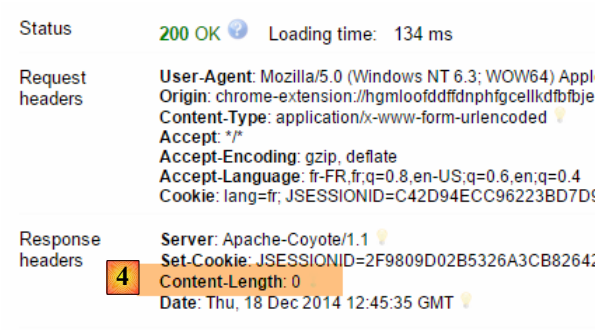
L'action [/ajax-05] se poursuit de la façon suivante :

```
1.        // on récupère les valeurs postées
2.        double a = formulaire.getA();
3.        double b = formulaire.getB();
4.        // on construit le modèle
5.        résultats.setAplusb(String.valueOf(a + b));
6.        résultats.setAmoinsb(String.valueOf(a - b));
7.        résultats.setAfoisb(String.valueOf(a * b));
8.        try {
9.           résultats.setAdivb(String.valueOf(a / b));
10.       } catch (RuntimeException e) {
11.          résultats.setAdivb("NaN");
12.       }
13.       // on rend le résultat
14. return résultats;
```

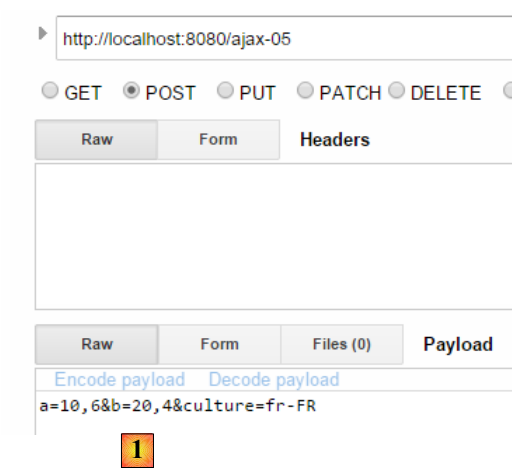
- lignes 2-3 : on récupère les valeurs de [a] et [b] ;
- lignes 5-12 : on construit les quatre résultats ;
- ligne 14 : la chaîne jSON [JsonResults] est envoyée au client ;

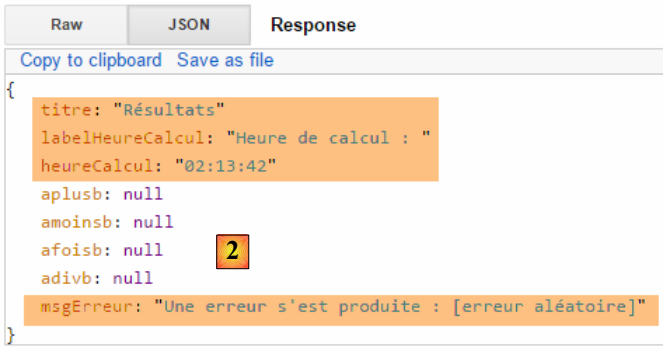
Voyons ce que ça donne avec le client [Advanced Rest Client] :



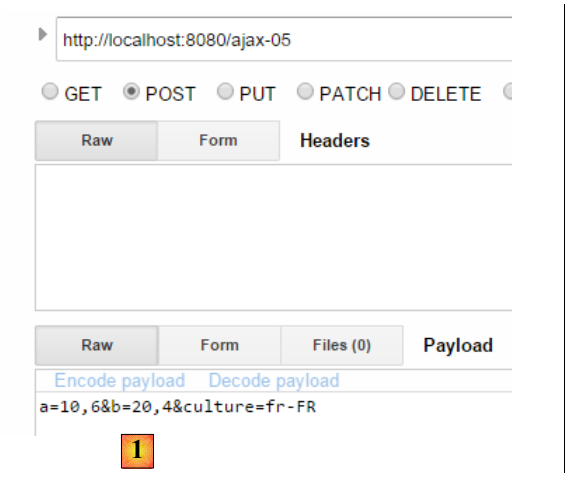


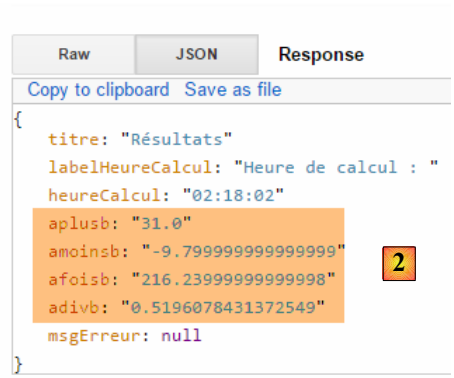
- en [1-2], on fait une requête POST à l'action [/ajax-05] ;
- en [3], on poste des valeurs incorrectes ;
- en [4], le serveur a renvoyé un flux vide ;



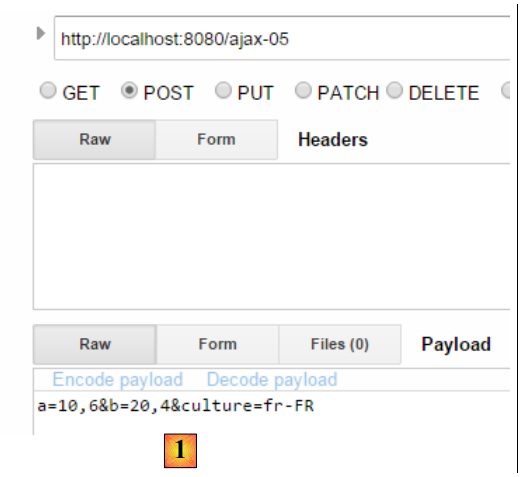


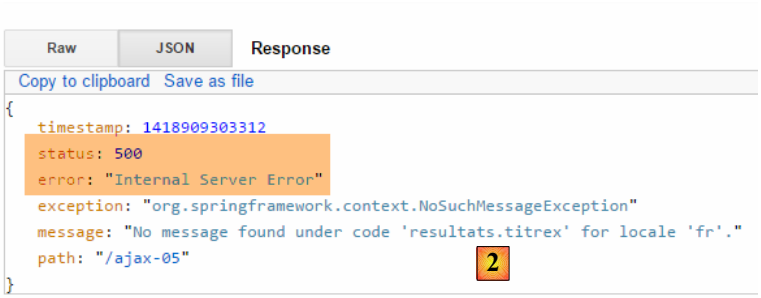
- en [1], on poste des valeurs correctes ;
- en [2], l'objet jSON renvoyé par le serveur, avec ici un message d'erreur ;





- en [1], on poste des valeurs correctes ;
- en [2], l'objet jSON renvoyé par le serveur, avec ici les quatre résultats ;





- en [1], on poste des valeurs correctes ;
- en [2], on s'est arranagé pour provoquer une exception côté serveur. On voit que le serveur envoie encore un objet jSON. Dans ce message, on voit que le statut HTTP de la réponse est [500], indiquant qu'il y a eu une erreur côté serveur ;

7.3.5 La fonction jS [postForm] - 2

Maintenant que nous connaissons l'objet jSON renvoyé par le serveur, on peut l'exploiter dans le javascript. La méthode [onSuccess] exécutée lorsque le serveur envoie une réponse avec le statut HTTP [200] est la suivante :

```
1.  // à réception de la réponse du serveur
2.  // en cas de succès
3.  function onSuccess(data) {
4.     console.log("onSuccess");
5.     // on remplit la zone des résultats
6.     titre.text(data.titre);
7.     labelHeureCalcul.text(data.labelHeureCalcul);
8.     heureCalcul.text(data.heureCalcul);
9.     entete.show();
10.    // résultats sans erreur
11.    if (!data.msgErreur) {
12.       aplusb.text(data.aplusb);
13.       amoinsb.text(data.amoinsb);
14.       afoisb.text(data.afoisb);
15.       adivb.text(data.adivb);
16.       résultats.show();
17.       return;
18.    }
19.    // résultats avec erreur
20.    msgErreur.text(data.msgErreur);
21.    erreur.show();
22. }
```

- ligne 3 : le paramètre [data] est l'objet jSON renvoyé par le serveur :

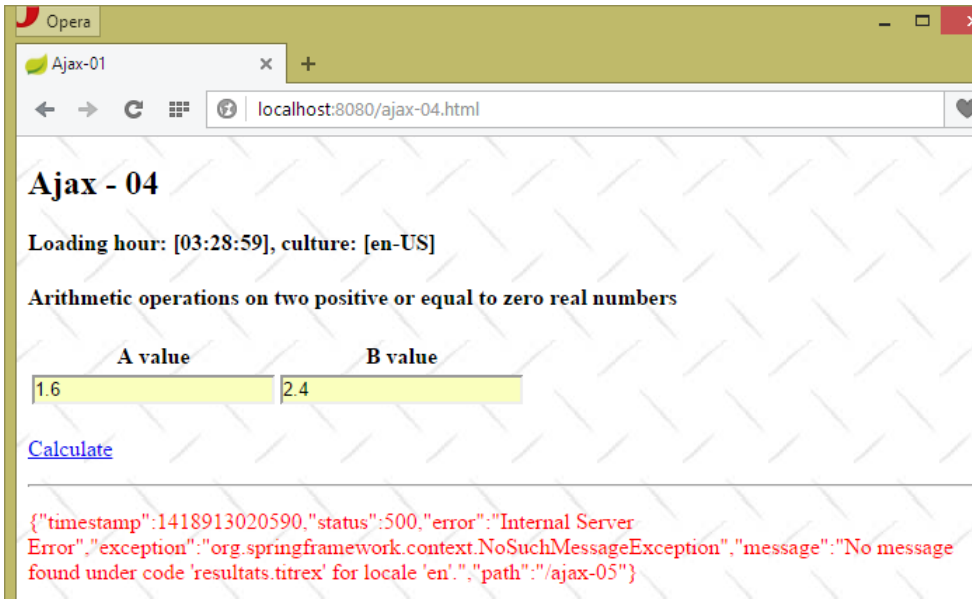
La méthode [onError] exécutée lorsque le statut de la réponse HTTP est [500] est la suivante :

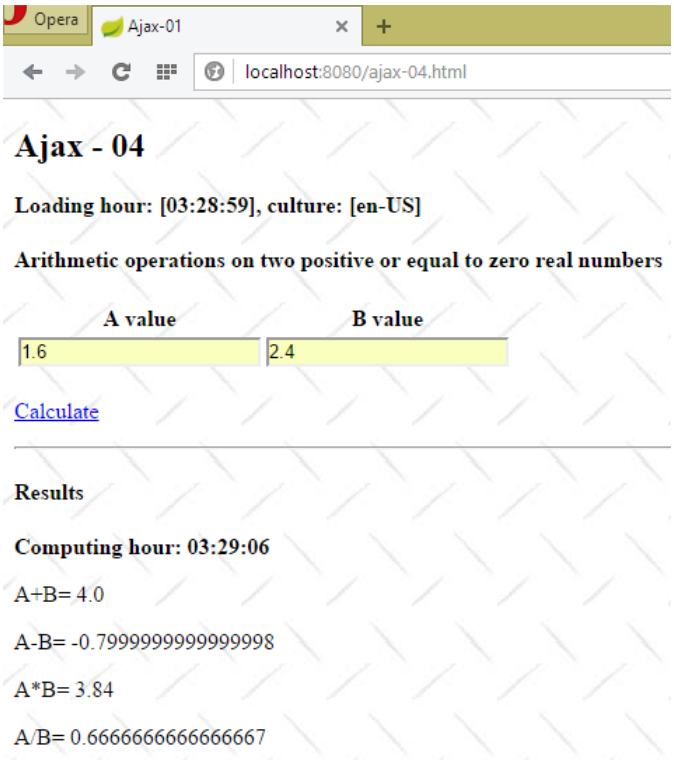
```
1.  // à réception de la réponse du serveur
2.  // en cas d'échec
3.  function onError(jqXHR) {
4.     console.log("onError");
5.     // erreur système
6.     msgErreur.text(jqXHR.responseText);
7.     erreur.show();
8.  }
```

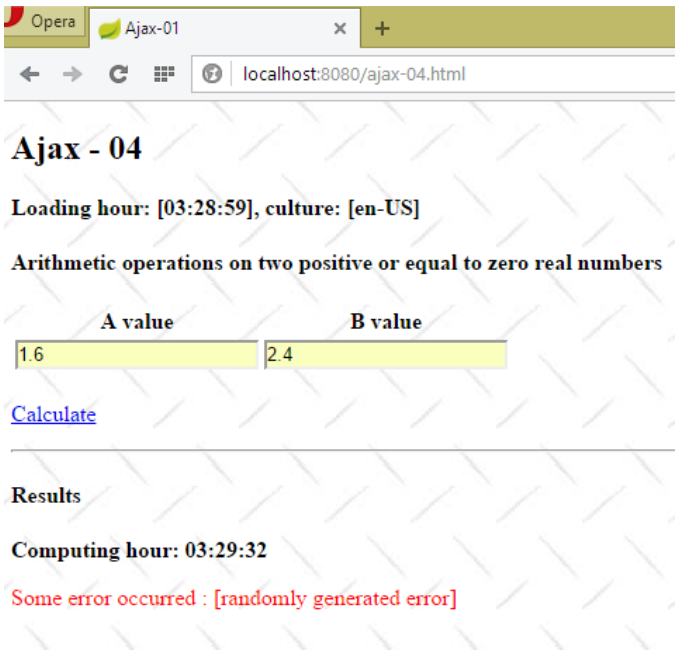
- ligne 3 : l'objet JQuery [**jqXHR**] a parmi ses propriétés les suivantes :
  - **responseText** : le texte de la réponse du serveur,
  - **status** : le code d'erreur retourné par le serveur,
  - **statusText** : le texte associé à ce code d'erreur ;
- ligne 6 : l'objet [jqXHR.responseText] est l'objet jSON suivant :

7.3.6 Tests

Voyons quelques copies d'écran d'exécution de l'application web :







7.4 Application web à page unique

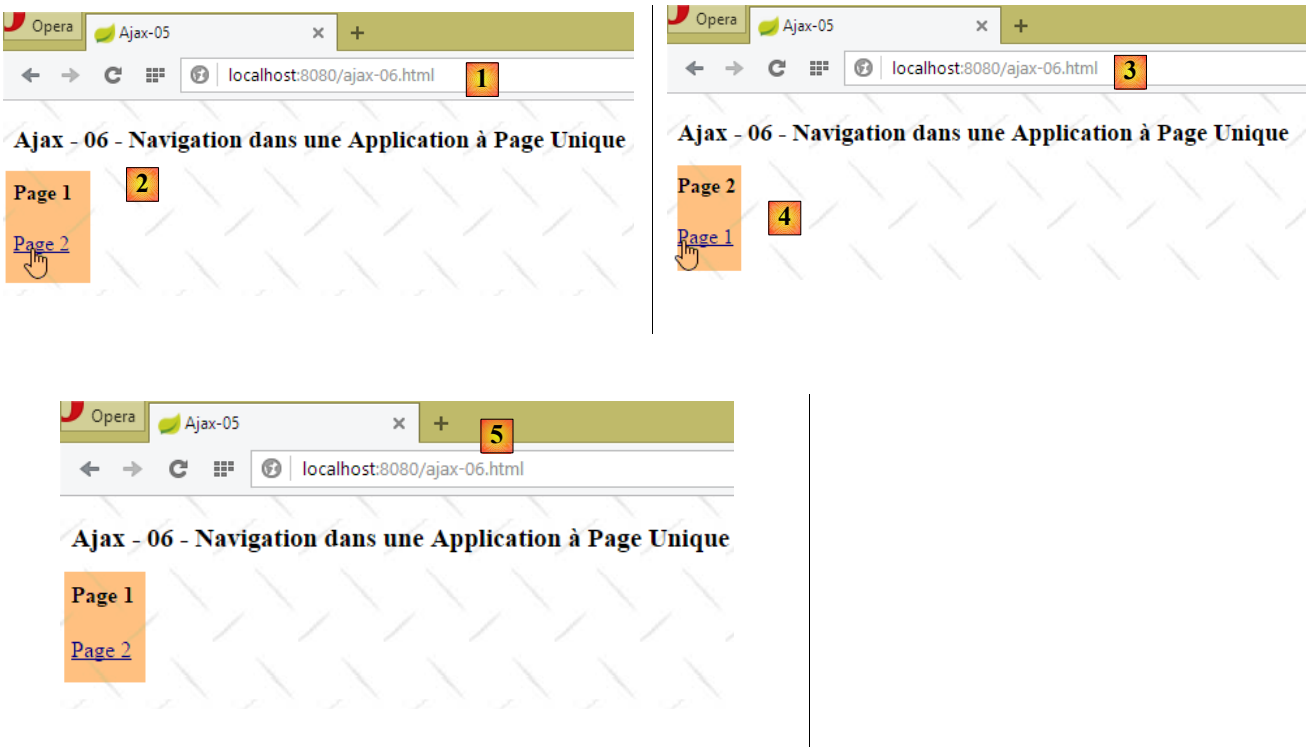
7.4.1 Introduction

La technologie Ajax permet de construire des applications à **page unique** :

- la première page est issue d'une requête classique d'un navigateur ;

- les pages suivantes sont obtenues avec des appels Ajax. Aussi, au final le navigateur ne change jamais d'URL et ne charge jamais de nouvelle page. On appelle ce type d'application, **Application à Page Unique** (APU) ou en anglais **Single Page Application** (SPA).

Voici un exemple basique d'une telle application. La nouvelle application aura deux vues :



- en [1], l'action [/ajax-06] nous permet d'avoir la première page, la **page 1** ;
- en [2], un lien nous permet de passer à la **page 2** grâce à un appel Ajax ;
- en [3], l'URL n'a pas changé. La page présentée est la **page 2** ;
- en [4], un lien nous permet de revenir à la **page 1** grâce à un appel Ajax ;
- en [5], l'URL n'a pas changé. La page présentée est la **page 1**.

7.4.2 L'action [/ajax-06]

Le code de l'action [/ajax-06] est le suivant :

```
1.     @RequestMapping(value = "/ajax-06", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.     public String ajax06() {
3.         return "vue-06";
4.  }
```

- lignes 1-4 : l'action [/ajax-06] se contente de rendre la vue [vue-06.xml] ;

7.4.3 La vue [vue-06.xml]

La vue [vue-06.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>Ajax-06</title>
6.        <link rel="stylesheet" href="/css/ajax01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
8.        <script type="text/javascript" src="/js/local6.js"></script>
9.     </head>
10.    <body>
11.       <h3>Ajax - 06 - Navigation dans une Application à Page Unique</h3>
12.       <div id="content" th:include="vue-07" />
```

```
13.     </body>
14. </html>
```

- ligne 8 : la vue utilise un script [local6.js] ;
- ligne 12 : on inclut la vue [vue-07.xml] dans la zone d'id [content] de la vue [vue-06.xml] ;

7.4.4 La vue [vue-07.xml]

La vue [vue-07.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h4>Page 1</h4>
5.        <p>
6.           <a href="javascript:gotoPage(2)">Page 2</a>
7.        </p>
8.     </body>
9.  </html>
```

7.4.5 La fonction jS [gotoPage]

Le lien [Page 2] de la vue [vue-07.xml] utilise la fonction jS [gotoPage] définie dans le fichier [local6.js] suivant :

```
1.  // données globales
2.  var content;
3.
4.  function gotoPage(num) {
5.     // on fait un appel Ajax à la main
6.     $.ajax({
7.        url : '/ajax-07',
8.        type : 'POST',
9.        data : 'num=' + num,
10.       dataType : 'html',
11.       beforeSend : function() {
12.       },
13.       success : function(data) {
14.          content.html(data)
15.       },
16.       complete : function() {
17.       },
18.       error : function(jqXHR) {
19.          // erreur système
20.          content.html(jqXHR.responseText);
21.       }
22.    })
23. }
24.
25. // au chargement du document
26. $(document).ready(function() {
27.    // on récupère les références des différents composants de la page
28.    content = $("#content");
29. });
```

- ligne 28 : au chargement de la page, on mémorise la zone d'id [content] et on en fait une variable globale (ligne 2) ;
- ligne 4 : la fonction [gotoPage] reçoit comme paramètre le n° de la page (1 ou 2) à afficher dans la vue actuelle ;
- ligne 7 : l'URL cible du POST ;
- ligne 8 : l'URL de la ligne 7 est demandée via un POST ;
- ligne 9 : la chaîne postée. C'est un paramètre nommé [num] qui est posté. Sa valeur est le n° de page (ligne 4) à afficher dans la vue actuelle ;
- ligne 10 : le serveur va renvoyer du HTML, celui de la page à afficher ;
- lignes 13-15 : en cas de succès (statut HTTP égal à 200), le HTML envoyé par le serveur est mis dans la zone d'id [content] ;
- lignes 18-20 : en cas d'échec (statut HTTP égal à 500), le HTML envoyé par le serveur est mis dans la zone d'id [content] ;

7.4.6 L'action [/ajax-07]

Le code de l'action [/ajax-07] est le suivant :

```
1.  @RequestMapping(value = "/ajax-07", method = RequestMethod.POST, produces = "text/html; charset=UTF-8")
```

```
2.      public String ajax07(int num) {
3.         // num : numéro de page
4.         switch (num) {
5.         case 1:
6.            return "vue-07";
7.         case 2:
8.            return "vue-08";
9.         default:
10.           return "vue-07";
11.        }
12.     }
```

- ligne 2 : on récupère le paramètre posté qui s'appelle [num]. On rappelle que le paramètre ligne 2 doit porter le nom du paramètre posté, ici [num]. [num] est un n° de page ou de vue ;
- lignes 5-6 : dans le cas où [num==1], on renvoie la vue [vue-07.xml] ;
- lignes 7-8 : dans le cas où [num==2], on renvoie la vue [vue-08.xml] ;
- lignes 9-10 : dans les autres cas, (impossible normalement), on renvoie la vue [vue-07.xml] ;

7.4.7 La vue [vue-08.xml]

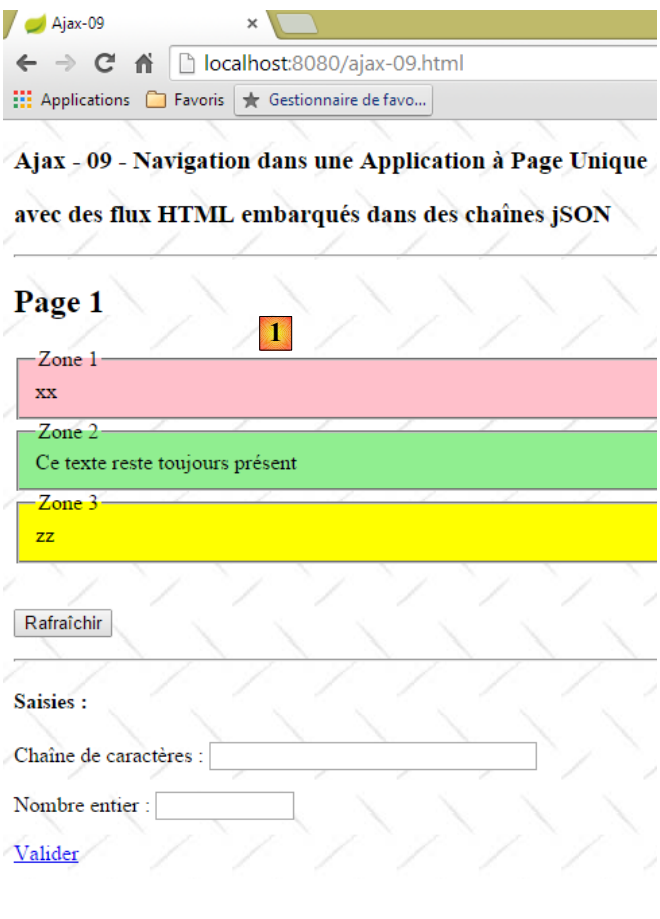
La vue [vue-08.xml] forme la page n° 2 de l'application :

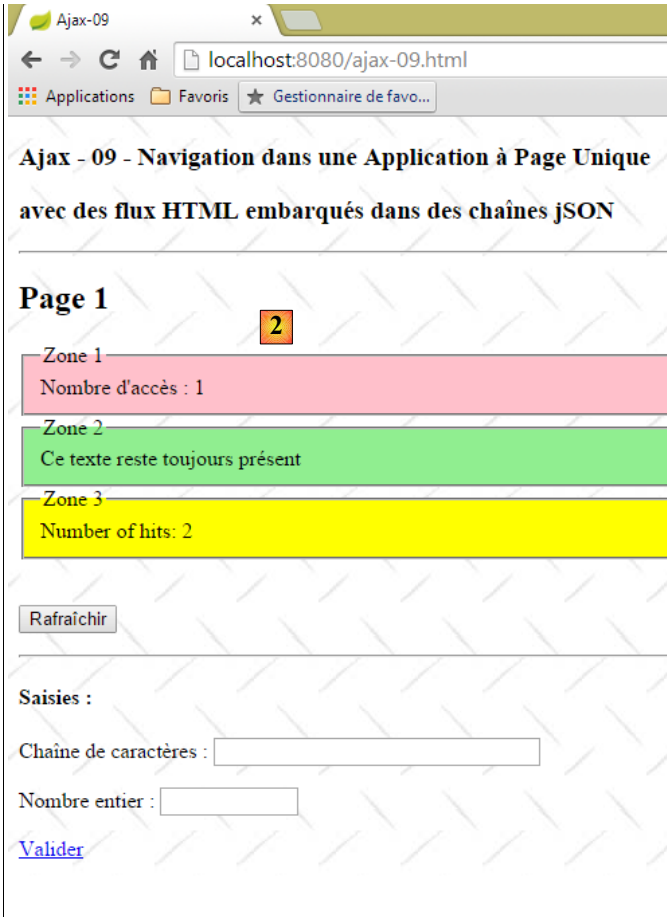
```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h4>Page 2</h4>
5.        <p>
6.           <a href="javascript:gotoPage(1)">Page 1</a>
7.        </p>
8.     </body>
9.  </html>
```

7.5 Embarquer plusieurs flux HTML dans une réponse jSON

7.5.1 Introduction

Nous considérons l'application suivante :

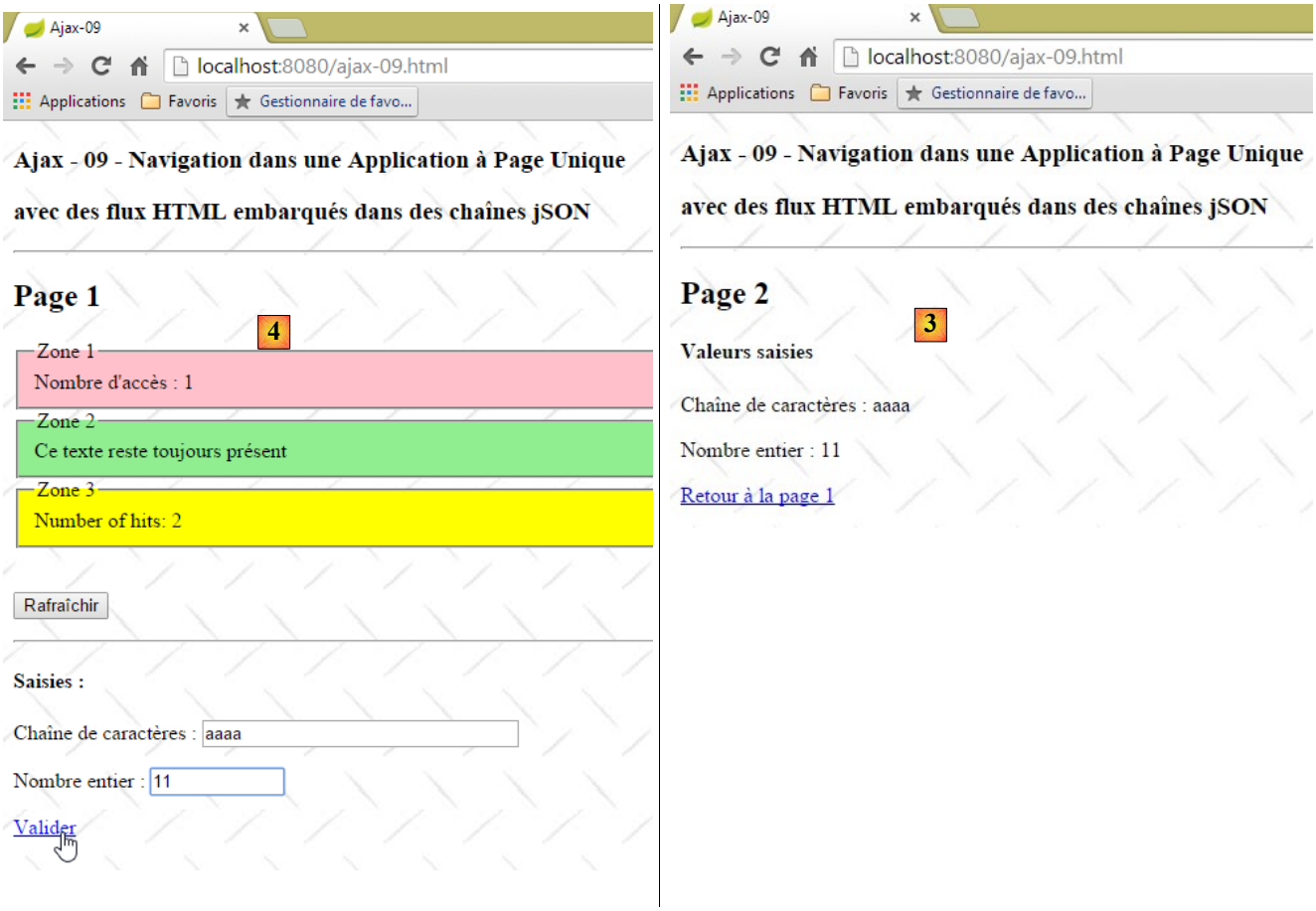




La page [1] a quatre zones :

- [Zone 1, Zone 3] sont des zones qui apparaissent / disparaissent sur un clic sur le bouton [Rafraîchir]. On compte le nombre d'apparitions de chacune de ces deux zones [2]. La zone [Zone 1] utilise la langue française alors que la zone [Zone 3] utilise la langue anglaise ;
- la zone [Zone 2] est présente en permanence ;
- la zone [Saisies] est présente en permanence ;

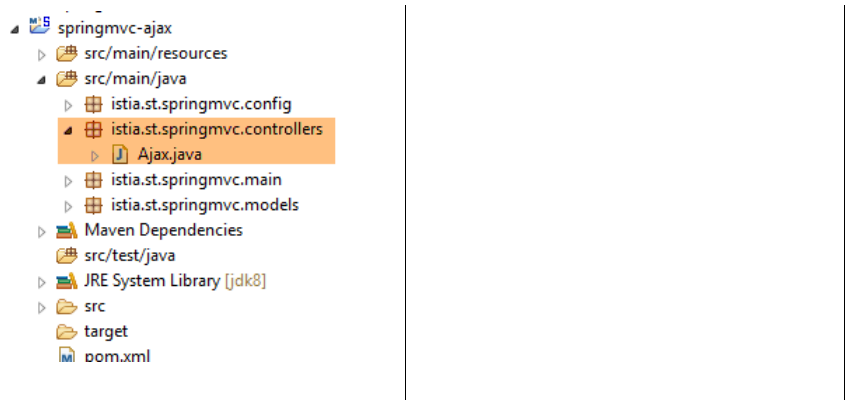
Le lien [Valider] affiche la page suivante [3] :



- le lien [Retour à la page 1] ramène la page n° 1 dans l'état où elle était [4] ;

L'application est à page unique. La première page est demandée au serveur par le navigateur. Les suivantes sont obtenues auprès du serveur par des appels Ajax.

7.5.2 L'action [/ajax-09]

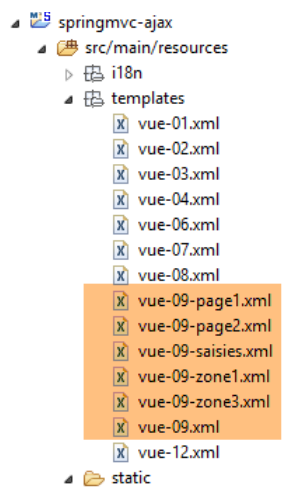


L'action [/ajax-09] est la suivante :

```
1.      @RequestMapping(value = "/ajax-09", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.      public String ajax09() {
3.          return "vue-09";
4.  }
```

Elle se contente d'afficher la vue [vue-09.xml].

7.5.3 Les vues XML



La vue [vue-09.xml] est la page maître de l'application :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>Ajax-09</title>
6.        <link rel="stylesheet" href="/css/ajax01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
8.        <script type="text/javascript" src="/js/json3.js"></script>
9.        <script type="text/javascript" src="/js/local9.js"></script>
10.    </head>
11.    <body>
12.       <h3>Ajax - 09 - Navigation dans une Application à Page Unique</h3>
13.       <h3>avec des flux HTML embarqués dans des chaînes jSON</h3>
14.       <hr />
15.       <div id="content" th:include="vue-09-page1" />
16.       <img id="loading" src="/images/loading.gif" />
17.       <div id="erreur" style="background-color:lightgrey"></div>
18.    </body>
19. </html>
```

- ligne 9 : le fichier JS utilisé dans l'application ;
- ligne 15 : le contenu de la page maître ;
- ligne 16 : une image animée d'attente :
- ligne 17 : zone pour afficher une éventuelle erreur ;

La vue [vue-09-page1.xml] est la page 1 de l'application :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h2>Page 1</h2>
5.        <!-- zone 1 -->
6.        <fieldset id="zone1" style="background-color:pink">
7.           <legend>Zone 1</legend>
8.           <span id="zone1-content" th:text="xx">xx</span>
9.        </fieldset>
10.       <!-- zone 2 -->
11.       <fieldset id="zone2" style="background-color:lightgreen">
12.          <legend>Zone 2</legend>
13.          <span>Ce texte reste toujours présent</span>
14.       </fieldset>
15.       <!-- zone 3 -->
16.       <fieldset id="zone3" style="background-color:yellow">
17.          <legend>Zone 3</legend>
18.          <span id="zone3-content" th:text="zz">zz</span>
```

```
19.       </fieldset>
20.       <br />
21.       <p>
22.          <button onclick="javascript:postForm()">Rafraîchir</button>
23.       </p>
24.       <hr />
25.       <div id="saisies" th:include="vue-09-saisies">
26.       </div>
27.    </body>
28. </html>
```

- lignes 6-9 : la zone [Zone 1]. Son contenu est placé dans le composant [id="zone1-content"] ;
- lignes 11-14 : la zone [Zone 2] qui ne change pas ;
- lignes 16-19 : la zone [Zone 3]. Son contenu est placé dans le composant [id="zone3-content"] ;
- ligne 22 : la fonction JS qui poste le formulaire ;
- ligne 25 : inclusion de la zone de saisies ;

On notera que la page 1 n'a pas de balise [form]. Tout va être traité en javascript.

La vue [vue-09-saisies.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <div id="saisies">
4.        <h4>Saisies :</h4>
5.        <p>
6.           Chaîne de caractères :
7.           <input type="text" id="text1" size="30" th:value="${value1}" />
8.        </p>
9.        <p>
10.          Nombre entier :
11.          <input type="text" id="text2" size="10" th:value="${value2}" />
12.       </p>
13.       <p>
14.          <a href="javascript:valider()">Valider</a>
15.       </p>
16.    </div>
17. </html>
```

- lignes 5-8 : saisie d'une chaîne de caractères ;
- lignes 13-16 : saisie d'un nombre entier ;
- ligne 14 : la fonction JS qui poste les valeurs saisies ;

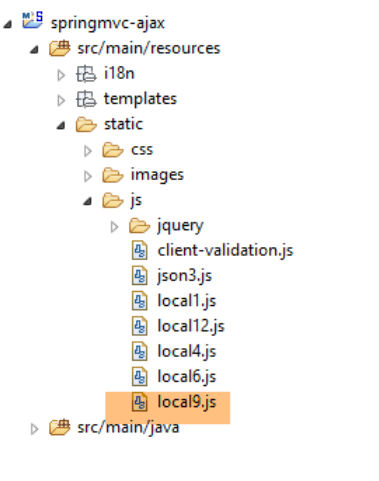
De nouveau, on notera que la zone de saisies n'a pas de balise [form].

Au total, la page n° 1 présente deux fonctionnalités :

- [Rafraîchir] : qui rafraîchit les zones 1 et 3. Cette action est traitée par le serveur qui renvoie aléatoirement :
  - la zone 1 avec son compteur d'accès et rien pour la zone 3,
  - la zone 3 avec son compteur d'accès et rien pour la zone 1,
  - les deux zones avec leurs compteurs d'accès ;
- [Valider] : qui affiche la page 2 avec les valeurs saisies ou bien un message d'erreur si les données saisies sont invalides ;

Nous allons nous intéresser d'abord au bouton [Rafraîchir].

7.5.4 Le code JS de gestion du bouton [Rafraîchir]



Le code du fichier [local9.js] est le suivant :

```
1. // variables globales
2. var content;
3. var loading;
4. var erreur;
5.
6. // au chargement du document
7. $(document).ready(function() {
8.    // on récupère les références des différents composants de la page
9.    loading = $("#loading");
10.   loading.hide();
11.   erreur = $("#erreur");
12.   erreur.hide();
13.   content = $("#content");
14. });
```

- lignes 9-13 : lorsque la page maître est chargée, on mémorise les références sur les trois composants identifiés par [loading, erreur, content] ;
- lignes 2-4 : les références de ces trois composants sont mémorisées dans des variables globales. Elles restent fixes parce que les trois zones concernées sont toujours présentes dans la page affichée, ceci quelque soit le moment. Parce qu'elles restent fixes elles peuvent être calculées dans [`$(document).ready`] et partagées avec les autres fonctions du fichier JS ;

La fonction [postForm] gère le clic sur le bouton [Rafraîchir] :

```
1.  function postForm() {
2.     console.log("postForm");
3.     // on fait un appel Ajax à la main
4.     $.ajax({
5.        url : '/ajax-10',
6.        headers : {
7.           'Accept' : 'application/json'
8.        },
9.        type : 'POST',
10.       dataType : 'json',
11.       beforeSend : onBegin,
12.       success : onSuccess,
13.       error : onError,
14.       complete : onComplete
15.    })
16. }
```

- lignes 4-15 : l'appel Ajax au serveur ;
- ligne 5 : c'est l'action [ajax-10] qui va traiter le POST ;
- lignes 6-8 : la réponse va être du jSON. Le client JS indique qu'il accepte les documents jSON ;
- ligne 9 : l'action [ajax-10] est appelée avec une opération POST ;
- igne 10 : on va recevoir du jSON ;
- ligne 11 : la fonction exécutée avant l'appel Ajax ;
- ligne 12 : la fonction exécutée à réception de la réponse du serveur, lorsque celle-ci est un succès [200 OK] ;

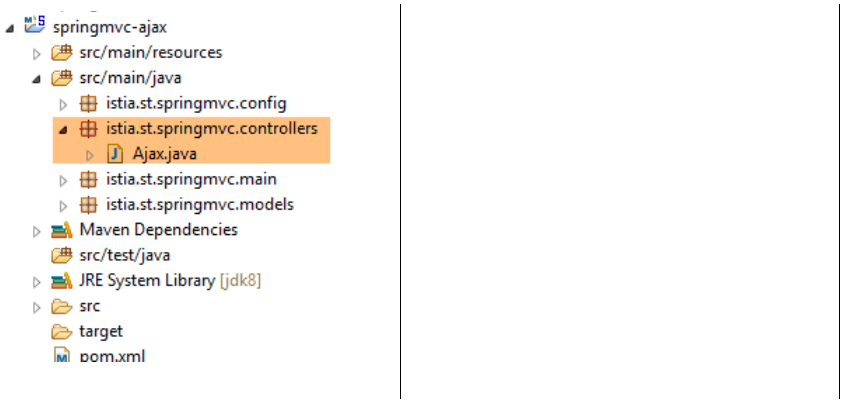
- ligne 13 : la fonction exécutée à réception de la réponse du serveur, lorsque celle-ci est un échec [500 Internal server error, ...] ;
- ligne 14 : la fonction exécutée après réception de la réponse ;

La fonction [onBegin] est la suivante :

```
1.  // avant l'appel Ajax
2.  function onBegin() {
3.      console.log("onBegin");
4.      // image d'attente
5.      loading.show();
6.  }
```

Elle se contente de mettre en route l'image animée de l'attente du résultat du serveur.

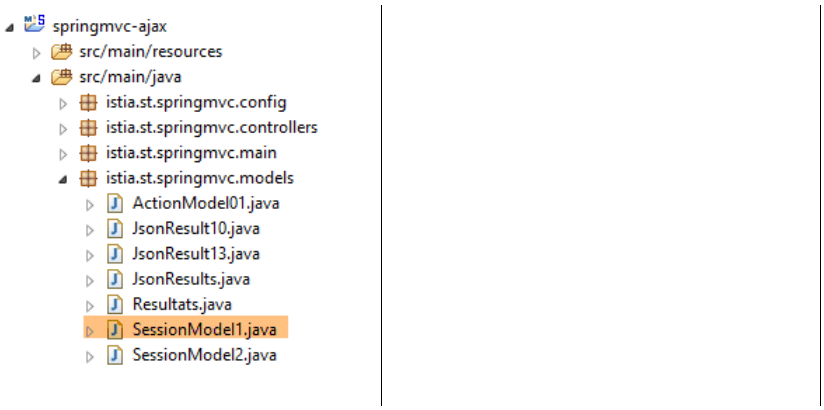
7.5.5 L'action [/ajax-10]



L'action [/ajax-10] est la suivante :

```
1.  // la session
2.      @Autowired
3.      private SessionModel1 session;
4.      // le moteur Thymeleaf / Spring
5.      @Autowired
6.      private SpringTemplateEngine engine;
7.
8.      @RequestMapping(value = "/ajax-10", method = RequestMethod.POST)
9.      @ResponseBody()
10.     public JsonResult10 ajax10(HttpServletRequest request, HttpServletResponse response) {
11.     ...
12.     }
```

- ligne 3 : on injecte la session. Celle-ci a le type [SessionModel1] suivant :




```
1.  package istia.st.springmvc.models;
2.
```

```
3.  import java.io.Serializable;
4.
5.  import org.springframework.context.annotation.Scope;
6.  import org.springframework.context.annotation.ScopedProxyMode;
7.  import org.springframework.stereotype.Component;
8.
9.  @Component
10. @Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
11. public class SessionModel1 implements Serializable {
12.
13.     private static final long serialVersionUID = 1L;
14.     // deux compteurs
15.     private int cpt1 = 0;
16.     private int cpt3 = 0;
17.     // les trois zones
18.     private String zone1 = "xx";
19.     private String zone3 = "zz";
20.     private String saisies;
21.     private boolean zone1Active = true;
22.     private boolean zone3Active = true;
23.
24.     // getters et setters
25.     ...
26. }
```

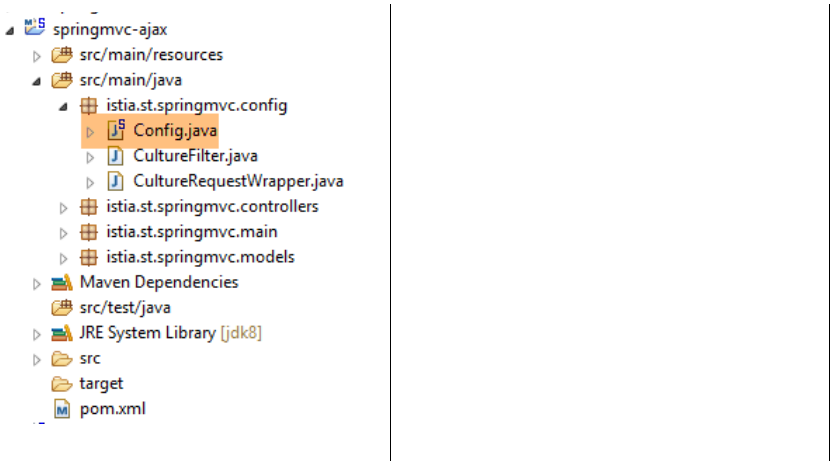
La session [SessionModel1] mémorise les éléments suivants :

- ligne 15 : le nombre de fois [cpt1] où la zone [Zone 1] est affichée ;
- ligne 16 : le nombre de fois [cpt3] où la zone [Zone 3] est affichée ;
- lignes 18-20 : les flux HTML des zones [Zone 1], [Zone 3] et [Saisies]. Ceci est nécessaire dans la séquence [Page 1] --> [Page 2] --> [Page 1]. Lorsqu'on passe de [Page 2] à [Page 1], il faut restaurer [Page 1] et donc ses trois zones ;
- lignes 21-22 : deux booléens qui indiquent si les zones [Zone 1] et [Zone 3] sont affichées (visibles) ;

L'autre élément injecté dans le contrôleur [AjaxController] est le suivant :

```
1.      // le moteur Thymeleaf / Spring
2.      @Autowired
3.  private SpringTemplateEngine engine;
```

Le bean de type [SpringTemplateEngine] est défini dans le fichier de configuration [Config] :



Il est défini de la façon suivante :

```
1.      @Bean
2.      public SpringResourceTemplateResolver templateResolver() {
3.         SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
4.         templateResolver.setPrefix("classpath:/templates/");
5.         templateResolver.setSuffix(".xml");
6.         templateResolver.setTemplateMode("HTML5");
7.         templateResolver.setCacheable(true);
8.         templateResolver.setCharacterEncoding("UTF-8");
```

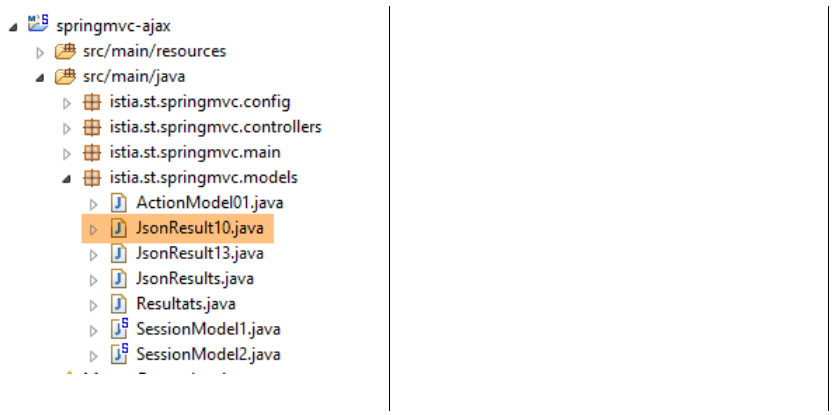
```
9.          return templateResolver;
10.     }
11.
12.     @Bean
13.     SpringTemplateEngine templateEngine(SpringResourceTemplateResolver templateResolver) {
14.         SpringTemplateEngine templateEngine = new SpringTemplateEngine();
15.         templateEngine.setTemplateResolver(templateResolver);
16.         return templateEngine;
17. }
```

- lignes 2-10 : nous connaissons le bean de type [SpringResourceTemplateResolver] qui nous permet de définir certaines caractéristiques des vues ;
- lignes 13-17 : le bean de type [SpringTemplateEngine] nous permet de définir le " moteur " de vues, la classe chargée de générer les réponses [Thymeleaf] aux clients. [Thymeleaf] a un " moteur " par défaut et un autre lorsqu'il est utilisé dans un environnement [Spring]. C'est ce dernier que nous utilisons ici ;

La signature de l'action [/ajax-10] est la suivante :

```
1.  @RequestMapping(value = "/ajax-10", method = RequestMethod.POST)
2.      @ResponseBody()
3.      public JsonResult10 ajax10(HttpServletRequest request, HttpServletResponse response) {
4.      ...
5.  }
```

- ligne 1 : l'action [/ajax-10] n'accepte qu'un POST ;
- ligne 2 : l'action [/ajax-10] rend elle-même la réponse au client. Celle-ci sera tranformée automatiquement en jSON ;
- ligne 3 : la réponse est de type [JsonResult10] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  public class JsonResult10 {
4.
5.      // data
6.      private String content;
7.      private String zone1;
8.      private String zone3;
9.      private String erreur;
10.     private String saisies;
11.     private boolean zone1Active;
12.     private boolean zone3Active;
13.
14.     public JsonResult10() {
15.     }
16.
17.     // getters et setters
18. ...
19. }
```

- ligne 6 : le contenu HTML de la zone identifiée par [content] ;
- ligne 7 : le contenu HTML de la zone [Zone 1] ;
- ligne 8 : le contenu HTML de la zone [Zone 3] ;
- ligne 9 : le contenu HTML de la zone [Erreur] ;

- ligne 10 : le contenu HTML de la zone [Saisies] ;
- ligne 11 : booléen indiquant si la zone [Zone 1] doit être affichée ;
- ligne 12 : booléen indiquant si la zone [Zone 3] doit être affichée ;

Le code de l'action [/ajax-10] est le suivant :

```
1.  @RequestMapping(value = "/ajax-10", method = RequestMethod.POST)
2.     @ResponseBody()
3.     public JsonResult10 ajax10(HttpServletRequest request, HttpServletResponse response) {
4.        // contexte Thymeleaf
5.        WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
6.        // réponse
7.        JsonResult10 result = new JsonResult10();
8.        // session
9.        session.setZone1(null);
10.       session.setZone3(null);
11.       session.setZone1Active(false);
12.       session.setZone3Active(false);
13.       // on rend une réponse aléatoire
14.       int cas = new Random().nextInt(3);
15.       switch (cas) {
16.       case 0:
17.          // zone 1 active
18.          setZone1(thymeleafContext, result);
19.          return result;
20.       case 1:
21.          // zone 3 active
22.          setZone3(thymeleafContext, result);
23.          return result;
24.       case 2:
25.          // zones 1 et 3 actives
26.          setZone1(thymeleafContext, result);
27.          setZone3(thymeleafContext, result);
28.          return result;
29.       }
30.       return null;
31.    }
```

- ligne 5 : nous récupérons le contexte [Thymeleaf]. Nous verrons ultérieurement à quoi il va nous servir ;
- ligne 7 : nous créons une réponse vide pour l'instant ;
- lignes 9-12 : nous mettons à [null] les deux zones contenues dans la session et nous indiquons qu'elles ne doivent pas être affichées. Ces deux zones vont être bientôt générées mais il est possible que seule l'une d'entre-elles le soit ;
- lignes 14-29 : les deux zones sont générées ;
- lignes 17-19 : seule la zone [Zone 1] est générée ;
- lignes 21-23 : seule la zone [Zone 3] est générée ;
- lignes 25-28 : les deux zones [Zone 1] et [Zone 3] sont générées ;

Le flux HTML de la zone [Zone 1] est généré par la méthode suivante :

```
1.     private void setZone1(WebContext thymeleafContext, JsonResult10 result) {
2.        // zone 1 active
3.        // flux HTML
4.        int cpt1 = session.getCpt1() + 1;
5.        thymeleafContext.setVariable("cpt1", cpt1);
6.        thymeleafContext.setLocale(new Locale("fr", "FR"));
7.        String zone1 = engine.process("vue-09-zone1", thymeleafContext);
8.        result.setZone1(zone1);
9.        result.setZone1Active(true);
10.       // session
11.       session.setCpt1(cpt1);
12.       session.setZone1(zone1);
13.       session.setZone1Active(true);
14. }
```

- ligne 1 : les paramètres sont :
  - le contexte [Thymeleaf] de type [WebContext],
  - la réponse au client en cours de construction de type [JsonResult10] ;
- ligne 3 : on incrémente le compteur [cpt1] de la session qui compte le nombre de fois où la zone [Zone 1] est affichée ;

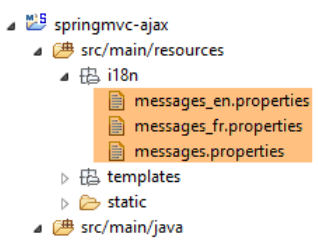
- ligne 4 : le contexte [Thymeleaf] de type [WebContext] se comporte un peu comme le modèle [Model] de Spring MVC. Pour ajouter un élément au modèle, on utilise [WebContext.setVariable]. Ici, on met donc le compteur [cpt1] dans le modèle [Thymeleaf]. Cela va permettre d'évaluer l'expression Thymeleaf [${cpt1}]
- ligne 5 : le contexte [Thymeleaf] a une locale. Cela lui permet d'évaluer les expression du type [#{clé_msg}]. Ici, on associe le contexte Thymeleaf a une locale française ;
- ligne 6 : c'est l'instruction la plus intéressante. Le moteur Thymeleaf va traiter la vue [vue-09-zone1.xml] avec le modèle et la locale que l'on vient de calculer et au lieu d'envoyer le flux HTML résultant au client, il le rend en tant que chaîne de caractères ;
- lignes 7-9 : le flux HTML de la zone [Zone 1] qui vient d'être calculé est mémorisé dans la session et dans le résultat qui va être envoyé au client. Par ailleurs, on indique que la zone [Zone 1] doit être affichée ;
- lignes 11-13 : on mémorise dans la session, les informations concernant la zone [Zone 1] afin d'être capables de la régénérer ;

La ligne 7 traite la vue [vue-09-zone1.xml] suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <span th:text="#{message.zone}"></span>
4.     <span th:text="${cpt1}"></span>
5.  </html>
```

- ligne 3 : l'expression [*#{message.zone}*] va être évaluée grâce à la locale ;
- ligne 4 : l'expression [*${cpt1}*] va être évaluée grâce au modèle Thymeleaf ;

Le message de clé [message.zone] est défini dans les fichiers de messages [messages_fr.properties] et [messages_en.properties] :



[messages_fr.properties]

```
message.zone=Nombre d'accès :
```

[messages_en.properties]

```
message.zone=Number of hits:
```

Le flux HTML de la zone [Zone 3] est généré par une méthode analogue :

```
1.     private void setZone3(WebContext thymeleafContext, JsonResult10 result) {
2.        // zone 3 active
3.        // flux HTML
4.        int cpt3 = session.getCpt3() + 1;
5.        thymeleafContext.setVariable("cpt3", cpt3);
6.        thymeleafContext.setLocale(new Locale("en", "US"));
7.        String zone3 = engine.process("vue-09-zone3", thymeleafContext);
8.        result.setZone3(zone3);
9.        result.setZone3Active(true);
10.       // session
11.       session.setCpt3(cpt3);
12.       session.setZone3(zone3);
13.       session.setZone3Active(true);
14.    }
```

- ligne 6 : la locale de la zone [Zone 3] est la locale anglaise ;

7.5.6 Traitement de la réponse de l'action [/ajax-10]

Revenons au code JS de [local9.js] qui va traiter la réponse du serveur :

```
1. // à réception de la réponse du serveur
2. // en cas de succès
3. function onSuccess(data) {
4.    console.log("onSuccess");
5.    // contenu
6.    if (data.content) {
7.       content.html(data.content);
8.    }
9.    // zone 1
10.   if (data.zone1Active) {
11.      $("#zone1").show();
12.      if (data.zone1) {
13.         $("#zone1-content").html(data.zone1);
14.      }
15.   } else {
16.      $("#zone1").hide();
17.   }
18.   // zone 3 active ?
19.   if (data.zone3Active) {
20.      $("#zone3").show();
21.      if (data.zone3) {
22.         $("#zone3-content").html(data.zone3);
23.      }
24.   } else {
25.      $("#zone3").hide();
26.   }
27.   // saisies ?
28.   if (data.saisies) {
29.      $("#saisies").html(data.saisies);
30.   }
31.   // erreur ?
32.   if (data.erreur) {
33.      erreur.text(data.erreur);
34.      erreur.show();
35.   } else {
36.      erreur.hide();
37.   }
38. }
```

Rappelons la structure Java de la réponse reçue ligne 3 dans la variable [data] :

```
1.  public class JsonResult10 {
2.
3.     // data
4.     private String content;
5.     private String zone1;
6.     private String zone3;
7.     private String erreur;
8.     private String saisies;
9.     private boolean zone1Active;
10.    private boolean zone3Active;
11.
12. }
```

- lignes 6-8 : si [data.content!=null], alors on initialise la zone [id=content] avec. Cette zonne représente [Page 1] ou [Page 2] dans sa totalité. Dans la démonstration présente, on a [data.content==null] et donc la zone [id=content] ne sera pas modifiée et continuera à afficher [Page 1] ;
- lignes 10-17 : affichage [Zone 1] si [data.zone1Active==true]. Si de plus [data.zone1!=null] alors le contenu de [Zone 1] est modifié sinon il reste ce qu'il était ;
- lignes 19-26 : même chose pour [Zone 3] ;

- lignes 28-30 : si on a [data.saisies!=null] alors la zone [Saisies] est régénérée. Dans la démonstration présente, on a [data.saisies==null] et donc la zone [Saisies] reste ce qu'elle était ;
- lignes 32-37 : raisonnement analogue pour la zone [Erreur] avec les nuances suivantes :
  - ligne 33 : [data.erreur] sera un message d'erreur au format texte ;
  - ligne 36 : si [data.erreur==null] alors la zone [Erreur] est cachée. En effet, elle a pu être affichée lors de la précédente requête ;

En cas d'erreur côté serveur (HTTP status du genre 500 Internal server error), la fonction suivante est exécutée :

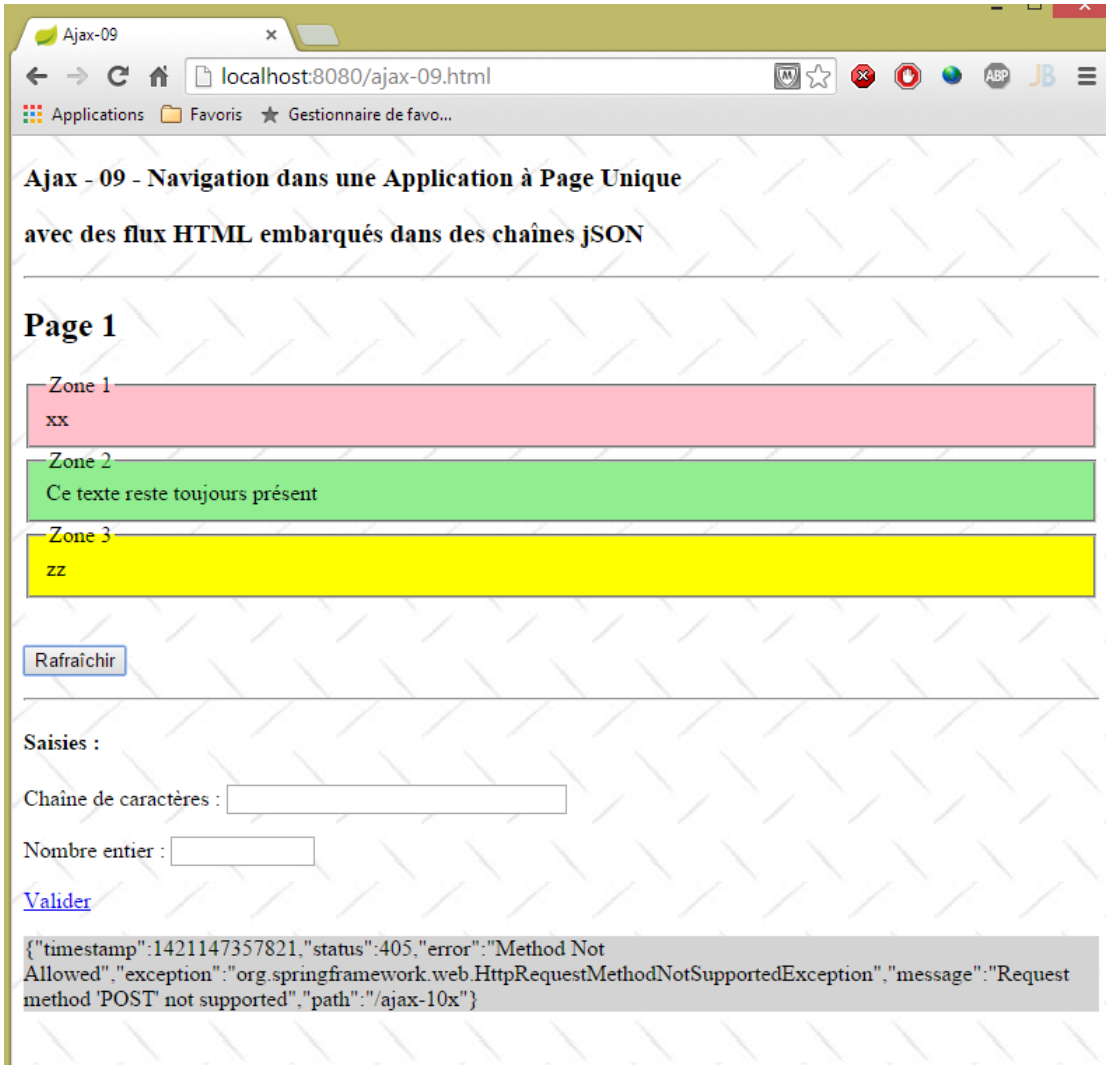
```
1.  // à réception de la réponse du serveur
2.  // en cas d'échec
3.  function onError(jqXHR) {
4.     console.log("onError");
5.     // erreur système
6.     erreur.text(jqXHR.responseText);
7.     erreur.show();
8.  }
```

Pour voir une telle erreur, modifions la fonction [postForm] de la façon suivante :

```
1.  function postForm() {
2.     console.log("postForm");
3.     // on récupère des références sur la page courante
4.     ...
5.     // on fait un appel Ajax à la main
6.     $.ajax({
7.        url : '/ajax-10x',
8.        ...
9.     })
10. }
```

- ligne 7 : on met une URL qui n'existe pas ;

Voici les résultats lorsqu'on clique sur le bouton [Rafraîchir] :



Il est intéressant de voir que l'erreur a été envoyée elle également sous la forme d'une chaîne jSON.

La méthode exécutée après réception de la réponse du serveur est la suivante :

```
1.  // après [onSuccess, onError]
2.  function onComplete() {
3.     console.log("onComplete");
4.     // image d'attente
5.     loading.hide();
6.  }
```

On se contente de cacher l'image animée de l'attente.

7.5.7 Affichage de la page [Page 2]

Le code HTML du lien [Valider] est le suivant :

```
<a href="javascript:valider()">Valider</a>
```

La fonction JS [valider] est la suivante :

```
1.  // validation des valeurs saisies
2.  function valider() {
3.     // valeur postée
4.     var post = JSON3.stringify({
5.        "value1" : $("#text1").val().trim(),
6.        "value2" : $("#text2").val().trim()
7.     });
8.     // on fait un appel Ajax à la main
```

```
9.      $.ajax({
10.        url : '/ajax-11A',
11.        headers : {
12.           'Accept' : 'application/json',
13.           'Content-Type' : 'application/json'
14.        },
15.        type : 'POST',
16.        data : post,
17.        dataType : 'json',
18.        beforeSend : onBegin,
19.        success : onSuccess,
20.        error : onError,
21.        complete : onComplete
22.     })
23. }
```

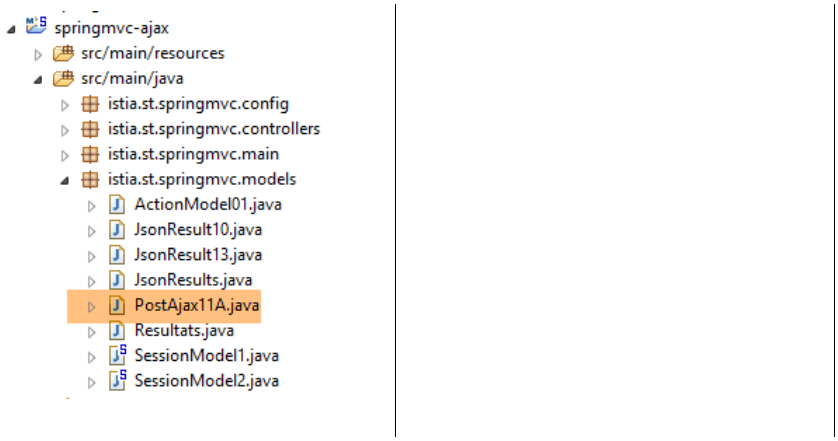
- lignes 4-7 : nous avons deux valeurs v1 et v2 à poster : celles des composants de saisie identifiés par [#text1] et [#text2]. Nous allons faire quelque chose de nouveau. Nous allons poster ces deux valeurs sous la forme d'une chaîne jSON {"value1":v1,"value2":v2} ;
- ligne 10 : les valeurs postées seront envoyées à l'action [ajax-11A] ;
- ligne 12 : parce qu'on sait qu'on va recevoir une réponse jSON, on indique qu'on peut en recevoir du jSON ;
- ligne 13 : on indique au serveur qu'on va lui envoyer la valeur postée sous la forme d'une chaîne jSON ;
- lignes 15-16 : on fait un POST de la valeur à poster ;
- ligne 17 : on va recevoir du jSON ;

7.5.8 L'action [ajax-11A]

L'action [ajax-11A] qui traite la chaîne jSON postée est la suivante :

```
1.  @RequestMapping(value = "/ajax-11A", method = RequestMethod.POST, consumes = "application/json")
2.     @ResponseBody
3.     public JsonResult10 ajax11A(@RequestBody @Valid PostAjax11A post, BindingResult bindingResult, Locale
    locale,   HttpServletRequest request, HttpServletResponse response) {
4.        ...
5.     }
```

- ligne 1 : on indique avec ["application/json"] que l'action attend un document sous forme jSON. Ce document est la valeur postée par le client ;
- ligne 3 : la valeur postée va être récupérée dans l'objet [PostAjax11A post] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import javax.validation.constraints.NotNull;
4.  import javax.validation.constraints.Size;
5.
6.  import org.hibernate.validator.constraints.Range;
7.
8.  public class PostAjax11A {
9.
10.    // data
11.    @Size(min = 4, max = 6)
12.    @NotNull
```

```
13.     private String value1;
14.     @Range(min = 10, max = 14)
15.     @NotNull
16.     private Integer value2;
17.
18.     // getters et setters
19.     ...
20. }
```

- la structure de l'objet [PostAjax11A] doit reprendre la structure de l'objet posté {"value1":v1,"value2":v2}. Il faut donc un champ [value1] (ligne 13) et [value2] (ligne 16) ;
- on a mis des contraintes d'intégrité sur les deux champs ;

Revenons au code de l'action [ajax-11A] :

```
1.  @RequestMapping(value = "/ajax-11A", method = RequestMethod.POST, consumes = "application/json")
2.      @ResponseBody
3.      public JsonResult10 ajax11A(@RequestBody @Valid PostAjax11A post, BindingResult bindingResult,
    Locale locale,  HttpServletRequest request, HttpServletResponse response) {
4.          // contexte Thymeleaf
5.          WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
6.          // réponse
7.          JsonResult10 result = new JsonResult10();
8.          // post valide ?
9.          if (bindingResult.hasErrors()) {
10.             // on renvoie la page 1 avec une erreur
11.             result.setZone1Active(session.isZone1Active());
12.             result.setZone3Active(session.isZone3Active());
13.             result.setErreur(getErreursForModel(bindingResult));
14.             return result;
15.         }
16.         ...
17. }
```

- ligne 3 : l'annotation [@RequestBody] désigne le document envoyé par le client. Il s'agit de la valeur postée en jSON par celui-ci. Celle-ci va donc être utilisée pour construire l'objet [PostAjax11A] ;
- ligne 3 : l'annotation [@Valid] force la validation de la valeur postée ;
- ligne 9 : si la validation échoue :
  - ligne 13 : on renvoie un message d'erreur,
  - lignes 11-12 : les zones 1 et 3 sont remises dans l'état où elles étaient (affichées ou non) ;

Le calcul du message d'erreur est fait de la façon suivante :

```
1.      private String getErreursForModel(BindingResult result) {
2.          StringBuffer buffer = new StringBuffer();
3.          for (FieldError error : result.getFieldErrors()) {
4.              StringBuffer bufferCodes = new StringBuffer("(");
5.              for (String code : error.getCodes()) {
6.                  bufferCodes.append(String.format("%s ", code));
7.              }
8.              bufferCodes.append(")");
9.              buffer.append(String.format("[%s:%s:%s:%s]", error.getField(), error.getRejectedValue(),
    bufferCodes,
10.                     error.getDefaultMessage()));
11.         }
12.         return buffer.toString();
13. }
```

C'est une fonction qu'on a déjà rencontrée.

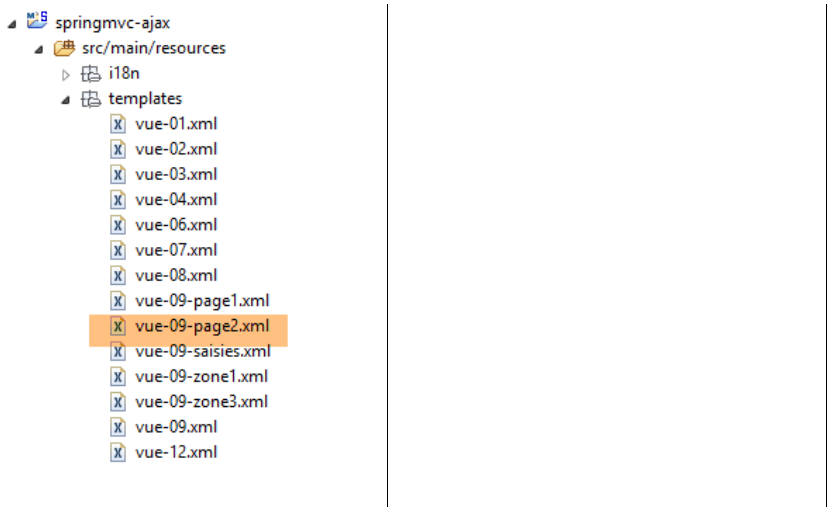
L'action [ajax-11A] se poursuit de la façon suivante :

```
1.  @RequestMapping(value = "/ajax-11A", method = RequestMethod.POST, consumes = "application/json")
2.      @ResponseBody
3.      public JsonResult10 ajax11A(@RequestBody @Valid PostAjax11A post, BindingResult bindingResult, Locale
    locale,  HttpServletRequest request, HttpServletResponse response) {
4.          // contexte Thymeleaf
5.          WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
6.          // réponse
7.          JsonResult10 result = new JsonResult10();
```

```
8.          // post valide ?
9.          if (bindingResult.hasErrors()) {
10.     ...
11.         }
12.         // on mémorise la zone de saisie
13.         thymeleafContext.setVariable("value1", post.getValue1());
14.         thymeleafContext.setVariable("value2", post.getValue2());
15.         session.setSaisies(engine.process("vue-09-saisies", thymeleafContext));
16.         // on envoie la page 2
17.         result.setContent(engine.process("vue-09-page2", thymeleafContext));
18.         return result;
19. }
```

- lignes 13-14 : les valeurs postées sont mises dans le contexte Thymeleaf ;
- ligne 15 : avec ce contexte, on calcule la vue [vue-09-saisies] et on la met dans la session afin de pouvoir la régénérer ultérieurement ;
- ligne 17 : la page 2 est mise dans le résultat qui va être envoyé au client ;

La vue [vue-09-page2.xml] est la suivante :




```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h2>Page 2</h2>
5.        <p>
6.           <h4>Valeurs saisies :</h4>
7.           <p>
8.              Chaîne de caractères :
9.              <span th:text="${value1}"></span>
10.          </p>
11.          <p>
12.             Nombre entier :
13.             <span th:text="${value2}"></span>
14.          </p>
15.          <a href="javascript:retourPage1()">Retour à la page 1</a>
16.       </p>
17.    </body>
18. </html>
```

- lignes 9 et 13, on affiche les valeurs [value1, value2] que l'action [/ajax-11A] a placées dans le contexte Thymeleaf ;

7.5.9 Traitement de la réponse de l'action [/ajax-11A]

Côté client, la réponse de l'action [/ajax-10] est traitée par la fonction [onSuccess] :

```
1.  function onSuccess(data) {
2.     console.log("onSuccess");
3.     // contenu
4.     if (data.content) {
5.        content.html(data.content);
```

```
6.      }
7.      // zone 1
8.      if (data.zone1Active) {
9.         $("#zone1").show();
10.        if (data.zone1) {
11.           $("#zone1-content").html(data.zone1);
12.        }
13.     } else {
14.        $("#zone1").hide();
15.     }
16.     // zone 3 active ?
17.     if (data.zone3Active) {
18.        $("#zone3").show();
19.        if (data.zone3) {
20.           $("#zone3-content").html(data.zone3);
21.        }
22.     } else {
23.        $("#zone3").hide();
24.     }
25.     // saisies ?
26.     if (data.saisies) {
27.        $("#saisies").html(data.saisies);
28.     }
29.     // erreur ?
30.     if (data.erreur) {
31.        erreur.text(data.erreur);
32.        erreur.show();
33.     } else {
34.        erreur.hide();
35.     }
36. }
```

Nous avons déjà commenté ce code. Considérons les deux cas, réponse avec ou sans erreur :

Avec erreur

Dans ce cas, l'action [/ajax-11A] a envoyé une réponse jSON de la forme **{"zone1":null, "zone3":null,"saisies":null,"erreur":erreur,"zone1Active":zone1Active,"zone3Active":zone3Active,"content":null }**. Si on suit le code ci-dessus, on voit que :

- la zone [content] ne change pas. Elle contenait la page n° 1 ;
- la zone [Erreur] est affichée ;
- les zones [Zone 1], [Zone 3], [Saisies] sont laissées dans l'état où elles étaient ;

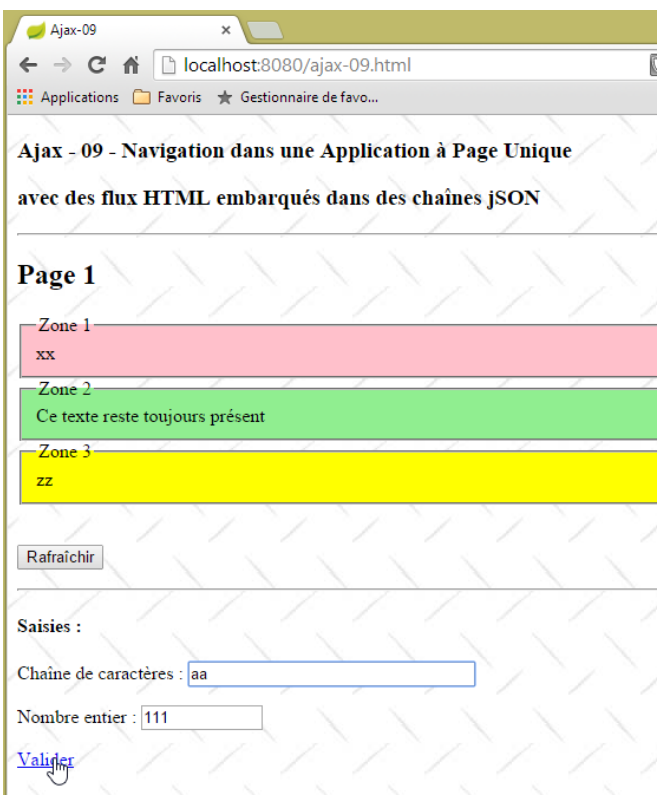
Sans erreur

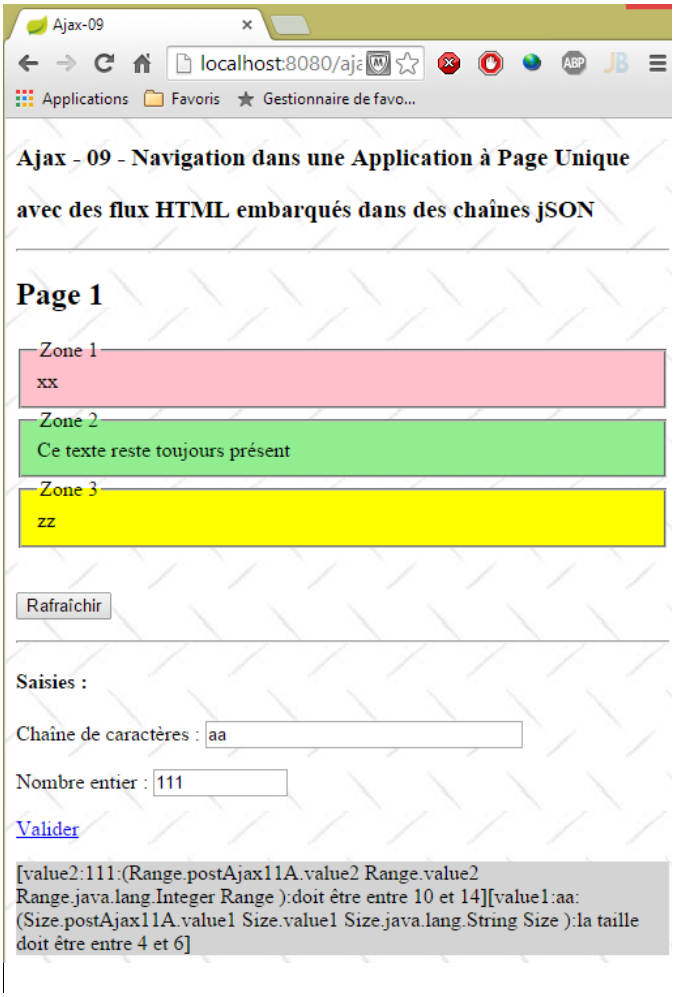
Dans ce cas, l'action [/ajax-11A] a envoyé une réponse jSON de la forme **{"zone1":null, "zone3":null,"saisies":null,"erreur":null,"zone1Active":false,"zone3Active":false,"content":content}**. Si on suit le code ci-dessus, on voit que :

- la zone [content] est affichée. Elle contient la page n° 2 ;

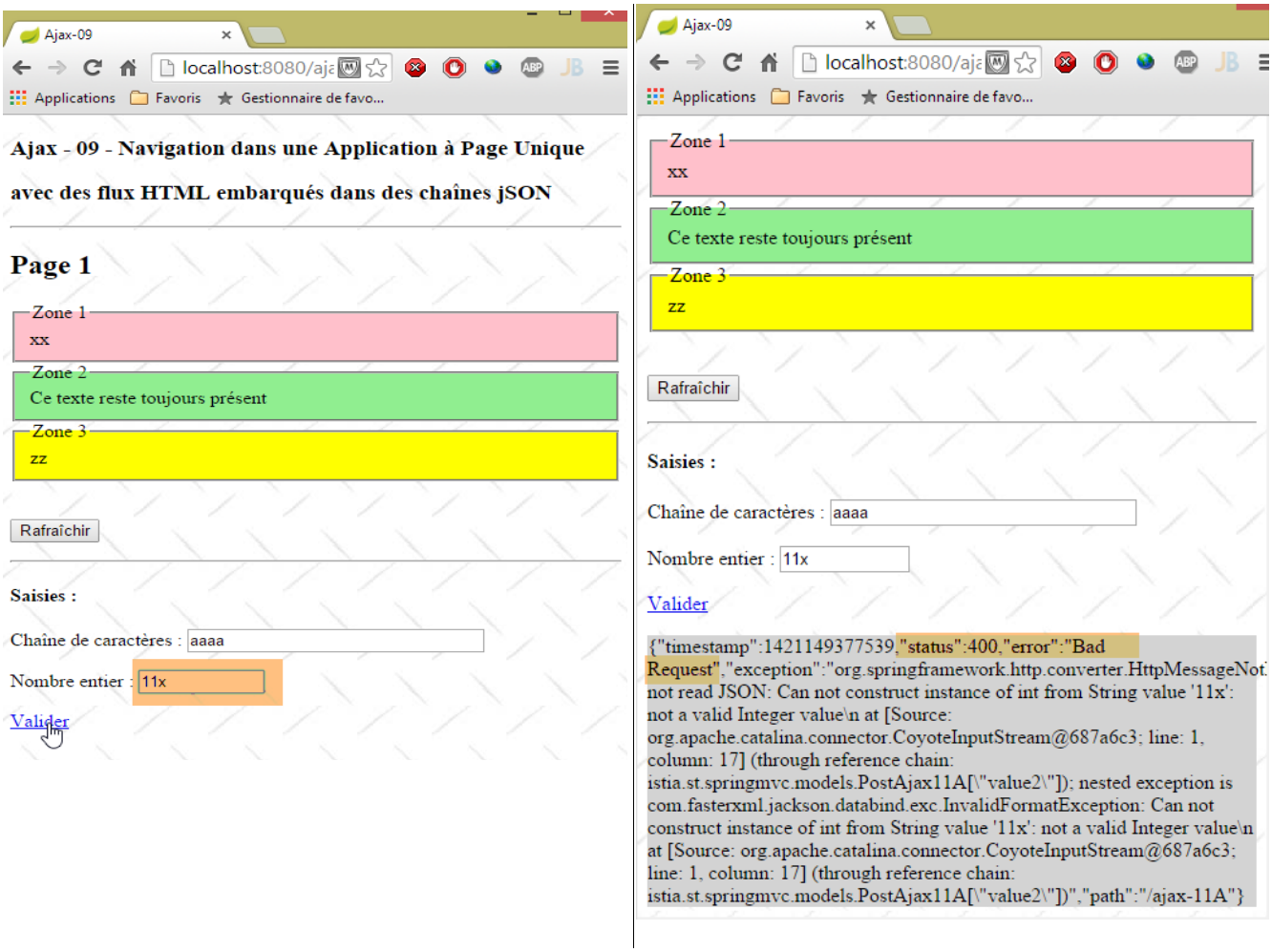
Voici trois exemples d'exécution :

Un cas avec erreur de validation :



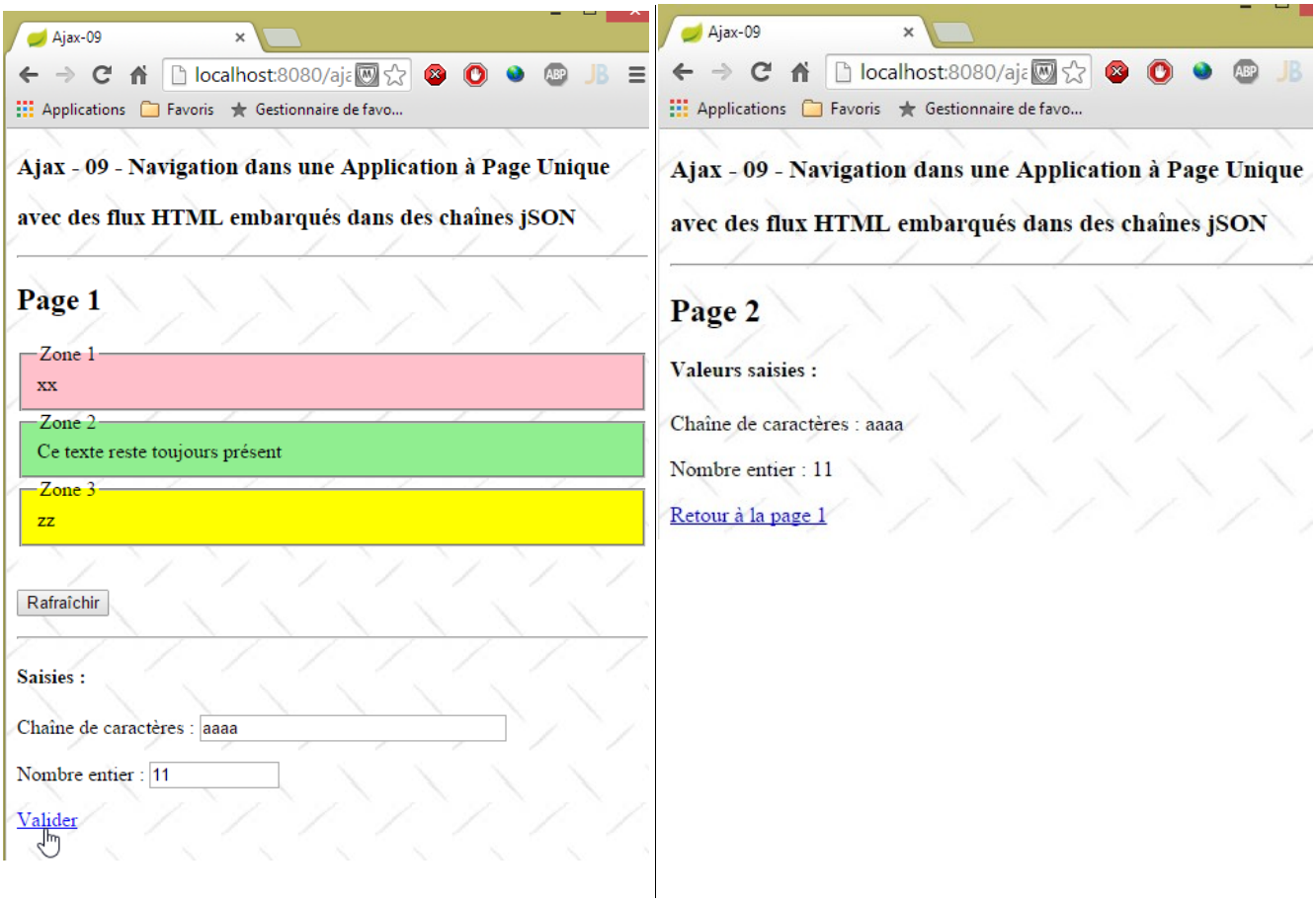


Un cas avec erreur de POST :



Ce type d'erreur est différent. Parce que Spring n'a pas pu convertir la chaîne jSON en type [PostAjax11A], il a renvoyé une réponse HTTP avec [status=400]. L'action [ajax-11A] n'a pas été exécutée ;

Un cas sans erreur :



7.5.10 Retour vers la page n° 1

Le lien [Retour vers la page 1] dans la page N° 2 est le suivant :

```
<a href="javascript:retourPage1()">Retour à la page 1</a>
```

La méthode JS [retourPage1] est la suivante :

```
1.  // retour page 1
2.  function retourPage1() {
3.     // on fait un appel Ajax à la main
4.     $.ajax({
5.        url : '/ajax-11B',
6.        headers : {
7.           'Accept' : 'application/json',
8.        },
9.        type : 'POST',
10.       dataType : 'json',
11.       beforeSend : onBegin,
12.       success : onSuccess,
13.       error : onError,
14.       complete : onComplete
15.    })
16. }
```

Elle fait un POST, sans valeur postée, vers l'action [/ajax-11B].

7.5.11 L'action [/ajax-11B]

L'action [/ajax-11B] est la suivante :

```
1.     @RequestMapping(value = "/ajax-11B", method = RequestMethod.POST)
```

```
2.      @ResponseBody
3.      public JsonResult10 ajax11B(HttpServletRequest request, HttpServletResponse response) {
4.          // contexte Thymeleaf
5.          WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
6.          // réponse
7.          JsonResult10 result = new JsonResult10();
8.          // on la rend la page 1 dans son état originel
9.          result.setContent(engine.process("vue-09-page1", thymeleafContext));
10.         result.setSaisies(session.getSaisies());
11.         result.setZone1(session.getZone1());
12.         result.setZone3(session.getZone3());
13.         result.setZone1Active(session.isZone1Active());
14.         result.setZone3Active(session.isZone3Active());
15.         return result;
16. }
```

L'action doit régénérer la page n°1 avec ses trois zones [Zone1, Zone3, Erreur] :

- ligne 9 : la page n° 1 est mise dans le résultat ;
- ligne 10 : la zone des saisies est mise dans le résultat ;
- ligne 11 : la zone [Zone 1] est mise dans le résultat ;
- ligne 12 : la zone [Zone 3] est mise dans le résultat ;
- lignes 13-14 : on met l'état des zones [Zone 1] et [Zone 3] dans le résultat ;

7.5.12 Traitement de la réponse de l'action [/ajax-11B]

La réponse de l'action [/ajax-11B] est traitée par la fonction [onSuccess] :

```
1.  function onSuccess(data) {
2.      console.log("onSuccess");
3.      // contenu
4.      if (data.content) {
5.          content.html(data.content);
6.      }
7.      // zone 1
8.      if (data.zone1Active) {
9.          $("#zone1").show();
10.         if (data.zone1) {
11.             $("#zone1-content").html(data.zone1);
12.         }
13.     } else {
14.         $("#zone1").hide();
15.     }
16.     // zone 3 active ?
17.     if (data.zone3Active) {
18.         $("#zone3").show();
19.         if (data.zone3) {
20.             $("#zone3-content").html(data.zone3);
21.         }
22.     } else {
23.         $("#zone3").hide();
24.     }
25.     // saisies ?
26.     if (data.saisies) {
27.         $("#saisies").html(data.saisies);
28.     }
29.     // erreur ?
30.     if (data.erreur) {
31.         erreur.text(data.erreur);
32.         erreur.show();
33.     } else {
34.         erreur.hide();
35.     }
36. }
```

L'action [/ajax-11B] a envoyé une réponse jSON de la forme {"zone1":zone1, "zone3":zone3,"saisies":saisies,"erreur":null,"zone1Active":zone1Active,"zone3Active":zone3Active,"content":content}. Si on suit le code ci-dessus, on voit que :

- la zone [content] est modifiée. Elle contenait la page n° 2. Elle va désormais contenir la page n° 1 ;
- la zone [Erreur] est cachée ;
- les zones [Zone 1], [Zone 3], [Saisies] sont affichées dans l'état où elles étaient ;

7.6 Gérer la session côté client

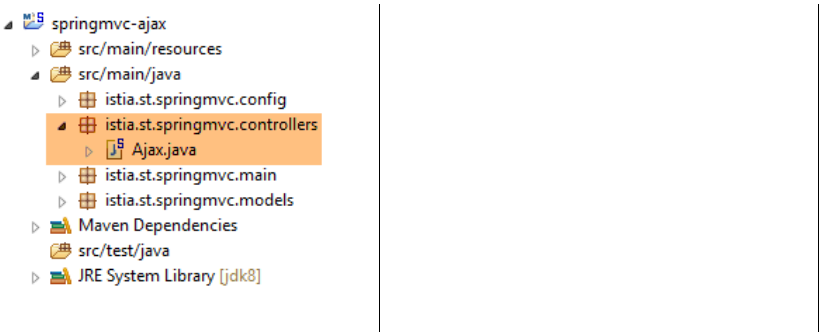
7.6.1 Introduction

Dans le paragraphe précédent, nous avons géré une session dont la structure était la suivante :

```
1.  public class SessionModel1 implements Serializable {
2.
3.      // deux compteurs
4.      private int cpt1 = 0;
5.      private int cpt3 = 0;
6.      // les trois zones
7.      private String zone1 = "xx";
8.      private String zone3 = "zz";
9.      private String saisies;
10.     private boolean zone1Active = true;
11.     private boolean zone3Active = true;
12. ...
13. }
```

Lorsqu'il y a de très nombreux utilisateurs, la mémoire occupée par les sessions de tous ces utilisateurs peut poser problème. La règle est donc de minimiser la taille de celle-ci. Le modèle APU (Application à Page Unique) permet de gérer la session côté client et d'avoir un serveur web sans session. En effet, la page unique est chargée initialement par le navigateur. Avec elle, arrive le fichier Javascript qui l'accompagne. Comme il n'y a pas de rechargement de page, ce fichier JS va rester en permanence au sein du navigateur tel qu'il a été chargé initialement. On peut alors utiliser ses variables globales pour y stocker de l'information sur les différentes actions de l'utilisateur. C'est ce que nous allons voir maintenant. Nous allons non seulement gérer la session côté client mais repenser l'application JS afin de solliciter le moins possible le serveur.

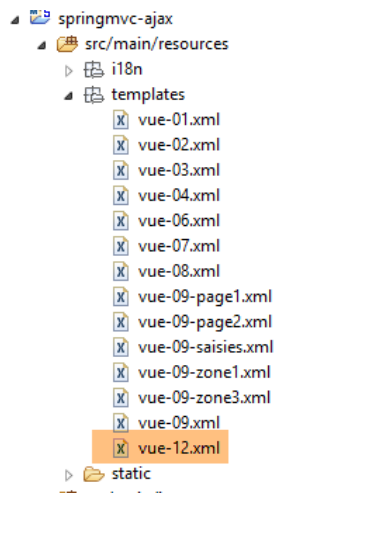
7.6.2 L'action [/ajax-12]



L'action [/ajax-12] est la suivante :

```
1.      @RequestMapping(value = "/ajax-12", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.      public String ajax12() {
3.          return "vue-12";
4.  }
```

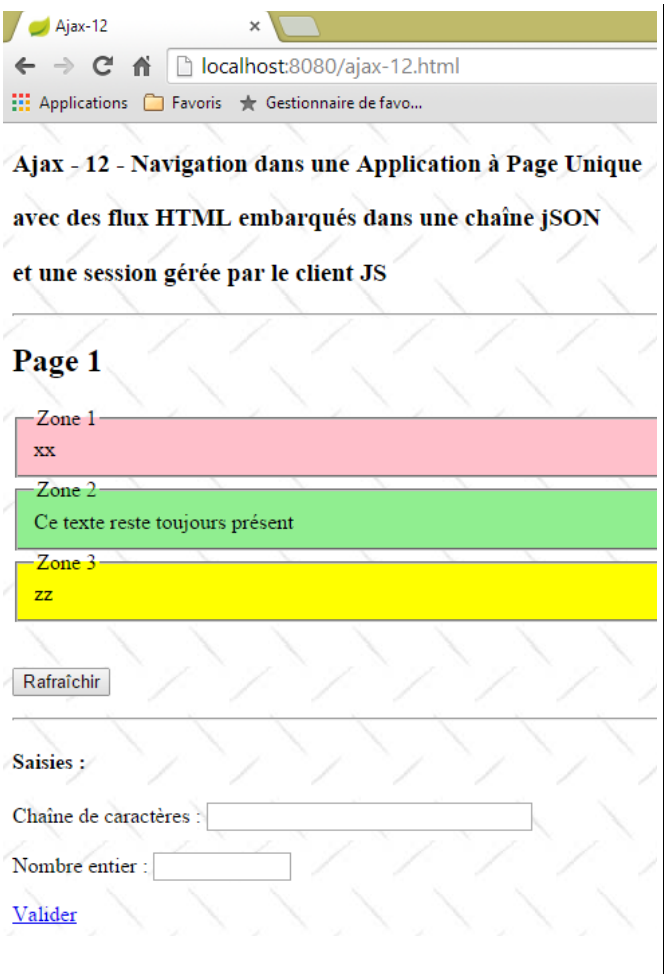
La vue [vue-12.xml] est la suivante :




```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>Ajax-12</title>
6.        <link rel="stylesheet" href="/css/ajax01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
8.        <script type="text/javascript" src="/js/json3.js"></script>
9.        <script type="text/javascript" src="/js/local12.js"></script>
10.    </head>
11.    <body>
12.       <h3>Ajax - 12 - Navigation dans une Application à Page Unique</h3>
13.       <h3>avec des flux HTML embarqués dans une chaîne jSON</h3>
14.       <h3>et une session gérée par le client JS</h3>
15.       <hr />
16.       <div id="content" th:include="vue-09-page1" />
17.       <img id="loading" src="/images/loading.gif" />
18.       <div id="erreur" style="background-color:lightgrey"></div>
19.    </body>
20. </html>
```

- cette vue est identique à la vue [vue-09] à la différence près du script JS utilisé en ligne 9 ;

La vue affichée est la suivante :



7.6.3 Le code JS de gestion du bouton [Rafraîchir]

Le code du fichier [local12.js] est le suivant :

```
1. // variables globales
2. var content;
3. var loading;
4. var erreur;
5. var page1;
6. var page2;
7. var value1;
```

```
8.  var value2;
9.  var session = {
10.       "cpt1" : 0,
11.       "cpt3" : 0
12.    };
13.
14. // au chargement du document
15. $(document).ready(function() {
16.    // on récupère les références des différents composants de la page
17.    loading = $("#loading");
18.    loading.hide();
19.    erreur = $("#erreur");
20.    erreur.hide();
21.    content = $("#content");
22. });
```

- lignes 17-21 : lorsque la page maître est chargée, on mémorise les références des trois composants identifiés par [loading, erreur, content] dans les variables globales des lignes 2-4 ;
- lignes 5-6 : pour mémoriser les deux pages ;
- lignes 7-8 : pour mémoriser les deux valeurs postées par le lien [Valider] ;
- ligne 9 : la session. Elle mémorise côté client les valeurs des compteurs [cpt1, cpt3] ;

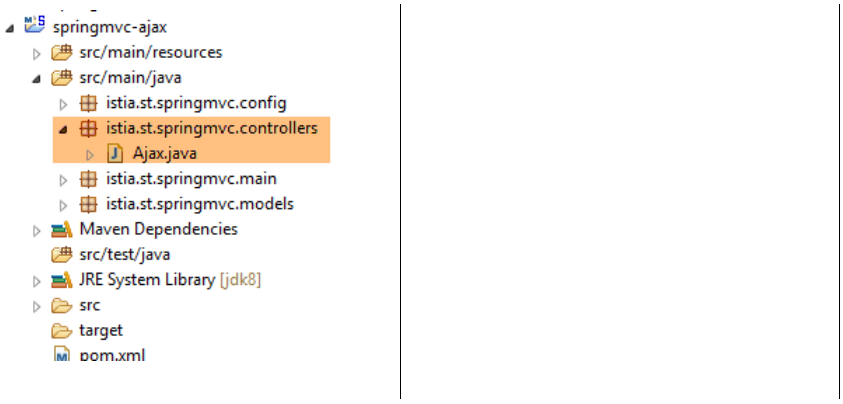
La fonction [postForm] gère le clic sur le bouton [Rafraîchir] :

```
1.  function postForm() {
2.     console.log("postForm");
3.     // on poste la session
4.     var post = JSON3.stringify(session);
5.     // on fait un appel Ajax à la main
6.     $.ajax({
7.        url : '/ajax-13',
8.        headers : {
9.           'Accept' : 'application/json',
10.          'Content-Type' : 'application/json'
11.       },
12.       type : 'POST',
13.       data : post,
14.       dataType : 'json',
15.       beforeSend : onBegin,
16.       success : function(data) {
17.          ...
18.       },
19.       error : onError,
20.       complete : onComplete
21.    })
22.    }
```

Les diffrences avec la version précédente sont les suivantes :

- l'URL de la ligne 7 est différente ;
- ligne 4 : on poste une valeur alors qu'auparavant on n'en postait pas. Cette valeur est la chaîne jSON de la session. Le principe est le suivant :
  - le client envoie la session au serveur,
  - celui-ci la modifie et lui renvoie,
  - le client mémorise la nouvelle session ;
- ligne 10 : on envoie un document au format jSON (valeur postée) ;
- ligne 13 : on a quelque chose à poster ;
- lignes 15-20 : les fonctions [beforeSend, error, complete] sont celles de la version précédente. Seule la fonction [success] change (lignes 16-18) ;

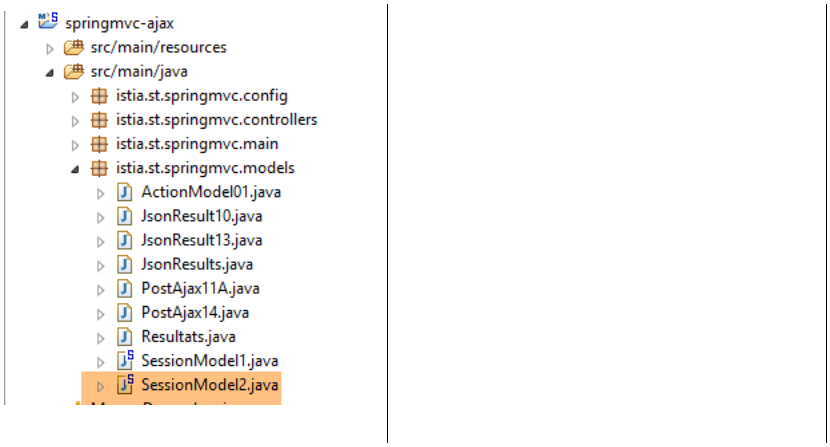
7.6.4 L'action [/ajax-13]



L'action [/ajax-13] est la suivante :

```
1.      @RequestMapping(value = "/ajax-13", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.      @ResponseBody()
3.      public JsonResult13 ajax13(@RequestBody SessionModel2 session2, HttpServletRequest request,
        HttpServletResponse response) {
4.      ...
5.  }
```

- ligne 3 : le paramètre [@RequestBody SessionModel2 session2] récupère la session postée par le client. Celle-ci a le type [SessionModel2] suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  import java.io.Serializable;
4.
5.  public class SessionModel2 implements Serializable {
6.
7.      private static final long serialVersionUID = 1L;
8.      // deux compteurs
9.      private int cpt1 = 0;
10.     private int cpt3 = 0;
11.
12.     // getters et setters
13.     ...
14. }
```

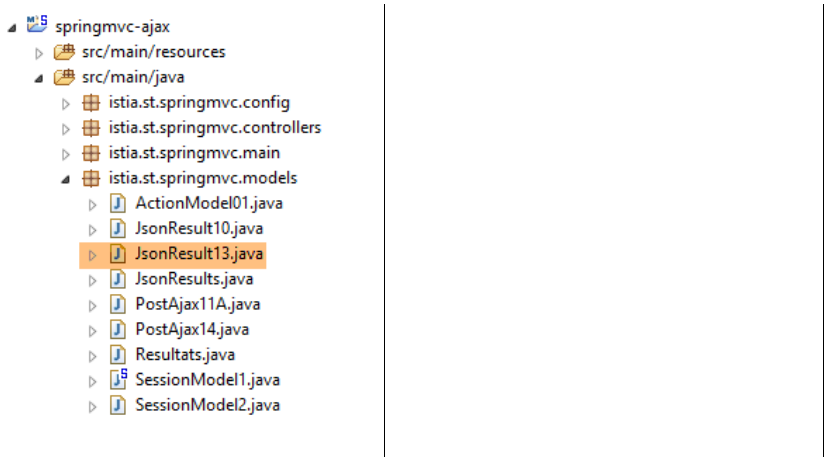
La session [SessionModel2] mémorise les éléments suivants :

- ligne 9 : le nombre de fois [cpt1] où la zone [Zone 1] est affichée ;
- ligne 10 : le nombre de fois [cpt3] où la zone [Zone 3] est affichée ;

Poursuivons l'étude du code de l'action [/ajax-13] :

```
1.     @RequestMapping(value = "/ajax-13", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.     @ResponseBody()
3.     public JsonResult13 ajax13(@RequestBody SessionModel2 session2, HttpServletRequest request,
       HttpServletResponse response) {
4.     ...
5.  }
```

- ligne 3, le type [JsonResult13] de la réponse est le suivant :




```
1.  package istia.st.springmvc.models;
2.
3.  public class JsonResult13 {
4.
5.     // data
6.     private String page2;
7.     private String zone1;
8.     private String zone3;
9.     private String erreur;
10.    private String value1;
11.    private Integer value2;
12.
13.    // session
14.    private SessionModel2 session;
15.
16.    // getters et setters
17.    ...
18. }
```

- ligne 14 : la session. Le serveur la renvoie au client pour mémorisation ;
- ligne 6 : le contenu HTML de la page n° 2 ;
- ligne 7 : le contenu HTML de la zone [Zone 1] ;
- ligne 8 : le contenu HTML de la zone [Zone 3] ;
- ligne 9 : le message d'erreur éventuel ;
- lignes 10-11 : deux informations calculées par le serveur et affichées par la page n° 2 ;

Poursuivons l'étude du code de l'action [/ajax-13] :

```
1.  @RequestMapping(value = "/ajax-13", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.     @ResponseBody()
3.     public JsonResult13 ajax13(@RequestBody SessionModel2 session2, HttpServletRequest request,
4.           HttpServletResponse response) {
5.        // contexte Thymeleaf
6.        WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
7.        // réponse
8.        JsonResult13 result = new JsonResult13();
9.        result.setSession(session2);
10.       // on rend une réponse aléatoire
11.       int cas = new Random().nextInt(3);
12.       switch (cas) {
13.       case 0:
```

```
14.          // zone 1 active
15.          setZone1B(thymeleafContext, result);
16.          return result;
17.       case 1:
18.          // zone 3 active
19.          setZone3B(thymeleafContext, result);
20.          return result;
21.       case 2:
22.          // zones 1 et 3 actives
23.          setZone1B(thymeleafContext, result);
24.          setZone3B(thymeleafContext, result);
25.          return result;
26.       }
27.       return null;
28.       }
```

- ligne 9 : la session est mise dans le résultat de l'action ;

La méthode [setZone1B] qui active la zone [Zone 1] est la suivante :

```
1.    private void setZone1B(WebContext thymeleafContext, JsonResult13 result) {
2.       // on récupère la session
3.       SessionModel2 session = result.getSession();
4.       // zone 1 active
5.       // flux HTML
6.       int cpt1 = session.getCpt1() + 1;
7.       thymeleafContext.setVariable("cpt1", cpt1);
8.       thymeleafContext.setLocale(new Locale("fr", "FR"));
9.       String zone1 = engine.process("vue-09-zone1", thymeleafContext);
10.      result.setZone1(zone1);
11.      // session
12.      session.setCpt1(cpt1);
13. }
```

- ligne 3 : on récupère la session. Elle va être modifiée ligne 12 avec le nouveau compteur [cpt1]. On rappelle que cette session va être renvoyée au client ;
- ligne 10 : la nouvelle zone [Zone 1] ;

La méthode [setZone3B] qui active la zone [Zone 3] est analogue :

```
1.  private void setZone3B(WebContext thymeleafContext, JsonResult13 result) {
2.       // on récupère la session
3.       SessionModel2 session = result.getSession();
4.       // zone 3 active
5.       // flux HTML
6.       int cpt3 = session.getCpt3() + 1;
7.       thymeleafContext.setVariable("cpt3", cpt3);
8.       thymeleafContext.setLocale(new Locale("en", "US"));
9.       String zone3 = engine.process("vue-09-zone3", thymeleafContext);
10.      result.setZone3(zone3);
11.      // session
12.      session.setCpt3(cpt3);
13.      }
```

7.6.5 Traitement de la réponse de l'action [/ajax-13]

Côté client, la réponse jSON de l'action [/ajax-13] est traitée par la fonction [onSuccess] suivante :

```
1.  function postForm() {
2.     console.log("postForm");
3.     // on poste la session
4.     var post = JSON3.stringify(session);
5.     // on fait un appel Ajax à la main
6.     $.ajax({
7.     ...
8.        success : function(data) {
9.           // on mémorise la session
10.          session = data.session;
```

```
11.          // on met à jour les deux zones
12.          if (data.zone1) {
13.             $("#zone1-content").html(data.zone1);
14.             $("#zone1").show();
15.          } else {
16.             $("#zone1").hide();
17.          }
18.          if (data.zone3) {
19.             $("#zone3").show();
20.             $("#zone3-content").html(data.zone3);
21.          } else {
22.             $("#zone3").hide();
23.          }
24.       },
25. ...
26.    })
27.    }
```

- lignes 12-17 : si le serveur a mis quelque chose dans le champ [zone1] de la réponse, alors il faut régénérer la zone [Zone 1] et l'afficher, sinon elle doit être cachée ;
- lignes 18-23 : même raisonnement pour la zone [Zone 3] ;

7.6.6 Affichage de la page [Page 2]

Le code HTML du lien [Valider] est le suivant :

```
<a href="javascript:valider()">Valider</a>
```

La fonction JS [valider] est la suivante :

```
1.  // validation des valeurs saisies
2.  function valider() {
3.     // on mémorise la page 1
4.     page1 = content.html();
5.     // on mémorise les valeurs saisies
6.     value1 = $("#text1").val().trim();
7.     value2 = $("#text2").val().trim();
8.     // valeur postée
9.     var post = JSON3.stringify({
10.       "value1" : value1,
11.       "value2" : value2,
12.       "pageRequired" : page2 ? false : true
13.    });
14.    // on fait un appel Ajax à la main
15.    $.ajax({
16.       url : '/ajax-14',
17.       headers : {
18.          'Accept' : 'application/json',
19.          'Content-Type' : 'application/json'
20.       },
21.       type : 'POST',
22.       data : post,
23.       dataType : 'json',
24.       beforeSend : onBegin,
25.       success : function(data) {
26.       ...
27.       },
28.       error : onError,
29.       complete : onComplete
30.    })
31. }
```

- on va faire un POST qui normalement va nous faire passer à la page n° 2 ;
- ligne 4 : on mémorise la page n° 1 afin de pouvoir y revenir ultérieurement ;
- lignes 6-7 : l'opération précédente ne mémorise pas les valeurs saisies, juste le code HTML de la page. Aussi mémorise-t-on maintenant les deux valeurs saisies dans le formulaire ;
- lignes 9-13 : les deux valeurs saisies sont mises dans une chaîne jSON. C'est elle qui sera postée ;
- ligne 12 : un paramètre pour indiquer au serveur si on a besoin de la page n° 2. Nous allons procéder ainsi. Nous allons demander la page n° 2 une première fois, puis la mémoriser dans la variable JS [page2]. Ensuite, nous ne la redemanderons plus. Nous utiliserons la page en cache. Ligne 2, [pageRequired] vaut [true] si la variable [page2] ne contient rien, [false] sinon ;

- on notera que la session n'est pas postée. En effet, celle-ci mémorise des compteurs que l'action [/ajax-14] de la ligne 20 ne modifie pas ;

7.6.7 L'action [/ajax-14]

L'action [/ajax-14] est la suivante :

```
1.  @RequestMapping(value = "/ajax-14", method = RequestMethod.POST)
2.     @ResponseBody
3.     public JsonResult13 ajax14(@RequestBody @Valid PostAjax14 post, BindingResult bindingResult, Locale
    locale,   HttpServletRequest request, HttpServletResponse response) {
4.         ...
5.     }
```

- ligne 3 : la réponse est toujours de type [JsonResult13] ;
- ligne 3 : la valeur postée est encapsulée dans le type [PostAjax14] suivant :

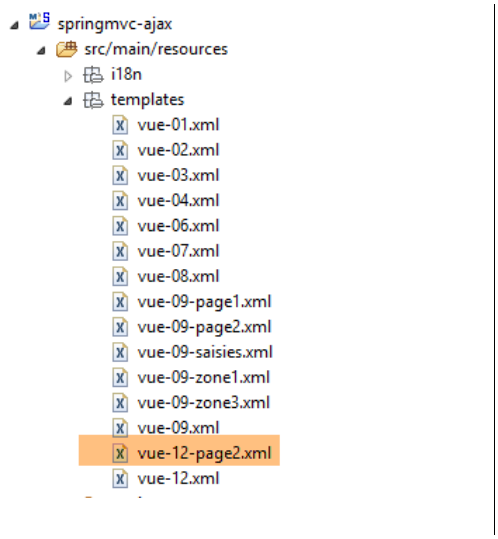
```
1.  package istia.st.springmvc.models;
2.
3.  public class PostAjax14 extends PostAjax11A {
4.
5.     // page 2
6.     private boolean pageRequired;
7.
8.     // getters et setters
9.     ...
10. }
```

- ligne 3 : la classe [PostAjax14] étend la classe [PostAjax11A] de la version précédente. Elle a donc une structure [value1, value2, pageRequired] ;

L'action [/ajax-14] se poursuit de la façon suivante :

```
1.     @RequestMapping(value = "/ajax-14", method = RequestMethod.POST)
2.     @ResponseBody
3.  public JsonResult13 ajax14(@RequestBody @Valid PostAjax14 post, BindingResult bindingResult, Locale
    locale,   HttpServletRequest request, HttpServletResponse response) {
4.         // contexte Thymeleaf
5.         WebContext thymeleafContext = new WebContext(request, response, request.getServletContext());
6.         // réponse
7.         JsonResult13 result = new JsonResult13();
8.         // post valide ?
9.         if (bindingResult.hasErrors()) {
10.           // on renvoie une erreur
11.           result.setErreur(getErreursForModel(bindingResult));
12.           return result;
13.        }
14.        // on envoie la page 2
15.        result.setValue1(post.getValue1());
16.        result.setValue2(post.getValue2());
17.        // page requise ?
18.        if (post.isPageRequired()) {
19.           result.setPage2(engine.process("vue-12-page2", thymeleafContext));
20.        }
21.        return result;
22. }
```

- lignes 9-13 : si les valeurs postées [value1, value2] sont invalides, on renvoie un message d'erreur ;
- lignes 15-16 : normalement, le serveur devrait faire un calcul avec les valeurs postées. Ici, il se contente de les renvoyer pour montrer qu'il les a bien reçues ;
- lignes 18-20 : la page n° 2 n'est renvoyée que si elle a été demandée par le client. Ligne 19, la vue [vue-12-page2] est nouvelle :




```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h2>Page 2</h2>
5.        <p>
6.           <h4>Valeurs saisies :</h4>
7.           <p>
8.              Chaîne de caractères :
9.              <span id="value1"></span>
10.          </p>
11.          <p>
12.             Nombre entier :
13.             <span id="value2"></span>
14.          </p>
15.          <a href="javascript:retourPage1()">Retour à la page 1</a>
16.       </p>
17.    </body>
18. </html>
```

- le code XML ne contient plus de valeurs évaluées par Thymeleaf comme c'était le cas auparavant ;
- on a identifié les zones où placer les valeurs renvoyées [value1, value2] par le serveur. Ligne 9, [id='value1'] désigne l'endroit où placer [value1]. Ligne 13, même chose pour [value2] ;

7.6.8 Traitement de la réponse de l'action [/ajax-14]

La réponse de l'action [/ajax-14] est traitée par la fonction [success] suivante :

```
1.  // validation des valeurs saisies
2.  function valider() {
3.     ...
4.     // on fait un appel Ajax à la main
5.     $.ajax({
6.        ...
7.        success : function(data) {
8.           // erreur ?
9.           if (data.erreur) {
10.             // affichage erreur
11.             erreur.html(data.erreur);
12.             erreur.show();
13.          } else {
14.             // pas d'erreur
15.             erreur.hide();
16.             // page 2
17.             if (page2) {
18.                // on utilise la page en cache
19.                content.html(page2);
20.             } else {
21.                // on mémorise la page 2
22.                page2 = data.page2;
23.                // on l'affiche
```

```
24.                 content.html(data.page2);
25.             }
26.             // on la met à jour avec les infos du serveur
27.             $("#value1").text(data.value1);
28.             $("#value2").text(data.value2);
29.         }
30.     },
31. ...
32.   })
33. }
```

- lignes 9-13 : si le serveur a renvoyé une erreur, on l'affiche ;
- lignes 14-29 : le cas où il n'y a pas eu d'erreur. On doit alors afficher la page n° 2 ;
- ligne 17 : on regarde si la page n° 2 est déjà enregistrée dans la variable [page2] ;
- ligne 19 : dans ce cas, on utilise la variable [page2] pour afficher la page n° 2 ;
- ligne 24 : sinon, on utilise le champ [data.page2] fourni par le serveur ;
- ligne 22 : on prend soin de mémoriser la page n° 2 pour ne plus la redemander par la suite ;
- lignes 27-28 : dans la page n° 2, on affiche les deux informations [value1, value2] envoyées par le serveur ;

7.6.9 Retour à la page n° 1

Le lien [Retour vers la page 1] dans la page N° 2 est le suivant :

```
<a href="javascript:retourPage1()">Retour à la page 1</a>
```

La méthode JS [retourPage1] est la suivante :

```
1.  // retour page 1
2.  function retourPage1() {
3.      // on régénère la page 1
4.      content.html(page1);
5.      // on régénère les saisies
6.      $("#text1").val(value1);
7.      $("#text2").val(value2);
8.  }
```

- c'est une action JS sans interaction avec le serveur car la page n° 1 a été mémorisée localement dans la variable [page1] ;
- ligne 4 : on régénère la page n° 1 ;
- ligne 6-7 : seule la partie HTML de la page n° 1 avait été mémorisée. Pas les saisies. On doit donc régénérer celles-ci ;

7.6.10 Conclusion

En exploitant les possibilité du modèle APU, nous avons réussi à simplifier le serveur web qui est maintenant sans état (absence de session) et est moins sollicité :

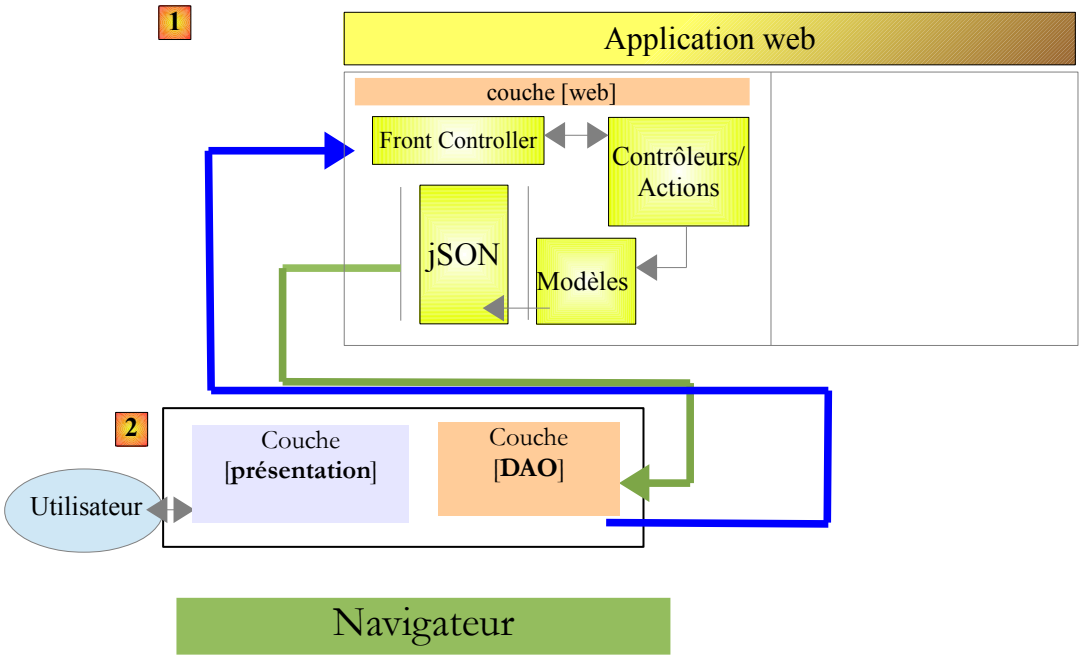
- nous avons supprimé l'interaction avec le serveur dans la fonction JS [retourPage1]) ;
- le serveur ne génère la page n° 2 qu'une fois ;

7.7 Structuration du code Javascript en couches

7.7.1 Introduction

Le code Javascript de l'application précédente commence à devenir complexe. Il est temps qu'on le structure en couches. L'application va rester la même que précédemment. Nous n'allons pas toucher au serveur sauf pour ce qui est de définir une nouvelle page de démarrage. Nous allons refaçonner le code JS.

La nouvelle architecture sera la suivante :



7.7.2 La page de démarrage

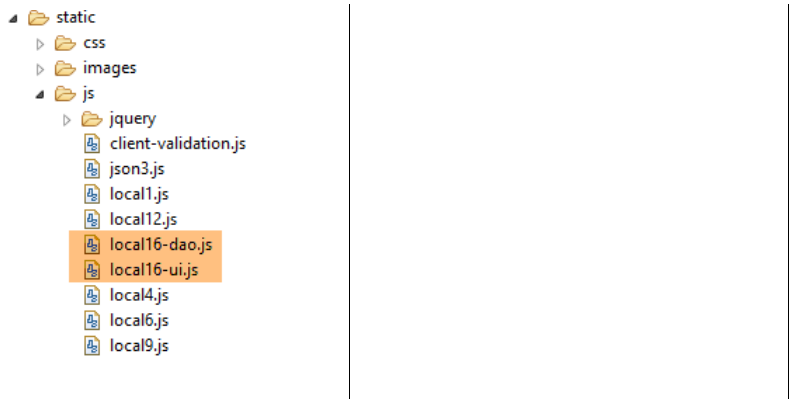
L'action qui lance l'application est l'action [/ajax-16] suivante :

```
1.     @RequestMapping(value = "/ajax-16", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
2.     public String ajax16() {
3.         return "vue-16";
4.  }
```

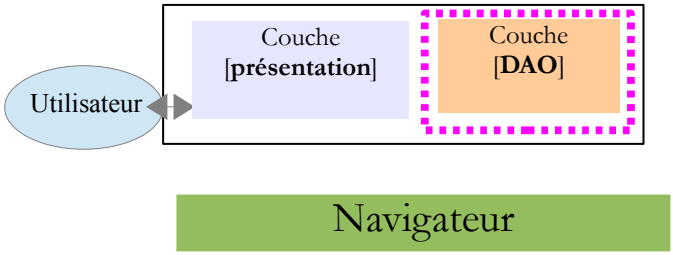
Elle affiche la vue [vue-16.xml] suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>Ajax-12</title>
6.        <link rel="stylesheet" href="/css/ajax01.css" />
7.        <script type="text/javascript" src="/js/jquery/jquery-2.1.1.min.js"></script>
8.        <script type="text/javascript" src="/js/json3.js"></script>
9.        <script type="text/javascript" src="/js/local16-dao.js"></script>
10.       <script type="text/javascript" src="/js/local16-ui.js"></script>
11.    </head>
12.    <body>
13.       <h3>Ajax - 16 - Navigation dans une Application à Page Unique</h3>
14.       <h3>Structuration du code JS</h3>
15.       <hr />
16.       <div id="content" th:include="vue-09-page1" />
17.       <img id="loading" src="/images/loading.gif" />
18.       <div id="erreur" style="background-color:lightgrey"></div>
19.    </body>
20. </html>
```

- lignes 9-10 : le code JS a été placé dans deux fichiers différents :
  - [local-ui] implémente la couche [présentation],
  - [local-dao] implémente la couche [DAO] ;



7.7.3 Implémentation de la couche [DAO]



7.7.4 Interface

La couche [DAO] dans [local-dao.js] va présenter l'interface suivante à la couche [présentation] :

`function updatePage1(deferred, sendMeBack)`	pour mettre à jour la page 1 avec le bouton [Rafraîchir]
`function getPage2(deferred, sendMeBack, value1, value2, pageRequired)`	pour afficher la page 2 avec le bouton [Valider]

Le Javascript n'a pas la notion d'interface. J'ai utilisé ce terme simplement pour indiquer que la couche [présentation] s'engageait à dialoguer avec la couche [DAO] uniquement via les deux fonctions précédentes.

7.7.5 Implémentation de l'interface

Le squelette de l'implémentation est le suivant :

```
1.  var session = {
2.      "cpt1" : 0,
3.      "cpt3" : 0
4.  };
5.
6.  // update Page 1
7.  function updatePage1(deferred, sendMeBack) {
8.  ...
9.  }
10.
11. // page 2
12. function getPage2(deferred, sendMeBack, value1, value2, pageRequired) {
13. ...
14. }
```

Le but de la couche [DAO] est de cacher à la couche [présentation] les détails des requêtes HTTP faites au serveur web. La session fait partie de ces détails. Elle est donc désormais gérée par la couche [DAO].

7.7.5.1 La fonction [updatePage1]

La fonction [updatePage1] est la fonction appelée par la couche [présentation] pour rafraîchir la page 1. Son code est le suivant :

```
1.  // update Page 1
```

```
2.  function updatePage1(deferred, sendMeBack) {
3.     // requête HTTP
4.     executePost(deferred, sendMeBack, '/ajax-13', session);
5.  }
```

- ligne 1 : la fonction [updatePage1] reçoit deux paramètres :
  1. un objet de type [jQuery.Deferred]. Ce type d'objet mémorise un état qui peut avoir trois valeurs ['pending', 'resolved', 'rejected']. Lorsqu'il arrive dans la fonction [updatePage1], il est dans l'état [pending] ;
  2. un objet JS à renvoyer dans la couche [présentation] ;

Toutes les requêtes HTTP sont effectuées par la fonction [executePost] suivante :

```
1.  // requête HTTP
2.  function executePost(deferred, sendMeBack, url, post) {
3.     // on fait un appel Ajax à la main
4.     $.ajax({
5.        headers : {
6.           'Accept' : 'application/json',
7.           'Content-Type' : 'application/json'
8.        },
9.        url : url,
10.       type : 'POST',
11.       data : JSON3.stringify(post),
12.       dataType : 'json',
13.       success : function(data) {
14.          // on mémorise la session
15.          if (data.session) {
16.             session = data.session;
17.          }
18.          // on rend le résultat
19.          deferred.resolve({
20.             "status" : 1,
21.             "data" : data,
22.             "sendMeBack" : sendMeBack
23.          });
24.       },
25.       error : function(jqXHR) {
26.          // on rend l'erreur
27.          deferred.resolve({
28.             "status" : 2,
29.             "data" : jqXHR.responseText,
30.             "sendMeBack" : sendMeBack
31.          });
32.       }
33.    });
34.    }
```

- ligne 1 : la fonction [executePost] exécute un appel Ajax de type POST. Elle attend quatre paramètres :
  1. un objet de type [jQuery.Deferred] dans l'état [pending] ;
  2. un objet JS à renvoyer dans la couche [présentation] ;
  3. l'URL du POST ;
  4. la valeur à poster en tant qu'objet JS ;
- lignes 5-8 : la fonction poste du jSON (ligne 7) et reçoit du jSON (ligne 6) ;
- ligne 11 : la valeur à poster est transformée en jSON ;
- lignes 13-24 : la fonction exécutée en cas de succès de l'appel Ajax ;
- lignes 19-23 : si le serveur a renvoyé une session, on la mémorise ;
- lignes 13-18 : passent l'objet [deferred] dans l'état [resolved] en passant de plus un résultat avec les champs suivants :
  - [status] : à 1 pour un succès, à 2 pour un échec,
  - [data] : la réponse jSON du serveur,
  - [sendMeBack] : le 2ième paramètre de la fonction qui est un objet que l'appelant veut récupérer ;
- lignes 17-31 : la fonction exécutée en cas d'échec de l'appel Ajax. On fait la même chose que précédemment avec deux différences :
  - [status] passe à 2 pour signaler une erreur ;
  - [data] est là encore la réponse jSON du serveur mais obtenue d'une façon différente ;

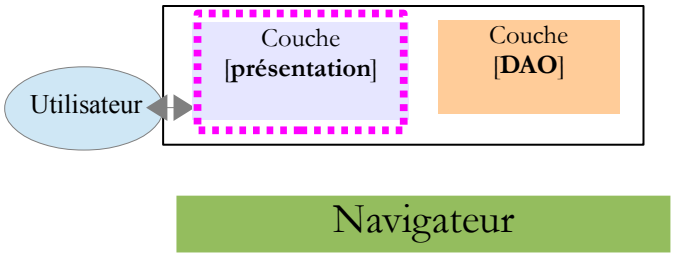
7.7.5.2 La fonction [getPage2]

La fonction [getPage2] est la suivante :

```
1.  // page 2
2.  function getPage2(deferred, sendMeBack, value1, value2, pageRequired) {
3.     // requête HTTP
4.     executePost(deferred, sendMeBack, '/ajax-14', {
5.        "value1" : value1,
6.        "value2" : value2,
7.        "pageRequired" : pageRequired,
8.     });
9.  }
```

- la fonction reçoit les paramètres suivants :
  1. [deferred] : un objet de type [jQuery.Deferred] dans l'état [pending],
  2. [sendMeBack] : un objet JS à renvoyer dans la couche [présentation],
  3. [value1] : la première saisie dans page 1,
  4. [value2] : la seconde saisie dans page 2,
  5. [pageRequired] : un booléen indiquant au serveur s'il doit ou non envoyer le flux HTML de la page n° 2 ;
- la fonction [executePost] est appelée pour exécuter la requête HTTP nécessaire ;

7.7.6 La couche [présentation]



La couche [présentation] est implémentée par le fichier [local-ui.js]. Ce dernier reprend le code du fichier [local12.js] refaçonné pour utiliser la couche [DAO] précédente. Seules deux fonctions changent : [postForm] et [valider].

7.7.6.1 La fonction [postForm]

La fonction [postForm] est la suivante :

```
1.  // update Page 1
2.  function postForm() {
3.     // on met à jour la page 1
4.     var deferred = $.Deferred();
5.     loading.show();
6.     updatePage1(deferred, {
7.        'sender' : "postForm",
8.        'info' : 10
9.     });
10.    // affichage résultats
11.    deferred.done(postFormDone);
12. }
```

- ligne 4 : on crée un objet [jQuery.Deferred]. Par défaut, il est dans l'état [pending] ;
- ligne 5 : l'image d'attente est affichée
- lignes 6-9 : la fonction [updatePage1] est exécutée. On passe un objet [sendMeBack] fictif, juste pour montrer à quoi ça peut servir ;
- ligne 11 : le paramètre de la fonction [deferred.done] est elle-même une fonction. C'est la fonction à exécuter lorsque l'état de l'objet [deferred] passe dans l'état [resolved]. On vient de voir que la fonction DAO [executePost] passait l'état de cet objet à [resolved] à réception de la réponse du serveur. Cela signifie que lorsque la fonction [postFormDone] s'exécute, la réponse du serveur a été reçue ;

La fonction [postFormDone] est la suivante :

```
1.  function postFormDone(result) {
2.     // fin attente
3.     loading.hide();
4.     // on récupère les données
5.     var data = result.data
6.     // pour démo
```

```
7.      console.log(JSON3.stringify(result.sendMeBack));
8.      // on analyse le status
9.      switch (result.status) {
10.     case 1:
11.        // on met à jour les deux zones
12.        if (data.zone1) {
13.           $("#zone1-content").html(data.zone1);
14.           $("#zone1").show();
15.        } else {
16.           $("#zone1").hide();
17.        }
18.        if (data.zone3) {
19.           $("#zone3").show();
20.           $("#zone3-content").html(data.zone3);
21.        } else {
22.           $("#zone3").hide();
23.        }
24.        break;
25.     case 2:
26.        // affichage erreur
27.        erreur.html(data);
28.        break;
29.     }
30. }
```

- ligne 1 : le paramètre [result] reçu est le paramètre passé à la méthode [deferred.resolve] dans la fonction [executePost], par exemple :

```
1.            // on rend le résultat
2.            deferred.resolve({
3.               "status" : 1,
4.               "data" : data,
5.               "sendMeBack" : sendMeBack
6.  });
```

- ligne 5 : on récupère la réponse du serveur ;
- lignes 10-24 : on a le code qui dans la version précédente était dans la fonction [onSuccess] de la fonction [postForm] ;
- lignes 25-28 : on a le code qui dans la version précédente était dans la fonction [onError] de la fonction [postForm] ;

7.7.6.2 Le rôle du paramètre [sendMeBack]

A quoi sert le paramètre [sendMeBack] ? Regardons le code d'appel de la fonction [updatePage1] :

```
1.  // update Page 1
2.  function postForm() {
3.     // on met à jour la page 1
4.     var deferred = $.Deferred();
5.     loading.show();
6.     updatePage1(deferred, {
7.        'sender' : "postForm",
8.        'info' : 10
9.     });
10.    // affichage résultats
11.    deferred.done(postFormDone);
12. }
```

et la signature de la fonction [validerDone] :

```
1.  function postFormDone(result) {
2.  }
```

Comment peut faire la fonction [postForm] pour passer des informations à la fonction [`postFormDone`] ? Celle-ci, n'a qu'un paramètre [result]. Celui-ci est créé par la fonction [executePost] de la couche [DAO]. Pour transmettre des informations à la fonction [`postFormDone`], la fonction [postForm] doit d'abord les transmettre à la fonction [updatePage1]. C'est le rôle du paramètre [sendMeBack]. Il s'utilise de la façon suivante :

```
1.  function postFormDone(result) {
2.     // fin attente
3.     loading.hide();
4.     // on récupère les données
```

```
5.      var data = result.data
6.      // pour démo
7.      console.log(JSON3.stringify(result.sendMeBack));
8.      // on analyse le status
9.      switch (result.status) {
10. ...
```

- ligne 7, la fonction [postFormDone] a retrouvé le paramètre [sendMeBack] initialement transmis à la fonction DAO [updatePage1] par la fonction [postForm] ;

7.7.7 La fonction [valider]

La fonction [valider] est la suivante :

```
1.  // validation valeurs saisies
2.  function valider() {
3.      // on mémorise la page 1
4.      page1 = content.html();
5.      // on mémorise les valeurs saisies
6.      value1 = $("#text1").val().trim();
7.      value2 = $("#text2").val().trim();
8.      // pas d'erreur
9.      erreur.hide();
10.     // on demande la page 2
11.     var deferred = $.Deferred();
12.     loading.show();
13.     getPage2(deferred, {
14.         'sender' : 'valider',
15.         'info' : 20
16.     }, value1, value2, page2 ? false : true);
17.     // affichage résultats
18.     deferred.done(validerDone);
19. }
```

et la fonction [validerDone] (ligne 18) la suivante :

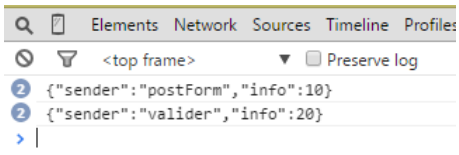
```
1.  function validerDone(result) {
2.      // fin attente
3.      loading.hide();
4.      // on récupère les données
5.      var data = result.data
6.      // pour démo
7.      console.log(JSON3.stringify(result.sendMeBack));
8.      // on analyse le status
9.      switch (result.status) {
10.     case 1:
11.         // erreur ?
12.         if (data.erreur) {
13.             // affichage erreur
14.             erreur.html(data.erreur);
15.             erreur.show();
16.         } else {
17.             // pas d'erreur
18.             erreur.hide();
19.             // page 2
20.             if (page2) {
21.                 // on utilise la page en cache
22.                 content.html(page2);
23.             } else {
24.                 // on mémorise la page 2
25.                 page2 = data.page2;
26.                 // on l'affiche
27.                 content.html(data.page2);
28.             }
29.             // on la met à jour avec les infos du serveur
30.             $("#value1").text(data.value1);
31.             $("#value2").text(data.value2);
32.         }
33.         break;
34.     case 2:
35.         // affichage erreur
36.         erreur.html(data);
37.         erreur.show();
```

```
38.         break;
39.     }
40. }
```

- ligne 5 : on récupère la réponse du serveur ;
- lignes 10-32 : on a le code qui dans la version précédente était dans la fonction [onSuccess] de la fonction [valider] ;
- lignes 34-38 : on a le code qui dans la version précédente était dans la fonction [onError] de la fonction [valider] ;

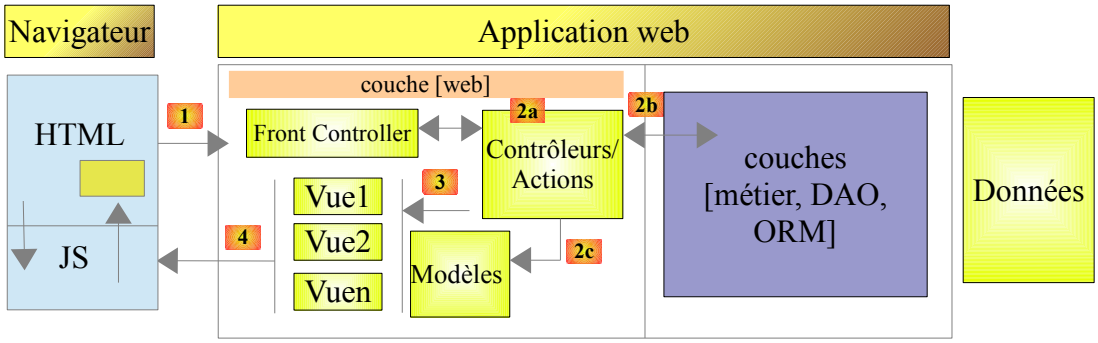
7.7.8 Tests

L'application continue à fonctionner comme auparavant et dans la console de Chrome, on peut voir les paramètres [sendMeBack] des fonctions [postForm] et [valider] :

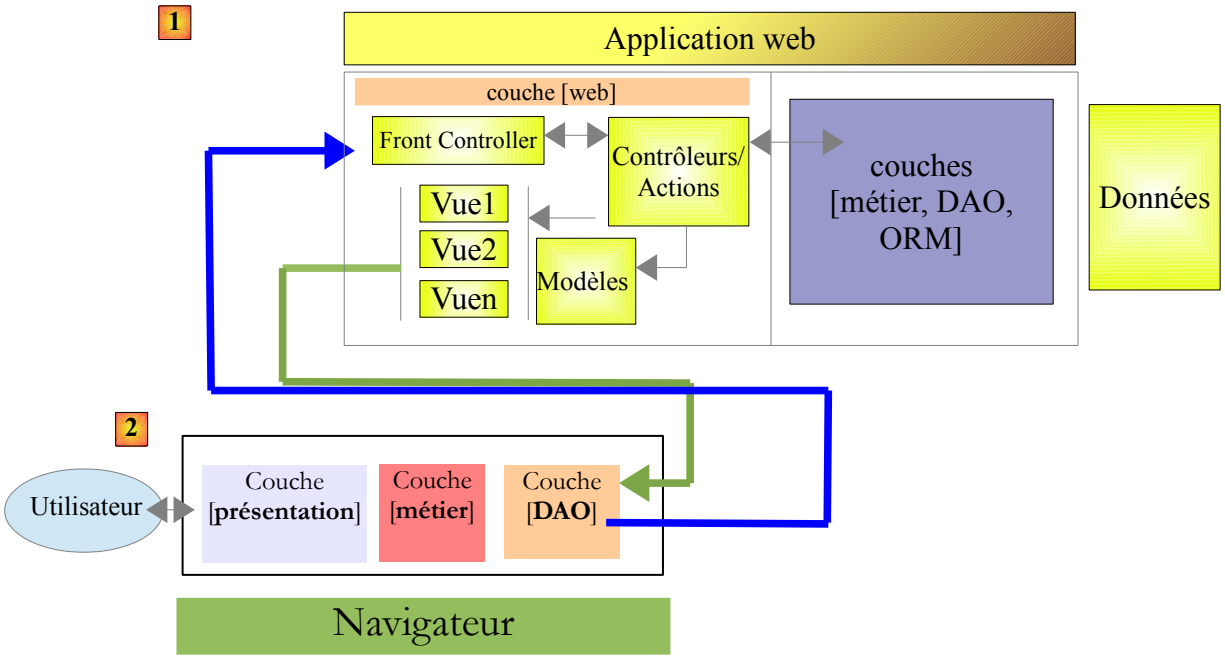


7.8 Conclusion

Revenons au schéma général d'une application Spring MVC :



Grâce au Javascript embarqué dans les pages HTML et exécuté dans le navigateur et grâce au modèle APU, on peut déporter du code sur le navigateur et aboutir à l'architecture suivante :

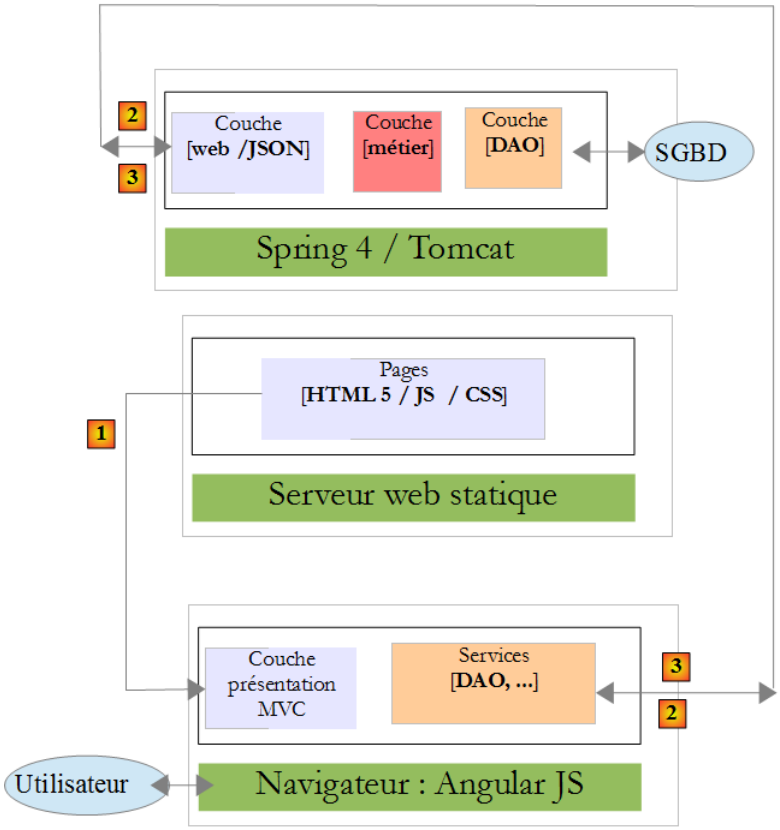


- on a une architecture client [2] / serveur [1] où le client et le serveur communiquent en jSON ;
- en [1], la couche web Spring MVC délivre des vues, des fragments de vue, des données dans du jSON ;
- en [2] : le code Javascript embarqué dans la vue chargée au démarrage de l'application peut être structuré en couches :
    - la couche [présentation] s'occupe des interactions avec l'utilisateur,
    - la couche [DAO] s'occupe de l'accès aux données via le serveur web [1] ,
    - la couche [métier] peut ne pas exister ou reprendre certaines des fonctionnalités non confidentielles de la couche [métier] du serveur afin de soulager celui-ci ;
- le client [2] peut mettre certaines vues en cache afin là encore de soulager le serveur. Il gère la session ;

8 Etude de cas

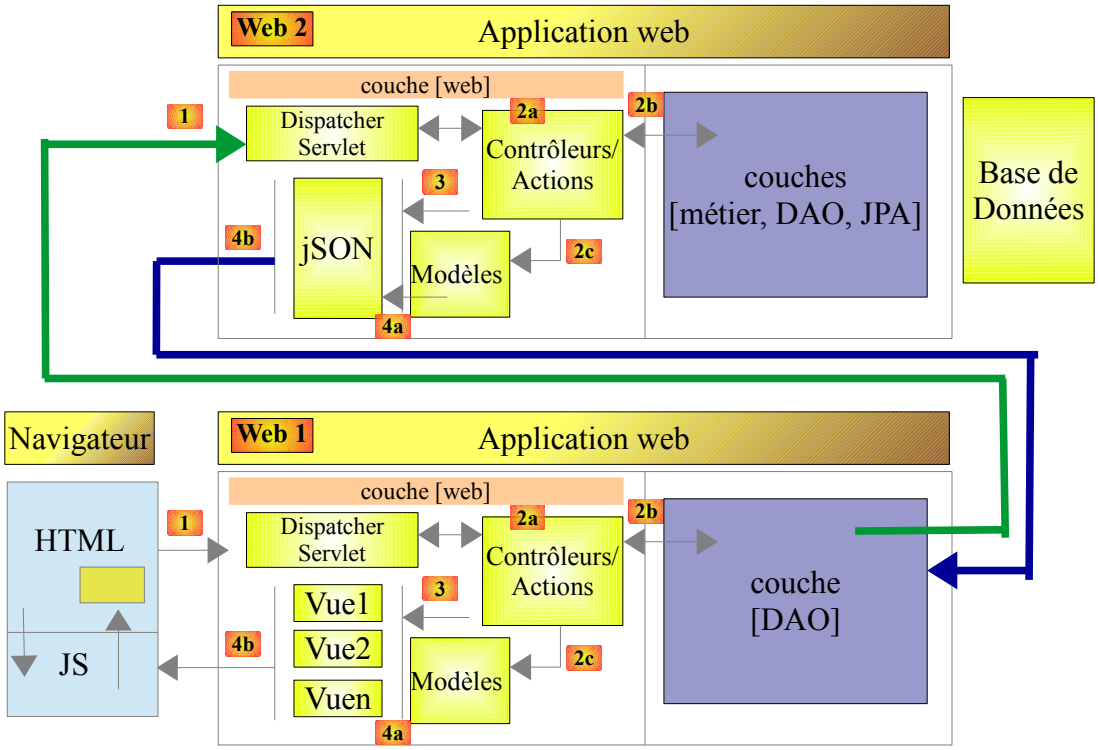
8.1 Introduction

Nous nous proposons d'écrire une application web de prise de rendez-vous pour un cabinet médical. Ce problème a été traité dans le document *'Tutoriel AngularJS / Spring 4'* à l'URL [http://tahe.developpez.com/angularjs-spring4/]. L'architecture de cette application était la suivante :



- en [1], un serveur web délivre des pages statiques à un navigateur. Ces pages contiennent une application AngularJS construite sur le modèle MVC (Modèle – Vue – Contrôleur). Le modèle ici est à la fois celui des vues et celui du domaine représenté ici par la couche [Services] ;
- l'utilisateur va interagir avec les vues qui lui sont présentées dans le navigateur. Ses actions vont parfois nécessiter l'interrogation du serveur Spring 4 [2]. Celui-ci traitera la demande et rendra une réponse jSON (JavaScript Object Notation) [3]. Celle-ci sera utilisée pour mettre à jour la vue présentée à l'utilisateur.

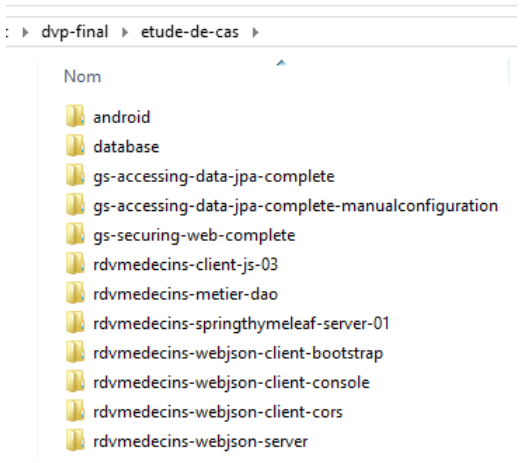
Nous nous proposons de reprendre cette application et de l'implémenter de bout en bout avec Spring MVC. L'architecture devient alors la suivante :

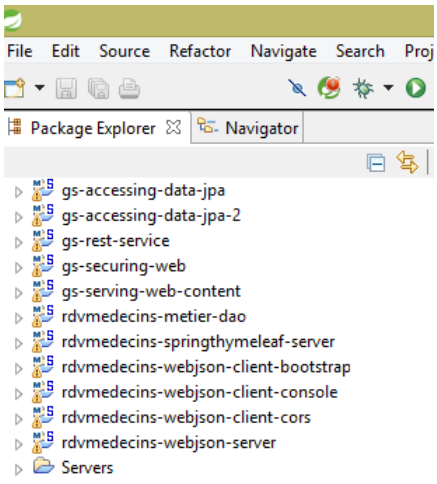


Le navigateur se connectera à une application [Web 1] implémentée par Spring MVC qui ira chercher ses données auprès d'un service web [Web 2] lui aussi implémenté avec Spring MVC.

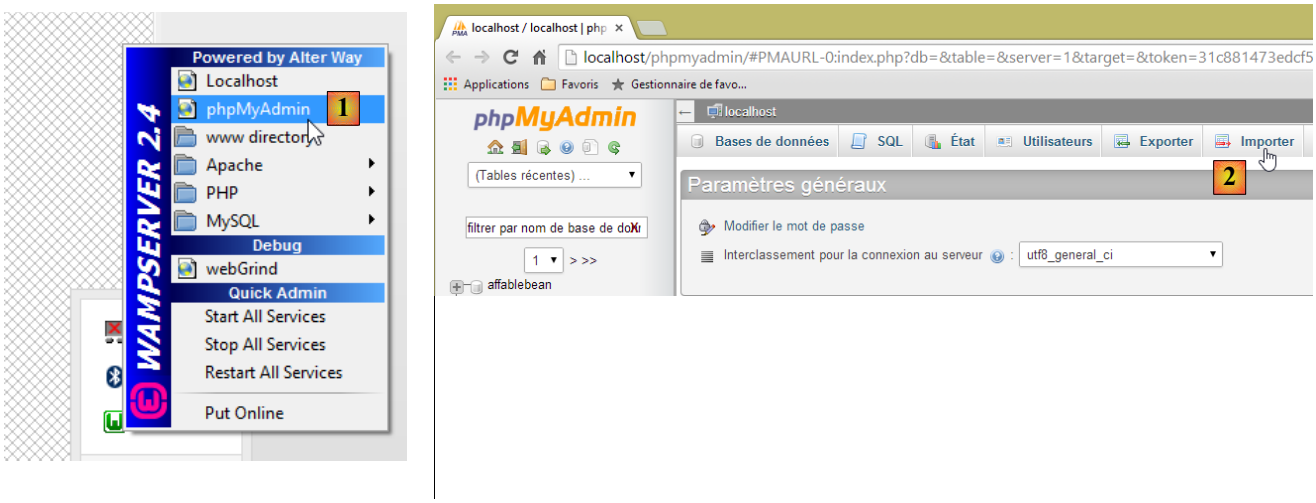
8.2 Fonctionnalités de l'application

Le lecteur est invité à découvrir les fonctionnalités de l'application en la testant. Nous chargeons dans STS les projets Maven du dossier [etude-de-cas] :

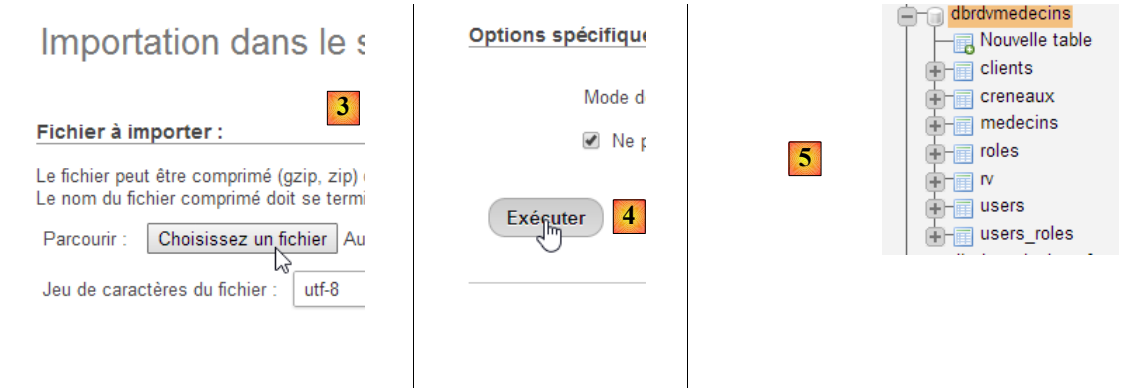




Tout d'abord nous allons créer la base de données MySQL 5 [dbrdvmedecins] avec l'outil [Wamp Server] (cf paragraphe 9.5, page 605) :

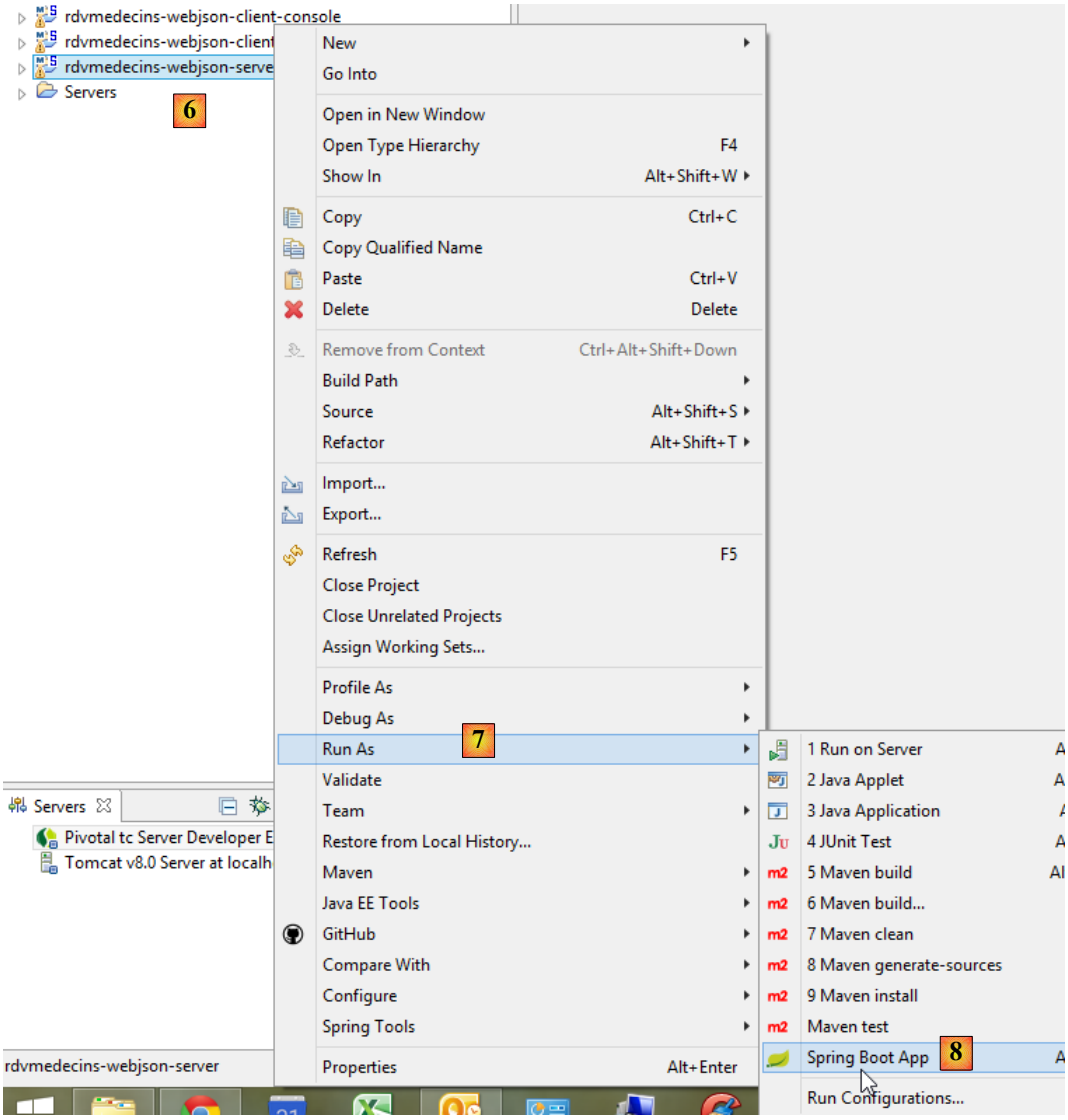


- en [1], on sélectionne l'outil [phpMyAdmin] de WampServer ;
- en [2], on choisit l'option [Importer] ;



- en [3], on sélectionne le fichier [database/dbrdvmedecins.sql] ;
- en [4], on l'exécute ;
- en [5], la base de données créée.

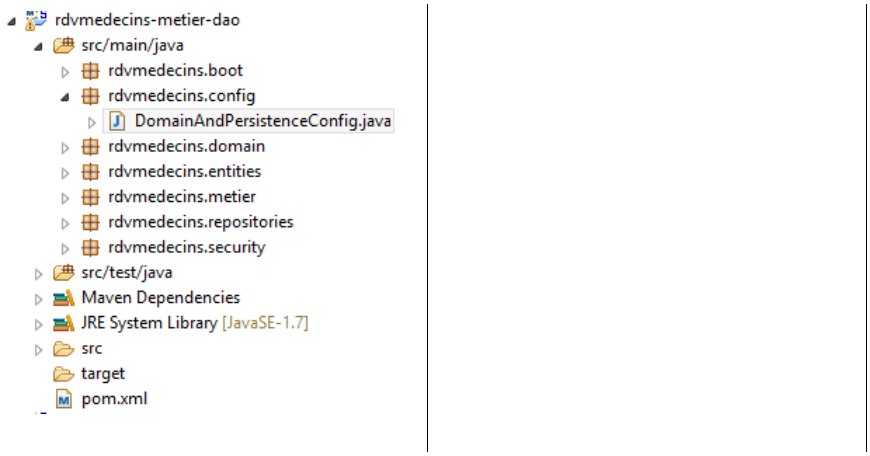
Ensuite, il nous faut lancer le serveur connecté à la base de données. C'est le projet [rdvmedecins-webjson-server]



Le serveur va être disponible à l'URL [http://localhost:8080]. Cela peut être changé dans le fichier [application.properties] du projet :


```
server.port=8080
```

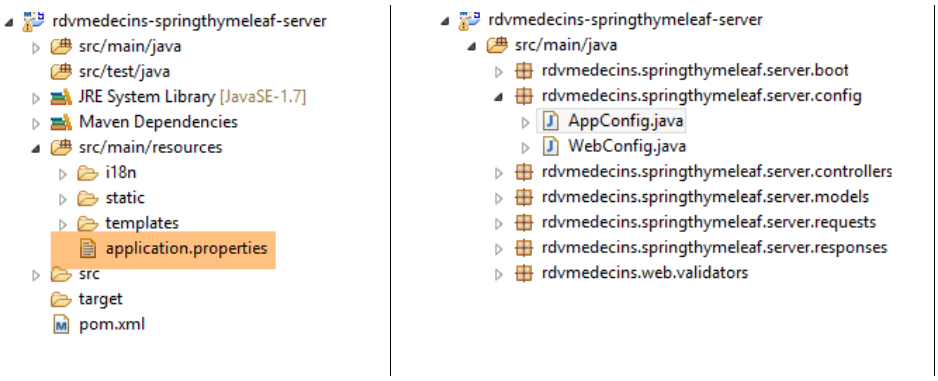
Les caractéristiques d'accès à la base de données sont enregistrées dans la classe [DomainAndPersistenceConfig] du projet [rdvmedecins-metier-dao] :




```
1.      // la source de données MySQL
2.      @Bean
3.      public DataSource dataSource() {
4.         BasicDataSource dataSource = new BasicDataSource();
5.         dataSource.setDriverClassName("com.mysql.jdbc.Driver");
6.         dataSource.setUrl("jdbc:mysql://localhost:3306/dbrdvmedecins");
7.         dataSource.setUsername("root");
8.         dataSource.setPassword("");
9.         return dataSource;
10.  }
```

Si vous accédez au SGBD MySQL avec d'autres identifiants, c'est là que ça se passe.

On lance ensuite, de la même façon que le serveur précédent, le serveur [rdvmedecins-springthymeleaf-server] :



Ce serveur est par défaut disponible à l'URL [http://localhost:8081]. De nouveau, c'est configurable dans le fichier [application.properties] du projet :

```
server.port=8081
```

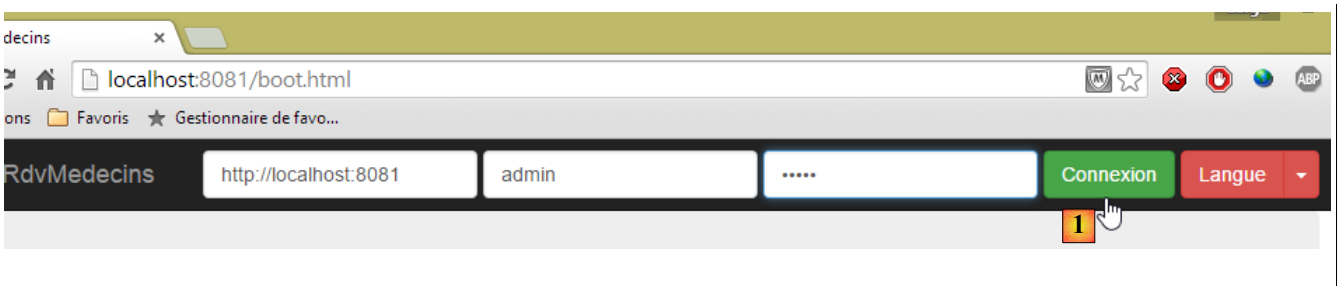
Par ailleurs, ce serveur doit connaître l'URL du serveur connecté à la base de données. Cette configuration se trouve dans la classe [AppConfig] ci-dessus :

```
1.      // admin / admin
2.      private final String USER_INIT = "admin";
3.      private final String MDP_USER_INIT = "admin";
4.      // racine service web / json
5.      private final String WEBJSON_ROOT = "http://localhost:8080";
6.      // timeout en millisecondes
7.      private final int TIMEOUT = 5000;
8.      // CORS
9.   private final boolean CORS_ALLOWED=true;
```

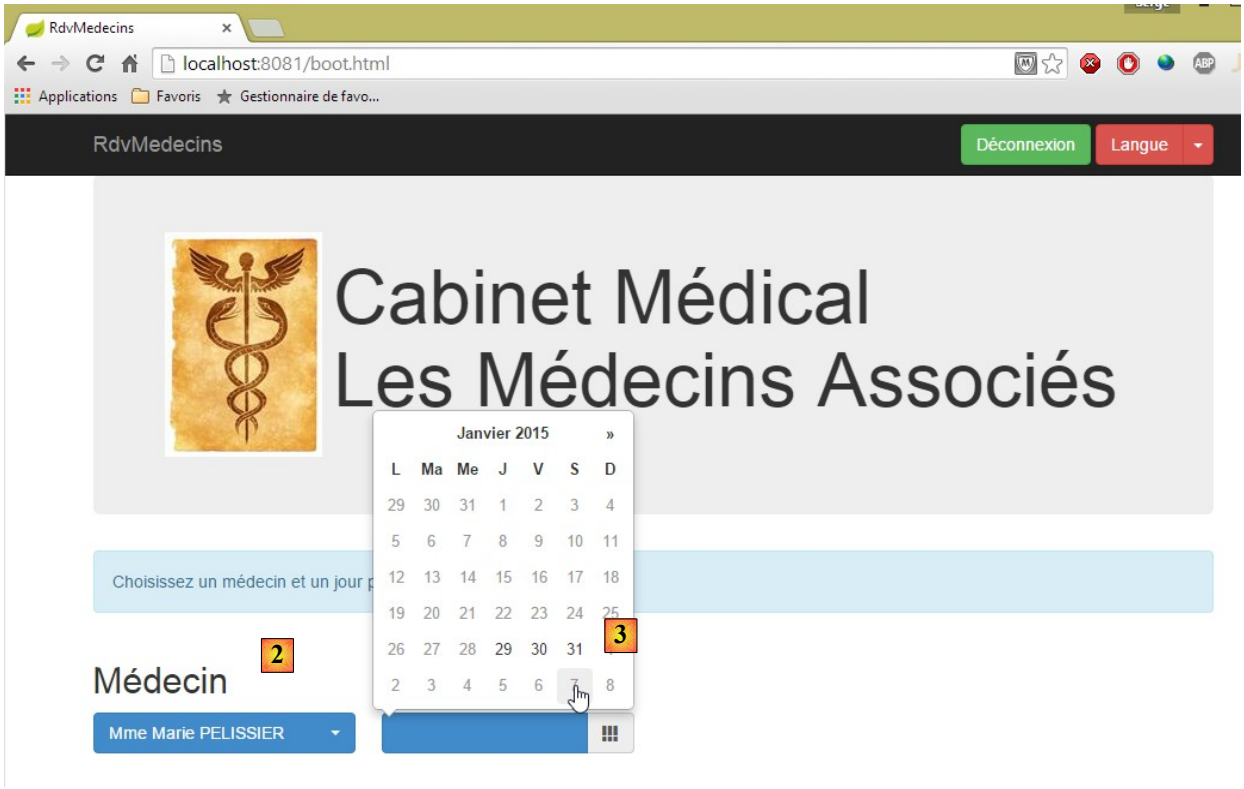
Si le premier serveur a été lancé sur un autre port que le 8080, il faut modifier la ligne 5.

Ensuite avec un navigateur, on demande l'URL [http://localhost:8081/boot.html] :

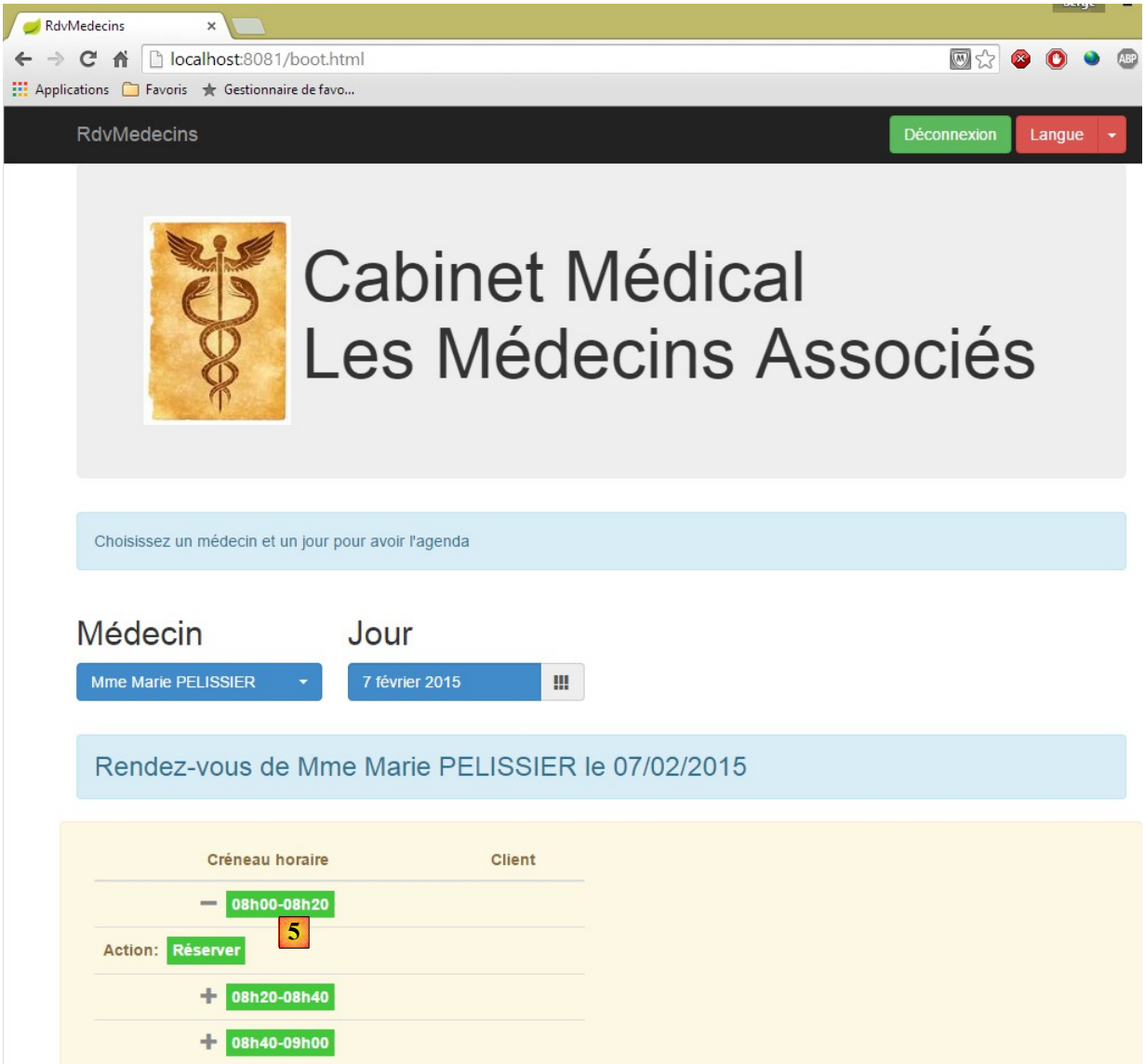
- en [1], la page d'entrée de l'application ;
- en [2] et [3], l'identifiant et le mot de passe de celui qui veut utiliser l'application. Il y a deux utilisateurs : **admin/admin** (login/password) avec un rôle (ADMIN) et **user/user** avec un rôle (USER). Seul le rôle ADMIN a le droit d'utiliser l'application. Le rôle USER n'est là que pour montrer ce que répond le serveur dans ce cas d'utilisation ;
- en [4], le bouton qui permet de se connecter au serveur ;
- en [5], la langue de l'application. Il y en a deux : le français par défaut et l'anglais ;
- en [6], l'URL du serveur [rdvmedecins-springthymeleaf-server] ;



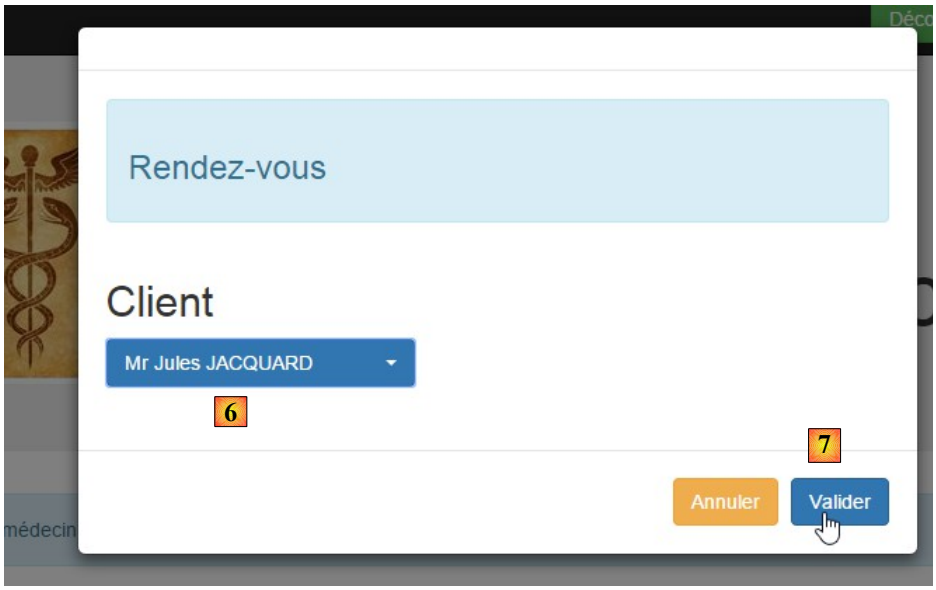
- en [1], on se connecte ;



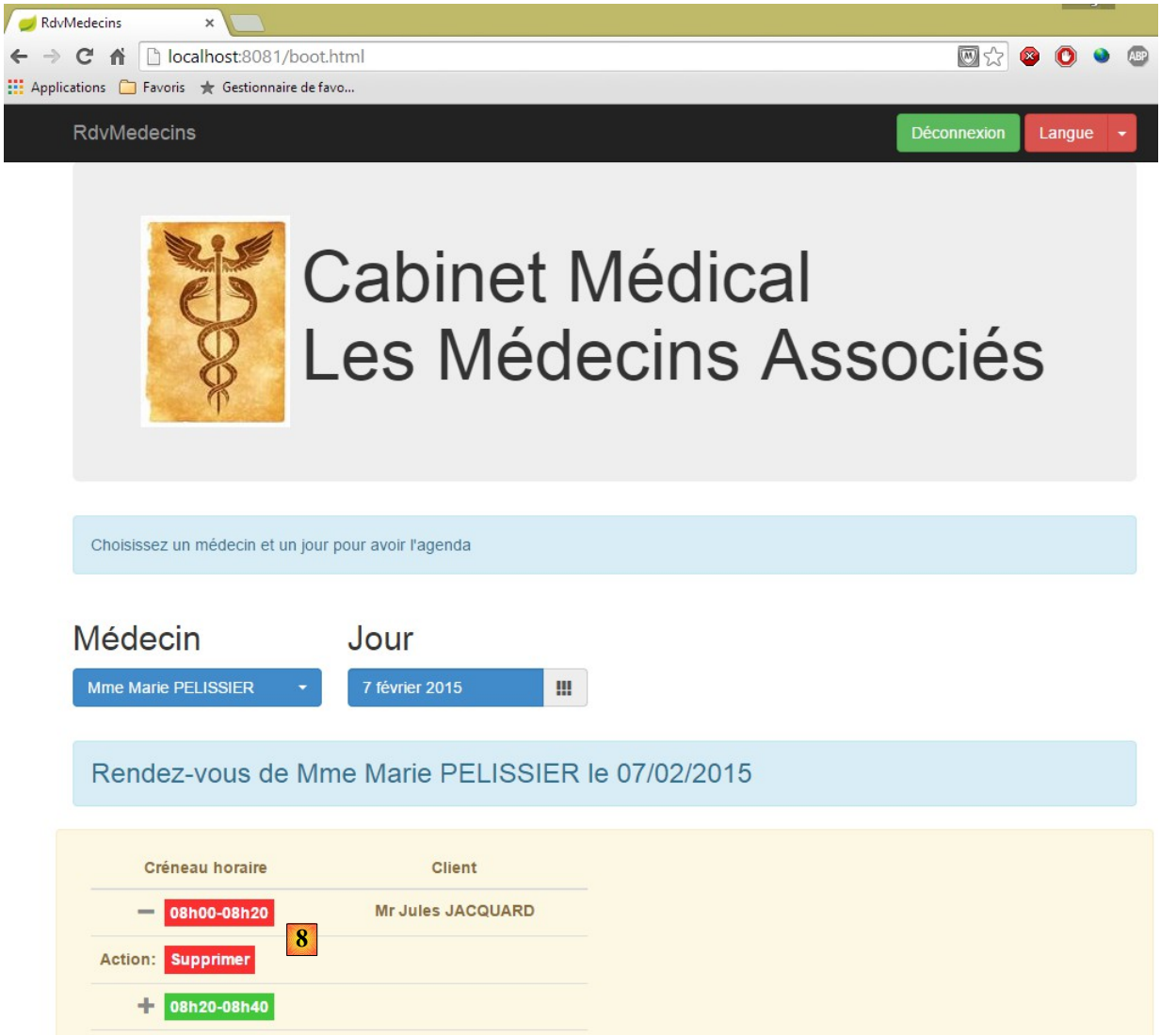
- une fois connecté, on peut choisir le médecin avec lequel on veut un rendez-vous [2] et le jour de celui-ci [3]. Dès qu'un médecin et un jour ont été renseignés, l'agenda est automatiquement affiché :



- une fois obtenu l'agenda du médecin, on peut réserver un créneau [5] ;



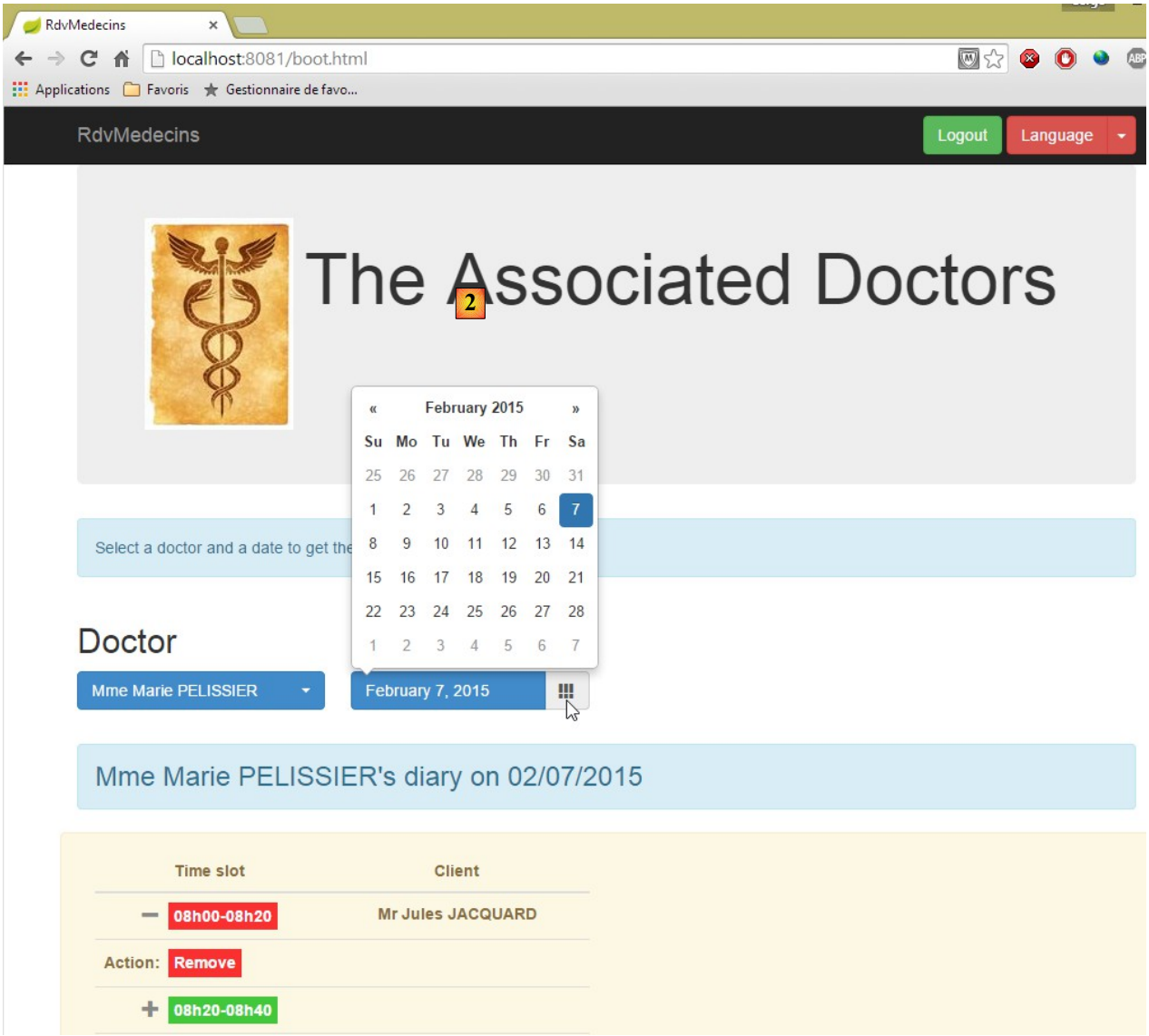
- en [6], on choisit le patient pour le rendez-vous et on valide ce choix en [7] ;



Une fois le rendez-vous validé, on est ramené automatiquement à l'agenda où le nouveau rendez-vous est désormais inscrit. Ce rendez-vous pourra être ultérieurement supprimé [8].

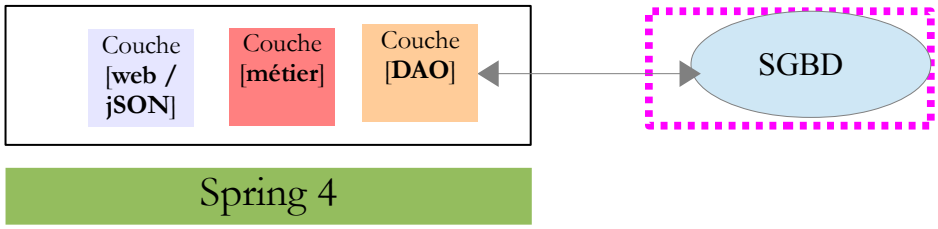
Les principales fonctionnalités ont été décrites. Elles sont simples. Terminons par la gestion de la langue :

- en [1], on passe du français à l'anglais ;

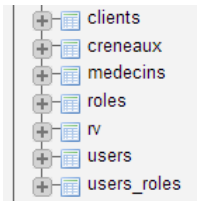


- en [2], la vue est passée en anglais, y-compris le calendrier ;

8.3 La base de données



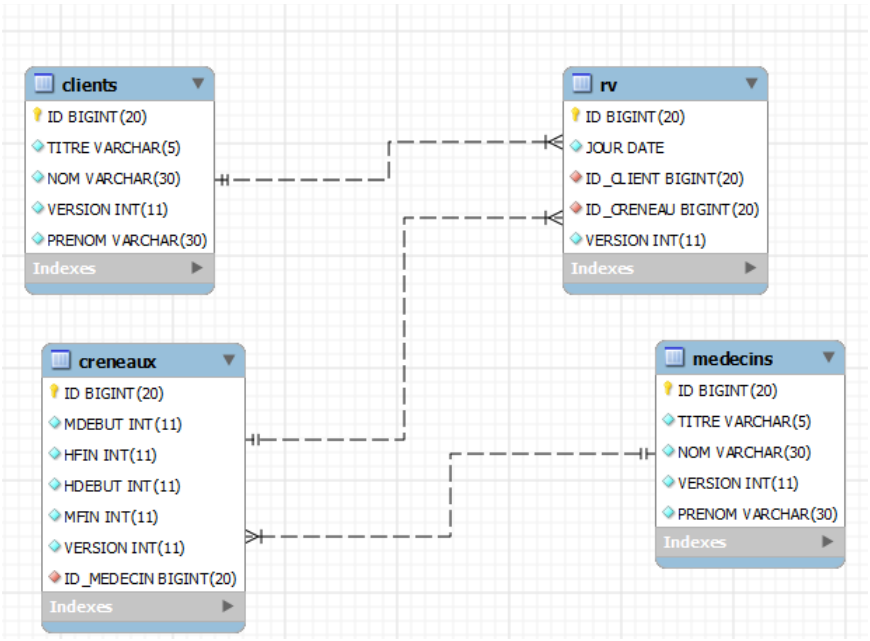
La base de données appelée par la suite [dbrdvmedecins] est une base de données MySQL5 avec les tables suivantes :



Les rendez-vous sont gérés par les tables suivantes :

- [medecins] : contient la liste des médecins du cabinet ;
- [clients] : contient la liste des patienst du cabinet ;
- [creneaux] : contient les créneaux horaires de chacun des médecins ;
- [rv] : contient la liste des rendez-vous des médecins.

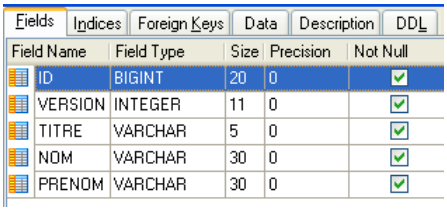
Les tables [roles], [users] et [users_roles] sont des tables liées à l'authentification. Dans un premier temps, nous n'allons pas nous en occuper. Les relations entre les tables gérant les rendez-vous sont les suivantes :

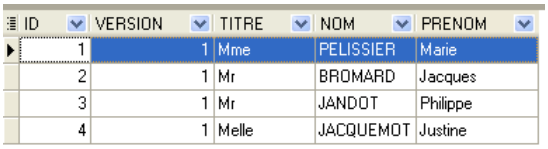


- un créneau horaire appartient à un médecin – un médecin a 0 ou plusieurs créneaux horaires ;
- un rendez-vous réunit à la fois un client et un médecin via un créneau horaire de ce dernier ;
- un client a 0 ou plusieurs rendez-vous ;
- à un créneau horaire est associé 0 ou plusieurs rendez-vous (à des jours différents).

8.3.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].



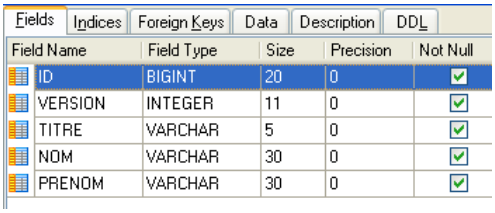


- ID : n° identifiant le médecin - clé primaire de la table

- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

8.3.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :



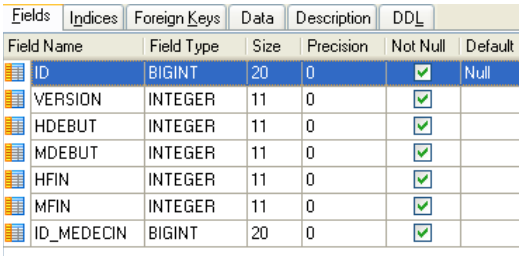
Field Name	Field Type	Size	Precision	Not Null
ID	BIGINT	20	0	☑
VERSION	INTEGER	11	0	☑
TITRE	VARCHAR	5	0	☑
NOM	VARCHAR	30	0	☑
PRENOM	VARCHAR	30	0	☑

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mr	MARTIN	Jules
2	1	Mme	GERMAN	Christine
3	1	Mr	JACQUARD	Jules
4	1	Melle	BISTROU	Brigitte

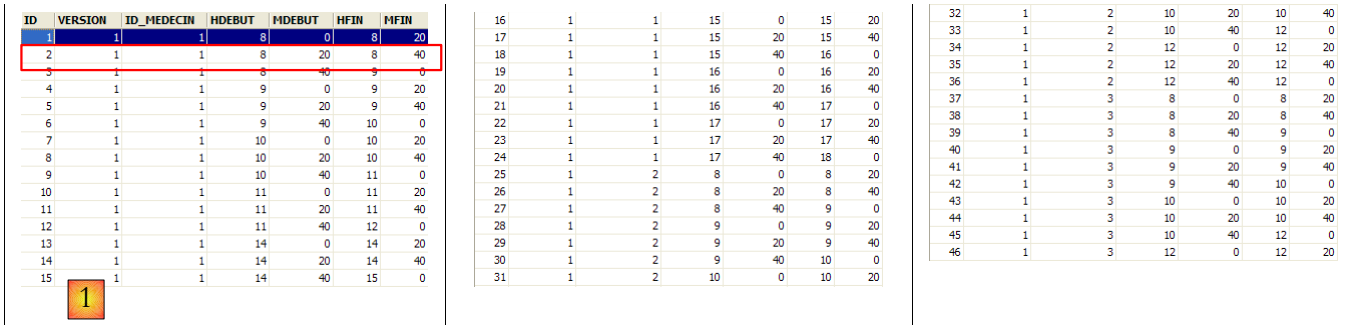
- ID : n° identifiant le client - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du client
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

8.3.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :



Field Name	Field Type	Size	Precision	Not Null	Default
ID	BIGINT	20	0	☑	Null
VERSION	INTEGER	11	0	☑	
HDEBUT	INTEGER	11	0	☑	
MDEBUT	INTEGER	11	0	☑	
HFIN	INTEGER	11	0	☑	
MFIN	INTEGER	11	0	☑	
ID_MEDECIN	BIGINT	20	0	☑	



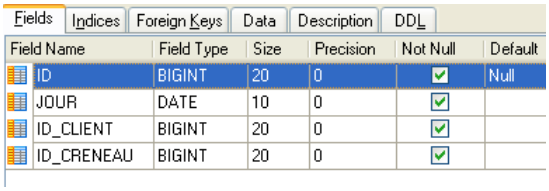
ID	VERSION	ID_MEDECIN	HDEBUT	MDEBUT	HFIN	MFIN
1	1	1	8	0	8	20
2	1	1	8	20	8	40
3	1	1	8	40	9	0
4	1	1	9	0	9	20
5	1	1	9	20	9	40
6	1	1	9	40	10	0
7	1	1	10	0	10	20
8	1	1	10	20	10	40
9	1	1	10	40	11	0
10	1	1	11	0	11	20
11	1	1	11	20	11	40
12	1	1	11	40	12	0
13	1	1	14	0	14	20
14	1	1	14	20	14	40
15	1	1	14	40	15	0
16	1	1	15	0	15	20
17	1	1	15	20	15	40
18	1	1	15	40	16	0
19	1	1	16	0	16	20
20	1	1	16	20	16	40
21	1	1	16	40	17	0
22	1	1	17	0	17	20
23	1	1	17	20	17	40
24	1	1	17	40	18	0
25	1	2	8	0	8	20
26	1	2	8	20	8	40
27	1	2	8	40	9	0
28	1	2	9	0	9	20
29	1	2	9	20	9	40
30	1	2	9	40	10	0
31	1	2	10	0	10	20
32	1	2	10	20	10	40
33	1	2	10	40	12	0
34	1	2	12	0	12	20
35	1	2	12	20	12	40
36	1	2	12	40	12	0
37	1	3	8	0	8	20
38	1	3	8	20	8	40
39	1	3	8	40	9	0
40	1	3	9	0	9	20
41	1	3	9	20	9	40
42	1	3	9	40	10	0
43	1	3	10	0	10	20
44	1	3	10	20	10	40
45	1	3	10	40	12	0
46	1	3	12	0	12	20

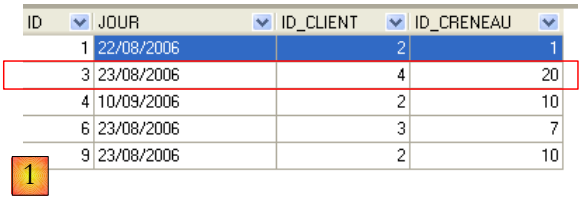
- ID : n° identifiant le créneau horaire - clé primaire de la table (ligne 8)
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – clé étrangère sur la colonne MEDECINS(ID).
- HDEBUT : heure début créneau
- MDEBUT : minutes début créneau
- HFIN : heure fin créneau
- MFIN : minutes fin créneau

La seconde ligne de la table [CRENEAUX] (cf [1] ci-dessus) indique, par exemple, que le créneau n° 2 commence à 8 h 20 et se termine à 8 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER).

8.3.4 La table [RV]

Elle liste les RV pris pour chaque médecin :





- ID : n° identifiant le RV de façon unique – clé primaire
- JOUR : jour du RV
- ID_CRENEAU : créneau horaire du RV - clé étrangère sur le champ [ID] de la table [CRENEAUX] – fixe à la fois le créneau horaire et le médecin concerné.
- ID_CLIENT : n° du client pour qui est faite la réservation – clé étrangère sur le champ [ID] de la table [CLIENTS]

Cette table a une contrainte d'unicité sur les valeurs des colonnes jointes (JOUR, ID_CRENEAU) :

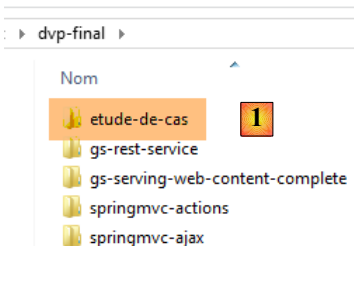
```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

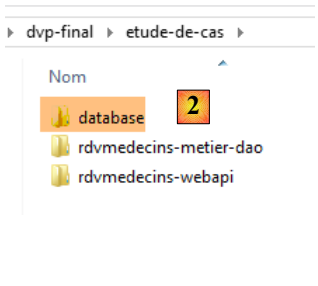
Si une ligne de la table[RV] a la valeur (JOUR1, ID_CRENEAU1) pour les colonnes (JOUR, ID_CRENEAU), cette valeur ne peut se retrouver nulle part ailleurs. Sinon, cela signifierait que deux RV ont été pris au même moment pour le même médecin. D'un point de vue programmation Java, le pilote JDBC de la base lance une *SQLException* lorsque ce cas se produit.

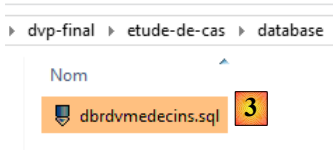
La ligne d'*id* égal à 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTROU.

8.3.5 Création de la base de données

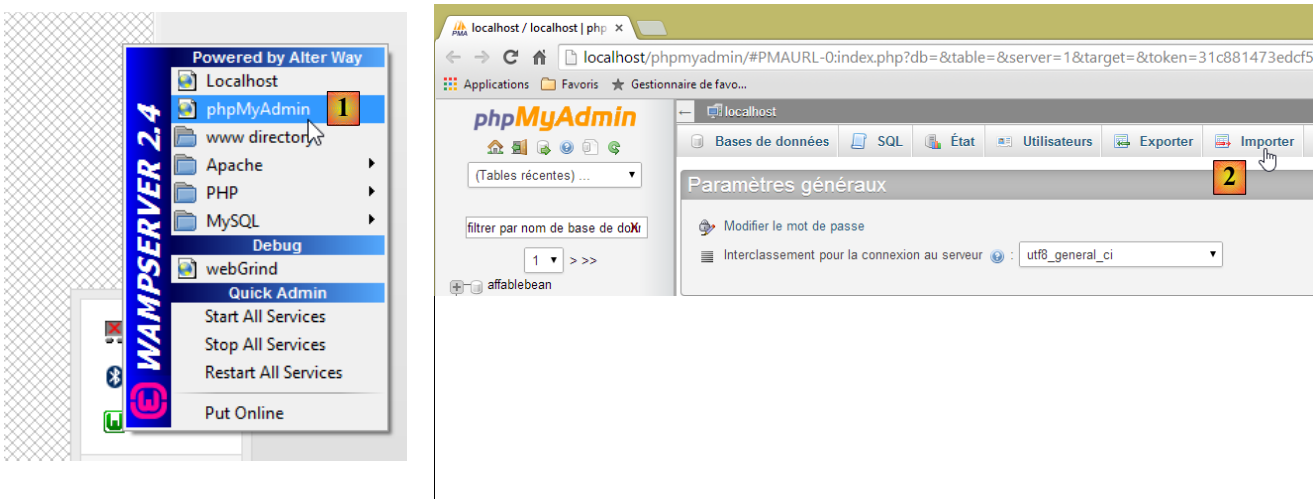
Pour créer la base de données [dbrdvmedecins], un script [dbrdvmedecins.sql] est fourni avec les exemples de ce document [1-3] :



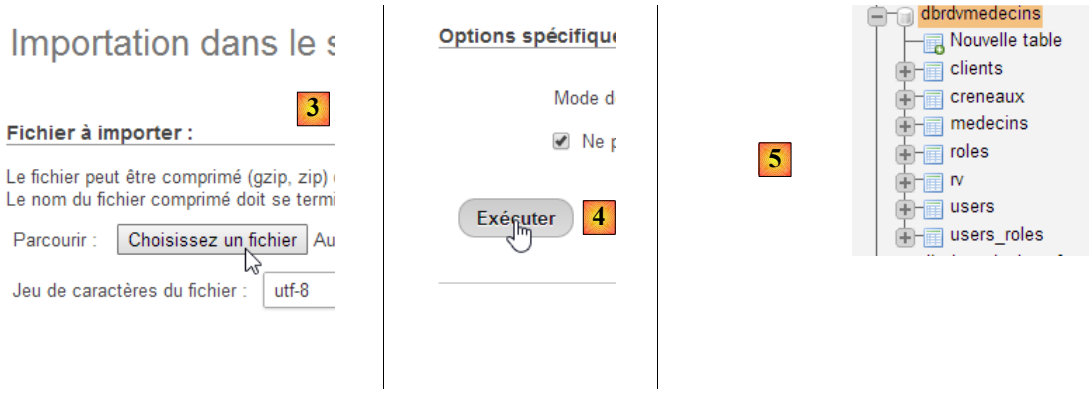




Nous utilisons l'outil [PhpMyAdmin] de WampServer :

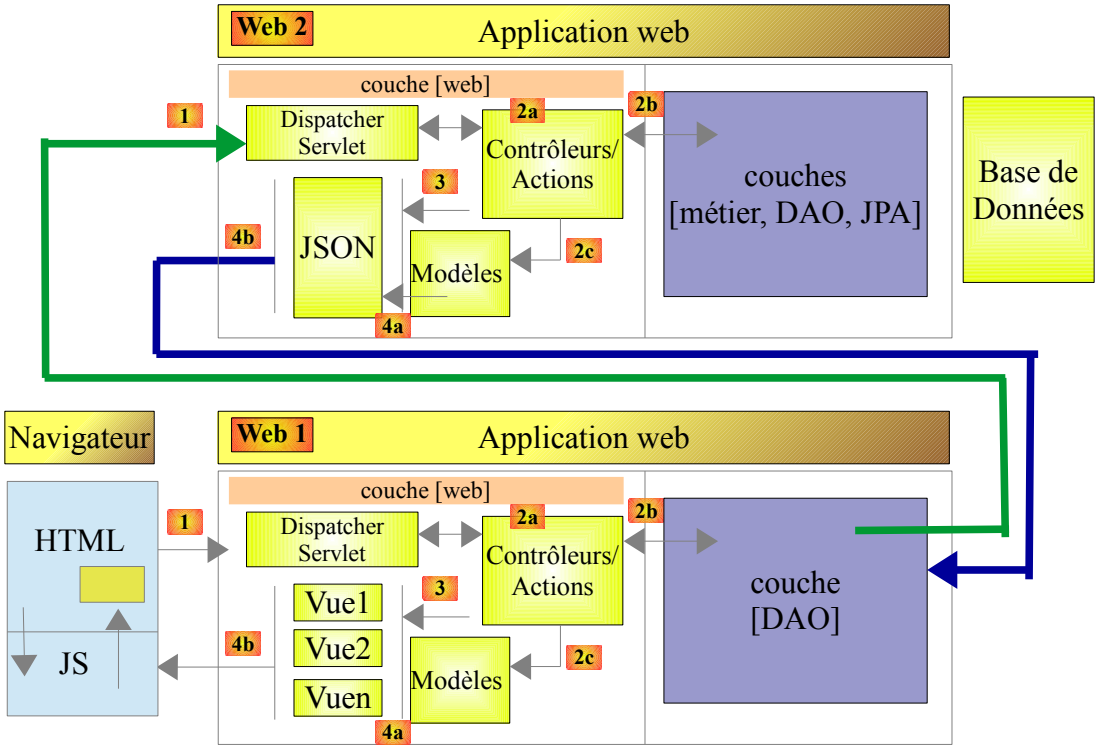


- en [1], on sélectionne l'outil [phpMyAdmin] de WampServer ;
- en [2], on choisit l'option [Importer] ;



- en [3], on sélectionne le fichier [database/dbrdvmedecins.sql] ;
- en [4], on l'exécute ;
- en [5], la base de données créée.

8.4 Le service web / jSON



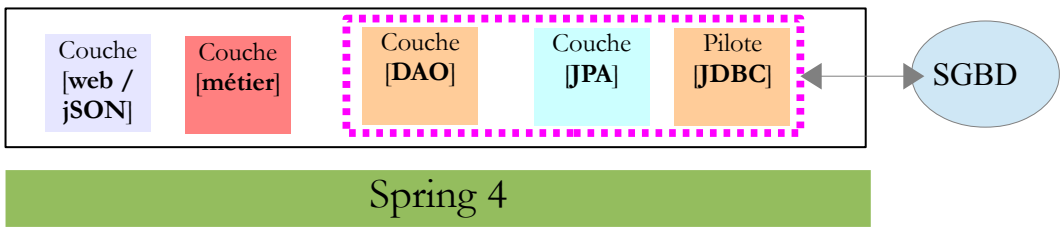
Dans l'architecture ci-dessus, nous abordons maintenant la construction du service web / jSON construit avec le framework Spring MVC. Nous allons l'écrire en plusieurs étapes :

- d'abord les couches [métier] et [DAO] (Data Access Object). Nous utiliserons ici Spring Data ;
- puis le service web jSON sans authentification. Nous utiliserons ici Spring MVC ;
- puis on ajoutera la partie authentification avec Spring Security.

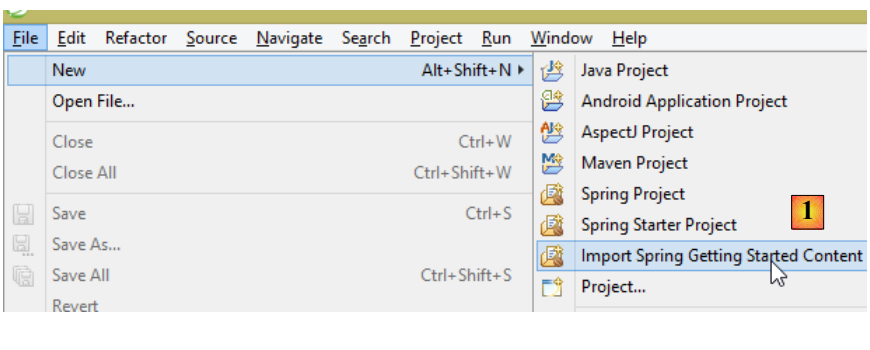
Ce qui suit est une recopie du document [http://tahe.developpez.com/angularjs-spring4/] avec cependant quelques modifications.

8.4.1 Introduction à Spring Data

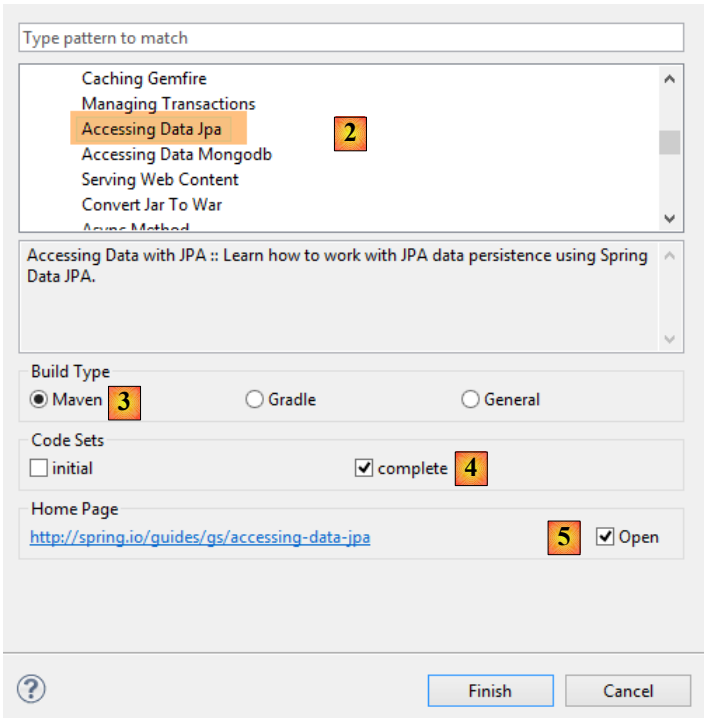
Nous allons implémenter la couche [DAO] du projet avec Spring Data, une branche de l'écosystème Spring.

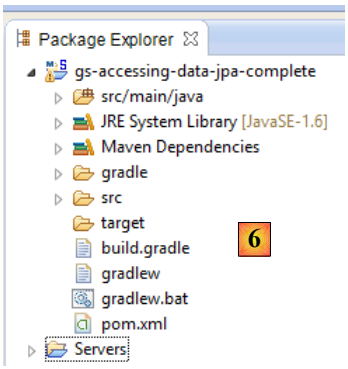


Sur le site de Spring existent de nombreux tutoriels pour démarrer avec Spring [http://spring.io/guides]. Nous allons utiliser l'un d'eux pour introduire Spring Data. Nous utilisons pour cela Spring Tool Suite (STS).



- en [1], nous importons l'un des tutoriels de [spring.io/guides] ;





- en [2], on choisit le tutoriel [Accessing Data Jpa] qui montre comment accéder à une base de données avec Spring Data ;
- en [3], on choisit un projet configuré par Maven ;
- en [4], le tutoriel peut être délivré sous deux formes : [initial] qui est une version vide qu'on remplit en suivant le tutoriel ou [complete] qui est la version finale du tutoriel. Nous choisissons cette dernière ;
- en [5], on peut choisir de visualiser le tutoriel dans un navigateur ;
- en [6], le projet final.

8.4.1.1 La configuration Maven du projet

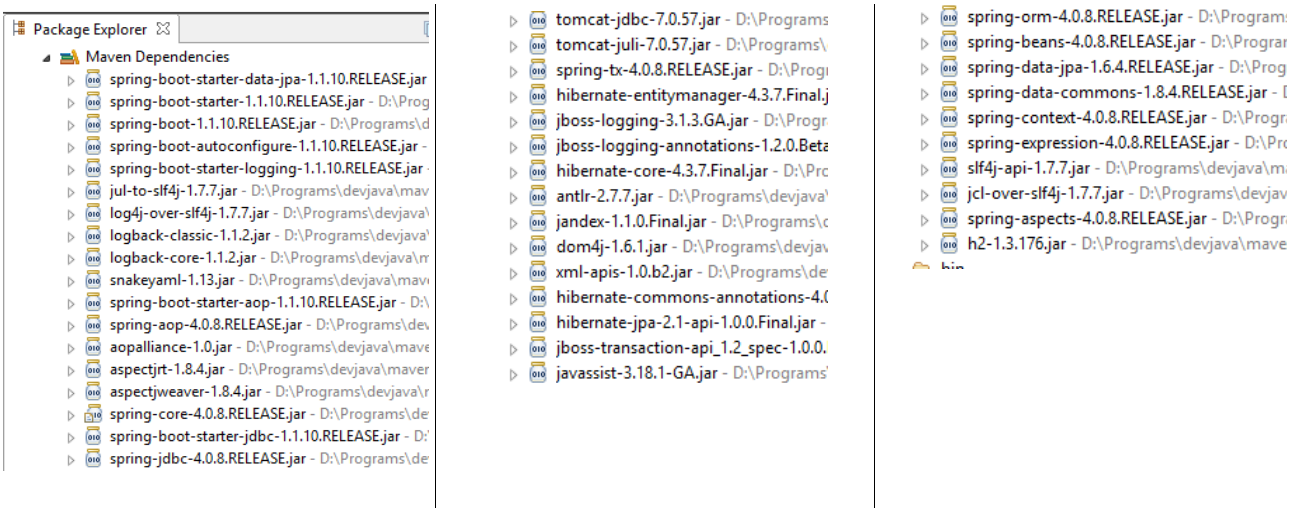
Les dépendances Maven du projet sont configurées dans le fichier [pom.xml] :

```
1.      <groupId>org.springframework</groupId>
2.      <artifactId>gs-accessing-data-jpa</artifactId>
3.      <version>0.1.0</version>
4.
5.      <parent>
6.          <groupId>org.springframework.boot</groupId>
7.          <artifactId>spring-boot-starter-parent</artifactId>
8.          <version>1.1.10.RELEASE</version>
9.      </parent>
10.
11.     <dependencies>
```

```
12.         <dependency>
13.             <groupId>org.springframework.boot</groupId>
14.             <artifactId>spring-boot-starter-data-jpa</artifactId>
15.         </dependency>
16.         <dependency>
17.             <groupId>com.h2database</groupId>
18.             <artifactId>h2</artifactId>
19.         </dependency>
20.     </dependencies>
21.
22.     <properties>
23.         <!-- use UTF-8 for everything -->
24.         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
25.         <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
26.         <start-class>hello.Application</start-class>
27. </properties>
```

- lignes 5-9 : définissent un projet Maven parent. C'est lui qui définit l'essentiel des dépendances du projet. Elles peuvent être suffisantes, auquel cas on n'en rajoute pas, ou pas, auquel cas on rajoute les dépendances manquantes ;
- lignes 12-15 : définissent une dépendance sur [spring-boot-starter-data-jpa]. Cet artifact contient les classes de Spring Data ;
- lignes 16-19 : définissent une dépendance sur le SGBD H2 qui permet de créer et gérer des bases de données en mémoire.

Regardons les classes amenées par ces dépendances :



Elles sont très nombreuses :

- certaines appartiennent à l'écosystème Spring (celles commençant par spring) ;
- d'autres appartiennent à l'écosystème Hibernate (hibernate, jboss) dont on utilise ici l'implémentation JPA ;
- d'autres sont des bibliothèques de tests (junit, hamcrest) ;
- d'autres des bibliothèques de logs (log4j, logback, slf4j) ;

Nous allons les garder toutes. Pour une application en production, il faudrait ne garder que celles qui sont nécessaires.

Ligne 26 du fichier [pom.xml] on trouve la ligne :

```
<start-class>hello.Application</start-class>
```

Cette ligne est liée aux lignes suivantes :

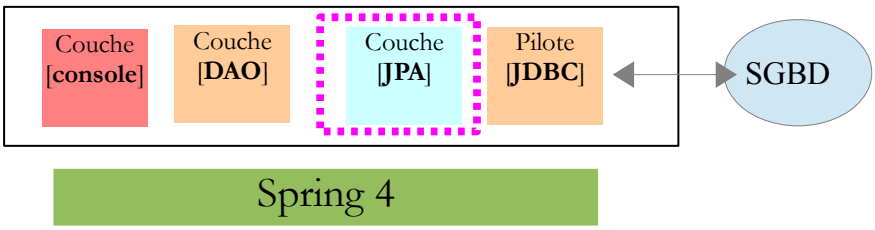
```
1.  <build>
2.          <plugins>
3.              <plugin>
4.                  <artifactId>maven-compiler-plugin</artifactId>
5.              </plugin>
6.              <plugin>
7.                  <groupId>org.springframework.boot</groupId>
8.                  <artifactId>spring-boot-maven-plugin</artifactId>
9.              </plugin>
```

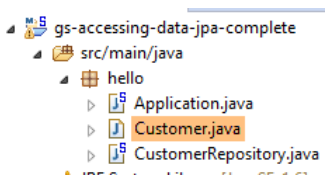
```
10.          </plugins>
11.      </build>
```

Lignes 6-9, le plugin [spring-boot-maven-plugin] permet de générer le jar exécutable de l'application. La ligne 26 du fichier [pom.xml] désigne alors la classe exécutable de ce jar.

8.4.1.2 La couche [JPA]

L'accès à la base de données se fait au travers d'une couche [JPA], Java Persistence API :





L'application est basique et gère des clients [Customer]. La classe [Customer] fait partie de la couche [JPA] et est la suivante :

```
1.  package hello;
2.
3.  import javax.persistence.Entity;
4.  import javax.persistence.GeneratedValue;
5.  import javax.persistence.GenerationType;
6.  import javax.persistence.Id;
7.
8.  @Entity
9.  public class Customer {
10.
11.     @Id
12.     @GeneratedValue(strategy = GenerationType.AUTO)
13.     private long id;
14.     private String firstName;
15.     private String lastName;
16.
17.     protected Customer() {
18.     }
19.
20.     public Customer(String firstName, String lastName) {
21.        this.firstName = firstName;
22.        this.lastName = lastName;
23.     }
24.
25.     @Override
26.     public String toString() {
27.        return String.format("Customer[id=%d, firstName='%s', lastName='%s']", id, firstName, lastName);
28.     }
29.
30. }
```

Un client a un identifiant [id], un prénom [firstName] et un nom [lastName]. Chaque instance [Customer] représente une ligne d'une table de la base de données.

- ligne 8 : annotation JPA qui fait que la persistence des instances [Customer] (Create, Read, Update, Delete) va être gérée par une implémentation JPA. D'après les dépendances Maven, on voit que c'est l'implémentation JPA / Hibernate qui est utilisée ;

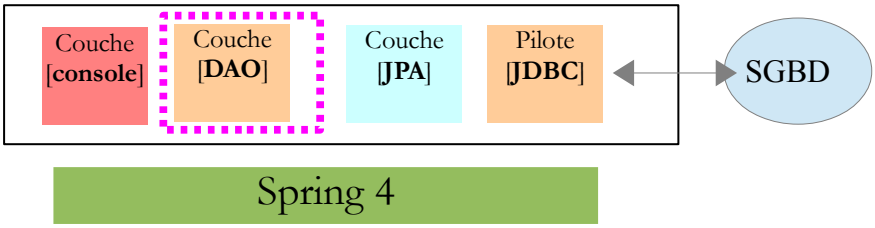
- lignes 11-12 : annotations JPA qui associent le champ [id] à la clé primaire de la table des [Customer]. La ligne 12, indique que l'implémentation JPA utilisera la méthode de génération de clé primaire propre au SGBD utilisé, ici H2 ;

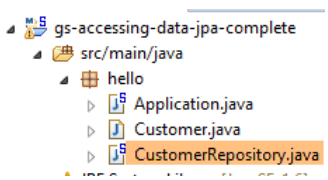
Il n'y a pas d'autres annotations JPA. Des valeurs par défaut seront alors utilisées :

- la table des [Customer] portera le nom de la classe, ç-à-d [Customer] ;
- les colonnes de cette table porteront le nom des champs de la classe : [id, firstName, lastName] sachant que la casse n'est pas prise en compte dans le nom d'une colonne de table ;

On notera qu'à aucun moment, l'implémentation JPA utilisée n'est nommée.

8.4.1.3 La couche [DAO]





La classe [CustomerRepository] implémente la couche [DAO]. Son code est le suivant :

```
1.  package hello;
2.
3.  import java.util.List;
4.
5.  import org.springframework.data.repository.CrudRepository;
6.
7.  public interface CustomerRepository extends CrudRepository<Customer, Long> {
8.
9.      List<Customer> findByLastName(String lastName);
10. }
```

C'est donc une interface et non une classe (ligne 7). Elle étend l'interface [CrudRepository], une interface de Spring Data (ligne 5). Cette interface est paramétrée par deux types : le premier est le type des éléments gérés, ici le type [Customer], le second le type de la clé primaire des éléments gérés, ici un type [Long]. L'interface [CrudRepository] est la suivante :

```
1.  package org.springframework.data.repository;
2.
3.  import java.io.Serializable;
4.
5.  @NoRepositoryBean
6.  public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {
7.
8.      <S extends T> S save(S entity);
9.
10.     <S extends T> Iterable<S> save(Iterable<S> entities);
11.
12.     T findOne(ID id);
13.
14.     boolean exists(ID id);
15.
16.     Iterable<T> findAll();
17.
18.     Iterable<T> findAll(Iterable<ID> ids);
19.
```

```
20.     long count();
21.
22.     void delete(ID id);
23.
24.     void delete(T entity);
25.
26.     void delete(Iterable<? extends T> entities);
27.
28.     void deleteAll();
29. }
```

Cette interface définit les opérations CRUD (Create – Read – Update – Delete) qu'on peut faire sur un type JPA T :

- ligne 8 : la méthode **save** permet de **persister** une entité T en base. Elle rend l'entité persistée avec la clé primaire que lui a donnée le SGBD. Elle permet également de **mettre à jour** une entité T identifiée par sa clé primaire *id*. Le choix de l'une ou l'autre action se fait selon la valeur de la clé primaire *id* : si celle-ci vaut **null** c'est l'opération de persistence qui a lieu, sinon c'est l'opération de mise à jour ;
- ligne 10 : idem mais pour une liste d'entités ;
- ligne 12 : la méthode **findOne** permet de retrouver une entité T identifiée par sa clé primaire *id* ;
- ligne 22 : la méthode **delete** permet de supprimer une entité T identifiée par sa clé primaire *id* ;
- lignes 24-28 : des variantes de la méthode [delete] ;
- ligne 16 : la méthode [findAll] permet de retrouver toutes les entités persistées T ;
- ligne 18 : idem mais limitées aux entités dont on a passé la liste des identifiants ;

Revenons à l'interface [CustomerRepository] :

```
1.  package hello;
2.
3.  import java.util.List;
4.
5.  import org.springframework.data.repository.CrudRepository;
6.
7.  public interface CustomerRepository extends CrudRepository<Customer, Long> {
8.
9.      List<Customer> findByLastName(String lastName);
10. }
```

- la ligne 9 permet de retrouver un [Customer] par son nom [lastName] ;

Et c'est tout pour la couche [DAO]. Il n'y a pas de classe d'implémentation de l'interface précédente. Celle-ci est générée à l'exécution par [Spring Data]. Les méthodes de l'interface [CrudRepository] sont automatiquement implémentées. Pour les méthodes rajoutées dans l'interface [CustomerRepository], ça dépend. Revenons à la définition de [Customer] :

```
1.      private long id;
2.      private String firstName;
3.  private String lastName;
```

La méthode de la ligne 9 est implémentée automatiquement par [Spring Data] parce qu'elle référence le champ [lastName] (ligne 3) de [Customer]. Lorsqu'il rencontre une méthode [findBySomething] dans l'interface à implémenter, Spring Data l'implémente par la requête JPQL (Java Persistence Query Language) suivante :

```
select t from T t where t.something=:value
```

Il faut donc que le type T ait un champ nommé [something]. Ainsi la méthode

```
List<Customer> findByLastName(String lastName);
```

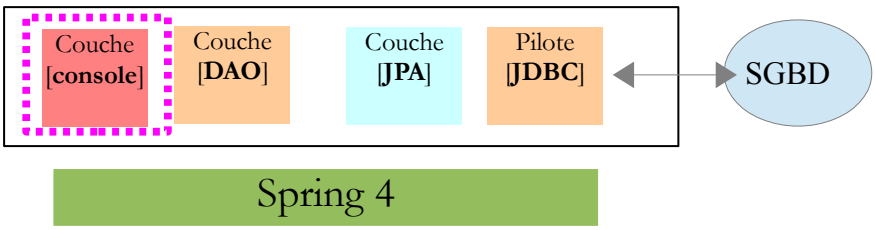
va être implémentée par un code ressemblant au suivant :

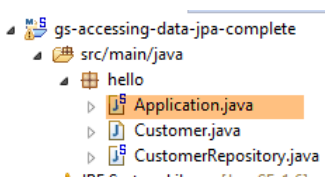
```
return [em].createQuery("select c from Customer c where
c.lastName=:value").setParameter("value",lastName).getResultList()
```

où [em] désigne le contexte de persistance JPA. Cela n'est possible que si la classe [Customer] a un champ nommé [lastName], ce qui est le cas.

En conclusion, dans les cas simples, Spring Data nous permet d'implémenter la couche [DAO] avec une simple interface.

8.4.1.4 La couche [console]





La classe [Application] est la suivante :

```
1.  package hello;
2.
3.  import java.util.List;
4.
5.  import org.springframework.boot.SpringApplication;
6.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
7.  import org.springframework.context.ConfigurableApplicationContext;
8.  import org.springframework.context.annotation.Configuration;
9.
10. @Configuration
11. @EnableAutoConfiguration
12. public class Application {
13.
14.     public static void main(String[] args) {
15.
16.         ConfigurableApplicationContext context = SpringApplication.run(Application.class);
17.         CustomerRepository repository = context.getBean(CustomerRepository.class);
18.
19.         // save a couple of customers
20.         repository.save(new Customer("Jack", "Bauer"));
21.         repository.save(new Customer("Chloe", "O'Brian"));
22.         repository.save(new Customer("Kim", "Bauer"));
23.         repository.save(new Customer("David", "Palmer"));
24.         repository.save(new Customer("Michelle", "Dessler"));
25.
26.         // fetch all customers
27.         Iterable<Customer> customers = repository.findAll();
28.         System.out.println("Customers found with findAll():");
29.         System.out.println("-------------------------------");
30.         for (Customer customer : customers) {
31.            System.out.println(customer);
32.         }
33.         System.out.println();
34.
35.         // fetch an individual customer by ID
36.         Customer customer = repository.findOne(1L);
37.         System.out.println("Customer found with findOne(1L):");
38.         System.out.println("--------------------------------");
39.         System.out.println(customer);
40.         System.out.println();
41.
42.         // fetch customers by last name
43.         List<Customer> bauers = repository.findByLastName("Bauer");
44.         System.out.println("Customer found with findByLastName('Bauer'):");
45.         System.out.println("--------------------------------------------");
46.         for (Customer bauer : bauers) {
```

```
47.           System.out.println(bauer);
48.       }
49.
50.       context.close();
51.    }
52.
53. }
```

- la ligne 10 : indique que la classe sert à configurer Spring. Les versions récentes de Spring peuvent en effet être configurées en Java plutôt qu'en XML. Les deux méthodes peuvent être utilisées simultanément. Dans le code d'une classe ayant l'annotation [Configuration] on trouve normalement des beans Spring, ç-à-d des définitions de classe à instancier. Ici aucun bean n'est défini. Il faut rappeler ici que lorsqu'on travaille avec un SGBD, divers beans Spring doivent être définis :
  - un [EntityManagerFactory] qui définit l'implémentation JPA à utiliser,
  - un [DataSource] qui définit la source de données à utiliser,
  - un [TransactionManager] qui définit le gestionnaire de transactions à utiliser ;

Ici aucun de ces beans n'est défini.

- la ligne 11 : l'annotation [EnableAutoConfiguration] est une annotation provenant du projet [Spring Boot] (lignes 5-6). Cette annotation demande à Spring Boot via la classe [SpringApplication] (ligne 16) de configurer l'application en fonction des bibliothèques trouvées dans son Classpath. Parce que les bibliothèques Hibernate sont dans le Classpath, le bean [entityManagerFactory] sera implémenté avec Hibernate. Parce que la bibliothèque du SGBD H2 est dans le Classpath, le bean [dataSource] sera implémenté avec H2. Dans le bean [dataSource], on doit définir également l'utilisateur et son mot de passe. Ici Spring Boot utilisera l'administrateur par défaut de H2, **sa** sans mot de passe. Parce que la bibliothèque [spring-tx] est dans le Classpath, c'est le gestionnaire de transactions de Spring qui sera utilisé.

  Par ailleurs, le dossier dans lequel se trouve la classe [Application] va être scanné à la recherche de beans implicitement reconnus par Spring ou définis explicitement par des annotations Spring. Ainsi les classes [Customer] et [CustomerRepository] vont-elles être inspectées. Parce que la première a l'annotation [@Entity] elle sera cataloguée comme entité à gérer par Hibernate. Parce que la seconde étend l'interface [CrudRepository] elle sera enregistrée comme bean Spring.

Examinons les lignes 16-17 du code :

```
1.  ConfigurableApplicationContext context = SpringApplication.run(Application.class);
2.  CustomerRepository repository = context.getBean(CustomerRepository.class);
```

- ligne 1 : la méthode statique [run] de la classe [SpringApplication] du projet Spring Boot est exécutée. Son paramètre est la classe qui a une annotation [Configuration] ou [EnableAutoConfiguration]. Tout ce qui a été expliqué précédemment va alors se dérouler. Le résultat est un contexte d'application Spring, ç-à-d un ensemble de beans gérés par Spring ;
- ligne 17 : on demande à ce contexte Spring, un bean implémentant l'interface [CustomerRepository]. Nous récupérons ici, la classe générée par Spring Data pour implémenter cette interface.

Les opérations qui suivent ne font qu'utiliser les méthodes du bean implémentant l'interface [CustomerRepository]. On notera ligne 50, que le contexte est fermé. Les résultats console sont les suivants :

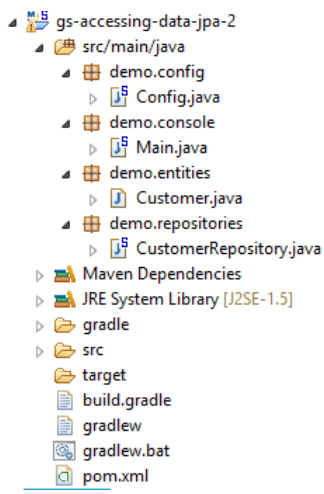
```
1.  .   ____          _            __ _ _
2.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
3.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
4.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
5.    '  |____| .__|_| |_|_| |_\__, | / / / /
6.   =========|_|==============|___/=/_/_/_/
7.   :: Spring Boot ::       (v1.1.10.RELEASE)
8.
9.  2014-12-19 11:13:46.612  INFO 10932 --- [           main] hello.Application                        :
    Starting Application on Gportpers3 with PID 10932 (started by ST in D:\data\istia-1415\spring mvc\dvp-
    final\etude-de-cas\gs-accessing-data-jpa-complete)
10. 2014-12-19 11:13:46.658  INFO 10932 --- [           main] s.c.a.AnnotationConfigApplicationContext :
    Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@279ad2e3: startup
    date [Fri Dec 19 11:13:46 CET 2014]; root of context hierarchy
11. 2014-12-19 11:13:48.234  INFO 10932 --- [           main] j.LocalContainerEntityManagerFactoryBean :
    Building JPA container EntityManagerFactory for persistence unit 'default'
12. 2014-12-19 11:13:48.258  INFO 10932 --- [           main] o.hibernate.jpa.internal.util.LogHelper  :
    HHH000204: Processing PersistenceUnitInfo [
13.     name: default
14.     ...]
15. 2014-12-19 11:13:48.337  INFO 10932 --- [           main] org.hibernate.Version                    :
    HHH000412: Hibernate Core {4.3.7.Final}
```

```
16. 2014-12-19 11:13:48.339  INFO 10932 --- [           main] org.hibernate.cfg.Environment            :
    HHH000206: hibernate.properties not found
17. 2014-12-19 11:13:48.341  INFO 10932 --- [           main] org.hibernate.cfg.Environment            :
    HHH000021: Bytecode provider name : javassist
18. 2014-12-19 11:13:48.620  INFO 10932 --- [           main] o.hibernate.annotations.common.Version   :
    HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
19. 2014-12-19 11:13:48.689  INFO 10932 --- [           main] org.hibernate.dialect.Dialect            :
    HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
20. 2014-12-19 11:13:48.853  INFO 10932 --- [           main] o.h.h.i.ast.ASTQueryTranslatorFactory    :
    HHH000397: Using ASTQueryTranslatorFactory
21. 2014-12-19 11:13:49.143  INFO 10932 --- [           main] org.hibernate.tool.hbm2ddl.SchemaExport  :
    HHH000227: Running hbm2ddl schema export
22. 2014-12-19 11:13:49.151  INFO 10932 --- [           main] org.hibernate.tool.hbm2ddl.SchemaExport  :
    HHH000230: Schema export complete
23. 2014-12-19 11:13:49.692  INFO 10932 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
    Registering beans for JMX exposure on startup
24. 2014-12-19 11:13:49.709  INFO 10932 --- [           main] hello.Application                        :
    Started Application in 3.461 seconds (JVM running for 4.435)
25. Customers found with findAll():
26. -------------------------------
27. Customer[id=1, firstName='Jack', lastName='Bauer']
28. Customer[id=2, firstName='Chloe', lastName='O'Brian']
29. Customer[id=3, firstName='Kim', lastName='Bauer']
30. Customer[id=4, firstName='David', lastName='Palmer']
31. Customer[id=5, firstName='Michelle', lastName='Dessler']
32.
33. Customer found with findOne(1L):
34. -------------------------------
35. Customer[id=1, firstName='Jack', lastName='Bauer']
36.
37. Customer found with findByLastName('Bauer'):
38. --------------------------------------------
39. Customer[id=1, firstName='Jack', lastName='Bauer']
40. Customer[id=3, firstName='Kim', lastName='Bauer']
41. 2014-12-19 11:13:49.931  INFO 10932 --- [           main] s.c.a.AnnotationConfigApplicationContext :
    Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@279ad2e3: startup date
    [Fri Dec 19 11:13:46 CET 2014]; root of context hierarchy
42. 2014-12-19 11:13:49.933  INFO 10932 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
    Unregistering JMX-exposed beans on shutdown
43. 2014-12-19 11:13:49.934  INFO 10932 --- [           main] j.LocalContainerEntityManagerFactoryBean :
    Closing JPA EntityManagerFactory for persistence unit 'default'
44. 2014-12-19 11:13:49.935  INFO 10932 --- [           main] org.hibernate.tool.hbm2ddl.SchemaExport  :
    HHH000227: Running hbm2ddl schema export
45. 2014-12-19 11:13:49.938  INFO 10932 --- [           main] org.hibernate.tool.hbm2ddl.SchemaExport  :
    HHH000230: Schema export complete
```

- lignes 1-8 : le logo du projet Spring Boot ;
- ligne 9 : la classe [hello.Application] est exécutée ;
- ligne 10 : [AnnotationConfigApplicationContext] est une classe implémentant l'interface [ApplicationContext] de Spring. C'est un conteneur de beans ;
- ligne 11 : le bean [entityManagerFactory] est implémentée avec la classe [LocalContainerEntityManagerFactory], une classe de Spring ;
- ligne 15 : on voit apparaître [Hibernate]. C'est cette implémentation JPA qui a été choisie ;
- ligne 19 : un dialecte Hibernate est la variante SQL à utiliser avec le SGBD. Ici le dialecte [H2Dialect] montre qu'Hibernate va travailler avec le SGBD H2 ;
- lignes 21-22 : la base de données est créée. La table [CUSTOMER] est créée. Cela signifie qu'Hibernate a été configuré pour générer les tables à partir des définitions JPA, ici la définition JPA de la classe [Customer] ;
- lignes 27-31 : les cinq clients insérés ;
- lignes 33635 : résultat de la méthode [findOne] de l'interface ;
- lignes 37-40 : résultats de la méthode [findByLastName] ;
- lignes 41 et suivantes : logs de la fermeture du contexte Spring.

8.4.1.5 Configuration manuelle du projet Spring Data

Nous dupliquons le projet précédent dans le projet [gs-accessing-data-jpa-2] :



Dans ce nouveau projet, nous n'allons pas nous reposer sur la configuration automatique faite par Spring Boot. Nous allons la faire manuellement. Cela peut être utile si les configurations par défaut ne nous conviennent pas.

Tout d'abord, nous allons expliciter les dépendances nécessaires dans le fichier [pom.xml] :

```
1.  ...
2.     <dependencies>
3.        <!-- Spring Core -->
4.        <dependency>
5.           <groupId>org.springframework</groupId>
6.           <artifactId>spring-core</artifactId>
7.           <version>4.1.2.RELEASE</version>
8.        </dependency>
9.        <dependency>
10.          <groupId>org.springframework</groupId>
11.          <artifactId>spring-context</artifactId>
12.          <version>4.1.2.RELEASE</version>
13.       </dependency>
14.       <dependency>
15.          <groupId>org.springframework</groupId>
16.          <artifactId>spring-beans</artifactId>
17.          <version>4.1.2.RELEASE</version>
18.       </dependency>
19.       <!-- Spring transactions -->
20.       <dependency>
21.          <groupId>org.springframework</groupId>
22.          <artifactId>spring-orm</artifactId>
23.          <version>4.1.2.RELEASE</version>
24.       </dependency>
25.       <dependency>
26.          <groupId>org.springframework</groupId>
27.          <artifactId>spring-aop</artifactId>
28.          <version>4.1.2.RELEASE</version>
29.       </dependency>
30.       <!-- Spring ORM -->
31.       <dependency>
32.          <groupId>org.springframework</groupId>
33.          <artifactId>spring-tx</artifactId>
34.          <version>4.1.2.RELEASE</version>
35.       </dependency>
36.       <!-- Spring Data -->
37.       <dependency>
38.          <groupId>org.springframework.data</groupId>
39.          <artifactId>spring-data-jpa</artifactId>
40.          <version>1.7.1.RELEASE</version>
41.       </dependency>
42.       <!-- Spring Boot -->
43.       <dependency>
44.          <groupId>org.springframework.boot</groupId>
45.          <artifactId>spring-boot</artifactId>
46.          <version>1.1.10.RELEASE</version>
```

```
47.         </dependency>
48.         <!-- Hibernate -->
49.         <dependency>
50.            <groupId>org.hibernate</groupId>
51.            <artifactId>hibernate-entitymanager</artifactId>
52.            <version>4.3.4.Final</version>
53.         </dependency>
54.         <!-- H2 Database -->
55.         <dependency>
56.            <groupId>com.h2database</groupId>
57.            <artifactId>h2</artifactId>
58.            <version>1.4.178</version>
59.         </dependency>
60.         <!-- Commons DBCP -->
61.         <dependency>
62.            <groupId>commons-dbcp</groupId>
63.            <artifactId>commons-dbcp</artifactId>
64.            <version>1.4</version>
65.         </dependency>
66.         <dependency>
67.            <groupId>commons-pool</groupId>
68.            <artifactId>commons-pool</artifactId>
69.            <version>1.6</version>
70.         </dependency>
71.      </dependencies>
72. ...
73.
74. </project>
```

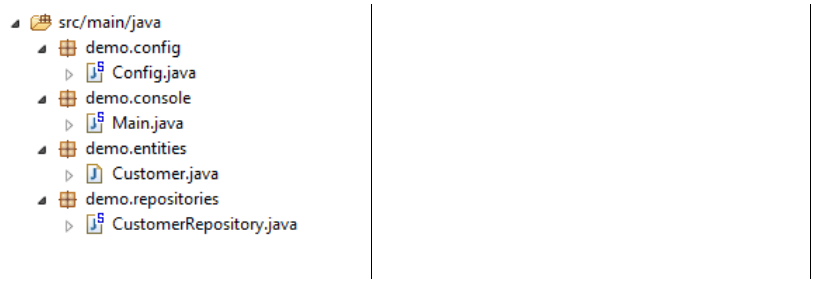
- lignes 2-18 : les bibliothèques de base de Spring ;
- lignes 19-29 : les bibliothèques de Spring pour gérer les transactions avec une base de données ;
- lignes 30-35 : la bibliothèque de Spring pour travailler avec un ORM (Object Relational Mapper) ;
- lignes 36-41 : Spring Data utilisé pour accéder à la base de données ;
- lignes 42-47 : Spring Boot pour lancer l'application ;
- lignes 54-59 : le SGBD H2 ;
- lignes 60-70 : les bases de données sont souvent utilisées avec des pools de connexions ouvertes qui évitent les ouvertures / fermetures de connexion à répétition. Ici, l'implémentation utilisée est celle de [commons-dbcp] ;

Toujours dans [pom.xml], on modifie le nom de la classe exécutable :

```
1.      <properties>
2.   ...
3.         <start-class>demo.console.Main</start-class>
4.   </properties>
```

Dans le nouveau projet, l'entité [Customer] et l'interface [CustomerRepository] ne changent pas. On va changer la classe [Application] qui va être scindée en deux classes :

- [Config] qui sera la classe de configuration :
- [Main] qui sera la classe exécutable ;



La classe exécutable [Main] est la même que précédemment sans les annotations de configuration :

```
1.  package demo.console;
2.
3.  import java.util.List;
4.
5.  import org.springframework.boot.SpringApplication;
6.  import org.springframework.context.ConfigurableApplicationContext;
```

```
7.
8.  import demo.config.Config;
9.  import demo.entities.Customer;
10. import demo.repositories.CustomerRepository;
11.
12. public class Main {
13.
14.     public static void main(String[] args) {
15.
16.         ConfigurableApplicationContext context = SpringApplication.run(Config.class);
17.         CustomerRepository repository = context.getBean(CustomerRepository.class);
18. ...
19.
20.         context.close();
21.     }
22.
23. }
```

- ligne 12 : la classe [Main] n'a plus d'annotations de configuration ;
- ligne 16 : l'application est lancée avec Spring Boot. Le paramètre [Config.class] est la nouvelle classe de configuration du projet ;

La classe [Config] qui configure le projet est la suivante :

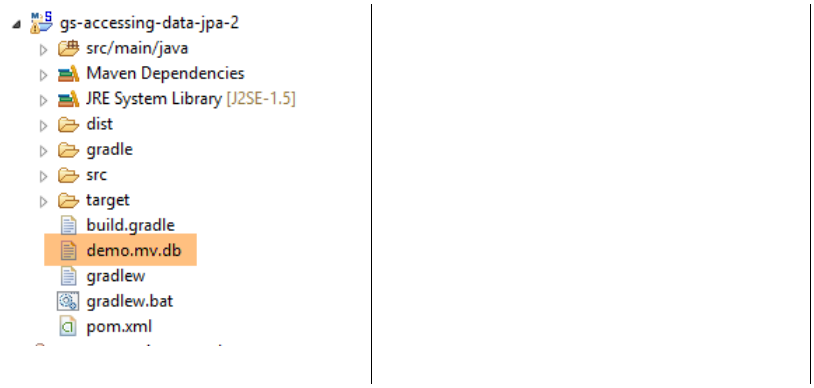
```
1.  package demo.config;
2.
3.  import javax.persistence.EntityManagerFactory;
4.  import javax.sql.DataSource;
5.
6.  import org.apache.commons.dbcp.BasicDataSource;
7.  import org.springframework.context.annotation.Bean;
8.  import org.springframework.context.annotation.Configuration;
9.  import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
10. import org.springframework.orm.jpa.JpaTransactionManager;
11. import org.springframework.orm.jpa.JpaVendorAdapter;
12. import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
13. import org.springframework.orm.jpa.vendor.Database;
14. import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
15. import org.springframework.transaction.PlatformTransactionManager;
16. import org.springframework.transaction.annotation.EnableTransactionManagement;
17.
18. //@ComponentScan(basePackages = { "demo" })
19. //@EntityScan(basePackages = { "demo.entities" })
20. @EnableTransactionManagement
21. @EnableJpaRepositories(basePackages = { "demo.repositories" })
22. @Configuration
23. public class Config {
24.     // la source de données H2
25.     @Bean
26.     public DataSource dataSource() {
27.         BasicDataSource dataSource = new BasicDataSource();
28.         dataSource.setDriverClassName("org.h2.Driver");
29.         dataSource.setUrl("jdbc:h2:./demo");
30.         dataSource.setUsername("sa");
31.         dataSource.setPassword("");
32.         return dataSource;
33.     }
34.
35.     // le provider JPA
36.     @Bean
37.     public JpaVendorAdapter jpaVendorAdapter() {
38.         HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new HibernateJpaVendorAdapter();
39.         hibernateJpaVendorAdapter.setShowSql(false);
40.         hibernateJpaVendorAdapter.setGenerateDdl(true);
41.         hibernateJpaVendorAdapter.setDatabase(Database.H2);
42.         return hibernateJpaVendorAdapter;
43.     }
44.
45.     // EntityManagerFactory
46.     @Bean
47.     public EntityManagerFactory entityManagerFactory(JpaVendorAdapter jpaVendorAdapter, DataSource
    dataSource) {
48.         LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
```

```
49.         factory.setJpaVendorAdapter(jpaVendorAdapter);
50.         factory.setPackagesToScan("demo.entities");
51.         factory.setDataSource(dataSource);
52.         factory.afterPropertiesSet();
53.         return factory.getObject();
54.     }
55.
56.     // Transaction manager
57.     @Bean
58.     public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
59.         JpaTransactionManager txManager = new JpaTransactionManager();
60.         txManager.setEntityManagerFactory(entityManagerFactory);
61.         return txManager;
62.     }
63.
64. }
```

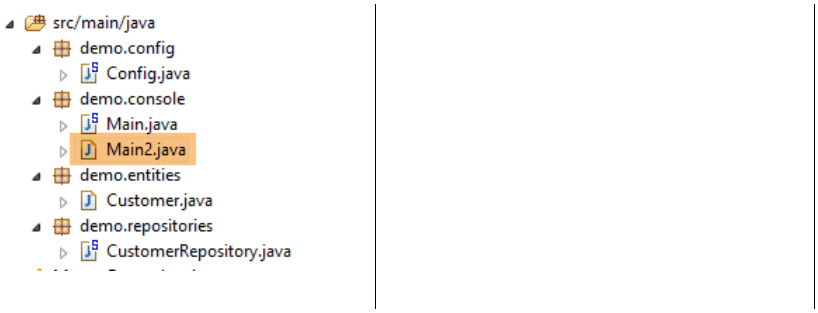
- ligne 22 : l'annotation [@Configuration] fait de la classe [Config] une classe de configuration Spring ;
- ligne 21 : l'annotation [@EnableJpaRepositories] permet de désigner les dossiers où se trouvent les interfaces Spring Data [CrudRepository]. Ces interfaces vont devenir des composants Spring et être disponibles dans son contexte ;
- ligne 20 : l'annotation [@EnableTransactionManagement] indique que les méthodes des interfaces [CrudRepository] doivent se dérouler à l'intérieur d'une transaction ;
- ligne 19 : l'annotation [@EntityScan] permet de nommer les dossiers où doivent être cherchées les entités JPA. Ici elle a été mise en commentaires, parce que cette information a été donnée explicitement ligne 50. Cette annotation devrait être présente si on utilise le mode [@EnableAutoConfiguration] et que les entités JPA ne sont pas dans le même dossier que la classe de configuration ;
- ligne 18 : l'annotation [@ComponentScan] permet de lister les dossiers où les composants Spring doivent être recherchés. Les composants Spring sont des classes taguées avec des annotations Spring telles que **@Service**, **@Component**, **@Controller**, ... Ici il n'y en a pas d'autres que ceux qui sont définis au sein de la classe [Config], aussi l'annotation a-t-elle été mise en commentaires ;
- lignes 25-33 : définissent la source de données, la base de données H2. C'est l'annotation @Bean de la ligne 25 qui fait de l'objet créé par cette méthode un composant géré par Spring. Le nom de la méthode peut être ici quelconque. Cependant elle doit être appelée [**dataSource**] si l'EntityManagerFactory de la ligne 47 est absent et défini par autoconfiguration ;
- ligne 29 : la base de données s'appellera [demo] et sera générée dans le dossier du projet ;
- lignes 36-43 : définissent l'implémentation JPA utilisée, ici une implémentation Hibernate. Le nom de la méthode peut être ici quelconque ;
- ligne 39 : pas de logs SQL ;
- ligne 30 : la base de données sera créée si elle n'existe pas ;
- lignes 46-54 : définissent l'EntityManagerFactory qui va gérer la persistance JPA. La méthode doit s'appeler obligatoirement [**entityManagerFactory**] ;
- ligne 47 : la méthode reçoit deux paramètres ayant le type des deux beans définis précédemment. Ceux-ci seront alors construits puis injectés par Spring comme paramètres de la méthode ;
- ligne 49 : fixe l'implémentation JPA utilisée ;
- ligne 50 : fixent les dossiers où trouver les entités JPA ;
- ligne 51 : fixe la source de données à gérer ;
- lignes 57-62 : le gestionnaire de transactions. La méthode doit s'appeler obligatoirement [**transactionManager**]. Elle reçoit pour paramètre le bean des lignes 46-54 ;
- ligne 60 : le gestionnaire de transactions est associé à l'EntityManagerFactory ;

Les méthodes précédentes peuvent être définies dans un ordre quelconque.

L'exécution du projet donne les mêmes résultats. Un nouveau fichier apparaît dans le dossier du projet, celui de la base de données H2 :



Enfin, on peut se passer de Spring Boot. On crée une seconde classe exécutable [Main2] :



La classe [Main2] a le code suivant :

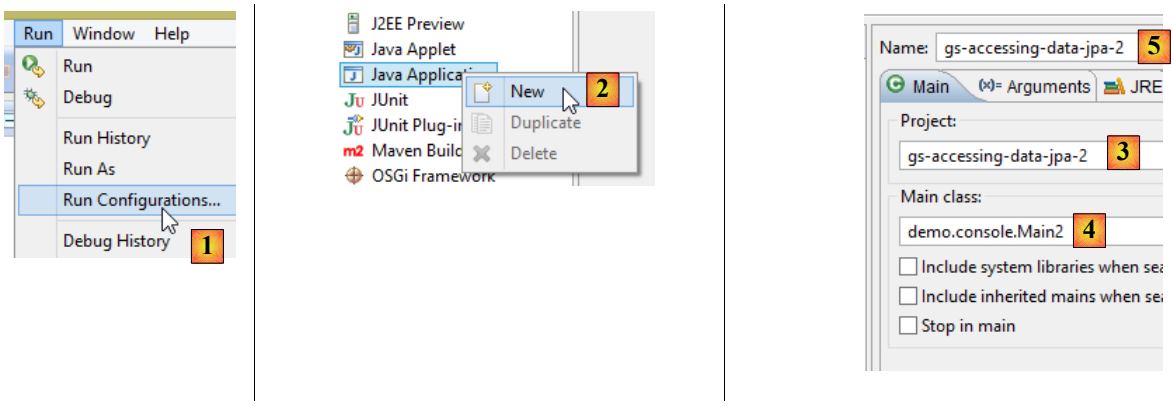
```
1.  package demo.console;
2.
3.  import java.util.List;
4.
5.  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6.
7.  import demo.config.Config;
8.  import demo.entities.Customer;
9.  import demo.repositories.CustomerRepository;
10.
11. public class Main2 {
12.
13.     public static void main(String[] args) {
14.
15.         AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
16.         CustomerRepository repository = context.getBean(CustomerRepository.class);
17. ....
18.
19.         context.close();
20.     }
21.
22. }
```

- ligne 15 : la classe de configuration [Config] est désormais exploitée par la classe Spring [AnnotationConfigApplicationContext]. On peut voir ligne 5 qu'il n'y a maintenant plus de dépendances vis à vis de Spring Boot.

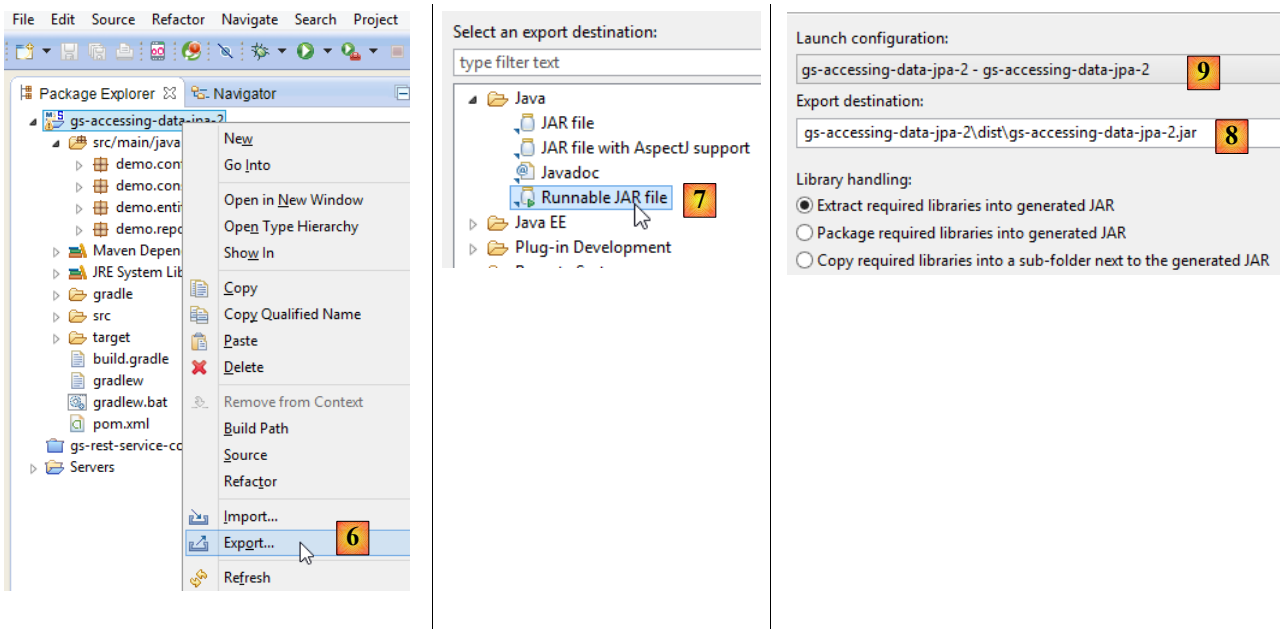
L'exécution donne les mêmes résultats que précédemment.

8.4.1.6 Création d'une archive exécutable

Pour créer une archive exécutable du projet, on peut procéder ainsi :



- en [1] : on crée une configuration d'exécution ;
- en [2] : de type [Java Application]
- en [3] : désigne le projet à exécuter (utiliser le bouton *Browse*) ;
- en [4] : désigne la classe à exécuter ;
- en [5] : le nom de la configuration d'exécution – peut être quelconque ;



- en [6] : on exporte le projet ;
- en [7] : sous la forme d'une archive JAR exécutable ;
- en [8] : indique le chemin et le nom du fichier exécutable à créer ;
- en [9] : le nom de la configuration d'exécution créée en [5] ;

Ceci fait, on ouvre une console dans le dossier contenant l'archive exécutable :

```
.....\dist>dir
12/06/2014  09:11        15 104 869 gs-accessing-data-jpa-2.jar
```

L'archive est exécutée de la façon suivante :

```
1.  .....\dist>java -jar gs-accessing-data-jpa-2.jar
```

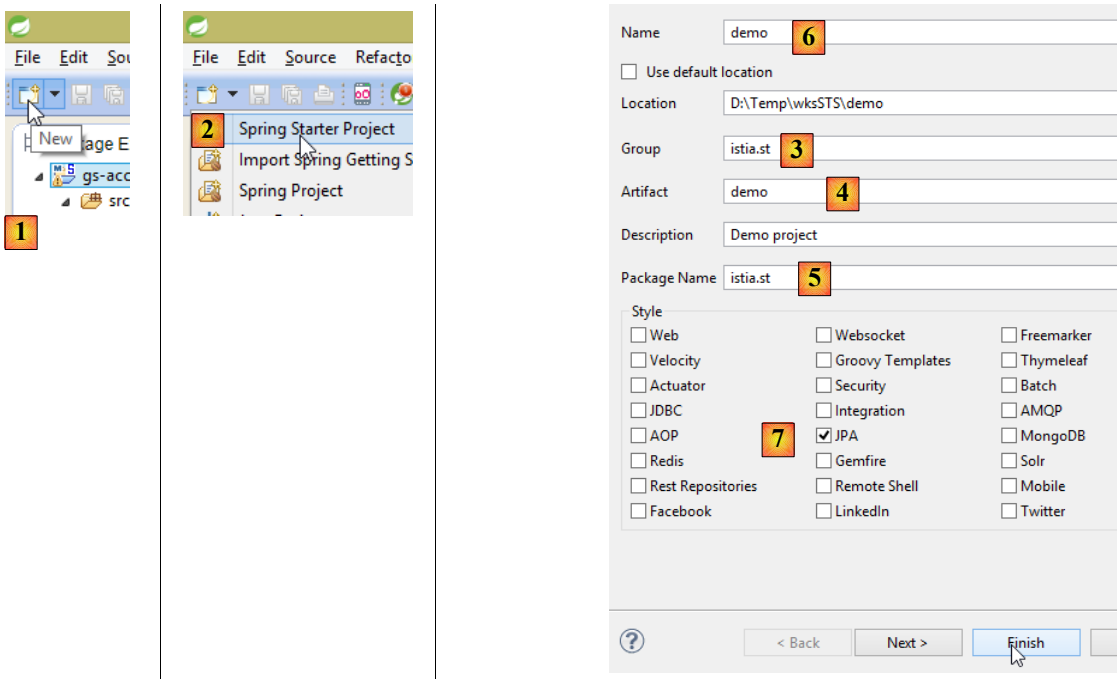
Les résultats obtenus dans la console sont les suivants :

```
2.  SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
3.  SLF4J: Defaulting to no-operation (NOP) logger implementation
4.  SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

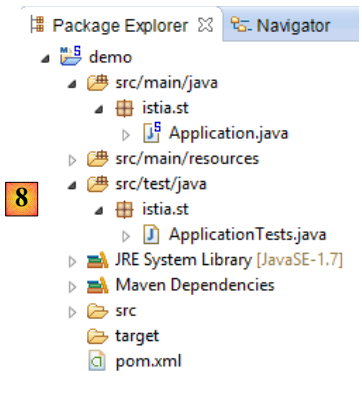
```
5.  juin 12, 2014 9:48:38 AM org.hibernate.ejb.HibernatePersistence logDeprecation
6.  WARN: HHH015016: Encountered a deprecated javax.persistence.spi.PersistenceProvider
    [org.hibernate.ejb.HibernatePersistence]; use [org.hibernate.jpa.HibernatePersistenceProvider] instead.
7.  juin 12, 2014 9:48:38 AM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
8.  INFO: HHH000204: Processing PersistenceUnitInfo [
9.           name: default
10.          ...]
11. juin 12, 2014 9:48:38 AM org.hibernate.Version logVersion
12. INFO: HHH000412: Hibernate Core {4.3.4.Final}
13. juin 12, 2014 9:48:38 AM org.hibernate.cfg.Environment <clinit>
14. INFO: HHH000206: hibernate.properties not found
15. juin 12, 2014 9:48:38 AM org.hibernate.cfg.Environment buildBytecodeProvider
16. INFO: HHH000021: Bytecode provider name : javassist
17. juin 12, 2014 9:48:39 AM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
18. INFO: HCANN000001: Hibernate Commons Annotations {4.0.4.Final}
19. juin 12, 2014 9:48:39 AM org.hibernate.dialect.Dialect <init>
20. INFO: HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
21. juin 12, 2014 9:48:39 AM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>
22. INFO: HHH000397: Using ASTQueryTranslatorFactory
23. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
24. INFO: HHH000228: Running hbm2ddl schema update
25. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
26. INFO: HHH000102: Fetching database metadata
27. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
28. INFO: HHH000396: Updating schema
29. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
30. INFO: HHH000262: Table not found: Customer
31. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
32. INFO: HHH000262: Table not found: Customer
33. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata getTableMetadata
34. INFO: HHH000262: Table not found: Customer
35. juin 12, 2014 9:48:40 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
36. INFO: HHH000232: Schema update complete
37. Customers found with findAll():
38. -------------------------------
39. Customer[id=1, firstName='Jack', lastName='Bauer']
40. Customer[id=2, firstName='Chloe', lastName='O'Brian']
41. Customer[id=3, firstName='Kim', lastName='Bauer']
42. Customer[id=4, firstName='David', lastName='Palmer']
43. Customer[id=5, firstName='Michelle', lastName='Dessler']
44.
45. Customer found with findOne(1L):
46. --------------------------------
47. Customer[id=1, firstName='Jack', lastName='Bauer']
48.
49. Customer found with findByLastName('Bauer'):
50. ----------------------------------------
51. Customer[id=1, firstName='Jack', lastName='Bauer']
52. Customer[id=3, firstName='Kim', lastName='Bauer']
```

8.4.1.7 Créer un nouveau projet Spring Data

Pour créer un squelette de projet Spring Data, on peut procéder de la façon suivante :



- en [1], on crée un nouveau projet ;
- en [2] : de type [Spring Starter Project] ;
- le projet généré sera un projet Maven. En [3], on indique le nom du groupe du projet ;
- en [4] : on indique le nom de l'artifact (un jar ici) qui sera créé par construction du projet ;
- en [5] : on indique le package de la classe exécutable qui va être créée dans le projet ;
- en [6] : le nom Eclipse du projet – peut être quelconque (n'a pas à être identique à [4]) ;
- en [7] : on indique qu'on va créer un projet ayant une couche [JPA]. Les dépendances nécessaires à un tel projet vont alors être incluses dans le fichier [pom.xml] ;



- en [8] : le projet créé ;

Le fichier [pom.xml] intègre les dépendances nécessaires à un projet JPA :

```
1.      <parent>
2.         <groupId>org.springframework.boot</groupId>
3.         <artifactId>spring-boot-starter-parent</artifactId>
4.         <version>1.2.0.RELEASE</version>
5.         <relativePath/> <!-- lookup parent from repository -->
6.      </parent>
7.
8.      <dependencies>
9.         <dependency>
10.           <groupId>org.springframework.boot</groupId>
```

```
11.           <artifactId>spring-boot-starter-data-jpa</artifactId>
12.       </dependency>
13.       <dependency>
14.           <groupId>org.springframework.boot</groupId>
15.           <artifactId>spring-boot-starter-test</artifactId>
16.           <scope>test</scope>
17.       </dependency>
18. </dependencies>
```

- lignes 9-12 : les dépendances nécessaires à JPA – vont inclure [Spring Data] ;
- lignes 13-17 : les dépendances nécessaires aux tests JUnit intégrés avec Spring ;

La classe exécutable [Application] ne fait rien mais est pré-configurée :

```
1.  package istia.st;
2.
3.  import org.springframework.boot.SpringApplication;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.  import org.springframework.context.annotation.ComponentScan;
6.  import org.springframework.context.annotation.Configuration;
7.
8.  @Configuration
9.  @ComponentScan
10. @EnableAutoConfiguration
11. public class Application {
12.
13.     public static void main(String[] args) {
14.         SpringApplication.run(Application.class, args);
15.     }
16. }
```

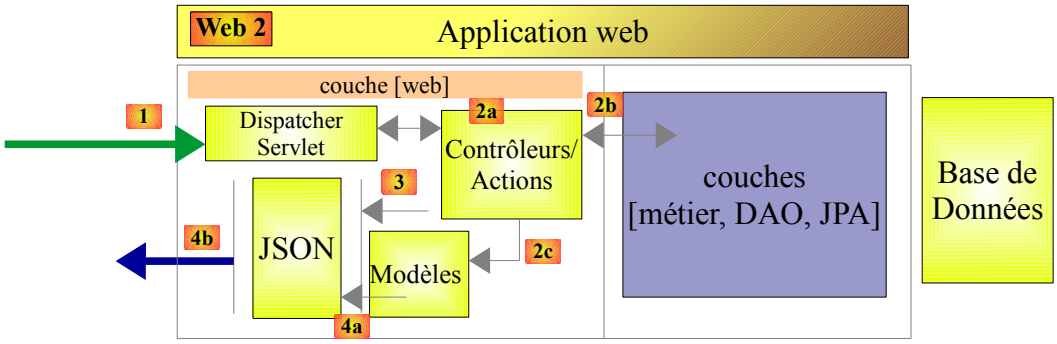
La classe de tests [ApplicationTests] ne fait rien mais est pré-configurée :

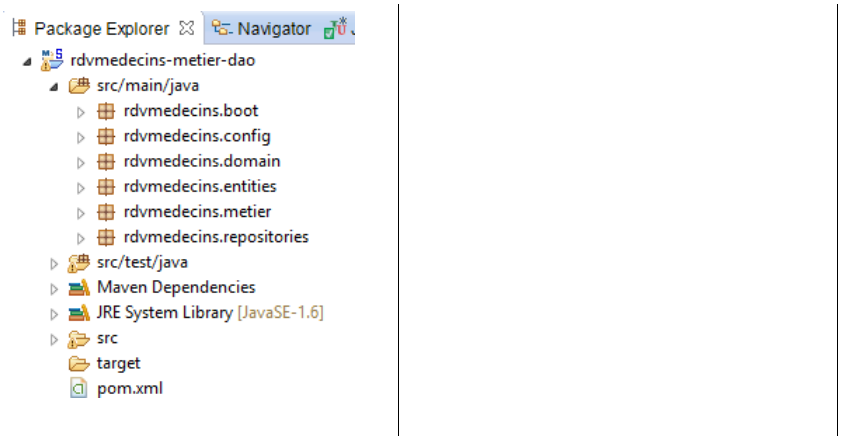
```
1.  package istia.st;
2.
3.  import org.junit.Test;
4.  import org.junit.runner.RunWith;
5.  import org.springframework.boot.test.SpringApplicationConfiguration;
6.  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
7.
8.  @RunWith(SpringJUnit4ClassRunner.class)
9.  @SpringApplicationConfiguration(classes = Application.class)
10. public class ApplicationTests {
11.
12.     @Test
13.     public void contextLoads() {
14.     }
15.
16. }
```

- ligne 9 : l'annotation [@SpringApplicationConfiguration] permet d'exploiter le fichier de configuration [Application]. La classe de test bénéficiera ainsi de tous les beans qui seront définis par ce fichier ;
- ligne 8 : l'annotation [@RunWith] permet l'intégration de Spring avec JUnit : la classe va pouvoir être exécutée comme un test JUnit. [@RunWith] est une annotation JUnit (ligne 4) alors que la classe [SpringJUnit4ClassRunner] est une classe Spring (ligne 6) ;

Maintenant que nous avons un squelette d'application JPA, nous pouvons le compléter pour écrire le projet de la couche de persistance du serveur de notre application de gestion de rendez-vous.

8.4.2 Le projet Eclipse du serveur

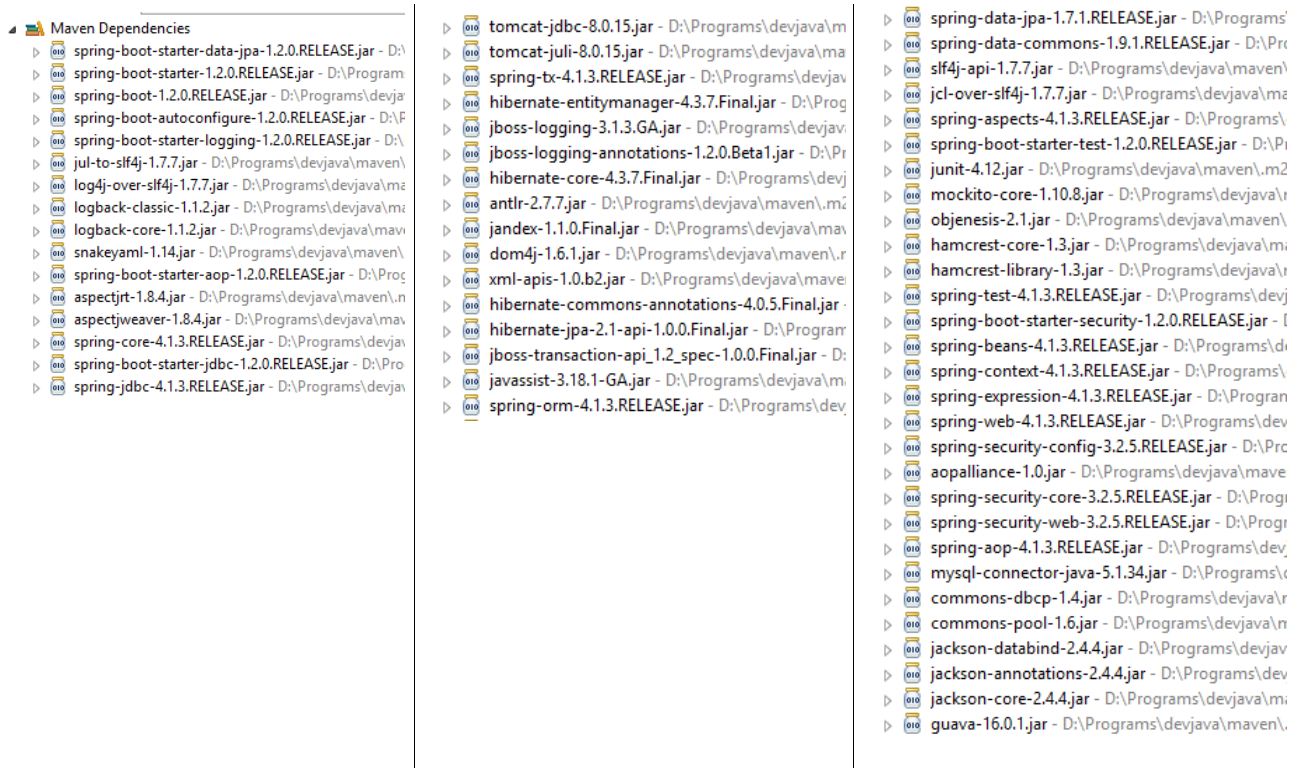




Les éléments principaux du projet sont les suivants :

- [pom.xml] : fichier de configuration Maven du projet ;
- [rdvmedecins.entities] : les entités JPA ;
- [rdvmedecins.repositories] : les interfaces Spring Data d'accès aux entités JPA ;
- [rdvmedecins.metier] : la couche [métier] ;
- [rdvmedecins.domain] : les entités manipulées par la couche [métier] ;
- [rdvmdecins.config] : les classes de configuration de la couche de persistance ;
- [rdvmedecins.boot] : une application console basique ;

8.4.3 La configuration Maven



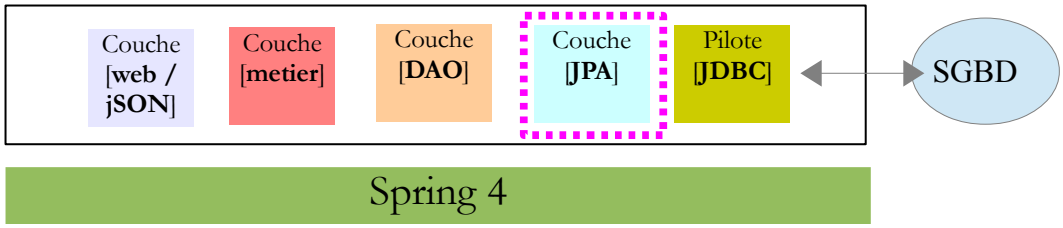
Le fichier [pom.xml] du projet est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4.      <modelVersion>4.0.0</modelVersion>
5.      <groupId>istia.st.spring4.rdvmedecins</groupId>
6.      <artifactId>rdvmedecins-metier-dao</artifactId>
7.      <version>0.0.1-SNAPSHOT</version>
8.      <parent>
9.          <groupId>org.springframework.boot</groupId>
10.         <artifactId>spring-boot-starter-parent</artifactId>
11.         <version>1.2.0.RELEASE</version>
12.     </parent>
13.     <dependencies>
14.         <dependency>
15.             <groupId>org.springframework.boot</groupId>
16.             <artifactId>spring-boot-starter-data-jpa</artifactId>
17.         </dependency>
18.         <dependency>
19.             <groupId>org.springframework.boot</groupId>
20.             <artifactId>spring-boot-starter-test</artifactId>
21.             <scope>test</scope>
22.         </dependency>
23.         <dependency>
24.             <groupId>mysql</groupId>
25.             <artifactId>mysql-connector-java</artifactId>
26.         </dependency>
27.         <dependency>
28.             <groupId>commons-dbcp</groupId>
29.             <artifactId>commons-dbcp</artifactId>
30.         </dependency>
31.         <dependency>
32.             <groupId>commons-pool</groupId>
33.             <artifactId>commons-pool</artifactId>
34.         </dependency>
35.         <dependency>
36.             <groupId>com.fasterxml.jackson.core</groupId>
37.             <artifactId>jackson-databind</artifactId>
38.         </dependency>
39.         <dependency>
40.             <groupId>com.google.guava</groupId>
```

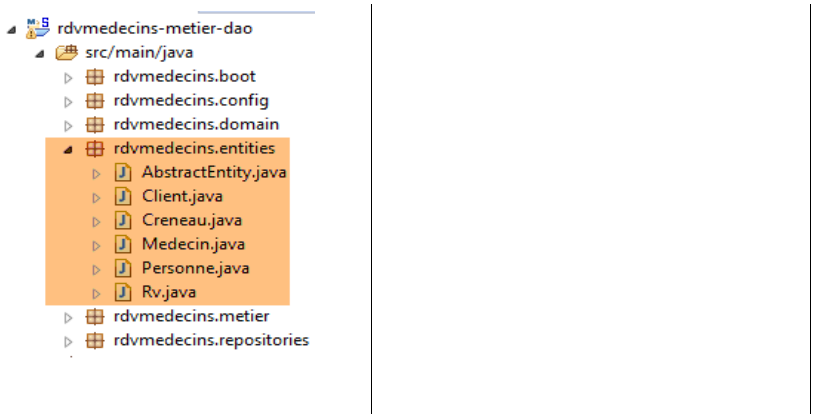
```
41.           <artifactId>guava</artifactId>
42.           <version>16.0.1</version>
43.        </dependency>
44.     </dependencies>
45.     <properties>
46.        <!-- use UTF-8 for everything -->
47.        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
48.        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
49.        <start-class>istia.st.spring.data.main.Application</start-class>
50.     </properties>
51.     <build>
52.        <plugins>
53.           <plugin>
54.              <artifactId>maven-compiler-plugin</artifactId>
55.           </plugin>
56.           <plugin>
57.              <groupId>org.springframework.boot</groupId>
58.              <artifactId>spring-boot-maven-plugin</artifactId>
59.           </plugin>
60.        </plugins>
61.     </build>
62.     <repositories>
63.        <repository>
64.           <id>spring-milestones</id>
65.           <name>Spring Milestones</name>
66.           <url>http://repo.spring.io/libs-milestone</url>
67.           <snapshots>
68.              <enabled>false</enabled>
69.           </snapshots>
70.        </repository>
71.        <repository>
72.           <id>org.jboss.repository.releases</id>
73.           <name>JBoss Maven Release Repository</name>
74.           <url>https://repository.jboss.org/nexus/content/repositories/releases</url>
75.           <snapshots>
76.              <enabled>false</enabled>
77.           </snapshots>
78.        </repository>
79.     </repositories>
80.     <pluginRepositories>
81.        <pluginRepository>
82.           <id>spring-milestones</id>
83.           <name>Spring Milestones</name>
84.           <url>http://repo.spring.io/libs-milestone</url>
85.           <snapshots>
86.              <enabled>false</enabled>
87.           </snapshots>
88.        </pluginRepository>
89.     </pluginRepositories>
90. </project>
```

- lignes 8-12 : le projet s'appuie sur le projet parent [spring-boot-starter-parent]. Pour les dépendances déjà présentes dans le projet parent, on ne précise pas de version. C'est la version définie dans le parent qui sera utilisée. Pour les autres dépendances, on les déclare normalement ;
- lignes 14-17 : pour Spring Data ;
- lignes 18-22 : pour les tests JUnit ;
- lignes 23-26 : pilote JDBC du SGBD MySQL5 ;
- lignes 27-34 : pool de connexions Commons DBCP ;
- lignes 35-38 : bibliothèque Jackson de gestion du jSON ;
- lignes 39-43 : bibliothèque Google de gestion des collections ;

8.4.4 Les entités JPA



Les entités JPA sont les objets qui vont encapsuler les lignes des tables de la base de données.



La classe [AbstractEntity] est la classe parent des entités [Personne, Creneau, Rv]. Sa définition est la suivante :

```
1.  package rdvmedecins.entities;
2.
3.  import java.io.Serializable;
4.
5.  import javax.persistence.GeneratedValue;
6.  import javax.persistence.GenerationType;
7.  import javax.persistence.Id;
8.  import javax.persistence.MappedSuperclass;
9.  import javax.persistence.Version;
10.
11. @MappedSuperclass
12. public class AbstractEntity implements Serializable {
13.
14.    private static final long serialVersionUID = 1L;
15.    @Id
16.    @GeneratedValue(strategy = GenerationType.AUTO)
17.    protected Long id;
18.    @Version
19.    protected Long version;
20.
21.    @Override
22.    public int hashCode() {
23.       int hash = 0;
24.       hash += (id != null ? id.hashCode() : 0);
25.       return hash;
26.    }
27.
28.    // initialisation
29.    public AbstractEntity build(Long id, Long version) {
30.       this.id = id;
31.       this.version = version;
32.       return this;
33.    }
34.
35.    @Override
36.    public boolean equals(Object entity) {
37.       String class1 = this.getClass().getName();
38.       String class2 = entity.getClass().getName();
39.       if (!class2.equals(class1)) {
```

```
40.             return false;
41.         }
42.         AbstractEntity other = (AbstractEntity) entity;
43.         return this.id == other.id;
44.     }
45.
46.     // getters et setters
47.     ..
48. }
```

- ligne 11 : l'annotation [@MappedSuperclass] indique que la classe annotée est parente d'entités JPA [@Entity] ;
- lignes 15-17 : définissent la clé primaire [id] de chaque entité. C'est l'annotation [@Id] qui fait du champ [id] une clé primaire. L'annotation [@GeneratedValue(strategy = GenerationType.*AUTO*)] indique que la valeur de cette clé primaire est générée par le SGBD et qu'aucun mode de génération n'est imposé ;
- lignes 18-19 : définissent la version de chaque entité. L'implémentation JPA va incrémenter ce n° de version à chaque fois que l'entité sera modifiée. Ce n° sert à empêcher la mise à jour simultanée de l'entité par deux utilisateur différents : deux utilisateurs U1 et U2 lisent l'entité E avec un n° de version égal à V1. U1 modifie E et persiste cette modification en base : le n° de version passe alors à V1+1. U2 modifie E à son tour et persiste cette modification en base : il recevra une exception car il possède une version (V1) différente de celle en base (V1+1) ;
- lignes 29-33 : la méthode [build] permet d'initialiser les deux champs de [AbstractEntity]. Cette méthode rend la référence de l'instance [AbstractEntity] ainsi initialisée ;
- lignes 36-44 : la méthode [equals] de la classe est redéfinie : deux entités seront dites égales si elles ont le même nom de classe et le même identifiant **id** ;

L'entité [Personne] est la classe parente des entités [Medecin] et [Client] :

```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Column;
4.  import javax.persistence.MappedSuperclass;
5.
6.  @MappedSuperclass
7.  public class Personne extends AbstractEntity {
8.      private static final long serialVersionUID = 1L;
9.      // attributs d'une personne
10.     @Column(length = 5)
11.     private String titre;
12.     @Column(length = 20)
13.     private String nom;
14.     @Column(length = 20)
15.     private String prenom;
16.
17.     // constructeur par défaut
18.     public Personne() {
19.     }
20.
21.     // constructeur avec paramètres
22.     public Personne(String titre, String nom, String prenom) {
23.         this.titre = titre;
24.         this.nom = nom;
25.         this.prenom = prenom;
26.     }
27.
28.     // toString
29.     public String toString() {
30.         return String.format("Personne[%s, %s, %s, %s, %s]", id, version, titre, nom, prenom);
31.     }
32.
33.     // getters et setters
34.     ...
35. }
```

- ligne 6 : l'annotation [@MappedSuperclass] indique que la classe annotée est parente d'entités JPA [@Entity] ;
- lignes 10-15 : une personne a un titre (Melle), un prénom (Jacqueline), un nom (Tatou). aucune information n'est donnée sur les colonnes de la table. Elles porteront donc par défaut les mêmes noms que les champs ;

L'entité [Medecin] est la suivante :

```
1.  package rdvmedecins.entities;
2.
```

```
3.  import javax.persistence.Entity;
4.  import javax.persistence.Table;
5.
6.  @Entity
7.  @Table(name = "medecins")
8.  public class Medecin extends Personne {
9.
10.    private static final long serialVersionUID = 1L;
11.
12.    // constructeur par défaut
13.    public Medecin() {
14.    }
15.
16.    // constructeur avec paramètres
17.    public Medecin(String titre, String nom, String prenom) {
18.       super(titre, nom, prenom);
19.    }
20.
21.    public String toString() {
22.       return String.format("Medecin[%s]", super.toString());
23.    }
24.
25. }
```

- ligne 6 : la classe est une entité JPA ;
- ligne 7 : associée à la table [MEDECINS] de la base de données ;
- ligne 8 : l'entité [Medecin] dérive de l'entité [Personne] ;

Un médecin pourra être initialisé de la façon suivante :

```
Medecin m=new Medecin("Mr","Paul","Tatou");
```

Si de plus, on veut lui affecter un identifiant et une version on pourra écrire :

```
Medecin m=new Medecin("Mr","Paul","Tatou").build(10,1);
```

où la méthode [build] est celle définie dans [AbstractEntity].

L'entité [Client] est la suivante :

```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Entity;
4.  import javax.persistence.Table;
5.
6.  @Entity
7.  @Table(name = "clients")
8.  public class Client extends Personne {
9.
10.    private static final long serialVersionUID = 1L;
11.
12.    // constructeur par défaut
13.    public Client() {
14.    }
15.
16.    // constructeur avec paramètres
17.    public Client(String titre, String nom, String prenom) {
18.       super(titre, nom, prenom);
19.    }
20.
21.    // identité
22.    public String toString() {
23.       return String.format("Client[%s]", super.toString());
24.    }
25.
26. }
```

- ligne 6 : la classe est une entité JPA ;
- ligne 7 : associée à la table [CLIENTS] de la base de données ;
- ligne 8 : l'entité [Client] dérive de l'entité [Personne] ;

L'entité [Creneau] est la suivante :

```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Column;
4.  import javax.persistence.Entity;
5.  import javax.persistence.FetchType;
6.  import javax.persistence.JoinColumn;
7.  import javax.persistence.ManyToOne;
8.  import javax.persistence.Table;
9.
10. @Entity
11. @Table(name = "creneaux")
12. public class Creneau extends AbstractEntity {
13.
14.    private static final long serialVersionUID = 1L;
15.    // caractéristiques d'un créneau de RV
16.    private int hdebut;
17.    private int mdebut;
18.    private int hfin;
19.    private int mfin;
20.
21.    // un créneau est lié à un médecin
22.    @ManyToOne(fetch = FetchType.LAZY)
23.    @JoinColumn(name = "id_medecin")
24.    private Medecin medecin;
25.
26.    // clé étrangère
27.    @Column(name = "id_medecin", insertable = false, updatable = false)
28.    private long idMedecin;
29.
30.    // constructeur par défaut
31.    public Creneau() {
32.    }
33.
34.    // constructeur avec paramètres
35.    public Creneau(Medecin medecin, int hdebut, int mdebut, int hfin, int mfin) {
36.       this.medecin = medecin;
37.       this.hdebut = hdebut;
38.       this.mdebut = mdebut;
39.       this.hfin = hfin;
40.       this.mfin = mfin;
41.    }
42.
43.    // toString
44.    public String toString() {
45.       return String.format("Créneau[%d, %d, %d, %d:%d, %d:%d]", id, version, idMedecin, hdebut, mdebut,
    hfin, mfin);
46.    }
47.
48.    // clé étrangère
49.    public long getIdMedecin() {
50.       return idMedecin;
51.    }
52.
53.    // setters - getters
54.    ...
55. }
```

- ligne 10 : la classe est une entité JPA ;
- ligne 11 : associée à la table [CRENEAUX] de la base de données ;
- ligne 12 : l'entité [Creneau] dérive de l'entité [AbstractEntity] et hérite donc de l'identifiant [id] et de la version [version] ;
- ligne 16 : heure de début du créneau (14) ;
- ligne 17 : minutes de début du créneau (20) ;
- ligne 18 : heure de fin du créneau (14) ;
- ligne 19 : minutes de fin du créneau (40) ;
- lignes 22-24 : le médecin propriétaire du créneau. La table [CRENEAUX] a une clé étrangère sur la table [MEDECINS]. Cette relation est matérialisée par les lignes 22-24 ;
- ligne 22 : l'annotation [@ManyToOne] signale une relation **plusieurs** (créneaux) **à un** (médecin). L'attribut [`fetch=FetchType.LAZY`] indique que lorsqu'on demande une entité [Creneau] au contexte de persistance et que celle-ci doit être cherchée dans la base de données, alors l'entité [Medecin] n'est pas ramenée avec elle. L'intérêt de ce mode est

que l'entité [Medecin] n'est cherchée que si le développeur le demande. On économise ainsi la mémoire et on gagne en performances ;
- ligne 23 : indique le nom de la colonne clé étrangère dans la table [CRENEAUX] ;
- lignes 27-28 : la clé étrangère sur la table [MEDECINS] ;
- ligne 27 : la colonne [ID_MEDECIN] a déjà été utilisée ligne 23. Cela veut dire qu'elle peut être modifiée par deux voies différentes ce que n'accepte pas la norme JPA. On ajoute donc les attributs [insertable = **false**, updatable = **false**], ce qui fait que la colonne ne peut qu'être lue ;

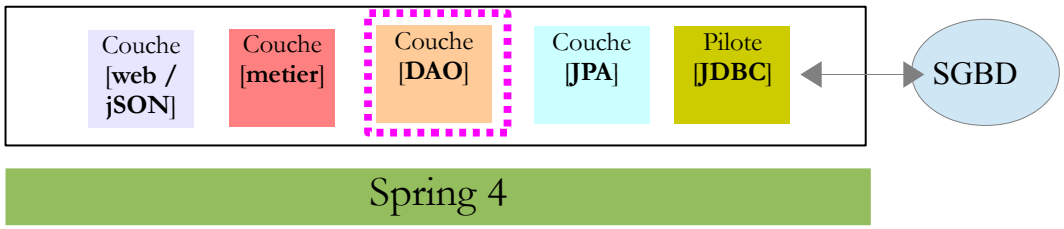
L'entité [Rv] est la suivante :

```
1.  package rdvmedecins.entities;
2.
3.  import java.util.Date;
4.
5.  import javax.persistence.Column;
6.  import javax.persistence.Entity;
7.  import javax.persistence.FetchType;
8.  import javax.persistence.JoinColumn;
9.  import javax.persistence.ManyToOne;
10. import javax.persistence.Table;
11. import javax.persistence.Temporal;
12. import javax.persistence.TemporalType;
13.
14. @Entity
15. @Table(name = "rv")
16. public class Rv extends AbstractEntity {
17.    private static final long serialVersionUID = 1L;
18.
19.    // caractéristiques d'un Rv
20.    @Temporal(TemporalType.DATE)
21.    private Date jour;
22.
23.    // un rv est lié à un client
24.    @ManyToOne(fetch = FetchType.LAZY)
25.    @JoinColumn(name = "id_client")
26.    private Client client;
27.
28.    // un rv est lié à un créneau
29.    @ManyToOne(fetch = FetchType.LAZY)
30.    @JoinColumn(name = "id_creneau")
31.    private Creneau creneau;
32.
33.    // clés étrangères
34.    @Column(name = "id_client", insertable = false, updatable = false)
35.    private long idClient;
36.    @Column(name = "id_creneau", insertable = false, updatable = false)
37.    private long idCreneau;
38.
39.    // constructeur par défaut
40.    public Rv() {
41.    }
42.
43.    // avec paramètres
44.    public Rv(Date jour, Client client, Creneau creneau) {
45.       this.jour = jour;
46.       this.client = client;
47.       this.creneau = creneau;
48.    }
49.
50.    // toString
51.    public String toString() {
52.       return String.format("Rv[%d, %s, %d, %d]", id, jour, client.id, creneau.id);
53.    }
54.
55.    // clés étrangères
56.    public long getIdCreneau() {
57.       return idCreneau;
58.    }
59.
60.    public long getIdClient() {
61.       return idClient;
62.    }
63.
```

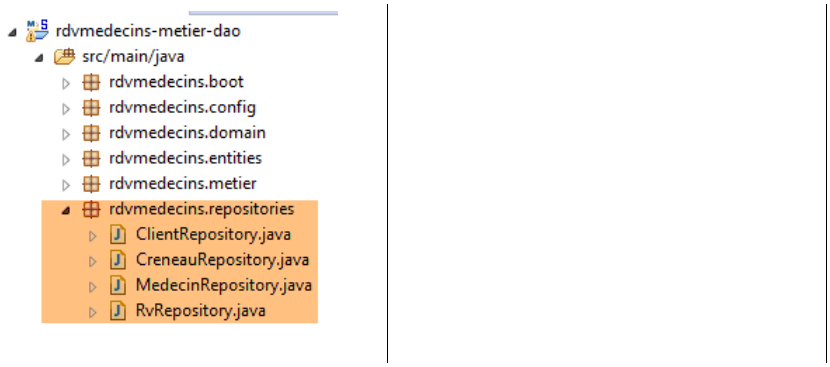
```
64.     // getters et setters
65. ...
66. }
```

- ligne 14 : la classe est une entité JPA ;
- ligne 15 : associée à la table [RV] de la base de données ;
- ligne 16 : l'entité [Rv] dérive de l'entité [AbstractEntity] et hérite donc de l'identifiant [id] et de la version [version] ;
- ligne 21 : la date du rendez-vous ;
- ligne 20 : le type [Date] de Java contient à la fois une date et une heure. Ici on précise que seule la date est utilisée ;
- lignes 24-26 : le client pour lequel ce rendez-vous a été pris. La table [RV] a une clé étrangère sur la table [CLIENTS]. Cette relation est matérialisée par les lignes 24-26 ;
- lignes 29-31 : le créneau horaire du rendez-vous. La table [RV] a une clé étrangère sur la table [CRENEAUX]. Cette relation est matérialisée par les lignes 29-31 ;
- lignes 34-35 : la clé étrangère [idClient] ;
- lignes 36-37 : la clé étrangère [idCreneau] ;

8.4.5 La couche [DAO]



Nous allons implémenter la couche [DAO] avec Spring Data :



La couche [DAO] est implémentée avec quatre interfaces Spring Data :

- [ClientRepository] : donne accès aux entités JPA [Client] ;
- [CreneauRepository] : donne accès aux entités JPA [Creneau] ;
- [MedecinRepository] : donne accès aux entités JPA [Medecin] ;
- [RvRepository] : donne accès aux entités JPA [Rv] ;

L'interface [**MedecinRepository**] est la suivante :

```
1.  package rdvmedecins.repositories;
2.
3.  import org.springframework.data.repository.CrudRepository;
4.
5.  import rdvmedecins.entities.Medecin;
6.
7.  public interface MedecinRepository extends CrudRepository<Medecin, Long> {
8.  }
```

- ligne 7 : l'interface [MedecinRepository] se contente d'hériter des méthodes de l'interface [CrudRepository] sans en ajouter d'autres ;

L'interface [**ClientRepository**] est la suivante :

```
1.  package rdvmedecins.repositories;
2.
3.  import org.springframework.data.repository.CrudRepository;
4.
5.  import rdvmedecins.entities.Client;
6.
7.  public interface ClientRepository extends CrudRepository<Client, Long> {
8.  }
```

- ligne 7 : l'interface [ClientRepository] se contente d'hériter des méthodes de l'interface [CrudRepository] sans en ajouter d'autres ;

L'interface [**CreneauRepository**] est la suivante :

```
1.  package rdvmedecins.repositories;
2.
3.  import org.springframework.data.jpa.repository.Query;
4.  import org.springframework.data.repository.CrudRepository;
5.
6.  import rdvmedecins.entities.Creneau;
7.
8.  public interface CreneauRepository extends CrudRepository<Creneau, Long> {
9.      // liste des créneaux horaires d'un médecin
10.     @Query("select c from Creneau c where c.medecin.id=?1")
11.     Iterable<Creneau> getAllCreneaux(long idMedecin);
12. }
```

- ligne 8 : l'interface [CreneauRepository] hérite des méthodes de l'interface [CrudRepository] ;
- lignes 10-11 : la méthode [getAllCreneaux] permet d'avoir les créneaux horaires d'un médecin ;
- ligne 11 : le paramètre est l'identifiant du médecin. Le résultat est une liste de créneaux horaires sous la forme d'un objet [Iterable<Creneau>] ;
- ligne 10 : l'annotation [@Query] permet de spécifier la requête JPQL (Java Persistence Query Language) qui implémente la méthode. Le paramètre [?1] sera remplacé par le paramètre [idMedecin] de la méthode ;

L'interface [**RvRepository**] est la suivante :

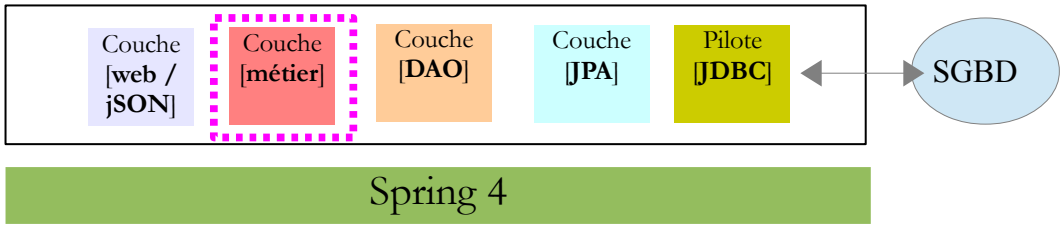
```
1.  package rdvmedecins.repositories;
2.
3.  import java.util.Date;
4.
5.  import org.springframework.data.jpa.repository.Query;
6.  import org.springframework.data.repository.CrudRepository;
7.
8.  import rdvmedecins.entities.Rv;
9.
10. public interface RvRepository extends CrudRepository<Rv, Long> {
11.
12.     @Query("select rv from Rv rv left join fetch rv.client c left join fetch rv.creneau cr where
    cr.medecin.id=?1 and rv.jour=?2")
13.     Iterable<Rv> getRvMedecinJour(long idMedecin, Date jour);
14. }
```

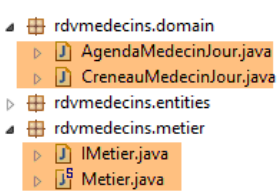
- ligne 10 : l'interface [RvRepository] hérite des méthodes de l'interface [CrudRepository] ;
- lignes 12-13 : la méthode [getRvMedecinJour] permet d'avoir les rendez-vous d'un médecin pour un jour donné ;
- ligne 13 : les paramètres sont l'identifiant du médecin et le jour. Le résultat est une liste de rendez-vous sous la forme d'un objet [Iterable<Rv>] ;
- ligne 12 : l'annotation [@Query] permet de spécifier la requête JPQL qui implémente la méthode. Le paramètre [?1] sera remplacé par le paramètre [idMedecin] de la méthode et le paramètre [?2] sera remplacé par le paramètre [jour] de la méthode. On ne peut se contenter de la requête JPQL suivante :

```
select rv from Rv rv where rv.creneau.medecin.id=?1 and rv.jour=?2
```

car les champs de la classe Rv, de types [Client] et [Creneau] sont obtenus en mode [FetchType.LAZY], ce qui signifie qu'ils doivent être demandés explicitement pour être obtenus. Ceci est fait dans la requête JPQL avec la syntaxe [left join fetch entité] qui demandent qu'une jointure soit faite avec la table sur laquelle pointe la clé étrangère afin de récupérer l'entité pointée ;

8.4.6 La couche [métier]





- [IMetier] est l'interface de la couche [métier] et [Metier] son implémentation ;
- [AgendaMedecinJour] et [CreneauMedecinJour] sont deux entités métier ;

8.4.6.1 Les entités

L'entité [CreneauMedecinJour] associe un créneau horaire et le rendez-vous éventuel pris dans ce créneau :

```
1.  package rdvmedecins.domain;
2.
3.  import java.io.Serializable;
4.
5.  import rdvmedecins.entities.Creneau;
6.  import rdvmedecins.entities.Rv;
7.
8.  public class CreneauMedecinJour implements Serializable {
9.
10.    private static final long serialVersionUID = 1L;
11.    // champs
12.    private Creneau creneau;
13.    private Rv rv;
14.
15.    // constructeurs
16.    public CreneauMedecinJour() {
17.
18.    }
19.
20.    public CreneauMedecinJour(Creneau creneau, Rv rv) {
21.       this.creneau=creneau;
22.       this.rv=rv;
23.    }
24.
25.    // toString
26.    @Override
27.    public String toString() {
28.       return String.format("[%s %s]", creneau, rv);
29.    }
30.
31.    // getters et setters
32. ...
33. }
```

- ligne 12 : le créneau horaire ;
- ligne 13 : l'éventuel rendez-vous – *null* sinon ;

L'entité [AgendaMedecinJour] est l'agenda d'un médecin pour un jour donné, ç-à-d la liste de ses rendez-vous :

```
1.  package rdvmedecins.domain;
2.
3.  import java.io.Serializable;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Date;
6.
7.  import rdvmedecins.entities.Medecin;
8.
9.  public class AgendaMedecinJour implements Serializable {
10.
11.    private static final long serialVersionUID = 1L;
12.    // champs
13.    private Medecin medecin;
14.    private Date jour;
15.    private CreneauMedecinJour[] creneauxMedecinJour;
16.
17.    // constructeurs
18.    public AgendaMedecinJour() {
19.
20.    }
21.
22.    public AgendaMedecinJour(Medecin medecin, Date jour, CreneauMedecinJour[] creneauxMedecinJour) {
23.       this.medecin = medecin;
24.       this.jour = jour;
25.       this.creneauxMedecinJour = creneauxMedecinJour;
26.    }
27.
28.    public String toString() {
29.       StringBuffer str = new StringBuffer("");
30.       for (CreneauMedecinJour cr : creneauxMedecinJour) {
31.          str.append(" ");
32.          str.append(cr.toString());
33.       }
34.       return String.format("Agenda[%s,%s,%s]", medecin, new SimpleDateFormat("dd/MM/yyyy").format(jour),
    str.toString());
35.    }
36.
37.    // getters et setters
38. ...
39. }
```

- ligne 13 : le médecin ;
- ligne 14 : le jour dans l'agenda ;
- ligne 15 : ses créneaux horaires avec ou sans rendez-vous ;

8.4.6.2 Le service

L'interface de la couche [métier] est la suivante :

```
1.  package rdvmedecins.metier;
2.
3.  import java.util.Date;
4.  import java.util.List;
5.
6.  import rdvmedecins.domain.AgendaMedecinJour;
7.  import rdvmedecins.entities.Client;
8.  import rdvmedecins.entities.Creneau;
9.  import rdvmedecins.entities.Medecin;
10. import rdvmedecins.entities.Rv;
11.
12. public interface IMetier {
13.
14.    // liste des clients
15.    public List<Client> getAllClients();
16.
17.    // liste des Médecins
18.    public List<Medecin> getAllMedecins();
```

```
19.
20.     // liste des créneaux horaires d'un médecin
21.     public List<Creneau> getAllCreneaux(long idMedecin);
22.
23.     // liste des Rv d'un médecin, un jour donné
24.     public List<Rv> getRvMedecinJour(long idMedecin, Date jour);
25.
26.     // trouver un client identifié par son id
27.     public Client getClientById(long id);
28.
29.     // trouver un client identifié par son id
30.     public Medecin getMedecinById(long id);
31.
32.     // trouver un Rv identifié par son id
33.     public Rv getRvById(long id);
34.
35.     // trouver un créneau horaire identifié par son id
36.     public Creneau getCreneauById(long id);
37.
38.     // ajouter un RV
39.     public Rv ajouterRv(Date jour, Creneau créneau, Client client);
40.
41.     // supprimer un RV
42.     public void supprimerRv(Rv rv);
43.
44.     // metier
45.     public AgendaMedecinJour getAgendaMedecinJour(long idMedecin, Date jour);
46.
47. }
```

Les commentaires expliquent le rôle de chacune des méthodes.

L'implémentation de l'interface [IMetier] est la classe [Metier] suivante :

```
1.  package rdvmedecins.metier;
2.
3.  import java.util.Date;
4.  import java.util.Hashtable;
5.  import java.util.List;
6.  import java.util.Map;
7.
8.  import org.springframework.beans.factory.annotation.Autowired;
9.  import org.springframework.stereotype.Service;
10.
11. import rdvmedecins.domain.AgendaMedecinJour;
12. import rdvmedecins.domain.CreneauMedecinJour;
13. import rdvmedecins.entities.Client;
14. import rdvmedecins.entities.Creneau;
15. import rdvmedecins.entities.Medecin;
16. import rdvmedecins.entities.Rv;
17. import rdvmedecins.repositories.ClientRepository;
18. import rdvmedecins.repositories.CreneauRepository;
19. import rdvmedecins.repositories.MedecinRepository;
20. import rdvmedecins.repositories.RvRepository;
21.
22. import com.google.common.collect.Lists;
23.
24. @Service("métier")
25. public class Metier implements IMetier {
26.
27.     // répositories
28.     @Autowired
29.     private MedecinRepository medecinRepository;
30.     @Autowired
31.     private ClientRepository clientRepository;
32.     @Autowired
33.     private CreneauRepository creneauRepository;
34.     @Autowired
35.     private RvRepository rvRepository;
36.
37.     // implémentation interface
38.     @Override
39.     public List<Client> getAllClients() {
40.         return Lists.newArrayList(clientRepository.findAll());
```

```
41.     }
42.
43.     @Override
44.     public List<Medecin> getAllMedecins() {
45.         return Lists.newArrayList(medecinRepository.findAll());
46.     }
47.
48.     @Override
49.     public List<Creneau> getAllCreneaux(long idMedecin) {
50.         return Lists.newArrayList(creneauRepository.getAllCreneaux(idMedecin));
51.     }
52.
53.     @Override
54.     public List<Rv> getRvMedecinJour(long idMedecin, Date jour) {
55.         return Lists.newArrayList(rvRepository.getRvMedecinJour(idMedecin, jour));
56.     }
57.
58.     @Override
59.     public Client getClientById(long id) {
60.         return clientRepository.findOne(id);
61.     }
62.
63.     @Override
64.     public Medecin getMedecinById(long id) {
65.         return medecinRepository.findOne(id);
66.     }
67.
68.     @Override
69.     public Rv getRvById(long id) {
70.         return rvRepository.findOne(id);
71.     }
72.
73.     @Override
74.     public Creneau getCreneauById(long id) {
75.         return creneauRepository.findOne(id);
76.     }
77.
78.     @Override
79.     public Rv ajouterRv(Date jour, Creneau créneau, Client client) {
80.         return rvRepository.save(new Rv(jour, client, créneau));
81.     }
82.
83.     @Override
84.     public void supprimerRv(Rv rv) {
85.         rvRepository.delete(rv.getId());
86.     }
87.
88.     public AgendaMedecinJour getAgendaMedecinJour(long idMedecin, Date jour) {
89.     ...
90.     }
91.
92. }
```

- ligne 24 : l'annotation [@Service] est une annotation Spring qui fait de la classe annotée un composant géré par Spring. On peut ou non donner un nom à un composant. Celui-ci est nommé [métier] ;
- ligne 25 : la classe [Metier] implémente l'interface [IMetier] ;
- ligne 28 : l'annotation [@Autowired] est une annotation Spring. La valeur du champ ainsi annoté sera initialisée (injectée) par Spring avec la référence d'un composant Spring du type ou du nom précisés. Ici l'annotation [@Autowired] ne précise pas de nom. Ce sera donc une injection par type qui sera faite ;
- ligne 29 : le champ [medecinRepository] sera initialisé avec la référence d'un composant Spring de type [MedecinRepository]. Ce sera la référence de la classe générée par Spring Data pour implémenter l'interface [MedecinRepository] que nous avons déjà présentée ;
- lignes 30-35 : ce processus est répété pour les trois autres interfaces étudiées ;
- lignes 39-41 : implémentation de la méthode [getAllClients] ;
- ligne 40 : nous utilisons la méthode [findAll] de l'interface [ClientRepository]. Cette méthode rend un type [Iterable<Client>] que nous transformons en [List<Client>] avec la méthode statique [Lists.newArrayList]. La classe [Lists] est définie dans la bibliothèque Google Guava. Dans [pom.xml] cette dépendance a été importée :

```
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>16.0.1</version>
```

```
        </dependency>
```

- lignes 38-86 : les méthodes de l'interface [IMetier] sont implémentées avec l'aide des classes de la couche [DAO] ;

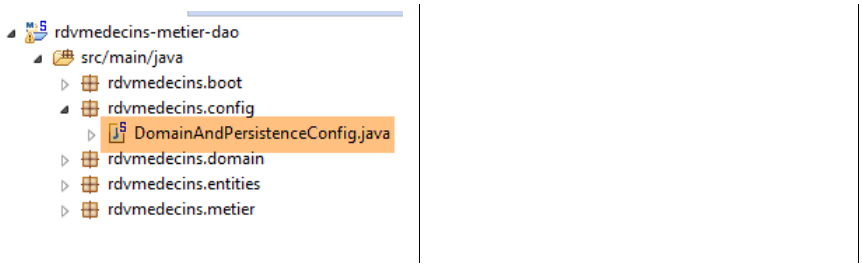
Seule la méthode de la ligne 88 est spécifique à la couche [métier]. Elle a été placée ici parce qu'elle fait un traitement métier qui n'est pas qu'un simple accès aux données. Sans cette méthode, il n'y avait pas de raison de créer une couche [métier]. La méthode [getAgendaMedecinJour] est la suivante :

```
1.  public AgendaMedecinJour getAgendaMedecinJour(long idMedecin, Date jour) {
2.        // liste des créneaux horaires du médecin
3.        List<Creneau> creneauxHoraires = getAllCreneaux(idMedecin);
4.        // liste des réservations de ce même médecin pour ce même jour
5.        List<Rv> reservations = getRvMedecinJour(idMedecin, jour);
6.        // on crée un dictionnaire à partir des Rv pris
7.        Map<Long, Rv> hReservations = new Hashtable<Long, Rv>();
8.        for (Rv resa : reservations) {
9.           hReservations.put(resa.getCreneau().getId(), resa);
10.       }
11.       // on crée l'agenda pour le jour demandé
12.       AgendaMedecinJour agenda = new AgendaMedecinJour();
13.       // le médecin
14.       agenda.setMedecin(getMedecinById(idMedecin));
15.       // le jour
16.       agenda.setJour(jour);
17.       // les créneaux de réservation
18.       CreneauMedecinJour[] creneauxMedecinJour = new CreneauMedecinJour[creneauxHoraires.size()];
19.       agenda.setCreneauxMedecinJour(creneauxMedecinJour);
20.       // remplissage des créneaux de réservation
21.       for (int i = 0; i < creneauxHoraires.size(); i++) {
22.          // ligne i agenda
23.          creneauxMedecinJour[i] = new CreneauMedecinJour();
24.          // créneau horaire
25.          Creneau créneau = creneauxHoraires.get(i);
26.          long idCreneau = créneau.getId();
27.          creneauxMedecinJour[i].setCreneau(créneau);
28.          // le créneau est-il libre ou réservé ?
29.          if (hReservations.containsKey(idCreneau)) {
30.             // le créneau est occupé - on note la résa
31.             Rv resa = hReservations.get(idCreneau);
32.             creneauxMedecinJour[i].setRv(resa);
33.          }
34.       }
35.       // on rend le résultat
36.       return agenda;
37.    }
```

Le lecteur est invité à lire les commentaires. L'algorithme est le suivant :

- on récupère tous les créneaux horaires du médecin indiqué ;
- on récupère tous ses rendez-vous pour le jour indiqué ;
- avec ces deux informations, on est capable de dire si un créneau horaire est libre ou occupé ;

8.4.7 La configuration du projet



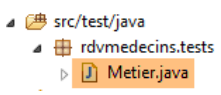
La classe [DomainAndPersistenceConfig] configure l'ensemble du projet :

```
1.  package rdvmedecins.config;
2.
3.  import javax.sql.DataSource;
```

```
4.
5.  import org.apache.commons.dbcp.BasicDataSource;
6.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
7.  import org.springframework.boot.orm.jpa.EntityScan;
8.  import org.springframework.context.annotation.Bean;
9.  import org.springframework.context.annotation.ComponentScan;
10. import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
11. import org.springframework.orm.jpa.JpaVendorAdapter;
12. import org.springframework.orm.jpa.vendor.Database;
13. import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
14. import org.springframework.transaction.annotation.EnableTransactionManagement;
15.
16. @EnableJpaRepositories(basePackages = { "rdvmedecins.repositories" })
17. @EnableAutoConfiguration
18. @ComponentScan(basePackages = { "rdvmedecins" })
19. @EntityScan(basePackages = { "rdvmedecins.entities" })
20. @EnableTransactionManagement
21. public class DomainAndPersistenceConfig {
22.
23.     // la source de données MySQL
24.     @Bean
25.     public DataSource dataSource() {
26.         BasicDataSource dataSource = new BasicDataSource();
27.         dataSource.setDriverClassName("com.mysql.jdbc.Driver");
28.         dataSource.setUrl("jdbc:mysql://localhost:3306/dbrdvmedecins");
29.         dataSource.setUsername("root");
30.         dataSource.setPassword("");
31.         return dataSource;
32.     }
33.
34.     // le provider JPA - n'est pas nécessaire si on est satisfait des valeurs par défaut utilisées par
    Spring boot
35.     // ici on le définit pour activer / désactiver les logs SQL
36.     @Bean
37.     public JpaVendorAdapter jpaVendorAdapter() {
38.         HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new HibernateJpaVendorAdapter();
39.         hibernateJpaVendorAdapter.setShowSql(false);
40.         hibernateJpaVendorAdapter.setGenerateDdl(false);
41.         hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
42.         return hibernateJpaVendorAdapter;
43.     }
44.
45.     // l'EntityManagerFactory et le TransactionManager sont définis avec des valeurs par défaut par Spring
    boot
46.
47. }
```

- lignes 45 : nous n'allons pas définir les beans [EntityManagerFactory] et [TransactionManager]. Nous allons pour cela nous appuyer sur l'annotation [@EnableAutoConfiguration] de Spring Boot (ligne 17) ;
- lignes 24-32 : définissent la source de données MySQL5. C'est un bean qui en général ne peut être deviné par Spring Boot ;
- lignes 36-43 : nous configurons également l'implémentation JPA pour mettre l'attribut [showSql] d'Hibernate à faux (ligne 39). Par défaut, il est à vrai ;
- pour l'instant, les seuls composants gérés par Spring sont les beans des lignes 25 et 37 plus les beans [EntityManagerFactory] et [TransactionManager] par autoconfiguration. Il nous faut ajouter les beans des couches [métier] et [DAO] ;
- la ligne 16 ajoute au contexte de Spring les interfaces du package [rdvmdecins.repositories] qui héritent de l'interface [CrudRepository] ;
- la ligne 18 ajoute au contexte de Spring toutes les classes du package [rdvmedecins] et ses descendants ayant une annotation Spring. Dans le package [rdvmdecins.metier], la classe [Metier] avec son annotation [@Service] va être trouvée et ajoutée au contexte Spring ;
- ligne 45 : un bean [entityManagerFactory] va être défini par défaut par Spring Boot. On doit indiquer à ce bean où se trouvent les entités JPA qu'il doit gérer. C'est la ligne 19 qui fait cela ;
- ligne 20 : indique que les méthodes des interfaces héritant de l'interface [CrudRepository] doivent être exécutées au sein d'une transaction ;

8.4.8 Les tests de la couche [métier]



La classe [rdvmedecins.tests.Metier] est une classe de test Spring / JUnit 4 :

```
1.  package rdvmedecins.tests;
2.
3.  import java.text.ParseException;
4.  import java.util.Date;
5.  import java.util.List;
6.
7.  import org.junit.Assert;
8.  import org.junit.Test;
9.  import org.junit.runner.RunWith;
10. import org.springframework.beans.factory.annotation.Autowired;
11. import org.springframework.boot.test.SpringApplicationConfiguration;
12. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
13.
14. import rdvmedecins.config.DomainAndPersistenceConfig;
15. import rdvmedecins.domain.AgendaMedecinJour;
16. import rdvmedecins.entities.Client;
17. import rdvmedecins.entities.Creneau;
18. import rdvmedecins.entities.Medecin;
19. import rdvmedecins.entities.Rv;
20. import rdvmedecins.metier.IMetier;
21.
22. @SpringApplicationConfiguration(classes = DomainAndPersistenceConfig.class)
23. @RunWith(SpringJUnit4ClassRunner.class)
24. public class Metier {
25.
26.    @Autowired
27.    private IMetier métier;
28.
29.    @Test
30.    public void test1(){
31.       // affichage clients
32.       List<Client> clients = métier.getAllClients();
33.       display("Liste des clients :", clients);
34.       // affichage médecins
35.       List<Medecin> medecins = métier.getAllMedecins();
36.       display("Liste des médecins :", medecins);
37.       // affichage créneaux d'un médecin
38.       Medecin médecin = medecins.get(0);
39.       List<Creneau> creneaux = métier.getAllCreneaux(médecin.getId());
40.       display(String.format("Liste des créneaux du médecin %s", médecin), creneaux);
41.       // liste des Rv d'un médecin, un jour donné
42.       Date jour = new Date();
43.       display(String.format("Liste des rv du médecin %s, le [%s]", médecin, jour),
    métier.getRvMedecinJour(médecin.getId(), jour));
44.       // ajouter un RV
45.       Rv rv = null;
46.       Creneau créneau = creneaux.get(2);
47.       Client client = clients.get(0);
48.       System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s", jour,
    créneau,
49.           client));
50.       rv = métier.ajouterRv(jour, créneau, client);
51.       // vérification
52.       Rv rv2 = métier.getRvById(rv.getId());
53.       Assert.assertEquals(rv, rv2);
54.       display(String.format("Liste des Rv du médecin %s, le [%s]", médecin, jour),
    métier.getRvMedecinJour(médecin.getId(), jour));
55.       // ajouter un RV dans le même créneau du même jour
56.       // doit provoquer une exception
57.       System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s", jour,
    créneau,
58.           client));
59.       Boolean erreur = false;
60.       try {
61.          rv = métier.ajouterRv(jour, créneau, client);
62.          System.out.println("Rv ajouté");
```

```
63.         } catch (Exception ex) {
64.            Throwable th = ex;
65.            while (th != null) {
66.               System.out.println(ex.getMessage());
67.               th = th.getCause();
68.            }
69.            // on note l'erreur
70.            erreur = true;
71.         }
72.         // on vérifie qu'il y a eu une erreur
73.         Assert.assertTrue(erreur);
74.         // liste des RV
75.         display(String.format("Liste des Rv du médecin %s, le [%s]", médecin, jour),
    métier.getRvMedecinJour(médecin.getId(), jour));
76.         // affichage agenda
77.         AgendaMedecinJour agenda = métier.getAgendaMedecinJour(médecin.getId(), jour);
78.         System.out.println(agenda);
79.         Assert.assertEquals(rv, agenda.getCreneauxMedecinJour()[2].getRv());
80.         // supprimer un RV
81.         System.out.println("Suppression du Rv ajouté");
82.         métier.supprimerRv(rv);
83.         // vérification
84.         rv2 = métier.getRvById(rv.getId());
85.         Assert.assertNull(rv2);
86.         display(String.format("Liste des Rv du médecin %s, le [%s]", médecin, jour),
    métier.getRvMedecinJour(médecin.getId(), jour));
87.      }
88.
89.      // méthode utilitaire - affiche les éléments d'une collection
90.      private void display(String message, Iterable<?> elements) {
91.         System.out.println(message);
92.         for (Object element : elements) {
93.            System.out.println(element);
94.         }
95.      }
96.
97. }
```

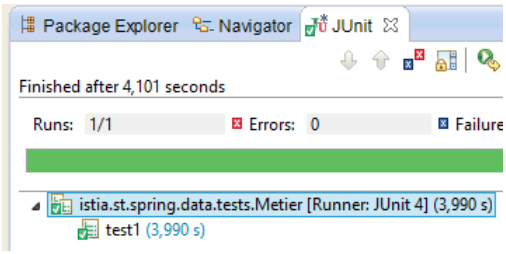
- ligne 22 : l'annotation [@SpringApplicationConfiguration] permet d'exploiter le fichier de configuration [DomainAndPersistenceConfig] étudié précédemment. La classe de test bénéficie ainsi de tous les beans définis par ce fichier ;
- ligne 23 : l'annotation [@RunWith] permet l'intégration de Spring avec JUnit : la classe va pouvoir être exécutée comme un test JUnit. [@RunWith] est une annotation JUnit (ligne 9) alors que la classe [SpringJUnit4ClassRunner] est une classe Spring (ligne 12) ;
- lignes 26-27 : injection dans la classe de test d'une référence sur la couche [métier] ;
- beaucoup de tests ne sont que de simples tests visuels :
  - lignes 32-33 : liste des clients ;
  - lignes 35-36 : liste des médecins ;
  - lignes 39-40 : liste des créneaux d'un médecin ;
  - ligne 43 : liste des rendez-vous d'un médecin ;
- ligne 50 : ajout d'un nouveau rendez-vous. La méthode [ajouterRv] rend le rendez-vous avec une information supplémentaire, sa clé primaire **id** ;
- ligne 53 : on utilise cette clé primaire pour rechercher le rendez-vous en base ;
- ligne 54 : on vérifie que le rendez-vous cherché et le rendez-vous trouvé sont les mêmes. On rappelle que la méthode [equals] de l'entité [Rv] a été redéfinie : deux rendez-vous sont égaux s'ils ont le même **id**. Ici, cela nous montre que le rendez-vous ajouté a bien été mis en base ;
- lignes 61-73 : on essaie d'ajouter une deuxième fois le même rendez-vous. Cela doit être rejeté par le SGBD car on a une contrainte d'unicité :

```
1. CREATE TABLE IF NOT EXISTS `rv` (
2.   `ID` bigint(20) NOT NULL AUTO_INCREMENT,
3.   `JOUR` date NOT NULL,
4.   `ID_CLIENT` bigint(20) NOT NULL,
5.   `ID_CRENEAU` bigint(20) NOT NULL,
6.   `VERSION` int(11) NOT NULL DEFAULT '0',
7.   PRIMARY KEY (`ID`),
8.   UNIQUE KEY `UNQ1_RV` (`JOUR`,`ID_CRENEAU`),
9.   KEY `FK_RV_ID_CRENEAU` (`ID_CRENEAU`),
10.  KEY `FK_RV_ID_CLIENT` (`ID_CLIENT`)
11.) ENGINE=InnoDB  DEFAULT CHARSET=utf8 COLLATE=utf8_swedish_ci AUTO_INCREMENT=60 ;
```

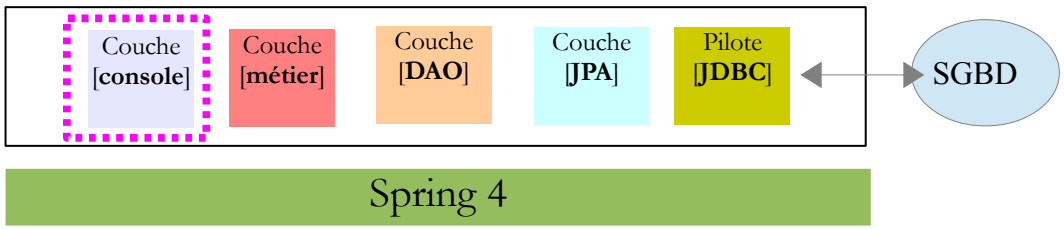
La ligne 8 ci-dessus indique que la combinaison [JOUR, ID_CRENEAU] doit être unique, ce qui empêche de mettre deux rendez-vous le même jour dans le même créneau horaire.

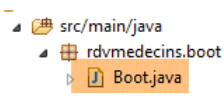
- ligne 73 : on vérifie qu'une exception s'est bien produite ;
- ligne 77 : on demande l'agenda du médecin pour lequel on vient d'ajouter un rendez-vous ;
- ligne 79 : on vérifie que le rendez-vous ajouté est bien présent dans son agenda ;
- ligne 82 : on supprime le rendez-vous ajouté ;
- ligne 84 : on va chercher en base le rendez-vous supprimé ;
- ligne 85 : on vérifie qu'on a récupéré un pointeur *null*, montrant par là que le rendez-vous cherché n'existe pas ;

L'exécution du test réussit :



8.4.9 Le programme console





Le programme console est basique. Il illustre comment récupérer une clé étrangère :

```
1.  package rdvmedecins.boot;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.
6.  import org.springframework.boot.SpringApplication;
7.  import org.springframework.context.ConfigurableApplicationContext;
8.
9.  import rdvmedecins.config.DomainAndPersistenceConfig;
10. import rdvmedecins.entities.Client;
11. import rdvmedecins.entities.Creneau;
12. import rdvmedecins.entities.Rv;
13. import rdvmedecins.metier.IMetier;
14.
15. public class Boot {
16.     // le boot
17.     public static void main(String[] args) {
18.         // on prépare la configuration
19.         SpringApplication app = new SpringApplication(DomainAndPersistenceConfig.class);
```

```
20.         app.setLogStartupInfo(false);
21.         // on la lance
22.         ConfigurableApplicationContext context = app.run(args);
23.         // métier
24.         IMetier métier = context.getBean(IMetier.class);
25.         try {
26.            // ajouter un RV
27.            Date jour = new Date();
28.            System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau 1 pour le client 1", new
    SimpleDateFormat("dd/MM/yyyy").format(jour)));
29.            Client client = (Client) new Client().build(1L, 1L);
30.            Creneau créneau = (Creneau) new Creneau().build(1L, 1L);
31.            Rv rv = métier.ajouterRv(jour, créneau, client);
32.            System.out.println(String.format("Rv ajouté = %s", rv));
33.            // vérification
34.            créneau = métier.getCreneauById(1L);
35.            long idMedecin = créneau.getIdMedecin();
36.            display("Liste des rendez-vous", métier.getRvMedecinJour(idMedecin, jour));
37.         } catch (Exception ex) {
38.            System.out.println("Exception : " + ex.getCause());
39.         }
40.         // fermeture du contexte Spring
41.         context.close();
42.      }
43.
44.      // méthode utilitaire - affiche les éléments d'une collection
45.      private static <T> void display(String message, Iterable<T> elements) {
46.         System.out.println(message);
47.         for (T element : elements) {
48.            System.out.println(element);
49.         }
50.      }
51.
52. }
```

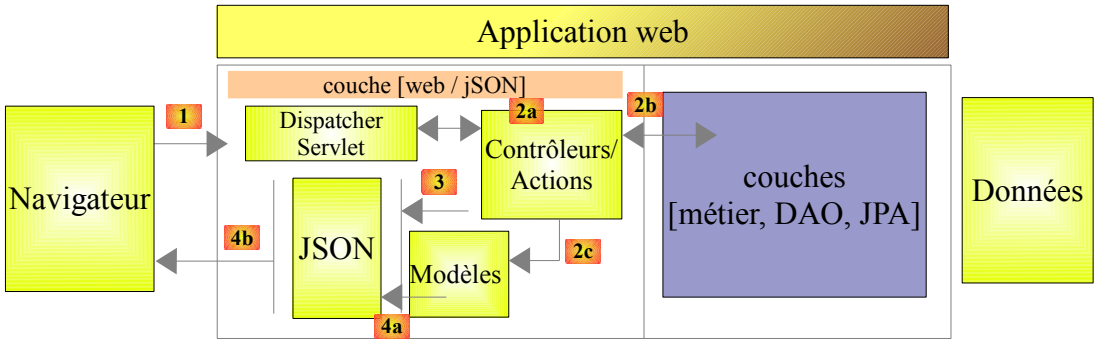
Le programme ajoute un rendez-vous et ensuite vérifie qu'il a été ajouté.

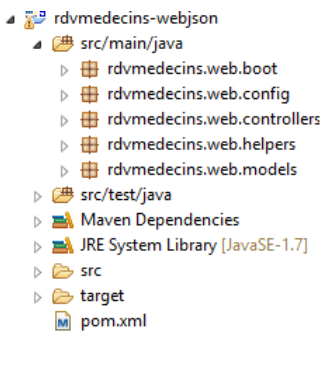
- ligne 19 : la classe [SpringApplication] va exploiter la classe de configuration [DomainAndPersistenceConfig] ;
- ligne 20 : suppression des logs de démarrage de l'application ;
- ligne 22 : la classe [SpringApplication] est exécutée. Elle rend un contexte Spring, ç-à-d la liste des beans enregistrés ;
- ligne 24 : on récupère une référence sur le bean implémentant l'interface [IMetier]. Il s'agit donc d'une référence sur la couche [métier] ;
- lignes 27-31 : ajout d'un nouveau rendez-vous pour aujourd'hui, pour le client n°1 dans le créneau n° 1. Le client et le créneau ont été créés de toute pièce pour montrer que seuls les identifiants sont utilisés. On a initialisé ici la version mais on n'aurait pu mettre n'importe quoi. Elle n'est pas utilisée ici ;
- ligne 34 : on veut connaître le médecin ayant le créneau n° 1. Pour cela on a besoin d'aller en base chercher le créneau n° 1. Parce qu'on est en mode [FetchType.LAZY], le médecin n'est pas ramené avec le créneau. Cependant, on a pris soin de prévoir un champ [idMedecin] dans l'entité [Creneau] pour récupérer la clé primaire du médecin ;
- ligne 35 : on récupère la primaire du médecin ;
- ligne 36 : on affiche la liste des rendez-vous du médecin ;

Les résultats console sont les suivants :

```
1.  Ajout d'un Rv le [10/06/2014] dans le créneau 1 pour le client 1
2.  Rv ajouté = Rv[113, Tue Jun 10 16:51:01 CEST 2014, 1, 1]
3.  Liste des rendez-vous
4.  Rv[113, 2014-06-10, 1, 1]
```

8.4.10 La couche [web / jSON]





Nous allons construire la couche [web / jSON] en plusieurs étapes :

- étape 1 : une couche web opérationnelle sans authentification ;
- étape 2 : mise en place de l'authentification avec Spring Security ;
- étape 3 : mise en place des CORS [**Cross-origin resource sharing** (**CORS**) is a mechanism that allows many resources (e.g. fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain the resource originated from. (Wikipedia)]. Le client de notre service web sera un client web Angular qui n'appartiendra pas nécessairement au même domaine que notre service web. Par défaut, il ne peut alors pas y accéder sauf si le service web l'y autorise. Nous verrons comment ;

8.4.10.1 Configuration Maven

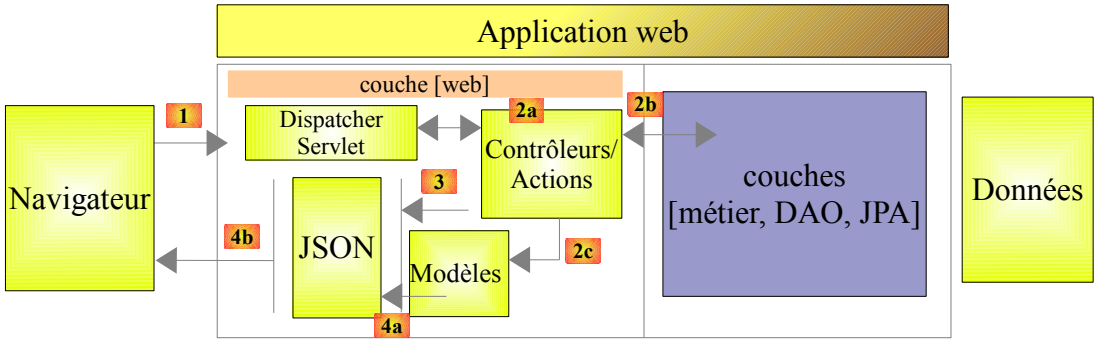
Le fichier [pom.xml] du projet est le suivant :

```
1.  <modelVersion>4.0.0</modelVersion>
2.      <groupId>istia.st.spring4.mvc</groupId>
3.      <artifactId>rdvmedecins-webapi-v1</artifactId>
4.      <version>0.0.1-SNAPSHOT</version>
5.      <name>rdvmedecins-webapi-v1</name>
6.      <description>Gestion de RV Médecins</description>
7.      <parent>
8.          <groupId>org.springframework.boot</groupId>
9.          <artifactId>spring-boot-starter-parent</artifactId>
10.         <version>1.2.0.RELEASE</version>
11.     </parent>
12.     <dependencies>
13.         <dependency>
14.             <groupId>org.springframework.boot</groupId>
15.             <artifactId>spring-boot-starter-web</artifactId>
16.         </dependency>
17.         <dependency>
18.             <groupId>istia.st.spring4.rdvmedecins</groupId>
19.             <artifactId>rdvmedecins-metier-dao</artifactId>
20.             <version>0.0.1-SNAPSHOT</version>
21.         </dependency>
22.     </dependencies>
```

- lignes 7-11 : le projet Maven parent ;
- lignes 13-16 : les dépendances pour un projet Spring MVC ;

- lignes 17-21 : les dépendances sur le projet des couches [métier, DAO, JPA] ;

8.4.10.2 *L'interface du service web*



- en [1], ci-dessus, le navigateur ne peut demander qu'un nombre restreint d'URL avec une syntaxe précise ;
- en [4], il reçoit une réponse jSON ;

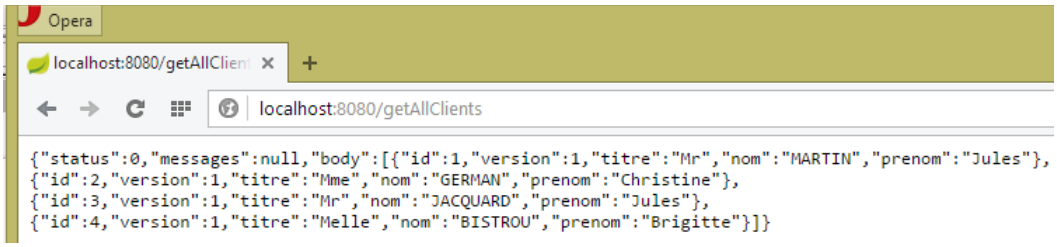
Les réponses de notre service web auront toutes la même forme correspondant à la transformation jSON d'un objet de type [Response] suivant :

```
1.  package rdvmedecins.web.models;
2.
3.  import java.util.List;
4.
5.  public class Response<T> {
6.
7.      // ----------------- propriétés
8.      // statut de l'opération
9.      private int status;
10.     // les éventuels messages d'erreur
11.     private List<String> messages;
12.     // le corps de la réponse
13.     private T body;
14.
15.     // constructeurs
16.     public Response() {
17.
18.     }
19.
20.     public Response(int status, List<String> messages, T body) {
21.         this.status = status;
22.         this.messages = messages;
23.         this.body = body;
24.     }
25.
26.     // getters et setters
27.     ...
28. }
```

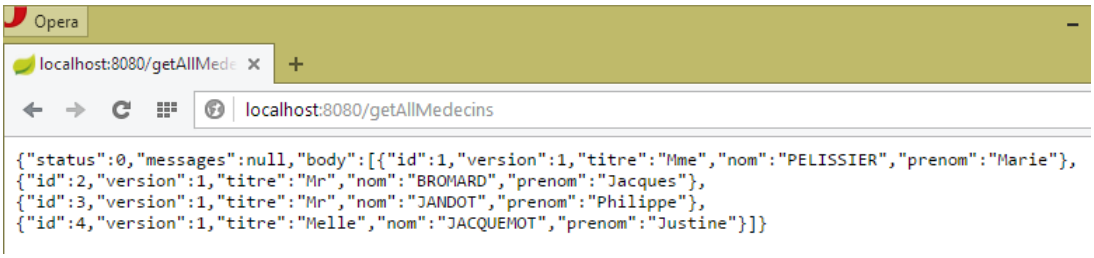
- ligne 7 : code d'erreur de la réponse 0: OK, autre chose : KO ;
- ligne 11 : une liste de messages d'erreur, si erreur il y a ;
- ligne 13 : le corps de la réponse ;

Nous présentons maintenant les copies d'écran qui illustrent l'interface du service web / jSON :

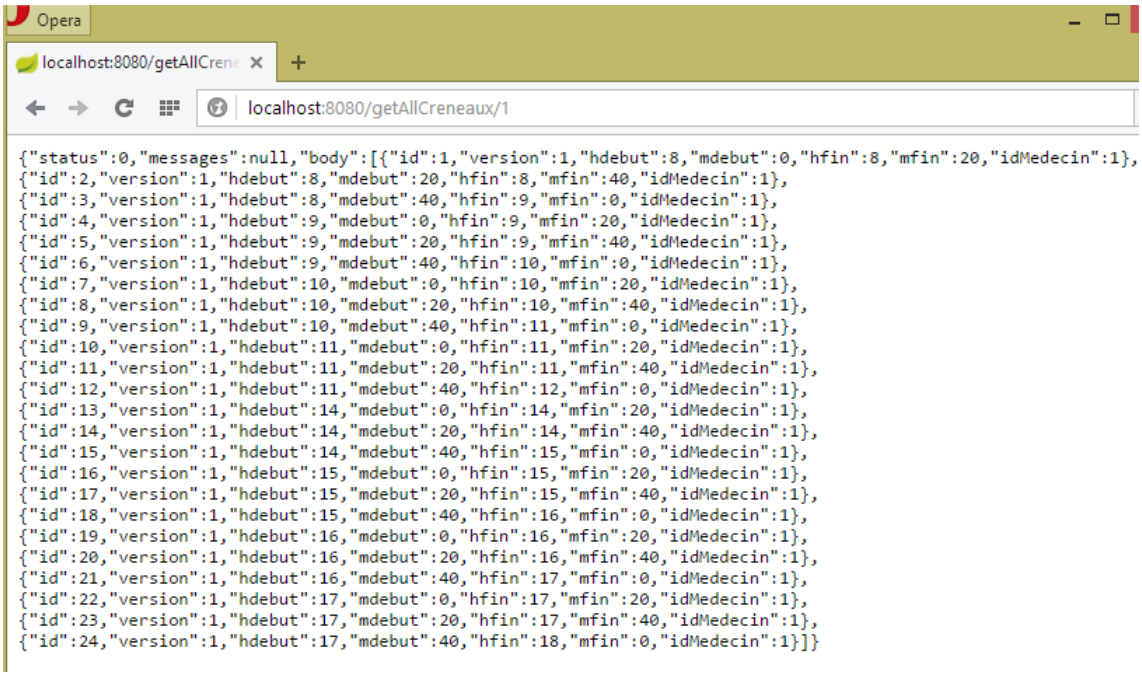
Liste de tous les patients du cabinet médical [/getAllClients]



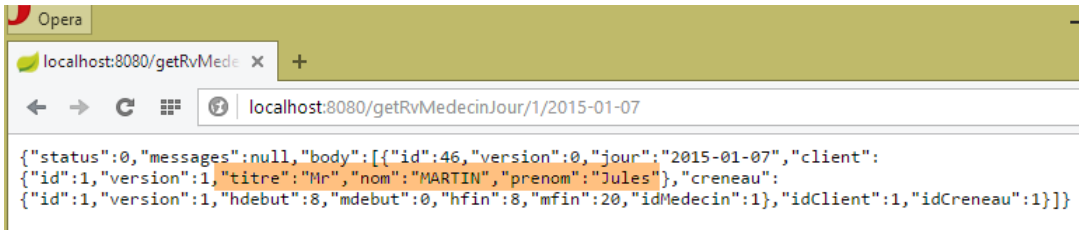
Liste de tous les médecins du cabinet médical [/getAllMedecins]



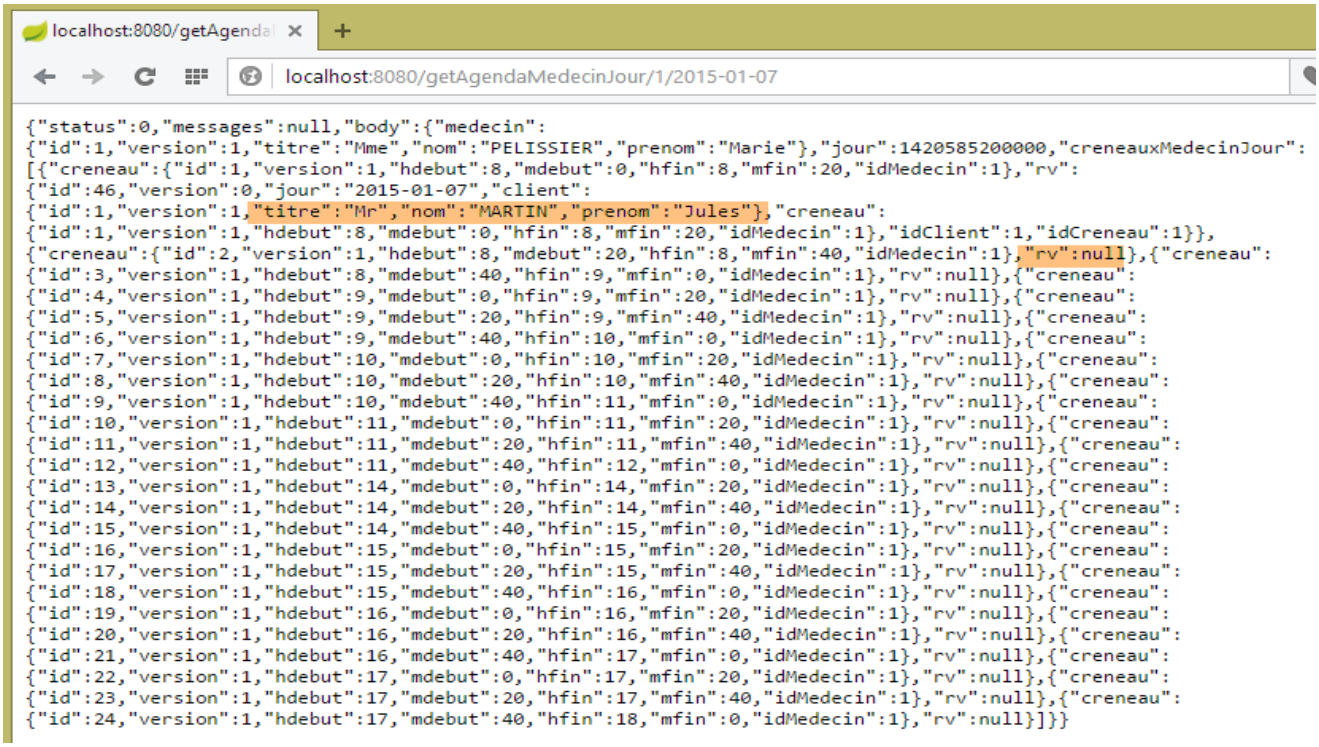
Liste des créneaux horaires d'un médecin [/getAllCreneaux/{idMedecin}]



Liste des rendez-vous d'un médecin [/getRvMedecinJour/{idMedecin}/{aaaa-mm-jj}

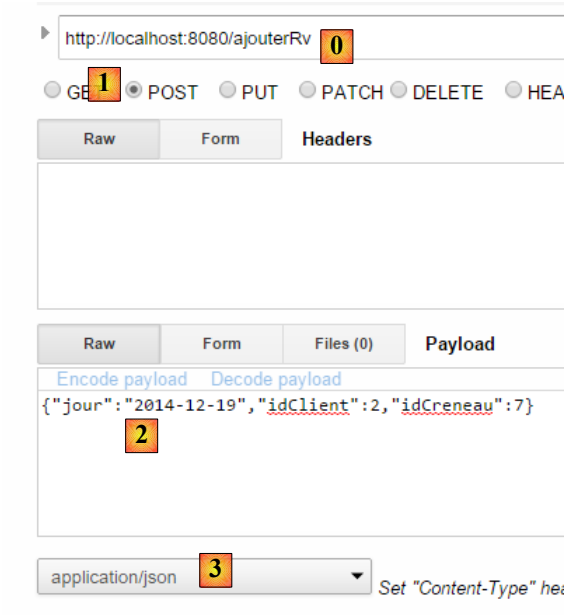


Agenda d'un médecin [/getAgendaMedecinJour/{idMedecin}/{aaaa-mm-jj}]



Pour ajouter / supprimer un rendez-vous nous utilisons le complément Chrome [Advanced Rest Client] car ces opérations se font avec un POST.

Ajouter un rendez-vous [/ajouterRv]

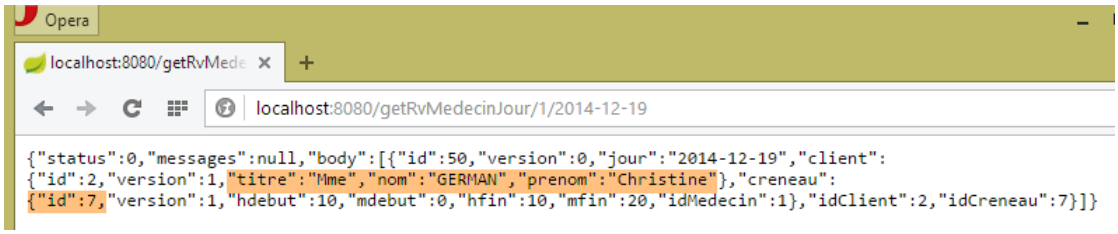


- en [0], l'URL du service web ;
- en [1], la méthode POST est utilisée ;
- en [2], le texte jSON des informations tarnsmises au service web sous la forme {jour, idClient, idCreneau} ;
- en [3], le client précise au service web qu'il lui envoie des informations au format jSON ;

La réponse est alors la suivante :

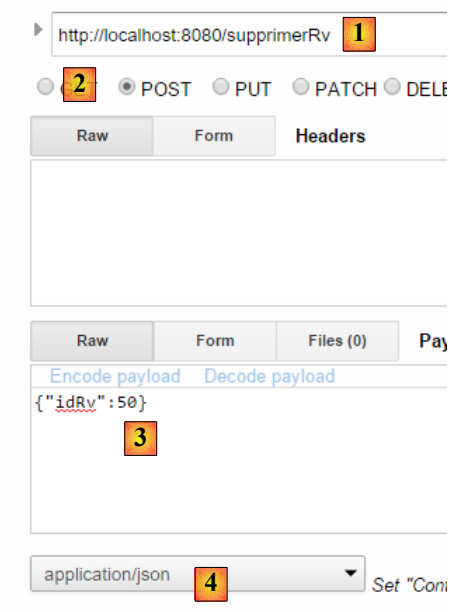
- en [4] : le client envoie l'entête signifiant que les données qu'il envoie sont au format jSON ;
- en [5] : le service web répond qu'il envoie lui aussi du jSON ;
- en [6] : la réponse jSON du service web. Le champ [body] contient la forme jSON du rendez-vous ajouté ;

La présence du nouveau rendez-vous peut être vérifié :



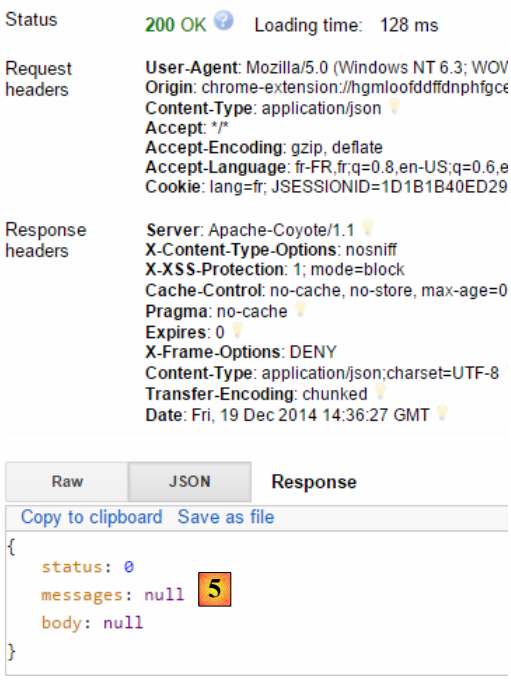
On notera l'id [50] du rendez-vous. Nous allons supprimer celui-ci.

Supprimer un rendez-vous [/supprimerRv]



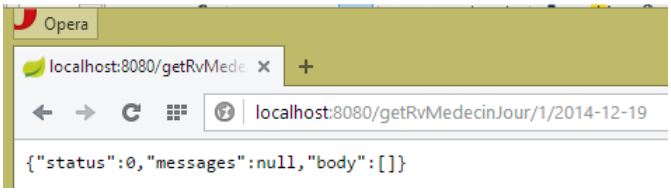
- en [1], l'URL du service web ;
- en [2], la méthode POST est utilisée;
- en [3], le texte jSON des informations transmises au service web sous la forme {idRv} ;
- en [4], le client précise au service web qu'il lui envoie des informations jSON ;

La réponse est alors la suivante :



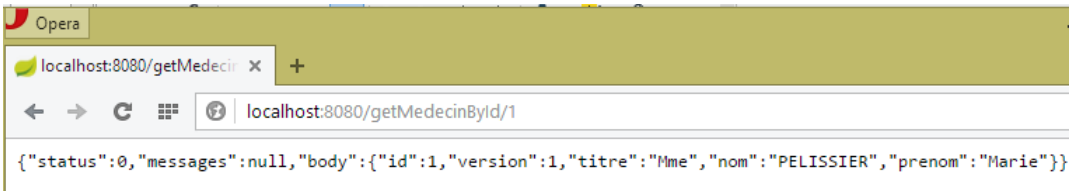
- en [5] : le champ [status] est à 0, montrant par là que l'opération a réussi ;

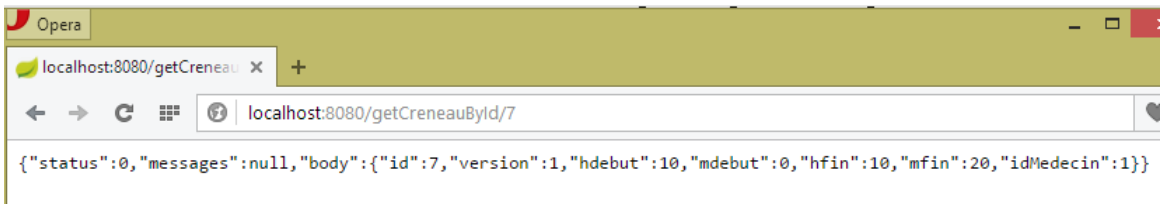
La suppression du rendez-vous peut être vérifiée :

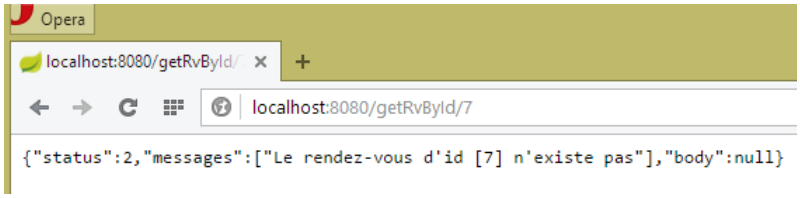


Ci-dessus, le rendez-vous du patient [Mme GERMAN] n'est plus présent.

Le service web permet également de récupérer des entités via leur identifiant :

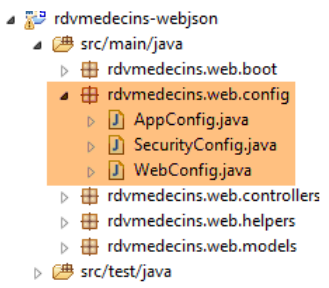






Toutes ces URL sont traitées par le contrôleur [RdvMedecinsController] que nous allons présenter prochainement.

8.4.10.3 Configuration du service web



La classe de configuration [AppConfig] est la suivante :

```
1.  package rdvmedecins.web.config;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.context.annotation.ComponentScan;
5.  import org.springframework.context.annotation.Import;
6.
7.  import rdvmedecins.config.DomainAndPersistenceConfig;
8.
9.  @EnableAutoConfiguration
10. @ComponentScan(basePackages = { "rdvmedecins.web" })
11. @Import({ DomainAndPersistenceConfig.class, SecurityConfig.class, WebConfig.class })
12. public class AppConfig {
13.
14. }
```

- ligne 12 : la classe [AppConfig] configure la totalité de l'application ;
- ligne 9 : on se met en mode [AutoConfiguration] afin que Spring Boot puisse configurer le projet en fonction des archives qu'il trouvera dans le Classpath du projet ;
- ligne 10 : on demande à ce que les composants Spring soient cherchés dans le package [rdvmedecins.web] et ses descendants. C'est ainsi que seront découverts les composants :
  - [**@RestController RdvMedecinsController**] dans le package [rdvmedecins.web.controllers] ;
  - [**@Component ApplicationModel**] dans le package [rdvmedecins.web.models] ;
- ligne 11 : on importe la classe [DomainAndPersistenceConfig] qui configure le projet [rdvmedecins-metier-dao] afin d'avoir accès aux beans de ce projet ;
- ligne 11 : la classe [SecurityConfig] configure la sécurité de l'application web. Nous allons l'ignorer pour l'instant ;
- ligne 11 : la classe [WebConfig] configure la couche [web / jSON] ;

La classe [WebConfig] est la suivante :

```
1.      // mapping jSON
2.      @Bean
3.      public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
4.         final MappingJackson2HttpMessageConverter converter = new MappingJackson2HttpMessageConverter();
5.         final ObjectMapper objectMapper = new ObjectMapper();
6.         converter.setObjectMapper(objectMapper);
7.         return converter;
8.      }
9.
10.
11.     @Override
12.     public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
13.        converters.add(mappingJackson2HttpMessageConverter());
14.        super.configureMessageConverters(converters);
15. }
```

- lignes 12-15 : ajoutent un convertisseur jSON à la liste des convertisseurs de l'application web. La bibliothèque jSON Jackson étant dans le Classpath du projet, c'est elle qui est utilisée par défaut. Si on ne faisait rien, Spring Boot ferait la même chose par défaut. Alors pourquoi le faire ? Nous allons découvrir que nous avons besoin de contrôler la sérialisation des objets rendus par le service web / jSON. Si on ne fait rien, la bibliothèque jSON sérialise la totalité de l'objet. Or nous verrons que dans certains cas, ce n'est pas possible. Pour pouvoir contrôler cette sérialisation, nous avons besoin du bean défini aux lignes 3-8 ;
- lignes 3-8 : définissent le bean [mappingJackson2HttpMessageConverter]. Le type [MappingJackson2HttpMessageConverter] représente le sérialiseur jSON de la bibliothèque Jackson. Ainsi défini, le bean

[mappingJackson2HttpMessageConverter] va pouvoir être injecté dans le contrôleur, ce qui va nous permettre de contrôler la sérialisation des objets rendus par le service web ;

8.4.10.4 *La classe [ApplicationModel]*

La classe [ApplicationModel] va nous servir à deux choses :

- de cache pour stocker les listes de médecins et de patients (clients) ;
- d'interface unique pour les contrôleurs ;

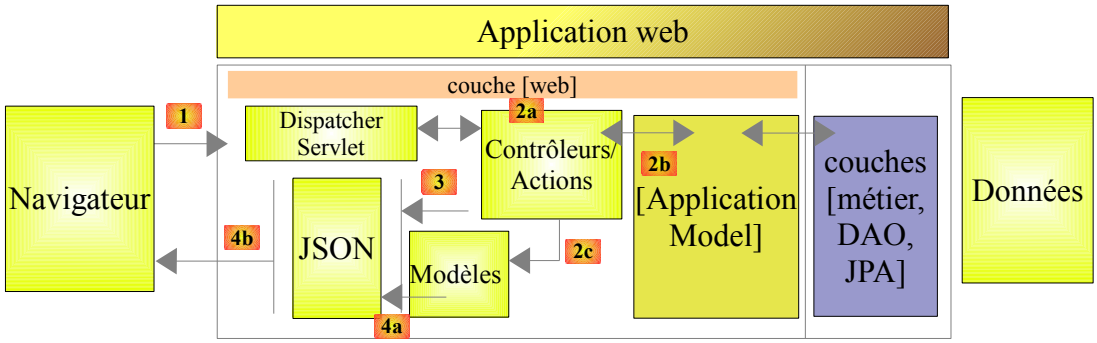
```
1.  package rdvmedecins.web.models;
2.
3.  import java.util.Date;
4.  import java.util.List;
5.
6.  import javax.annotation.PostConstruct;
7.
8.  import org.springframework.beans.factory.annotation.Autowired;
9.  import org.springframework.stereotype.Component;
10.
11. import rdvmedecins.domain.AgendaMedecinJour;
12. import rdvmedecins.entities.Client;
13. import rdvmedecins.entities.Creneau;
14. import rdvmedecins.entities.Medecin;
15. import rdvmedecins.entities.Rv;
16. import rdvmedecins.metier.IMetier;
17. import rdvmedecins.web.helpers.Static;
18.
19. @Component
20. public class ApplicationModel implements IMetier {
21.
22.     // la couche [métier]
23.     @Autowired
24.     private IMetier métier;
25.
26.     // données provenant de la couche [métier]
27.     private List<Medecin> médecins;
28.     private List<Client> clients;
29.     private List<String> messages;
30.     // données de configuration
31.     private boolean CORSneeded = false;
32.     private boolean secured = false;
33.
34.     @PostConstruct
35.     public void init() {
36.         // on récupère les médecins et les clients
37.         try {
38.             médecins = métier.getAllMedecins();
39.             clients = métier.getAllClients();
40.         } catch (Exception ex) {
41.             messages = Static.getErreursForException(ex);
42.         }
43.     }
44.
45.     // getter
46.     public List<String> getMessages() {
47.         return messages;
48.     }
```

```
49.
50.     // ------------------------ interface couche [métier]
51.     @Override
52.     public List<Client> getAllClients() {
53.         return clients;
54.     }
55.
56.     @Override
57.     public List<Medecin> getAllMedecins() {
58.         return médecins;
59.     }
60.
61.     @Override
62.     public List<Creneau> getAllCreneaux(long idMedecin) {
63.         return métier.getAllCreneaux(idMedecin);
64.     }
65.
66.     @Override
67.     public List<Rv> getRvMedecinJour(long idMedecin, Date jour) {
68.         return métier.getRvMedecinJour(idMedecin, jour);
69.     }
70.
71.     @Override
72.     public Client getClientById(long id) {
73.         return métier.getClientById(id);
74.     }
75.
76.     @Override
77.     public Medecin getMedecinById(long id) {
78.         return métier.getMedecinById(id);
79.     }
80.
81.     @Override
82.     public Rv getRvById(long id) {
83.         return métier.getRvById(id);
84.     }
85.
86.     @Override
87.     public Creneau getCreneauById(long id) {
88.         return métier.getCreneauById(id);
89.     }
90.
91.     @Override
92.     public Rv ajouterRv(Date jour, Creneau creneau, Client client) {
93.         return métier.ajouterRv(jour, creneau, client);
94.     }
95.
96.     @Override
97.     public void supprimerRv(long idRv) {
98.         métier.supprimerRv(idRv);
99.     }
100.
101.    @Override
102.    public AgendaMedecinJour getAgendaMedecinJour(long idMedecin, Date jour) {
103.        return métier.getAgendaMedecinJour(idMedecin, jour);
104.    }
105.
106.    // getters et setters
107.public boolean isCORSneeded() {
108.        return CORSneeded;
109.    }
110.
111.    public boolean isSecured() {
112.        return secured;
113.    }
114.
115.}
```

- ligne 19 : l'annotation [@**Component**] fait de la classe [ApplicationModel] un composant Spring. Comme tous les composants Spring vus jusqu'ici (à l'exception de @**Controller**), un seul objet de ce type sera instancié (**singleton**) ;
- ligne 20 : la classe [ApplicationModel] implémente l'interface [IMetier] ;
- lignes 23-24 : une référence sur la couche [métier] est injectée par Spring ;

- ligne 34 : l'annotation [**@PostConstruct**] fait que la méthode [init] va être exécutée juste après l'instanciation de la classe [ApplicationModel] ;
- lignes 38-39 : on récupère les listes de médecins et de clients auprès de la couche [métier] ;
- ligne 41 : si une exception se produit, on stocke les messages de la pile d'exceptions dans le champ de la ligne 17 ;

L'architecture de la couche web évolue comme suit :

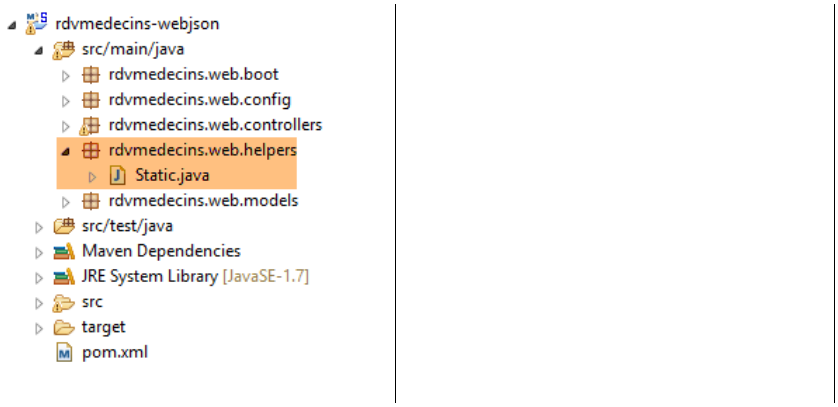


- en [2b], les méthodes du ou des contrôleurs communiquent avec le singleton [ApplicationModel] ;

Cette stratégie amène de la souplesse quant à la gestion du cache. Actuellement les créneaux horaires des médecins ne sont pas mis en cache. Pour les y mettre, il suffit de modifier la classe [ApplicationModel]. Cela n'a aucun impact sur le contrôleur qui continuera à utiliser la méthode [List<Creneau> getAllCreneaux(**long** idMedecin)] comme il le faisait auparavant. C'est l'implémentation de cette méthode dans [ApplicationModel] qui sera changée.

8.4.10.5 La classe Static

La classe [Static] regroupe un ensemble de méthodes statiques utilitaires qui n'ont pas d'aspect " métier " ou " web " :



Son code est le suivant :

```
1.  package rdvmedecins.web.helpers;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class Static {
7.
8.     public Static() {
9.     }
10.
11.    // liste des messages d'erreur d'une exception
12.    public static List<String> getErreursForException(Exception exception) {
13.       // on récupère la liste des messages d'erreur de l'exception
14.       Throwable cause = exception;
15.       List<String> erreurs = new ArrayList<String>();
16.       while (cause != null) {
17.          erreurs.add(cause.getMessage());
18.          cause = cause.getCause();
```

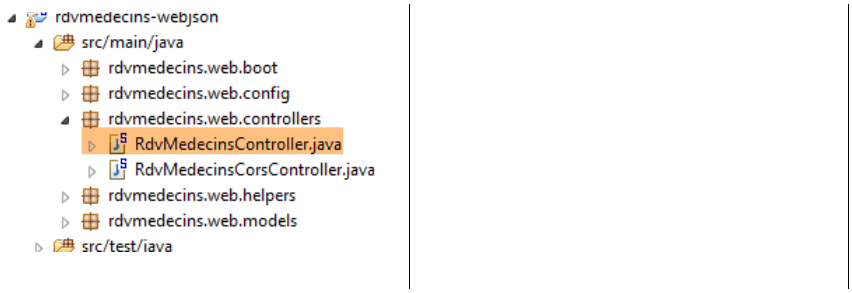
```
19.         }
20.         return erreurs;
21.     }
22. }
```

- ligne 12 : la méthode [Static.getErreursForException] qui a été utilisée (ligne 8 ci-dessous) dans la méthode [init] de la classe [ApplicationModel] :

```
1.     @PostConstruct
2.     public void init() {
3.         // on récupère les médecins et les clients
4.         try {
5.           médecins = métier.getAllMedecins();
6.           clients = métier.getAllClients();
7.         } catch (Exception ex) {
8.           messages = Static.getErreursForException(ex);
9.         }
10.}
```

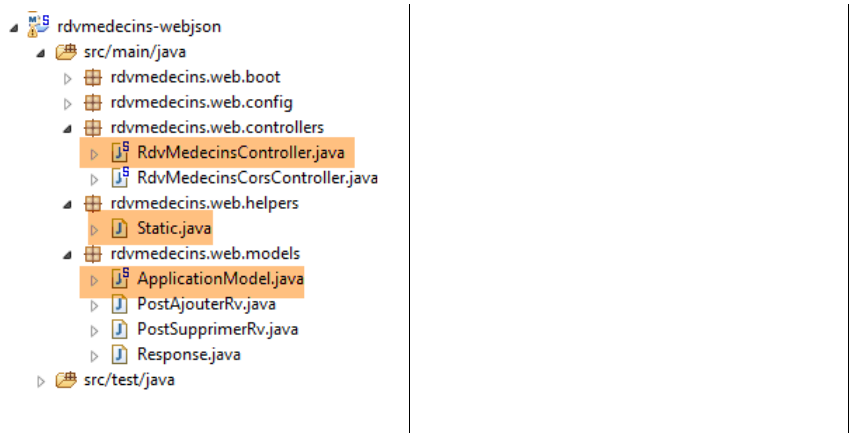
La méthode construit un objet [List<String>] avec les messages d'erreur [exception.getMessage()] d'une exception [exception] et de celles qu'elle contient [exception.getCause()].

8.4.10.6 Le squelette du contrôleur [RdvMedecinsController]



Nous allons maintenant détailler le traitement des URL du service web. Trois classes principales sont en jeu dans ce traitement :

- le contrôleur [**RdvMedecinsController**] ;
- la classe de méthodes utilitaires [**Static**] ;
- la classe de cache [**ApplicationModel**] ;



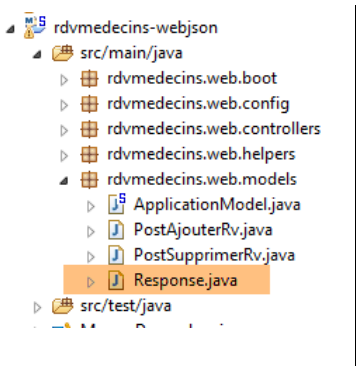
Le contrôleur [RdvMedecinsController] est le suivant :

```
1.  package rdvmedecins.web.controllers;
2.
3.  import java.text.ParseException;
4.  ...
5.
6.  @RestController
7.  public class RdvMedecinsController {
```

```
8.
9.      @Autowired
10.     private ApplicationModel application;
11.
12.     @Autowired
13.     private RdvMedecinsCorsController rdvMedecinsCorsController;
14.
15.     @Autowired
16.     private MappingJackson2HttpMessageConverter converter;
17.
18.     // liste de messages
19.     private List<String> messages;
20.     // mapper JSON
21.     private ObjectMapper mapper;
22.
23.     @PostConstruct
24.     public void init() {
25.        // messages d'erreur de l'application
26.        messages = application.getMessages();
27.        // mapper JSON
28.        mapper = converter.getObjectMapper();
29.     }
30.
31.     // début méthodes locales
32.     // ----------------------------------------------------------------
33.     // liste des médecins
34.     @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
35.     public Reponse getAllMedecins() {
36. ...
37.     }
38.
39.     // liste des clients
40.     @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
41.     public Reponse getAllClients() {
42. ...
43.     }
44.
45.     // liste des créneaux d'un médecin
46.     @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method = RequestMethod.GET)
47.     public Reponse getAllCreneaux(@PathVariable("idMedecin") long idMedecin) {
48. ...
49.     }
50.
51.     // liste des rendez-vous d'un médecin
52.     @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method = RequestMethod.GET)
53.     public Reponse getRvMedecinJour(@PathVariable("idMedecin") long idMedecin,
54.           @PathVariable("jour") String jour) {
55. ...
56.     }
57.
58.     @RequestMapping(value = "/getClientById/{id}", method = RequestMethod.GET)
59.     public Reponse getClientById(@PathVariable("id") long id) {
60. ...
61.     }
62.
63.     @RequestMapping(value = "/getMedecinById/{id}", method = RequestMethod.GET)
64.     public Reponse getMedecinById(@PathVariable("id") long id) {
65. ...
66.     }
67.
68.     @RequestMapping(value = "/getRvById/{id}", method = RequestMethod.GET)
69.     public Reponse getRvById(@PathVariable("id") long id) {
70. ...
71.     }
72.
73.     @RequestMapping(value = "/getCreneauById/{id}", method = RequestMethod.GET)
74.     public Reponse getCreneauById(@PathVariable("id") long id) {
75. ...
76.     }
77.
78.     @RequestMapping(value = "/ajouterRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
79.     public Reponse ajouterRv(@RequestBody PostAjouterRv post) {
80. ...
81.     }
```

```
82.
83.     @RequestMapping(value = "/supprimerRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
84.     public Reponse supprimerRv(@RequestBody PostSupprimerRv post) {
85. ...
86.     }
87.
88.     @RequestMapping(value = "/getAgendaMedecinJour/{idMedecin}/{jour}", method = RequestMethod.GET)
89.     public Reponse getAgendaMedecinJour(
90.             @PathVariable("idMedecin") long idMedecin,
91.             @PathVariable("jour") String jour) {
92. ...
93.     }
94. }
```

- ligne 6 : l'annotation [**@RestController**] fait de la classe [RdvMedecinsController] un contrôleur Spring. Par ailleurs, elle entraîne également que les méthodes traitant les URL vont générer une réponse qui sera automatiquement transformée en jSON ;
- lignes 9-10 : un objet de type [ApplicationModel] sera injecté ici par Spring. Nous l'avons présenté ;
- lignes 12-13 : un objet de type [RdvMedecinsCorsController] sera injecté ici par Spring. Nous ne présenterons cet objet qu'ultérieurement ;
- lignes 15-16 : un objet de type [MappingJackson2HttpMessageConverter] sera injecté ici par Spring. Nous l'avons présenté. Cet objet a été défini dans la classe [WebConfig] et va nous servir à contrôler la sérialisation des objets rendus par le service [web / jSON] ;
- ligne 24 : l'annotation [**@PostConstruct**] tague une méthode à exécuter juste après l'instanciation de la classe. Lorsqu'elle celle-ci s'exécute, les objets injectés par Spring sont disponibles ;
- ligne 26 : on récupère les éventuels messages d'erreur de l'objet [ApplicationModel]. Cet objet a été instancié au démarrage de l'application et a essayé de mettre en cache, les médecins et les clients. S'il a échoué, alors on a [messages!=null]. Cela va permettre aux méthodes du contrôleur de savoir si l'application s'est initialisée correctement.
- ligne 28 : on crée un objet Jackson jSON de type [ObjectMapper]. C'est cet objet précis qui permet de contrôler la sérialisation jSON ;
- lignes 34-93 : les URL exposées par le service [web / jSON]. Toutes les méthodes rendent un objet de type [Response] suivant :




```
1. package rdvmedecins.web.models;
2.
3. import java.util.List;
4.
5. public class Response<T> {
6.
7.     // ----------------- propriétés
8.     // statut de l'opération
9.     private int status;
10.    // les éventuels messages d'erreur
11.    private List<String> messages;
12.    // le corps de la réponse
13.    private T body;
14.
15.    // constructeurs
16.    public Response() {
17.
18.    }
19.
20.    public Response(int status, List<String> messages, T body) {
```

```
21.      this.status = status;
22.      this.messages = messages;
23.      this.body = body;
24.    }
25.
26.    // getters et setters
27.    ...
28.}
```

- ligne 9 : un code d'erreur : 0 signifie pas d'erreur ;
- ligne 11 : si [status!=0], alors [messages] est une liste de messages d'erreur ;
- ligne 13 : un objet T encapsulé dans la réponse. T vaut *null* en cas d'erreur ;

Cet objet est sérialisé en jSON avant d'être envoyé au navigateur client ;

- ligne 34 : l'annotation [**@RequestMapping**] fixe les conditions d'appel de la méthode. Ici la méthode traite une demande GET de l'URL [/getAllMedecins]. Si cette URL était demandée par un POST, elle serait refusée et Spring MVC enverrait un code HTTP d'erreur au client web ;
- ligne 46 : l'URL est paramétrée par {idMedecin}. Ce paramètre est récupéré avec l'annotation [**@PathVariable**] ligne 47 ;
- ligne 47 : l'unique paramètre [**long** idMedecin] reçoit sa valeur du paramètre {idMedecin} de l'URL [@PathVariable("idMedecin")]. Le paramètre dans l'URL et celui de la méthode peuvent porter des noms différents. Il faut noter ici que [@PathVariable("idMedecin")] est de type String (toute l'URL est un String) alors que le paramètre [**long** idMedecin] est de type [long]. Le changement de type est fait automatiquement. Un code d'erreur HTTP est renvoyé si ce changement de type échoue ;
- ligne 79 : l'annotation [**@RequestBody**] désigne le corps de la requête. Dans une requête GET, il n'y a quasiment jamais de corps (mais il est possible d'en mettre un). Dans une requête POST, il y en a le plus souvent (mais il est possible de ne pas en mettre). Pour l'URL [ajouterRv], le client web envoie dans son POST la chaîne jSON suivante :

```
{"jour":"2014-06-12", "idClient":3, "idCreneau":7}
```

La syntaxe [@RequestBody PostAjouterRv post] (ligne 79) ajoutée au fait que la méthode attend du jSON [consumes = "application/json; charset=UTF-8"] ligne 78 va faire que la chaîne jSON envoyée par le client web va être désérialisée en un objet de type [PostAjouter]. Celui-ci est le suivant :

```
1. package rdvmedecins.web.models;
2.
3. public class PostAjouterRv {
4.
5.     // données du post
6.     private String jour;
7.     private long idClient;
8.     private long idCreneau;
9.
10.    // getters et setters
11.    ...
12.}
```

Là également, les changements de type nécessaires auront lieu automatiquement ;

- lignes 83-84, on trouve un mécanisme similaire pour l'URL [/supprimerRv]. La chaîne jSON postée est la suivante :

```
{"idRv":116}
```

et le type [PostSupprimerRv] le suivant :

```
1. package rdvmedecins.web.models;
2.
3. public class PostSupprimerRv {
4.
5.     // données du post
6.     private long idRv;
7.
8.     // getters et setters
9.     ...
10.}
```

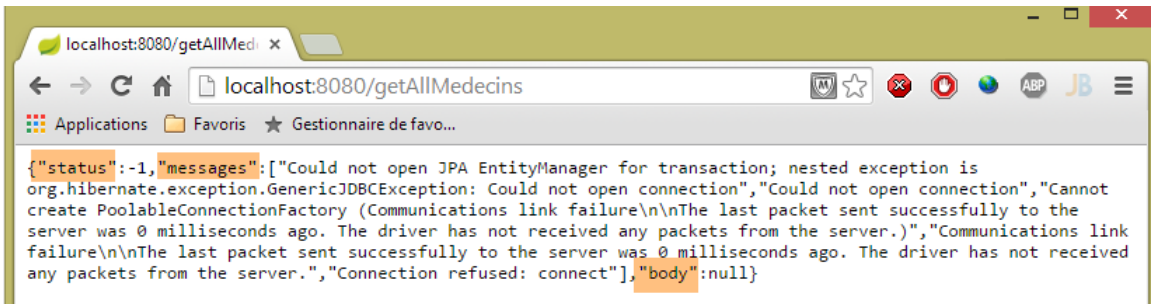
8.4.10.7 L'URL [/getAllMedecins]

L'URL [/getAllMedecins] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

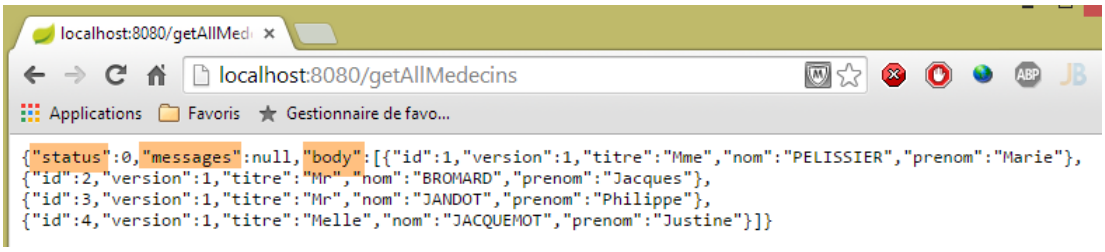
```
1.     @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
2.     public Response<List<Medecin>> getAllMedecins(HttpServletResponse response) {
3.        // état de l'application
4.        if (messages != null) {
5.           return new Response<List<Medecin>>(-1, messages, null);
6.        }
7.        // liste des médecins
8.        try {
9.           return new Response<List<Medecin>>(0, null, application.getAllMedecins());
10.       } catch (Exception e) {
11.          return new Response<List<Medecin>>(1, Static.getErreursForException(e), null);
12.       }
13. }
```

- ligne 5 : on regarde si l'application s'est correctement initialisée (*messages==null*). Si ce n'est pas le cas, on renvoie une réponse avec *status=-1* et *body=messages* ;
- ligne 10 : sinon on renvoie la liste des médecins avec un *status* égal à 0. La méthode [application.getAllMedecins()] ne lance pas d'exception car elle se contente de rendre une liste qui est en cache. Néanmoins on gardera cette gestion d'exception pour le cas où les médecins ne seraient plus mis en cache ;

Nous n'avons pas encore illustré le cas où l'application s'est mal initialisée. Arrêtons le SGBD MySQL5, lançons le service web puis demandons l'URL [/getAllMedecins] :



On obtient bien une erreur. Dans un contexte normal, on obtient la vue suivante :



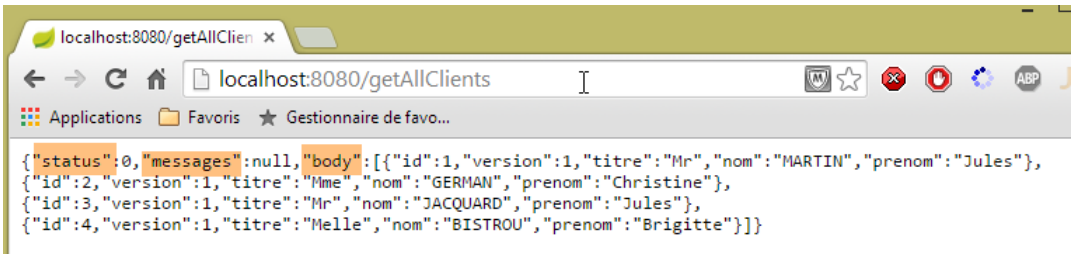
8.4.10.8 L'URL [/getAllClients]

L'URL [/getAllClients] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.  @RequestMapping(value = "/getAllClients", method = RequestMethod.GET)
2.     public Response<List<Client>> getAllClients(HttpServletResponse response) {
3.        // état de l'application
4.        if (messages != null) {
5.           return new Response<List<Client>>(-1, messages, null);
6.        }
7.        // liste des clients
8.        try {
9.           return new Response<List<Client>>(0, null, application.getAllClients());
```

```
10.         } catch (Exception e) {
11.             return new Response<List<Client>>(1, Static.getErreursForException(e), null);
12.         }
13.     }
```

Elle est analogue à la méthode [getAllMedecins] déjà étudiée. Les résultats obtenus sont les suivants :



8.4.10.9 L'URL [/getAllCreneaux/{idMedecin}]

L'URL [/getAllCreneaux/{idMedecin}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.  @RequestMapping(value = "/getAllCreneaux/{idMedecin}", method = RequestMethod.GET)
2.      public Response<List<Creneau>> getAllCreneaux(@PathVariable("idMedecin") long idMedecin) {
3.          // état de l'application
4.          if (messages != null) {
5.              return new Response<List<Creneau>>(-1, messages, null);
6.          }
7.          // on récupère le médecin
8.          Response<Medecin> responseMedecin = getMedecin(idMedecin);
9.          if (responseMedecin.getStatus() != 0) {
10.             return new Response<List<Creneau>>(responseMedecin.getStatus(), responseMedecin.getMessages(),
    null);
11.         }
12.         Medecin médecin = responseMedecin.getBody();
13.         // créneaux du médecin
14.         List<Creneau> créneaux = null;
15.         try {
16.             créneaux = application.getAllCreneaux(médecin.getId());
17.         } catch (Exception e1) {
18.             return new Response<List<Creneau>>(3, Static.getErreursForException(e1), null);
19.         }
20.         // on rend la réponse
21.         SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
22.         mapper.setFilters(new SimpleFilterProvider().addFilter("creneauFilter", creneauFilter));
23.         return new Response<List<Creneau>>(0, null, créneaux);
24.     }
```

- ligne 8 : le médecin identifié par le paramètre [id] est demandé à une méthode locale :

```
1.  private Response<Medecin> getMedecin(long id) {
2.          // on récupère le médecin
3.          Medecin médecin = null;
4.          try {
5.              médecin = application.getMedecinById(id);
6.          } catch (Exception e1) {
7.              return new Response<Medecin>(1, Static.getErreursForException(e1), null);
8.          }
9.          // médecin existant ?
10.         if (médecin == null) {
11.             List<String> messages = new ArrayList<String>();
12.             messages.add(String.format("Le médecin d'id [%s] n'existe pas", id));
13.             return new Response<Medecin>(2, messages, null);
14.         }
15.         // ok
16.         return new Response<Medecin>(0, null, médecin);
17.     }
```

On revient de cette méthode avec un *status* dans [0,1,2]. Revenons au code de la méthode [getAllCreneaux] :

- lignes 9-11 : si *status!=0*, on rend immédiatement la réponse ;
- ligne 12 : on récupère le médecin ;
- ligne 16 : on récupère les créneaux de ce médecin ;
- lignes 21-23 : on envoie comme réponse un objet [List<Creneau>] ;

Rappelons la définition de la classe [Creneau] :

```
1.  @Entity
2.  @Table(name = "creneaux")
3.  public class Creneau extends AbstractEntity {
4.
5.      private static final long serialVersionUID = 1L;
6.      // caractéristiques d'un créneau de RV
7.      private int hdebut;
8.      private int mdebut;
9.      private int hfin;
10.     private int mfin;
11.
12.     // un créneau est lié à un médecin
13.     @ManyToOne(fetch = FetchType.LAZY)
14.     @JoinColumn(name = "id_medecin")
15.     private Medecin medecin;
16.
17.     // clé étrangère
18.     @Column(name = "id_medecin", insertable = false, updatable = false)
19.     private long idMedecin;
20. ...
21. }
```

- ligne 13 : le médecin est cherché en mode [FetchType.LAZY] ;

Rappelons la requête JPQL qui implémente la méthode [getAllCreneaux] dans la couche [DAO] :

```
@Query("select c from Creneau c where c.medecin.id=?1")
```

La notation [c.medecin.id] force la jointure entre les tables [CRENEAUX] et [MEDECINS]. Aussi la requête ramène-t-elle tous les créneaux du médecin avec dans chacun d'eux le médecin. Lorsqu'on sérialize en jSON ces créneaux, on voit apparaître la chaîne jSON du médecin dans chacun d'eux. C'est inutile. Pour contrôler la sérialisation, il nous faut deux choses :

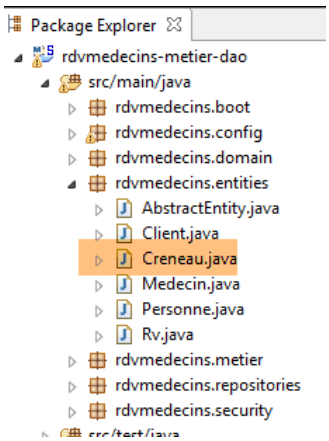
1. avoir accès à l'objet qui sérialise ;
2. configurer l'objet à sérialiser ;

Le point 1 est vérifié avec l'injection du convertisseur jSON Jackson dans le contrôleur :

```
1.  @Autowired
2.      private MappingJackson2HttpMessageConverter converter;
3.
4.      // mapper JSON
5.      private ObjectMapper mapper;
6.
7.      @PostConstruct
8.      public void init() {
9.  ...
10.         // mapper JSON
11.         mapper = converter.getObjectMapper();
12.     }
```

- ligne 5 : [ObjectMapper mapper] est l'objet qui nous permet de contrôler la sérialisation jSON des objets rendus par les méthodes du contrôleur ;

Le point 2 est obtenu en ajoutant une annotation à la classe [Creneau] définie dans le projet [rdvmedecins-metier-dao] :




```
1.  @Entity
2.  @Table(name = "creneaux")
3.  @JsonFilter("creneauFilter")
4.  public class Creneau extends AbstractEntity {
5.  ...
```

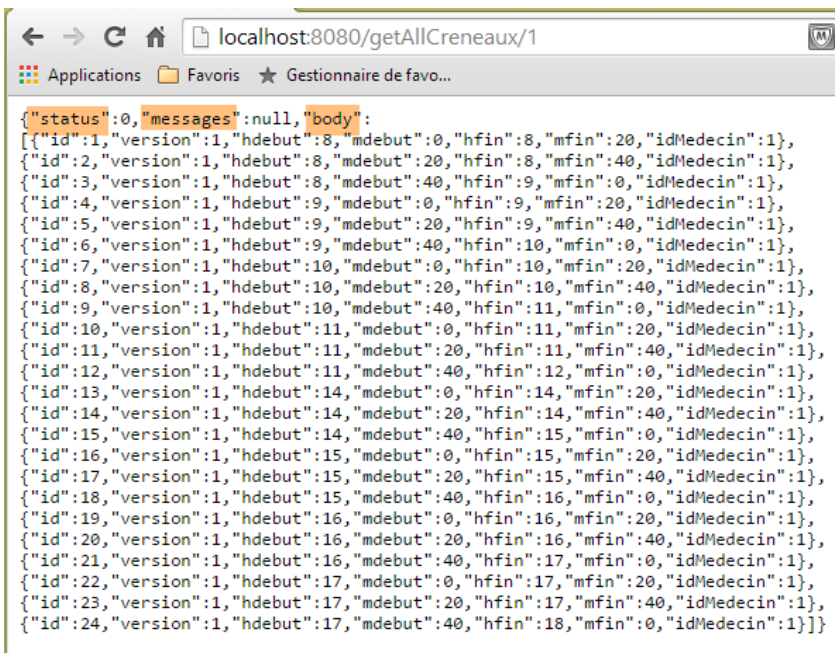
- ligne 3 : une annotation de la bibliothèque jSON Jackson. Elle crée un filtre appelé [creneauFilter]. A l'aide de ce filtre, nous allons pouvoir définir par programmation les champs qui doivent être ou non sérialisés ;

La sérialisation de l'objet [Creneau] se fait dans les lignes suivantes de la méthode [getAllCreneaux] :

```
1.          // on rend la réponse
2.          SimpleBeanPropertyFilter creneauFilter =
    SimpleBeanPropertyFilter.serializeAllExcept("medecin");
3.          mapper.setFilters(new SimpleFilterProvider().addFilter("creneauFilter", creneauFilter));
4.  return new Response<List<Creneau>>(0, null, créneaux);
```

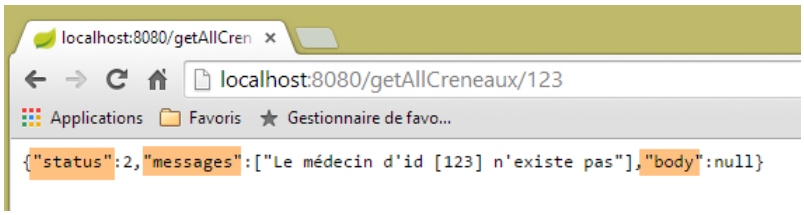
- ligne 2 : définit le filtre [creneauFilter]. Il sérialise tous les champs de l'objet [Creneau] sauf le champ [medecin] ;
- ligne 3 : ce filtre est ajouté aux filtres jSON de l'objet [mapper]. Si ce filtre existait déjà, son ancienne définition est écrasée ;

Les résultats obtenus sont les suivants :




```
{"status":0,"messages":null,"body":
[{"id":1,"version":1,"hdebut":8,"mdebut":0,"hfin":8,"mfin":20,"idMedecin":1},
{"id":2,"version":1,"hdebut":8,"mdebut":20,"hfin":8,"mfin":40,"idMedecin":1},
{"id":3,"version":1,"hdebut":8,"mdebut":40,"hfin":9,"mfin":0,"idMedecin":1},
{"id":4,"version":1,"hdebut":9,"mdebut":0,"hfin":9,"mfin":20,"idMedecin":1},
{"id":5,"version":1,"hdebut":9,"mdebut":20,"hfin":9,"mfin":40,"idMedecin":1},
{"id":6,"version":1,"hdebut":9,"mdebut":40,"hfin":10,"mfin":0,"idMedecin":1},
{"id":7,"version":1,"hdebut":10,"mdebut":0,"hfin":10,"mfin":20,"idMedecin":1},
{"id":8,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"idMedecin":1},
{"id":9,"version":1,"hdebut":10,"mdebut":40,"hfin":11,"mfin":0,"idMedecin":1},
{"id":10,"version":1,"hdebut":11,"mdebut":0,"hfin":11,"mfin":20,"idMedecin":1},
{"id":11,"version":1,"hdebut":11,"mdebut":20,"hfin":11,"mfin":40,"idMedecin":1},
{"id":12,"version":1,"hdebut":11,"mdebut":40,"hfin":12,"mfin":0,"idMedecin":1},
{"id":13,"version":1,"hdebut":14,"mdebut":0,"hfin":14,"mfin":20,"idMedecin":1},
{"id":14,"version":1,"hdebut":14,"mdebut":20,"hfin":14,"mfin":40,"idMedecin":1},
{"id":15,"version":1,"hdebut":14,"mdebut":40,"hfin":15,"mfin":0,"idMedecin":1},
{"id":16,"version":1,"hdebut":15,"mdebut":0,"hfin":15,"mfin":20,"idMedecin":1},
{"id":17,"version":1,"hdebut":15,"mdebut":20,"hfin":15,"mfin":40,"idMedecin":1},
{"id":18,"version":1,"hdebut":15,"mdebut":40,"hfin":16,"mfin":0,"idMedecin":1},
{"id":19,"version":1,"hdebut":16,"mdebut":0,"hfin":16,"mfin":20,"idMedecin":1},
{"id":20,"version":1,"hdebut":16,"mdebut":20,"hfin":16,"mfin":40,"idMedecin":1},
{"id":21,"version":1,"hdebut":16,"mdebut":40,"hfin":17,"mfin":0,"idMedecin":1},
{"id":22,"version":1,"hdebut":17,"mdebut":0,"hfin":17,"mfin":20,"idMedecin":1},
{"id":23,"version":1,"hdebut":17,"mdebut":20,"hfin":17,"mfin":40,"idMedecin":1},
{"id":24,"version":1,"hdebut":17,"mdebut":40,"hfin":18,"mfin":0,"idMedecin":1}]}
```

ou bien ceux-ci si le créneau n'existe pas :



8.4.10.10 L'URL [/getRvMedecinJour/{idMedecin}/{jour}]

L'URL [/getRvMedecinJour/{idMedecin}/{jour}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.  @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method = RequestMethod.GET)
2.     public Response<List<Rv>> getRvMedecinJour(@PathVariable("idMedecin") long idMedecin,
3.           @PathVariable("jour") String jour) {
4.        // état de l'application
5.        if (messages != null) {
6.           return new Response<List<Rv>>(-1, messages, null);
7.        }
8.        // on vérifie la date
9.        Date jourAgenda = null;
10.       SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
11.       sdf.setLenient(false);
12.       try {
13.          jourAgenda = sdf.parse(jour);
14.       } catch (ParseException e) {
15.          List<String> messages = new ArrayList<String>();
16.          messages.add(String.format("La date [%s] est invalide", jour));
17.          return new Response<List<Rv>>(3, messages, null);
18.       }
19.       // on récupère le médecin
20.       Response<Medecin> responseMedecin = getMedecin(idMedecin);
21.       if (responseMedecin.getStatus() != 0) {
22.          return new Response<List<Rv>>(responseMedecin.getStatus(), responseMedecin.getMessages(),
    null);
23.       }
24.       Medecin médecin = responseMedecin.getBody();
25.       // liste de ses rendez-vous
26.       List<Rv> rvs = null;
27.       try {
28.          rvs = application.getRvMedecinJour(médecin.getId(), jourAgenda);
29.       } catch (Exception e1) {
30.          return new Response<List<Rv>>(4, Static.getErreursForException(e1), null);
31.       }
32.       // on rend la réponse
33.       SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept();
34.       SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
35.       mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter",
    rvFilter).addFilter("creneauFilter",creneauFilter));
36.       return new Response<List<Rv>>(0, null, rvs);
37.    }
```

- lignes 33-36 : on contrôle la séralisation des objets [Creneau] (ligne 34) et de l'objet [Rv] (ligne 33) ;

Rappelons la définition de la classe [Rv] dans le projet [rdvmedecins-metier-dao] :

```
1.  @Entity
2.  @Table(name = "rv")
3.  public class Rv extends AbstractEntity {
4.     private static final long serialVersionUID = 1L;
5.
6.     // caractéristiques d'un Rv
7.     @Temporal(TemporalType.DATE)
8.     private Date jour;
9.
10.    // un rv est lié à un client
```

```
11.     @ManyToOne(fetch = FetchType.LAZY)
12.     @JoinColumn(name = "id_client")
13.     private Client client;
14.
15.     // un rv est lié à un créneau
16.     @ManyToOne(fetch = FetchType.LAZY)
17.     @JoinColumn(name = "id_creneau")
18.     private Creneau creneau;
19.
20.     // clés étrangères
21.     @Column(name = "id_client", insertable = false, updatable = false)
22.     private long idClient;
23.     @Column(name = "id_creneau", insertable = false, updatable = false)
24.     private long idCreneau;
25.
26. ...
27.
28. }
```

- ligne 11 : le client est recherché avec le mode [FetchType.LAZY] ;
- ligne 18 : le créneau est recherché avec le mode [FetchType.LAZY] ;

Rappelons la requête JPQL qui va chercher les rendez-vous :

```
@Query("select rv from Rv rv left join fetch rv.client c left join fetch rv.creneau cr where cr.medecin.id=?1 and rv.jour=?2")
```

De jointures sont faites explicitement pour ramener les champs [client] et [creneau]. Par ailleurs à cause de la jointure [cr.medecin.id=?1], nous aurons également le médecin. Le médecin va donc apparaître dans la chaîne jSON de chaque rendez-vous. Or cette information dupliquée est en outre inutile. Nous avons vu comment résoudre ce problème à l'aide d'un filtre jSON sur l'objet [Creneau]. A cause des modes [FetchType.LAZY] des champs [client] et [creneau] de la classe [Rv], nous allons découvrir bientôt la nécessité de poser un filtre jSON sur la classe [RV] du projet [rdvmedecins-metier-dao] :

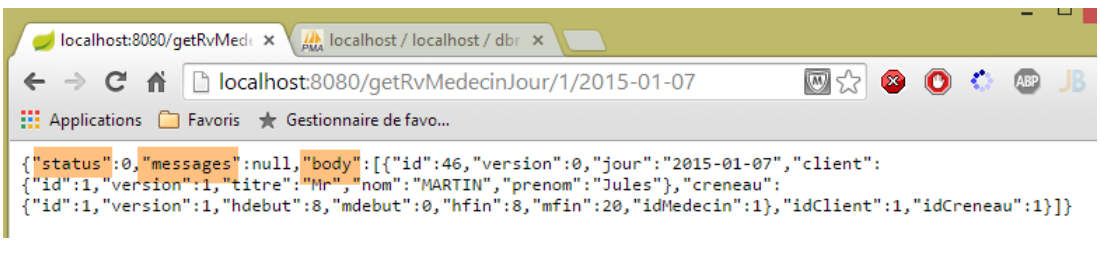
```
1.  @Entity
2.  @Table(name = "rv")
3.  @JsonFilter("rvFilter")
4.  public class Rv extends AbstractEntity {
5.  ...
```

Nous contrôlerons la sérialisation de l'objet [Rv] avec le filtre [rvFilter]. Apparemment ici, nous n'avons pas besoin de filtrer car nous avons besoin de tous les champs de l'objet de type [Rv]. Néanmoins, parce que nous avons indiqué que la classe avait un filtre jSON, nous devons définir celui-ci pour toute sérialisation d'un objet de type [Rv] sinon nous récupérons une exception. Revenons au code du contrôleur :

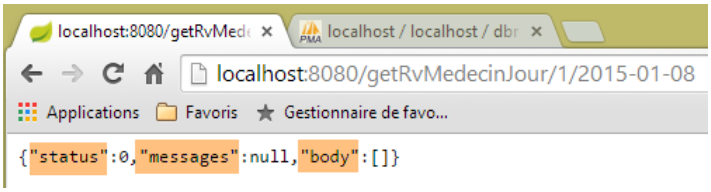
```
1.          SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept();
2.          SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
3.  mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter",
    rvFilter).addFilter("creneauFilter",creneauFilter));
```

- ligne 1 : nous indiquons que tous les champs de l'objet [Rv] doivent être sérialisés ;
- ligne 2 : nous indiquons que dans l'objet [Creneau], il ne faut pas sérialiser le champ [medecin] ;
- ligne 3 : nous ajoutons les deux filtres [rvFilter] et [creneauFilter] aux filtres jSON de l'objet [mapper] ;

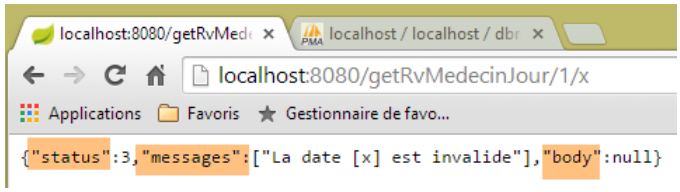
Les résultats obtenus sont les suivants :



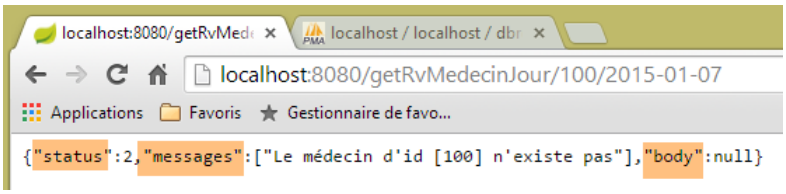
ou encore ceux-ci avec un jour sans rendez-vous :



ou encore ceux-ci avec un jour incorrect :



ou encore ceux-ci avec un médecin incorrect :



8.4.10.11 L'URL [/getAgendaMedecinJour/{idMedecin}/{jour}]

L'URL [/getAgendaMedecinJour/{idMedecin}/{jour}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.  public Response<AgendaMedecinJour> getAgendaMedecinJour(@PathVariable("idMedecin") long idMedecin,
       @PathVariable("jour") String jour, HttpServletResponse response) {
2.          // état de l'application
3.          if (messages != null) {
4.             return new Response<AgendaMedecinJour>(-1, messages, null);
5.          }
6.          // on vérifie la date
7.          Date jourAgenda = null;
8.          SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
9.          sdf.setLenient(false);
10.         try {
11.            jourAgenda = sdf.parse(jour);
12.         } catch (ParseException e) {
13.            List<String> messages = new ArrayList<String>();
14.            messages.add(String.format("La date [%s] est invalide", jour));
15.            return new Response<AgendaMedecinJour>(3, messages, null);
16.         }
17.         // on récupère le médecin
18.         Response<Medecin> responseMedecin = getMedecin(idMedecin);
19.         if (responseMedecin.getStatus() != 0) {
20.            return new Response<AgendaMedecinJour>(responseMedecin.getStatus(),
    responseMedecin.getMessages(), null);
21.         }
22.         Medecin médecin = responseMedecin.getBody();
23.         // on récupère son agenda
24.         AgendaMedecinJour agenda = null;
25.         try {
26.            agenda = application.getAgendaMedecinJour(médecin.getId(), jourAgenda);
```

```
27.         } catch (Exception e1) {
28.            return new Response<AgendaMedecinJour>(4, Static.getErreursForException(e1), agenda);
29.         }
30.         // on rend l'agenda
31.         SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept();
32.         SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
33.         mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter",
    rvFilter).addFilter("creneauFilter", creneauFilter));
34.         return new Response<AgendaMedecinJour>(0, null, agenda);
35.     }
```

- ligne 34 : on rend un agenda de type [AgendaMedecinJour] encapsulé dans un objet [Response] ;

Le type [AgendaMedecinJour] est le suivant :

```
1.  public class AgendaMedecinJour implements Serializable {
2.      // champs
3.      private Medecin medecin;
4.      private Date jour;
5.      private CreneauMedecinJour[] creneauxMedecinJour;
```

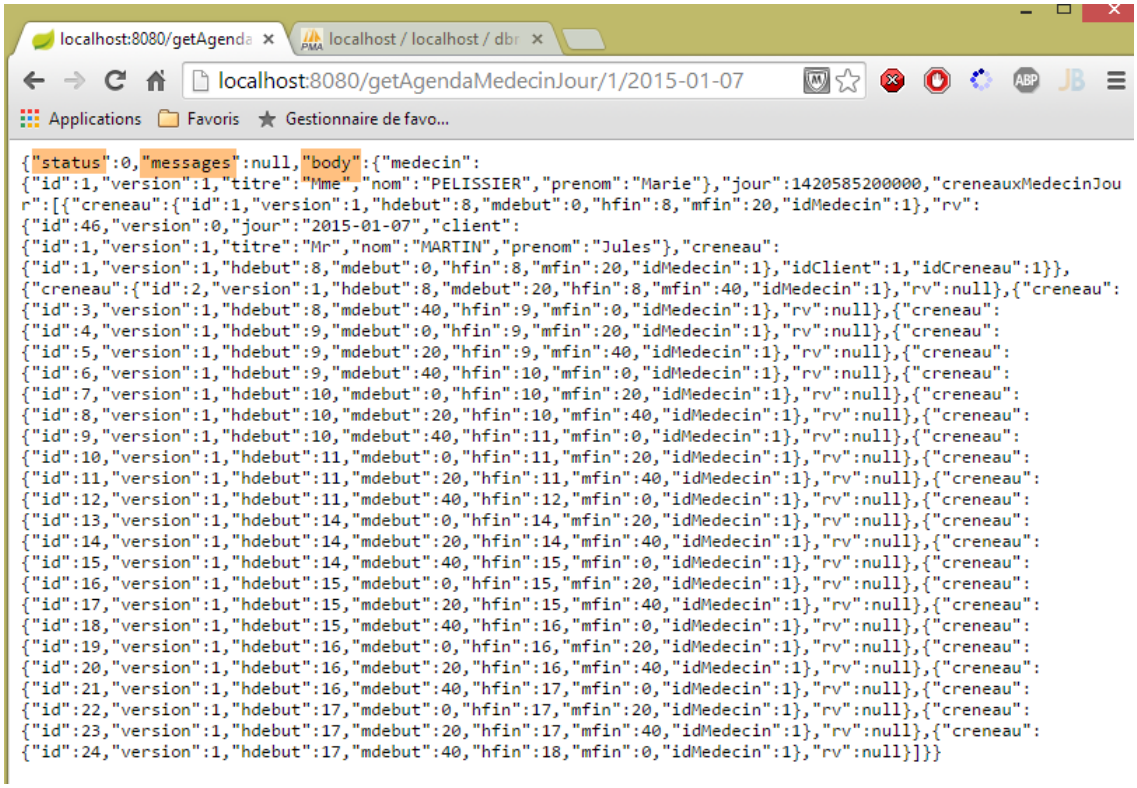
Le type [CreneauMedecinJour] est le suivant :

```
1.  public class CreneauMedecinJour implements Serializable {
2.
3.      private static final long serialVersionUID = 1L;
4.      // champs
5.      private Creneau creneau;
6.      private Rv rv;
```

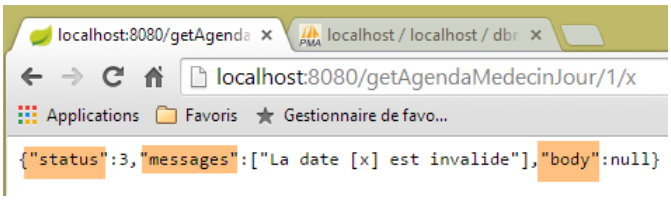
Le tableau [creneauxMedecinJour] de la classe [AgendaMedecinJour] (ligne 5) est construit à partir des résultats de la méthode précédente [`getRvMedecinJour`]. Le champ [Rv rv] du type [CreneauMedecinJour] est soit le pointeur *null*, soit un rendez-vous avec tous ses champs définis. Aussi retrouve-t-on les deux mêmes filtres jSON que dans l'URL précédente :

```
36.         SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept();
37.         SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
38.         mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter",
    rvFilter).addFilter("creneauFilter", creneauFilter));
```

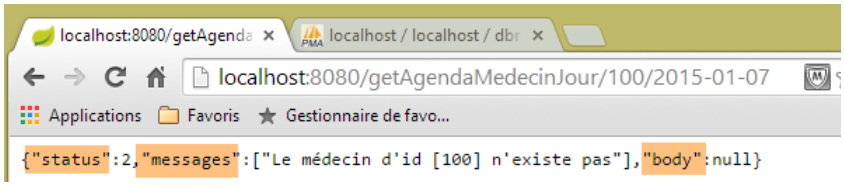
Les résultats obtenus sont les suivants :



ou bien ceux-ci si le jour est erroné :



ou bien ceux-ci si le n° du médecin est invalide :



8.4.10.12 L'URL [/getMedecinById/{id}]

L'URL [/getMedecinById/{id}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

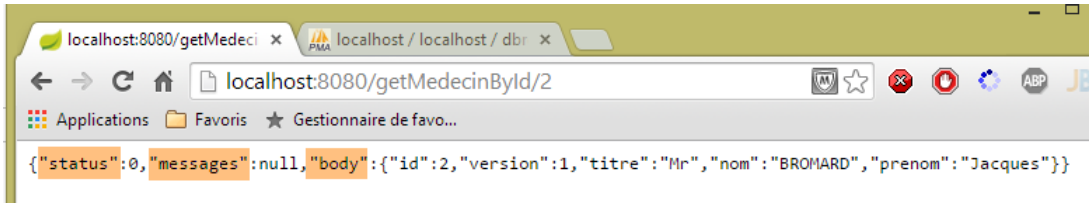
```
1.      @RequestMapping(value = "/getMedecinById/{id}", method = RequestMethod.GET)
2.      public Response<Medecin> getMedecinById(@PathVariable("id") long id) {
3.          // état de l'application
4.          if (messages != null) {
5.              return new Response<Medecin>(-1, messages, null);
6.          }
7.          // on récupère le médecin
```

```
8.          return getMedecin(id);
9.      }
```

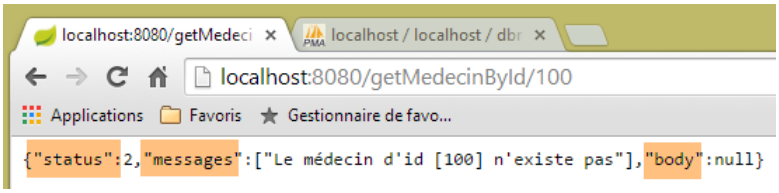
Ligne 8, la méthode [getMedecin] est la suivante :

```
1.      private Response<Medecin> getMedecin(long id) {
2.          // on récupère le médecin
3.          Medecin médecin = null;
4.          try {
5.              médecin = application.getMedecinById(id);
6.          } catch (Exception e1) {
7.              return new Response<Medecin>(1, Static.getErreursForException(e1), null);
8.          }
9.          // médecin existant ?
10.         if (médecin == null) {
11.             List<String> messages = new ArrayList<String>();
12.             messages.add(String.format("Le médecin d'id [%s] n'existe pas", id));
13.             return new Response<Medecin>(2, messages, null);
14.         }
15.         // ok
16.         return new Response<Medecin>(0, null, médecin);
17.     }
```

Les résultats obtenus sont les suivants :



ou bien ceux-ci si le n° du médecin est incorrect :



8.4.10.13 L'URL [/getClientById/{id}]

L'URL [/getClientById/{id}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.      @RequestMapping(value = "/getClientById/{id}", method = RequestMethod.GET)
2.      public Response<Client> getClientById(@PathVariable("id") long id) {
3.          // état de l'application
4.          if (messages != null) {
5.              return new Response<Client>(-1, messages, null);
6.          }
7.          // on récupère le client
8.          return getClient(id);
9.      }
```

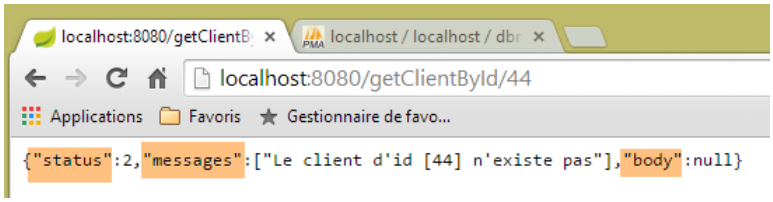
Ligne 8, la méthode [getClient] est la suivante :

```
1.      private Response<Client> getClient(long id) {
2.          // on récupère le client
3.          Client client = null;
4.          try {
```

```
5.            client = application.getClientById(id);
6.         } catch (Exception e1) {
7.            return new Response<Client>(1, Static.getErreursForException(e1), null);
8.         }
9.         // client existant ?
10.        if (client == null) {
11.           List<String> messages = new ArrayList<String>();
12.           messages.add(String.format("Le client d'id [%s] n'existe pas", id));
13.           return new Response<Client>(2, messages, null);
14.        }
15.        // ok
16.        return new Response<Client>(0, null, client);
17. }
```

Les résultats obtenus sont les suivants :

ou bien ceux-ci si le n° du client est incorrect :



8.4.10.14 L'URL [/getCreneauById/{id}]

L'URL [/getCreneauById/{id}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.     @RequestMapping(value = "/getCreneauById/{id}", method = RequestMethod.GET)
2.     public Response<Creneau> getCreneauById(@PathVariable("id") long id) {
3.        // état de l'application
4.        if (messages != null) {
5.           return new Response<Creneau>(-1, messages, null);
6.        }
7.        // on rend le créneau
8.        SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
9.        mapper.setFilters(new SimpleFilterProvider().addFilter("creneauFilter", creneauFilter));
10.       return getCreneau(id);
11. }
```

Ligne 10, la méthode [getCreneau] est la suivante :

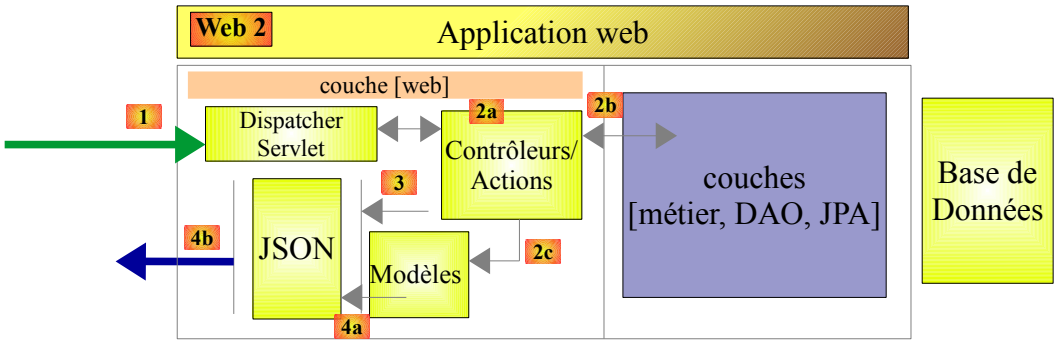
```
private Response<Creneau> getCreneau(long id) {
   // on récupère le créneau
   Creneau créneau = null;
   try {
      créneau = application.getCreneauById(id);
   } catch (Exception e1) {
      return new Response<Creneau>(1, Static.getErreursForException(e1), null);
   }
   // créneau existant ?
   if (créneau == null) {
      List<String> messages = new ArrayList<String>();
```

```
            messages.add(String.format("Le créneau d'id [%s] n'existe pas", id));
            return new Response<Creneau>(2, messages, null);
        }
        // ok
        return new Response<Creneau>(0, null, créneau);
    }
```

Rappelons le code de l'entité [Creneau] :

```
1.  @Entity
2.  @Table(name = "creneaux")
3.  @JsonFilter("creneauFilter")
4.  public class Creneau extends AbstractEntity {
5.
6.      private static final long serialVersionUID = 1L;
7.      // caractéristiques d'un créneau de RV
8.      private int hdebut;
9.      private int mdebut;
10.     private int hfin;
11.     private int mfin;
12.
13.     // un créneau est lié à un médecin
14.     @ManyToOne(fetch = FetchType.LAZY)
15.     @JoinColumn(name = "id_medecin")
16.     private Medecin medecin;
17.
18.     // clé étrangère
19.     @Column(name = "id_medecin", insertable = false, updatable = false)
20. private long idMedecin;
```

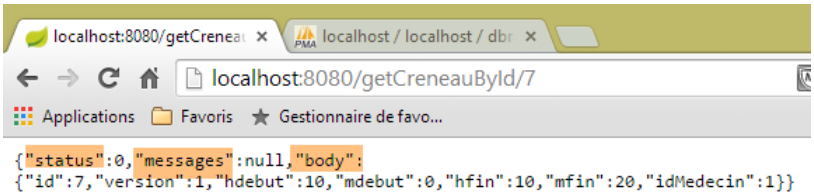
- lignes 14-16 : parce que le champ [medecin] est en mode [fetch = FetchType.***LAZY***], il n'est pas ramené lorsqu'on va chercher un créneau via son [id]. Il est donc nécessaire de l'exclure de la sérialisation. Sans cette exclusion, on a une exception. Celle-ci est dûe au fait que l'objet de sérialisation [mapper] va appeler la méthode [getMedecin] pour obtenir le champ [medecin]. Or le mode [fetch = FetchType.***LAZY***] du champ [medecin] a ramené un objet [Creneau] dont la méthode [getMedecin] est programmée pour aller chercher le médecin dans le contexte JPA. On appelle cela un objet [proxy]. Or rappelons-nous l'architecture de l'application web :



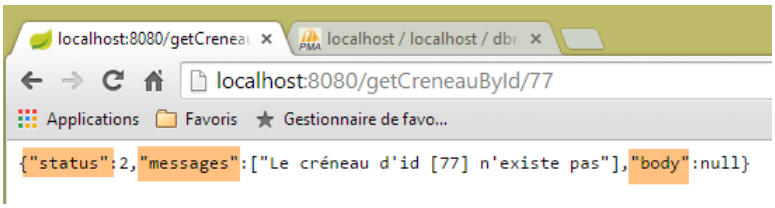
Le contrôleur se trouve dans le bloc [Contrôleurs / Actions]. Lorsqu'on est dans ce bloc, il n'y a plus de notion de contexte JPA. Ce dernier est créé le temps des opérations de la couche [DAO]. Il ne vit pas au-delà. Donc lorsque le contrôleur essaie d'avoir accès au contexte JPA, une exception se produit indiquant que celui-ci est fermé. Pour éviter cette exception, il faut empêcher la sérialisation du champ [medecin] de la classe [Rv]. C'est ce que font les instructions suivantes du contrôleur :

```
1.          // on rend le créneau
2.          SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
3.  mapper.setFilters(new SimpleFilterProvider().addFilter("creneauFilter", creneauFilter));
```

Les résultats obtenus sont les suivants :



ou ceux-ci si le n° du créneau est incorrect :



8.4.10.15 L'URL [/getRvById/{id}]

L'URL [/getRvById/{id}] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.     @RequestMapping(value = "/getRvById/{id}", method = RequestMethod.GET)
2.     public Response<Rv> getRvById(@PathVariable("id") long id) {
3.         // état de l'application
4.         if (messages != null) {
5.             return new Response<Rv>(-1, messages, null);
6.         }
7.         // on rend le rv
8.         SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept("client",
    "creneau");
9.         mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter", rvFilter));
10.        return getRv(id);
11. }
```

Ligne 10, la méthode [getRv] est la suivante :

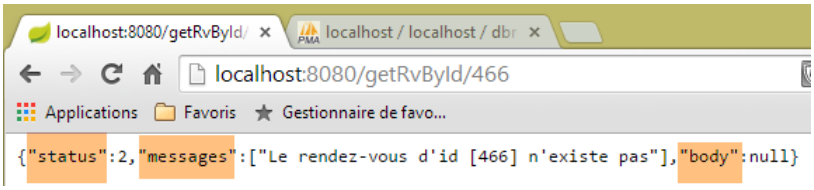
```
1.     private Response<Rv> getRv(long id) {
2.         // on récupère le Rv
3.         Rv rv = null;
4.         try {
5.             rv = application.getRvById(id);
6.         } catch (Exception e1) {
7.             return new Response<Rv>(1, Static.getErreursForException(e1), null);
8.         }
9.         // Rv existant ?
10.        if (rv == null) {
11.            List<String> messages = new ArrayList<String>();
12.            messages.add(String.format("Le rendez-vous d'id [%s] n'existe pas", id));
13.            return new Response<Rv>(2, messages, null);
14.        }
15.        // ok
16.        return new Response<Rv>(0, null, rv);
17. }
```

La classe [Rv] a deux champ avec l'annotation [`fetch = FetchType.LAZY`], les champs [creneau] et [client]. Ces champs ne sont donc pas ramenés lorsqu'on va chercher un [Rv] via sa clé primaire. Il faut donc, pour les mêmes raisons que précédemment, les exclure de la sérialisation. C'est ce que font les lignes suivantes de la méthode [getRvById] :

```
1.         SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept("client",
    "creneau");
2.  mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter", rvFilter));
```

Les résultats obtenus sont les suivants :

ou bien ceux-ci si le n° du rendez-vous est incorrect :



8.4.10.16 L'URL [/ajouterRv]

L'URL [/ajouterRv] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

```
1.  @RequestMapping(value = "/ajouterRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.      public Response<Rv> ajouterRv(@RequestBody PostAjouterRv post, HttpServletResponse response) {
3.          // état de l'application
4.          if (messages != null) {
5.              return new Response<Rv>(-1, messages, null);
6.          }
7.          // on récupère les valeurs postées
8.          String jour = post.getJour();
9.          long idCreneau = post.getIdCreneau();
10.         long idClient = post.getIdClient();
11.         // on vérifie la date
12.         Date jourAgenda = null;
13.         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
14.         sdf.setLenient(false);
15.         try {
16.             jourAgenda = sdf.parse(jour);
17.         } catch (ParseException e) {
18.             List<String> messages = new ArrayList<String>();
19.             messages.add(String.format("La date [%s] est invalide", jour));
20.             return new Response<Rv>(6, messages, null);
21.         }
22.         // on récupère le créneau
23.         Response<Creneau> responseCréneau = getCreneau(idCreneau);
24.         if (responseCréneau.getStatus() != 0) {
25.             return new Response<Rv>(responseCréneau.getStatus(), responseCréneau.getMessages(), null);
26.         }
27.         Creneau créneau = (Creneau) responseCréneau.getBody();
28.         // on récupère le client
29.         Response<Client> responseClient = getClient(idClient);
30.         if (responseClient.getStatus() != 0) {
31.             return new Response<Rv>(responseClient.getStatus() + 2, responseClient.getMessages(), null);
32.         }
33.         Client client = responseClient.getBody();
34.         // on ajoute le Rv
35.         Rv rv = null;
36.         try {
37.             rv = application.ajouterRv(jourAgenda, créneau, client);
38.         } catch (Exception e1) {
39.             return new Response<Rv>(5, Static.getErreursForException(e1), null);
40.         }
41.         // on rend la réponse
```

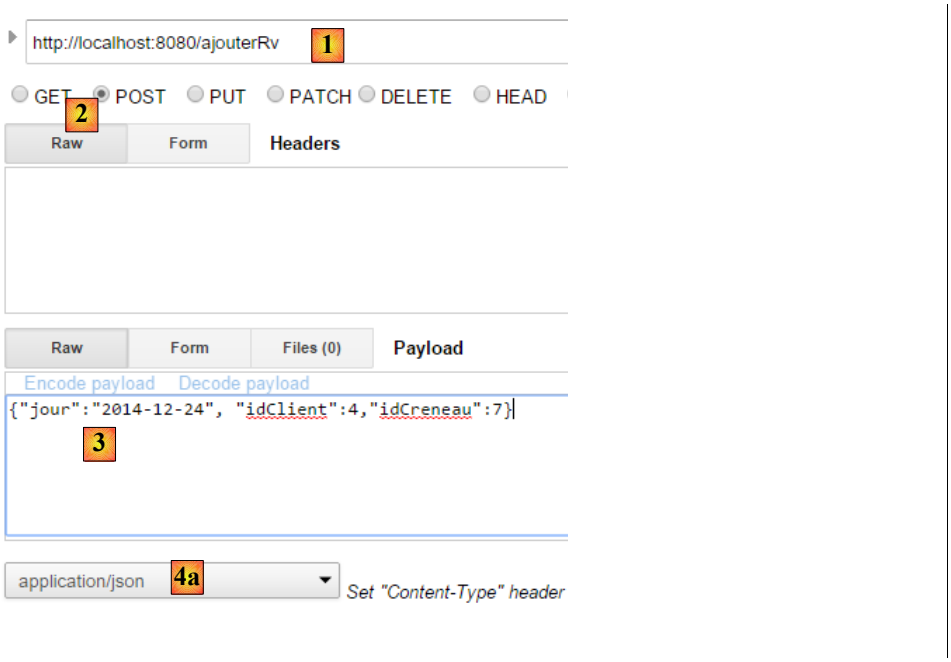
```
42.         SimpleBeanPropertyFilter rvFilter = SimpleBeanPropertyFilter.serializeAllExcept();
43.         SimpleBeanPropertyFilter creneauFilter = SimpleBeanPropertyFilter.serializeAllExcept("medecin");
44.         mapper.setFilters(new SimpleFilterProvider().addFilter("rvFilter",
    rvFilter).addFilter("creneauFilter", creneauFilter));
45.         return new Response<Rv>(0, null, rv);
46.     }
```

- ligne 2 : l'annotation [@RequestBody PostAjouterRv post] récupère le corps du POST et le met dans le paramètre [PostAjouterRv post]. Ce corps est du jSON [consumes = "application/json; charset=UTF-8"] qui va être désérialisé automatiquement dans le type [PostAjouterRv] suivant :

```
1. public class PostAjouterRv {
2.
3.     // données du post
4.     private String jour;
5.     private long idClient;
6.     private long idCreneau;
7. ...
```

- ensuite il y a du code qui a déjà été rencontré sous une forme ou une autre ;
- lignes 42-44 : la mise en place des filtres jSON [creneauFilter] et [rvFilter]. La méthode rend un type [Response<Rv>] (ligne 2). L'objet [Rv] est celui obtenu ligne 37. Si on examine la méthode [ajouterRv], on découvre qu'elle rend un objet [Rv] encapsulant un objet [Creneau] ainsi qu'un objet [Client]. L'objet [Creneau] a une dépendance [`FetchType.LAZY`] sur un objet [Medecin] et a été obtenu lignes 23-27. Il a été cherché dans le contexte JPA via sa clé primaire et a été obtenu sans sa dépendance [[`FetchType.LAZY`]. Au final,
  - l'objet [Rv] a toutes ses dépendances. Elles peuvent être sérialisées (ligne 42) ;
  - l'objet [Creneau] n'a pas sa dépendance [medecin]. Il faut donc que celle-ci ne soit pas sérialisée (ligne 43) ;

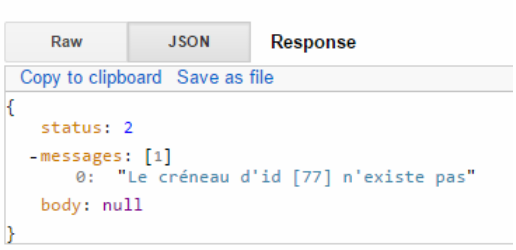
Les résultats obtenus ressemblent à ceci avec le client [Advanced Rest Client] :



- en [1], l'URL du POST ;
- en [2], le POST ;
- en [3], la valeur postée ;
- en [4a], cette valeur postée est du jSON ;

- en [4b], le client indique qu'il envoie du jSON ;
- en [5], le serveur indique qu'il renvoie du jSON ;
- en [6], la réponse jSON du serveur qui représente le rendez-vous ajouté ;
- en [7], l'identifiant du rendez-vous ajouté ;

On obtient la chose suivante avec un n° de créneau inexistant :



8.4.10.17 L'URL [/supprimerRv]

L'URL [/supprimerRv] est traitée par la méthode suivante du contrôleur [RdvMedecinsController] :

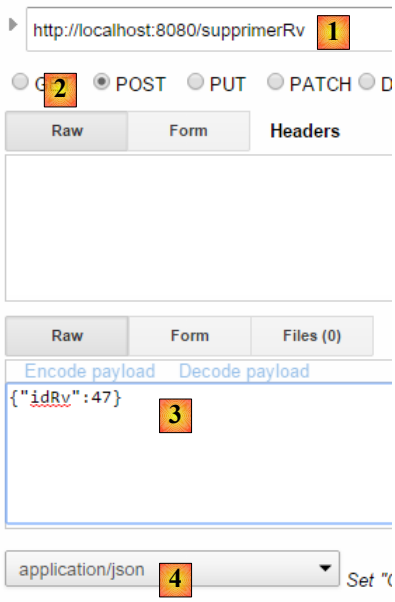
```
1.  @RequestMapping(value = "/supprimerRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.      public Response<Void> supprimerRv(@RequestBody PostSupprimerRv post, HttpServletResponse response) {
3.          // état de l'application
4.          if (messages != null) {
5.              return new Response<Void>(-1, messages, null);
6.          }
7.          // on récupère les valeurs postées
8.          long idRv = post.getIdRv();
9.          // on récupère le rv
10.         Response<Rv> responseRv = getRv(idRv);
11.         if (responseRv.getStatus() != 0) {
12.             return new Response<Void>(responseRv.getStatus(), responseRv.getMessages(), null);
13.         }
14.         // suppression du rv
15.         try {
16.             application.supprimerRv(idRv);
17.         } catch (Exception e1) {
18.             return new Response<Void>(3, Static.getErreursForException(e1), null);
19.         }
20.         // ok
21.         return new Response<Void>(0, null, null);
22.     }
```

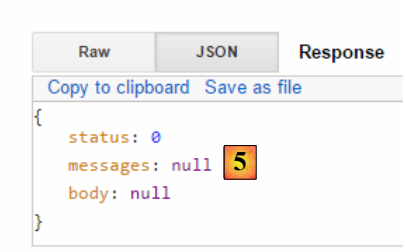
- ligne 2 : le type [Void] est la classe correspondant au type primitif [void] ;
- ligne 2 : la méthode a pour paramètre le corps du POST, ç-à-d la valeur postée. Celle-ci est reçue sous forme jSON [consumes = "application/json; charset=UTF-8"] et désérialisée automatiquement dans le type [PostSupprimerRv] suivant :

```
1. public class PostSupprimerRv {
2.
3.     // données du post
4.     private long idRv;
5.
```

- ligne 21 : lorsque la suppression a réussi, on envoie une réponse avec [status=0] ;

Les résultats obtenus sont les suivants :



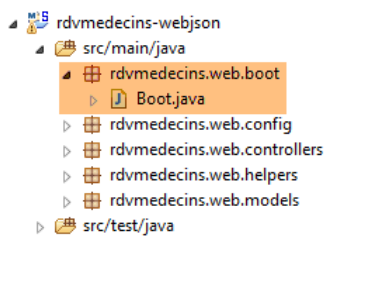


- en [5], le champ [status=0] indique que la suppression a réussi ;

Avec un n° de rendez-vous qui n'existe pas, on obtient la chose suivante :

Nous en avons terminé avec le contrôleur. Nous voyons maintenant comment exécuter le projet.

8.4.10.18 La classe exécutable du service web



La classe [Boot] est la suivante :

```
1.  package rdvmedecins.web.boot;
2.
3.  import org.springframework.boot.SpringApplication;
4.
5.  import rdvmedecins.web.config.AppConfig;
6.
7.  public class Boot {
8.
9.      public static void main(String[] args) {
10.        SpringApplication.run(AppConfig.class, args);
11.     }
12. }
```

Ligne 10, la méthode statique [SpringApplication.*run*] est exécutée avec comme premier paramètre, la classe [AppConfig] de configuration du projet. Cette méthode va procéder à l'auto-configuration du projet, lancer le serveur Tomcat embarqué dans les dépendances et y déployer le contrôleur [RdvMedecinsController].

Les logs à l'exécution sont les suivants :

```
1.  .   ____          _            __ _ _
2.   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
3.  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
4.   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
5.    '  |____| .__|_| |_|_| |_\__, | / / / /
6.   =========|_|==============|___/=/_/_/_/
7.   :: Spring Boot ::        (v1.2.0.RELEASE)
8.
9.  2014-12-24 14:11:19.900  INFO 10860 --- [           main] rdvmedecins.web.boot.Boot                :
    Starting Boot on Gportpers3 with PID 10860 (started by ST in D:\data\istia-1415\spring mvc\dvp-
    final\etude-de-cas\rdvmedecins-webjson)
10. 2014-12-24 14:11:19.940  INFO 10860 --- [           main] ationConfigEmbeddedWebApplicationContext :
    Refreshing
    org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@dbd940d: startup
    date [Wed Dec 24 14:11:19 CET 2014]; root of context hierarchy
11. 2014-12-24 14:11:21.028  INFO 10860 --- [           main] o.s.b.f.s.DefaultListableBeanFactory     :
    Overriding bean definition for bean 'beanNameViewResolver': replacing [Root bean: class [null]; scope=;
    abstract=false; lazyInit=false; autowireMode=3; dependencyCheck=0; autowireCandidate=true;
    primary=false;
    factoryBeanName=org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorView
    Configuration; factoryMethodName=beanNameViewResolver; initMethodName=null;
    destroyMethodName=(inferred); defined in class path resource
    [org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration.c
    lass]] with [Root bean: class [null]; scope=; abstract=false; lazyInit=false; autowireMode=3;
    dependencyCheck=0; autowireCandidate=true; primary=false;
    factoryBeanName=org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurati
    onAdapter; factoryMethodName=beanNameViewResolver; initMethodName=null; destroyMethodName=(inferred);
    defined in class path resource
    [org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter.class
    ]]
12. 2014-12-24 14:11:21.513  INFO 10860 --- [           main] trationDelegate$BeanPostProcessorChecker :
    Bean 'org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration' of
    type [class org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration]
    is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-
    proxying)
13. 2014-12-24 14:11:21.570  INFO 10860 --- [           main] trationDelegate$BeanPostProcessorChecker :
    Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of type [class
    org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$
    $EnhancerBySpringCGLIB$$5060b4d6] is not eligible for getting processed by all BeanPostProcessors (for
    example: not eligible for auto-proxying)
```

```
14. 2014-12-24 14:11:21.597  INFO 10860 --- [           main] trationDelegate$BeanPostProcessorChecker :
    Bean 'transactionAttributeSource' of type [class
    org.springframework.transaction.annotation.AnnotationTransactionAttributeSource] is not eligible for
    getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
15. 2014-12-24 14:11:21.609  INFO 10860 --- [           main] trationDelegate$BeanPostProcessorChecker :
    Bean 'transactionInterceptor' of type [class
    org.springframework.transaction.interceptor.TransactionInterceptor] is not eligible for getting
    processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
16. 2014-12-24 14:11:21.615  INFO 10860 --- [           main] trationDelegate$BeanPostProcessorChecker :
    Bean 'org.springframework.transaction.config.internalTransactionAdvisor' of type [class
    org.springframework.transaction.interceptor.BeanFactoryTransactionAttributeSourceAdvisor] is not
    eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
17. 2014-12-24 14:11:21.984  INFO 10860 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer :
    Tomcat initialized with port(s): 8080/http
18. 2014-12-24 14:11:22.268  INFO 10860 --- [           main] o.apache.catalina.core.StandardService   :
    Starting service Tomcat
19. 2014-12-24 14:11:22.270  INFO 10860 --- [           main] org.apache.catalina.core.StandardEngine  :
    Starting Servlet Engine: Apache Tomcat/8.0.15
20. 2014-12-24 14:11:22.387  INFO 10860 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
    Initializing Spring embedded WebApplicationContext
21. 2014-12-24 14:11:22.387  INFO 10860 --- [ost-startStop-1] o.s.web.context.ContextLoader            :
    Root WebApplicationContext: initialization completed in 2450 ms
22. 2014-12-24 14:11:23.943  INFO 10860 --- [ost-startStop-1] j.LocalContainerEntityManagerFactoryBean :
    Building JPA container EntityManagerFactory for persistence unit 'default'
23. 2014-12-24 14:11:23.955  INFO 10860 --- [ost-startStop-1] o.hibernate.jpa.internal.util.LogHelper  :
    HHH000204: Processing PersistenceUnitInfo [
24.     name: default
25.     ...]
26. 2014-12-24 14:11:24.037  INFO 10860 --- [ost-startStop-1] org.hibernate.Version                    :
    HHH000412: Hibernate Core {4.3.7.Final}
27. 2014-12-24 14:11:24.039  INFO 10860 --- [ost-startStop-1] org.hibernate.cfg.Environment            :
    HHH000206: hibernate.properties not found
28. 2014-12-24 14:11:24.041  INFO 10860 --- [ost-startStop-1] org.hibernate.cfg.Environment            :
    HHH000021: Bytecode provider name : javassist
29. 2014-12-24 14:11:24.356  INFO 10860 --- [ost-startStop-1] o.hibernate.annotations.common.Version   :
    HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
30. 2014-12-24 14:11:24.495  INFO 10860 --- [ost-startStop-1] org.hibernate.dialect.Dialect            :
    HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
31. 2014-12-24 14:11:24.703  INFO 10860 --- [ost-startStop-1] o.h.h.i.ast.ASTQueryTranslatorFactory    :
    HHH000397: Using ASTQueryTranslatorFactory
32. 2014-12-24 14:11:26.242  INFO 10860 --- [ost-startStop-1] o.s.s.web.DefaultSecurityFilterChain     :
    Creating filter chain: org.springframework.security.web.util.matcher.AnyRequestMatcher@1,
    [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@57305956,
    org.springframework.security.web.context.SecurityContextPersistenceFilter@2839e1b6,
    org.springframework.security.web.header.HeaderWriterFilter@41521aa0,
    org.springframework.security.web.authentication.logout.LogoutFilter@5463224a,
    org.springframework.security.web.savedrequest.RequestCacheAwareFilter@37db2908,
    org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@58c45681,
    org.springframework.security.web.authentication.AnonymousAuthenticationFilter@13f93a48,
    org.springframework.security.web.session.SessionManagementFilter@7d499acf,
    org.springframework.security.web.access.ExceptionTranslationFilter@1aa64422]
33. 2014-12-24 14:11:26.328  INFO 10860 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
    Mapping servlet: 'dispatcherServlet' to [/]
34. 2014-12-24 14:11:26.334  INFO 10860 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
    Mapping filter: 'springSecurityFilterChain' to: [/*]
35. 2014-12-24 14:11:26.335  INFO 10860 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
    Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
36. 2014-12-24 14:11:26.335  INFO 10860 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
    Mapping filter: 'characterEncodingFilter' to: [/*]
37. 2014-12-24 14:11:26.574  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter :
    Looking for @ControllerAdvice:
    org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@dbd940d: startup
    date [Wed Dec 24 14:11:19 CET 2014]; root of context hierarchy
38. 2014-12-24 14:11:26.650  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAgendaMedecinJour/{idMedecin}/
    {jour}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    rdvmedecins.web.models.Response<rdvmedecins.domain.AgendaMedecinJour>
    rdvmedecins.web.controllers.RdvMedecinsController.getAgendaMedecinJour(long,java.lang.String,javax.servl
    et.http.HttpServletResponse)
39. 2014-12-24 14:11:26.650  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getMedecinById/{id}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}"
    onto public rdvmedecins.web.models.Response<rdvmedecins.entities.Medecin>
    rdvmedecins.web.controllers.RdvMedecinsController.getMedecinById(long)
40. 2014-12-24 14:11:26.651  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getRvMedecinJour/{idMedecin}/
```

```
    {jour}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    rdvmedecins.web.models.Response<java.util.List<rdvmedecins.entities.Rv>>
    rdvmedecins.web.controllers.RdvMedecinsController.getRvMedecinJour(long,java.lang.String)
41. 2014-12-24 14:11:26.651  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getRvById/{id}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
    public rdvmedecins.web.models.Response<rdvmedecins.entities.Rv>
    rdvmedecins.web.controllers.RdvMedecinsController.getRvById(long)
42. 2014-12-24 14:11:26.651  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAllCreneaux/
    {idMedecin}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    rdvmedecins.web.models.Response<java.util.List<rdvmedecins.entities.Creneau>>
    rdvmedecins.web.controllers.RdvMedecinsController.getAllCreneaux(long)
43. 2014-12-24 14:11:26.652  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/ajouterRv],methods=[POST],params=[],headers=[],consumes=[application/json;charset=UTF-
    8],produces=[],custom=[]}" onto public rdvmedecins.web.models.Response<rdvmedecins.entities.Rv>
    rdvmedecins.web.controllers.RdvMedecinsController.ajouterRv(rdvmedecins.web.models.PostAjouterRv,javax.s
    ervlet.http.HttpServletResponse)
44. 2014-12-24 14:11:26.652  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAllClients],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
    public rdvmedecins.web.models.Response<java.util.List<rdvmedecins.entities.Client>>
    rdvmedecins.web.controllers.RdvMedecinsController.getAllClients(javax.servlet.http.HttpServletResponse)
45. 2014-12-24 14:11:26.652  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAllMedecins],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
    public rdvmedecins.web.models.Response<java.util.List<rdvmedecins.entities.Medecin>>
    rdvmedecins.web.controllers.RdvMedecinsController.getAllMedecins(javax.servlet.http.HttpServletResponse)
46. 2014-12-24 14:11:26.652  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getCreneauById/{id}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}"
    onto public rdvmedecins.web.models.Response<rdvmedecins.entities.Creneau>
    rdvmedecins.web.controllers.RdvMedecinsController.getCreneauById(long)
47. 2014-12-24 14:11:26.653  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getClientById/{id}],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}"
    onto public rdvmedecins.web.models.Response<rdvmedecins.entities.Client>
    rdvmedecins.web.controllers.RdvMedecinsController.getClientById(long)
48. 2014-12-24 14:11:26.653  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/supprimerRv],methods=[POST],params=[],headers=[],consumes=[application/json;charset=UTF-
    8],produces=[],custom=[]}" onto public rdvmedecins.web.models.Response<java.lang.Void>
    rdvmedecins.web.controllers.RdvMedecinsController.supprimerRv(rdvmedecins.web.models.PostSupprimerRv,jav
    ax.servlet.http.HttpServletResponse)
49. 2014-12-24 14:11:26.654  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAgendaMedecinJour/{idMedecin}/
    {jour}],methods=[OPTIONS],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public void
    rdvmedecins.web.controllers.RdvMedecinsCorsController.getAgendaMedecinJour(javax.servlet.http.HttpServle
    tResponse)
50. 2014-12-24 14:11:26.654  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/ajouterRv],methods=[OPTIONS],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
    public void
    rdvmedecins.web.controllers.RdvMedecinsCorsController.ajouterRv(javax.servlet.http.HttpServletResponse)
51. 2014-12-24 14:11:26.654  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAllClients],methods=[OPTIONS],params=[],headers=[],consumes=[],produces=[],custom=[]}"
    onto public void
    rdvmedecins.web.controllers.RdvMedecinsCorsController.getAllClients(javax.servlet.http.HttpServletRespon
    se)
52. 2014-12-24 14:11:26.655  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/getAllMedecins],methods=[OPTIONS],params=[],headers=[],consumes=[],produces=[],custom=[]}"
    onto public void
    rdvmedecins.web.controllers.RdvMedecinsCorsController.getAllMedecins(javax.servlet.http.HttpServletRespo
    nse)
53. 2014-12-24 14:11:26.655  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/supprimerRv],methods=[OPTIONS],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto
    public void
    rdvmedecins.web.controllers.RdvMedecinsCorsController.supprimerRv(javax.servlet.http.HttpServletResponse
    )
54. 2014-12-24 14:11:26.657  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public
    org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>>
    org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequ
    est)
55. 2014-12-24 14:11:26.657  INFO 10860 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping :
    Mapped "{[/error],methods=[],params=[],headers=[],consumes=[],produces=[text/html],custom=[]}" onto
    public org.springframework.web.servlet.ModelAndView
    org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServlet
    Request)
56. 2014-12-24 14:11:26.690  INFO 10860 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
    Mapped URL path [/webjars/**] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
```

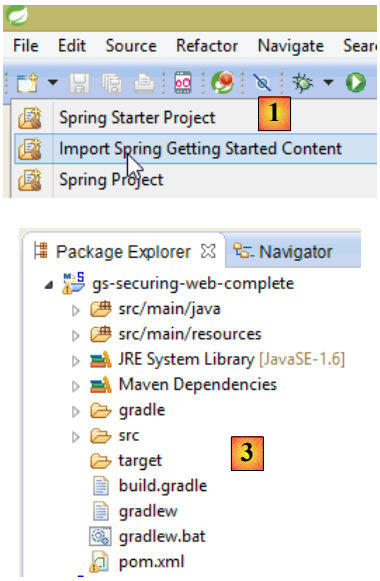
```
57. 2014-12-24 14:11:26.691  INFO 10860 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
    Mapped URL path [/**] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
58. 2014-12-24 14:11:26.736  INFO 10860 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  :
    Mapped URL path [/**/favicon.ico] onto handler of type [class
    org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
59. 2014-12-24 14:11:27.009  INFO 10860 --- [           main] b.a.s.AuthenticationManagerConfiguration :
60.
61. Using default security password: 444a7843-5ec2-41c8-9e1b-a457615aafb7
62.
63.
64. 2014-12-24 14:11:27.022  INFO 10860 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
    Registering beans for JMX exposure on startup
65. 2014-12-24 14:11:27.093  INFO 10860 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer :
    Tomcat started on port(s): 8080/http
66. 2014-12-24 14:11:27.095  INFO 10860 --- [           main] rdvmedecins.web.boot.Boot                :
    Started Boot in 7.415 seconds (JVM running for 7.995)
67. 2014-12-24 14:11:27.318  INFO 10860 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
    Initializing Spring FrameworkServlet 'dispatcherServlet'
68. 2014-12-24 14:11:27.318  INFO 10860 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        :
    FrameworkServlet 'dispatcherServlet': initialization started
69. 2014-12-24 14:11:27.381  INFO 10860 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        :
    FrameworkServlet 'dispatcherServlet': initialization completed in 62 ms
```

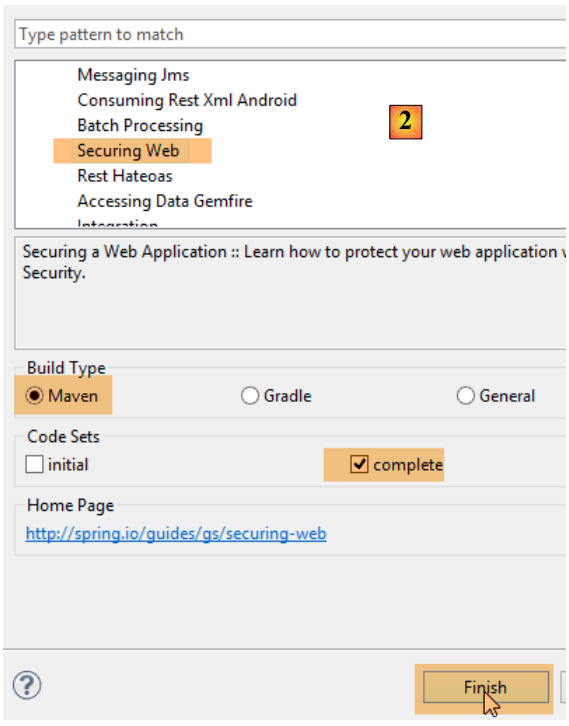
- ligne 17 : le serveur Tomcat démarre ;
- lignes 22-30 : les couches [métier, DAO, JPA] s'initialisent ;
- ligne 38 : la méthode traitant l'URL [/getAgendaMedecinJour/{idMedecin}/{jour}] a été découverte. Ce processus de découverte des méthodes du contrôleur se répète jusqu'à la ligne 53 ;
- ligne 69 : la servlet de Spring MVC [DispatcherServlet] est prête à répondre aux demandes de clients web ;

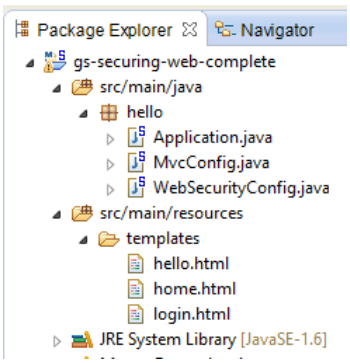
Nous avons désormais un service web opérationnel interrogeable avec un client web. Nous abordons maintenant la sécurisation de ce service : nous voulons que seules certaines personnes puissent gérer les rendez-vous des médecins. Nous allons utiliser pour cela le framework Spring Security, une branche de l'écosystème Spring.

8.4.11 Introduction à Spring Security

Nous allons de nouveau importer un guide Spring en suivant les étapes 1 à 3 ci-dessous :







Le projet se compose des éléments suivants :

- dans le dossier [templates], on trouve les pages HTML du projet ;
- [Application] : est la classe exécutable du projet ;
- [MvcConfig] : est la classe de configuration de Spring MVC ;
- [WebSecurityConfig] : est la classe de configuration de Spring Security ;

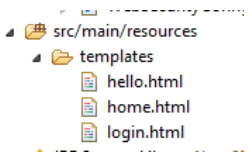
8.4.11.1 Configuration Maven

Le projet [3] est un projet Maven. Examinons son fichier [pom.xml] pour connaître ses dépendances :

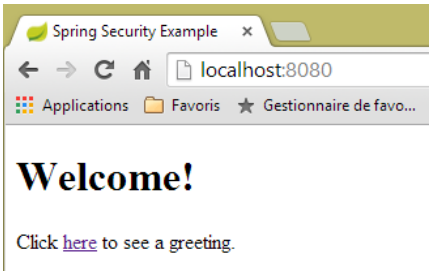
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>org.springframework</groupId>
7.     <artifactId>gs-securing-web</artifactId>
8.     <version>0.1.0</version>
9.
10.    <parent>
11.       <groupId>org.springframework.boot</groupId>
12.       <artifactId>spring-boot-starter-parent</artifactId>
13.       <version>1.1.10.RELEASE</version>
14.    </parent>
15.
16.    <dependencies>
17.       <dependency>
18.          <groupId>org.springframework.boot</groupId>
19.          <artifactId>spring-boot-starter-thymeleaf</artifactId>
20.       </dependency>
21.       <!-- tag::security[] -->
22.       <dependency>
23.          <groupId>org.springframework.boot</groupId>
24.          <artifactId>spring-boot-starter-security</artifactId>
25.       </dependency>
26.       <!-- end::security[] -->
27.    </dependencies>
28.
29.    <properties>
30.       <start-class>hello.Application</start-class>
31.    </properties>
32.
33.    <build>
34.       <plugins>
35.          <plugin>
36.             <groupId>org.springframework.boot</groupId>
37.             <artifactId>spring-boot-maven-plugin</artifactId>
38.          </plugin>
39.       </plugins>
40.    </build>
41.
42. </project>
```

- lignes 10-14 : le projet est un projet Spring Boot ;
- lignes 17-20 : dépendance sur le framework [Thymeleaf] ;
- lignes 22-25 : dépendance sur le framework Spring Security ;

8.4.11.2 Les vues Thymeleaf



La vue [home.html] est la suivante :



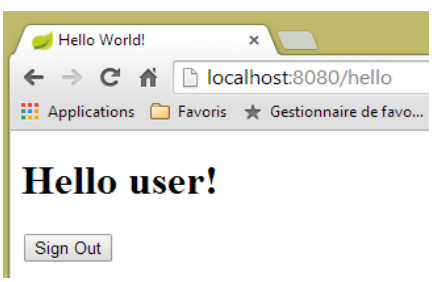

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml"
3.     xmlns:th="http://www.thymeleaf.org"
4.     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
5.  <head>
6.  <title>Spring Security Example</title>
7.  </head>
8.  <body>
9.     <h1>Welcome!</h1>
10.
11.    <p>
12.       Click <a th:href="@{/hello}">here</a> to see a greeting.
13.    </p>
14. </body>
15. </html>
```

- ligne 12 : l'attribut [th:href="@{/*hello*}"] va générer l'attribut [href] de la balise <a>. La valeur [@{/*hello*}] va générer le chemin [<context>/hello] où [context] est le contexte de l'application web ;

Le code HTML généré est le suivant :

```
1.  <!DOCTYPE html>
2.
3.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-
    springsecurity3">
4.     <head>
5.        <title>Spring Security Example</title>
6.     </head>
7.     <body>
8.        <h1>Welcome!</h1>
9.
10.       <p>
11.          Click
12.          <a href="/hello">here</a>
13.          to see a greeting.
14.       </p>
15.    </body>
16. </html>
```

La vue [hello.html] est la suivante :




```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml"
3.     xmlns:th="http://www.thymeleaf.org"
4.     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
5.  <head>
6.  <title>Hello World!</title>
7.  </head>
8.  <body>
9.     <h1 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!</h1>
10.    <form th:action="@{/logout}" method="post">
11.       <input type="submit" value="Sign Out" />
12.    </form>
13. </body>
14. </html>
```

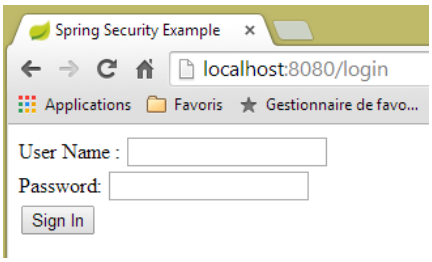
- ligne 9 : L'attribut [th:inline="*text*"] va générer le texte de la balise <h1>. Ce texte contient une expression $ qui doit être évaluée. L'élément [[${#httpServletRequest.remoteUser}]] est la valeur de l'attribut [RemoteUser] de la requête HTTP courante. C'est le nom de l'utilisateur connecté ;
- ligne 10 : un formulaire HTML. L'attribut [th:action="*@{/logout}*"] va générer l'attribut [action] de la balise [form]. La valeur [*@{/logout}*] va générer le chemin [<context>/logout] où [context] est le contexte de l'application web ;

Le code HTML généré est le suivant :

```
1.  <!DOCTYPE html>
2.
3.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-
    springsecurity3">
4.     <head>
5.        <title>Hello World!</title>
6.     </head>
7.     <body>
8.        <h1>Hello user!</h1>
9.        <form method="post" action="/logout">
10.          <input type="submit" value="Sign Out" />
11.          <input type="hidden" name="_csrf" value="b152e5b9-d1a4-4492-b89d-b733fe521c91" />
12.       </form>
13.    </body>
14. </html>
```

- ligne 8 : la traduction de *Hello [[${#httpServletRequest.remoteUser}]]!*;
- ligne 9 : la traduction de *@{/logout}* ;
- ligne 11 : un champ caché appelé (attribut *name*) **_csrf** ;

La dernière vue [login.html] est la suivante :




```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml"
3.     xmlns:th="http://www.thymeleaf.org"
4.     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
5.  <head>
6.  <title>Spring Security Example</title>
7.  </head>
8.  <body>
9.     <div th:if="${param.error}">Invalid username and password.</div>
10.    <div th:if="${param.logout}">You have been logged out.</div>
11.    <form th:action="@{/login}" method="post">
12.       <div>
13.          <label> User Name : <input type="text" name="username" />
14.          </label>
15.       </div>
16.       <div>
17.          <label> Password: <input type="password" name="password" />
18.          </label>
19.       </div>
20.       <div>
21.          <input type="submit" value="Sign In" />
22.       </div>
23.    </form>
24. </body>
25. </html>
```

- ligne 9 : l'attribut [th:if="${param.error}"] fait que la balise ne sera générée que si l'URL qui affiche la page de login contient le paramètre [error] (http://context/login?error);
- ligne 10 : l'attribut [th:if="${param.logout}"] fait que la balise ne sera générée que si l'URL qui affiche la page de login contient le paramètre [logout] (http://context/login?logout);
- lignes 11-23 : un formulaire HTML ;
- ligne 11 : le formulaire sera posté à l'URL [<context>/login] où <context> est le contexte de l'application web ;
- ligne 13 : un champ de saisie nommé [username] ;
- ligne 17 : un champ de saisie nommé [password] ;

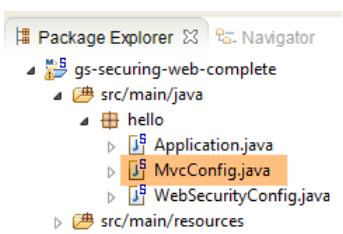
Le code HTML généré est le suivant :

```
1.  <!DOCTYPE html>
2.
3.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-
    springsecurity3">
4.     <head>
5.        <title>Spring Security Example </title>
6.     </head>
7.     <body>
8.
9.        <div>
10.          You have been logged out.
11.       </div>
12.       <form method="post" action="/login">
13.          <div>
14.             <label>
15.                User Name :
16.                <input type="text" name="username" />
17.             </label>
18.          </div>
19.          <div>
20.             <label>
```

```
21.                 Password:
22.                 <input type="password" name="password" />
23.             </label>
24.         </div>
25.         <div>
26.             <input type="submit" value="Sign In" />
27.         </div>
28.         <input type="hidden" name="_csrf" value="ef809b0a-88b4-4db9-bc53-342216b77632" />
29.     </form>
30. </body>
31. </html>
```

On notera ligne 28 que Thymeleaf a ajouté un champ caché nommé [**_csrf**].

8.4.11.3 Configuration Spring MVC



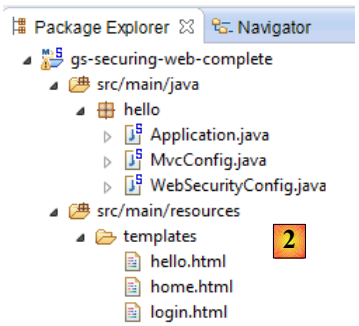
La classe [MvcConfig] configure le framework Spring MVC :

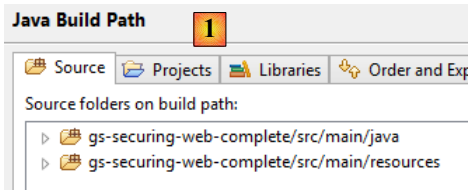
```
1.  package hello;
2.
3.  import org.springframework.context.annotation.Configuration;
4.  import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
5.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
6.
7.  @Configuration
8.  public class MvcConfig extends WebMvcConfigurerAdapter {
9.
10.     @Override
11.     public void addViewControllers(ViewControllerRegistry registry) {
12.         registry.addViewController("/home").setViewName("home");
13.         registry.addViewController("/").setViewName("home");
14.         registry.addViewController("/hello").setViewName("hello");
15.         registry.addViewController("/login").setViewName("login");
16.     }
17.
18. }
```

- ligne 7 : l'annotation [@Configuration] fait de la classe [MvcConfig] une classe de configuration ;
- ligne 8 : la classe [MvcConfig] étend la classe [WebMvcConfigurerAdapter] pour en redéfinir certaines méthodes ;
- ligne 10 : redéfinition d'une méthode de la classe parent ;
- lignes 11- 16 : la méthode [addViewControllers] permet d'associer des URL à des vues HTML. Les associations suivantes y sont faites :

URL	vue
/, /home	/templates/home.html
/hello	/templates/hello.html
/login	/templates/login.html

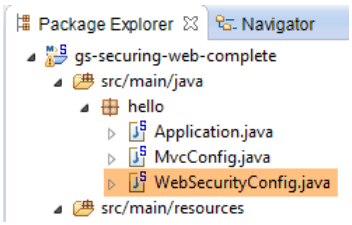
Le suffixe [html] et le dossier [templates] sont les valeurs par défaut utilisées par Thymeleaf. Elles peuvent être changées par configuration. Le dossier [templates] doit être à la racine du Classpath du projet :





Ci-dessus [1], les dossiers [java] et [resources] sont tous les deux des dossier source (source folders). Cela implique que leur contenu sera à la racine du Classpath du projet. Donc en [2], les dossiers [hello] et [templates] seront à la racine du Classpath.

8.4.11.4 Configuration Spring Security



La classe [WebSecurityConfig] configure le framework Spring Security :

```
1.  package hello;
2.
3.  import org.springframework.context.annotation.Configuration;
4.  import
    org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
5.  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6.  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
7.  import org.springframework.security.config.annotation.web.servlet.configuration.EnableWebMvcSecurity;
8.
9.  @Configuration
10. @EnableWebMvcSecurity
11. public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
12.     @Override
13.     protected void configure(HttpSecurity http) throws Exception {
14.         http.authorizeRequests().antMatchers("/", "/home").permitAll().anyRequest().authenticated();
15.         http.formLogin().loginPage("/login").permitAll().and().logout().permitAll();
16.     }
17.
18.     @Override
19.     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
20.         auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");
21.     }
22. }
```

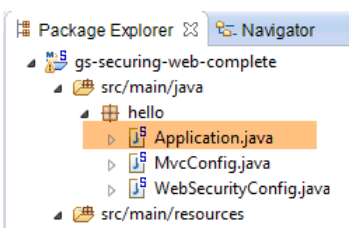
- ligne 9 : l'annotation [@Configuration] fait de la classe [WebSecurityConfig] une classe de configuration ;
- ligne 10 : l'annotation [@EnableWebSecurity] fait de la classe [WebSecurityConfig] une classe de configuration de Spring Security ;
- ligne 11 : la classe [WebSecurity] étend la classe [WebSecurityConfigurerAdapter] pour en redéfinir certaines méthodes ;
- ligne 12 : redéfinition d'une méthode de la classe parent ;
- lignes 13- 16 : la méthode [configure(HttpSecurity http)] est redéfinie pour définir les droits d'accès aux différentes URL de l'application ;
- ligne 14 : la méthode [http.authorizeRequests()] permet d'associer des URL à des droits d'accès. Les associations suivantes y sont faites :

URL	**régle**	**code**

/, /home	accès sans être authentifié	http.authorizeRequests().antMatchers("/", "/home").permitAll()
autres URL	accès authentifié uniquement	http.anyRequest().authenticated();

- ligne 15 : définit la méthode d'authentification. L'authentification se fait via un formulaire d'URL [/login] accessible à tous [http.formLogin().loginPage("/login").permitAll()]. La déconnexion (logout) est également accessible à tous ;
- lignes 19-21 : redéfinissent la méthode [configure(AuthenticationManagerBuilder auth)] qui gère les utilisateurs ;
- ligne 20 : l'autentification se fait avec des utilisateurs définis en " dur " [auth.inMemoryAuthentication()]. Un utilisateur est ici défini avec le login [user], le mot de passe [password] et le rôle [USER]. On peut accorder les mêmes droits à des utilisateurs ayant le même rôle ;

8.4.11.5 *Classe exécutable*



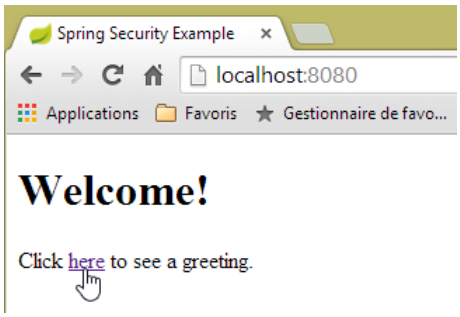
La classe [Application] est la suivante :

```
1.  package hello;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.boot.SpringApplication;
5.  import org.springframework.context.annotation.ComponentScan;
6.  import org.springframework.context.annotation.Configuration;
7.
8.  @EnableAutoConfiguration
9.  @Configuration
10. @ComponentScan
11. public class Application {
12.
13.     public static void main(String[] args) throws Throwable {
14.         SpringApplication.run(Application.class, args);
15.     }
16.
17. }
```

- ligne 8 : l'annotation [@EnableAutoConfiguration] demande à Spring Boot (ligne 3) de faire la configuration que le développeur n'aura pas fait explicitement ;
- ligne 9 : fait de la classe [Application] une classe de configuration Spring ;
- ligne 10 : demande le scan du dossier de la classe [Application] afin de rechercher des composants Spring. Les deux classes [MvcConfig] et [WebSecurityConfig] vont être ainsi découvertes car elles ont l'annotation [@Configuration] ;
- ligne 13 : la méthode [main] de la classe exécutable ;
- ligne 14 : la méthode statique [SpringApplication.run] est exécutée avec comme paramètre la classe de configuration [Application]. Nous avons déjà rencontré ce processus et nous savons que le serveur Tomcat embarqué dans les dépendances Maven du projet va être lancé et le projet déployé dessus. Nous avons vu que quatre URL étaient gérées [/, /home, /login, /hello] et que certaines étaient protégées par des droits d'accès.

8.4.11.6 *Tests de l'application*

Commençons par demander l'URL [/] qui est l'une des quatre URL acceptées. Elle est associée à la vue [/templates/home.html] :



L'URL demandée [/] est accessible à tous. C'est pourquoi nous l'avons obtenue. Le lien [here] est le suivant :

```
Click <a href="/hello">here</a> to see a greeting.
```

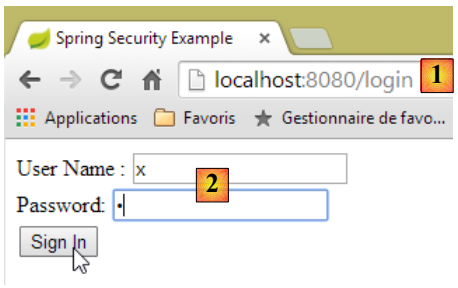
L'URL [/hello] va être demandée lorsqu'on va cliquer sur le lien. Celle-ci est protégée :

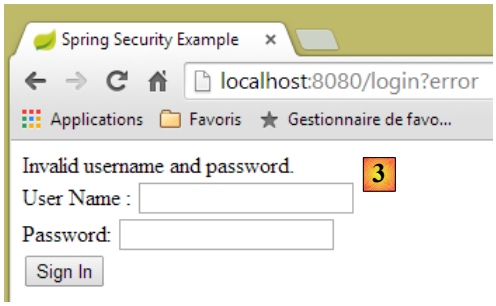
URL	règle	code
/, /home	accès sans être authentifié	`http.authorizeRequests().antMatchers("/", "/home").permitAll()`
autres URL	accès authentifié uniquement	`http.anyRequest().authenticated();`

Il faut être authentifié pour l'obtenir. Spring Security va alors rediriger le navigateur client vers la page d'authentification. D'après la configuration vue, c'est la page d'URL [/login]. Celle-ci est accessible à tous :

```
http.formLogin().loginPage("/login").permitAll().and().logout().permitAll();
```

Nous l'obtenons donc [1] :





Le code source de la page obtenue est le suivant :

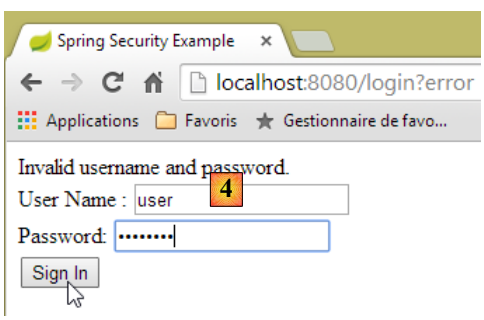
```
1.  <!DOCTYPE html>
2.
3.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-
    springsecurity3">
4.  ...
5.     <form method="post" action="/login">
6.  ...
7.        <input type="hidden" name="_csrf" value="87bea06a-a177-459d-b279-c6068a7ad3eb" />
8.     </form>
9.  </body>
10. </html>
```

- ligne 7, un champ caché apparaît qui n'est pas dans la page [login.html] d'origine. C'est Thymeleaf qui l'a ajouté. Ce code appelé CSRF (Cross Site Request Forgery) vise à éliminer une faille de sécurité. Ce jeton doit être renvoyé à Spring Security avec l'authentification pour que cette dernière soit acceptée ;

Nous nous souvenons que seul l'utilisateur user/password est reconnu par Spring Security. Si nous entrons autre chose en [2], nous obtenons la même page avec un message d'erreur en [3]. Spring Security a redirigé le navigateur vers l'URL [http://localhost:8080/login?error]. La présence du paramètre [error] a déclenché l'affichage de la balise :

```
<div th:if="${param.error}">Invalid username and password.</div>
```

Maintenant, entrons les valeurs attendues user/password [4] :



- en [4], nous nous identifions ;
- en [5], Spring Security nous redirige vers l'URL [/hello] car c'est l'URL que nous demandions lorsque nous avons été redirigés vers la page de login. L'identité de l'utilisateur a été affichée par la ligne suivante de [hello.html] :

```
<h1 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!</h1>
```

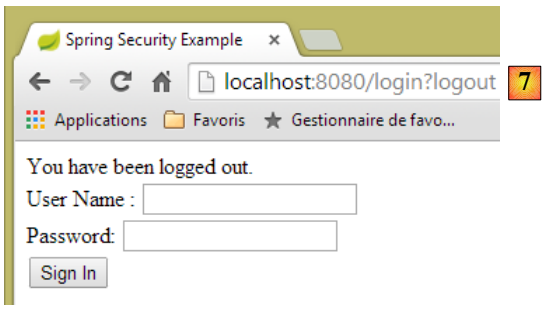
La page [5] affiche le formulaire suivant :

```
1.      <form th:action="@{/logout}" method="post">
2.          <input type="submit" value="Sign Out" />
3.  </form>
```

Lorsqu'on clique sur le bouton [Sign Out], un POST va être fait sur l'URL [/logout]. Celle-ci comme l'URL [/login] est accessible à tous :

```
http.formLogin().loginPage("/login").permitAll().and().logout().permitAll();
```

Dans notre association URL / vues, nous n'avons rien défini pour l'URL [/logout]. Que va-t-il se passer ? Essayons :



- en [6], nous cliquons sur le bouton [Sign Out] ;
- en [7], nous voyons que nous avons été redirigés vers l'URL [http://localhost:8080/login?logout]. C'est Spring Security qui a demandé cette redirection. La présence du paramètre [logout] dans l'URL a fait afficher la ligne suivante de la vue :

```
<div th:if="${param.logout}">You have been logged out.</div>
```

8.4.11.7 Conclusion

Dans l'exemple précédent, nous aurions pu écrire l'application web d'abord puis la sécuriser ensuite. Spring Security n'est pas intrusif. On peut mettre en place la sécurité d'une application web déjà écrite. Par ailleurs, nous avons découvert les points suivants :

- il est possible de définir une page d'authentification ;
- l'authentification doit être accompagnée du jeton CSRF délivré par Spring Security ;

- si l'authentification échoue, on est redirigé vers la page d'authentification avec de plus un paramètre **error** dans l'URL ;
- si l'authentification réussit, on est redirigé vers la page demandée lorsque l'autentification a eu lieu. Si on demande directement la page d'authentification sans passer par une page intermédiaire, alors Spring Security nous redirige vers l'URL [/] (ce cas n'a pas été présenté) ;
- on se déconnecte en demandant l'URL [/logout] avec un POST. Spring Security nous redirige alors vers la page d'authentification avec le paramètre **logout** dans l'URL ;

Toutes ces conclusions reposent sur des comportements par défaut de Spring Security. Ces comportements peuvent être changés par configuration en redéfinissant certaines méthodes de la classe [`WebSecurityConfigurerAdapter`].

Le tutoriel précédent nous aidera peu dans la suite. Nous allons en effet utiliser :

- une base de données pour stocker les utilisateurs, leurs mots de passe et leurs rôles ;
- une authentification par entête HTTP ;

On trouve assez peu de tutoriels pour ce qu'on veut faire ici. La solution qui va être proposée est un assemblage de codes trouvés ici et là.

8.4.12 Mise en place de la sécurité sur le service web de rendez-vous

8.4.12.1 La base de données

La base de données [rdvmedecins] évolue pour prendre en compte les utilisateurs, leurs mots de passe et leur rôles. Trois nouvelles tables apparaissent :

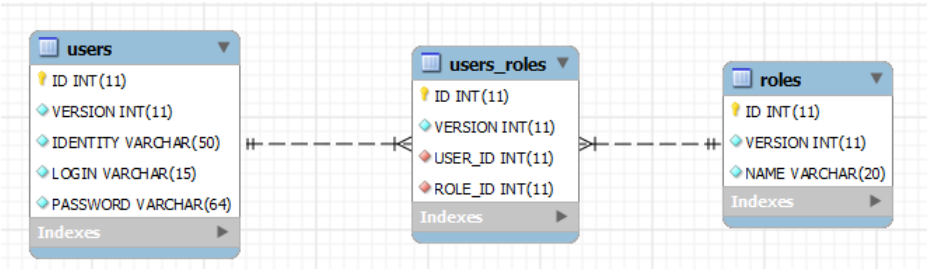


Table [USERS] : les utilisateurs

- ID : clé primaire ;
- VERSION : colonne de versioning de la ligne ;
- IDENTITY : une identité descriptive de l'utilisateur ;
- LOGIN : le login de l'utilisateur ;
- PASSWORD : son mot de passe ;

Dans la table USERS, les mots de passe ne sont pas stockés en clair :

ID	VERSION	IDENTITY	LOGIN	PASSWORD
7	0	guest	guest	$2a$10$Gzyp54mvkgMH0SPQkXo.Zeu.DwJ/QI50PRXLf2FkoIMTs7fr6A2J2
8	0	admin	admin	$2a$10$m79V6MKt9GPDdpjSulyqReqUioqYwXy8oIlt/.ia15FhX2fym3AE6
9	0	user	user	$2a$10$ph5y/1H89YC11oGVLB49fON.dZwnu44bAOKMK1FFI//xjAvsr/Ese

L'algorithme qui crypte les mots de passe est l'algorithme BCRYPT.

Table [ROLES] : les rôles

- ID : clé primaire ;
- VERSION : colonne de versioning de la ligne ;
- NAME : nom du rôle. Par défaut, Spring Security attend des noms de la forme ROLE_XX, par exemple ROLE_ADMIN ou ROLE_GUEST ;

ID	VERSION	NAME
1	1	ROLE_ADMIN
2	1	ROLE_USER

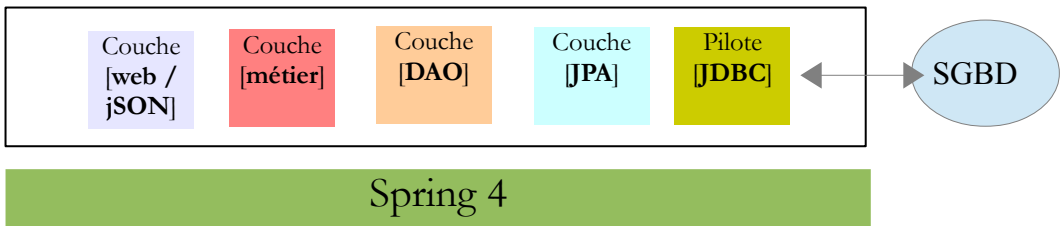
Table [USERS_ROLES] : table de jointure USERS / ROLES

Un utilisateur peut avoir plusieurs rôles, un rôle peut rassembler plusieurs utilisateurs. On a une relation **plusieurs à plusieurs** matérialisée par la table [USERS_ROLES].

- ID : clé primaire ;
- VERSION : colonne de versioning de la ligne ;
- USER_ID : identifiant d'un utilisateur ;
- ROLE_ID : identifiant d'un rôle ;

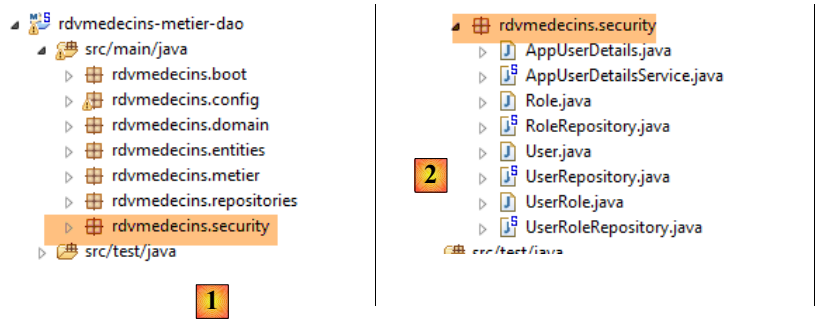
ID	VERSION	USER_ID	ROLE_ID
1	1	1	2
2	1	2	1

Parce que nous modifions la base de données, l'ensemble des couches du projet [métier, DAO, JPA] doit être modifié :



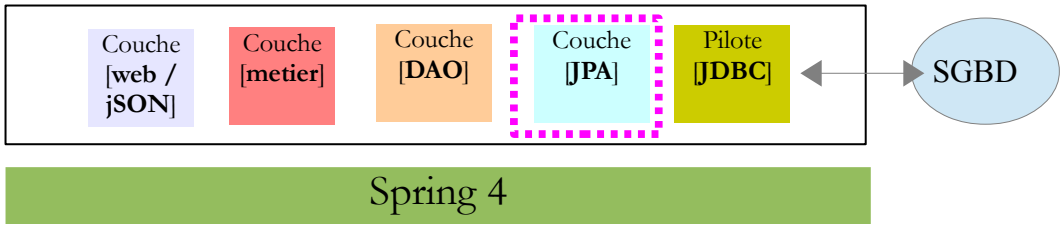
8.4.12.2 Le nouveau projet STS du [métier, DAO, JPA]

Le projet [rdvmedecins-metier-dao] évolue de la façon suivante :

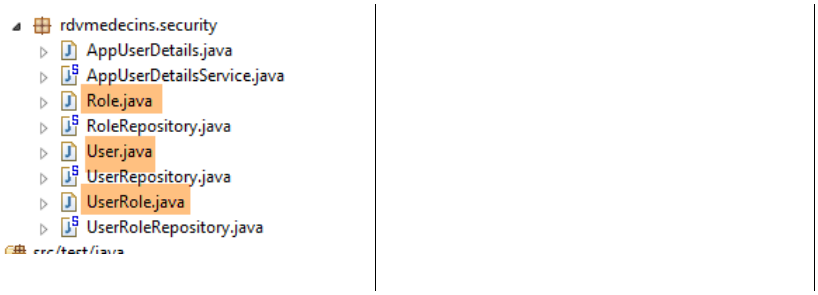


- en [1] : le nouveau projet ;
- en [2] : les modifications amenées par la prise en compte de la sécurité ont été rassemblées dans un unique paquetage [rdvmedecins.security]. Ces nouveaux éléments appartiennent aux couches [JPA] et [DAO] mais par simplicité elles ont été rassemblées dans un même paquetage.

8.4.12.3 Les nouvelles entités [JPA]



La couche JPA définit trois nouvelles entités :



La classe [User] est l'image de la table [USERS] :

```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Column;
4.  import javax.persistence.Entity;
5.  import javax.persistence.Table;
6.
7.  @Entity
8.  @Table(name = "USERS")
9.  public class User extends AbstractEntity {
10.    private static final long serialVersionUID = 1L;
11.
12.    // propriétés
13.    private String identity;
14.    private String login;
15.    private String password;
16.
17.    // constructeur
18.    public User() {
19.    }
20.
21.    public User(String identity, String login, String password) {
22.       this.identity = identity;
23.       this.login = login;
24.       this.password = password;
25.    }
26.
27.    // identité
28.    @Override
29.    public String toString() {
30.       return String.format("User[%s,%s,%s]", identity, login, password);
31.    }
32.
33.    // getters et setters
34. ....
35. }
```

- ligne 9 : la classe étend la classe [AbstractEntity] déjà utilisée pour les autres entités ;
- lignes 13-15 : on ne précise pas de nom pour les colonnes parce qu'elles portent le même nom que les champs qui leur sont associés ;

La classe [Role] est l'image de la table [ROLES] :

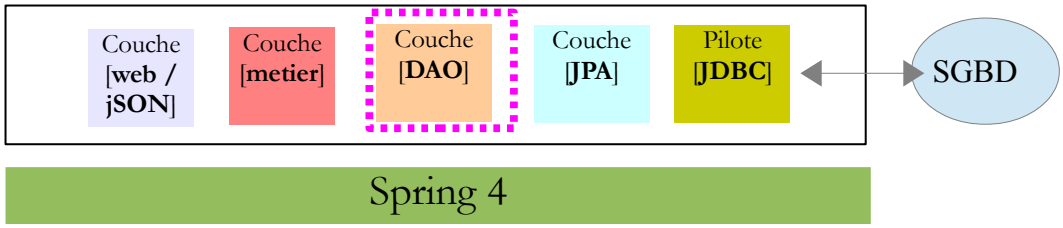
```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Column;
4.  import javax.persistence.Entity;
5.  import javax.persistence.Table;
6.
7.  @Entity
8.  @Table(name = "ROLES")
9.  public class Role extends AbstractEntity {
10.
11.     private static final long serialVersionUID = 1L;
12.
13.     // propriétés
14.     private String name;
15.
16.     // constructeurs
17.     public Role() {
18.     }
19.
20.     public Role(String name) {
21.        this.name = name;
22.     }
23.
24.     // identité
25.     @Override
26.     public String toString() {
27.        return String.format("Role[%s]", name);
28.     }
29.
30.     // getters et setters
31. ...
32. }
```

La classe [UserRole] est l'image de la table [USERS_ROLES] :

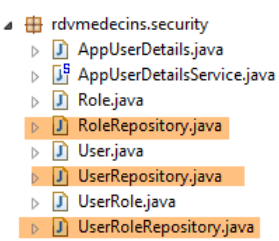
```
1.  package rdvmedecins.entities;
2.
3.  import javax.persistence.Entity;
4.  import javax.persistence.JoinColumn;
5.  import javax.persistence.ManyToOne;
6.  import javax.persistence.Table;
7.
8.  @Entity
9.  @Table(name = "USERS_ROLES")
10. public class UserRole extends AbstractEntity {
11.
12.     private static final long serialVersionUID = 1L;
13.
14.     // un UserRole référence un User
15.     @ManyToOne
16.     @JoinColumn(name = "USER_ID")
17.     private User user;
18.     // un UserRole référence un Role
19.     @ManyToOne
20.     @JoinColumn(name = "ROLE_ID")
21.     private Role role;
22.
23.     // getters et setters
24. ...
25. }
```

- lignes 15-17 : matérialisent la clé étrangère de la table [USERS_ROLES] vers la table [USERS] ;
- lignes 19-21 : matérialisent la clé étrangère de la table [USERS_ROLES] vers la table [ROLES] ;

8.4.12.4 Modifications de la couche [DAO]



La couche [DAO] s'enrichit de trois nouveaux [Repository] :



L'interface [UserRepository] gère les accès aux entités [User] :

```
1.  package rdvmedecins.repositories;
2.
3.  import org.springframework.data.jpa.repository.Query;
4.  import org.springframework.data.repository.CrudRepository;
5.
6.  import rdvmedecins.entities.Role;
7.  import rdvmedecins.entities.User;
8.
9.  public interface UserRepository extends CrudRepository<User, Long> {
10.
11.     // liste des rôles d'un utilisateur identifié par son id
12.     @Query("select ur.role from UserRole ur where ur.user.id=?1")
13.     Iterable<Role> getRoles(long id);
14.
15.     // liste des rôles d'un utilisateur identifié par son login et son mot de passe
16.     @Query("select ur.role from UserRole ur where ur.user.login=?1 and ur.user.password=?2")
17.     Iterable<Role> getRoles(String login, String password);
18.
19.     // recherche d'un utilisateur via son login
20.     User findUserByLogin(String login);
21. }
```

- ligne 9 : l'interface [UserRepository] étend la l'interface [CrudRepository] de Spring Data (ligne 4) ;
- lignes 12-13 : la méthode [getRoles(User user)] permet d'avoir tous les rôles d'un utilisateur identifié par son [id]
- lignes 16-17 : idem mais pour un utilisateur identifié pas ses login / mot de passe ;
- ligne 20 : pour trouver un utilisateur via son login ;

L'interface [RoleRepository] gère les accès aux entités [Role] :

```
1.  package rdvmedecins.security;
2.
3.  import org.springframework.data.repository.CrudRepository;
4.
5.  public interface RoleRepository extends CrudRepository<Role, Long> {
6.
7.      // recherche d'un rôle via son nom
8.      Role findRoleByName(String name);
9.
10. }
```

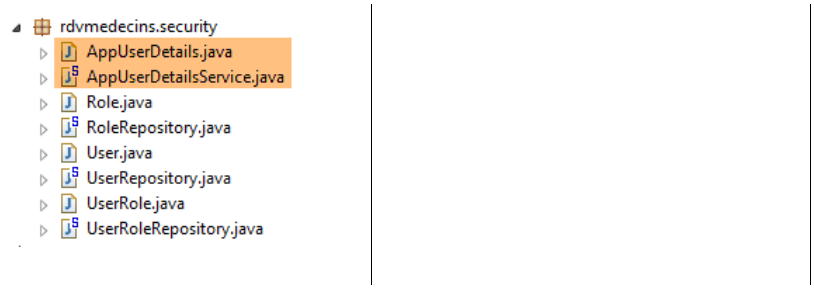
- ligne 5 : l'interface [RoleRepository] étend l'interface [CrudRepository] ;
- ligne 8 : on peut chercher un rôle via son nom ;

L'interface [userRoleRepository] gère les accès aux entités [UserRole] :

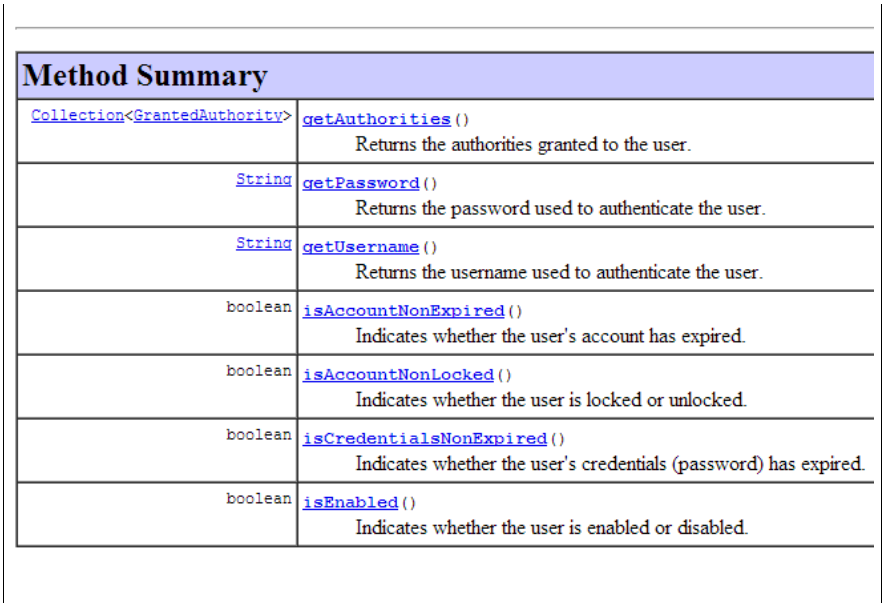
```
1.  package rdvmedecins.security;
2.
3.  import org.springframework.data.repository.CrudRepository;
4.
5.  public interface UserRoleRepository extends CrudRepository<UserRole, Long> {
6.
7.  }
```

- ligne 5 : l'interface [UserRoleRepository] se contente d'étendre l'interface [CrudRepository] sans lui ajouter de nouvelles méthodes ;

8.4.12.5 *Les classes de gestion des utilisateurs et des rôles*



Spring Security impose la création d'une classe implémentant l'interface [UsersDetail] suivante :



Method Summary

Collection<GrantedAuthority>	getAuthorities() Returns the authorities granted to the user.
String	getPassword() Returns the password used to authenticate the user.
String	getUsername() Returns the username used to authenticate the user.
boolean	isAccountNonExpired() Indicates whether the user's account has expired.
boolean	isAccountNonLocked() Indicates whether the user is locked or unlocked.
boolean	isCredentialsNonExpired() Indicates whether the user's credentials (password) has expired.
boolean	isEnabled() Indicates whether the user is enabled or disabled.

Cette interface est ici implémentée par la classe [AppUserDetails] :

```
1.  package rdvmedecins.security;
2.
3.  import java.util.ArrayList;
4.  import java.util.Collection;
5.
6.  import org.springframework.security.core.GrantedAuthority;
7.  import org.springframework.security.core.authority.SimpleGrantedAuthority;
8.  import org.springframework.security.core.userdetails.UserDetails;
9.
10. public class AppUserDetails implements UserDetails {
11.
12.     private static final long serialVersionUID = 1L;
```

```
13.
14.     // propriétés
15.     private User user;
16.     private UserRepository userRepository;
17.
18.     // constructeurs
19.     public AppUserDetails() {
20.     }
21.
22.     public AppUserDetails(User user, UserRepository userRepository) {
23.        this.user = user;
24.        this.userRepository = userRepository;
25.     }
26.
27.     // -------------------------interface
28.     @Override
29.     public Collection<? extends GrantedAuthority> getAuthorities() {
30.        Collection<GrantedAuthority> authorities = new ArrayList<>();
31.        for (Role role : userRepository.getRoles(user.getId())) {
32.           authorities.add(new SimpleGrantedAuthority(role.getName()));
33.        }
34.        return authorities;
35.     }
36.
37.     @Override
38.     public String getPassword() {
39.        return user.getPassword();
40.     }
41.
42.     @Override
43.     public String getUsername() {
44.        return user.getLogin();
45.     }
46.
47.     @Override
48.     public boolean isAccountNonExpired() {
49.        return true;
50.     }
51.
52.     @Override
53.     public boolean isAccountNonLocked() {
54.        return true;
55.     }
56.
57.     @Override
58.     public boolean isCredentialsNonExpired() {
59.        return true;
60.     }
61.
62.     @Override
63.     public boolean isEnabled() {
64.        return true;
65.     }
66.
67.     // getters et setters
68.     ...
69. }
```

- ligne 10 : la classe [AppUserDetails] implémente l'interface [UserDetails] ;
- lignes 15-16 : la classe encapsule un utilisateur (ligne 15) et le repository qui permet d'avoir les détails de cet utilisateur (ligne 16) ;
- lignes 22-25 : le constructeur qui instancie la classe avec un utilisateur et son repository ;
- lignes 28-35 : implémentation de la méthode [getAuthorities] de l'interface [UserDetails]. Elle doit construire une collection d'éléments de type [GrantedAuthority] ou dérivé. Ici, nous utilisons le type dérivé [SimpleGrantedAuthority] (ligne 32) qui encapsule le nom d'un des rôles de l'utilisateur de la ligne 15 ;
- lignes 31-33 : on parcourt la liste des rôles de l'utilisateur de la ligne 15 pour construire une liste d'éléments de type [SimpleGrantedAuthority] ;
- lignes 38-40 : implémentent la méthode [getPassword] de l'interface [UserDetails]. On rend le mot de passe de l'utilisateur de la ligne 15 ;
- lignes 38-40 : implémentent la méthode [getUserName] de l'interface [UserDetails]. On rend le login de l'utilisateur de la ligne 15 ;
- lignes 47-50 : le compte de l'utilisateur n'expire jamais ;

- lignes 52-55 : le compte de l'utilisateur n'est jamais bloqué ;
- lignes 57-60 : les identifiants de l'utilisateur n'expirent jamais ;
- lignes 62-65 : le compte de l'utilisateur est toujours actif ;

Spring Security impose également l'existence d'une classe implémentant l'interface [`AppUserDetailsService`] :

Method Summary

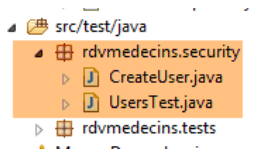
UserDetails	loadUserByUsername(String username) Locates the user based on the username.

Cette interface est implémentée par la classe [AppUserDetailsService] suivante :

```
1.  package rdvmedecins.security;
2.
3.  import org.springframework.beans.factory.annotation.Autowired;
4.  import org.springframework.security.core.userdetails.UserDetails;
5.  import org.springframework.security.core.userdetails.UserDetailsService;
6.  import org.springframework.security.core.userdetails.UsernameNotFoundException;
7.  import org.springframework.stereotype.Service;
8.
9.  @Service
10. public class AppUserDetailsService implements UserDetailsService {
11.
12.    @Autowired
13.    private UserRepository userRepository;
14.
15.    @Override
16.    public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {
17.       // on cherche l'utilisateur via son login
18.       User user = userRepository.findUserByLogin(login);
19.       // trouvé ?
20.       if (user == null) {
21.          throw new UsernameNotFoundException(String.format("login [%s] inexistant", login));
22.       }
23.       // on rend les détails de l'utilsateur
24.       return new AppUserDetails(user, userRepository);
25.    }
26.
27. }
```

- ligne 9 : la classe sera un composant Spring, donc disponible dans son contexte ;
- lignes 12-13 : le composant [UserRepository] sera injecté ici ;
- lignes 16-25 : implémentation de la méthode [loadUserByUsername] de l'interface [UserDetailsService] (ligne 10). Le paramètre est le login de l'utilisateur ;
- ligne 18 : l'utilisateur est recherché via son login ;
- lignes 20-22 : s'il n'est pas trouvé, une exception est lancée ;
- ligne 24 : un objet [AppUserDetails] est construit et rendu. Il est bien de type [UserDetails] (ligne 16) ;

8.4.12.6 Tests de la couche [DAO]



Tout d'abord, nous créons une classe exécutable [CreateUser] capable de créer un utilisateur avec un rôle :

```
1.  package rdvmedecins.security;
2.
```

```
3.  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
4.  import org.springframework.security.crypto.bcrypt.BCrypt;
5.
6.  import rdvmedecins.config.DomainAndPersistenceConfig;
7.  import rdvmedecins.security.Role;
8.  import rdvmedecins.security.RoleRepository;
9.  import rdvmedecins.security.User;
10. import rdvmedecins.security.UserRepository;
11. import rdvmedecins.security.UserRole;
12. import rdvmedecins.security.UserRoleRepository;
13.
14. public class CreateUser {
15.
16.    public static void main(String[] args) {
17.       // syntaxe : login password roleName
18.
19.       // il faut trois paramètres
20.       if (args.length != 3) {
21.          System.out.println("Syntaxe : [pg] user password role");
22.          System.exit(0);
23.       }
24.       // on récupère les paramètres
25.       String login = args[0];
26.       String password = args[1];
27.       String roleName = String.format("ROLE_%s", args[2].toUpperCase());
28.       // contexte Spring
29.       AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(DomainAndPersistenceConfig.class);
30.       UserRepository userRepository = context.getBean(UserRepository.class);
31.       RoleRepository roleRepository = context.getBean(RoleRepository.class);
32.       UserRoleRepository userRoleRepository = context.getBean(UserRoleRepository.class);
33.       // le rôle existe-t-il déjà ?
34.       Role role = roleRepository.findRoleByName(roleName);
35.       // s'il n'existe pas on le crée
36.       if (role == null) {
37.          role = roleRepository.save(new Role(roleName));
38.       }
39.       // l'utilisateur existe-t-il déjà ?
40.       User user = userRepository.findUserByLogin(login);
41.       // s'il n'existe pas on le crée
42.       if (user == null) {
43.          // on hashe le mot de passe avec bcrypt
44.          String crypt = BCrypt.hashpw(password, BCrypt.gensalt());
45.          // on sauvegarde l'utilisateur
46.          user = userRepository.save(new User(login, login, crypt));
47.          // on crée la relation avec le rôle
48.          userRoleRepository.save(new UserRole(user, role));
49.       } else {
50.          // l'utilisateur existe déjà- a-t-il le rôle demandé ?
51.          boolean trouvé = false;
52.          for (Role r : userRepository.getRoles(user.getId())) {
53.             if (r.getName().equals(roleName)) {
54.                trouvé = true;
55.                break;
56.             }
57.          }
58.          // si pas trouvé, on crée la relation avec le rôle
59.          if (!trouvé) {
60.             userRoleRepository.save(new UserRole(user, role));
61.          }
62.       }
63.
64.       // fermeture contexte Spring
65.       context.close();
66.    }
67.
68. }
```

- ligne 17 : la classe attend trois arguments définissant un utilisateur : son login, son mot de passe, son rôle ;
- lignes 25-27 : les trois paramètres sont récupérés ;
- ligne 29 : le contexte Spring est construit à partir de la classe de configuration [DomainAndPersistenceConfig]. Cette classe existait déjà dans le projet initial. Elle doit évoluer de la façon suivante :

```
1.  @EnableJpaRepositories(basePackages = { "rdvmedecins.repositories", "rdvmedecins.security" })
2.  @EnableAutoConfiguration
3.  @ComponentScan(basePackages = { "rdvmedecins" })
4.  @EntityScan(basePackages = { "rdvmedecins.entities", "rdvmedecins.security" })
5.  @EnableTransactionManagement
6.  public class DomainAndPersistenceConfig {
7.  ....
8.  }
```

- ligne 1 : il faut indiquer qu'il y a maintenant des composants [Repository] dans le paquetage [rdvmedecins.security] ;
- ligne 4 : il faut indiquer qu'il y a maintenant des entités JPA dans le paquetage [rdvmedecins.security] ;

Revenons au code de création d'un utilisateur :

- lignes 30-32 : on récupère les références des trois [Repository] qui peuvent nous être utiles pour créer l'utilisateur ;
- ligne 34 : on regarde si le rôle existe déjà ;
- lignes 36-38 : si ce n'est pas le cas, on le crée en base. Il aura un nom du type [ROLE_XX] ;
- ligne 40 : on regarde si le login existe déjà ;
- lignes 42-49 : si le login n'existe pas, on le crée en base ;
- ligne 44 : on crypte le mot de passe. On utilise ici, la classe [BCrypt] de Spring Security (ligne 4). On a donc besoin des archives de ce framework. Le fichier [pom.xml] inclut une nouvelle dépendance :

```
1.        <dependency>
2.           <groupId>org.springframework.boot</groupId>
3.           <artifactId>spring-boot-starter-security</artifactId>
4.  </dependency>
```

- ligne 46 : l'utilisateur est persisté en base ;
- ligne 48 : ainsi que la relation qui le lie à son rôle ;
- lignes 51-57 : cas où le login existe déjà – on regarde alors si parmi ses rôles se trouve déjà le rôle qu'on veut lui attribuer ;
- ligne 59-61 : si le rôle cherché n'a pas été trouvé, on crée une ligne dans la table [USERS_ROLES] pour relier l'utilisateur à son rôle ;
- on ne s'est pas protégé des exceptions éventuelles. C'est une classe de soutien pour créer rapidement un utilisateur avec un rôle.

Lorsqu'on exécute la classe avec les arguments [x x guest], on obtient en base les résultats suivants :

Table [USERS]

+ Options

	ID	VERSION	IDENTITY	LOGIN	PASSWORD
Modifier Copier Effacer	14	0	admin	admin	$2a$10$FN1LMKjPU46aPffh9Zaw4exJOLo51JJPWrxqzak/eJrbt3CO9WzVG
Modifier Copier Effacer	15	0	user	user	$2a$10$SJehR9Mv2VdyRZo9F0rXa.hKAoGLhJg6kSdyfExi40mEJrNOj0BTq
Modifier Copier Effacer	16	0	guest	guest	$2a$10$ubyWJb/vg2XZnUOAUjspZuz9jpHP3fIbPTbwQU115EtLdeSZ2PB7q
Modifier Copier Effacer	17	0	x	x	$2a$10$kEXA56wpKHFReVqwQTyWguKguK8I4uhA2zb6t3wGxag8Dyv7AhLom

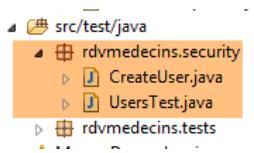
Table [ROLES]

ID	VERSION	NAME
5	0	ROLE_GUEST
6	0	ROLE_ADMIN
7	0	ROLE_USER

Table [USERS_ROLES]

ID	VERSION	USER_ID	ROLE_ID
11	0	14	6
12	0	15	7
13	0	16	5
14	0	17	5

Considérons maintenant la seconde classe [UsersTest] qui est un test JUnit :




```
1.  package rdvmedecins.security;
2.
3.  import java.util.List;
4.
5.  import org.junit.Assert;
6.  import org.junit.Test;
7.  import org.junit.runner.RunWith;
8.  import org.springframework.beans.factory.annotation.Autowired;
9.  import org.springframework.boot.test.SpringApplicationConfiguration;
10. import org.springframework.security.core.authority.SimpleGrantedAuthority;
11. import org.springframework.security.crypto.bcrypt.BCrypt;
12. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
13.
14. import rdvmedecins.config.DomainAndPersistenceConfig;
15.
16. import com.google.common.collect.Lists;
17.
18. @SpringApplicationConfiguration(classes = DomainAndPersistenceConfig.class)
19. @RunWith(SpringJUnit4ClassRunner.class)
20. public class UsersTest {
21.
22.    @Autowired
23.    private UserRepository userRepository;
24.    @Autowired
25.    private AppUserDetailsService appUserDetailsService;
26.
27.    @Test
28.    public void findAllUsersWithTheirRoles() {
29.       Iterable<User> users = userRepository.findAll();
30.       for (User user : users) {
31.          System.out.println(user);
32.          display("Roles :", userRepository.getRoles(user.getId()));
33.       }
34.    }
35.
36.    @Test
37.    public void findUserByLogin() {
38.       // on récupère l'utilisateur [admin]
39.       User user = userRepository.findUserByLogin("admin");
40.       // on vérifie que son mot de passe est [admin]
41.       Assert.assertTrue(BCrypt.checkpw("admin", user.getPassword()));
42.       // on vérifie le rôle de admin / admin
43.       List<Role> roles = Lists.newArrayList(userRepository.getRoles("admin", user.getPassword()));
44.       Assert.assertEquals(1L, roles.size());
45.       Assert.assertEquals("ROLE_ADMIN", roles.get(0).getName());
46.    }
47.
48.    @Test
49.    public void loadUserByUsername() {
50.       // on récupère l'utilisateur [admin]
51.       AppUserDetails userDetails = (AppUserDetails) appUserDetailsService.loadUserByUsername("admin");
52.       // on vérifie que son mot de passe est [admin]
```

```
53.         Assert.assertTrue(BCrypt.checkpw("admin", userDetails.getPassword()));
54.         // on vérifie le rôle de admin / admin
55.         @SuppressWarnings("unchecked")
56.         List<SimpleGrantedAuthority> authorities = (List<SimpleGrantedAuthority>)
    userDetails.getAuthorities();
57.         Assert.assertEquals(1L, authorities.size());
58.         Assert.assertEquals("ROLE_ADMIN", authorities.get(0).getAuthority());
59.     }
60.
61.     // méthode utilitaire - affiche les éléments d'une collection
62.     private void display(String message, Iterable<?> elements) {
63.         System.out.println(message);
64.         for (Object element : elements) {
65.             System.out.println(element);
66.         }
67.     }
68. }
```

- lignes 27-34 : test visuel. On affiche tous les utilisateurs avec leurs rôles ;
- lignes 36-46 : on vérifie que l'utilisateur [admin] a le mot de passe [admin] et le rôle [ROLE_ADMIN] en utilisant le repository [UserRepository] ;
- ligne 41 : [admin] est le mot de passe en clair. En base, il est crypté selon l'algorithme BCrypt. La méthode [BCrypt.checkpw] permet de vérifier que le mot de passe en clair une fois crypté est bien égal à celui qui est en base ;
- lignes 48-59 : on vérifie que l'utilisateur [admin] a le mot de passe [admin] et le rôle [ROLE_ADMIN] en utilisant le service [appUserDetailsService] ;

L'exécution des tests réussit avec les logs suivants :

```
1.  User[admin,admin,$2a$10$FN1LMKjPU46aPffh9Zaw4exJOLo51JJPWrxqzak/eJrbt3CO9WzVG]
2.  Roles :
3.  Role[ROLE_ADMIN]
4.  User[user,user,$2a$10$SJehR9Mv2VdyRZo9F0rXa.hKAoGLhJg6kSdyfExi40mEJrNOj0BTq]
5.  Roles :
6.  Role[ROLE_USER]
7.  User[guest,guest,$2a$10$ubyWJb/vg2XZnUOAUjspZuz9jpHP3fIbPTbwQU115EtLdeSZ2PB7q]
8.  Roles :
9.  Role[ROLE_GUEST]
10. User[x,x,$2a$10$kEXA56wpKHFReVqwQTyWguKguK8I4uhA2zb6t3wGxag8Dyv7AhLom]
11. Roles :
12. Role[ROLE_GUEST]
```

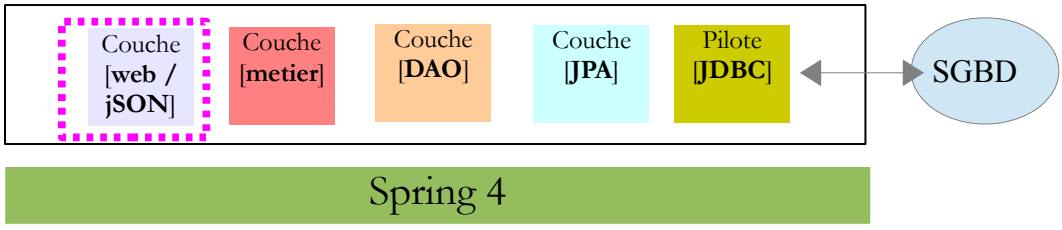
8.4.12.7 Conclusion intermédiaire

L'ajout des classes nécessaires à Spring Security a pu se faire avec peu de modifications du projet originel. Rappelons-les :

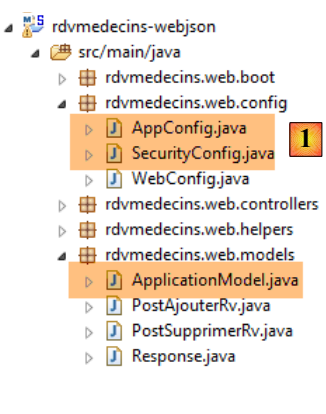
- ajout d'une dépendance sur Spring Security dans le fichier [pom.xml] ;
- création de trois tables supplémentaires dans la base de données ;
- création d'entités JPA et de composants Spring dans le package [rdvmedecins.security] ;

Ce cas très favorable découle du fait que les trois tables ajoutées dans la base de données sont indépendantes des tables existantes. On aurait même pu les mettre dans une base de données séparée. Ceci a été possible parce qu'on a décidé qu'un utilisateur avait une existence indépendante des médecins et des clients. Si ces derniers avaient été des utilisateurs potentiels, il aurait fallu créer des liens entre la table [USERS] et les tables [MEDECINS] et [CLIENTS]. Cela aurait eu alors un impact important sur le projet existant.

8.4.12.8 Le projet STS de la couche [web]



Le projet [rdvmedecins-webjson] évolue de la façon suivante[1] :



Les principales modifications sont à faire dans le package [rdvmedecins.web.config] où il faut configurer Spring Security. Il y en a d'autres, mineures, dans les classes [AppConfig] et [ApplicationModel]. Nous avons déjà rencontré une classe de configuration de Spring Security :

```
1.  package hello;
2.
3.  import org.springframework.context.annotation.Configuration;
4.  import
    org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
5.  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6.  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
7.  import org.springframework.security.config.annotation.web.servlet.configuration.EnableWebMvcSecurity;
8.
9.  @Configuration
10. @EnableWebMvcSecurity
11. public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
12.    @Override
13.    protected void configure(HttpSecurity http) throws Exception {
14.       http.authorizeRequests().antMatchers("/", "/home").permitAll().anyRequest().authenticated();
15.       http.formLogin().loginPage("/login").permitAll().and().logout().permitAll();
16.    }
17.
18.    @Override
19.    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
20.       auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");
21.    }
22. }
```

Nous allons suivre la même démarche :

- ligne 11 : définir une classe qui étend la classe [WebSecurityConfigurerAdapter] ;
- ligne 13 : définir une méthode [configure(HttpSecurity http)] qui définit les droits d'accès aux différentes URL du service web ;
- ligne 19 : définir une méthode [configure(AuthenticationManagerBuilder auth)] qui définit les utilisateurs et leurs rôles ;

La configuration de Spring Security est assurée par la classe [`SecurityConfig`] :

```
1.  package rdvmedecins.web.config;
2.
3.  import org.springframework.beans.factory.annotation.Autowired;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.  import org.springframework.http.HttpMethod;
6.  import
    org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
7.  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
8.  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
9.  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
10. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11.
12. import rdvmedecins.security.AppUserDetailsService;
```

```
13. import rdvmedecins.web.models.ApplicationModel;
14.
15. @EnableAutoConfiguration
16. @EnableWebSecurity
17. public class SecurityConfig extends WebSecurityConfigurerAdapter {
18.     @Autowired
19.     private AppUserDetailsService appUserDetailsService;
20.     @Autowired
21.     private ApplicationModel application;
22.
23.     @Override
24.     protected void configure(AuthenticationManagerBuilder registry) throws Exception {
25.         // l'authentification est faite par le bean [appUserDetailsService]
26.         // le mot de passe est crypté par l'algorithme de hachage BCrypt
27.         registry.userDetailsService(appUserDetailsService).passwordEncoder(new BCryptPasswordEncoder());
28.     }
29.
30.     @Override
31.     protected void configure(HttpSecurity http) throws Exception {
32.         // CSRF
33.         http.csrf().disable();
34.         // application sécurisée ?
35.         if (application.isSecured()) {
36.             // le mot de passe est transmis par le header Authorization: Basic xxxx
37.             http.httpBasic();
38.             // seul le rôle ADMIN peut utiliser l'application
39.             http.authorizeRequests() //
40.                     .antMatchers("/", "/**") // toutes les URL
41.                     .hasRole("ADMIN");
42.         }
43.     }
44. }
```

- lignes 15-16 : on a repris les annotations de l'exemple ;
- lignes 18-19 : la classe [AppUserDetails] qui donne accès aux utilisateurs de l'application est injectée ;
- lignes 20-21 : la classe [ApplicationModel] qui sert de cache à l'application web est injectée. On décide ici de l'utiliser également, pour configurer l'application web en un unique endroit. C'est elle qui définit le booléen [isSecured] de la ligne 35. Ce booléen sécurise (true) ou non (false) l'application web ;
- lignes 24-28 : la méthode [configure(HttpSecurity http)] définit les utilisateurs et leurs rôles. Elle reçoit en paramètre un type [AuthenticationManagerBuilder]. Ce paramètre est enrichi de deux informations (ligne 27) :
  - une référence sur le service [appUserDetailsService] de la ligne 19 qui donne accès aux utilisateurs enregistrés. On notera ici que le fait qu'ils soient enregistrés dans une base de données n'apparaît pas. Ils pourraient donc être dans un cache, délivrés par un service web, ...
  - le type de cryptage utilisé pour le mot de passe. On rappelle ici que nous avons utilisé l'algorithme BCrypt ;
- lignes 37-41 : la méthode [configure(HttpSecurity http)] définit les droits d'accès aux URL du service web ;
- ligne 33 : nous avons vu dans le projet d'introduction que par défaut Spring Security gérait un jeton CSRF (Cross Site Request Forgery) que l'utilisateur qui voulait s'authentifier devait renvoyer au serveur. Ici ce mécanisme est désactivé. Ceci allié au booléen (isSecured=false) permet d'utiliser l'application web sans sécurité ;
- ligne 37 : on active le mode d'authentification par entête HTTP. Le client devra envoyer l'entête HTTP suivant :

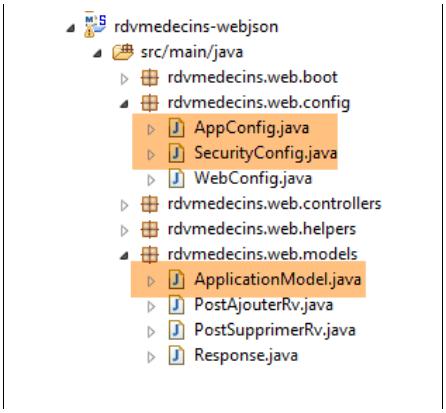
```
Authorization:Basic code
```

où code est le codage de la chaîne **login:password** par l'algorithme Base64. Par exemple, le codage Base64 de la chaîne *admin:admin* est *YWRtaW46YWRtaW4=*. Donc l'utilisateur de login [admin] et de mot de passe [admin] enverra l'entête HTTP suivant pour s'authentifier :

```
Authorization:Basic YWRtaW46YWRtaW4=
```

- lignes 39-41 : indiquent que toutes les URL du service web sont accessibles aux utilisateurs ayant le rôle [ROLE_ADMIN]. Cela veut dire qu'un utilisateur n'ayant pas ce rôle ne peut accéder au service web ;

La classe [AppConfig] qui configure l'ensemble de l'application évolue comme suit :




```
1.  package rdvmedecins.web.config;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.context.annotation.ComponentScan;
5.  import org.springframework.context.annotation.Import;
6.
7.  import rdvmedecins.config.DomainAndPersistenceConfig;
8.
9.  @EnableAutoConfiguration
10. @ComponentScan(basePackages = { "rdvmedecins.web" })
11. @Import({ DomainAndPersistenceConfig.class, SecurityConfig.class })
12. public class AppConfig {
13.
14. }
```

- la modification a lieu ligne 11 : on indique qu'il y a maintenant deux fichiers de configuration à exploiter [DomainAndPersistenceConfig] et [SecurityConfig].

Enfin la classe [ApplicationModel] s'enrichit d'un booléen :

```
1.  @Component
2.  public class ApplicationModel implements IMetier {
3.
4.  ...
5.      // données de configuration
6.      private boolean secured = false;
7.
8.      public boolean isSecured() {
9.          return secured;
10. }
```

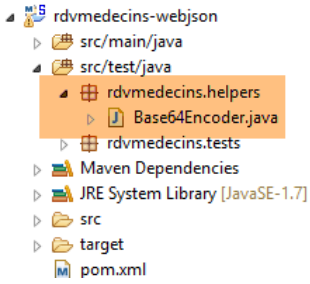
- ligne 6 : on positionne le booléen [secured] à [true / false] selon qu'on veut ou non activer la sécurisation.

8.4.12.9 Tests du service web

Nous allons tester le service web avec le client Chrome [Advanced Rest Client]. Nous allons avoir besoin de préciser l'entête HTTP d'authentification :

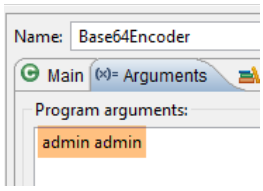
```
Authorization:Basic code
```

où [code] est le code Base64 de la chaîne [login:password]. Pour générer ce code, on peut utiliser le programme suivant :




```
1.  package rdvmedecins.helpers;
2.
3.  import org.springframework.security.crypto.codec.Base64;
4.
5.  public class Base64Encoder {
6.
7.     public static void main(String[] args) {
8.        // on attend deux arguments : login password
9.        if (args.length != 2) {
10.          System.out.println("Syntaxe : login password");
11.          System.exit(0);
12.       }
13.       // on récupère les deux arguments
14.       String chaîne = String.format("%s:%s", args[0], args[1]);
15.       // on encode la chaîne
16.       byte[] data = Base64.encode(chaîne.getBytes());
17.       // on affiche son encodage Base64
18.       System.out.println(new String(data));
19.    }
20.
21. }
```

Si nous exécutons ce programme avec les deux arguments [admin admin] :



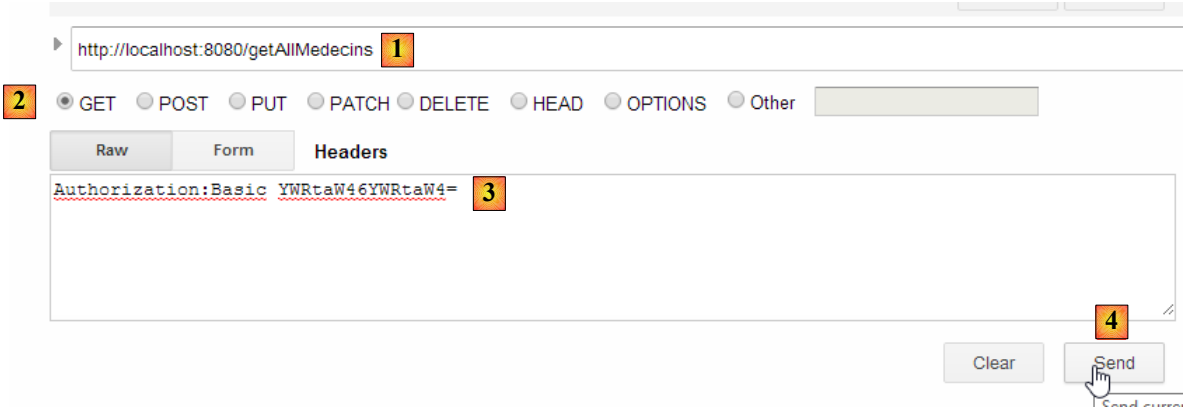
nous obtenons le résultat suivant :

```
YWRtaW46YWRtaW4=
```

Maintenant que nous savons générer l'entête HTTP d'authentification, nous lançons le service web maintenant sécurisé :

```
1.  @Component
2.  public class ApplicationModel implements IMetier {
3.  ...
4.  private boolean secured = true;
```

Puis avec le client Chrome [Advanced Rest Client], nous demandons la liste des tous les médecins :

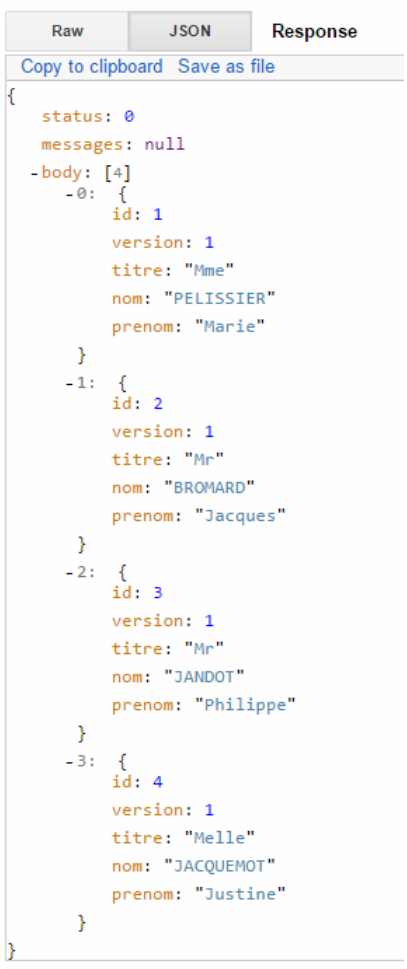


- en [1], nous demandons l'URL des médecins ;
- en [2], avec une méthode GET ;
- en [3], nous donnons l'entête HTTP de l'authentification. Le code [YWRtaW46YWRtaW4=] est le codage Base64 de la chaîne [admin:admin] ;
- en [4], nous envoyons la commande HTTP ;

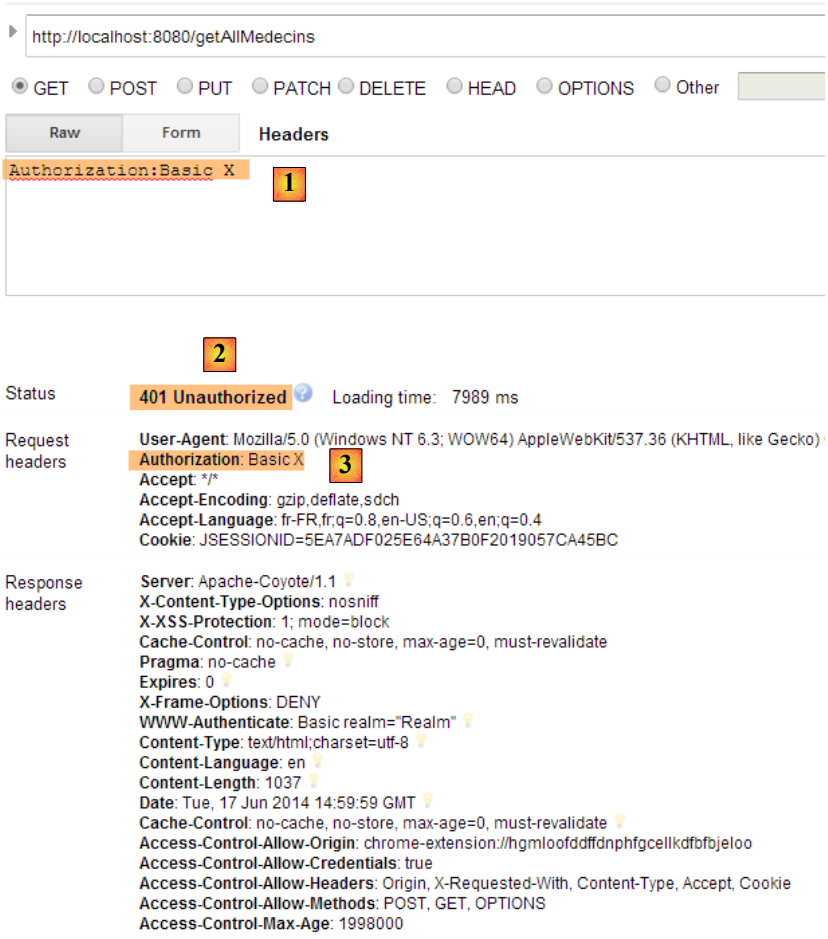
La réponse du serveur est la suivante :

- en [1], l'entête HTTP d'authentification ;
- en [2], le serveur renvoie une réponse jSON ;
- en [3], une liste d'entêtes HTTP liés à la sécurisation de l'application web ;

On obtient bien la liste des médecins :

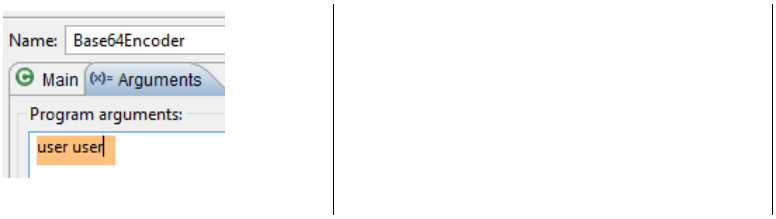


Tentons maintenant une requête HTTP avec un entête d'authentification incorrect. La réponse est alors la suivante :



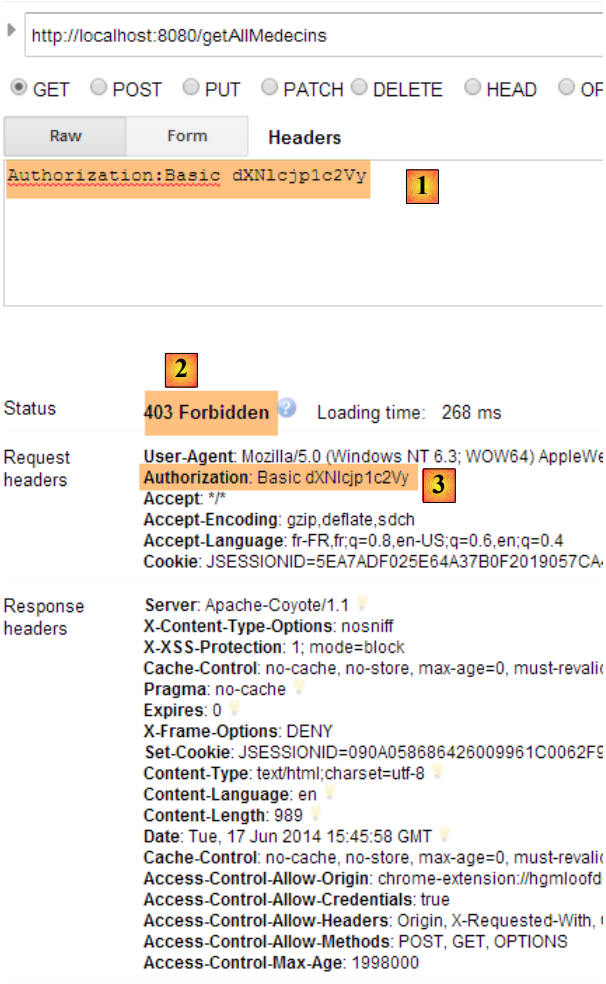
- en [1] et [3] : l'entête HTTP d'authentification ;
- en [2] : la réponse du service web ;

Maintenant, essayons l'utilisateur user / user. Il existe mais n'a pas accès au service web. Si nous exécutons le programme d'encodage Base64 avec les deux arguments [user user] :



nous obtenons le résultat suivant :

```
dXNlcjp1c2Vy
```

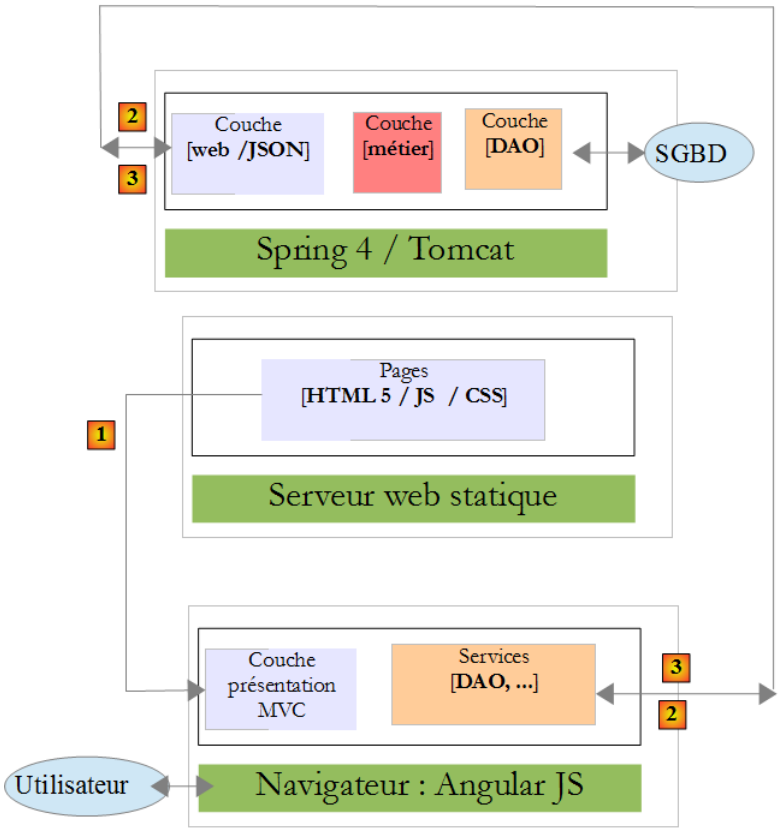


- en [1] et [3] : l'entête HTTP d'authentification ;
- en [2] : la réponse du service web. Elle est différente de la précédente qui était [401 Unauthorized]. Cette fois-ci, l'utilisateur s'est authentifié correctement mais n'a pas les droits suffisants pour accéder à l'URL ;

Un service web sécurisé est maintenant opérationnel. Nous allons le compléter pour qu'il autorise des requêtes inter-domaines. Ce besoin est apparu dans le document [Tutoriel AngularJS / Spring 4] et bien que ce besoin n'existe pas ici, nous allons quand même y répondre.

8.4.13 Mise en place des requêtes inter-domaines

Examinons le problème des requêtes inter-domaines. Dans le document [Tutoriel AngularJS / Spring 4], on développe une application client / serveur où le client est une application AngularJS :



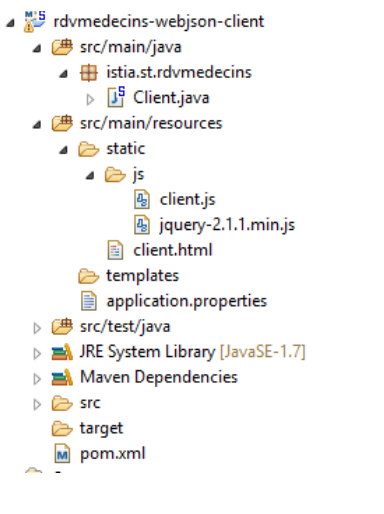
- les pages HTML / CSS / JS de l'application Angular viennent du serveur [1] ;
- en [2], le service [dao] fait une requête à un autre serveur, le serveur [2]. Et bien ça, c'est interdit par le navigateur qui exécute l'application Angular parce que c'est une faille de sécurité. L'application ne peut interroger que le serveur d'où elle vient, ç-à-d le serveur [1] ;

En fait, il est inexact de dire que le navigateur interdit à l'application Angular d'interroger le serveur [2]. Elle l'interroge en fait pour lui demander s'il autorise un client qui ne vient pas de chez lui à l'interroger. On appelle cette technique de partage, le **CORS** (Cross-Origin Resource Sharing). Le serveur [2] donne son accord en envoyant des entêtes HTTP précis.

Pour montrer les problèmes que l'on peut rencontrer, nous allons créer une application client / serveur où :

- le serveur sera notre serveur web / jSON ;
- le client sera une simple page HTML équipée d'un code Javascript qui fera des requêtes au serveur web / jSON ;

8.4.13.1 Le projet du client



Le projet est un projet Maven avec le fichier [pom.xml] suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>istia.st</groupId>
7.     <artifactId>springmvc-webjson-client</artifactId>
8.     <version>0.0.1-SNAPSHOT</version>
9.     <packaging>jar</packaging>
10.
11.    <name>springmvc-webjson-client</name>
12.    <description>Client for webjson server</description>
13.
14.    <parent>
15.       <groupId>org.springframework.boot</groupId>
16.       <artifactId>spring-boot-starter-parent</artifactId>
17.       <version>1.2.0.RELEASE</version>
18.       <relativePath/> <!-- lookup parent from repository -->
19.    </parent>
20.
21.    <properties>
22.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23.       <start-class>istia.st.rdvmedecins.Client</start-class>
24.       <java.version>1.7</java.version>
25.    </properties>
26.
27.    <dependencies>
28.       <dependency>
29.          <groupId>org.springframework.boot</groupId>
30.          <artifactId>spring-boot-starter-web</artifactId>
31.       </dependency>
32.       <dependency>
33.          <groupId>org.springframework.boot</groupId>
34.          <artifactId>spring-boot-starter-test</artifactId>
35.          <scope>test</scope>
36.       </dependency>
37.    </dependencies>
38.
39.    <build>
40.       <plugins>
41.          <plugin>
42.             <groupId>org.springframework.boot</groupId>
43.             <artifactId>spring-boot-maven-plugin</artifactId>
44.          </plugin>
45.       </plugins>
46.    </build>
47.
48. </project>
```

- lignes 14-19 : c'est un projet Spring Boot ;

- lignes 29-30 : on utilise la dépendance [spring-boot-starter-web] qui amène avec elle un serveur Tomcat et Spring MVC ;

La page HTML est la suivante :

Elle est générée par le code suivant :

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.  <meta charset="UTF-8">
5.  <title>Spring MVC</title>
6.  <script type="text/javascript" src="/js/jquery-2.1.1.min.js"></script>
7.  <script type="text/javascript" src="/js/client.js"></script>
8.  </head>
9.  <body>
10.    <h2>Client du service web / jSON</h2>
11.    <form id="formulaire">
12.       <!--  méthode HTTP -->
13.       Méthode HTTP :
14.       <!--  -->
15.       <input type="radio" id="get" name="method" value="get" checked="checked" />GET
16.       <!--  -->
17.       <input type="radio" id="post" name="method" value="post" />POST
18.       <!--  URL -->
19.       <br /> <br />URL cible : <input type="text" id="url" size="30"><br />
20.       <!-- valeur postée -->
21.       <br /> Chaîne jSON à poster : <input type="text" id="posted" size="50" />
22.       <!-- bouton de validation -->
23.       <br /> <br /> <input type="submit" value="Valider" onclick="javascript:requestServer(); return
    false;"></input>
24.    </form>
25.    <hr />
26.    <h2>Réponse du serveur</h2>
27.    <div id="response"></div>
28. </body>
29. </html>
```

- ligne 6 : on importe la bibliothèque jQuery ;
- ligne 7 : on importe un code que nous allons écrire ;

Le code [client.js] est le suivant :

```
1.  // données globales
2.  var url;
3.  var posted;
4.  var response;
5.  var method;
6.
7.  function requestServer() {
8.     // on récupère les informations du formulaire
9.     var urlValue = url.val();
```

```
10.     var postedValue = posted.val();
11.     method = document.forms[0].elements['method'].value;
12.     // on fait un appel Ajax à la main
13.     if (method === "get") {
14.        doGet(urlValue);
15.     } else {
16.        doPost(urlValue, postedValue);
17.     }
18. }
19.
20. function doGet(url) {
21.     // on fait un appel Ajax à la main
22.     $.ajax({
23.        headers : {
24.           'Authorization' : 'Basic YWRtaW46YWRtaW4='
25.        },
26.        url : 'http://localhost:8080' + url,
27.        type : 'GET',
28.        dataType : 'tex/plain',
29.        beforeSend : function() {
30.        },
31.        success : function(data) {
32.           // résultat texte
33.           response.text(data);
34.        },
35.        complete : function() {
36.        },
37.        error : function(jqXHR) {
38.           // erreur système
39.           response.text(jqXHR.responseText);
40.        }
41.     })
42. }
43.
44. function doPost(url, posted) {
45.     // on fait un appel Ajax à la main
46.     $.ajax({
47.        headers : {
48.           'Authorization' : 'Basic YWRtaW46YWRtaW4='
49.        },
50.        url : 'http://localhost:8080' + url,
51.        type : 'POST',
52.        contentType : 'application/json',
53.        data : posted,
54.        dataType : 'tex/plain',
55.        beforeSend : function() {
56.        },
57.        success : function(data) {
58.           // résultat texte
59.           response.text(data);
60.        },
61.        complete : function() {
62.        },
63.        error : function(jqXHR) {
64.           // erreur système
65.           response.text(jqXHR.responseText);
66.        }
67.     })
68. }
69.
70. // au chargement du document
71. $(document).ready(function() {
72.     // on récupère les références des composants de la page
73.     url = $("#url");
74.     posted = $("#posted");
75.     response = $("#response");
76. });
```

Nous laissons le lecteur comprendre ce code. Tout a déjà été rencontré à un moment ou à un autre. Certaines lignes méritent cependant une explication :

- ligne 11 :
  - [**document**] désigne le document chargé par le navigateur, ce qu'on appelle le DOM (Document Object Model),

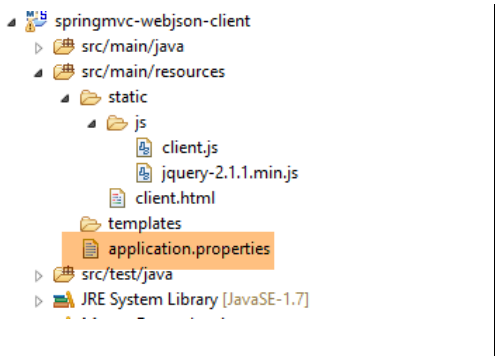
- [**document.forms[0]**] désigne le 1er formulaire du document, un document pouvant en avoir plusieurs. Ici, il n'y en qu'un,
- [**document.forms[0].elements['method']**] désigne l'élément du formulaire qui a l'attribut [name='method']. Il y en a deux :

```
1. <input type="radio" id="get" name="method" value="get" checked="checked" />GET
2. <input type="radio" id="post" name="method" value="post" />POST
```

- [**document.forms[0].elements['method'].value**] est la valeur qui va être postée pour le composant qui a l'attribut [name='method']. On sait que la valeur postée est la valeur de l'attribut [value] du bouton radio coché. Ici, ce sera donc l'une des chaînes ['get', 'post'] ;

- lignes 23-25 : on s'adresse à un serveur qui exige un entête HTTP [Authorization: Basic code]. Nous créons cette entête pour l'utilisateur [admin / admin] qui est le seul à pouvoir interroger le serveur ;
- ligne 26 : l'utilisateur saisira des URL du type [/getAllMedecins, /supprimerRv, ...]. Il faut donc compléter ces URL ;
- ligne 28 : le serveur renvoie du jSON qui est une forme de texte. On indique le type [text/plain] comme type de résultat afin de l'afficher tel qu'il a été reçu ;
- ligne 33 : affichage de la réponse texte du serveur ;
- ligne 39 : affichage du message d'erreur éventuel au format texte ;
- ligne 52 : pour indiquer que le client envoie du jSON ;

Le fichier [application.properties] nous permet de fixer le port de l'application web client :



Son contenu est le suivant :

```
server.port=8081
```

Ainsi :

- le client est une application web disponible à l'URL [http://localhost:8081];
- le serveur est une application web disponible à l'URL [http://localhost:8080];

Parce que le client n'est pas obtenu à partir du même port que le serveur, le problème des requêtes inter-domaines surgit. [http://localhost:8080] et [http://localhost:8081] sont deux domaines différents.

L'application Spring Boot est une application console lancée par la classe exécutable [Client] suivante :

```
1.  package istia.st.rdvmedecins;
2.
3.  import org.springframework.boot.SpringApplication;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.
6.  @EnableAutoConfiguration
7.  public class Client {
8.
9.      public static void main(String[] args) {
10.         SpringApplication.run(Client.class, args);
11.     }
12. }
```

8.4.13.2 L'URL [/getAllMedecins]

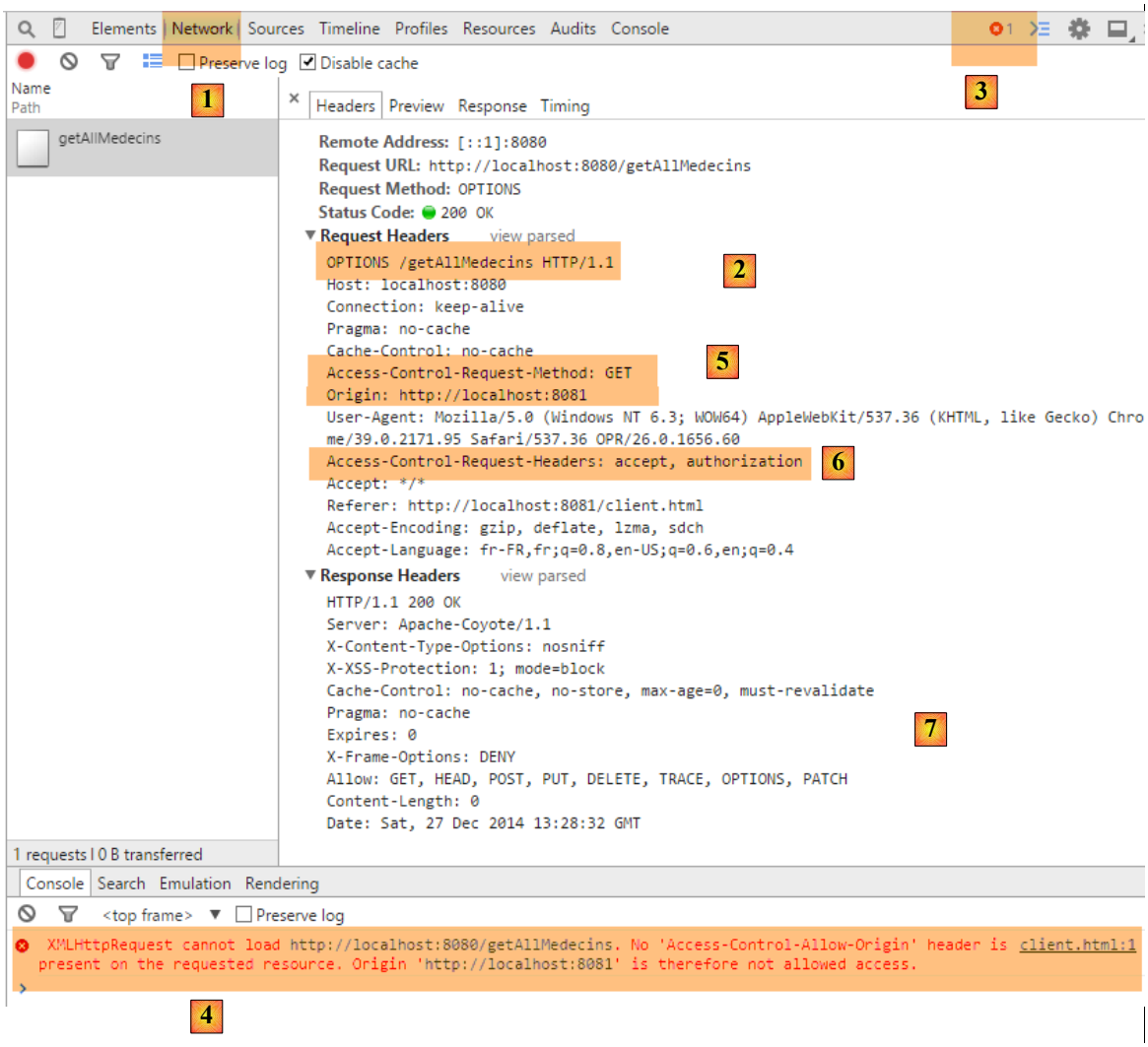
Nous lançons :

- le serveur web / json sur le port 8080 ;
- le client de ce serveur sur le port 8081 ;

puis nous demandons l'URL [http://localhost:8081/client.html] [1] :

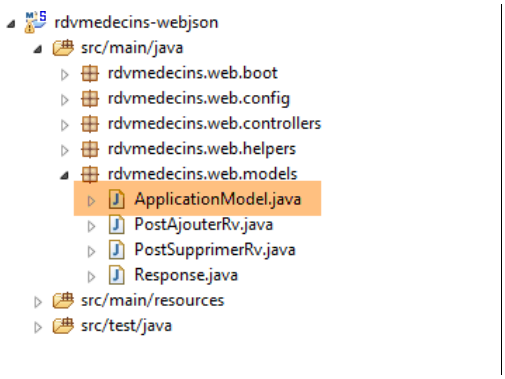
- en [2], nous faisons un GET sur l'URL [http://localhost:8080/getAllMedecins];

Nous n'obtenons pas de réponse du serveur. Lorsqu'on regarde la console de développement (Ctrl-Maj-I) on découvre une erreur :



- en [1], on est dans l'onglet [Network] ;
- en [2], on voit que la requête HTTP qui a été faite n'est pas [GET] mais [OPTIONS]. Dans le cas d'une requête inter-domaines, le navigateur vérifie auprès du serveur qu'un certain nombre de conditions sont vérifiées en lui envoyant une requête HTTP [OPTIONS]. En l'occurrence, les requêtes sont celles pointées par les pastilles [5-6] ;
- en [5], le navigateur demande si l'URL cible peut être atteinte avec un GET. L'entête de la requête [Access-Control-Request-Method] demande une réponse avec un entête HTTP [Access-Control-Allow-Methods] indiquant que la méthode demandée est acceptée ;
- en [5], le navigateur envoie l'entête HTTP [Origin: http://localhost:8081]. Cet entête demande une réponse dans un entête HTTP [Access-Control-Allow-Origin] indiquant que l'origine indiquée est acceptée ;
- en [6], le navigateur demande si les entêtes HTTP [accept] et [authorization] sont acceptés. L'entête de la requête [Access-Control-Request-Headers] attend une réponse avec un entête HTTP [Access-Control-Allow-Headers] indiquant que les entêtes demandés sont acceptés ;
- on a une erreur en [3]. En cliquant sur l'icône, on a l'erreur [4] ;
- en [4], le message indique que le serveur n'a pas envoyé l'entête HTTP [Access-Control-Allow-Origin] qui indique si l'origine de la requête est acceptée ;
- en [7], on peut constater que le serveur n'a effectivement pas envoyé cet entête. Du coup le navigateur a refusé de faire la requête HTTP GET demandée initialement ;

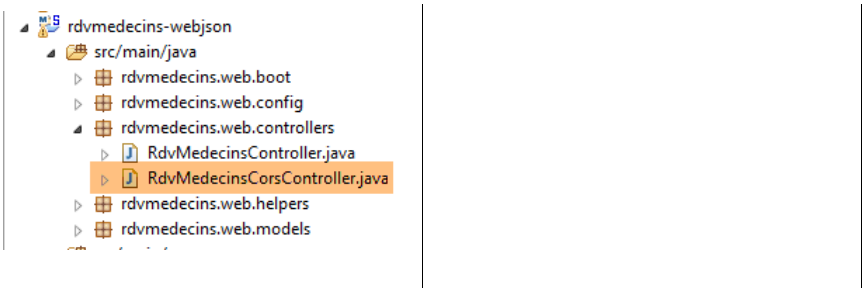
Il nous faut modifier le serveur web / jSON. Nous faisons une première modification dans [ApplicationModel] qui est l'un des éléments de configuration du service web :




```
1.  @Component
2.  public class ApplicationModel implements IMetier {
3.
4.      ...
5.      // données de configuration
6.      private boolean corsAllowed = true;
7.      private boolean secured = true;
8.
9.  ...
10.     public boolean isCorsAllowed() {
11.        return corsAllowed;
12. }
```

- ligne 6 : nous créons un booléen qui indique si on accepte ou non les clients étrangers au domaine du serveur ;
- lignes 10-12 : la méthode d'accès à cette information ;

Puis nous créons un nouveau contrôleur Spring MVC :



La classe [RdvMedecinsCorsController] est la suivante :

```
1.  package rdvmedecins.web.controllers;
2.
3.  import javax.servlet.http.HttpServletResponse;
4.
5.  import org.springframework.beans.factory.annotation.Autowired;
6.  import org.springframework.stereotype.Controller;
7.  import org.springframework.web.bind.annotation.RequestMapping;
8.  import org.springframework.web.bind.annotation.RequestMethod;
9.
10. import rdvmedecins.web.models.ApplicationModel;
11.
12. @Controller
13. public class RdvMedecinsCorsController {
14.
15.     @Autowired
16.     private ApplicationModel application;
17.
18.     // envoi des options au client
19.     public void sendOptions(String origin, HttpServletResponse response) {
20.        // Cors allowed ?
21.        if (!application.isCorsAllowed() || origin==null || !origin.startsWith("http://localhost")) {
22.           return;
```

```
23.         }
24.         // on fixe le header CORS
25.         response.addHeader("Access-Control-Allow-Origin", origin);
26.         // on autorise certains headers
27.         response.addHeader("Access-Control-Allow-Headers", "accept, authorization");
28.         // on autorise le GET
29.         response.addHeader("Access-Control-Allow-Methods", "GET");
30.     }
31.
32.     // liste des médecins
33.     @RequestMapping(value = "/getAllMedecins", method = RequestMethod.OPTIONS)
34.     public void getAllMedecins(@RequestHeader(value = "Origin", required = false) String origin,
        HttpServletResponse response) {
35.         sendOptions(origin, response);
36.     }
37. }
```

- lignes 12-13 : la classe [RdvMedecinsCorsController] est un contrôleur Spring ;
- lignes 33-36 : définissent une action traitant l'URL [/getAllMedecins] lorsqu'elle est demandée avec la commande HTTP [OPTIONS] ;
- ligne 34 : la méthode [getAllMedecins] admet pour paramètres :
  - l'objet [@RequestHeader(value = "Origin", required = **false**)] qui va récupérer l'entête HTTP [Origin] de la requête. Cet entête a été envoyé par l'émetteur de la requête :

    ```
    Origin:http://localhost:8081
    ```

    On indique que l'entête HTTP [Origin] est facultatif [required = **false**]. Dans ce cas, si l'entête est absent, le paramètre [String origin] aura la valeur *null*. Avec [required = **true**] qui est la valeur par défaut, une exception est lancée si l'entête est absent. On a voulu éviter ce cas ;
  - l'objet [HttpServletResponse response] qui va être envoyé au client qui a fait la demande ;

  Ces deux paramètres sont injectés par Spring ;

- ligne 35 : on délègue le traitement de la demande à la méthode des lignes 19-30 ;
- lignes 15-16 : l'objet [ApplicationModel] est injecté ;
- lignes 21-23 : si l'application est configurée pour accepter les requêtes inter-domaines et si l'émetteur a envoyé l'entête HTTP [Origin] et si cette origine commence par [http://localhost], alors on va accepter la requête inter-domaines, sinon on la rejette ;
- lignes 25 : si le client est dans le domaine [http://localhost:port], on envoie l'entête HTTP :

  ```
  Access-Control-Allow-Origin:  http://localhost:port
  ```

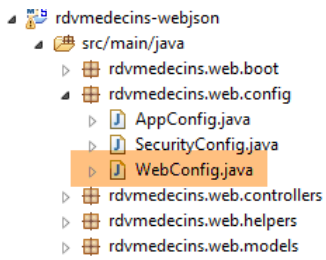
  qui signifie que le serveur accepte l'origine du client ;
- ligne 25 : nous avons signalé deux entêtes HTTP particuliers dans la requête HTTP [OPTIONS] :

  ```
  Access-Control-Request-Method: GET
  Access-Control-Request-Headers: accept, authorization
  ```

  A l'entête HTTP [Access-Control-Request-X], le serveur répond avec un entête HTTP [Access-Control-Allow-X] dans lequel il indique ce qui est autorisé. Les lignes 23-26 se contentent de reprendre la demande du client pour indiquer qu'elle est acceptée ;

Nous sommes désormais prêts pour de nouveaux tests. Nous lançons la nouvelle version du service web et nous découvrons que le problème reste entier. Rien n'a changé. Si ligne 35 ci-dessus, on met un affichage console, celui-ci n'est jamais affiché montrant par là que la méthode [getAllMedecins] de la ligne 34 n'est jamais appelée.

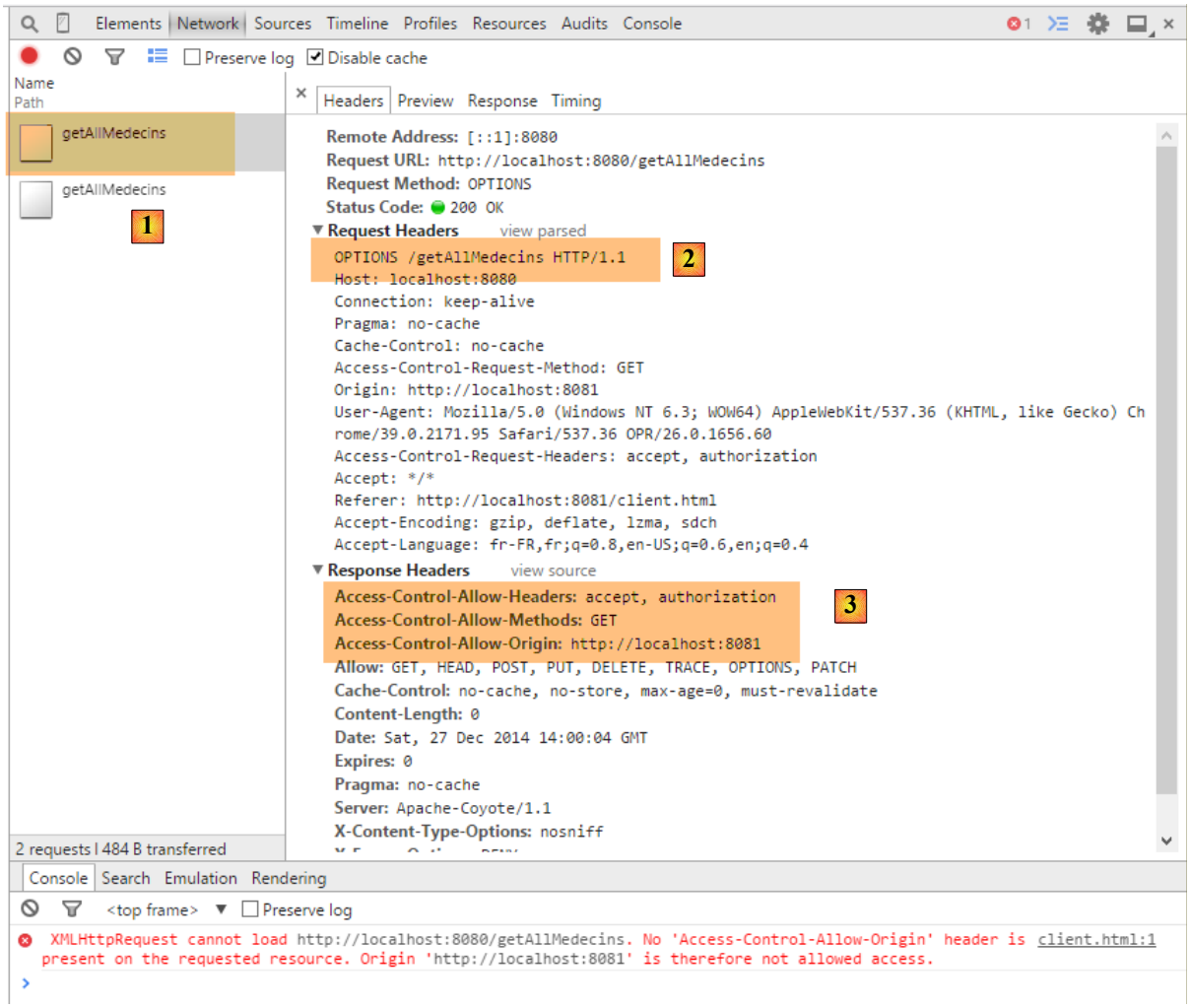
Après quelques recherches, on découvre que Spring MVC traite lui-même les commandes HTTP [OPTIONS] avec un traitement par défaut. Aussi c'est toujours Spring qui répond et jamais la méthode [getAllMedecins] de la ligne 34. Ce comportement par défaut de Spring MVC peut être changé. Nous modifions la classe [WebConfig] existante :




```
1.  package rdvmedecins.web.config;
2.
3.  ...
4.  import org.springframework.web.servlet.DispatcherServlet;
5.
6.  @Configuration
7.  public class WebConfig extends WebMvcConfigurerAdapter {
8.
9.     // configuration dispatcherservlet pour les headers CORS
10.    @Bean
11.    public DispatcherServlet dispatcherServlet() {
12.       DispatcherServlet servlet = new DispatcherServlet();
13.       servlet.setDispatchOptionsRequest(true);
14.       return servlet;
15.    }
16.
17.    // mapping jSON
18. ...
```

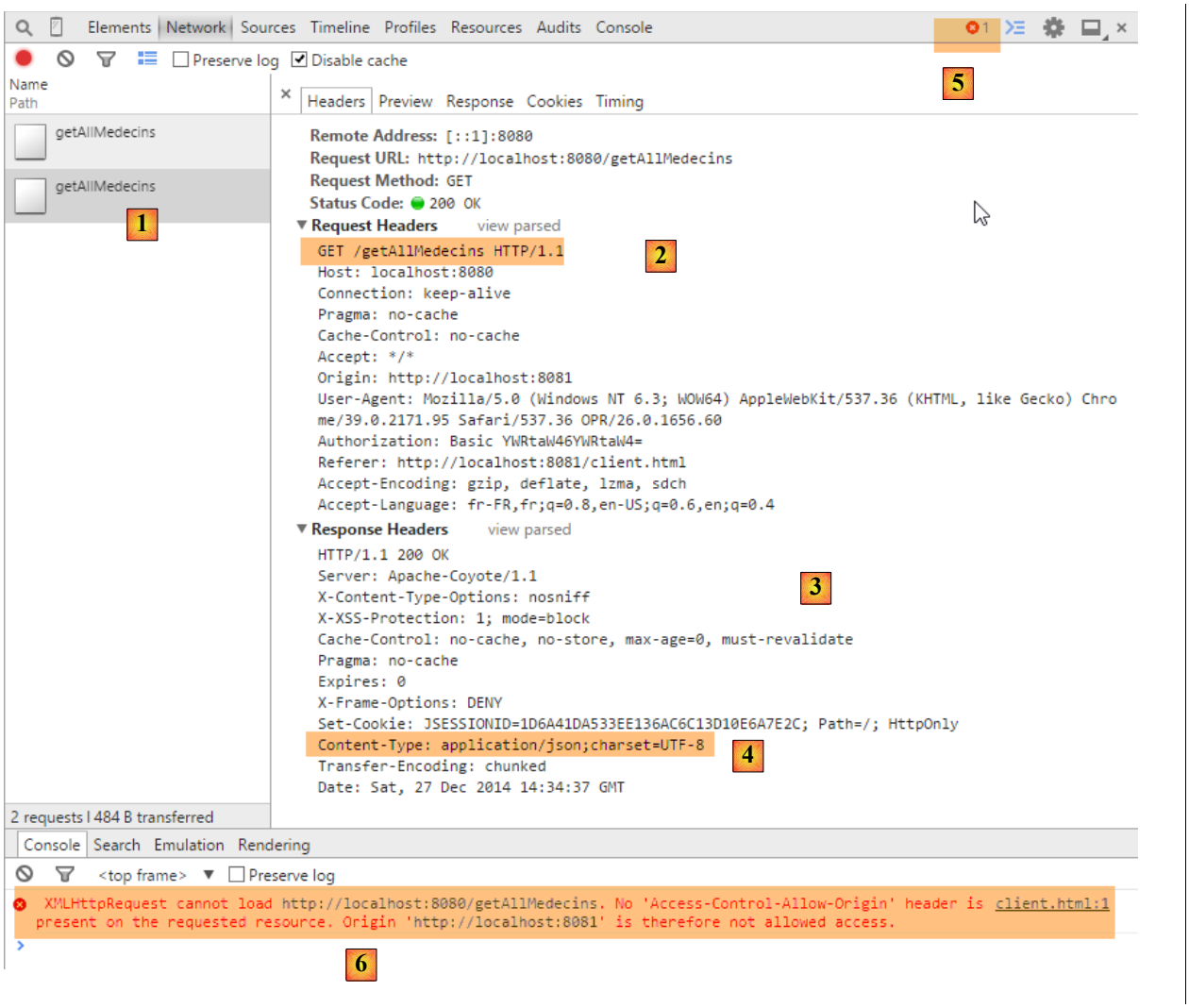
- lignes 10-11 : le bean [dispatcherServlet] sert à définir la servlet qui gère les demandes des clients. Elle est de type [DispatcherServlet]. Cette servlet est normalement créée par défaut. Si on la crée nous-mêmes, on peut alors la configurer ;
- ligne 12 : on crée une instance de type [DispatcherServlet] ;
- ligne 13 : on demande à ce que la servlet fasse suivre à l'application les commandes HTTP [OPTIONS] ;
- ligne 14 : on rend la servlet ainsi configurée ;

Nous refaisons les tests avec cette nouvelle configuration. On obtient le résultat suivant :



- en [1], nous voyons qu'il y a deux requêtes HTTP vers l'URL [http://localhost:8080/getAllMedecins];
- en [2], la requête [OPTIONS] ;
- en [3], les trois entêtes HTTP que nous venons de configurer dans la réponse du serveur ;

Examinons maintenant la seconde requête :



- en [1], la requête examinée ;
- en [2], c'est la requête GET. Grâce à la première requête [OPTIONS], le navigateur a reçu les informations qu'il demandait. Il réalise maintenant la requête [GET] demandée initialement ;
- en [3], la réponse du serveur ;
- en [4], le serveur envoie du jSON ;
- en [5], une erreur s'est produite ;
- en [6], le message d'erreur ;

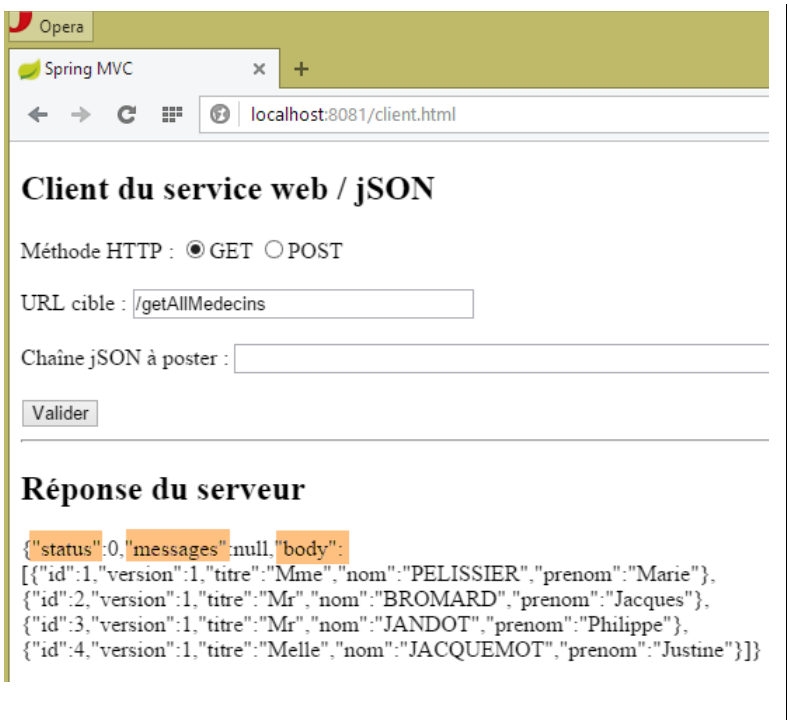
Il est plus difficile d'expliquer ce qui s'est passé ici. La réponse [3] du serveur est normale [HTTP/1.1 200 OK]. On devrait donc avoir le document demandé. Il est possible que le serveur ait bien envoyé le document mais que c'est le navigateur qui empêche son utilisation parce qu'il veut que pour la requête GET également, la réponse comporte l'entête HTTP [Access-Control-Allow-Origin:http://localhost:8081].

Nous modifions le contrôleur [RdvMedecinsController] de la façon suivante :

```
1.    @Autowired
2.    private RdvMedecinsCorsController rdvMedecinsCorsController;
3.  ...
4.
5.  @RequestMapping(value = "/getAllMedecins", method = RequestMethod.GET)
6.    public Response<List<Medecin>> getAllMedecins(HttpServletResponse response,
    @RequestHeader(value = "Origin", required = false) String origin) {
7.        // entêtes CORS
8.        rdvMedecinsCorsController.sendOptions(origin, response);
9.        // état de l'application
10.       ...
11. }
```

- lignes 1-2 : le contrôleur [RdvMedecinsCorsController] est injecté ;
- ligne 6 : on injecte dans les paramètres de la méthode [getAllMedecins], l'entête HTTP [Origin] ;
- ligne 8 : on fait appel à la méthode [sendOptions] du contrôleur [RdvMedecinsCorsController], celle-la même qui a été appelée pour traiter la requête HTTP [OPTIONS]. Elle va donc envoyer les mêmes entêtes HTTP que pour cette requête ;

Après cette modification, les résultats sont les suivants :



Nous avons bien obtenu la liste des médecins.

8.4.13.3 Les autres URL [GET]

Nous montrons maintenant les autres URL interrogées via un GET. Dans les contrôleurs, le code des actions qui les traitent suit le modèle des actions qui ont traité précédemment l'URL [/getAllMedecins]. Le lecteur peut vérifier le code dans les exemples livrés avec ce document. Voici un exemple :

dans [RdvMedecinsCorsController]

```
1.      // liste des Rv d'un médecin
2.      @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method = RequestMethod.OPTIONS)
3.      public void getRvMedecinJour(@RequestHeader(value = "Origin", required = false) String origin,
        HttpServletResponse response) {
4.          sendOptions(origin, response);
5.  }
```

dans [RdvMedecinsController]

```
1.      // liste des rendez-vous d'un médecin
2.      @RequestMapping(value = "/getRvMedecinJour/{idMedecin}/{jour}", method = RequestMethod.GET)
3.      public Response<List<Rv>> getRvMedecinJour(@PathVariable("idMedecin") long idMedecin,
4.              @PathVariable("jour") String jour, HttpServletResponse response,
5.              @RequestHeader(value = "Origin", required = false) String origin) {
6.          // entêtes CORS
7.          rdvMedecinsCorsController.sendOptions(origin, response);
8.          // état de l'application
9.          if (messages != null) {
10. ...
```

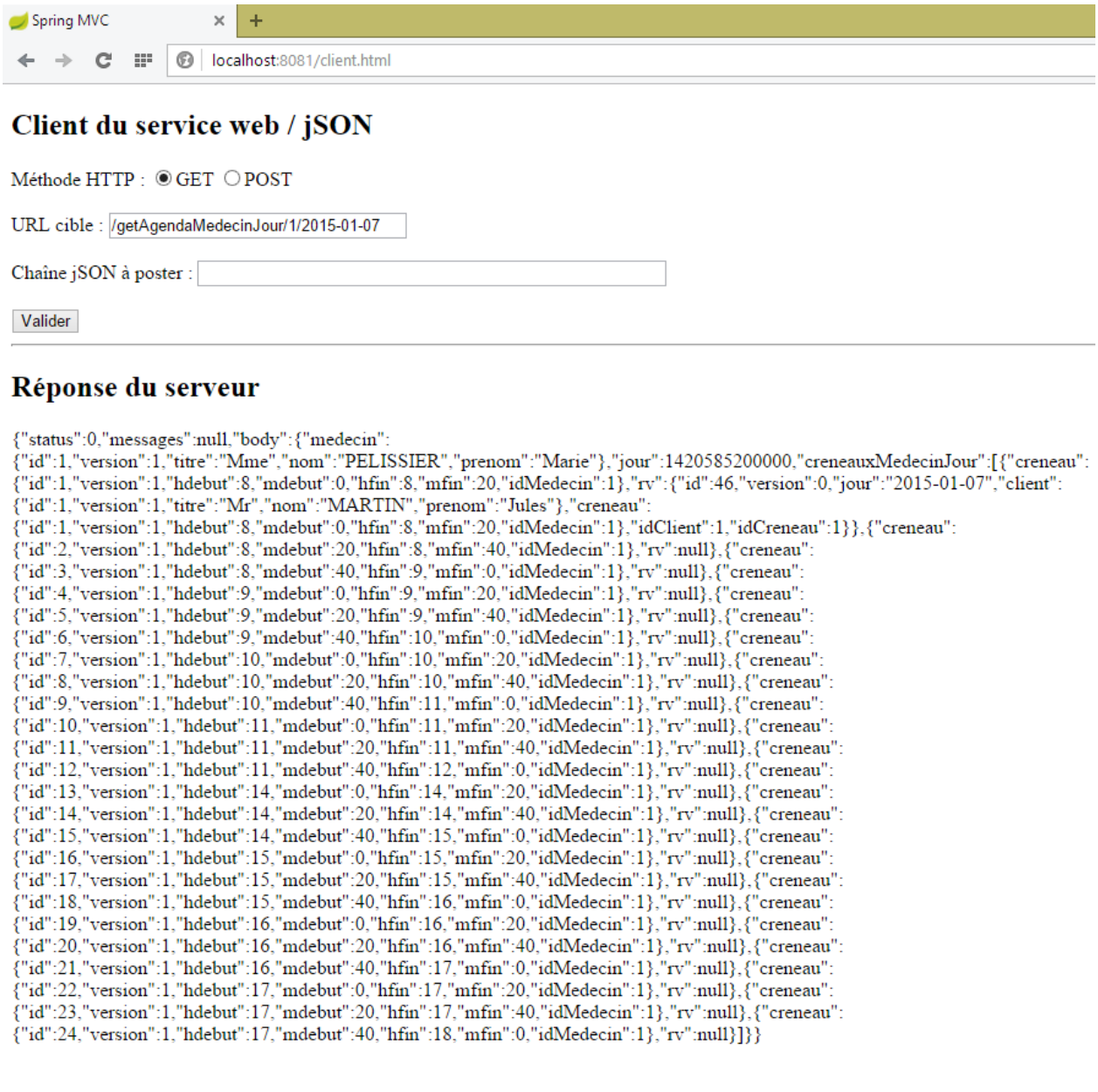
Voici maintenant des copies d'écran d'exécution :


Spring MVC
localhost:8081/client.html
Client du service web / jSON
Méthode HTTP : GET POST
URL cible : /getAllClients
Chaîne jSON à poster :
Valider
Réponse du serveur
{"status":0,"messages":null,"body":
[{"id":1,"version":1,"titre":"Mr","nom":"MARTIN","prenom":"Jules"},
{"id":2,"version":1,"titre":"Mme","nom":"GERMAN","prenom":"Christine"},
{"id":3,"version":1,"titre":"Mr","nom":"JACQUARD","prenom":"Jules"},
{"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"}]}


Opera
Spring MVC
localhost:8081/client.html
Client du service web / jSON
Méthode HTTP : GET POST
URL cible : /getRvMedecinJour/1/2015-01-07
Chaîne jSON à poster :
Valider
Réponse du serveur
{"status":0,"messages":null,"body":[{"id":46,"version":0,"jour":"2015-01-07","client":
{"id":1,"version":1,"titre":"Mr","nom":"MARTIN","prenom":"Jules"},"creneau":
{"id":1,"version":1,"hdebut":8,"mdebut":0,"hfin":8,"mfin":20,"idMedecin":1},"idClient":1,"idCreneau":1}]}


Spring MVC
localhost:8081/client.html
Client du service web / jSON
Méthode HTTP : GET POST
URL cible : /getClientById/4
Chaîne jSON à poster :
Valider
Réponse du serveur
{"status":0,"messages":null,"body":{"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"}}


Spring MVC
localhost:8081/client.html
Client du service web / jSON
Méthode HTTP : GET POST
URL cible : /getCreneauById/4
Chaîne jSON à poster :
Valider
Réponse du serveur
{"status":0,"messages":null,"body":
{"id":4,"version":1,"hdebut":9,"mdebut":0,"hfin":9,"mfin":20,"idMedecin":1}}

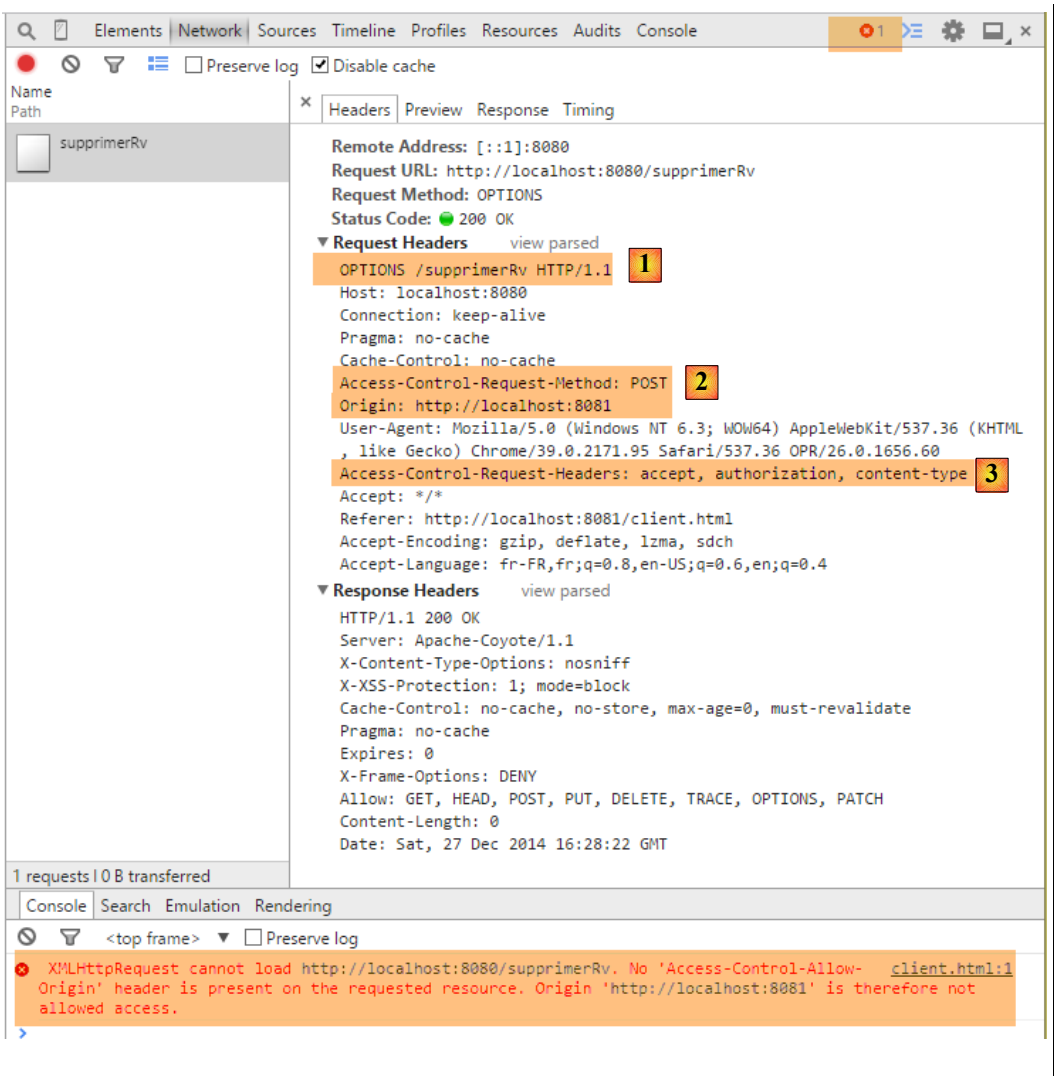


8.4.13.4 Les URL [POST]

Examinons le cas suivant :

- on fait un POST [1] vers l'URL [2] ;
- en [3], la valeur postée. Il s'agit d'une chaîne jSON ;
- au total, on cherche à supprimer le rendez-vous ayant l'[id] 100 ;

Nous ne modifions pour l'instant aucun code. Le résultat obtenu est alors le suivant :



- en [1], comme pour les requêtes [GET], une requête [OPTIONS] est faite par le navigateur ;
- en [2], il demande une autorisation d'accès pour une requête [POST]. Auparavant c'était [GET] ;
- en [3], il demande une autorisation d'envoyer les entêtes HTTP [accept, authorization, content-type]. Auparavant, on avait seulement les deux premiers entêtes ;

Nous modifions la méthode [RdvMedecinsCorsController.sendOptions] de la façon suivante :

```
1.     public void sendOptions(String origin, HttpServletResponse response) {
2.        // Cors allowed ?
3.        if (!application.isCorsAllowed() || origin==null || !origin.startsWith("http://localhost")) {
4.           return;
5.        }
6.        // on fixe le header CORS
7.        response.addHeader("Access-Control-Allow-Origin", origin);
8.        // on autorise certains headers
9.        response.addHeader("Access-Control-Allow-Headers", "accept, authorization, content-type");
10.       // on autorise le GET
11.       response.addHeader("Access-Control-Allow-Methods", "GET, POST");
12. }
```

- ligne 9 : on a ajouté l'entête HTTP [Content-Type] (la casse n'a pas d'importance) ;
- ligne 11 : on a ajouté la méthode HTTP [POST] ;

Ceci fait les méthodes [POST] sont traitées de la même façon que les requêtes [GET]. Voici l'exemple de l'URL [/supprimerRv] :

dans [RdvMedecinsController]

```
1.    @RequestMapping(value = "/supprimerRv", method = RequestMethod.POST, consumes =
   "application/json; charset=UTF-8")
2.    public Response<Void> supprimerRv(@RequestBody PostSupprimerRv post, HttpServletResponse
   response, @RequestHeader(value = "Origin", required = false) String origin) {
3.       // entêtes CORS
4.       rdvMedecinsCorsController.sendOptions(origin, response);
5.       // état de l'application
6.       if (messages != null) {
7. ...
```

dans [RdvMedecinsCorsController]

```
1.    @RequestMapping(value = "/supprimerRv", method = RequestMethod.OPTIONS)
2.    public void supprimerRv(@RequestHeader(value = "Origin", required = false) String origin,
   HttpServletResponse response) {
3.       sendOptions(origin, response);
4. }
```

Le résultat obtenu est le suivant :

Pour l'URL [/ajouterRv], on obtient le résultat suivant :

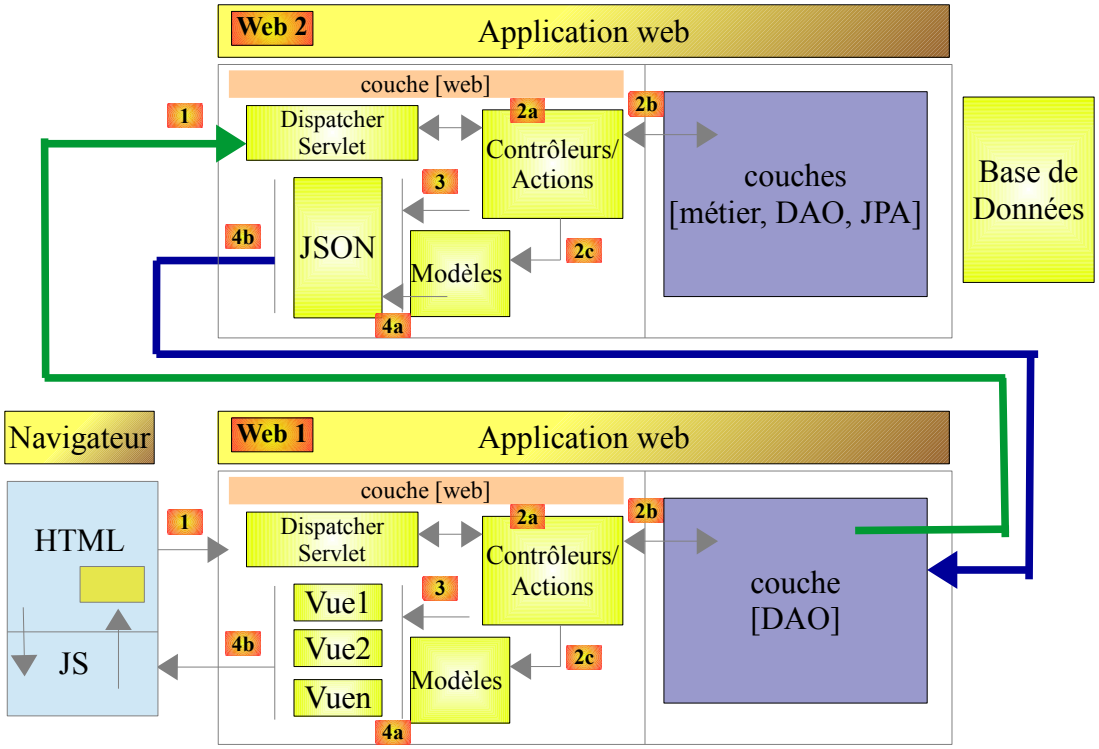
8.4.13.5 Conclusion

Notre application supporte désormais les requêtes inter-domaines. Celles-ci peuvent être autorisées ou non par configuration dans la classe [ApplicationModel] :

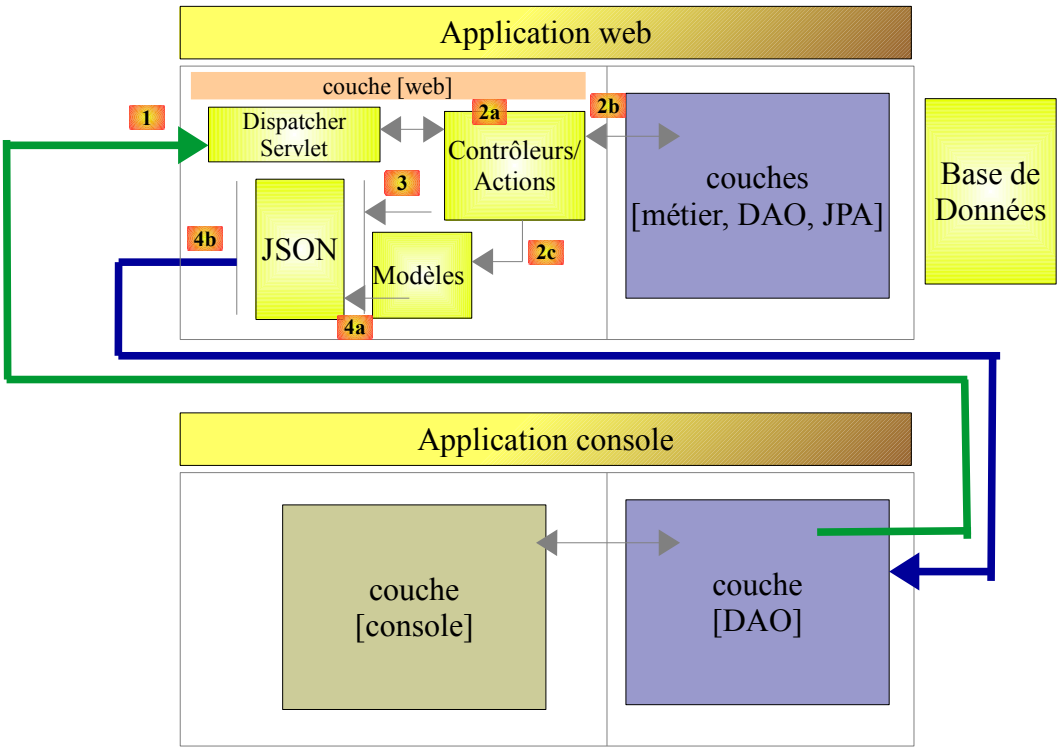
```
  // données de configuration
  private boolean corsAllowed = false;
```

8.5 Client programmé du service web / jSON

Revenons à l'architecture générale de l'application que nous voulons écrire :

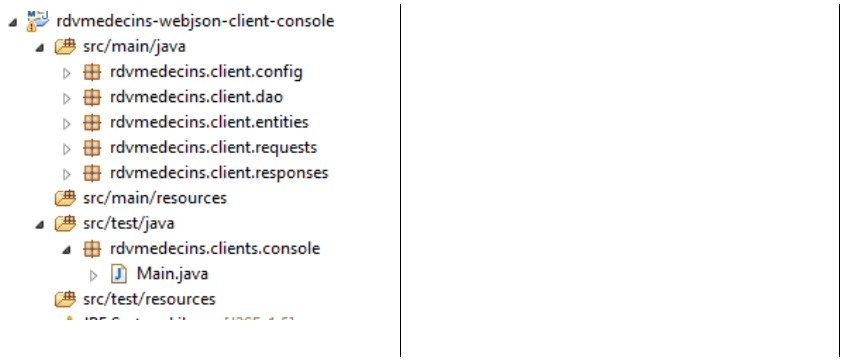


La partie haute du schéma a été écrite. C'est le serveur web / jSON. Nous nous attaquons maintenant à la partie basse et d'abord à sa couche [DAO]. Nous allons écrire celle-ci puis la tester avec un client console. L'architecture de test sera la suivante :



8.5.1 Le projet du client console

Le projet STS du client console sera le suivant :



8.5.2 Configuration Maven

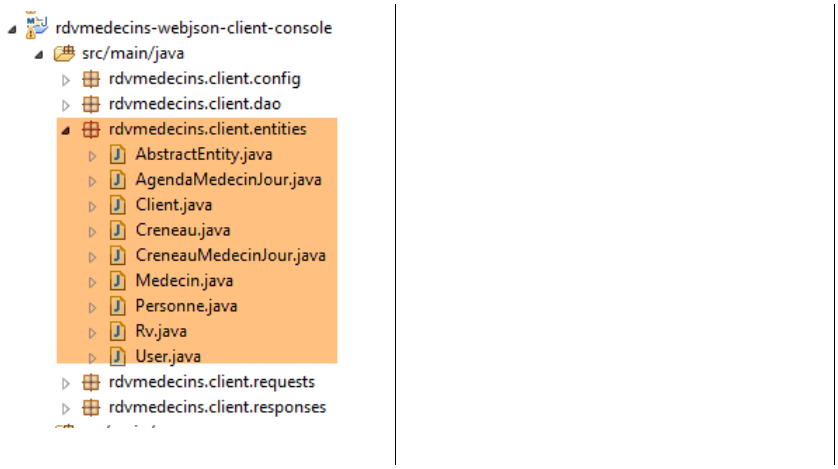
Le fichier [pom.xml] du client console est le suivant :

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.     <modelVersion>4.0.0</modelVersion>
4.     <groupId>istia.st.rdvmedecins</groupId>
5.     <artifactId>rdvmedecins-webjson-client-console</artifactId>
6.     <version>0.0.1-SNAPSHOT</version>
7.     <name>rdvmedecins-webjson-client-console</name>
8.     <description>rdvmedecins-webjson-client-console</description>
9.
10.    <properties>
11.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
12.       <java.version>1.7</java.version>
13.    </properties>
14.
15.    <dependencies>
16.       <!-- Spring -->
17.       <dependency>
18.          <groupId>org.springframework</groupId>
19.          <artifactId>spring-web</artifactId>
20.          <version>4.1.3.RELEASE</version>
21.       </dependency>
22.       <!-- librairie jSON utilisée par Spring -->
23.       <dependency>
24.          <groupId>com.fasterxml.jackson.core</groupId>
25.          <artifactId>jackson-core</artifactId>
26.          <version>2.4.4</version>
27.       </dependency>
28.       <dependency>
29.          <groupId>com.fasterxml.jackson.core</groupId>
30.          <artifactId>jackson-databind</artifactId>
31.          <version>2.4.4</version>
32.       </dependency>
33.       <!-- composant utilisé par Spring RestTemplate -->
34.       <dependency>
35.          <groupId>org.apache.httpcomponents</groupId>
36.          <artifactId>httpclient</artifactId>
37.          <version>4.3.5</version>
38.       </dependency>
39.    </dependencies>
40. </project>
```

- lignes 17-21 : le client console du serveur web / jSON est basé sur un composant appelé [RestTemplate] fourni par la dépendance [spring-web] ;
- lignes 23-32 : la sérialisation / désérialisation des objets jSON nécessite une bibliothèque jSON. Nous utilisons une variante de la librairie Jackson utilisée par Spring web ;

- lignes 34-38 : au plus bas niveau, le composant [RestTemplate] communique avec le serveur via des sockets TCP/IP. Nous voulons fixer le [timeout] de celles-ci, ç-à-d le temps maximum d'attente d'une réponse du serveur. Le composant [RestTemplate] ne nous permet pas de fixer celui-ci. Pour le faire, nous allons passer au constructeur [RestTemplate] un composant de bas niveau fourni par la dépendance [org.apache.httpcomponents.httpclient]. C'est cette dépendance qui va nous permettre de fixer le [timeout] de la communication ;

8.5.3 Le package [rdvmedecins.client.entities]

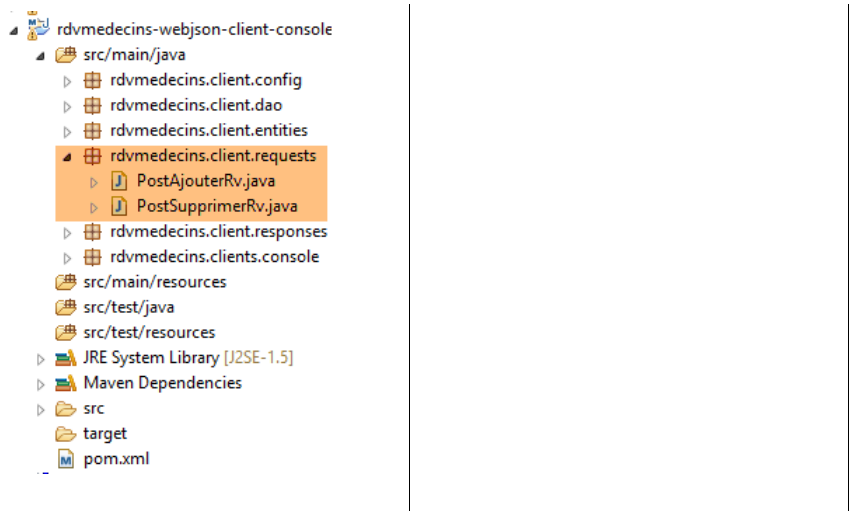


Le package [rdvmedecins.client.entities] rassemble toutes les entités que le service web / jSON envoie via ses différentes URL. Nous n'allons pas les détailler de nouveau. On se contentera de dire que les entités JPA [Client, Creneau, Medecin, Rv, Personne] ont été débarrassées de toutes leurs annotations JPA ainsi que de leurs annotations jSON. Voici par exemple, la classe [Rv] :

```
1.  package rdvmedecins.client.entities;
2.
3.  import java.util.Date;
4.
5.  public class Rv extends AbstractEntity {
6.     private static final long serialVersionUID = 1L;
7.
8.     // jour du Rv
9.     private Date jour;
10.
11.    // un rv est lié à un client
12.    private Client client;
13.
14.    // un rv est lié à un créneau
15.    private Creneau creneau;
16.
17.    // clés étrangères
18.    private long idClient;
19.    private long idCreneau;
20.
21.    // constructeur par défaut
22.    public Rv() {
23.    }
24.
25.    // avec paramètres
26.    public Rv(Date jour, Client client, Creneau creneau) {
27.       this.jour = jour;
28.       this.client = client;
29.       this.creneau = creneau;
30.    }
31.
32.    // toString
33.    public String toString() {
34.       return String.format("Rv[%d, %s, %d, %d]", id, jour, client.id, creneau.id);
```

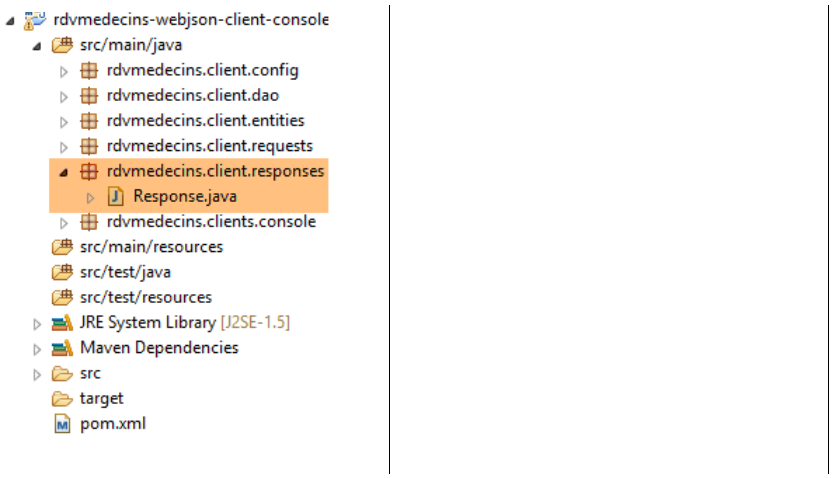
```
35.    }
36.
37. // getters et setters
38. ...
39. }
```

8.5.4 Le package [rdvmedecins.client.requests]



Le package [rdvmedecins.client.requests] rassemple les deux classes dont la valeur jSON est postée aux URL [/ajouterRv] et [supprimerRv]. Elles sont identiques à ce qu'elles sont côté serveur.

8.5.5 Le package [rdvmedecins.client.responses]



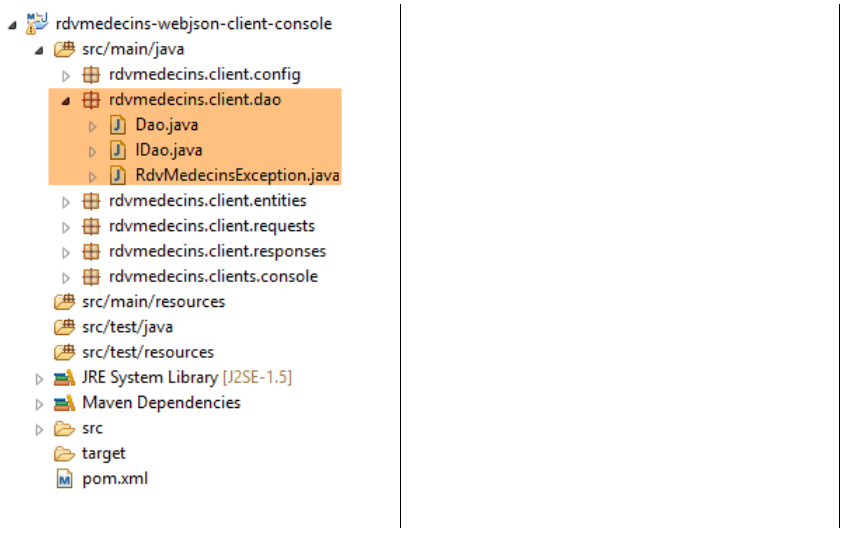
[Response] est le type de toutes les réponses du service web / jSON. C'est un type générique :

```
1.  package rdvmedecins.client.responses;
2.
3.  import java.util.List;
4.
5.  public class Response<T> {
6.
7.      // ----------------- propriétés
8.      // statut de l'opération
9.      private int status;
10.     // les éventuels messages d'erreur
11.     private List<String> messages;
12.     // le corps de la réponse
13.     private T body;
14.
```

```
15.     // constructeurs
16.     public Response() {
17.
18.     }
19.
20.     public Response(int status, List<String> messages, T body) {
21.        this.status = status;
22.        this.messages = messages;
23.        this.body = body;
24.     }
25.
26.     // getters et setters
27. ...
28. }
```

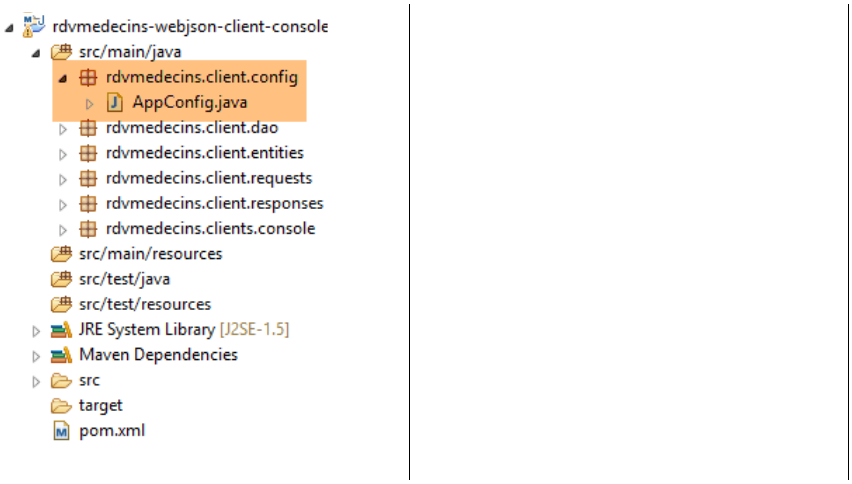
- ligne 5 : le type [T] varie selon l'URL du service web / jSON ;

8.5.6 Le package [rdvmedecins.client.dao]



- [IDao] est l'interface de la couche [DAO] et [Dao] son implémentation. Nous allons revenir sur cette implémentation ;

8.5.7 Le package [rdvmedecins.client.config]



La classe [AppConfig] configure l'application. Son code est le suivant :

```
1.  package rdvmedecins.client.config;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
```

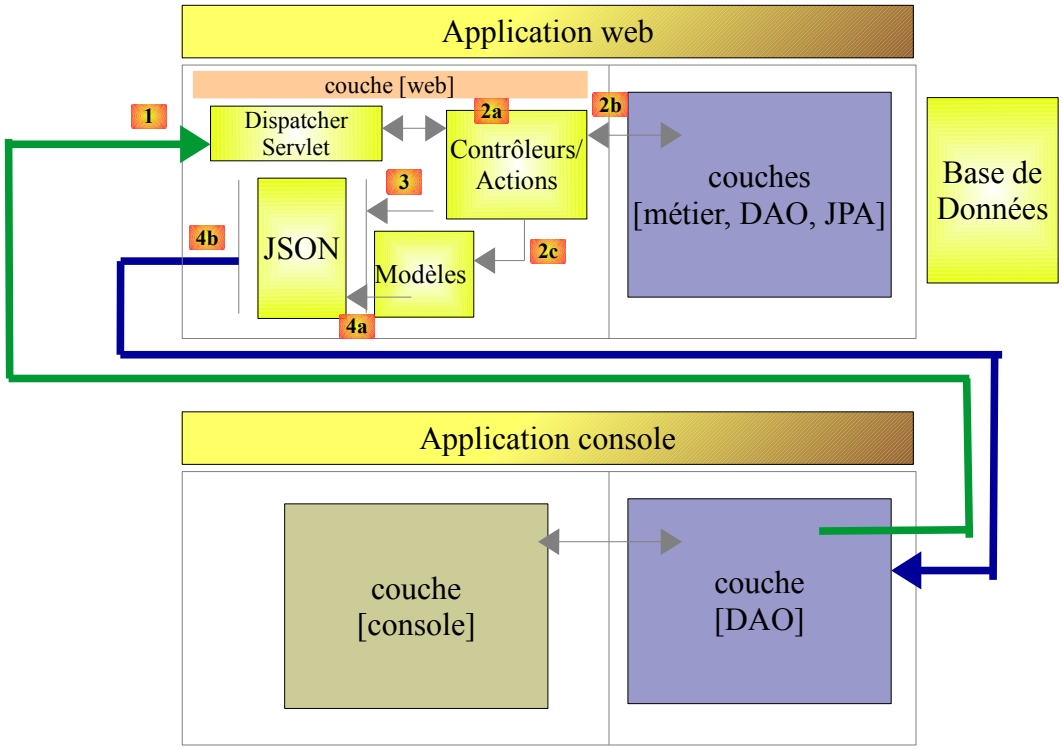
```
5.
6.  import org.springframework.context.annotation.Bean;
7.  import org.springframework.context.annotation.ComponentScan;
8.  import org.springframework.context.annotation.Configuration;
9.  import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
10. import org.springframework.http.converter.HttpMessageConverter;
11. import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
12. import org.springframework.web.client.RestTemplate;
13.
14. @Configuration
15. @ComponentScan({ "rdvmedecins.client.dao" })
16. public class AppConfig {
17.
18.    @Bean
19.    public RestTemplate restTemplate() {
20.       // création du composant RestTemplate
21.       HttpComponentsClientHttpRequestFactory factory = new HttpComponentsClientHttpRequestFactory();
22.       RestTemplate restTemplate = new RestTemplate(factory);
23.       // convertisseur jSON
24.       List<HttpMessageConverter<?>> messageConverters = new ArrayList<HttpMessageConverter<?>>();
25.       messageConverters.add(new MappingJackson2HttpMessageConverter());
26.       restTemplate.setMessageConverters(messageConverters);
27.       // résultat
28.       return restTemplate;
29.    }
30. }
```

- ligne 14 : la classe [AppConfig] est une classe de configuration Spring ;
- ligne 15 : la package [rdvmedecins.client.dao] sera exploré pour y chercher des composants Spring. On y trouvera le composant [Dao] ;
- lignes 18-29 : définissent un singleton Spring de nom [restTemplate] (le nom de la méthode). Cette méthode rend une instance [RestTemplate] qui est l'outil de base que Spring fournit pour communiquer avec un service web / jSON ;
- ligne 22 : on pourrait écrire [RestTemplate restTemplate = **new** RestTemplate() ;]. C'est suffisant dans la plupart des cas. Mais ici, nous voulons fixer les [timeout] du client. Pour cela, on injecte dans le composant [RestTemplate], un composant de bas niveau de type [HttpComponentsClientHttpRequestFactory] (ligne 21) qui va nous permettre de fixer ces [timeout]. La dépendance Maven nécessaire a été présentée ;
- lignes 24-26 : fixe la liste des convertisseurs du client. Un convertisseur est une classe capable de :
  - sérialiser le corps [body] d'une requête POST du client ;
  - de désérialiser le corps [body] de la réponse du serveur ;

  Ici, nous n'avons besoin que d'un sérialiseur / désérialiseur jSON. Ligne 25, nous utilisons la classe [MappingJackson2HttpMessageConverter] qui est fournie par Spring et qui s'appuie sur la bibliothèque Jackson dont nous avons présenté la dépendance Maven ;
- ligne 28 : on rend l'objet [RestTemplate]. Rappelons que l'intérêt d'un bean Spring est d'être injectable dans d'autres beans ;

8.5.8 L'interface [IDao]

Revenons à l'architecture de l'application :



La couche [DAO] est un adaptateur entre la couche [console] et les URL exposées par le service web / jSON. Son interface [IDao] sera la suivante :

```
1.  package rdvmedecins.client.dao;
2.
3.  import java.util.List;
4.
5.  import rdvmedecins.client.entities.AgendaMedecinJour;
6.  import rdvmedecins.client.entities.Client;
7.  import rdvmedecins.client.entities.Creneau;
8.  import rdvmedecins.client.entities.Medecin;
9.  import rdvmedecins.client.entities.Rv;
10. import rdvmedecins.client.entities.User;
11.
12. public interface IDao {
13.     // Url du service web
14.     public void setUrlServiceWebJson(String url);
15.
16.     // timeout
17.     public void setTimeout(int timeout);
18.
19.     // authentification
20.     public void authenticate(User user);
21.
22.     // liste des clients
23.     public List<Client> getAllClients(User user);
24.
25.     // liste des Médecins
26.     public List<Medecin> getAllMedecins(User user);
27.
28.     // liste des créneaux horaires d'un médecin
29.     public List<Creneau> getAllCreneaux(User user, long idMedecin);
30.
31.     // trouver un client identifié par son id
32.     public Client getClientById(User user, long id);
33.
34.     // trouver un client identifié par son id
35.     public Medecin getMedecinById(User user, long id);
36.
37.     // trouver un Rv identifié par son id
38.     public Rv getRvById(User user, long id);
39.
40.     // trouver un créneau horaire identifié par son id
```

```
41.     public Creneau getCreneauById(User user, long id);
42.
43.     // ajouter un RV
44.     public Rv ajouterRv(User user, String jour, long idCreneau, long idClient);
45.
46.     // supprimer un RV
47.     public void supprimerRv(User user, long idRv);
48.
49.     // liste des Rv d'un médecin, un jour donné
50.     public List<Rv> getRvMedecinJour(User user, long idMedecin, String jour);
51.
52.     // agenda
53.     public AgendaMedecinJour getAgendaMedecinJour(User user, long idMedecin, String jour);
54.
55. }
```

- ligne 14 : la méthode permettant de fixer l'URL racine du service web / jSON, par exemple [http://localhost:8080];
- ligne 17 : la méthode qui permet de fixer les [timeout] côté client. On veut contrôler ce paramètre car certains clients HTTP sont parfois très longs à attendre une réponse qui ne viendra pas ;
- ligne 20 : la méthode permettant d'identifier un utilisateur [login, passwd]. Lance une exception si l'utilisateur n'est pas reconnu ;
- lignes 22-53 : à chaque URL exposée par le service web / jSON est associée une méthode de l'interface dont la signature découle de la signature de la méthode côté serveur traitant l'URL exposée. Prenons par exemple, l'URL serveur suivante :

```
1.     @RequestMapping(value = "/getAgendaMedecinJour/{idMedecin}/{jour}", method = RequestMethod.GET)
2.     public Response<AgendaMedecinJour> getAgendaMedecinJour(@PathVariable("idMedecin") long idMedecin,
       @PathVariable("jour") String jour, HttpServletResponse response, @RequestHeader(value = "Origin",
   required = false) String origin) {
```

- ligne 1 : on voit que [idMedecin] et [jour] sont les paramètres de l'URL. Ce seront les paramètres d'entrée de la méthode associée à cette URL côté client ;
- ligne 2 : on voit que la méthode serveur rend un type [Response<AgendaMedecinJour>]. Le type du résultat de la méthode associée à cette URL côté client sera [AgendaMedecinJour] ;

Côté client, on déclare la méthode suivante :

```
public AgendaMedecinJour getAgendaMedecinJour(User user, long idMedecin, String jour);
```

Cette signature convient lorsque le serveur envoie une réponse [int status, List<String> messages, AgendaMedecinJour body] avec [status==0]. Dans ce cas nous avons [messages==null && body!=null]. Elle ne convient pas lorsque [status!=0]. Dans ce cas nous avons [messages!=null && body==null]. Il nous faut d'une façon ou d'une autre signaler qu'il y a eu une erreur. Pour cela nous lancerons une exception de type [RdvMedecinsException] suivant :

```
1. package rdvmedecins.client.dao;
2.
3. import java.util.List;
4.
5. public class RdvMedecinsException extends RuntimeException {
6.
7.     private static final long serialVersionUID = 1L;
8.     // code d'erreur
9.     private int status;
10.    // liste de messages d'erreur
11.    private List<String> messages;
12.
13.    public RdvMedecinsException() {
14.    }
15.
16.    public RdvMedecinsException(int code, List<String> messages) {
17.      super();
18.      this.status = code;
19.      this.messages = messages;
20.    }
21.
22.    // getters et setters
23. ...
24. }
```

- lignes 9 et 11 : l'exception reprendra les valeurs des champs [status, messages] de l'objet [Response<T>] envoyé par le serveur ;

- ligne 5 : la classe [RdvMedecinsException] étend la classe [RuntimeException]. C'est donc une exception non contrôlée, ç-à-d qu'il n'y a pas obligation de la gérer avec un try / catch et de la déclarer dans la signature des méthodes de l'interface ;

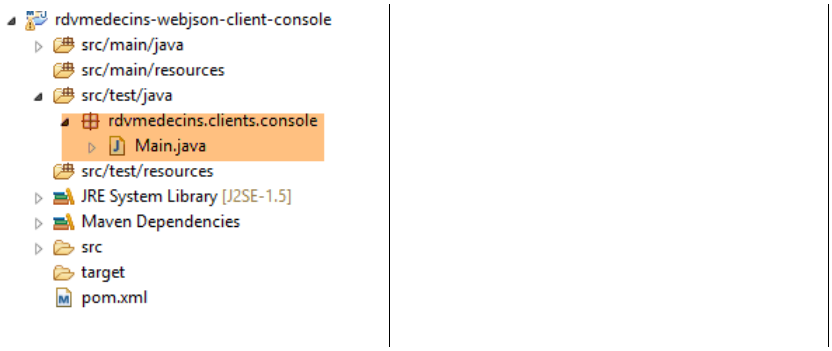
Par ailleurs, toutes les méthodes de l'interface [IDao] qui interrogent le service web / jSON ont pour paramètre, le type [User] suivant :

```
1.  package rdvmedecins.client.entities;
2.
3.  public class User {
4.
5.      // data
6.      private String login;
7.      private String passwd;
8.
9.      // constructeurs
10.     public User() {
11.     }
12.
13.     public User(String login, String passwd) {
14.        this.login = login;
15.        this.passwd = passwd;
16.     }
17.
18.     // getters et setters
19.     ...
20. }
```

En effet, chaque échange avec le service web / jSON doit être accompagné d'un entête HTTP d'authentification.

8.5.9 Le package [rdvmedecins.clients.console]

Maintenant que nous connaissons l'interface de la couche [DAO], nous pouvons présenter l'application console.



La classe [Main] est la suivante :

```
1.  package rdvmedecins.clients.console;
2.
3.  import java.io.IOException;
4.
5.  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6.
7.  import rdvmedecins.client.config.DaoConfig;
8.  import rdvmedecins.client.dao.IDao;
9.  import rdvmedecins.client.dao.RdvMedecinsException;
10. import rdvmedecins.client.entities.Rv;
11. import rdvmedecins.client.entities.User;
12.
13. import com.fasterxml.jackson.core.JsonProcessingException;
14. import com.fasterxml.jackson.databind.ObjectMapper;
15.
16. public class Main {
17.
18.     // sérialiseur jSON
19.     static private ObjectMapper mapper = new ObjectMapper();
```

```
20.    // timeout des connexions en millisecondes
21.    static private int TIMEOUT = 1000;
22.
23.    public static void main(String[] args) throws IOException {
24.       // on récupère une référence sur la couche [DAO]
25.       AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(DaoConfig.class);
26.       IDao dao = context.getBean(IDao.class);
27.       // on fixe l'URL du service web / json
28.       dao.setUrlServiceWebJson("http://localhost:8080");
29.       // on fixe les timeout en millisecondes
30.       dao.setTimeout(TIMEOUT);
31.
32.       // Authentification
33.       String message = "/authenticate [admin,admin]";
34.       try {
35.          dao.authenticate(new User("admin", "admin"));
36.          System.out.println(String.format("%s : OK", message));
37.       } catch (RdvMedecinsException e) {
38.          showException(message, e);
39.       }
40.
41.       message = "/authenticate [user,user]";
42.       try {
43.          dao.authenticate(new User("user", "user"));
44.          System.out.println(String.format("%s : OK", message));
45.       } catch (RdvMedecinsException e) {
46.          showException(message, e);
47.       }
48.
49.       message = "/authenticate [user,x]";
50.       try {
51.          dao.authenticate(new User("user", "x"));
52.          System.out.println(String.format("%s : OK", message));
53.       } catch (RdvMedecinsException e) {
54.          showException(message, e);
55.       }
56.
57.       message = "/authenticate [x,x]";
58.       try {
59.          dao.authenticate(new User("x", "x"));
60.          System.out.println(String.format("%s : OK", message));
61.       } catch (RdvMedecinsException e) {
62.          showException(message, e);
63.       }
64.
65.       message = "/authenticate [admin,x]";
66.       try {
67.          dao.authenticate(new User("admin", "x"));
68.          System.out.println(String.format("%s : OK", message));
69.       } catch (RdvMedecinsException e) {
70.          showException(message, e);
71.       }
72.
73.       // liste des clients
74.       message = "/getAllClients";
75.       try {
76.          showResponse(message, dao.getAllClients(new User("admin", "admin")));
77.       } catch (RdvMedecinsException e) {
78.          showException(message, e);
79.       }
80.
81.       // liste des médecins
82.       message = "/getAllMedecins";
83.       try {
84.          showResponse(message, dao.getAllMedecins(new User("admin", "admin")));
85.       } catch (RdvMedecinsException e) {
86.          showException(message, e);
87.       }
88.
89.       // liste des créneaux du médecin 2
90.       message = "/getAllCreneaux/2";
91.       try {
92.          showResponse(message, dao.getAllCreneaux(new User("admin", "admin"), 2L));
93.       } catch (RdvMedecinsException e) {
```

```
94.             showException(message, e);
95.         }
96.
97.         // client n° 1
98.         message = "/getClientById/1";
99.         try {
100.            showResponse(message, dao.getClientById(new User("admin", "admin"), 1L));
101.        } catch (RdvMedecinsException e) {
102.            showException(message, e);
103.        }
104.
105.        // médecin n° 2
106.        message = "/getMedecinById/2";
107.        try {
108.            showResponse(message, dao.getMedecinById(new User("admin", "admin"), 2L));
109.        } catch (RdvMedecinsException e) {
110.            showException(message, e);
111.        }
112.
113.        // créneau n° 3
114.        message = "/getCreneauById/3";
115.        try {
116.            showResponse(message, dao.getCreneauById(new User("admin", "admin"), 3L));
117.        } catch (RdvMedecinsException e) {
118.            showException(message, e);
119.        }
120.
121.        // rv n° 4
122.        message = "/getRvById/4";
123.        try {
124.            showResponse(message, dao.getRvById(new User("admin", "admin"), 4L));
125.        } catch (RdvMedecinsException e) {
126.            showException(message, e);
127.        }
128.
129.        // ajout d'un rv
130.        message = "/AjouterRv [idClient=4,idCreneau=8,jour=2015-01-08]";
131.        long idRv = 0;
132.        try {
133.            Rv response = dao.ajouterRv(new User("admin", "admin"), "2015-01-08", 8L, 4L);
134.            idRv = response.getId();
135.            showResponse(message, response);
136.        } catch (RdvMedecinsException e) {
137.            showException(message, e);
138.        }
139.
140.        // liste des rv du médecin 1 le 2015-01-08
141.        message = "/getRvMedecinJour/1/2015-01-08";
142.        try {
143.            showResponse(message, dao.getRvMedecinJour(new User("admin", "admin"), 1L, "2015-01-08"));
144.        } catch (RdvMedecinsException e) {
145.            showException(message, e);
146.        }
147.
148.        // agenda du médecin 1 le 2015-01-08
149.        message = "/getAgendaMedecinJour/1/2015-01-08";
150.        try {
151.            showResponse(message, dao.getAgendaMedecinJour(new User("admin", "admin"), 1L, "2015-01-08"));
152.        } catch (RdvMedecinsException e) {
153.            showException(message, e);
154.        }
155.        // suppression du rv ajouté
156.        message = String.format("/supprimerRv [idRv=%s]", idRv);
157.        try {
158.            dao.supprimerRv(new User("admin", "admin"), idRv);
159.        } catch (RdvMedecinsException e) {
160.            showException(message, e);
161.        }
162.
163.        // liste des rv du médecin 1 le 2015-01-08
164.        message = "/getRvMedecinJour/1/2015-01-08";
165.        try {
166.            showResponse(message, dao.getRvMedecinJour(new User("admin", "admin"), 1L, "2015-01-08"));
167.        } catch (RdvMedecinsException e) {
168.            showException(message, e);
```

```
169.       }
170.       // fermeture contexte
171.       context.close();
172.    }
173.
174.    private static void showException(String message, RdvMedecinsException e) {
175.       System.out.println(String.format("URL [%s]", message));
176.       System.out.println(String.format("L'erreur n° [%s] s'est produite :", e.getStatus()));
177.       for (String msg : e.getMessages()) {
178.          System.out.println(msg);
179.       }
180.    }
181.
182.    private static <T> void showResponse(String message, T response) throws JsonProcessingException {
183.       System.out.println(String.format("URL [%s]", message));
184.       System.out.println(mapper.writeValueAsString(response));
185.    }
186.}
```

- ligne 19 : le sérialiseur jSON qui va nous permettre d'afficher la réponse du serveur, ligne 184 ;
- ligne 25 : le composant [AnnotationConfigApplicationContext] est un composant Spring capable d'exploiter les annotations de configuration d'une application Spring. Nous passons à son constructeur, la classe [AppConfig] qui configure l'application ;
- ligne 26 : on récupère une référence sur la couche [DAO] ;
- lignes 27-30 : on la configure ;
- lignes 32-169 : on teste toutes les méthodes de l'interface [IDao] ;

Les résultats obtenus sont les suivants :

```
1.  /authenticate [admin,admin] : OK
2.  URL [/authenticate [user,user]]
3.  L'erreur n° [111] s'est produite :
4.  403 Forbidden
5.  URL [/authenticate [user,x]]
6.  L'erreur n° [111] s'est produite :
7.  403 Forbidden
8.  URL [/authenticate [x,x]]
9.  L'erreur n° [111] s'est produite :
10. 403 Forbidden
11. URL [/authenticate [admin,x]]
12. L'erreur n° [111] s'est produite :
13. 401 Unauthorized
14. URL [/getAllClients]
15. [{"id":1,"version":1,"titre":"Mr","nom":"MARTIN","prenom":"Jules"},
    {"id":2,"version":1,"titre":"Mme","nom":"GERMAN","prenom":"Christine"},
    {"id":3,"version":1,"titre":"Mr","nom":"JACQUARD","prenom":"Jules"},
    {"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"}]
16. URL [/getAllMedecins]
17. [{"id":1,"version":1,"titre":"Mme","nom":"PELISSIER","prenom":"Marie"},
    {"id":2,"version":1,"titre":"Mr","nom":"BROMARD","prenom":"Jacques"},
    {"id":3,"version":1,"titre":"Mr","nom":"JANDOT","prenom":"Philippe"},
    {"id":4,"version":1,"titre":"Melle","nom":"JACQUEMOT","prenom":"Justine"}]
18. URL [/getAllCreneaux/2]
19. [{"id":25,"version":1,"hdebut":8,"mdebut":0,"hfin":8,"mfin":20,"medecin":null,"idMedecin":2},
    {"id":26,"version":1,"hdebut":8,"mdebut":20,"hfin":8,"mfin":40,"medecin":null,"idMedecin":2},
    {"id":27,"version":1,"hdebut":8,"mdebut":40,"hfin":9,"mfin":0,"medecin":null,"idMedecin":2},
    {"id":28,"version":1,"hdebut":9,"mdebut":0,"hfin":9,"mfin":20,"medecin":null,"idMedecin":2},
    {"id":29,"version":1,"hdebut":9,"mdebut":20,"hfin":9,"mfin":40,"medecin":null,"idMedecin":2},
    {"id":30,"version":1,"hdebut":9,"mdebut":40,"hfin":10,"mfin":0,"medecin":null,"idMedecin":2},
    {"id":31,"version":1,"hdebut":10,"mdebut":0,"hfin":10,"mfin":20,"medecin":null,"idMedecin":2},
    {"id":32,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"medecin":null,"idMedecin":2},
    {"id":33,"version":1,"hdebut":10,"mdebut":40,"hfin":11,"mfin":0,"medecin":null,"idMedecin":2},
    {"id":34,"version":1,"hdebut":11,"mdebut":0,"hfin":11,"mfin":20,"medecin":null,"idMedecin":2},
    {"id":35,"version":1,"hdebut":11,"mdebut":20,"hfin":11,"mfin":40,"medecin":null,"idMedecin":2},
    {"id":36,"version":1,"hdebut":11,"mdebut":40,"hfin":12,"mfin":0,"medecin":null,"idMedecin":2}]
20. URL [/getClientById/1]
21. {"id":1,"version":1,"titre":"Mr","nom":"MARTIN","prenom":"Jules"}
22. URL [/getMedecinById/2]
23. {"id":2,"version":1,"titre":"Mr","nom":"BROMARD","prenom":"Jacques"}
24. URL [/getCreneauById/3]
25. {"id":3,"version":1,"hdebut":8,"mdebut":40,"hfin":9,"mfin":0,"medecin":null,"idMedecin":1}
26. URL [/getRvById/4]
```

```
27. L'erreur n° [2] s'est produite :
28. Le rendez-vous d'id [4] n'existe pas
29. URL [/AjouterRv [idClient=4,idCreneau=8,jour=2015-01-08]]
30. {"id":50,"version":0,"jour":1420671600000,"client":
    {"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"},"creneau":
    {"id":8,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"medecin":null,"idMedecin":1},"idClient":0
    ,"idCreneau":0}
31. URL [/getRvMedecinJour/1/2015-01-08]
32. [{"id":50,"version":0,"jour":1420675200000,"client":
    {"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"},"creneau":
    {"id":8,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"medecin":null,"idMedecin":1},"idClient":4
    ,"idCreneau":8}]
33. URL [/getAgendaMedecinJour/1/2015-01-08]
34. {"medecin":
    {"id":1,"version":1,"titre":"Mme","nom":"PELISSIER","prenom":"Marie"},"jour":1420671600000,"creneauxMedeci
    nJour":[{"creneau":
    {"id":1,"version":1,"hdebut":8,"mdebut":0,"hfin":8,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":2,"version":1,"hdebut":8,"mdebut":20,"hfin":8,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":3,"version":1,"hdebut":8,"mdebut":40,"hfin":9,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":4,"version":1,"hdebut":9,"mdebut":0,"hfin":9,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":5,"version":1,"hdebut":9,"mdebut":20,"hfin":9,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":6,"version":1,"hdebut":9,"mdebut":40,"hfin":10,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":7,"version":1,"hdebut":10,"mdebut":0,"hfin":10,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":8,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"medecin":null,"idMedecin":1},"rv":
    {"id":50,"version":0,"jour":1420675200000,"client":
    {"id":4,"version":1,"titre":"Melle","nom":"BISTROU","prenom":"Brigitte"},"creneau":
    {"id":8,"version":1,"hdebut":10,"mdebut":20,"hfin":10,"mfin":40,"medecin":null,"idMedecin":1},"idClient":4
    ,"idCreneau":8}},{"creneau":
    {"id":9,"version":1,"hdebut":10,"mdebut":40,"hfin":11,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":10,"version":1,"hdebut":11,"mdebut":0,"hfin":11,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":11,"version":1,"hdebut":11,"mdebut":20,"hfin":11,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":12,"version":1,"hdebut":11,"mdebut":40,"hfin":12,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":13,"version":1,"hdebut":14,"mdebut":0,"hfin":14,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":14,"version":1,"hdebut":14,"mdebut":20,"hfin":14,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":15,"version":1,"hdebut":14,"mdebut":40,"hfin":15,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":16,"version":1,"hdebut":15,"mdebut":0,"hfin":15,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":17,"version":1,"hdebut":15,"mdebut":20,"hfin":15,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":18,"version":1,"hdebut":15,"mdebut":40,"hfin":16,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":19,"version":1,"hdebut":16,"mdebut":0,"hfin":16,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":20,"version":1,"hdebut":16,"mdebut":20,"hfin":16,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":21,"version":1,"hdebut":16,"mdebut":40,"hfin":17,"mfin":0,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":22,"version":1,"hdebut":17,"mdebut":0,"hfin":17,"mfin":20,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":23,"version":1,"hdebut":17,"mdebut":20,"hfin":17,"mfin":40,"medecin":null,"idMedecin":1},"rv":null},
    {"creneau":
    {"id":24,"version":1,"hdebut":17,"mdebut":40,"hfin":18,"mfin":0,"medecin":null,"idMedecin":1},"rv":null}]}
35. URL [/getRvMedecinJour/1/2015-01-08]
36. []
```

Nous laissons au lecteur le soin d'associer les résultats au code. Celui-ci montre comment appeler chaque méthode de la couche [DAO]. Notons simplement quelques points :

- lignes 3-13 : montrent que lors d'une erreur d'authentification, le serveur renvoie un status HTTP [403 Forbidden] ou [401 Unauthorized] selon les cas ;
- lignes 29-30 : on ajoute un Rv au médecin n° 1 ;
- lignes 31-32 : on voit ce rendez-vous. C'est le seul dans la journée ;
- lignes 33-34 : on le voit également dans l'agenda du médecin ;
- lignes 35-36 : le rendez-vous a disparu. Le code l'a entre-temps supprimé ;

8.5.10 Implémentation de la couche [DAO]

Il nous reste maintenant à présenter le coeur de la couche [DAO], l'implémentation de son interface [IDao]. Nous allons le faire progressivement.

```
1.  package rdvmedecins.client.dao;
2.
3.  import java.net.URI;
4.  import java.util.ArrayList;
5.  import java.util.Base64;
6.  import java.util.List;
7.
8.  import org.springframework.beans.factory.annotation.Autowired;
9.  import org.springframework.core.ParameterizedTypeReference;
10. import org.springframework.http.HttpMethod;
11. import org.springframework.http.MediaType;
12. import org.springframework.http.RequestEntity;
13. import org.springframework.http.ResponseEntity;
14. import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
15. import org.springframework.stereotype.Service;
16. import org.springframework.web.client.RestTemplate;
17.
18. import rdvmedecins.client.entities.AgendaMedecinJour;
19. import rdvmedecins.client.entities.Client;
20. import rdvmedecins.client.entities.Creneau;
21. import rdvmedecins.client.entities.Medecin;
22. import rdvmedecins.client.entities.Rv;
23. import rdvmedecins.client.entities.User;
24. import rdvmedecins.client.requests.PostAjouterRv;
25. import rdvmedecins.client.requests.PostSupprimerRv;
26. import rdvmedecins.client.responses.Response;
27.
28. import com.fasterxml.jackson.core.type.TypeReference;
29. import com.fasterxml.jackson.databind.ObjectMapper;
30.
31. @Service
32. public class Dao implements IDao {
33.
34.     // data
35.     @Autowired
36.     private RestTemplate restTemplate;
37.     private String urlServiceWebJson;
38.     // sérialiseur jSON
39.     static private ObjectMapper mapper = new ObjectMapper();
40.
41.     // URL service web / jSON
42.     public void setUrlServiceWebJson(String url) {
43.         this.urlServiceWebJson = url;
44.     }
45.
46.     public void setTimeout(int timeout) {
47.         // on fixe le timeout des requêtes du client web
48.         HttpComponentsClientHttpRequestFactory factory = (HttpComponentsClientHttpRequestFactory)
    restTemplate
49.                 .getRequestFactory();
50.         factory.setConnectTimeout(timeout);
51.         factory.setReadTimeout(timeout);
52.     }
```

- ligne 31 : l'annotation [@Service] fait de la classe [Dao] un composant Spring. C'est ce qui nous a permis de l'injecter dans la classe [Main] de la couche console ;
- lignes 35-36 : nous injectons le bean [restTemplate] que nous avons défini dans la classe de configuration [AppConfig] ;
- ligne 37 : l'URL racine du service web / jSON ;
- ligne 39 : le sérialiseur / désérialiseur jSON ;

- lignes 46-52 : fixent le timeout du client lorsqu'il attend une réponse du serveur ;
- ligne 48 : nous récupérons le composant [HttpComponentsClientHttpRequestFactory] que nous avions injecté dans le bean [restTemplate] lors de la création de celui-ci (cf [AppConfig]) ;
- ligne 50 : nous fixons le temps maximum d'attente du client lorsqu'il établit une connexion avec le serveur ;
- ligne 51 : nous fixons le temps maximum d'attente du client lorsqu'il attend une réponse à l'une de ses requêtes ;

L'implémentation des méthodes de communication avec le serveur va être factorisée dans la méthode générique suivante :

```
1.  // requête générique
2.      private <T1, T2> T1 getResponse(User user, String url, HttpMethod method, int errStatus, T2 body) {
3.          ...
4.      }
```

- lignes 1 : la méthode [getResponse] est une méthode générique paramétrée par deux types :
  - [T1] : est le type de réponse attendu du serveur dans [Response<T1>], par exemple [AgendaMedecinJour],
  - [T2] : est le type du paramètre jSON posté par les opérations POST, par exemple [PostAjouterRv] ;
- ligne 1 : la méthode [getResponse] rend un résultat de type T1, par exemple [List<Client>] ;
- ligne 1 : les paramètres de [getResponse] sont les suivants :
  - [User user] : l'utilisateur qui fait la connexion ;
  - [String url] : l'URL à interroger. Il s'agit de la fin de l'URL, la premièr partie étant fournie par le champ [`urlServiceWebJson`] de la classe,
  - [HttpMethod method] : méthode HTTP de la requête, GET ou POST selon les cas,
  - [int errStatus] : code d'erreur à utiliser dans la classe [RdvMedecinsException], si erreur il y a lors de la communication avec le serveur,
  - [T2 body] : la valeur à poster si POST il y a ;

Continuons :

```
1.  // requête générique
2.      private <T1, T2> T1 getResponse(User user, String url, HttpMethod method, int errStatus, T2 body) {
3.          // la réponse du serveur
4.          ResponseEntity<Response<T1>> response;
5.          try {
6.              // on prépare la requête
7.              RequestEntity<?> request = null;
8.              if (method == HttpMethod.GET) {
9.                  request = RequestEntity.get(new URI(String.format("%s%s", urlServiceWebJson, url)))
10.                         .header("Authorization", getBase64(user)).accept(MediaType.APPLICATION_JSON).build();
11.             }
12.             if (method == HttpMethod.POST) {
13.                 request = RequestEntity.post(new URI(String.format("%s%s", urlServiceWebJson, url)))
14.                         .header("Authorization", getBase64(user)).header("Content-Type", "application/json")
15.                         .accept(MediaType.APPLICATION_JSON).body(body);
16.             }
17.             // on exécute la requête
18.             response = restTemplate.exchange(request, new ParameterizedTypeReference<Response<T1>>() {
19.             });
20.         } catch (Exception e) {
21.             // on encapsule l'exception
22.             throw new RdvMedecinsException(errStatus, getMessagesForException(e));
23.         }
24.         ...
25.     }
```

- ligne 18 : l'instruction qui fait la requête au serveur et reçoit sa réponse. Le composant [RestTemplate] offre un nombre important de méthodes d'échange avec le serveur mais seule la méthode [exchange] admet des paramètres génériques. C'est pour cette raison qu'elle a été choisie. Le second paramètre fixe le type de la réponse attendue. Le premier paramètre est la requête de type [RequestEntity] (ligne 7). Le résultat de la méthode [exchange] est de type [ResponseEntity<Response<T1>>] (ligne 4). Le type [ResponseEntity] encapsule la réponse complète du serveur, entêtes HTTP et document envoyés par celui-ci. De même le type [RequestEntity] encapsule toute la requête du client incluant les entêtes HTTP et l'éventuelle valeur postée ;
- lignes 5-16 : il nous faut construire la requête de type [RequestEntity]. Elle est différente selon que l'on utilise un GET ou un POST pour faire la requête ;
- ligne 9 : la requête pour un GET. La classe [RequestEntity] offre des méthodes statiques pour créer les requêtes GET, POST, HEAD,... La méthode [RequestEntity.get] permet de créer une requête GET en chaînant les différentes méthodes qui construisent celle-ci :
  - la méthode [RequestEntity.get] admet pour paramètre l'URL cible sous la forme d'une instance URI,

    - la méthode [header] permet de définir les entêtes HTTP que l'on souhaite utiliser, ici celui de l'autorisation. Nous allons revenir sur la méthode [getBase64] ;
    - la méthode [accept] permet de définir les éléments de l'entête HTTP [Accept]. Ici, nous indiquons que nous acceptons le type [application/json] que va envoyer le serveur ;
    - la méthode [build] utilise ces différentes informations pour construire le type [RequestEntity] de la requête ;
- ligne 13 : la requête pour un POST. La méthode [RequestEntity.post] permet de créer une requête POST en chaînant les différentes méthodes qui construisent celle-ci :
    - la méthode [RequestEntity.post] admet pour paramètre l'URL cible sous la forme d'une instance URI,
    - la méthode [header] permet de définir les entêtes HTTP que l'on souhaite utiliser, ici celui de l'autorisation,
    - la méthode [header] qui suit envoie au serveur l'entête [Content-Type: application/json] pour lui indiquer que la valeur postée va lui arriver sous la forme d'une chaîne jSON ;
    - la méthode [accept] permet d'indiquer que nous acceptons le type [application/json] que va envoyer le serveur ;
    - la méthode [body] fixe la valeur postée. Celle-ci est le 4ième paramètre de la méthode générique [getResponse] (ligne 1) ;
- lignes 20-23 : s'il se produit une erreur de communication avec le serveur on lance une exception de type [RdvMedecinsException] avec pour code d'erreur, le paramètre [errStatus] passé en 3ième paramètre de la méthode générique [getResponse] (ligne 2) ;

La méthode privée [getBase64] fournit le code Base64 de la chaîne 'login:passwd' pour l'entête HTTP d'authentification :

```
1.     private String getBase64(User user) {
2.        // on encode en base 64 l'utilisateur et son mot de passe - nécessite java 8
3.        String chaîne = String.format("%s:%s", user.getLogin(), user.getPasswd());
4.        return String.format("Basic %s", new String(Base64.getEncoder().encode(chaîne.getBytes())));
5.  }
```

La méthode [getResponse] se poursuit de la façon suivante :

```
1.  // requête générique
2.     private <T1, T2> T1 getResponse(User user, String url, HttpMethod method, int errStatus, T2 body) {
3.        ...
4.        // on récupère le corps de la réponse
5.        Response<T1> entity = response.getBody();
6.        int status = entity.getStatus();
7.        // des erreurs côté serveur ?
8.        if (status != 0) {
9.           // on crée une exception
10.          throw new RdvMedecinsException(status, entity.getMessages());
11.       } else {
12.          // c'est bon
13.          return entity.getBody();
14.       }
15.    }
```

- ligne 5 : nous avons reçu la réponse du serveur. Elle est de type [ResponseEntity<Response<T1>>] (ligne 4 du précédent code étudié). Nous récupérons le document de type [Response<T1>] encapsulé dans la réponse. Ce type a les champs [int status, List<String> messages, T1 body] ;
- ligne 6 : nous récupérons le [status] de la réponse qui est un code d'erreur ;
- lignes 8-11 : s'il y a erreur, alors on lance une exception reprenant les deux informations [status, messages] de la réponse du serveur ;
- ligne 13 : sinon nous rendons le type [T1] contenu dans la réponse de type [Response<T1>] ;

Examinons maintenant une requête GET puis une requête POST utilisant la méthode générique [getResponse]. Tout d'abord une requête [GET] :

```
1.     public AgendaMedecinJour getAgendaMedecinJour(User user, long idMedecin, String jour) {
2.        // l'agenda LinkHashedMap
3.        Object map = this.<AgendaMedecinJour, Void> getResponse(user,
4.              String.format("%s/%s/%s", "/getAgendaMedecinJour", idMedecin, jour), HttpMethod.GET, 110,
    (Void) null);
5.        try {
6.           // l'agenda AgendaMedecinJour
7.           return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<AgendaMedecinJour>() {
8.           });
9.        } catch (Exception e) {
10.          throw new RdvMedecinsException(310, getMessagesForException(e));
11.       }
12. }
```

- ligne 3 : on appelle la méthode générique [getResponse]. Nous l'appelons sous la forme [this.<T1,T2> getResponse(...)] où [T1,T2] sont les deux types paramètres de la méthode [getResponse]. Il n'y a pas obligation à écrire explicitement la valeur des types génériques. Dans ce cas, le compilateur déduit ces types des arguments utilisés pour l'appel de [getResponse]. Ainsi le type [T2] sera déduit du type du 4ième paramètre de [getResponse] et le type [T1] sera le type de la variable recevant le résultat de [getResponse]. Pour des raisons de clarté, j'ai préféré déclarer explicitement la nature des paramètres [T1, T2]. Les cinq paramètres effectifs utilisés ligne 3 sont les suivants :
  - 1 : l'utilisateur ;
  - 2 : l'URL cible ;
  - 3 : la méthode HTTP à utiliser ;
  - 4 : le code d'erreur de l'exception si exception il y a ;
  - 5 : la valeur postée. Ici, il n'y en a pas. On a mis [null] transtypé en type [Void]. [Void] est la classe qui correspond au type primitif [void] ;
- ligne 3 : normalement on aurait du écrire [AgendaMedecinJour agenda=...]. Si on le fait, on a une exception à l'exécution disant qu'on ne peut convertir un type [LinkHashedMap] en type [AgendaMedecinJour]. Donc en fait, la méthode [getResponse] ne nous a pas rendu un type [AgendaMedecinJour] mais un type [LinkHashedMap]. Ce problème s'appelle [type erasure] [http://en.wikipedia.org/wiki/Generics_in_Java#Problems_with_type_erasure] et est lié à l'utilisation de méthodes ou classes génériques ;
- ligne 7 : nous utilisons le sérialiseur / désérialiseur jSON pour créer le type [AgendaMedecinJour] dont nous avons besoin :
  - [*mapper*.writeValueAsString(map)] va créer la chaîne [c] jSON de l'objet récupéré ligne 3,
  - [mapper.readValue([c], new TypeReference<AgendaMedecinJour>() {})] va, à partir de cette chaîne [c], créer un objet de type [AgendaMedecinJour] ;
- lignes 9-11 : si la sérialisation / désérialisation se passe mal, on lance une exception ;

La requête POST examinée sera la suivante :

```
1.  public Rv ajouterRv(User user, String jour, long idCreneau, long idClient) {
2.        // le Rv LinkHashedMap
3.        Object map = this.<Rv, PostAjouterRv> getResponse(user, "/ajouterRv", HttpMethod.POST, 108, new
    PostAjouterRv(
4.              idClient, idCreneau, jour));
5.        try {
6.           // le Rv Rv
7.           return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<Rv>() {
8.           });
9.        } catch (Exception e) {
10.          throw new RdvMedecinsException(308, getMessagesForException(e));
11.       }
12.    }
```

- ligne 3 : la méthode [getResponse] est appelée avec pour types paramètres [Rv, PostAjouterRv]. [Rv] est le type du résultat attendu et [PostAjouterRv] le type de la valeur postée ;
- ligne 3 : la méthode [getResponse] est appelée avec les cinq paramètres suivants :
  - 1 : l'utilisateur ;
  - 2 : l'URL cible,
  - 3 : la méthode HTTP à utiliser, ici POST,
  - 4 : le code d'erreur à utiliser en cas d'exception de communication avec le serveur,
  - 5 : la valeur postée ;
- ligne 7 : comme précédemment, nous avons reçu en ligne 3, non pas un type [Rv] mais un type [LinkHashedMap]. Nous le transformons en type [Rv] ;
- lignes 9-11 : gestion de l'erreur de sérialisation / désérialisation jSON ;

Les autres méthodes sont construites sur le même modèle. Voici le reste de la classe [Dao] :

```
1.  public List<Client> getAllClients(User user) {
2.        // la liste des clients List<LinkedHashMap>
3.        Object map = this.<List<Client>, Void> getResponse(user, "/getAllClients", HttpMethod.GET, 100,
    null);
4.        try {
5.           // la liste des clients List<Client>
6.           return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<List<Client>>() {
7.           });
8.        } catch (Exception e) {
9.           throw new RdvMedecinsException(300, getMessagesForException(e));
10.       }
11.    }
```

```
12.
13.     public List<Medecin> getAllMedecins(User user) {
14.         // la liste des médecins List<LinkedHashMap>
15.         Object map = this.<List<Medecin>, Void> getResponse(user, "/getAllMedecins", HttpMethod.GET, 101,
    null);
16.         try {
17.             // la liste des médecins List<Medecin>
18.             return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<List<Medecin>>() {
19.             });
20.         } catch (Exception e) {
21.             throw new RdvMedecinsException(301, getMessagesForException(e));
22.         }
23.     }
24.
25.     public List<Creneau> getAllCreneaux(User user, long idMedecin) {
26.         // la liste des créneaux List<LinkedHashMap>
27.         Object map = this.<List<Creneau>, Void> getResponse(user, String.format("%s/%s", "/getAllCreneaux",
    idMedecin),
28.                 HttpMethod.GET, 102, (Void) null);
29.         try {
30.             // la liste des créneaux List<Creneau>
31.             return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<List<Creneau>>() {
32.             });
33.         } catch (Exception e) {
34.             throw new RdvMedecinsException(302, getMessagesForException(e));
35.         }
36.     }
37.
38.     public List<Rv> getRvMedecinJour(User user, long idMedecin, String jour) {
39.         // la liste des Rv List<LinkedHashMap>
40.         Object map = this.<List<Rv>, Void> getResponse(user,
41.                 String.format("%s/%s/%s", "/getRvMedecinJour", idMedecin, jour), HttpMethod.GET, 103, (Void)
    null);
42.         try {
43.             // la liste des Rv List<Rv>
44.             return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<List<Rv>>() {
45.             });
46.         } catch (Exception e) {
47.             throw new RdvMedecinsException(303, getMessagesForException(e));
48.         }
49.     }
50.
51.     public Client getClientById(User user, long id) {
52.         // le client LinkHashedMap
53.         Object map = this.<Client, Void> getResponse(user, String.format("%s/%s", "/getClientById", id),
    HttpMethod.GET,
54.                 104, (Void) null);
55.         try {
56.             // le client Client
57.             return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<Client>() {
58.             });
59.         } catch (Exception e) {
60.             throw new RdvMedecinsException(304, getMessagesForException(e));
61.         }
62.     }
63.
64.     public Medecin getMedecinById(User user, long id) {
65.         // le médecin LinkHashedMap
66.         Object map = this.<Medecin, Void> getResponse(user, String.format("%s/%s", "/getMedecinById", id),
    HttpMethod.GET,
67.                 105, (Void) null);
68.         try {
69.             // le médecin Medecin
70.             return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<Medecin>() {
71.             });
72.         } catch (Exception e) {
73.             throw new RdvMedecinsException(305, getMessagesForException(e));
74.         }
75.     }
76.
77.     public Rv getRvById(User user, long id) {
78.         // le Rv LinkHashedMap
79.         Object map = this.<Rv, Void> getResponse(user, String.format("%s/%s", "/getRvById", id),
    HttpMethod.GET, 106,
80.                 (Void) null);
```

```
81.         try {
82.            // le Rv Rv
83.            return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<Rv>() {
84.            });
85.         } catch (Exception e) {
86.            throw new RdvMedecinsException(306, getMessagesForException(e));
87.         }
88.     }
89.
90.     public Creneau getCreneauById(User user, long id) {
91.        // le créneau LinkHashedMap
92.        Object map = this.<Creneau, Void> getResponse(user, String.format("%s/%s", "/getCreneauById", id),
    HttpMethod.GET,
93.              107, (Void) null);
94.        try {
95.           // le créneau Creneau
96.           return mapper.readValue(mapper.writeValueAsString(map), new TypeReference<Creneau>() {
97.           });
98.        } catch (Exception e) {
99.           throw new RdvMedecinsException(307, getMessagesForException(e));
100.       }
101.    }
102.
103.    public void supprimerRv(User user, long idRv) {
104.       this.<Void, PostSupprimerRv> getResponse(user, "/supprimerRv", HttpMethod.POST, 109, new
    PostSupprimerRv(idRv));
105.    }
106.
107.    // liste des messages d'erreur d'une exception
108.    private static List<String> getMessagesForException(Exception exception) {
109.       // on récupère la liste des messages d'erreur de l'exception
110.       Throwable cause = exception;
111.       List<String> erreurs = new ArrayList<String>();
112.       while (cause != null) {
113.          // on récupère le message seulement s'il est !=null et non blanc
114.          String message = cause.getMessage();
115.          if (message != null) {
116.             message = message.trim();
117.             if (message.length() != 0) {
118.                erreurs.add(message);
119.             }
120.          }
121.          // cause suivante
122.          cause = cause.getCause();
123.       }
124.       return erreurs;
125.    }
```

Il nous reste à implémenter la méthode [authenticate] :

```
1.      // vérification [user,mdp]
2.      public void authenticate(User user) {
3.         this.<Void, Void> getResponse(user, "/authenticate", HttpMethod.GET, 111, (Void) null);
4.   }
```

- ligne 3 : on voit qu'on interroge l'URL [/authenticate] du serveur web / jSON. Pour l'instant, elle n'existe pas. Nous l'avons rajouté :

```
1.      @RequestMapping(value = "/authenticate", method = RequestMethod.GET)
2.      public Response<Void> authenticate(HttpServletResponse response,
3.            @RequestHeader(value = "Origin", required = false) String origin) {
4.         // entêtes CORS
5.         rdvMedecinsCorsController.sendOptions(origin, response);
6.         // réponse
7.         return new Response<Void>(0, null, null);
8.   }
```

- ligne 1 : l'URL [/authenticate] ;
- ligne 7 : l'action [/authenticate] renvoie une réponse vide avec [status=0] et un corps vide ;

Le raisonnement est le suivant : pour arriver jusqu'à l'action [/authenticate], la requête du client va devoir passer le barrage de l'authentification. Si celle-ci réussit, on renvoie une réponse vide avec [status=0]. Si l'autentification échoue, la requête n'arrivera

jamais à l'action [/authenticate] et le serveur répondra avec un statut HTTP [403 forbidden] ou [401 Unauthorized] , ce qui côté client va provoquer une exception. Donc si la méthode [authenticate] :

- renvoie une exception, cela signifie que l'authentification a échoué ;
- ne renvoie pas d'exception, cela signifie que l'authentification a réussi ;

8.5.11 Anomalie

En faisant divers tests j'ai rencontré l'anomalie suivante que je ne m'expliquais pas. Je l'ai résumée dans la classe [Anomalie] suivante :

```
1.  package rdvmedecins.clients.console;
2.
3.  import java.io.IOException;
4.
5.  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6.
7.  import rdvmedecins.client.config.DaoConfig;
8.  import rdvmedecins.client.dao.IDao;
9.  import rdvmedecins.client.dao.RdvMedecinsException;
10. import rdvmedecins.client.entities.User;
11.
12. import com.fasterxml.jackson.core.JsonProcessingException;
13. import com.fasterxml.jackson.databind.ObjectMapper;
14.
15. public class Anomalie {
16.
17.    // sérialiseur jSON
18.    static private ObjectMapper mapper = new ObjectMapper();
19.    // timeout des connexions en millisecondes
20.    static private int TIMEOUT = 1000;
21.
22.    public static void main(String[] args) throws IOException {
23.       // on récupère une référence sur la couche [DAO]
24.       AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(DaoConfig.class);
25.       IDao dao = context.getBean(IDao.class);
26.       // on fixe l'URL du service web / json
27.       dao.setUrlServiceWebJson("http://localhost:8080");
28.       // on fixe les timeout en millisecondes
29.       dao.setTimeout(TIMEOUT);
30.
31.       // Authentification
32.       String message = "/authenticate [admin,admin]";
33.       try {
34.          dao.authenticate(new User("admin", "admin"));
35.          System.out.println(String.format("%s : OK", message));
36.       } catch (RdvMedecinsException e) {
37.          showException(message, e);
38.       }
39.
40.       // Authentification
41.       message = "/authenticate [admin,x]";
42.       try {
43.          dao.authenticate(new User("admin", "x"));
44.          System.out.println(String.format("%s : OK", message));
45.       } catch (RdvMedecinsException e) {
46.          showException(message, e);
47.       }
48.
49.       // Authentification
50.       message = "/authenticate [user,user]";
51.       try {
52.          dao.authenticate(new User("user", "user"));
53.          System.out.println(String.format("%s : OK", message));
54.       } catch (RdvMedecinsException e) {
55.          showException(message, e);
56.       }
57.
58.       // fermeture contexte
59.       context.close();
60.    }
61.
62.    private static void showException(String message, RdvMedecinsException e) {
```

```
63.         System.out.println(String.format("URL [%s]", message));
64.         System.out.println(String.format("L'erreur n° [%s] s'est produite :", e.getStatus()));
65.         for (String msg : e.getMessages()) {
66.             System.out.println(msg);
67.         }
68.     }
69. }
```

- lignes 31-38 : on authentifie l'utilisateur [admin, admin] ;
- lignes 40-47 : on authentifie l'utilisateur [admin, x] qui a donc un mot de passe erroné ;
- lignes 49-56 : on authentifie l'utilisateur [user, user] qui est un utilisateur existant mais non autorisé ;

Voici les résultats :

```
1.  /authenticate [admin,admin] : OK
2.  /authenticate [admin,x] : OK
3.  URL [/authenticate [user,user]]
4.  L'erreur n° [111] s'est produite :
5.  403 Forbidden
```

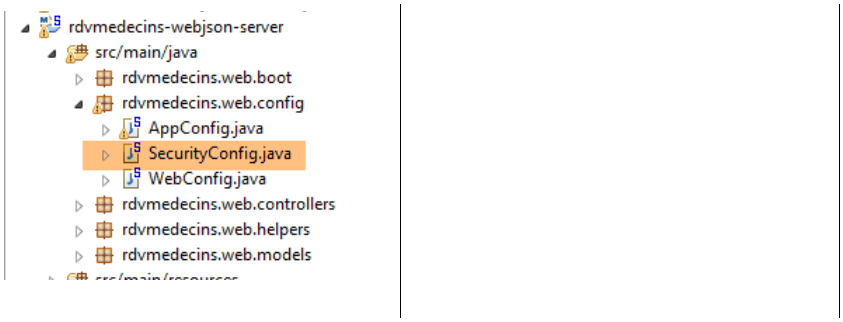
- ligne 2 : contre toute attente, l'utilisateur [admin, x] a été accepté ;

Si on passe les lignes 33-38 du code en commentaires, on obtient le résultat suivant :

```
1.  URL [/authenticate [admin,x]]
2.  L'erreur n° [111] s'est produite :
3.  401 Unauthorized
4.  URL [/authenticate [user,user]]
5.  L'erreur n° [111] s'est produite :
6.  403 Forbidden
```

ce qui est le résultat attendu. Tout se passe comme si lorsque l'utilisateur [admin, admin] s'est identifié avec succès une 1ère fois, son mot de passe n'était plus nécessaire pour les fois suivantes. Pour le moins bizarre... Ce phénomène est reproductible avec les autres méthodes de la couche [DAO].

J'ai pensé que c'était peut-être un phénomène de session : Spring Security utiliserait une session qui ferait qu'une fois qu'un utilisateur s'était authentifié, il n'avait plus besoin de le refaire dans les requêtes suivantes. En cherchant le terme [Spring Security session] sur la toile j'ai trouvé qu'effectivement [Spring Security] gérait une session. J'ai donc modifié la configuration de [Spring Security] dans le serveur web / jSON pour que ce ne soit plus le cas :



Le fichier [SecurityConfig] doit être modifié de la façon suivante :

```
1.      @Override
2.      protected void configure(HttpSecurity http) throws Exception {
3.          ...
4.              // pas de session
5.              http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
6.  }
```

- la ligne 5 demande à ce qu'il n'y ait pas de session de sécurité ;

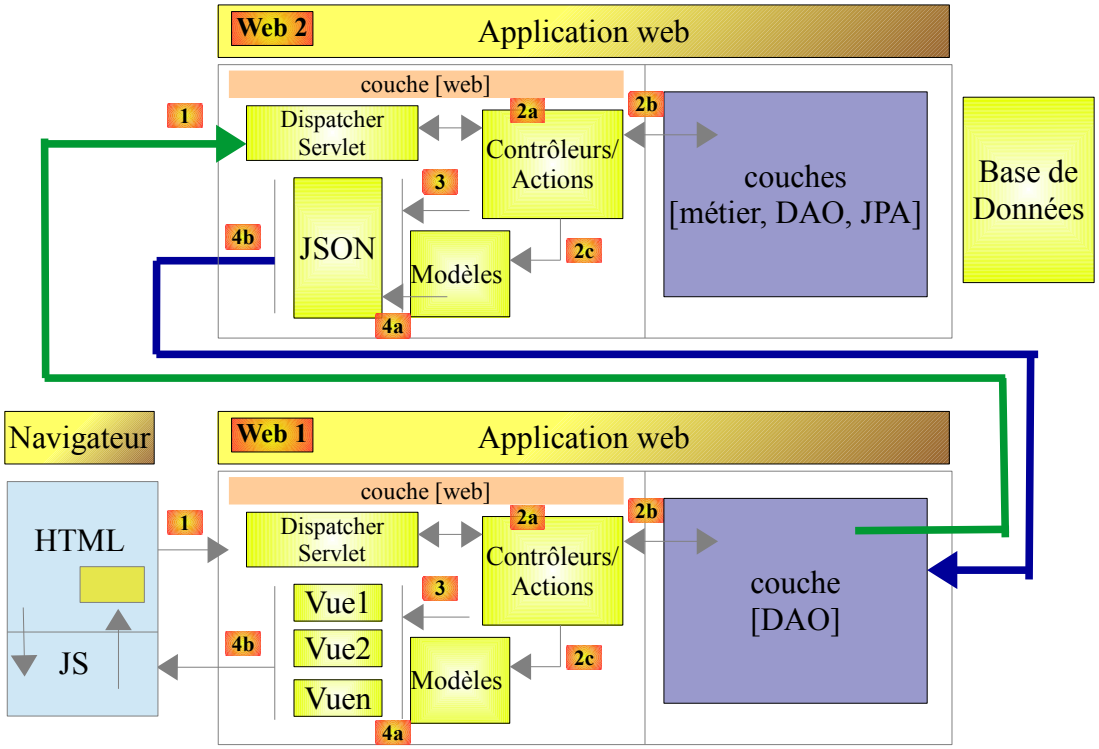
Cela a résolu le problème de l'anomalie.

Note : je ne sais pas quelle est la bonne pratique : utiliser ou non une session.

8.6 Ecriture du serveur Spring / Thymeleaf

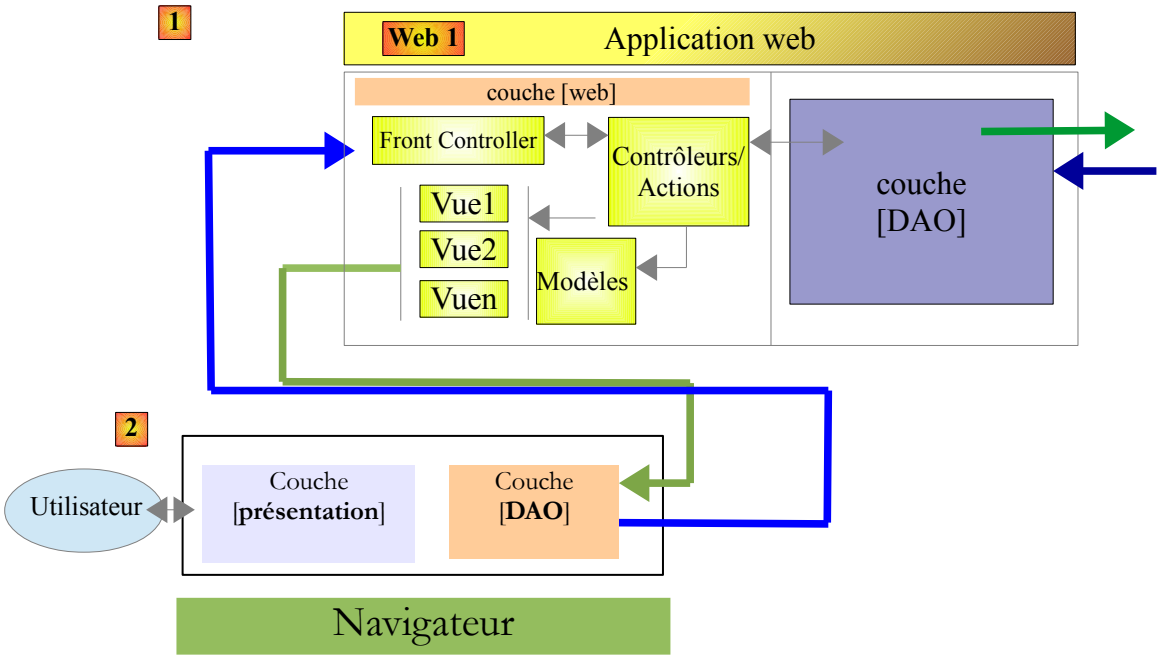
8.6.1 Introduction

Revenons à l'architecture de l'application client / serveur à construire :



- le serveur [Web2] web / jSON a été construit ;
- le couche [DAO] du client [Web1] a été construite ;

La relation entre le serveur [Web1] et les navigateurs clients est une relation client / serveur où le serveur est un serveur web / jSON. En effet, [Web1] va délivrer des flux HTML encapsulés dans une chaîne jSON. L'architecture client / serveur est la suivante :



- on a une architecture client [2] / serveur [1] où le client et le serveur communiquent en jSON ;
- en [1], la couche web Spring MVC / Thymeleaf délivre des vues, des fragments de vue, des données dans du jSON. Le serveur est donc un serveur web / jSON comme le serveur [Web1]. Il est lui aussi sans état ;
- en [2] : le code Javascript embarqué dans la vue chargée au démarrage de l'application est structuré en couches :
    - la couche [présentation] s'occupe des interactions avec l'utilisateur,
    - la couche [DAO] s'occupe de l'accès aux données via le serveur [Web2] ;
- le client [2] mettra certaines vues en cache afin de soulager le serveur ;

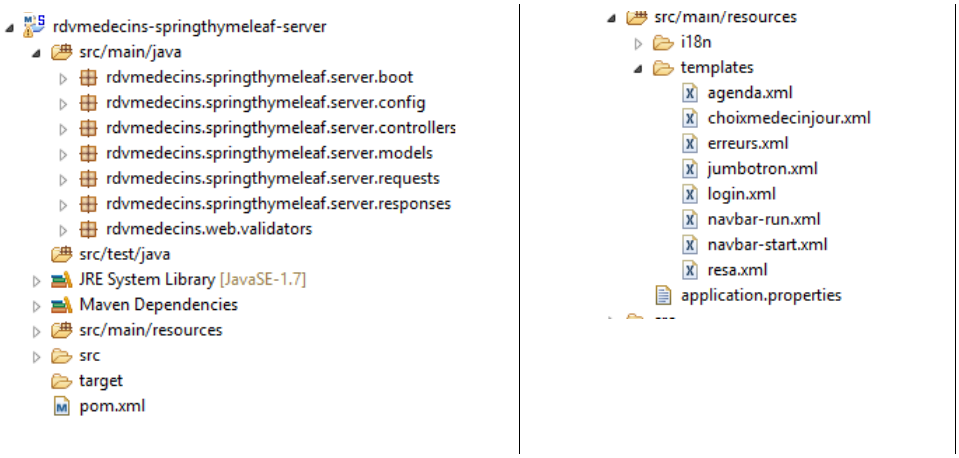
Nous allons construire le serveur web / jSON [Web1] implémenté avec Spring MVC / Thymeleaf en plusieurs étapes :

- découverte du framework CSS Bootstrap ;
- écriture des vues ;
- écriture du contrôleur ;

Puis ensuite et à part, nous construirons le client JS du serveur [Web1]. Pour bien montrer que ce client a une certaine indépendance vis à vis du serveur [Web1], nous le construirons avec l'outil [Webstorm] plutôt qu'avec STS.

Dans la suite, certains détails seront ignorés parce qu'ils risqueraient de nous faire oublier l'important qui est l'organisation du code. Le lecteur intéressé pourra trouver le code complet sur le site de ce document.

8.6.2 Le projet STS



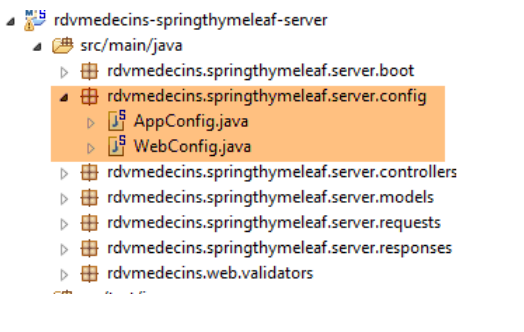
- en [1], les codes Java ;
- en [2], les vues ;

La configuration Maven dans [pom.xml] est la suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.     <groupId>istia.st.rdvmedecins</groupId>
6.     <artifactId>rdvmedecins-springthymeleaf-server</artifactId>
7.     <version>0.0.1-SNAPSHOT</version>
8.     <name>rdvmedecins-springthymeleaf-server</name>
9.     <description>Gestion de RV Médecins</description>
10.    <parent>
11.       <groupId>org.springframework.boot</groupId>
12.       <artifactId>spring-boot-starter-parent</artifactId>
13.       <version>1.2.0.RELEASE</version>
14.    </parent>
15.    <dependencies>
16.       <dependency>
17.          <groupId>org.springframework.boot</groupId>
18.          <artifactId>spring-boot-starter-thymeleaf</artifactId>
19.       </dependency>
20.       <dependency>
21.          <groupId>istia.st.rdvmedecins</groupId>
22.          <artifactId>rdvmedecins-webjson-client-console</artifactId>
23.          <version>0.0.1-SNAPSHOT</version>
24.       </dependency>
25.    </dependencies>
26.    <properties>
27.       <start-class>rdvmedecins.springthymeleaf.server.boot.Boot</start-class>
28.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
29.       <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
30.       <java.version>1.7</java.version>
31.    </properties>
32.    <build>
33.       <plugins>
34.          <plugin>
35.             <artifactId>maven-compiler-plugin</artifactId>
36.             <configuration>
37.                <source>1.7</source>
38.                <target>1.7</target>
39.             </configuration>
40.          </plugin>
41.          <plugin>
42.             <groupId>org.springframework.boot</groupId>
43.             <artifactId>spring-boot-maven-plugin</artifactId>
44.          </plugin>
45.       </plugins>
46.    </build>
47.    ...
48. </project>
```

- lignes 16-19 : le projet est un projet Thymeleaf ;
- lignes 20-24 : qui s'appuie sur la couche [DAO] que nous venons de construire ;

La configuration Java est assurée par deux fichiers :



La couche [web] est configurée par le fichier [WebConfig] suivant :

```
1.  package rdvmedecins.springthymeleaf.server.config;
2.
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.context.MessageSource;
5.  import org.springframework.context.annotation.Bean;
6.  import org.springframework.context.support.ResourceBundleMessageSource;
7.  import org.springframework.web.servlet.DispatcherServlet;
8.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
9.  import org.thymeleaf.spring4.SpringTemplateEngine;
10. import org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver;
11.
12. @EnableAutoConfiguration
13. public class WebConfig extends WebMvcConfigurerAdapter {
14.
15.    // ----------------- configuration couche [web]
16.    @Bean
17.    public MessageSource messageSource() {
18.       ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
19.       messageSource.setBasename("i18n/messages");
20.       return messageSource;
21.    }
22.
23.    @Bean
24.    public SpringResourceTemplateResolver templateResolver() {
25.       SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
26.       templateResolver.setPrefix("classpath:/templates/");
27.       templateResolver.setSuffix(".xml");
28.       templateResolver.setTemplateMode("HTML5");
29.       templateResolver.setCacheable(true);
30.       templateResolver.setCharacterEncoding("UTF-8");
31.       return templateResolver;
32.    }
33.
34.    @Bean
35.    SpringTemplateEngine templateEngine(SpringResourceTemplateResolver templateResolver) {
36.       SpringTemplateEngine templateEngine = new SpringTemplateEngine();
37.       templateEngine.setTemplateResolver(templateResolver);
38.       return templateEngine;
39.    }
40.
41.    // configuration dispatcherservlet pour les headers CORS
42.    @Bean
43.    public DispatcherServlet dispatcherServlet() {
44.       DispatcherServlet servlet = new DispatcherServlet();
45.       servlet.setDispatchOptionsRequest(true);
46.       return servlet;
47.    }
48.
49. }
```

Nous avons rencontré, à un moment ou à un autre, tous les éléments de cette configuration. Rappelons simplement que les lignes 42-47 sont nécessaires lorsqu'on veut pouvoir interroger le serveur avec des requêtes inter-domaines (CORS). Cela va être le cas ici.

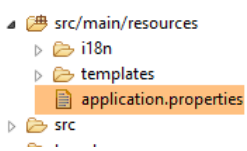
La classe [AppConfig] configure l'ensemble de l'application :

```
1.  package rdvmedecins.springthymeleaf.server.config;
2.
```

```
3.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
4.  import org.springframework.context.annotation.ComponentScan;
5.  import org.springframework.context.annotation.Import;
6.
7.  import rdvmedecins.client.config.DaoConfig;
8.
9.  @EnableAutoConfiguration
10. @ComponentScan(basePackages = { "rdvmedecins.springthymeleaf.server" })
11. @Import({ WebConfig.class, DaoConfig.class })
12. public class AppConfig {
13.
14.     // admin / admin
15.     private final String USER_INIT = "admin";
16.     private final String MDP_USER_INIT = "admin";
17.     // racine service web / json
18.     private final String WEBJSON_ROOT = "http://localhost:8080";
19.     // timeout en millisecondes
20.     private final int TIMEOUT = 5000;
21.     // CORS
22.     private final boolean CORS_ALLOWED=true;
23.
24.     ...
25.
26. }
```

- lignes 11 : [AppConfig] importe la configuration de la couche [DAO] et de la couche [web] ;
- lignes 15-16 : les identifiants qui vont permettre à l'application de faire un accès au boot de l'application afin de mettre en cache les médecins et les clients ;
- ligne 18 : l'URL du service web / jSON [Web1] ;
- ligne 20 : le *timeout* des appels HTTP de l'application ;
- ligne 22 : un booléen pour autoriser ou non les appels inter-domaines ;

Enfin dans [application.properties], le serveur Tomcat est configuré pour travailler sur le port 8081 :

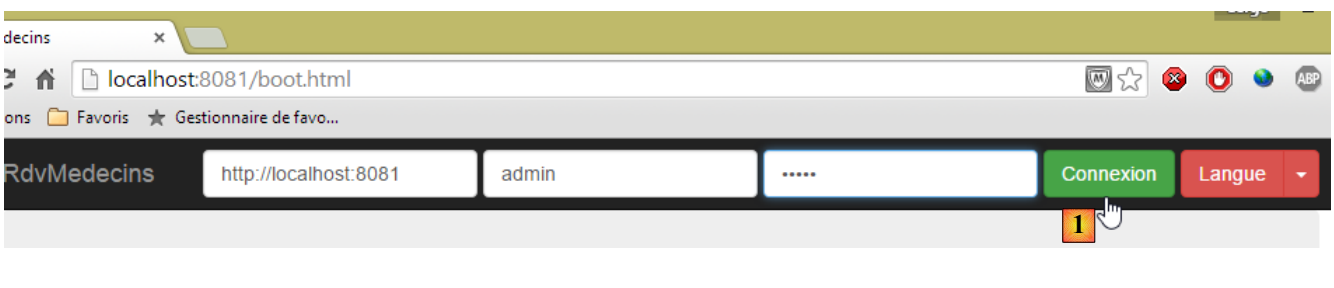



```
server.port=8081
```

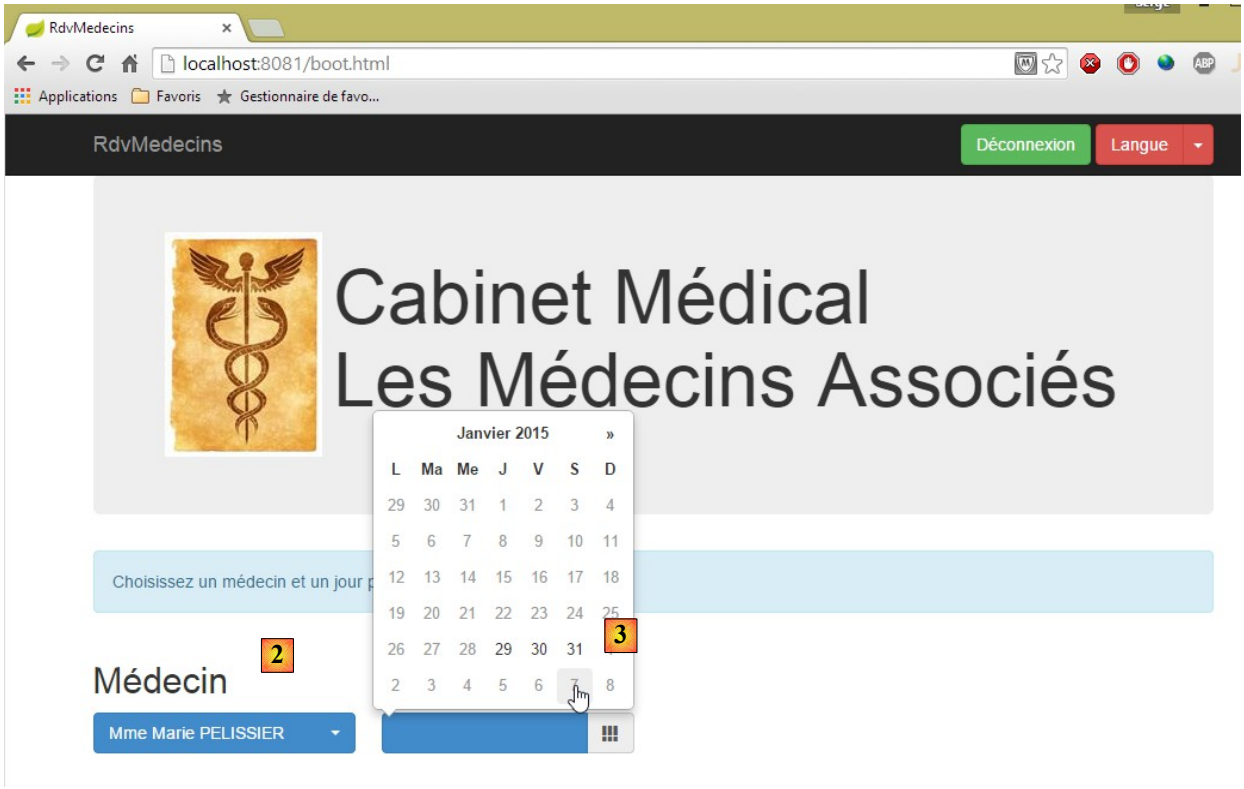
8.6.3 Les fonctionnalités de l'application

Elles ont été décrites au paragraphe 8.2, page 334. Nous les rappelons maintenant. Avec un navigateur, on demande l'URL [http://localhost:8081/boot.html] :

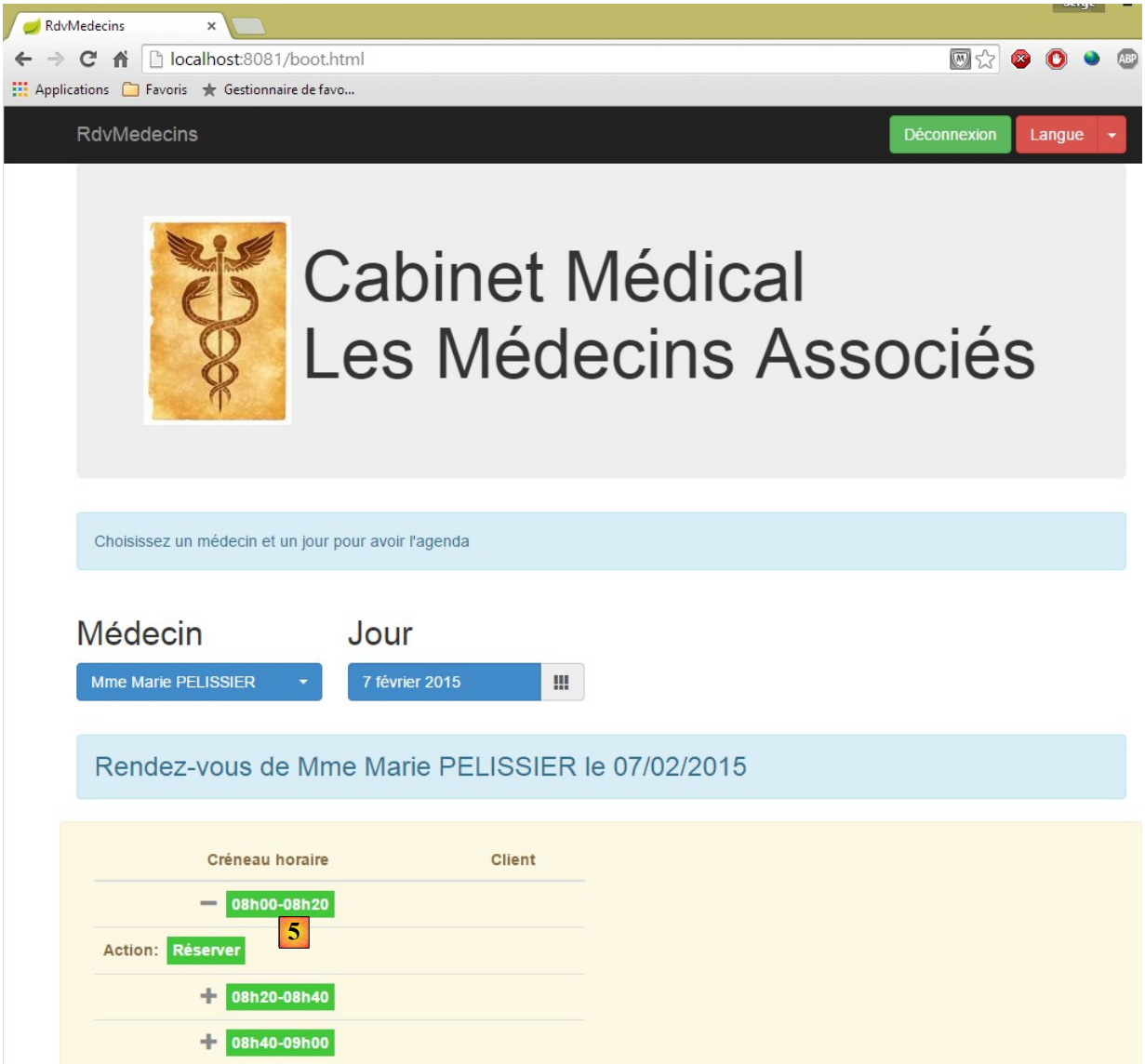
- en [1], la page d'entrée de l'application ;
- en [2] et [3], l'identifiant et le mot de passe de celui qui veut utiliser l'application. Il y a deux utilisateurs : **admin/admin** (login/password) avec un rôle (ADMIN) et **user/user** avec un rôle (USER). Seul le rôle ADMIN a le droit d'utiliser l'application. Le rôle USER n'est là que pour montrer ce que répond le serveur dans ce cas d'utilisation ;
- en [4], le bouton qui permet de se connecter au serveur ;
- en [5], la langue de l'application. Il y en a deux : le français par défaut et l'anglais ;
- en [6], l'URL du serveur [rdvmedecins-springthymeleaf-server] ;



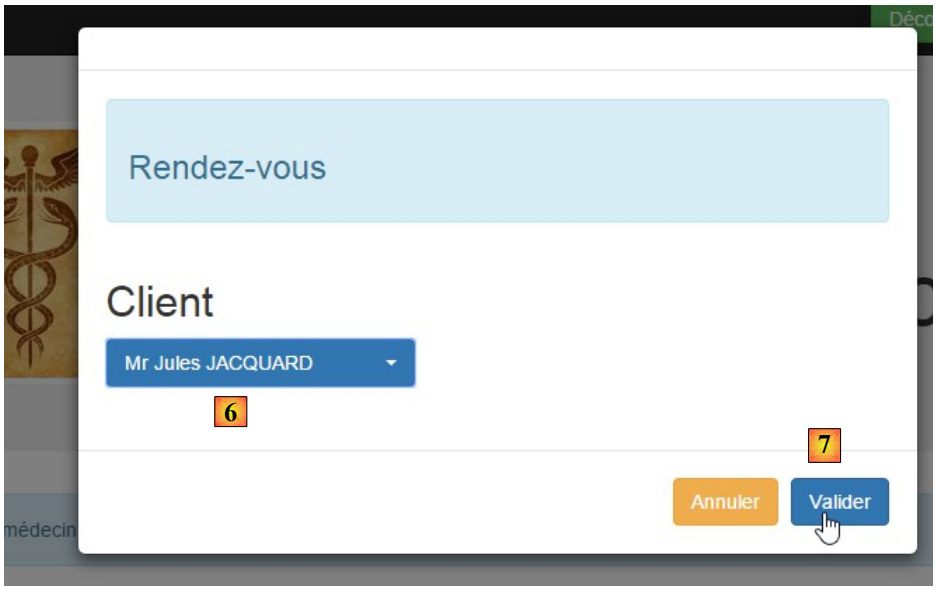
- en [1], on se connecte ;



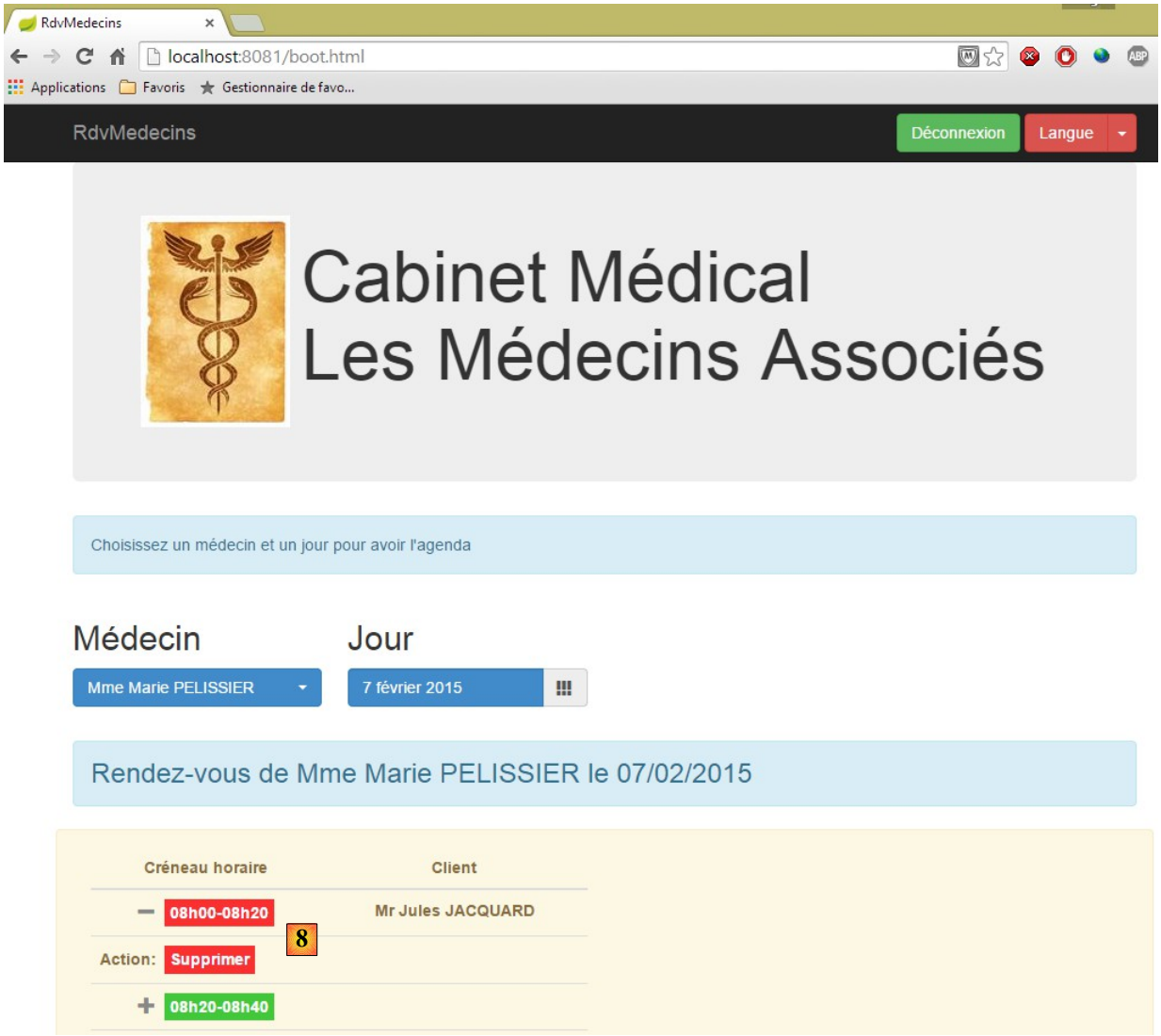
- une fois connecté, on peut choisir le médecin avec lequel on veut un rendez-vous [2] et le jour de celui-ci [3]. Dès qu'un médecin et un jour ont été renseignés, l'agenda est automatiquement affiché :



- une fois obtenu l'agenda du médecin, on peut réserver un créneau [5] ;



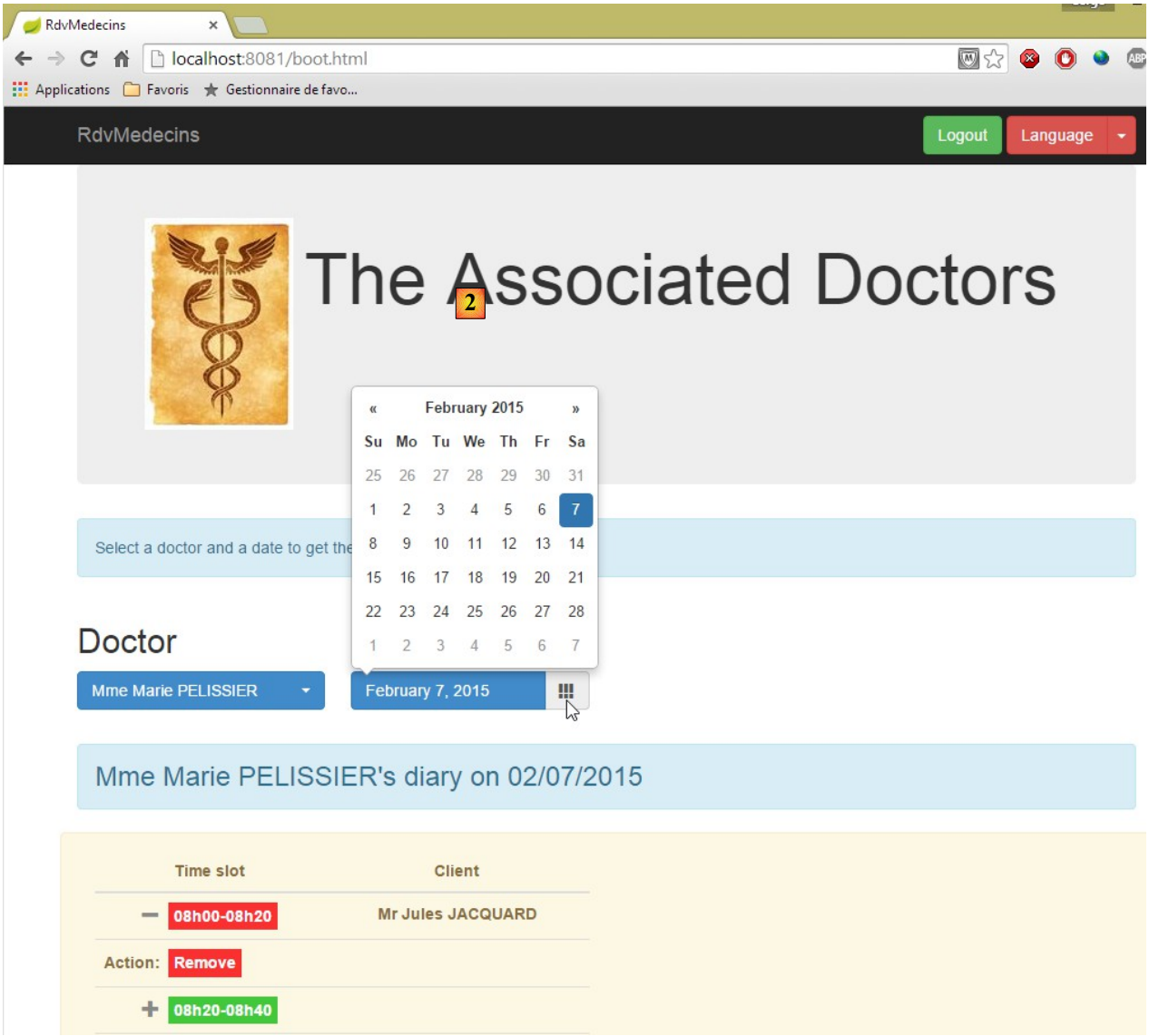
- en [6], on choisit le patient pour le rendez-vous et on valide ce choix en [7] ;



Une fois le rendez-vous validé, on est ramené automatiquement à l'agenda où le nouveau rendez-vous est désormais inscrit. Ce rendez-vous pourra être ultérieurement supprimé [8].

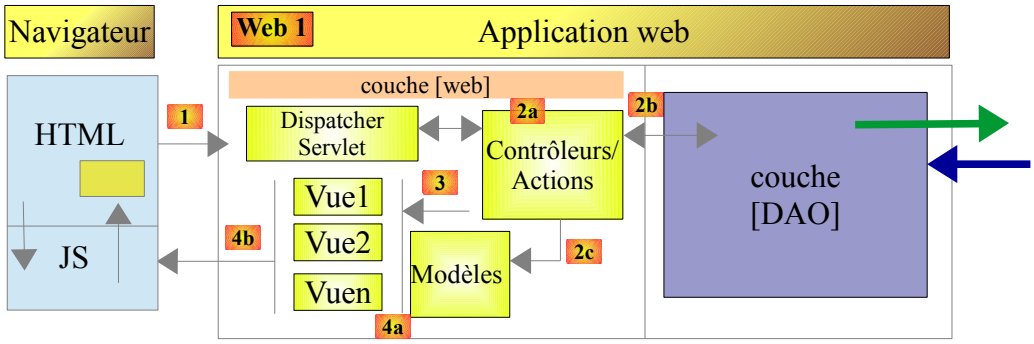
Les principales fonctionnalités ont été décrites. Elles sont simples. Terminons par la gestion de la langue :

- en [1], on passe du français à l'anglais ;



- en [2], la vue est passée en anglais, y-compris le calendrier ;

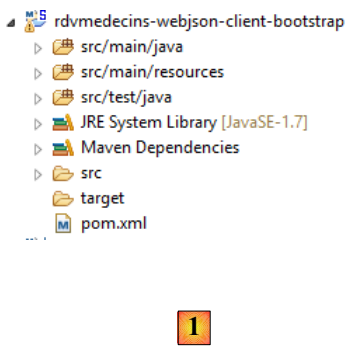
8.6.4 Étape 1 : introduction au framework CSS Bootstrap

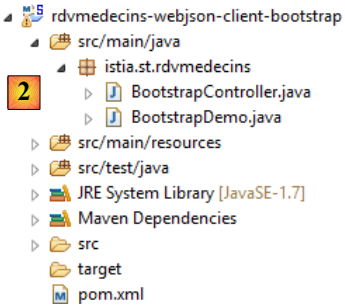


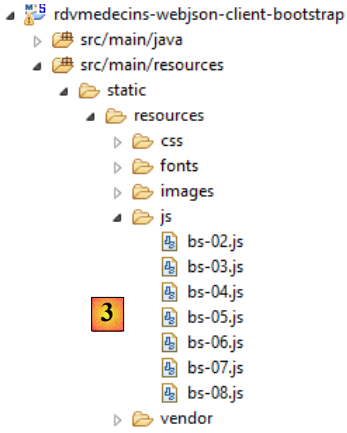
Dans le client web ci-dessus, les pages HTML vont utiliser le framework CSS Bootstrap [http://getbootstrap.com/] que nous présentons maintenant.

8.6.4.1 Le projet des exemples

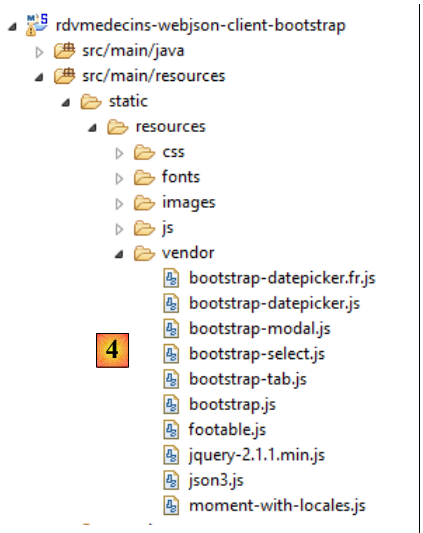
Le projet des exemples sera le suivant :

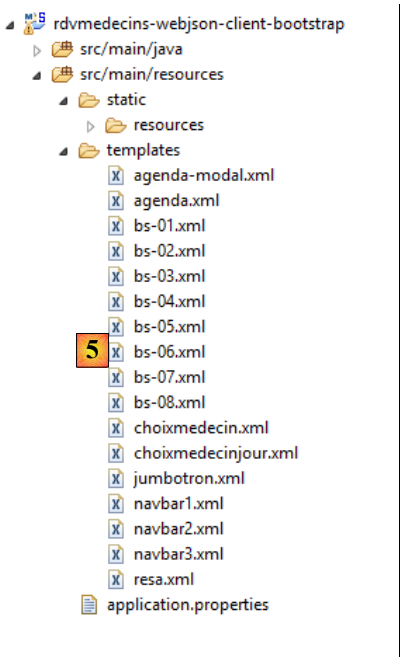


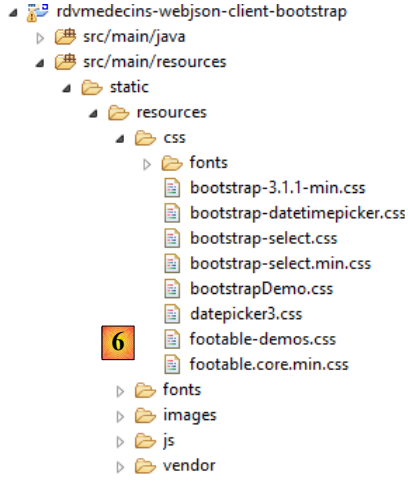




- en [1] : le projet dans sa globalité ;
- en [2] : les codes Java ;
- en [3] : les scripts Javascript ;







- en [4] : les bibliothèques Javascript ;
- en [5] : les vues Thymeleaf ;
- en [6] : les feuilles de style ;

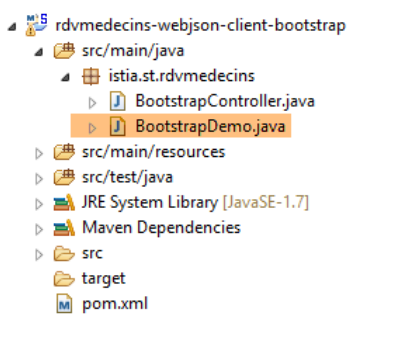
8.6.4.1.1 Configuration Maven

Le fichier [pom.xml] est celui d'un projet Maven Thymeleaf :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.
6.     <groupId>istia.st</groupId>
7.     <artifactId>rdvmedecins-webjson-client-bootstrap</artifactId>
8.     <version>0.0.1-SNAPSHOT</version>
9.     <packaging>jar</packaging>
10.
11.    <name>rdvmedecins-webjson-client-bootstrap</name>
12.    <description>Démos Bootstrap</description>
13.
14.    <parent>
15.       <groupId>org.springframework.boot</groupId>
16.       <artifactId>spring-boot-starter-parent</artifactId>
17.       <version>1.2.0.RELEASE</version>
18.       <relativePath /> <!-- lookup parent from repository -->
19.    </parent>
20.
21.    <properties>
22.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23.       <start-class>istia.st.rdvmedecins.BootstrapDemo</start-class>
24.       <java.version>1.7</java.version>
25.    </properties>
26.
27.    <dependencies>
28.       <dependency>
29.          <groupId>org.springframework.boot</groupId>
30.          <artifactId>spring-boot-starter-thymeleaf</artifactId>
31.       </dependency>
32.    </dependencies>
33.
34.    <build>
35.       <plugins>
36.          <plugin>
37.             <groupId>org.springframework.boot</groupId>
38.             <artifactId>spring-boot-maven-plugin</artifactId>
39.          </plugin>
```

```
40.         </plugins>
41.     </build>
42.
43. </project>
```

8.6.4.1.2 Configuration Java

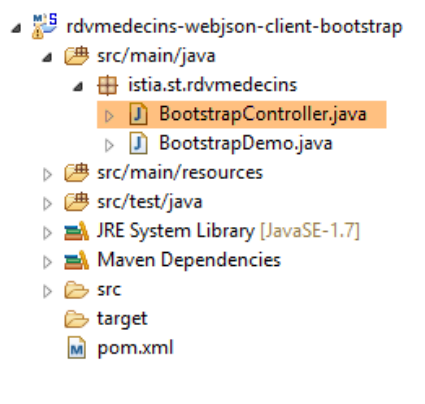


La classe [BootstrapDemo] configure l'application Spring / Thymeleaf :

```
1.  package istia.st.rdvmedecins;
2.
3.  import org.springframework.boot.SpringApplication;
4.  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5.  import org.springframework.context.annotation.Bean;
6.  import org.springframework.context.annotation.ComponentScan;
7.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
8.  import org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver;
9.
10. @EnableAutoConfiguration
11. @ComponentScan({ "istia.st.rdvmedecins" })
12. public class BootstrapDemo extends WebMvcConfigurerAdapter {
13.
14.     public static void main(String[] args) {
15.         SpringApplication.run(BootstrapDemo.class, args);
16.     }
17.
18.     @Bean
19.     public SpringResourceTemplateResolver templateResolver() {
20.         SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
21.         templateResolver.setPrefix("classpath:/templates/");
22.         templateResolver.setSuffix(".xml");
23.         templateResolver.setTemplateMode("HTML5");
24.         templateResolver.setCacheable(true);
25.         templateResolver.setCharacterEncoding("UTF-8");
26.         return templateResolver;
27.     }
28. }
```

Nous avons déjà rencontré ce type de code.

8.6.4.1.3 Le contrôleur Spring



Le contrôleur [BootstrapController] est le suivant :

```
1.  package istia.st.rdvmedecins;
2.
3.  import org.springframework.stereotype.Controller;
4.  import org.springframework.web.bind.annotation.RequestMapping;
5.  import org.springframework.web.bind.annotation.RequestMethod;
6.
7.  @Controller
8.  public class BootstrapController {
9.
10.    @RequestMapping(value = "/bs-01", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
11.    public String bso1() {
12.       return "bs-01";
13.    }
14.
15.    @RequestMapping(value = "/bs-02", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
16.    public String bs02() {
17.       return "bs-02";
18.    }
19.
20.    @RequestMapping(value = "/bs-03", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
21.    public String bs03() {
22.       return "bs-03";
23.    }
24.
25.    @RequestMapping(value = "/bs-04", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
26.    public String bs04() {
27.       return "bs-04";
28.    }
29.
30.    @RequestMapping(value = "/bs-05", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
31.    public String bs05() {
32.       return "bs-05";
33.    }
34.
35.    @RequestMapping(value = "/bs-06", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
36.    public String bs06() {
37.       return "bs-06";
38.    }
39.
40.    @RequestMapping(value = "/bs-07", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
41.    public String bs07() {
42.       return "bs-07";
43.    }
44.
45.    @RequestMapping(value = "/bs-08", method = RequestMethod.GET, produces = "text/html; charset=UTF-8")
46.    public String bs08() {
47.       return "bs-08";
48.    }
49. }
```

Les actions ne sont là que pour afficher des vues traitées par Thymeleaf.

8.6.4.1.4 Le fichier [application.properties]

Le fichier [application.properties] configure le serveur Tomcat embarqué :

```
server.port=8082
```

8.6.4.2 *Exemple n° 1 : le jumbotron*

L'action [/bs-01] affiche la vue [bs-01.xml] suivante :

La vue [bs-01.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
9.     </head>
10.    <body id="body">
11.       <div class="container">
12.          <!-- Bootstrap Jumbotron -->
13.          <div th:include="jumbotron"></div>
14.          <!-- contenu -->
15.          <div id="content">
16.             <h1>Ici un contenu</h1>
17.          </div>
18.          <!-- erreur -->
19.          <div id="erreur" class="alert alert-danger">
20.             <span>Ici, un texte d'erreur</span>
21.          </div>
22.       </div>
23.    </body>
24. </html>
```

- ligne 7 : le fichier CSS du framework Bootstrap ;
- ligne 8 : un fichier CSS local ;
- ligne 13 : affiche [1] ;
- lignes 19-21 : affichent [2] ;
- ligne 11 : la classe CSS [container] définit une zone d'affichage à l'intérieur du navigateur ;

- ligne 19 : la classe CSS [alert] affiche une zone colorée. La classe [alert-danger] utilise une couleur prédéfinie. Il en existe plusieurs [alert-info, alert-warning,...] ;

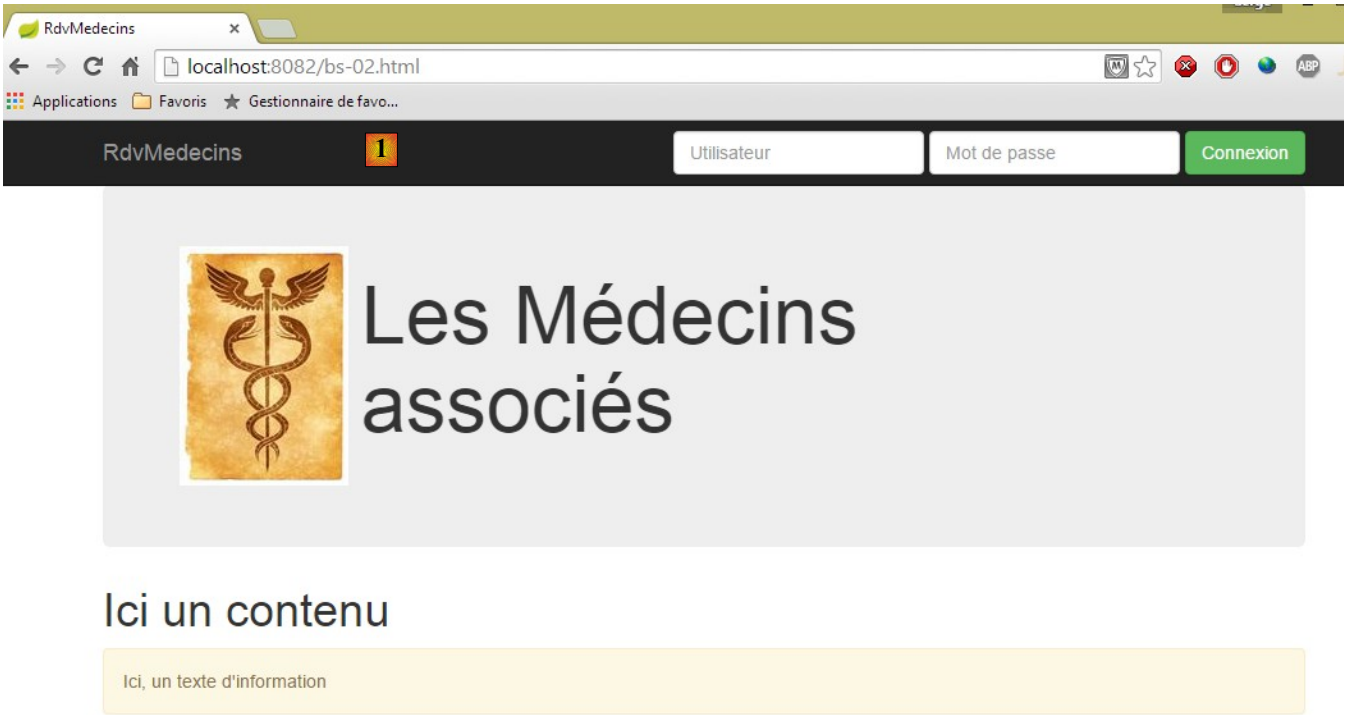
Le jumbotron [1] est généré par la vue [jumbotron.xml] suivante :

```
1. <!DOCTYPE html>
2. <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.    <!-- Bootstrap Jumbotron -->
4.    <div class="jumbotron">
5.       <div class="row">
6.          <div class="col-md-2">
7.             <img src="resources/images/caduceus.jpg" alt="RvMedecins" />
8.          </div>
9.          <div class="col-md-10">
10.            <h1>
11.               Les Médecins
12.               <br />
13.               associés
14.            </h1>
15.         </div>
16.      </div>
17.   </div>
18. </section>
```

- ligne 4 : la zone a la classe CSS [jumbotron] ;
- ligne 5 : la classe [row] définit une ligne à 12 colonnes ;
- ligne 6 : la classe [col-md-2] définit une zone de deux colonnes dans la ligne ;
- ligne 7 : dans ces deux colonnes on met une image ;
- lignes 9-15 : dans les 10 autres colonnes, on met le texte ;

8.6.4.3 Exemple n° 2 : la barre de navigation

L'action [/bs-02] affiche la vue [bs-02.xml] suivante :



La nouveauté est la barre de navigation [1] avec son formulaire de saisie et ses boutons :

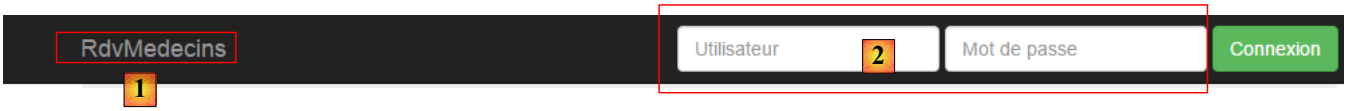
La vue [bs-02.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
9.        <!-- scripts JS -->
10.       <script src="resources/vendor/jquery-2.1.1.min.js"></script>
11.       <script type="text/javascript" src="resources/js/bs-02.js"></script>
12.    </head>
13.    <body id="body">
14.       <div class="container">
15.          <!-- barre de navigation -->
16.          <div th:include="navbar1"></div>
17.          <!-- Bootstrap Jumbotron -->
18.          <div th:include="jumbotron"></div>
19.          <!-- contenu -->
20.          <div id="content">
21.             <h1>Ici un contenu</h1>
22.          </div>
23.          <!-- info -->
24.          <div class="alert alert-warning">
25.             <span id="info">Ici, un texte d'information</span>
26.          </div>
27.       </div>
28.    </body>
29. </html>
```

- ligne 10 : on importe jQuery ;
- ligne 11 : un script JS local ;
- ligne 16 : la barre de navigation ;

La barre de navigation est générée par la vue [navbar1.xml] suivante :

```
1.  <!DOCTYPE HTML>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
4.        <div class="container">
5.           <div class="navbar-header">
6.              <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
    collapse">
7.                 <span class="sr-only">Toggle navigation</span>
8.                 <span class="icon-bar"></span>
9.                 <span class="icon-bar"></span>
10.                <span class="icon-bar"></span>
11.             </button>
12.             <a class="navbar-brand" href="#">RdvMedecins</a>
13.          </div>
14.          <div class="navbar-collapse collapse">
15.             <img id="loading" src="resources/images/loading.gif" alt="waiting..." style="display: none" />
16.             <!-- formulaire d'identification -->
17.             <div class="navbar-form navbar-right" role="form" id="formulaire" method="post">
18.                <div class="form-group">
19.                   <input type="text" placeholder="Utilisateur" class="form-control" />
20.                </div>
21.                <div class="form-group">
22.                   <input type="password" placeholder="Mot de passe" class="form-control" />
23.                </div>
24.                <button type="button" class="btn btn-success"
    onclick="javascript:connecter()">Connexion</button>
25.             </div>
26.          </div>
27.       </div>
28.    </div>
29. </section>
```



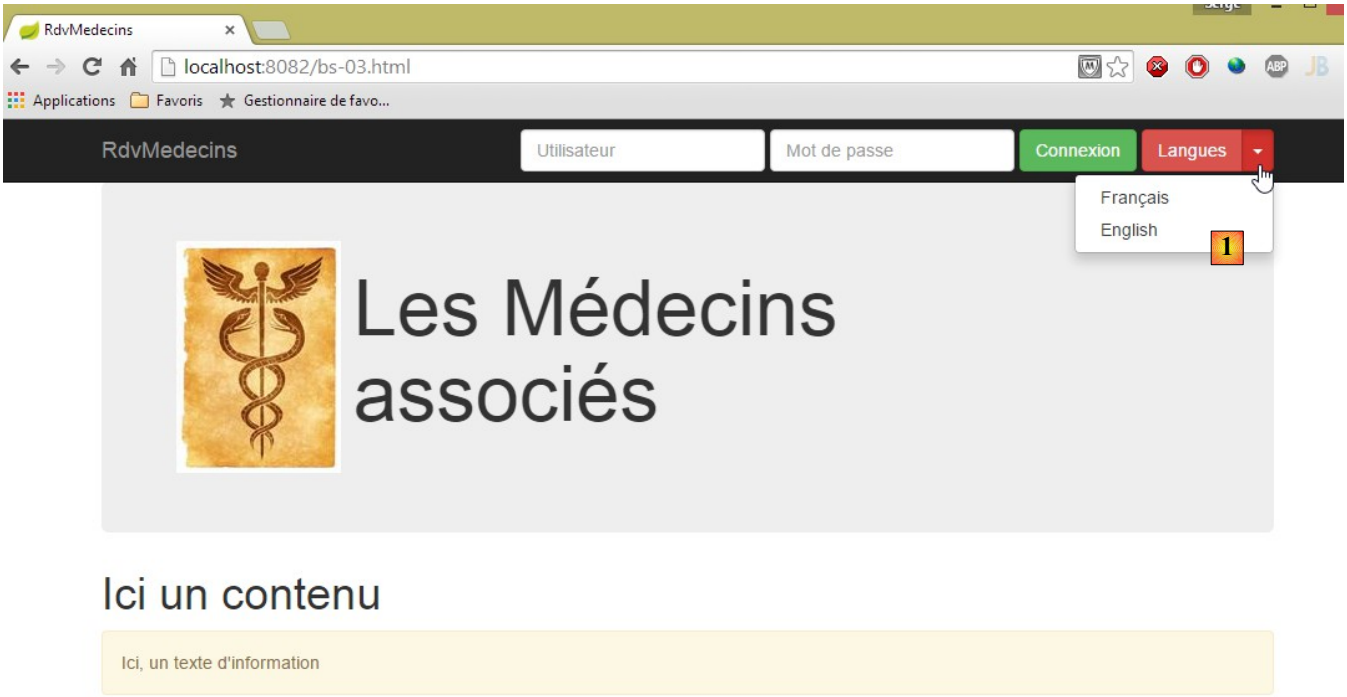
- ligne 3 : la classe [**navbar**] va styler la barre de navigation. La classe [**navbar-inverse**] lui donne le fond noir. La classe [**navbar-fixed-top**] va faire en sorte que lorsqu'on 'scrolle' la page affichée par le navigateur, la barre de navigation va rester en haut de l'écran ;
- lignes 5-13 : définissent la zone [1]. C'est typiquement une série de classes que je ne comprends pas. J'utilise le composant tel quel ;
- lignes 14-26 : définissent une zone 'responsive' de la barre de commande. Sur un smartphone, cette zone disparaît dans une zone de menu ;
- ligne 15 : une image actuellement cachée ;
- lignes 17-25 : la classe [**navbar-form**] habille un formulaire de la barre de commande. La classe [**navbar-right**] le rejette à droite de celle-ci ;
- lignes 21-23 : les deux zones de saisie du formulaire de la ligne 17 [2]. Elles sont à l'intérieur d'une classe [**form-group**] qui habille les éléments d'un formulaire et chacune d'elles a la classe [**form-control**] ;
- ligne 24 : la classe [**btn**] qui définit un bouton, enrichie de la classe [**btn-success**] qui lui donne sa couleur verte ;
- ligne 24 : lorsqu'on clique sur le bouton [Connexion], la fonction JS suivante est exécutée :

```
1.  function connecter() {
2.     showInfo("Connexion demandée...");
3.  }
4.
5.  function showInfo(message) {
6.     $("#info").text(message);
7.  }
```

Voici un exemple :

8.6.4.4 Exemple n° 3 : le bouton à liste

L'action [/bs-03] affiche la vue [bs-03.xml] suivante :



- la nouveauté est le boutron à liste [1] appelé aussi 'dropdown' ;

Le code de la vue [bs-03.xml] est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
9.        <!-- Bootstrap core JavaScript ================================================== -->
10.       <script src="resources/vendor/jquery-2.1.1.min.js"></script>
11.       <script src="resources/vendor/bootstrap.js"></script>
12.       <!-- script local -->
13.       <script type="text/javascript" src="resources/js/bs-03.js"></script>
14.    </head>
15.    <body id="body">
16.       <div class="container">
17.          <!-- barre de navigation -->
18.          <div th:include="navbar2"></div>
19.          <!-- Bootstrap Jumbotron -->
20.          <div th:include="jumbotron"></div>
21.          <!-- contenu -->
22.          <div id="content">
23.             <h1>Ici un contenu</h1>
24.          </div>
25.          <!-- info -->
26.          <div class="alert alert-warning">
27.             <span id="info">Ici, un texte d'information</span>
28.          </div>
29.       </div>
30.    </body>
31. </html>
```

- ligne 11 : le bouton à liste nécessite le fichier JS de Bootstrap ;
- ligne 18 : la nouvelle barre de navigation ;

La vue [navbar2.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
```

```
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
4.        <div class="container">
5.           <div class="navbar-header">
6.              <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
    collapse">
7.                 <span class="sr-only">Toggle navigation</span>
8.                 <span class="icon-bar"></span>
9.                 <span class="icon-bar"></span>
10.                <span class="icon-bar"></span>
11.             </button>
12.             <a class="navbar-brand" href="#">RdvMedecins</a>
13.          </div>
14.          <div class="navbar-collapse collapse">
15.             <img id="loading" src="resources/images/loading.gif" alt="waiting..." style="display: none" />
16.             <!-- formulaire d'identification -->
17.             <div class="navbar-form navbar-right" role="form" id="formulaire" method="post">
18.                <div class="form-group">
19.                   <input type="text" placeholder="Utilisateur" class="form-control" />
20.                </div>
21.                <div class="form-group">
22.                   <input type="password" placeholder="Mot de passe" class="form-control" />
23.                </div>
24.                <button type="button" class="btn btn-success"
    onclick="javascript:connecter()">Connexion</button>
25.                <!-- langues -->
26.                <div class="btn-group">
27.                   <button type="button" class="btn btn-danger">Langues</button>
28.                   <button type="button" class="btn btn-danger dropdown-toggle" data-toggle="dropdown">
29.                      <span class="caret"></span>
30.                      <span class="sr-only">Toggle Dropdown</span>
31.                   </button>
32.                   <ul class="dropdown-menu" role="menu">
33.                      <li>
34.                         <a href="javascript:setLang('fr')">Français</a>
35.                      </li>
36.                      <li>
37.                         <a href="javascript:setLang('en')">English</a>
38.                      </li>
39.                   </ul>
40.                </div>
41.             </div>
42.          </div>
43.       </div>
44.    </div>
45.    <!-- init page -->
46.    <script th:inline="javascript">
47.       /*<![CDATA[*/
48.          // on initialise la page
49.          initNavBar2();
50.       /*]]>*/
51.    </script>
52. </section>
```

- lignes 25-40 : définissent le bouton à liste ;
- ligne 27 : la classe [*btn-danger*] lui donne sa couleur rouge ;
- lignes 32-39 : les éléments de la liste. Ce sont des liens associés chacun à une fonction JS ;
- lignes 46-51 : un script JS exécuté après le chargement du document ;

Le script JS [bs-03.js] est le suivant :

```
1.  function initNavBar2() {
2.     // dropdown des langues
3.     $('.dropdown-toggle').dropdown();
4.  }
5.
6.  function connecter() {
7.     showInfo("Connexion demandée...");
8.  }
9.
10. function setLang(lang) {
11.    var msg;
12.    switch (lang) {
```

```
13.     case 'fr':
14.        msg = "Vous avez choisi la langue française...";
15.        break;
16.     case 'en':
17.        msg = "You have selected english language...";
18.        break;
19.     }
20.     showInfo(msg);
21. }
22.
23. function showInfo(message) {
24.     $("#info").text(message);
25. }
```

- lignes 1-4 : la fonction qui initialise le [dropdown]. [$('.dropdown-toggle')] localise l'élément qui a la classe [dropdown-toggle]. C'est le bouton à liste (ligne 28 de la vue). On lui applique la fonction JS [dropdown()] qui est définie dans le fichier JS [bootstrap.js]. Ce n'est qu'après cette opération que le bouton se comporte comme un bouton à liste ;
- lignes 10-21 : la fonction exécutée lors du choix d'une langue ;

Voici un exemple :

8.6.4.5 Exemple n° 4 : un menu

L'action [/bs-04] affiche la vue [bs-04.xml] suivante :

On a ajouté un menu [1].

La vue [bs-04.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
9.        <!-- Bootstrap core JavaScript ================================================== -->
10.       <script src="resources/vendor/jquery-2.1.1.min.js"></script>
11.       <script src="resources/vendor/bootstrap.js"></script>
12.       <!-- script local -->
13.       <script type="text/javascript" src="resources/js/bs-04.js"></script>
14.    </head>
15.    <body id="body">
16.       <div class="container">
17.          <!-- barre de navigation -->
18.          <div th:include="navbar3"></div>
19.          <!-- Bootstrap Jumbotron -->
20.          <div th:include="jumbotron"></div>
21.          <!-- contenu -->
22.          <div id="content">
23.             <h1>Ici un contenu</h1>
24.          </div>
25.          <!-- info -->
26.          <div class="alert alert-warning">
27.             <span id="info">Ici, un texte d'information</span>
28.          </div>
29.       </div>
30.    </body>
31. </html>
```

- ligne 18 : on insère une nouvelle barre de navigation ;

La vue [navbar3.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
```

```
3.     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
4.        <div class="container">
5.           <div class="navbar-header">
6.              <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
    collapse">
7.                 <span class="sr-only">Toggle navigation</span>
8.                 <span class="icon-bar"></span>
9.                 <span class="icon-bar"></span>
10.                <span class="icon-bar"></span>
11.             </button>
12.             <a class="navbar-brand" href="#">RdvMedecins</a>
13.          </div>
14.          <div class="collapse navbar-collapse">
15.             <img id="loading" src="resources/images/loading.gif" alt="waiting..." style="display: none" />
16.             <ul class="nav navbar-nav">
17.                <li class="active" id="lnkAfficherAgenda">
18.                   <a href="javascript:afficherAgenda()">Agenda </a>
19.                </li>
20.                <li class="active" id="lnkAccueil">
21.                   <a href="javascript:retourAccueil()">Retour Accueil </a>
22.                </li>
23.                <li class="active" id="lnkRetourAgenda">
24.                   <a href="javascript:retourAgenda()">Retour Agenda </a>
25.                </li>
26.                <li class="active" id="lnkValiderRv">
27.                   <a href="javascript:validerRv()">Valider </a>
28.                </li>
29.             </ul>
30.             <!-- boutons de droite -->
31.             <div class="navbar-form navbar-right" role="form">
32.                <!-- déconnexion -->
33.                <button type="button" class="btn btn-success"
    onclick="javascript:deconnecter()">Déconnexion</button>
34.                <!-- langues -->
35.                <div class="btn-group">
36.                   <button type="button" class="btn btn-danger">Langues</button>
37.                   <button type="button" class="btn btn-danger dropdown-toggle" data-toggle="dropdown">
38.                      <span class="caret"></span>
39.                      <span class="sr-only">Toggle Dropdown</span>
40.                   </button>
41.                   <ul class="dropdown-menu" role="menu">
42.                      <li>
43.                         <a href="javascript:setLang('fr')">Français</a>
44.                      </li>
45.                      <li>
46.                         <a href="javascript:setLang('en')">English</a>
47.                      </li>
48.                   </ul>
49.                </div>
50.             </div>
51.          </div>
52.       </div>
53.    </div>
54.    <!-- init page -->
55.    <script th:inline="javascript">
56.       /*<![CDATA[*/
57.          // on initialise la page
58.          initNavBar3();
59.       /*]]>*/
60.    </script>
61. </section>
```

- lignes 16-29 : créent le menu avec quatre options, chacune d'elles étant reliée à un script JS ;
- lignes 55-60 : un script exécuté au chargement de la page ;

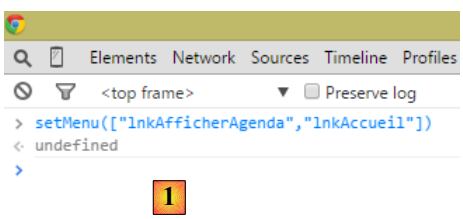
Le script JS [bs-04.js] est le suivant :

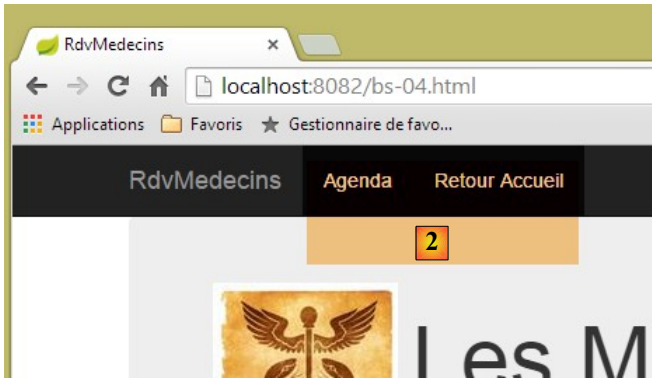
```
1. ...
2. function initNavBar3() {
3.    // dropdown des langues
4.    $('.dropdown-toggle').dropdown();
5.    // l'image animée
```

```
6.     loading = $("#loading");
7.     loading.hide();
8.  }
9.
10. function afficherAgenda() {
11.    showInfo("option [Agenda] cliquée...");
12. }
13.
14. function retourAccueil() {
15.    showInfo("option [Retour accueil] cliquée...");
16. }
17.
18. function retourAgenda() {
19.    showInfo("option [Retour agenda] cliquée...");
20. }
21.
22. function validerRv() {
23.    showInfo("option [Valider] cliquée...");
24. }
25.
26. function setMenu(show) {
27.    // les liens du menu
28.    var lnkAfficherAgenda = $("#lnkAfficherAgenda");
29.    var lnkAccueil = $("#lnkAccueil");
30.    var lnkValiderRv = $("#lnkValiderRv");
31.    var lnkRetourAgenda = $("#lnkRetourAgenda");
32.    // on les met dans un dictionnaire
33.    var options = {
34.       "lnkAccueil" : lnkAccueil,
35.       "lnkAfficherAgenda" : lnkAfficherAgenda,
36.       "lnkValiderRv" : lnkValiderRv,
37.       "lnkRetourAgenda" : lnkRetourAgenda
38.    }
39.    // on cache tous les liens
40.    for ( var key in options) {
41.       options[key].hide();
42.    }
43.    // on affiche ceux qui sont demandés
44.    for (var i = 0; i < show.length; i++) {
45.       var option = show[i];
46.       options[option].show();
47.    }
48. }
```

- lignes 2-18 : la fonction d'initialisation de la page ;
- ligne 4 : pour avoir le bouton à liste des langues ;
- lignes 6-7 : l'image animée est cachée ;
- lignes 26-48 : une fonction [setMenu] qui permet d'indiquer quelles options doivent être visibles ;

Allons dans la console de développement (Ctrl-Maj-I) et entrons le code suivant [1] :

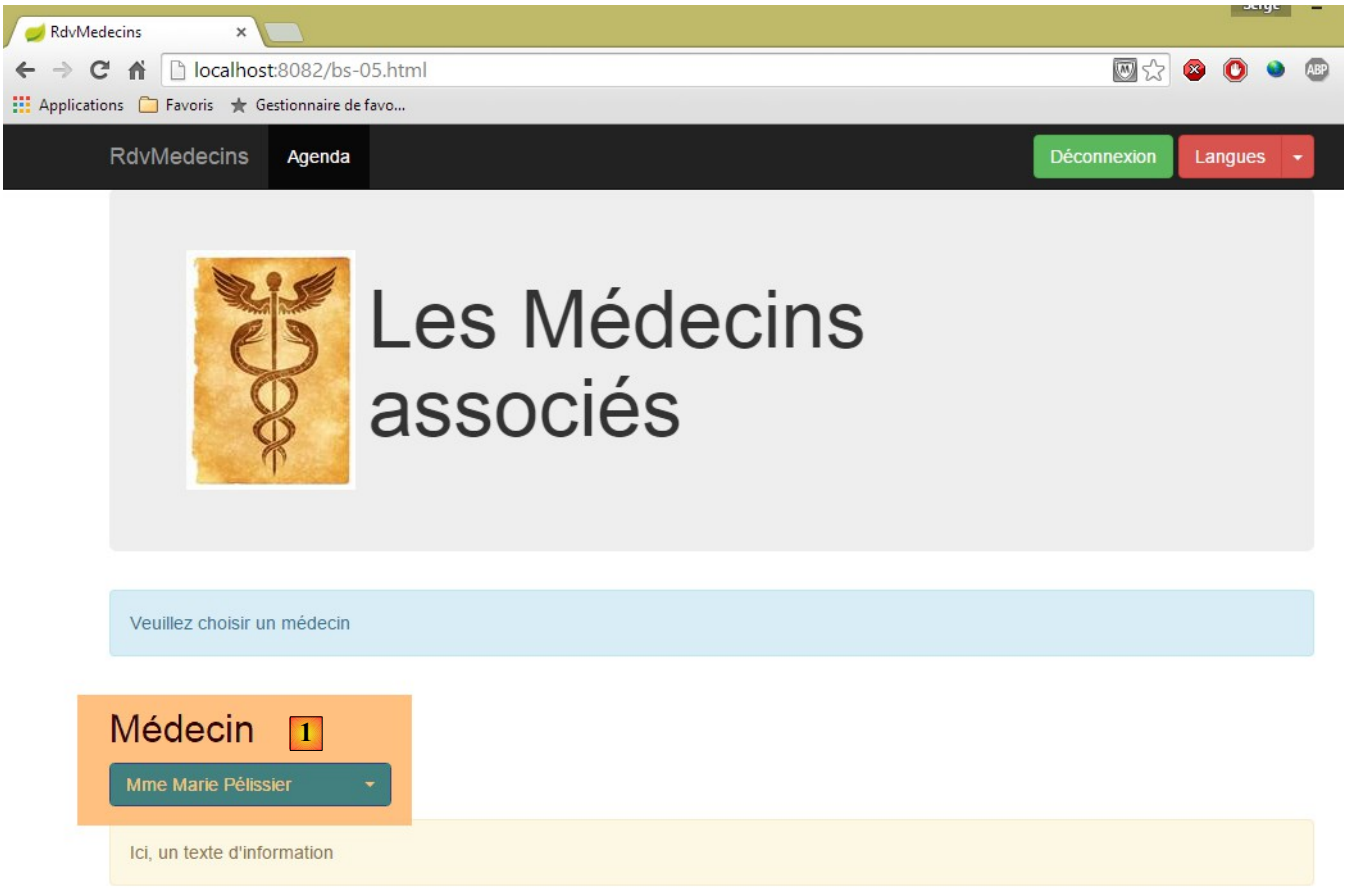




Puis revenons au navigateur. Le menu a changé [2] :

8.6.4.6 Exemple n° 5 : une liste déroulante

L'action [/bs-05] affiche la vue [bs-05.xml] suivante :



La nouveauté est en [1]. Nous utilisons ici un composant fourni en-dehors de Bootstrap, [bootstrap-select] [http://silviomoreto.github.io/bootstrap-select/].

Le code de la vue [bs-05.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.      <head>
4.          <meta name="viewport" content="width=device-width" />
5.          <title>RdvMedecins</title>
6.          <!-- Bootstrap core CSS -->
```

```
7.          <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.          <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-select.min.css" />
9.          <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
10.         <!-- Bootstrap core JavaScript ================================================= -->
11.         <script type="text/javascript" src="resources/vendor/jquery-2.1.1.min.js"></script>
12.         <script type="text/javascript" src="resources/vendor/bootstrap.js"></script>
13.         <script type="text/javascript" src="resources/vendor/bootstrap-select.js"></script>
14.         <!-- script local -->
15.         <script type="text/javascript" src="resources/js/bs-05.js"></script>
16.      </head>
17.      <body id="body">
18.         <div class="container">
19.            <!-- barre de navigation -->
20.            <div th:include="navbar3"></div>
21.            <!-- Bootstrap Jumbotron -->
22.            <div th:include="jumbotron"></div>
23.            <!-- contenu -->
24.            <div id="content" th:include="choixmedecin">
25.            </div>
26.            <!-- info -->
27.            <div class="alert alert-warning">
28.               <span id="info">Ici, un texte d'information</span>
29.            </div>
30.         </div>
31.      </body>
32. </html>
```

- ligne 8 : le CSS nécessaire à la liste déroulante ;
- ligne 13 : le fichier JS nécessaire à la liste déroulante ;
- ligne 24 : la liste déroulante ;

La vue [choixmedecin.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div class="alert alert-info">Veuillez choisir un médecin</div>
4.     <div class="row">
5.        <div class="col-md-3">
6.           <h2>Médecin</h2>
7.           <select id="idMedecin" class="combobox" data-style="btn-primary">
8.              <option value="1">Mme Marie Pélissier</option>
9.              <option value="2">Mr Jean Pardon</option>
10.             <option value="3">Mlle Jeanne Jirou</option>
11.             <option value="4">Mr Paul Macou</option>
12.          </select>
13.       </div>
14.    </div>
15.    <!-- script local -->
16.    <script th:inline="javascript">
17.       /*<![CDATA[*/
18.          // on initialise la page
19.          initChoixMedecin();
20.       /*]]>*/
21.    </script>
22. </section>
```

- ligne 7-12 : on a là une balise [select] classique avec cependant une classe particulière [*combobox*]. L'attribut [data-style="*btn-primary*"] donne au composant sa couleur bleue ;
- lignes 16-21 : un script exécuté au chargement de la page ;

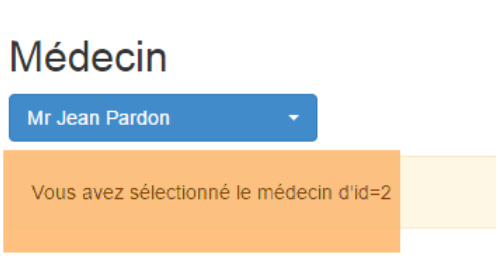
Le fichier JS [bs-05.js] est le suivant :

```
1.  ...
2.  function afficherAgenda() {
3.     var idMedecin = $('#idMedecin option:selected').val();
4.     showInfo("Vous avez sélectionné le médecin d'id=" + idMedecin);
5.  }
6.
7.  function initChoixMedecin() {
8.     // le select des médecins
9.     $('#idMedecin').selectpicker();
10.    // le menu
```

```
11.     setMenu([ "lnkAfficherAgenda" ]);
12. }
```

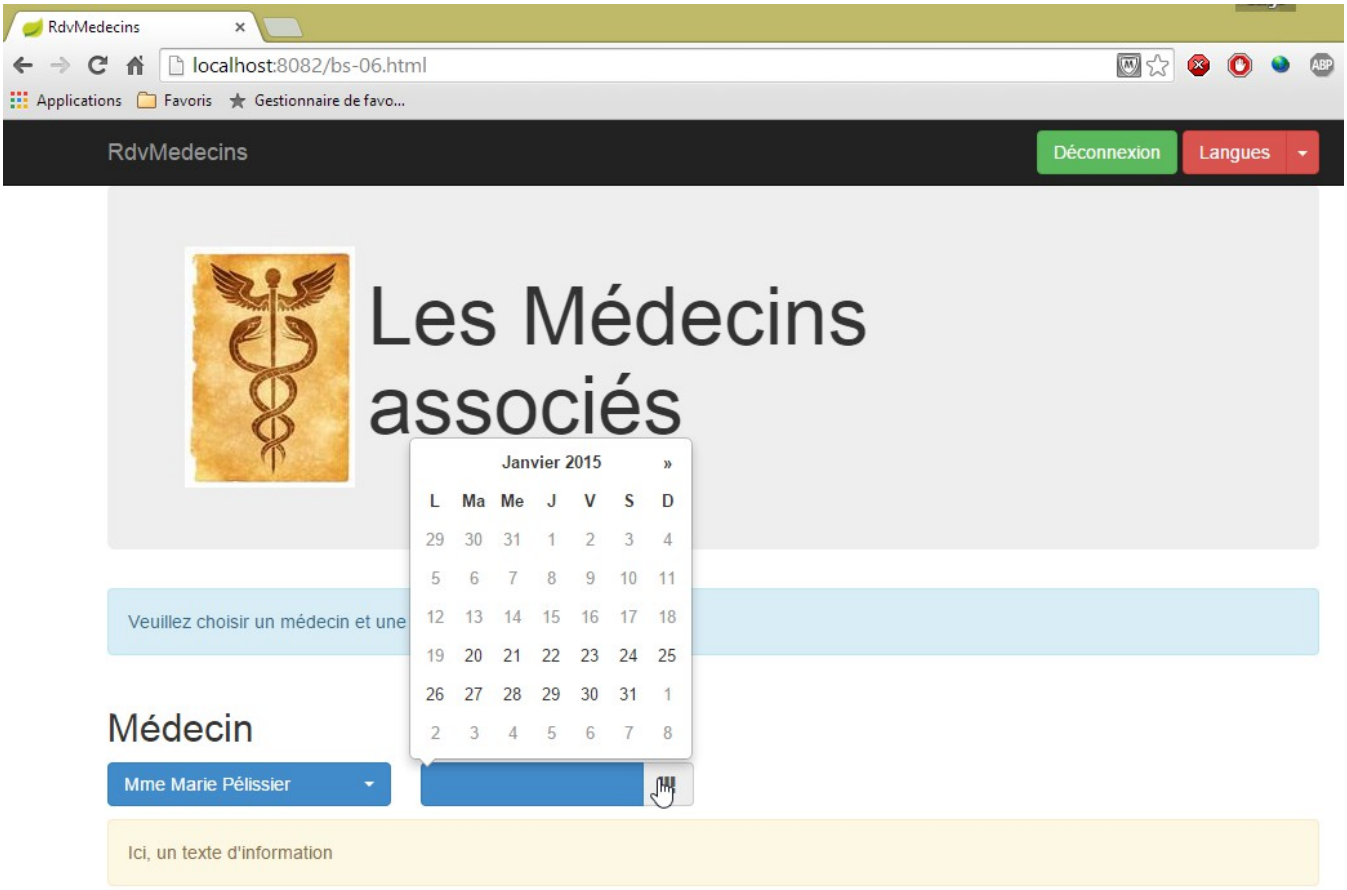
- lignes 7-12 : la fonction exécutée au chargement de la page ;
- ligne 9 : l'instruction qui transforme le [select] de la page en liste déroulante Bootstrap. [`$('#idMedecin')`] référence le [select] (ligne 7 de la vue [choixmedecin]) et la fonction JS [selectpicker] provient du fichier JS [*bootstrap-select.js*] ;
- ligne 11 : on n'affiche qu'une des options du menu ;
- lignes 2-5 : la fonction JS exécutée lorsque qu'on clique sur l'option de menu [Agenda] ;
- ligne 3 : on récupère la valeur de l'option sélectionnée dans la liste déroulante : [`$('#idMedecin option:selected')`] trouve d'abord le composant [id=idMedecin] puis dans ce composant l'option sélectionnée. L'opération [`..`].val() récupère ensuite la valeur de l'élément trouvé, ç-à-d l'attribut [value] de l'option sélectionnée ;

Voici un exemple de choix d'un médecin :

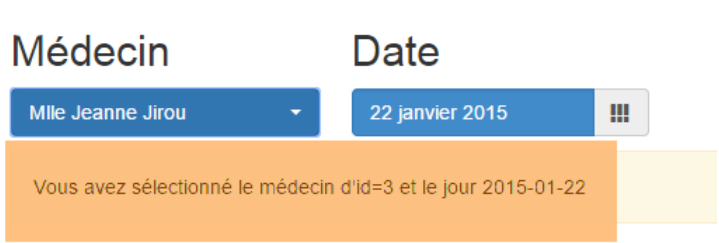


8.6.4.7 Exemple n° 6 : un calendrier

L'action [/bs-06] affiche la vue [bs-06.xml] suivante :



Le choix d'un médecin ou d'une date déclenche une fonction JS qui affiche et le médecin et la date choisies. Voici un exemple :



Grâce au bouton liste des langues, on peut passer le calendrier (et seulement le calendrier) en anglais :

C'est l'exemple le plus complexe de la série. Le calendrier est un composant [bootstrap-datepicker] [http://eternicode.github.io/bootstrap-datepicker].

La vue [bs-06.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-select.min.css" />
9.        <link rel="stylesheet" type="text/css" href="resources/css/datepicker3.css" />
10.       <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
11.       <!-- Bootstrap core JavaScript ================================================= -->
12.       <script type="text/javascript" src="resources/vendor/jquery-2.1.1.min.js"></script>
13.       <script type="text/javascript" src="resources/vendor/bootstrap.js"></script>
14.       <script type="text/javascript" src="resources/vendor/bootstrap-select.js"></script>
15.       <script type="text/javascript" src="resources/vendor/moment-with-locales.js"></script>
16.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.js"></script>
17.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.fr.js"></script>
18.       <!-- script local -->
19.       <script type="text/javascript" src="resources/js/bs-06.js"></script>
20.    </head>
21.    <body id="body">
22.       <div class="container">
23.          <!-- barre de navigation -->
24.          <div th:include="navbar3"></div>
25.          <!-- Bootstrap Jumbotron -->
26.          <div th:include="jumbotron"></div>
27.          <!-- contenu -->
28.          <div id="content" th:include="choixmedecinjour">
29.          </div>
30.          <!-- info -->
31.          <div class="alert alert-warning">
```

```
32.             <span id="info">Ici, un texte d'information</span>
33.          </div>
34.       </div>
35.    </body>
36. </html>
```

- ligne 8 : le fichier CSS du composant [bootstrap-datepicker] ;
- ligne 16 : le fichier JS du composant [bootstrap-datepicker] ;
- ligne 17 : le fichier JS pour gérer un calendrier français. Par défaut, il est en anglais ;
- ligne 15 : le fichier JS d'une bibliothèque appelée [moment] qui donne accès à de très nombreuses fonctions de calcul du temps [http://momentjs.com/];
- ligne 28 : la vue du calendrier ;

La vue [choixmedecinjour.xml] est la suivante :

```
1.  <!DOCTYPE html>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div class="alert alert-info">Veuillez choisir un médecin et une date</div>
4.     <div class="row">
5.        <div class="col-md-3">
6.           <h2>Médecin</h2>
7.           <select id="idMedecin" class="combobox" data-style="btn-primary">
8.              <option value="1">Mme Marie Pélissier</option>
9.              <option value="2">Mr Jean Pardon</option>
10.             <option value="3">Mlle Jeanne Jirou</option>
11.             <option value="4">Mr Paul Macou</option>
12.          </select>
13.       </div>
14.       <div class="col-md-3">
15.          <h2>Date</h2>
16.          <section id="calendar_container">
17.             <div id="calendar" class="input-group date">
18.                <input id="displayjour" type="text" class="form-control btn-primary" disabled="true">
19.                   <span class="input-group-addon">
20.                      <i class="glyphicon glyphicon-th"></i>
21.                   </span>
22.                </input>
23.             </div>
24.          </section>
25.       </div>
26.    </div>
27.    <!-- script local -->
28.    <script th:inline="javascript">
29.       /*<![CDATA[*/
30.          // on initialise la page
31.          initChoixMedecinJour();
32.       /*]]>*/
33.    </script>
34. </section>
```

- lignes 17-23 : le calendrier ;
- ligne 18 : la classe [btn-primary] lui donne sa couleur bleue ;
- ligne 18 : l'attribut [`disabled="true"`] fait qu'on ne peut pas saisir la date à la main. Il faut forcément passer par le calendrier ;
- ligne 16 : le calendrier a été placé dans une section [`id="calendar_container"`]. Pour changer la langue du calendrier, on est obligé de supprimer celui-ci puis de le régénérer. On supprimera donc le contenu du composant [`id="calendar_container"`] puis on y mettra le nouveau calendrier avec la nouvelle langue ;
- lignes 28-33 : le code d'initialisation de la page ;

Le fichier JS [bs-06.js] est le suivant :

```
1.  ...
2.  var calendar_infos = {};
3.
4.  function initChoixMedecinJour() {
5.     // calendrier
6.     var calendar_container = $("#calendar_container");
7.     calendar_infos = {
8.        "container" : calendar_container,
9.        "html" : calendar_container.html(),
```

```
10.         "today" : moment().format('YYYY-MM-DD'),
11.         "langue" : "fr"
12.     }
13.     // création calendrier
14.     updateCalendar();
15.     // le select des médecins
16.     $('#idMedecin').selectpicker();
17.     $('#idMedecin').change(function(e) {
18.         afficherAgenda();
19.     })
20.     // le menu
21.     setMenu([]);
22. }
```

- ligne 2 : le calendrier est géré par plusieurs fonctions JS. La variable [calendar_infos] va rassembler des informations sur le calendrier. Elle est globale pour être vue par les différentes fonctions ;
- ligne 6 : on repère le conteneur du calendrier ;
- lignes 7-12 : les informations mémorisées pour le calendrier ;
  - ligne 8 : une référence sur son conteneur,
  - ligne 9 : le code HTML du calendrier. Avec ces deux informations, on est capable de supprimer le calendrier et de le régénérer,
  - ligne 10 : la date d'aujourd'hui au format [aaaa-mm-jj],
  - ligne 11 : la langue du calendrier ;
- ligne 14 : création du calendrier ;
- ligne 16 : le combo des médecins ;
- lignes 17-19 : à chaque fois que la valeur sélectionnée dans ce combo changera, la méthode [afficherAgenda] sera exécutée ;
- ligne 21 : pas de menu dans la barre de navigation ;

La fonction [updateCalendar] est la suivante :

```
1.  function updateCalendar(renew) {
2.      if (renew) {
3.          // régénération du calendrier actuel
4.          calendar_infos.container.html(calendar_infos.html);
5.      }
6.      // initialisation du calendrier
7.      var calendar = $("#calendar");
8.      var settings = {
9.          format : "yyyy-mm-dd",
10.         startDate : calendar_infos.today,
11.         language : calendar_infos.langue,
12.     };
13.     calendar.datepicker(settings);
14.     // sélection de la date courante
15.     if (calendar_infos.date) {
16.         calendar.datepicker('setDate', calendar_infos.date)
17.     }
18.     // évts
19.     calendar.datepicker().on('hide', function(e) {
20.         // affichage jour sélectionné
21.         displayJour();
22.     });
23.     calendar.datepicker().on('changeDate', function(e) {
24.         // on note la nouvelle date
25.         calendar_infos.date = moment(calendar.datepicker('getDate')).format("YYYY-MM-DD");
26.         // affichage infos agenda
27.         afficherAgenda();
28.         // affichage jour sélectionné
29.         displayJour();
30.     });
31.     // affichage jour sélectionné
32.     displayJour();
33. }
```

- ligne 1 : la fonction [updateCalendar] admet un paramètre qui peut être présent ou non. S'il est présent, alors le calendrier est régénéré (ligne 4) à partir des informations contenues dans [calendar_infos] ;
- ligne 7 : on référence le calendrier ;
- lignes 8-12 : ses paramètres d'initialisation ;
  - ligne 9 : le format des dates gérées [aaaa-mm-jj],

    - ligne 10 : la 1ère date qui peut être sélectionnée dans le calendrier. Ici, la date d'aujourd'hui. Les dates qui précèdent ne pourront pas être sélectionnées,
    - ligne 11 : la langue du calendrier. Il y en aura deux ['en'] et ['fr'] ;
- ligne 13 : le calendrier est configuré ;
- lignes 15-17 : si la date de [calendar_infos] a été initialisée, alors on donne cette date comme date actuelle du calendrier ;
- lignes 19-22 : à chaque fois que le calendrier se refermera, on affichera la date sélectionnée ;
- lignes 23-30 : à chaque fois qu'il y aura un changement de date dans le calendrier :
    - ligne 25 : on note la date sélectionnée dans [calendar_infos],
    - ligne 27 : on affiche des informations sur l'agenda,
    - ligne 29 : on affiche le jour sélectionné ;
- ligne 32 : affichage du jour sélectionné s'il y en a un ;

La méthode [displayJour] qui affiche le jour sélectionné est la suivante :

```
1.  // affiche le jour sélectionné
2.  function displayJour() {
3.      if (calendar_infos.date) {
4.          var displayjour = $("#displayjour");
5.          moment.locale(calendar_infos.langue);
6.          jour = moment(calendar_infos.date).format('LL');
7.          displayjour.val(jour);
8.      }
9.  }
```

- ligne 3 : si une date a déjà été sélectionnée (au début le calendrier n'a pas de date sélectionnée) ;
- ligne 4 : on localise le composant où on va écrire la date ;
- ligne 5 : cette date peut être écrite en anglais ou français. On fixe la langue de la bibliothèque [moment] ;
- ligne 6 : on affiche la date sélectionnée dans la langue choisie et au format long ;
- ligne 7 : cette date est affichée ;

Voici deux exemples :

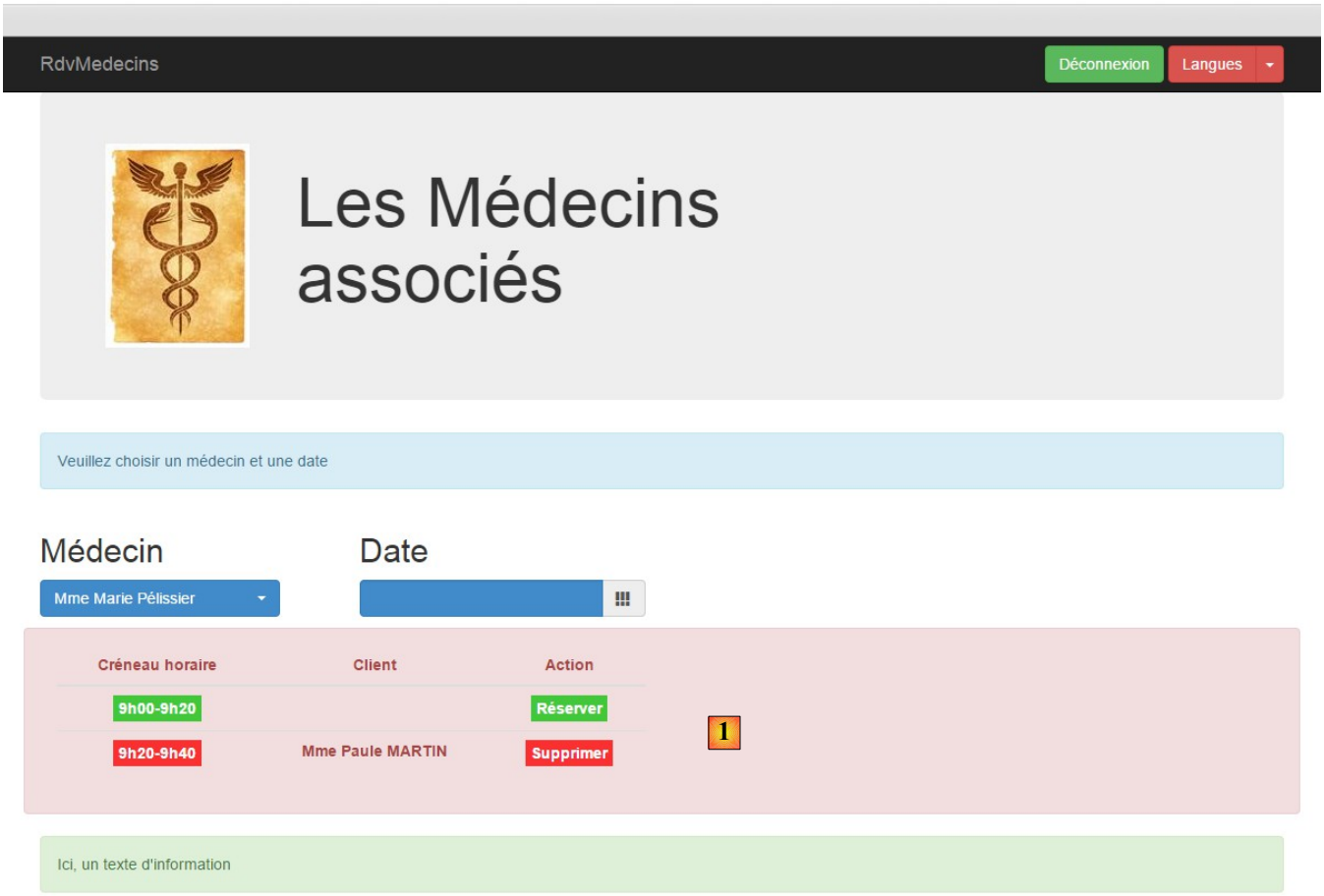
Lors d'un changement de médecin ou de date, la méthode [afficherAgenda] est exécutée :

```
1.  function afficherAgenda() {
2.      // on affiche médecin et date
3.      var idMedecin = $('#idMedecin option:selected').val();
4.      if (calendar_infos.date) {
5.          showInfo("Vous avez sélectionné le médecin d'id=" + idMedecin + " et le jour " +
    calendar_infos.date);
6.      }
7.  }
```

8.6.4.8 Exemple n° 7 : une table HTML 'responsive'

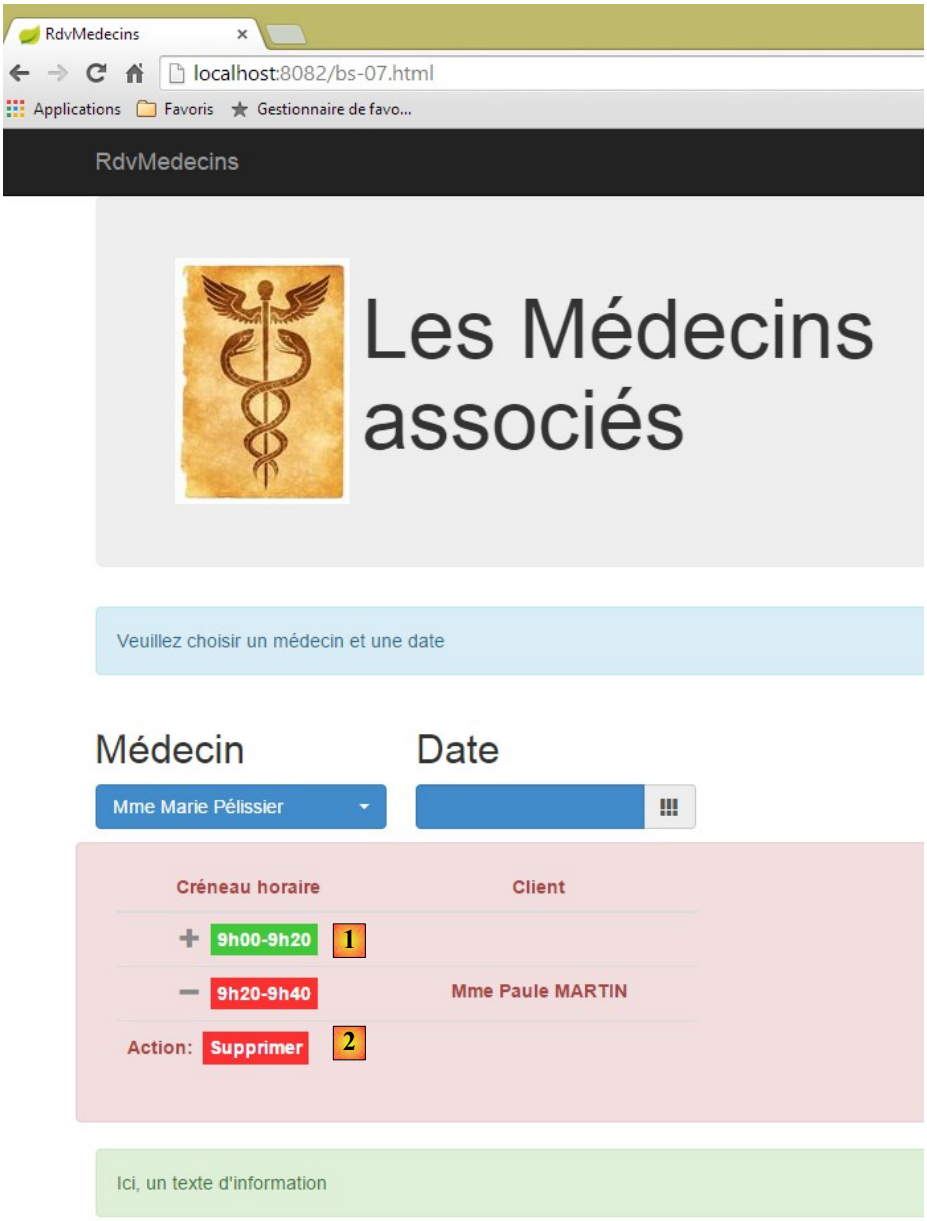
Note : 'responsive' est un terme anglais indiquant qu'un composant est capable de s'adapter à la taille de l'écran sur lequel il est visualisé. Nous allons en montrer un exemple.

L'action [/bs-07] affiche la vue [bs-07.xml] suivante (plein écran) :



La nouveauté est la table HTML [1]. Cette table est gérée par la bibliothèque JS [footable] : [https://github.com/fooplugins/FooTable].

Si on réduit la taille de la fenêtre du navigateur, on obtient la chose suivante :



- la table HTML s'est adaptée à la taille de l'écran ;
- en [1], pour voir le lien [Réserver], il faut cliquer sur le signe [+] ;
- en [2], ce qu'on voit lorsqu'on clique sur le signe [+] ;

La vue [bs-07.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-select.min.css" />
9.        <link rel="stylesheet" type="text/css" href="resources/css/datepicker3.css" />
10.       <link rel="stylesheet" type="text/css" href="resources/css/footable.core.min.css" />
11.       <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
12.       <!-- Bootstrap core JavaScript ================================================== -->
13.       <script type="text/javascript" src="resources/vendor/jquery-2.1.1.min.js"></script>
14.       <script type="text/javascript" src="resources/vendor/bootstrap.js"></script>
15.       <script type="text/javascript" src="resources/vendor/bootstrap-select.js"></script>
16.       <script type="text/javascript" src="resources/vendor/moment-with-locales.js"></script>
17.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.js"></script>
```

```
18.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.fr.js"></script>
19.       <script type="text/javascript" src="resources/vendor/footable.js"></script>
20.       <!-- script local -->
21.       <script type="text/javascript" src="resources/js/bs-07.js"></script>
22.    </head>
23.    <body id="body">
24.       <div class="container">
25.          <!-- barre de navigation -->
26.          <div th:include="navbar3" />
27.          <!-- Bootstrap Jumbotron -->
28.          <div th:include="jumbotron" />
29.          <!-- contenu -->
30.          <div id="content" th:include="choixmedecinjour" />
31.          <div id="agenda" th:include="agenda" />
32.          <!-- info -->
33.          <div class="alert alert-success">
34.             <span id="info">Ici, un texte d'information</span>
35.          </div>
36.       </div>
37.    </body>
38. </html>
```

- ligne 10 : le CSS de la bibliothèque [footable] ;
- ligne 19 : le JS de la bibliothèque [footable] ;
- ligne 31 : la table HTML d'un agenda ;

La vue [agenda.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <div class="row alert alert-danger">
5.           <div class="col-md-6">
6.              <table id="creneaux" class="table">
7.                 <thead>
8.                    <tr>
9.                       <th data-toggle="true">
10.                         <span>Créneau horaire</span>
11.                      </th>
12.                      <th>
13.                         <span>Client</span>
14.                      </th>
15.                      <th data-hide="phone">
16.                         <span>Action</span>
17.                      </th>
18.                   </tr>
19.                </thead>
20.                <tbody>
21.                   <tr>
22.                      <td>
23.                         <span class='status-metro status-active'>
24.                            9h00-9h20
25.                         </span>
26.                      </td>
27.                      <td>
28.                         <span></span>
29.                      </td>
30.                      <td>
31.                         <a href="javascript:reserver(14)" class="status-metro status-active">
32.                            Réserver
33.                         </a>
34.                      </td>
35.                   </tr>
36.                   <tr>
37.                      <td>
38.                         <span class='status-metro status-suspended'>
39.                            9h20-9h40
40.                         </span>
41.                      </td>
42.                      <td>
43.                         <span>Mme Paule MARTIN</span>
44.                      </td>
45.                      <td>
```

```
46.                         <a href="javascript:supprimer(17)" class="status-metro status-suspended">
47.                            Supprimer
48.                         </a>
49.                      </td>
50.                   </tr>
51.                </tbody>
52.             </table>
53.          </div>
54.       </div>
55.       <!-- init page -->
56.       <script th:inline="javascript">
57.          /*<![CDATA[*/
58.          // on initialise la page
59.          initAgenda();
60.       /*]]>*/
61.       </script>
62.    </body>
63. </html>
```

- ligne 4 : installe la table dans une ligne [row] et un encadré coloré [*alert alert-danger*] ;
- ligne 5 : la table va occuper 6 colonnes [col-md-6] ;
- ligne 6 : la table HTML est formatée par Bootstrap [class='table'] ;
- ligne 9 : l'attribut [data-toggle] indique la colonne qui héberge le symbole [+/-] qui déplie / replie la ligne ;
- ligne 15 : l'attribut [data-hide='phone'] indique que la colonne doit être cachée si l'écran a la taille d'un écran de téléphone. On peut également utiliser la valeur 'tablet' ;
- ligne 31 : on associe une fonction JS au lien [Réserver] ;
- ligne 46 : on associe une fonction JS au lien [Supprimer] ;
- lignes 56-61 : initialisation de la page ;

Un certain nombre de classes CSS utilisées ci-dessus proviennent du fichier CSS [bootstrapDemo.css] :

```
@CHARSET "UTF-8";

#creneaux th {
   text-align: center;
}

#creneaux td {
   text-align: center;
   font-weight: bold;
}

.status-metro {
  display: inline-block;
  padding: 2px 5px;
  color:#fff;
}

.status-metro.status-active {
  background: #43c83c;
}

.status-metro.status-suspended {
  background: #fa3031;
}
```

Les styles [status-*] proviennent d'un exemple d'utilisation de la table [footable] trouvé sur le site de la bibliothèque.

Dans le fichier JS [bs-07.js], la page est initialisée de la façon suivante :

```
1.  function initAgenda() {
2.     // le tableau des créneaux horaires
3.     $("#creneaux").footable();
4.  }
```

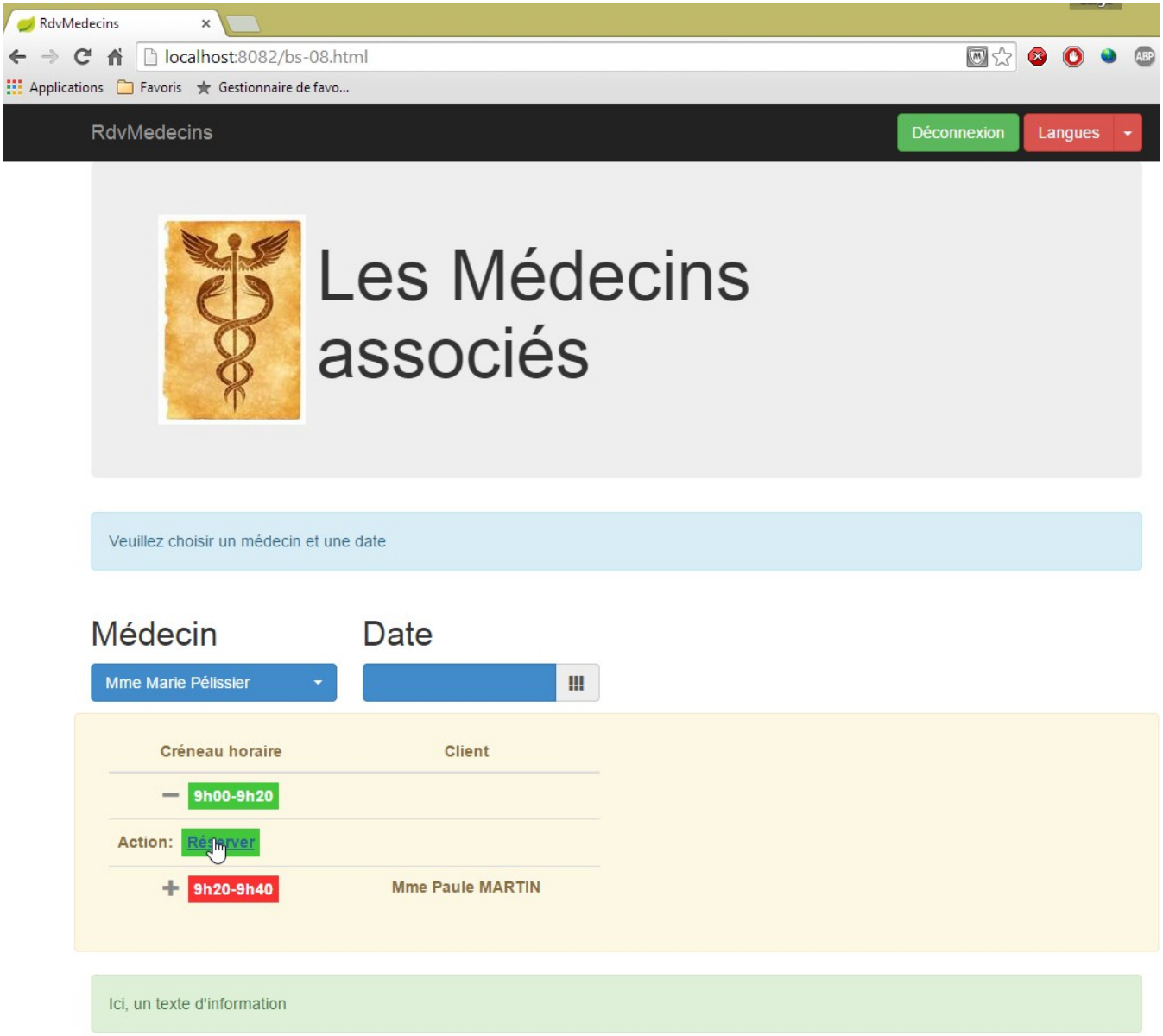
C'est tout. [$("#creneaux")] référence la table HTML qu'on veut rendre 'responsive'. Par ailleurs, on trouve les fonctions JS liées aux deux liens [Réserver] et [Supprimer] :

```
1.  function reserver(idCreneau) {
2.     showInfo("Réservation du créneau n° " + idCreneau);
```

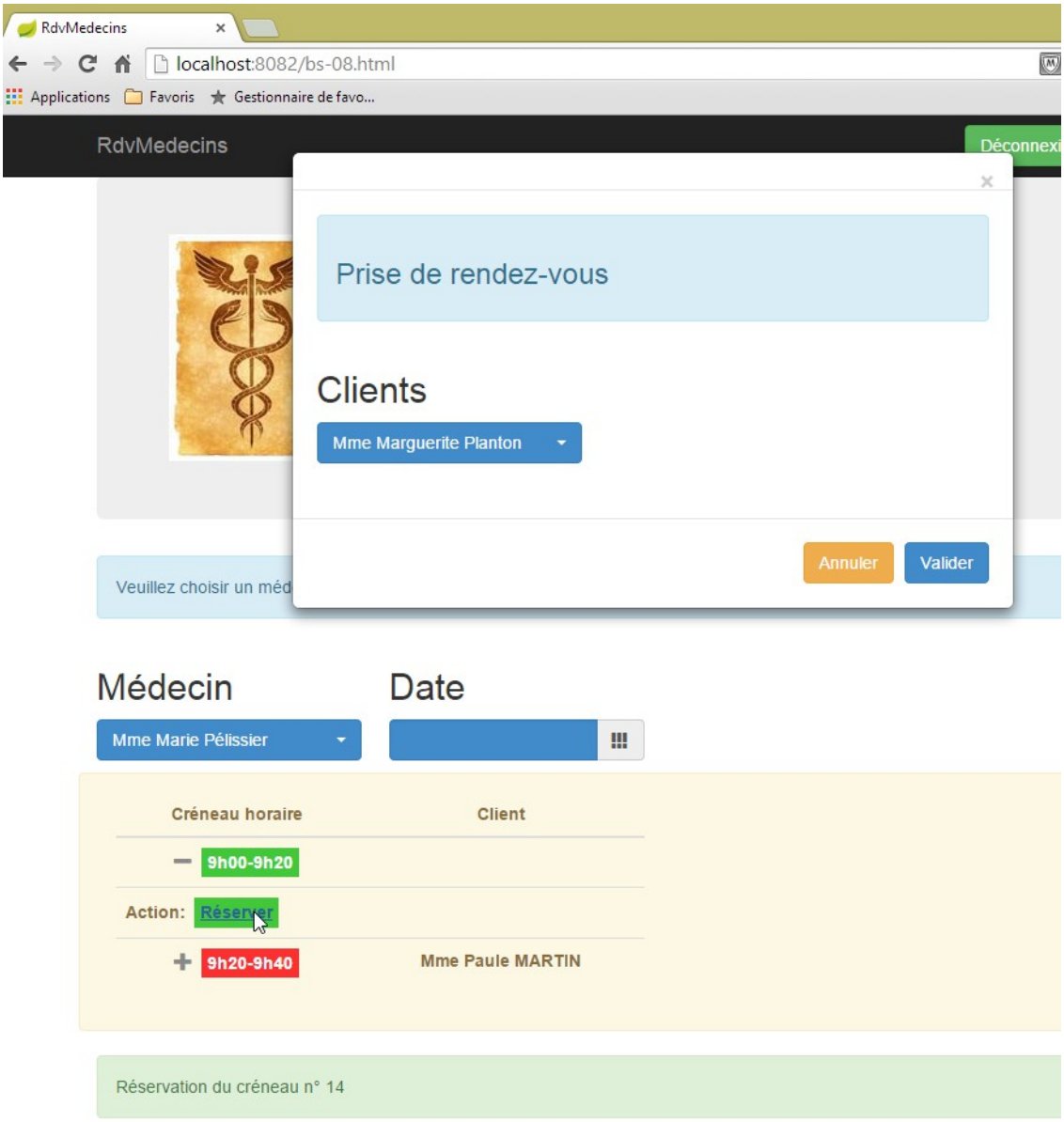
```
}

function supprimer(idRv) {
    showInfo("Suppression du rv n° " + idRv);
}
```

8.6.4.9 *Exemple n° 8 : une boîte modale*

L'action [/bs-08] affiche la vue [bs-08.xml] suivante :



Alors que précédemment, cliquer sur le lien [Réserver] affichait une information dans la boîte d'informations, ici on va faire apparaître une boîte modale pour sélectionner un client pour le RV :



Le composant utilisé est le composant [bootstrap-modal] [https://github.com/jschr/bootstrap-modal/].

La vue [bs-08.xml] est la suivante :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
3.     <head>
4.        <meta name="viewport" content="width=device-width" />
5.        <title>RdvMedecins</title>
6.        <!-- Bootstrap core CSS -->
7.        <link rel="stylesheet" href="resources/css/bootstrap-3.1.1-min.css" />
8.        <link rel="stylesheet" type="text/css" href="resources/css/bootstrap-select.min.css" />
9.        <link rel="stylesheet" type="text/css" href="resources/css/datepicker3.css" />
10.       <link rel="stylesheet" type="text/css" href="resources/css/footable.core.min.css" />
11.       <link rel="stylesheet" type="text/css" href="resources/css/bootstrapDemo.css" />
12.       <!-- Bootstrap core JavaScript ================================================== -->
13.       <script type="text/javascript" src="resources/vendor/jquery-2.1.1.min.js"></script>
14.       <script type="text/javascript" src="resources/vendor/bootstrap.js"></script>
15.       <script type="text/javascript" src="resources/vendor/bootstrap-select.js"></script>
16.       <script type="text/javascript" src="resources/vendor/moment-with-locales.js"></script>
17.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.js"></script>
18.       <script type="text/javascript" src="resources/vendor/bootstrap-datepicker.fr.js"></script>
19.       <script type="text/javascript" src="resources/vendor/bootstrap-modal.js"></script>
20.       <script type="text/javascript" src="resources/vendor/footable.js"></script>
```

```
21.       <!-- script local -->
22.       <script type="text/javascript" src="resources/js/bs-08.js"></script>
23.    </head>
24.    <body id="body">
25.       <div class="container">
26.          <!-- barre de navigation -->
27.          <div th:include="navbar3" />
28.          <!-- Bootstrap Jumbotron -->
29.          <div th:include="jumbotron" />
30.          <!-- contenu -->
31.          <div id="content" th:include="choixmedecinjour" />
32.          <div id="agenda" th:include="agenda-modal" />
33.          <div th:include="resa" />
34.          <!-- info -->
35.          <div class="alert alert-success">
36.             <span id="info">Ici, un texte d'information</span>
37.          </div>
38.       </div>
39.    </body>
40. </html>
```

- ligne 19 : le fichier JS nécessaire aux boîtes modales ;
- ligne 32 : la vue [agenda-modal] est identique à la vue [agenda] à un détail près : la fonction JS qui gère le lien [Réserver] :

```
<a href="javascript:showDialogResa(14)" class="status-metro status-active">Réserver</a>
```

La fonction [*showDialogResa*] est chargée de faire apparaître la boîte modale de sélection d'un client ;

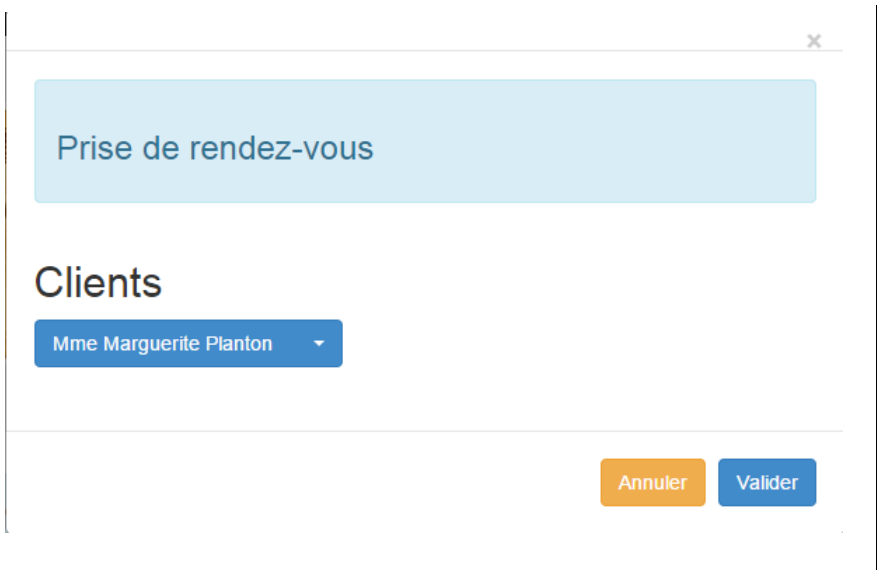
- ligne 33 : la vue [resa.xml] est la boîte modale de sélection d'un client :

```
1.  <!DOCTYPE HTML>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div id="resa" class="modal fade">
4.        <div class="modal-dialog">
5.           <div class="modal-content">
6.              <div class="modal-header">
7.                 <button type="button" class="close" data-dismiss="modal" aria-label="Close">
8.                    <span aria-hidden="true">
9.                    </span>
10.                </button>
11.                <!-- <h4 class="modal-title">Modal title</h4> -->
12.             </div>
13.             <div class="modal-body">
14.                <div class="alert alert-info">
15.                   <h3>
16.                      <span>Prise de rendez-vous</span>
17.                   </h3>
18.                </div>
19.                <div class="row">
20.                   <div class="col-md-3">
21.                      <h2>Clients</h2>
22.                      <select id="idClient" class="combobox" data-style="btn-primary">
23.                         <option value="1">Mme Marguerite Planton</option>
24.                         <option value="2">Mr Maxime Franck</option>
25.                         <option value="3">Mlle Elisabeth Oron</option>
26.                         <option value="4">Mr Gaëtan Calot</option>
27.                      </select>
28.                   </div>
29.                </div>
30.             </div>
31.             <div class="modal-footer">
32.                <button type="button" class="btn btn-warning"
    onclick="javascript:cancelDialogResa()">Annuler</button>
33.                <button type="button" class="btn btn-primary"
    onclick="javascript:validateResa()">Valider</button>
34.             </div>
35.          </div><!-- /.modal-content -->
```

```
36.       </div><!-- /.modal-dialog -->
37.    </div><!-- /.modal -->
38.    <!-- init page -->
39.    <script th:inline="javascript">
40.       /*<![CDATA[*/
41.          // on initialise la page
42.          initResa();
43.       /*]]>*/
44.    </script>
45. </section>
```

- lignes 3-37 : la boîte modale ;
- lignes 13-30 : le conenu de cette boîte (ce qui sera affiché) ;
- lignes 31-34 : les boutons de la boîte de dialogue ;
- ligne 32 : un bouton [Annuler] géré par la fonction JS [*cancelDialogResa*] ;
- ligne 33 : un bouton [Valider] géré par la fonction JS [*validateResa*] ;
- lignes 39-44 : le script d'initialisation de la boîte modale ;

Cela donne la vue suivante :



A noter que la boîte modale n'est pas affichée par défaut. C'est pourquoi, on ne la voit pas au démarrage de l'application bien que son code HTML soit présent dans le document.

Le fichier JS [bs-08.js] est le suivant :

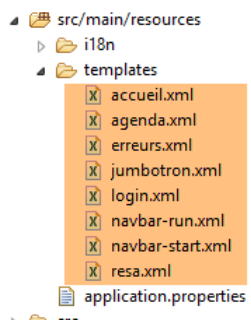
```
1.  var idCreneau;
2.  var idClient;
3.  var resa;
4.
5.  function showDialogResa(idCreneau) {
6.     // on mémorise l'id du créneau
7.     this.idCreneau = idCreneau;
8.     // on affiche le dialogue de réservation
9.     var resa = $("#resa");
10.    resa.modal('show');
11.    // log
12.    showInfo("Réservation du créneau n° " + idCreneau);
13. }
14.
15. function cancelDialogResa() {
16.    // on cache la boîte de dialogue
17.    resa.modal('hide');
18. }
19.
20. // validation résa
```

```
21. function validateResa() {
22.     // on récupère les infos
23.     var idClient = $('#idClient option:selected').val();
24.     // on cache la boîte de dialogue
25.     resa.modal('hide');
26.     // infos
27.     showInfo("Réservation du créneau n° " + idCreneau + " pour le client n° " + idClient)
28. }
29. 
30. function initResa() {
31.     // le select des clients
32.     $('#idClient').selectpicker();
33.     // boîte modale
34.     resa = $("#resa");
35.     resa.modal({});
36. }
```

- lignes 30-36 : la fonction d'initialisation de la boîte modale ;
- ligne 32 : la boîte modale contient une liste déroulante qu'il faut initialiser ;
- lignes 34-35 : initialisation de la boîte modale elle-même ;
- lignes 5-13 : la fonction JS attachée au lien [Réserver] ;
- ligne 7 : on mémorise le paramètre de la fonction dans la variable globale de la ligne 1 ;
- lignes 9-10 : la boîte modale est rendue visible ;
- ligne 12 : on logue une information dans la boîte d'informations ;
- lignes 15-18 : gestion du bouton [Annuler]. On se contente de cacher la boîte modale (ligne 17) ;
- lignes 21-31 : la fonction JS attachée au bouton [Valider] ;
- ligne 23 : on récupère l'attribut [value] du client sélectionné ;
- ligne 25 : on cache la boîte de dialogue ;
- ligne 27 : on logue les deux informations : n° du créneau réservé et pour quel client ;

8.6.5 Étape 2 : écriture des vues

Nous allons maintenant décrire les vues délivrées par le serveur [Web1] ainsi que leurs modèles.



8.6.5.1 La vue [navbar-start]

Elle affiche la barre de navigation de la page de boot :

Le code de [navbar-start.xml] est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <section xmlns:th="http://www.thymeleaf.org">
3.     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
4.        <div class="container">
5.           <div class="navbar-header">
6.              <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
    collapse">
7.                 <span class="sr-only">Toggle navigation</span>
8.                 <span class="icon-bar"></span>
9.                 <span class="icon-bar"></span>
10.                <span class="icon-bar"></span>
11.             </button>
12.             <a class="navbar-brand" href="#">RdvMedecins</a>
13.          </div>
14.          <div class="navbar-collapse collapse">
15.             <img id="loading" src="resources/images/loading.gif" alt="waiting..." style="display: none" />
16.             <!-- formulaire d'identification -->
17.             <div class="navbar-form navbar-right" role="form" id="formulaire">
18.                <div class="form-group">
19.                   <input type="text" th:placeholder="#{service.url}" class="form-control" id="urlService"
    />
20.                </div>
21.                <div class="form-group">
22.                   <input type="text" th:placeholder="#{username}" class="form-control" id="login" />
23.                </div>
24.                <div class="form-group">
25.                   <input type="password" th:placeholder="#{password}" class="form-control" id="passwd" />
26.                </div>
27.                <button type="button" class="btn btn-success" th:text="#{login}"
    onclick="javascript:connecter()">Sign in</button>
28.                <!-- langues -->
29.                <div class="btn-group">
30.                   <button type="button" class="btn btn-danger" th:text="#{langues}">Action</button>
31.                   <button type="button" class="btn btn-danger dropdown-toggle" data-toggle="dropdown">
32.                      <span class="caret"></span>
33.                      <span class="sr-only">Toggle Dropdown</span>
34.                   </button>
35.                   <ul class="dropdown-menu" role="menu">
```

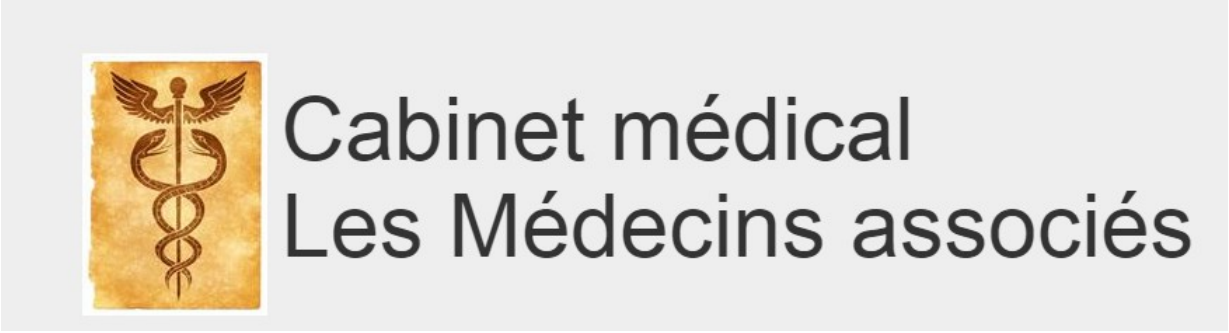
```
36.                         <li>
37.                             <a href="javascript:setLang('fr')" th:text="#{langues.fr}" />
38.                         </li>
39.                         <li>
40.                             <a href="javascript:setLang('en')" th:text="#{langues.en}" />
41.                         </li>
42.                     </ul>
43.                 </div>
44.             </div>
45.         </div>
46.     </div>
47.     <!-- init page -->
48.     <script th:inline="javascript">
49.         /*<![CDATA[*/
50.             // on initialise la page
51.             initNavBarStart();
52.         /*]]>*/
53.     </script>
54. </section>
```

Cette vue n'a pas de modèle. Elle a les gestionnaires d'événements suivants :

évt	gestionnaire
clic sur le bouton de connexion	*connecter() - ligne 27*
clic sur le lien [Français]	*setLang('fr') - ligne 37*
clic sur le lien [English]	*setLang('en') - ligne 40*

8.6.5.1 La vue [jumbotron]

C'est la vue qui est présentée sous la barre de navigation [navbar-start] dans la page de boot :



Son code [jumbotron.xml] est le suivant :

```
1.  <!DOCTYPE html>
2.  <section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.      <!-- Bootstrap Jumbotron -->
4.      <div class="jumbotron">
5.          <div class="row">
6.              <div class="col-md-2">
7.                  <img src="resources/images/caduceus.jpg" alt="RvMedecins" />
8.              </div>
9.              <div class="col-md-10">
10.                 <h1 th:utext="#{application.header}" />
11.             </div>
12.         </div>
13.     </div>
14. </section>
```

La vue [jumbotron] n'a ni modèle ni événements.

8.6.5.2 La vue [login]

C'est la vue qui est présentée sous le jumbotron dans la page de boot :

Son code [login.xml] est le suivant :

```
<!DOCTYPE html>
<section xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
    <div class="alert alert-info" th:text="#{identification}">Identification
    </div>
</section>
```

La vue n'a ni modèle ni événements.

8.6.5.3 La vue [navbar-run]

C'est la barre de navigation présentée lorsque la connexion a réussi :

Son code [navbar-run.xml] est le suivant :

```
<!DOCTYPE HTML>
<section xmlns:th="http://www.thymeleaf.org">
    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="#">RdvMedecins</a>
            </div>
            <div class="collapse navbar-collapse">
                <img id="loading" src="resources/images/loading.gif" alt="waiting..." style="display: none" />
                <!-- boutons de droite -->
                <form class="navbar-form navbar-right" role="form">
                    <!-- déconnexion -->
                    <button type="button" class="btn btn-success" th:text="#{options.deconnecter}"
onclick="javascript:deconnecter()">Déconnexion</button>
                    <!-- langues -->
                    <div class="btn-group">
                        <button type="button" class="btn btn-danger" th:text="#{langues}">Langue</button>
                        <button type="button" class="btn btn-danger dropdown-toggle" data-toggle="dropdown">
                            <span class="caret"></span>
                            <span class="sr-only">Toggle Dropdown</span>
                        </button>
                        <ul class="dropdown-menu" role="menu">
                            <li>
                                <a href="javascript:setLang('fr')" th:text="#{langues.fr}" />
                            </li>
                            <li>
                                <a href="javascript:setLang('en')" th:text="#{langues.en}" />
                            </li>
                        </ul>
                    </div>
                </form>
```

```
37.          </div>
38.       </div>
39.    </div>
40.    <!-- init page -->
41.    <script th:inline="javascript">
42.       /*<![CDATA[*/
43.          // on initialise la page
44.          initNavBarRun();
45.       /*]]>*/
46.    </script>
47. </section>
```

Cette vue n'a pas de modèle. Elle a les gestionnaires d'événements suivants :

évt	gestionnaire
clic sur le bouton de déconnexion	*deconnecter() - ligne 19*
clic sur le lien [Français]	*setLang('fr') - ligne 29*
clic sur le lien [English]	*setLang('en') - ligne 32*

8.6.5.4 La vue [accueil]

C'est la vue présentée immédiatement sous la barre de navigation [navbar-run] :

Son code [accueil.html] est le suivant :

```
1.  <!DOCTYPE html>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
3.     <div class="alert alert-info" th:text="#{choixmedecinjour.title}">Veuillez choisir un médecin et une
    date</div>
4.     <div class="row">
5.        <div class="col-md-3">
6.           <h2 th:text="#{rv.medecin}">Médecin</h2>
7.           <select name="idMedecin" id="idMedecin" class="combobox" data-style="btn-primary">
8.              <option th:each="medecinItem : ${rdvmedecins.medecinItems}" th:text="${medecinItem.texte}"
    th:value="${medecinItem.id}"/>
9.           </select>
10.       </div>
11.       <div class="col-md-3">
12.          <h2 th:text="#{rv.jour}">Date</h2>
13.          <section id="calendar_container">
14.             <div id="calendar" class="input-group date">
15.                <input id="displayjour" type="text" class="form-control btn-primary" disabled="true">
16.                   <span class="input-group-addon">
17.                      <i class="glyphicon glyphicon-th"></i>
18.                   </span>
19.                </input>
20.             </div>
21.          </section>
22.       </div>
23.    </div>
24.    <!-- agenda -->
25.    <div id="agenda"></div>
26.    <!-- script local -->
27.    <script th:inline="javascript">
28.       /*<![CDATA[*/
29.          // on initialise la page
30.          initChoixMedecinJour();
31.       /*]]>*/
32.    </script>
33. </html>
```

Son modèle est le suivant :

- [*rdvmedecins.medecinItems*] (ligne 8) : la liste des médecins ;

Dans sa forme actuelle, la vue ne semble pas avoir de gestionnaire d'événements. En réalité ceux-ci sont définis dans la fonction [initChoixMedecinJour]. Cette fonction a été présentée au paragraphe 8.6.4.7, page 521 et plus particulièrement page 524. On y trouve les gestionnaires d'événements suivants :

évt	gestionnaire
choix d'un médecin	*getAgenda*
choix d'une date	*getAgenda*

8.6.5.5 La vue [agenda]

La vue [agenda] présente une journée de l'agenda d'un médecin :

Son code [agenda.xml] est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns:th="http://www.thymeleaf.org">
3.     <body>
4.        <h3 class="alert alert-info" th:text="${agenda.titre}">Agenda de Mme Pélissier le 13/10/2014</h3>
5.        <h4 class="alert alert-danger" th:if="${agenda.creneaux.length}==0"
    th:text="#{agenda.medecinsanscreneaux}">Ce médecin n'a pas encore de créneaux
6.           de consultation</h4>
7.        <th:block th:if="${agenda.creneaux.length}!=0">
8.           <div class="row tab-content alert alert-warning">
9.              <div class="tab-pane active col-md-6">
10.                <table id="creneaux" class="table">
11.                   <thead>
12.                      <tr>
13.                         <th data-toggle="true">
14.                            <span th:text="#{agenda.creneauhoraire}">Créneau horaire</span>
15.                         </th>
16.                         <th>
17.                            <span th:text="#{agenda.client}">Client</span>
```

```
18.                         </th>
19.                         <th data-hide="phone">
20.                            <span th:text="#{agenda.action}">Action</span>
21.                         </th>
22.                      </tr>
23.                   </thead>
24.                   <tbody>
25.                      <tr th:each="creneau,iter : ${agenda.creneaux}">
26.                         <td>
27.                            <span th:if="${creneau.action}==1" class="status-metro status-active"
    th:text="${creneau.creneauHoraire}">Créneau horaire</span>
28.                            <span th:if="${creneau.action}==2" class="status-metro status-suspended"
    th:text="${creneau.creneauHoraire}">Créneau horaire</span>
29.                         </td>
30.                         <td>
31.                            <span th:text="${creneau.client}">Client</span>
32.                         </td>
33.                         <td>
34.                            <a th:if="${creneau.action}==1" th:href="@{'javascript:reserverCreneau('+$
    {creneau.id}+')'}" th:text="${creneau.commande}"
35.                               class="status-metro status-active">Réserver
36.                            </a>
37.                            <a th:if="${creneau.action}==2" th:href="@{'javascript:supprimerRv('+$
    {creneau.idRv}+')'}" th:text="${creneau.commande}"
38.                               class="status-metro status-suspended">Supprimer
39.                            </a>
40.                         </td>
41.                      </tr>
42.                   </tbody>
43.                </table>
44.             </div>
45.          </div>
46.          <!-- réservation -->
47.          <section th:include="resa" />
48.       </th:block>
49.       <!-- init page -->
50.       <script th:inline="javascript">
51.          /*<![CDATA[*/
52.          // on initialise la page
53.          initAgenda();
54.       /*]]>*/
55.       </script>
56.    </body>
57. </html>
```

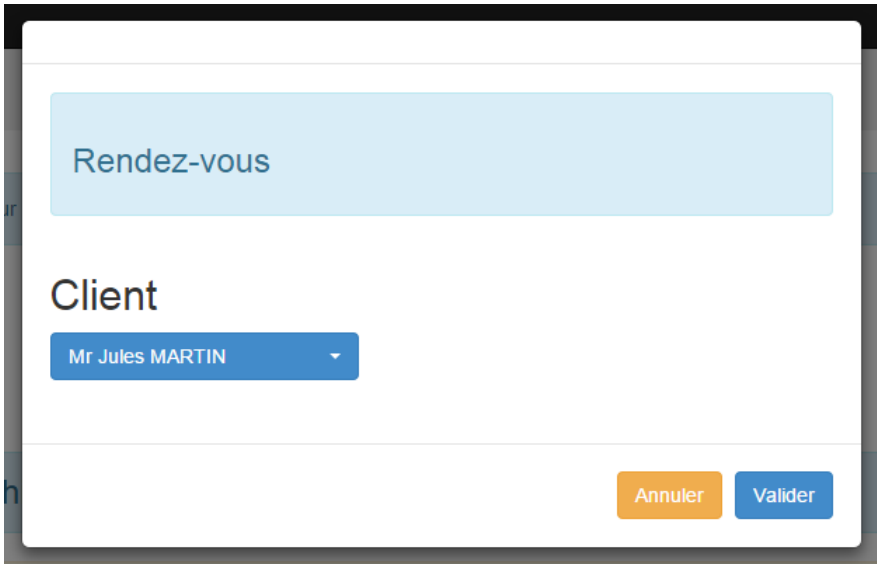
Le modèle de cette vue n'a qu'un élément :

- [agenda] (ligne 4) : un modèle un peu complexe spécialement construit pour l'affichage de l'agenda ;

Elle a les gestionnaires d'événements suivants :

évt	gestionnaire
clic sur le bouton [Supprimer]	*supprimerRv(idRv) - ligne 37*
clic sur le lien [Réserver]	*reserverCreneau(idCreneau) - ligne 34*

La vue [resa] de la ligne 47 est la vue qui est affichée lorsque l'utilisateur clique sur un lien [Réserver] :



Son code [resa.xml] est le suivant :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
   <body>
      <div id="resa" class="modal fade">
         <div class="modal-dialog">
            <div class="modal-content">
               <div class="modal-header">
                  <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                     <span aria-hidden="true">
                     </span>
                  </button>
                  <!-- <h4 class="modal-title">Modal title</h4> -->
               </div>
               <div class="modal-body">
                  <div class="alert alert-info">
                     <h3>
                        <span th:text="#{resa.titre}">Prise de rendez-vous</span>
                     </h3>
                  </div>
                  <div class="row">
                     <div class="col-md-3">
                        <h2 th:text="#{resa.client}">Client</h2>
                        <select name="idClient" id="idClient" class="combobox" data-style="btn-primary">
                           <option th:each="clientItem : ${clientItems}" th:text="${clientItem.texte}"
th:value="${clientItem.id}" />
                        </select>
                     </div>
                  </div>
               </div>
               <div class="modal-footer">
                  <button type="button" class="btn btn-warning" onclick="javascript:cancelDialogResa()"
th:text="#{resa.annuler}">Annuler</button>
                  <button type="button" class="btn btn-primary" onclick="javascript:validerRv()"
th:text="#{resa.valider}">Valider</button>
               </div>
            </div><!-- /.modal-content -->
         </div><!-- /.modal-dialog -->
      </div><!-- /.modal -->
      <!-- init page -->
      <script th:inline="javascript">
         /*<![CDATA[*/
         // on initialise la page
         initResa();
      /*]]>*/
      </script>
   </body>
</html>
```

Son modèle n'a qu'un élément :

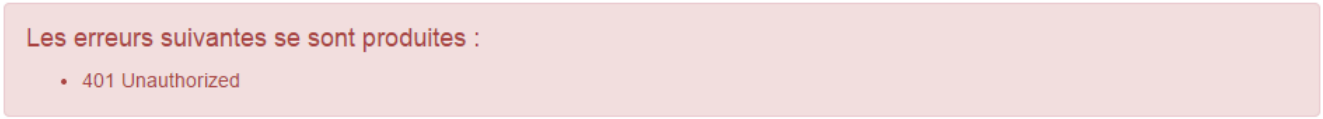
- [*clientItems*] (ligne 24) : la liste des clients ;

Elle a les gestionnaires d'événements suivants :

évt	gestionnaire
clic sur le bouton [Annuler]	*cancelDialogResa() - ligne 30*
clic sur le bouton [Valider]	*validerRv() - ligne 31*

8.6.5.6 *La vue [erreurs]*

C'est la vue qui s'affiche si l'action demandée par l'utilisateur n'a pu aboutir :



Le code [erreurs.xml] est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <section xmlns:th="http://www.thymeleaf.org">
3.     <div class="alert alert-danger">
4.        <h4>
5.           <span th:text="#{erreurs.titre}">Les erreurs suivantes se sont produites :</span>
6.        </h4>
7.        <ul>
8.           <li th:each="message : ${erreurs}" th:text="${message}" />
9.        </ul>
10.    </div>
11. </section>
```

Son modèle n'a qu'un élément :

- [*erreurs*] (ligne 8) : la liste des erreurs à afficher ;

La vue n'a pas de gestionnaire d'événements.

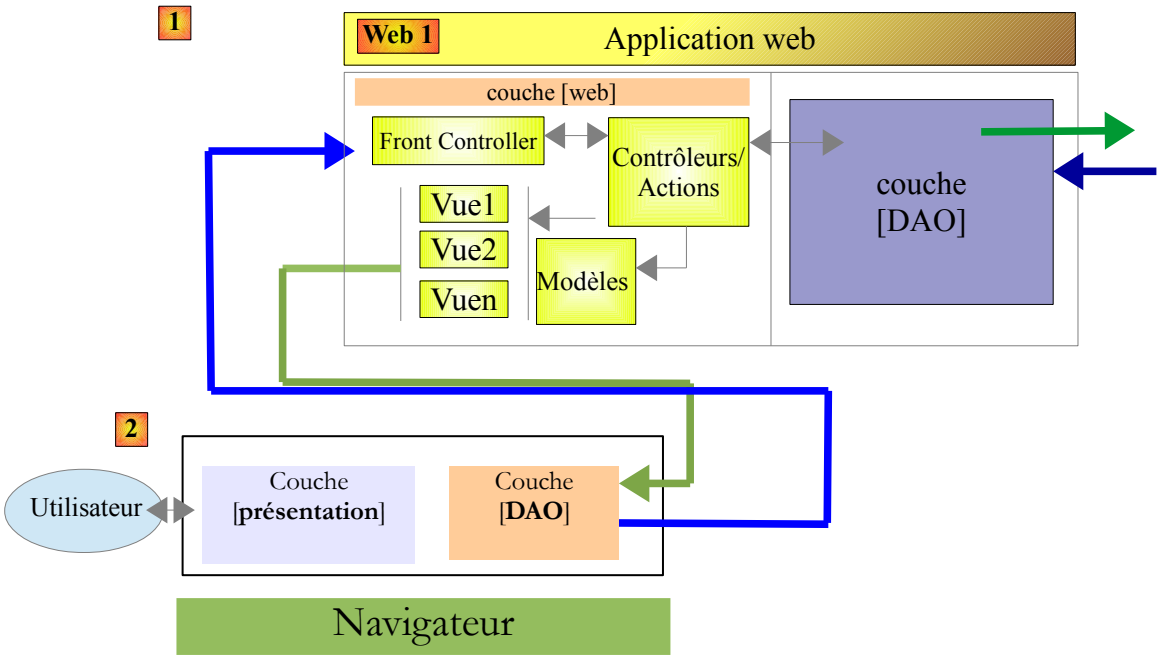
8.6.5.7 *Résumé*

Le tableau suivant redonne les vues et leurs modèles :

vue	modèle	gestionnaires d'événements
navbar-start		*connecter, setLang*
jumbotron		
login		
navbar-run		*deconnecter, setLang*
accueil	*rdvmedecins.medecinItems (liste des médecins)*	getAgenda
agenda	*agenda (une journée de l'agenda)*	supprimerRv, reserverCreneau
resa	*clientItems (liste des clients)*	cancelDialogResa, validerRv
erreurs	*erreurs (liste d'erreurs)*	

8.6.6 Étape 3 : écriture des actions

Revenons à l'architecture du service web [Web1] :



Nous allons voir maintenant quelles URL sont exposées par [Web1] et leur implémentation :

8.6.6.1 Les URL exposées par le service [Web1]

Ce sont les suivantes :

- une URL pour chacune des vues précédentes ou une composition de celles-ci ;
- une URL pour ajouter un RV ;
- une URL pour supprimer un RV ;

Elles rendent toutes une réponse du type [Reponse] suivant :

```
1.  public class Reponse {
2.
3.      // ----------------- propriétés
4.      // statut de l'opération
5.      private int status;
6.      // la barre de navigation
7.      private String navbar;
8.      // le jumbotron
9.      private String jumbotron;
10.     // le corps de la page
11.     private String content;
12.     // l'agenda
13.     private String agenda;
14. ...
15. }
```

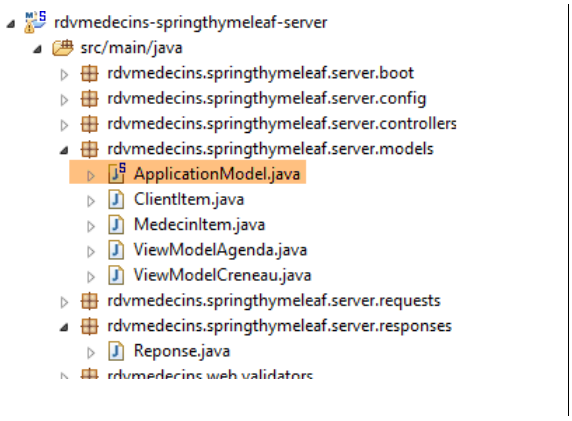
- ligne 5 : un état de la réponse : 1 (OK), 2 (erreur) ;
- ligne 7 : le flux HTML des vues [navbar-start] ou [navbar-run] selon les cas ;
- ligne 9 : le flux HTML de la vue [jumbotron] ;
- ligne 13 : le flux HTML de la vue [agenda] ;
- ligne 9 : le flux HTML des vues [accueil], [erreurs], [login] selon les cas ;

Les URL exposées sont les suivantes

`/getNavbarStart`	met la vue [navbar-start] dans [Reponse.navbar]

/getNavbarRun	met la vue [navbar-run] dans [Reponse.navbar]
/getAccueil	met la vue [accueil] dans [Reponse.content]
/getJumbotron	met la vue [jumbotron] dans [Reponse.jumbotron]
/getAgenda	met la vue [agenda] dans [Reponse.agenda]
/getLogin	met la vue [login] dans [Reponse.content]
/getNavbarRunJumbotronAccueil	• si connexion réussie, met la vue [navbar-run] dans [Reponse.navbar], la vue [jumbotron] dans [Reponse.jumbotron], la vue [accueil] dans [Reponse.content] • si connexion ratée, met la vue [erreurs] dans [Reponse.content] et [Reponse.status] à 2
/getNavbarRunJumbotronAccueilAgenda	met la vue [navbar-run] dans [Reponse.navbar], la vue [jumbotron] dans [Reponse.jumbotron], la vue [accueil] dans [Reponse.content], la vue [agenda] dans [Reponse.agenda]
/ajouterRv	ajoute le rendez-vous sélectionné et met le nouvel agenda dans [Reponse.agenda]
/supprimerRv	supprime le rendez-vous sélectionné et met le nouvel agenda dans [Reponse.agenda]

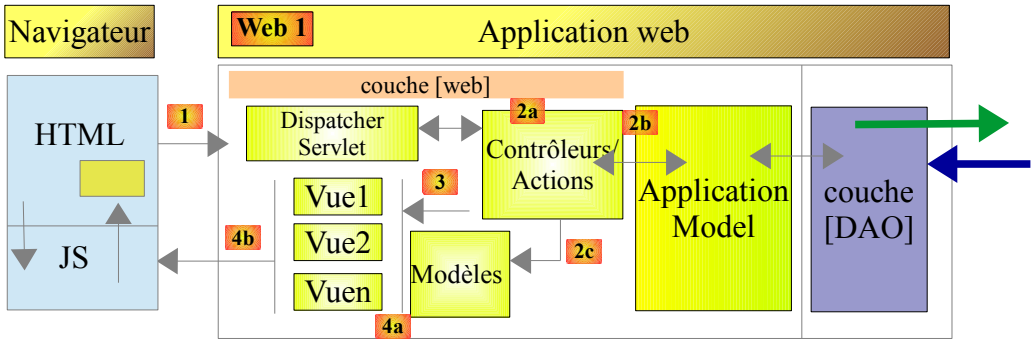
8.6.6.2 Le singleton [ApplicationModel]



La classe [ApplicationModel] est instanciée en un unique exemplaire et injectée dans le contrôleur de l'application. Son code est le suivant :

```
1.  package rdvmedecins.springthymeleaf.server.models;
2.
3.  import java.util.ArrayList;
4.  ...
5.
6.  @Component
7.  public class ApplicationModel implements IDao {
8.
9.  ....
10. }
```

- ligne 6 : [ApplicationModel] est un composant Spring ;
- ligne 7 : qui implémente l'interface de la couche [DAO]. Nous faisons cela pour que les actions n'aient pas à connaître la couche [DAO] mais seulement le singleton [ApplicationModel]. L'architecture de [Web1] devient alors la suivante :



Revenons sur le code de la classe [ApplicationModel] :

```
1.  package rdvmedecins.springthymeleaf.server.models;
2.
3.  import java.util.ArrayList;
4.  ...
5.
6.  @Component
7.  public class ApplicationModel implements IDao {
8.
9.      // la couche [DAO]
10.     @Autowired
11.     private IDao dao;
12.     // la configuration
13.     @Autowired
14.     private AppConfig appConfig;
15.
16.     // données provenant de la couche [DAO]
17.     private List<ClientItem> clientItems;
18.     private List<MedecinItem> medecinItems;
19.     // données de configuration
20.     private String userInit;
21.     private String mdpUserInit;
22.     private boolean corsAllowed;
23.     // exception
24.     private RdvMedecinsException rdvMedecinsException;
25.
26.     // constructeur
27.     public ApplicationModel() {
28.     }
29.
30.     @PostConstruct
31.     public void init() {
32.        // config
33.        userInit = appConfig.getUSER_INIT();
34.        mdpUserInit = appConfig.getMDP_USER_INIT();
35.        dao.setTimeout(appConfig.getTIMEOUT());
36.        dao.setUrlServiceWebJson(appConfig.getWEBJSON_ROOT());
37.        corsAllowed = appConfig.isCORS_ALLOWED();
38.        // on met en cache les listes déroulantes des médecins et des clients
39.        List<Medecin> medecins = null;
40.        List<Client> clients = null;
41.        try {
42.           medecins = dao.getAllMedecins(new User(userInit, mdpUserInit));
43.           clients = dao.getAllClients(new User(userInit, mdpUserInit));
44.        } catch (RdvMedecinsException ex) {
45.           rdvMedecinsException = ex;
46.        }
47.        if (rdvMedecinsException == null) {
48.           // on crée les éléments des listes déroulantes
49.           medecinItems = new ArrayList<MedecinItem>();
50.           for (Medecin médecin : medecins) {
51.              medecinItems.add(new MedecinItem(médecin));
52.           }
53.           clientItems = new ArrayList<ClientItem>();
54.           for (Client client : clients) {
55.              clientItems.add(new ClientItem(client));
56.           }
57.        }
```

```
58.     }
59.
60.     // getters et setters
61.     ...
62.
63.     // implémentation interface [IDao]
64.     @Override
65.     public void setUrlServiceWebJson(String url) {
66.         dao.setUrlServiceWebJson(url);
67.     }
68.
69.     @Override
70.     public void setTimeout(int timeout) {
71.         dao.setTimeout(timeout);
72.     }
73.
74.     @Override
75.     public Rv ajouterRv(User user, String jour, long idCreneau, long idClient) {
76.         return dao.ajouterRv(user, jour, idCreneau, idClient);
77.     }
78.
79.     ...
80. }
```

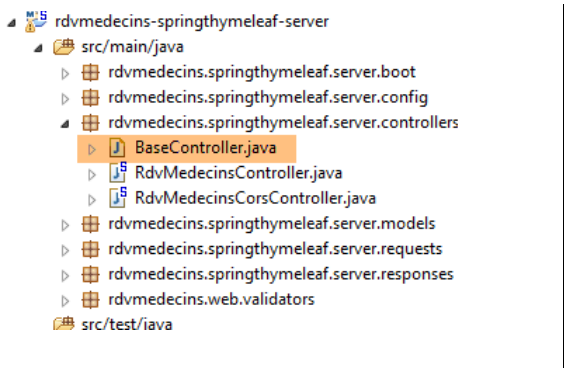
- ligne 11 : injection de la référence de l'implémentation de la couche [DAO]. C'est ensuite cette référence qui est utilisée pour implémenter l'interface [IDao] (lignes 64-80) ;
- ligne 14 : injection de la configuration de l'application ;
- lignes 33-37 : utilisation de cette configuration pour configurer divers éléments de l'architecture de l'application ;
- lignes 38-46 : on met en cache les informations qui vont alimenter les listes déroulantes des médecins et des clients. Nous faisons donc l'hypothèse que si un médecin ou un client change, l'application doit être rebootée. L'idée ici est de montrer qu'un singleton Spring peut servir de cache à l'application web ;

Les classes [MedecinItem] et [ClientItem] dérivent toutes deux de la classe [PersonneItem] suivante :

```
1.  package rdvmedecins.springthymeleaf.server.models;
2.
3.  import rdvmedecins.client.entities.Personne;
4.
5.  public class PersonneItem {
6.
7.      // élément d'une liste
8.      private Long id;
9.      private String texte;
10.
11.     // constructeur
12.     public PersonneItem() {
13.
14.     }
15.
16.     public PersonneItem(Personne personne) {
17.         id = personne.getId();
18.         texte = String.format("%s %s %s", personne.getTitre(), personne.getPrenom(), personne.getNom());
19.     }
20.
21.     // getters et setters
22. ...
23. }
```

- ligne 8 : le champ [id] sera la valeur de l'attribut [value] d'une option de la liste déroulante ;
- ligne 9 : le champ [texte] sera le texte affiché par une option de la liste déroulante ;

8.6.6.3 La classe [BaseController]



La classe [BaseController] est la classe parent des contrôleurs [RdvMedecinsController] et [RdvMedecinsCorsController]. Il n'était pas obligatoire de créer cette classe parent. On y a rassemblé des méthodes utilitaires de la classe [RdvMedecinsController] pas fondamentales sauf une. On peut les classer dans trois ensembles :

1. les méthodes utilitaires ;
2. les méthodes qui rendent les vues fusionnées avec leurs modèles ;
3. la méthode d'initialisation d'une action

`protected List<String> getErreursForException(Exception exception)`
 	
`protected List<String> getErreursForModel(BindingResult result, Locale locale, WebApplicationContext ctx)`
	deux méthodes utilitaires qui fournissent une liste de messages d'erreur. Nous les avons déjà rencontrées et utilisées ;

`protected String getPartialViewAccueil(WebContext thymeleafContext)`	rend la vue [accueil] sans modèle
`protected String getPartialViewAgenda(ActionContext actionContext, AgendaMedecinJour agenda, Locale locale)`	rend la vue [agenda] et son modèle
`protected String getPartialViewLogin(WebContext thymeleafContext)`	rend la vue [login] sans modèle
`protected Reponse getViewErreurs(WebContext thymeleafContext, List<String> erreurs)`	rend la réponse au client lorsque l'action demandée s'est terminée par une erreur
`protected ActionContext getActionContext(String lang, String origin, HttpServletRequest request, HttpServletResponse response, BindingResult result, RdvMedecinsCorsController rdvMedecinsCorsController)`	la méthode d'initialisation de toutes les actions du contrôleur [RdvMedecinsController]

Examinons deux de ces méthodes.

La méthode [getPartialViewAgenda] rend la vue la plus complexe à générer, celle de l'agenda. Son code est le suivant :

```
1.     // flux [agenda]
2.     protected String getPartialViewAgenda(ActionContext actionContext, AgendaMedecinJour agenda, Locale
    locale) {
3.         // contextes
4.         WebContext thymeleafContext = actionContext.getThymeleafContext();
5.         WebApplicationContext springContext = actionContext.getSpringContext();
6.         // on construit le modèle de la page [agenda]
7.         ViewModelAgenda modelAgenda = setModelforAgenda(agenda, springContext, locale);
8.         // l'agenda avec son modèle
9.         thymeleafContext.setVariable("agenda", modelAgenda);
10.        thymeleafContext.setVariable("clientItems", application.getClientItems());
11.        return engine.process("agenda", thymeleafContext);
12. }
```

- lignes 9-10 : les deux éléments du modèle de l'agenda :
  - ligne 9 : l'agenda affiché.
  - ligne 10 : la liste des clients affichée lorsque l'utilisateur prend un reendez-vous ;

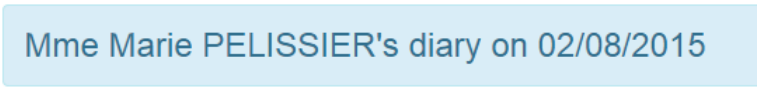
La méthode [setModelforAgenda] de la ligne 7 est la suivante :

```
1.  // modèle de la page [Agenda]
```

```
2.     private ViewModelAgenda setModelforAgenda(AgendaMedecinJour agenda, WebApplicationContext
    springContext, Locale locale) {
3.         // le titre de la page
4.         String dateFormat = springContext.getMessage("date.format", null, locale);
5.         Medecin médecin = agenda.getMedecin();
6.         String titre = springContext.getMessage("agenda.titre", new String[] { médecin.getTitre(),
    médecin.getPrenom(),
7.                 médecin.getNom(), new SimpleDateFormat(dateFormat).format(agenda.getJour()) }, locale);
8.         // les créneaux de réservation
9.         ViewModelCreneau[] modelCréneaux = new ViewModelCreneau[agenda.getCreneauxMedecinJour().length];
10.        int i = 0;
11.        for (CreneauMedecinJour creneauMedecinJour : agenda.getCreneauxMedecinJour()) {
12.           // créneau du médecin
13.           Creneau créneau = creneauMedecinJour.getCreneau();
14.           ViewModelCreneau modelCréneau = new ViewModelCreneau();
15.           modelCréneaux[i] = modelCréneau;
16.           // id
17.           modelCréneau.setId(créneau.getId());
18.           // créneau horaire
19.           modelCréneau.setCreneauHoraire(String.format("%02dh%02d-%02dh%02d", créneau.getHdebut(),
    créneau.getMdebut(),
20.                 créneau.getHfin(), créneau.getMfin()));
21.           Rv rv = creneauMedecinJour.getRv();
22.           // client et commande
23.           String commande;
24.           if (rv == null) {
25.              modelCréneau.setClient("");
26.              commande = springContext.getMessage("agenda.reserver", null, locale);
27.              modelCréneau.setCommande(commande);
28.              modelCréneau.setAction(ViewModelCreneau.ACTION_RESERVER);
29.
30.           } else {
31.              Client client = rv.getClient();
32.              modelCréneau.setClient(String.format("%s %s %s", client.getTitre(), client.getPrenom(),
    client.getNom()));
33.              commande = springContext.getMessage("agenda.supprimer", null, locale);
34.              modelCréneau.setCommande(commande);
35.              modelCréneau.setIdRv(rv.getId());
36.              modelCréneau.setAction(ViewModelCreneau.ACTION_SUPPRIMER);
37.           }
38.           // créneau suivant
39.           i++;
40.        }
41.        // on rend le modèle de l'agenda
42.        ViewModelAgenda modelAgenda = new ViewModelAgenda();
43.        modelAgenda.setTitre(titre);
44.        modelAgenda.setCreneaux(modelCréneaux);
45.        return modelAgenda;
46.    }
```

- ligne 6 : l'agenda a un titre :

ou bien :



On voit que le format de la date dépend de la langue. On va chercher ce format dans les fichiers de messages (ligne 4).

- lignes 11-40 : pour chaque créneau, on doit afficher la vue :

ou bien la vue :

- lignes 19-20 : affichent le créneau horaire ;
- lignes 25-28 : le cas où le créneau est libre. Il faut alors afficher le bouton [Réserver] ;
- lignes 31-36 : le cas où le créneau est occupé. Il faut alors afficher et le client et le bouton [Supprimer] ;

L'autre méthode sur laquelle nous donnons davantage d'explications est la méthode [getActionContext]. Elle est appelée au début de chacune des actions de [RdvMedecinsController]. Sa signature est la suivante :

```
protected ActionContext getActionContext(String lang, String origin, HttpServletRequest
request,HttpServletResponse response, BindingResult result, RdvMedecinsCorsController rdvMedecinsCorsController)
```

Elle rend le type [ActionContext] suivant :

```
1.  public class ActionContext {
2.
3.      // data
4.      private WebContext thymeleafContext;
5.      private WebApplicationContext springContext;
6.      private Locale locale;
7.      private List<String> erreurs;
8.  ...
9.  }
```

- ligne 4 : le contexte Thymeleaf de l'action ;
- ligne 5 : le contexte Spring de l'action ;
- ligne 6 : la locale de l'action ;
- ligne 7 : une éventuelle liste de messages d'erreurs ;

Ses paramètres sont les suivants :

- [lang] : la langue demandée pour l'action 'en' ou 'fr' ;
- [origin] : l'entête HTTP [origin] dans le cas d'un appel inter-domaines ;
- [request] : la requête HTTP en cours de traitement, ce qu'on appelle depuis un moment une action ;
- [response] : la réponse qui va être faite à cette requête ;
- [result] : chaque action de [RdvMedecinsController] reçoit une valeur postée dont on teste la validité. [result] est le résultat de ce test ;
- [rdvMedecinsController] : le contrôleur conteneur des actions ;

La méthode [getActionContext] est implémentée de la façon suivante :

```
1.      // contexte d'une action
2.      protected ActionContext getActionContext(String lang, String origin, HttpServletRequest
    request,HttpServletResponse response, BindingResult result, RdvMedecinsCorsController
    rdvMedecinsCorsController) {
3.          // langue ?
4.          if (lang == null) {
5.              lang = "fr";
6.          }
7.          // locale
8.          Locale locale = null;
9.          if (lang.trim().toLowerCase().equals("fr")) {
10.             // français
11.             locale = new Locale("fr", "FR");
12.         } else {
13.             // tout le reste en anglais
14.             locale = new Locale("en", "US");
15.         }
16.         // entêtes CORS
17.         rdvMedecinsCorsController.sendOptions(origin, response);
18.         // ActionContext
```

```
19.       ActionContext actionContext = new ActionContext(new WebContext(request, response,
    request.getServletContext(),locale),
    WebApplicationContextUtils.getWebApplicationContext(request.getServletContext()), locale, null);
20.       // erreurs d'initialisation
21.       RdvMedecinsException e = application.getRdvMedecinsException();
22.       if (e != null) {
23.          actionContext.setErreurs(e.getMessages());
24.          return actionContext;
25.       }
26.       // erreurs de POST ?
27.       if (result != null && result.hasErrors()) {
28.          actionContext.setErreurs(getErreursForModel(result, locale, actionContext.getSpringContext()));
29.          return actionContext;
30.       }
31.       // pas d'erreurs
32.       return actionContext;
33. }
```

- lignes 3-15 : à partir du paramètre [lang], on fixe la locale de l'action ;
- ligne 17 : on envoie les entêtes HTTP nécessaires aux requêtes inter-domaines. Nosu ne détaillons pas. La technique utilisée est celle du paragraphe 8.4.13, page 452 ;
- ligne 19 : cosntruction d'un objet [ActionContext] sans erreurs ;
- ligne 21 : nous avons vu au paragraphe 8.6.6.2, page 545 que le singleton [ApplicationModel] accédait à la base de données pour récupérer et les clients et les médecins. Cet accès peut échouer. On mémorise alors l'exception qui se produit. Ligne 21, nous récupérons cette exception ;
- lignes 22-25 : s'il y a eu exception au boot de l'application, toute action est impossible. On rend alors pour toute action un objet [ActionContext] avec les messages d'erreur de l'exception ;
- ligne 27-20 : on analyse le paramètre [result] pour savoir si la valeur postée était valide ou non. Si elle était invalide, on rend un objet [ActionContext] avec les messages d'erreur appropriés ;
- ligne 32 : cas sans erreurs ;

Nous examinons maintenant les actions du contrôleur [RdvMedecinsController]

8.6.6.4 L'action [/getNavBarStart]

L'action [/getNavBarStart] rend la vue [navbar-start]. Sa signature est la suivante :

```
1.  @RequestMapping(value = "/getNavbarStart", method = RequestMethod.POST)
2.     @ResponseBody
3.     public Reponse getNavbarStart(@Valid @RequestBody PostLang postLang, BindingResult result,
       HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin)
```

Elle rend le type [Reponse] suivant :

```
1.  public class Reponse {
2.
3.     // ----------------- propriétés
4.     // statut de l'opération
5.     private int status;
6.     // la barre de navigation
7.     private String navbar;
8.     // le jumbotron
9.     private String jumbotron;
10.    // le corps de la page
11.    private String content;
12.    // l'agenda
13.    private String agenda;
14. ...
15. }
```

et a les paramètres suivants :

- [PostLang postlang] : la valeur postée suivante :

```
1.  public class PostLang {
2.
3.     // data
4.     @NotNull
5.     private String lang;
```

```
6.  ...
7.  }
```

La classe [PostLang] est la classe parent de toutes les valeurs postées. En effet, le client doit toujours préciser la langue avec laquelle doit s'exécuter l'action.

La méthode [getNavbarStart] est implémentée de la façon suivante :

```
5.      // navbar-start
6.      @RequestMapping(value = "/getNavbarStart", method = RequestMethod.POST)
7.      @ResponseBody
8.      public Reponse getNavbarStart(@Valid @RequestBody PostLang postLang, BindingResult result,
        HttpServletRequest request, HttpServletResponse response,
9.              @RequestHeader(value = "Origin", required = false) String origin) {
10.        // contextes de l'action
11.        ActionContext actionContext = getActionContext(postLang.getLang(), origin, request, response,
    result,rdvMedecinsCorsController);
12.        WebContext thymeleafContext = actionContext.getThymeleafContext();
13.        // erreurs ?
14.        List<String> erreurs = actionContext.getErreurs();
15.        if (erreurs != null) {
16.           return getViewErreurs(thymeleafContext, erreurs);
17.        }
18.        // on renvoie la vue [navbar-start]
19.        Reponse reponse = new Reponse();
20.        reponse.setStatus(1);
21.        reponse.setNavbar(engine.process("navbar-start", thymeleafContext));
22.        return reponse;
23. }
```

- ligne 7 : initialisation de l'action ;
- lignes 10-13 : si la méthode d'initialisation de l'action a signalé des erreurs, on les envoie dans la réponse au client (ligne 12) avec le status 2 :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- lignes 15-18 : on envoie la vue [navbar-start] avec le status 1 :

```
{"status":1,"navbar": navbar-start, "jumbotron": null, "agenda":null, "content":null}
```

Dans la suite, nous ne détaillons que les nouveautés.

8.6.6.5 L'action [/getNavbarRun]

L'action [/getNavBarRun] rend la vue [navbar-run] :

```
1.      // navbar-run
2.      @RequestMapping(value = "/getNavbarRun", method = RequestMethod.POST)
3.      @ResponseBody
4.      public Reponse getNavbarRun(@Valid @RequestBody PostLang postLang, BindingResult result,
    HttpServletRequest request,
5.              HttpServletResponse response, @RequestHeader(value = "Origin", required = false) String origin) {
6.        // contextes de l'action
7.        ActionContext actionContext = getActionContext(postLang.getLang(), origin, request, response,
    result,rdvMedecinsCorsController);
8.        WebContext thymeleafContext = actionContext.getThymeleafContext();
9.        // erreurs ?
10.       List<String> erreurs = actionContext.getErreurs();
11.       if (erreurs != null) {
12.          return getViewErreurs(thymeleafContext, erreurs);
13.       }
14.       // on renvoie la vue [navbar-run]
15.       Reponse reponse = new Reponse();
16.       reponse.setStatus(1);
17.       reponse.setNavbar(engine.process("navbar-run", thymeleafContext));
18.       return reponse;
19. }
```

L'action peut rendre deux types de réponse :

- la réponse avec erreur (lignes 10-13) :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- la réponse avec la vue [navbar-run] :

```
{"status":1,"navbar": navbar-run, "jumbotron": null, "agenda":null, "content":null}
```

8.6.6.6 L'action [/getJumbotron]

L'action [/getJumbotron] rend la vue [jumbotron] :

```
1.      // jumbotron
2.      @RequestMapping(value = "/getJumbotron", method = RequestMethod.POST)
3.      @ResponseBody
4.      public Reponse getJumbotron(@Valid @RequestBody PostLang postLang, BindingResult result,
    HttpServletRequest request,
5.              HttpServletResponse response, @RequestHeader(value = "Origin", required = false) String
    origin) {
6.          // contextes de l'action
7.          ActionContext actionContext = getActionContext(postLang.getLang(), origin, request, response,
    result,rdvMedecinsCorsController);
8.          WebContext thymeleafContext = actionContext.getThymeleafContext();
9.          // erreurs ?
10.         List<String> erreurs = actionContext.getErreurs();
11.         if (erreurs != null) {
12.            return getViewErreurs(thymeleafContext, erreurs);
13.         }
14.         // on renvoie la vue [jumbotron]
15.         Reponse reponse = new Reponse();
16.         reponse.setStatus(1);
17.         reponse.setJumbotron(engine.process("jumbotron", thymeleafContext));
18.         return reponse;
19. }
```

L'action peut rendre deux types de réponse :

- la réponse avec erreur (lignes 10-13) :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- la réponse avec la vue [jumbotron] :

```
{"status":1,"navbar": null, "jumbotron": jumbotron, "agenda":null, "content":null}
```

8.6.6.7 L'action [/getLogin]

L'action [/getLogin] rend la vue [login] :

```
1.  @RequestMapping(value = "/getLogin", method = RequestMethod.POST)
2.      @ResponseBody
3.      public Reponse getLogin(@Valid @RequestBody PostLang postLang, BindingResult result, HttpServletRequest
    request,
4.              HttpServletResponse response, @RequestHeader(value = "Origin", required = false) String origin) {
5.          // contextes de l'action
6.          ActionContext actionContext = getActionContext(postLang.getLang(), origin, request, response,
    result,rdvMedecinsCorsController);
7.          WebContext thymeleafContext = actionContext.getThymeleafContext();
8.          // erreurs ?
9.          List<String> erreurs = actionContext.getErreurs();
10.         if (erreurs != null) {
11.            return getViewErreurs(thymeleafContext, erreurs);
12.         }
13.         // on renvoie la vue [login]
14.         Reponse reponse = new Reponse();
15.         reponse.setStatus(1);
16.         reponse.setJumbotron(engine.process("jumbotron", thymeleafContext));
17.         reponse.setNavbar(engine.process("navbar-start", thymeleafContext));
18.         reponse.setContent(getPartialViewLogin(thymeleafContext));
19.         return reponse;
20.     }
```

L'action peut rendre deux types de réponse :

- la réponse avec erreur (lignes 9-11) :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- la réponse avec la vue [login] :

```
{"status":1,"navbar": navbar-start, "jumbotron": jumbotron, "agenda":null, "content":login}
```

8.6.6.8 L'action [/getAccueil]

L'action [/getAccueil] rend la vue [accueil]. Sa signature est la suivante :

```
1.     @RequestMapping(value = "/getAccueil", method = RequestMethod.POST)
2.     @ResponseBody
3.     public Reponse getAccueil(@Valid @RequestBody PostUser postUser, BindingResult result,
    HttpServletRequest request,HttpServletResponse response, @RequestHeader(value = "Origin", required =
    false) String origin)
```

- ligne 3, la valeur postée est de type [PostUser] suivant :

```
1.  public class PostUser extends PostLang {
2.     // data
3.     @NotNull
4.     private User user;
5.  ...
6.  }
```

- ligne 1 : la classe [PostUser] étend la classe [PostLang] et donc embarque une langue ;
- ligne 4 : l'utilisateur qui cherche à obtenir la vue ;

Le code d'implémentation est le suivant :

```
1.     @RequestMapping(value = "/getAccueil", method = RequestMethod.POST)
2.     @ResponseBody
3.     public Reponse getAccueil(@Valid @RequestBody PostUser postUser, BindingResult result,
    HttpServletRequest request,
4.           HttpServletResponse response, @RequestHeader(value = "Origin", required = false) String
    origin) {
5.        // contextes de l'action
6.        ActionContext actionContext = getActionContext(postUser.getLang(), origin, request, response,
    result,rdvMedecinsCorsController);
7.        WebContext thymeleafContext = actionContext.getThymeleafContext();
8.        // erreurs ?
9.        List<String> erreurs = actionContext.getErreurs();
10.       if (erreurs != null) {
11.          return getViewErreurs(thymeleafContext, erreurs);
12.       }
13.       // la vue [accueil] est protégée
14.       try{
15.          // utilisateur
16.          User user = postUser.getUser();
17.          // on vérifie les identifiants [userName, password]
18.          application.authenticate(user);
19.       }catch(RdvMedecinsException e){
20.          // on renvoie une erreur
21.          return getViewErreurs(thymeleafContext, e.getMessages());
22.       }
23.       // on renvoie la vue [accueil]
24.       Reponse reponse = new Reponse();
25.       reponse.setStatus(1);
26.       reponse.setContent(getPartialViewAccueil(thymeleafContext));
27.       return reponse;
28. }
```

- lignes 15-22 : on notera que la page [accueil] est protégée et que donc l'utilisateur doit être authentifié ;

L'action peut rendre deux types de réponse :

- la réponse avec erreur (lignes 11 et 21) :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- la réponse avec la vue [accueil] (lignes 24-27) :

```
{"status":1,"navbar": null, "jumbotron": null, "agenda":null, "content":accueil}
```

8.6.6.9 L'action [/getNavbarRunJumbotronAccueil]

L'action [/getNavbarRunJumbotronAccueil] rend les vues [navbar-run, jumbotron, accueil]. Elle a la signature suivante :

```
1.  @RequestMapping(value = "/getNavbarRunJumbotronAccueil", method = RequestMethod.POST,
    consumes = "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse getNavbarRunJumbotronAccueil(@Valid @RequestBody PostUser post,
    BindingResult result, HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin)
```

- ligne 3 : la valeur postée est du type [PostUser] ;

L'implémentation de l'action est la suivante :

```
1.  // navbar+ jumbotron + accueil
2.     @RequestMapping(value = "/getNavbarRunJumbotronAccueil", method = RequestMethod.POST, consumes =
    "application/json; charset=UTF-8")
3.     @ResponseBody
4.     public Reponse getNavbarRunJumbotronAccueil(@Valid @RequestBody PostUser postUser, BindingResult
    result, HttpServletRequest request, HttpServletResponse response,
5.           @RequestHeader(value = "Origin", required = false) String origin) {
6.        // contextes de l'action
7.        ActionContext actionContext = getActionContext(postUser.getLang(), origin, request, response,
    result,
8.              rdvMedecinsCorsController);
9.        WebContext thymeleafContext = actionContext.getThymeleafContext();
10.       // erreurs ?
11.       List<String> erreurs = actionContext.getErreurs();
12.       if (erreurs != null) {
13.          return getViewErreurs(thymeleafContext, erreurs);
14.       }
15.       // la vue [accueil] est protégée
16.       try {
17.          // utilisateur
18.          User user = postUser.getUser();
19.          // on vérifie les identifiants [userName, password]
20.          application.authenticate(user);
21.       } catch (RdvMedecinsException e) {
22.          // on renvoie une erreur
23.          return getViewErreurs(thymeleafContext, e.getMessages());
24.       }
25.       // on envoie la réponse
26.       Reponse reponse = new Reponse();
27.       reponse.setStatus(1);
28.       reponse.setNavbar(engine.process("navbar-run", thymeleafContext));
29.       reponse.setJumbotron(engine.process("jumbotron", thymeleafContext));
30.       reponse.setContent(getPartialViewAccueil(thymeleafContext));
31.       return reponse;
32.    }
```

L'action peut rendre deux types de réponse :

- la réponse avec erreur (lignes 13, 23) :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

- la réponse avec les vues [navbar-run, jumbotron, accueil] (lignes 26-31) :

```
{"status":1,"navbar": navbar-run, "jumbotron": jumbotron, "agenda":null, "content":accueil}
```

8.6.6.10 L'action [/getAgenda]

L'action [/getAgenda] rend la vue [agenda]. Sa signature est la suivante :

```
1.  @RequestMapping(value = "/getAgenda", method = RequestMethod.POST, consumes =
    "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse getAgenda(@RequestBody @Valid PostGetAgenda postGetAgenda,
    BindingResult result, HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin)
```

- ligne 3 : la valeur postée est de type [PostGetAgenda] suivant :

```
1.  public class PostGetAgenda extends PostUser {
2.
3.     // données
4.     @NotNull
5.     private Long idMedecin;
6.     @NotNull
7.     @DateTimeFormat(pattern = "yyyy-MM-dd")
8.     private Date jour;
9.  ...
10. }
```

- ligne 1 : la classe [PostGetAgenda] étend la classe [PostUser] et donc embarque une langue et un utilisateur ;
- ligne 5 : le n° du médecin duquel on veut l'agenda ;
- ligne 8 : la journée de l'agenda désirée ;

L'implémentation est la suivante :

```
1.  @RequestMapping(value = "/getAgenda", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.     @ResponseBody
3.     public Reponse getAgenda(@RequestBody @Valid PostGetAgenda postGetAgenda, BindingResult result,
       HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin) {
5.        // contextes de l'action
6.        ActionContext actionContext = getActionContext(postGetAgenda.getLang(), origin, request,
    response, result,  rdvMedecinsCorsController);
7.        WebContext thymeleafContext = actionContext.getThymeleafContext();
8.        WebApplicationContext springContext = actionContext.getSpringContext();
9.        Locale locale = actionContext.getLocale();
10.       // erreurs ?
11.       List<String> erreurs = actionContext.getErreurs();
12.       if (erreurs != null) {
13.          return getViewErreurs(thymeleafContext, erreurs);
14.       }
15.       // on vérifie la validité du post
16.       if (result != null) {
17.          new PostGetAgendaValidator().validate(postGetAgenda, result);
18.          if (result.hasErrors()) {
19.             // on retourne la vue [erreurs]
20.             return getViewErreurs(thymeleafContext, getErreursForModel(result, locale,
    springContext));
21.          }
22.       }
23.       ...
24. }
```

- jusqu'à la ligne 14, on a un code désormais classique ;
- lignes 16-21 : on fait une vérification supplémentaire sur la valeur postée. La date doit être postérieure ou égale à celle d'aujourd'hui. Pour le vérifier on utilise un validateur :

```
1.  package rdvmedecins.web.validators;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.
6.  import org.springframework.validation.Errors;
7.  import org.springframework.validation.Validator;
```

```
8.
9.  import rdvmedecins.springthymeleaf.server.requests.PostGetAgenda;
10. import rdvmedecins.springthymeleaf.server.requests.PostValiderRv;
11.
12. public class PostGetAgendaValidator implements Validator {
13.
14.    public PostGetAgendaValidator() {
15.    }
16.
17.    @Override
18.    public boolean supports(Class<?> classe) {
19.       return PostGetAgenda.class.equals(classe) || PostValiderRv.class.equals(classe);
20.    }
21.
22.    @Override
23.    public void validate(Object post, Errors errors) {
24.       // le jour choisi pour le rdv
25.       Date jour = null;
26.       if (post instanceof PostGetAgenda) {
27.          jour = ((PostGetAgenda) post).getJour();
28.       } else {
29.          if (post instanceof PostValiderRv) {
30.             jour = ((PostValiderRv) post).getJour();
31.          }
32.       }
33.       // on transforme les dates au format yyyy-MM-dd
34.       SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
35.       String strJour = sdf.format(jour);
36.       String strToday = sdf.format(new Date());
37.       // le jour choisi ne doit pas précéder la date d'aujourd'hui
38.       if (strJour.compareTo(strToday) < 0) {
39.          errors.rejectValue("jour", "todayandafter.postChoixMedecinJour", null, null);
40.       }
41.    }
42.
43. }
```

- ligne 19 : le validateur travaille pour deux classes : [PostGetAgenda] et [PostValiderRv] ;

Revenons au code de l'action [/getAgenda] :

```
1.  @RequestMapping(value = "/getAgenda", method = RequestMethod.POST, consumes =
    "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse getAgenda(@RequestBody @Valid PostGetAgenda postGetAgenda,
    BindingResult result, HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin) {
5.        ...
6.              // action
7.        try {
8.           // agenda du médecin
9.           AgendaMedecinJour agenda =
    application.getAgendaMedecinJour(postGetAgenda.getUser(), postGetAgenda.getIdMedecin(),
10.                new SimpleDateFormat("yyyy-MM-dd").format(postGetAgenda.getJour()));
11.          // réponse
12.          Reponse reponse = new Reponse();
13.          reponse.setStatus(1);
14.          reponse.setAgenda(getPartialViewAgenda(actionContext, agenda, locale));
15.          return reponse;
16.       } catch (RdvMedecinsException e1) {
17.          // on retourne la vue [erreurs]
18.          return getViewErreurs(thymeleafContext, e1.getMessages());
19.       } catch (Exception e2) {
20.          // on retourne la vue [erreurs]
21.          return getViewErreurs(thymeleafContext, getErreursForException(e2));
22.       }
23. }
```

- lignes 9-10 : avec les paramètres postés, on demande l'agenda du médecin ;

- lignes 12-13 : on rend l'agenda :

```
{"status":1,"navbar": null, "jumbotron": null, "agenda":agenda, "content":null}
```

- lignes 17, 21 : on rend une réponse avec erreurs :

```
{"status":2,"navbar": null, "jumbotron": null, "agenda":null, "content":erreurs}
```

8.6.6.11 L'action [/getNavbarRunJumbotronAccueilAgenda]

L'action [/getNavbarRunJumbotronAccueilAgenda] rend les vues [navbar-run, jumbotron, accueil, agenda]. Son implémentation est la suivante :

```
1.     @RequestMapping(value = "/getNavbarRunJumbotronAccueilAgenda", method = RequestMethod.POST, consumes
   = "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse getNavbarRunJumbotronAccueilAgenda(@Valid @RequestBody PostGetAgenda post,
   BindingResult result,
4.           HttpServletRequest request, HttpServletResponse response,
5.           @RequestHeader(value = "Origin", required = false) String origin) {
6.        // contextes de l'action
7.        ActionContext actionContext = getActionContext(post.getLang(), origin, request, response,
   result,rdvMedecinsCorsController);
8.        WebContext thymeleafContext = actionContext.getThymeleafContext();
9.        // erreurs ?
10.       List<String> erreurs = actionContext.getErreurs();
11.       if (erreurs != null) {
12.          return getViewErreurs(thymeleafContext, erreurs);
13.       }
14.       // agenda
15.       Reponse agenda = getAgenda(post, result, request, response, null);
16.       if (agenda.getStatus() != 1) {
17.          return agenda;
18.       }
19.       // on envoie la réponse
20.       Reponse reponse = new Reponse();
21.       reponse.setStatus(1);
22.       reponse.setNavbar(engine.process("navbar-run", thymeleafContext));
23.       reponse.setJumbotron(engine.process("jumbotron", thymeleafContext));
24.       reponse.setContent(getPartialViewAccueil(thymeleafContext));
25.       reponse.setAgenda(agenda.getAgenda());
26.       return reponse;
27. }
```

- lignes 15-18 : on profite de l'existence de l'action [/getAgenda] pour l'appeler. Ensuite on regarde le *status* de la réponse (ligne 16). Si on détecte une erreur, on ne va plus loin et on renvoie la réponse ;
- lignes 20 : on envoie les vues demandées :

```
{"status":1,"navbar": navbar-run, "jumbotron": jumbotron, "agenda":agenda, "content":accueil}
```

8.6.6.12 L'action [/supprimerRv]

L'action [/supprimerRv] permet de supprimer un rendez-vous. Sa signature est la suivante :

```
1.  @RequestMapping(value = "/supprimerRv", method = RequestMethod.POST, consumes =
    "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse supprimerRv(@Valid @RequestBody PostSupprimerRv postSupprimerRv,
    BindingResult result, HttpServletRequest request, HttpServletResponse response,
4.           @RequestHeader(value = "Origin", required = false) String origin)
```

- ligne 3 : la valeur postée est du type [PostSupprimerRv] suivant :

```
1.  public class PostSupprimerRv extends PostUser {
2.
3.     // data
4.     @NotNull
5.     private Long idRv;
```

```
6.  ..
7.  }
```

- ligne 1 : la classe [PostSupprimerRv] étend la classe [PostUser] et donc embarque une langue et un utilisateur ;
- ligne 5 : le n° du rendez-vous à supprimer ;

L'implémentation de l'action est la suivante :

```
1.  @RequestMapping(value = "/supprimerRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
2.      @ResponseBody
3.      public Reponse supprimerRv(@Valid @RequestBody PostSupprimerRv postSupprimerRv, BindingResult
    result,   HttpServletRequest request, HttpServletResponse response,
4.              @RequestHeader(value = "Origin", required = false) String origin) {
5.          // contextes de l'action
6.          ActionContext actionContext = getActionContext(postSupprimerRv.getLang(), origin, request,
    response, result,
7.                  rdvMedecinsCorsController);
8.          WebContext thymeleafContext = actionContext.getThymeleafContext();
9.          Locale locale = actionContext.getLocale();
10.         // erreurs ?
11.         List<String> erreurs = actionContext.getErreurs();
12.         if (erreurs != null) {
13.             return getViewErreurs(thymeleafContext, erreurs);
14.         }
15.         // valeurs postées
16.         User user = postSupprimerRv.getUser();
17.         long idRv = postSupprimerRv.getIdRv();
18.         // on supprime le Rdv
19.         AgendaMedecinJour agenda = null;
20.         try {
21.             // on le récupère
22.             Rv rv = application.getRvById(user, idRv);
23.             Creneau creneau = application.getCreneauById(user, rv.getIdCreneau());
24.             long idMedecin = creneau.getIdMedecin();
25.             Date jour = rv.getJour();
26.             // on supprime le rv associé
27.             application.supprimerRv(user, idRv);
28.             // on régénère l'agenda du médecin
29.             agenda = application.getAgendaMedecinJour(user, idMedecin, new SimpleDateFormat("yyyy-MM-
    dd").format(jour));
30.             // on rend le nouvel agenda
31.             Reponse reponse = new Reponse();
32.             reponse.setStatus(1);
33.             reponse.setAgenda(getPartialViewAgenda(actionContext, agenda, locale));
34.             return reponse;
35.         } catch (RdvMedecinsException ex) {
36.             // on retourne la vue [erreurs]
37.             return getViewErreurs(thymeleafContext, ex.getMessages());
38.         } catch (Exception e2) {
39.             // on retourne la vue [erreurs]
40.             return getViewErreurs(thymeleafContext, getErreursForException(e2));
41.         }
42. }
```

- ligne 22 : on récupère le rendez-vous qu'il faut supprimer. S'il n'existe pas, on a une exception ;
- lignes 23-25 : à partir de ce rendez-vous, on trouve le médecin et le jour concerné. Ces informations sont nécessaires pour régénérer l'agenda du médecin ;
- ligne 27 : le rendez-vous est supprimé ;
- ligne 29 : on demande le nouvel agenda du médecin. C'est important. Outre le créneau qui vient d'être libéré, d'autres utilisateurs de l'application ont pu faire des modifications de l'agenda. Il est important de renvoyer à l'utilisateur la version la plus récente de celui-ci ;
- lignes 31-34 : on rend l'agenda :

```
{"status":1,"navbar": null, "jumbotron": null, "agenda":agenda, "content":null}
```

8.6.6.13 L'action [/validerRv]

L'action [/validerRv] ajoute un rendez-vous dans l'agenda d'un médecin. Sa signature est la suivante :

```
1.  @RequestMapping(value = "/validerRv", method = RequestMethod.POST, consumes =
    "application/json; charset=UTF-8")
2.     @ResponseBody
3.     public Reponse validerRv(@RequestBody PostValiderRv postValiderRv, BindingResult result,
    HttpServletRequest request, HttpServletResponse response, @RequestHeader(value = "Origin",
    required = false) String origin)
```

- ligne 3 : la valeur postée est du type [PostValiderRv] suivant :

```
1.  public class PostValiderRv extends PostUser {
2.
3.     // data
4.     @NotNull
5.     private Long idCreneau;
6.     @NotNull
7.     private Long idClient;
8.     @NotNull
9.     @DateTimeFormat(pattern = "yyyy-MM-dd")
10.    private Date jour;
11. ...
12. }
```

- ligne 1 : la classe [PostValiderRv] étend la classe [PostUser] et donc embarque une langue et un utilisateur ;
- ligne 5 : le n° du créneau horaire ;
- ligne 7 : le n° du client pour lequel est faite la réservation ;
- ligne 10 : le jour du rendez-vous ;

L'implémentation de l'action est la suivante :

```
1.  // validation d'un rendez-vous
2.     @RequestMapping(value = "/validerRv", method = RequestMethod.POST, consumes = "application/json;
    charset=UTF-8")
3.     @ResponseBody
4.     public Reponse validerRv(@RequestBody PostValiderRv postValiderRv, BindingResult result,
    HttpServletRequest request, HttpServletResponse response, @RequestHeader(value = "Origin", required =
    false) String origin) {
5.        // contextes de l'action
6.        ActionContext actionContext = getActionContext(postValiderRv.getLang(), origin, request,
    response, result,rdvMedecinsCorsController);
7.        WebApplicationContext springContext = actionContext.getSpringContext();
8.        WebContext thymeleafContext = actionContext.getThymeleafContext();
9.        Locale locale = actionContext.getLocale();
10.       // erreurs ?
11.       List<String> erreurs = actionContext.getErreurs();
12.       if (erreurs != null) {
13.          return getViewErreurs(thymeleafContext, erreurs);
14.       }
15.       // on vérifie la validité du jour du rendez-vous
16.       if (result != null) {
17.          new PostGetAgendaValidator().validate(postValiderRv, result);
18.          if (result.hasErrors()) {
19.             // on retourne la vue [erreurs]
20.             return getViewErreurs(thymeleafContext, getErreursForModel(result, locale,
    springContext));
21.          }
22.       }
23.       // valeurs postées
24.       User user = postValiderRv.getUser();
25.       long idClient = postValiderRv.getIdClient();
26.       long idCreneau = postValiderRv.getIdCreneau();
27.       Date jour = postValiderRv.getJour();
28.       // action
29.       try {
30.          // on récupère des infos sur le créneau
31.          Creneau créneau = application.getCreneauById(user, idCreneau);
32.          long idMedecin = créneau.getIdMedecin();
33.          // on ajoute le Rv
34.          application.ajouterRv(postValiderRv.getUser(), new SimpleDateFormat("yyyy-MM-
    dd").format(jour), idCreneau,idClient);
35.          // on régénère l'agenda
36.          AgendaMedecinJour agenda = application.getAgendaMedecinJour(user, idMedecin,
```

```
37.                 new SimpleDateFormat("yyyy-MM-dd").format(jour));
38.           // on rend le nouvel agenda
39.           Reponse reponse = new Reponse();
40.           reponse.setStatus(1);
41.           reponse.setAgenda(getPartialViewAgenda(actionContext, agenda, locale));
42.           return reponse;
43.       } catch (RdvMedecinsException ex) {
44.           // on retourne la vue [erreurs]
45.           return getViewErreurs(thymeleafContext, ex.getMessages());
46.       } catch (Exception e2) {
47.           // on retourne la vue [erreurs]
48.           return getViewErreurs(thymeleafContext, getErreursForException(e2));
49.       }
50.   }
51. }
```

Le code est analogue à celui de l'action [/supprimerRv].

8.6.7 Étape 4 : tests du serveur Spring/Thymeleaf

Nous allons maintenant tester les différentes actions précédentes avec le plugin Chrome [Advanced Rest Client] (cf paragraphe 9.6, page 606).

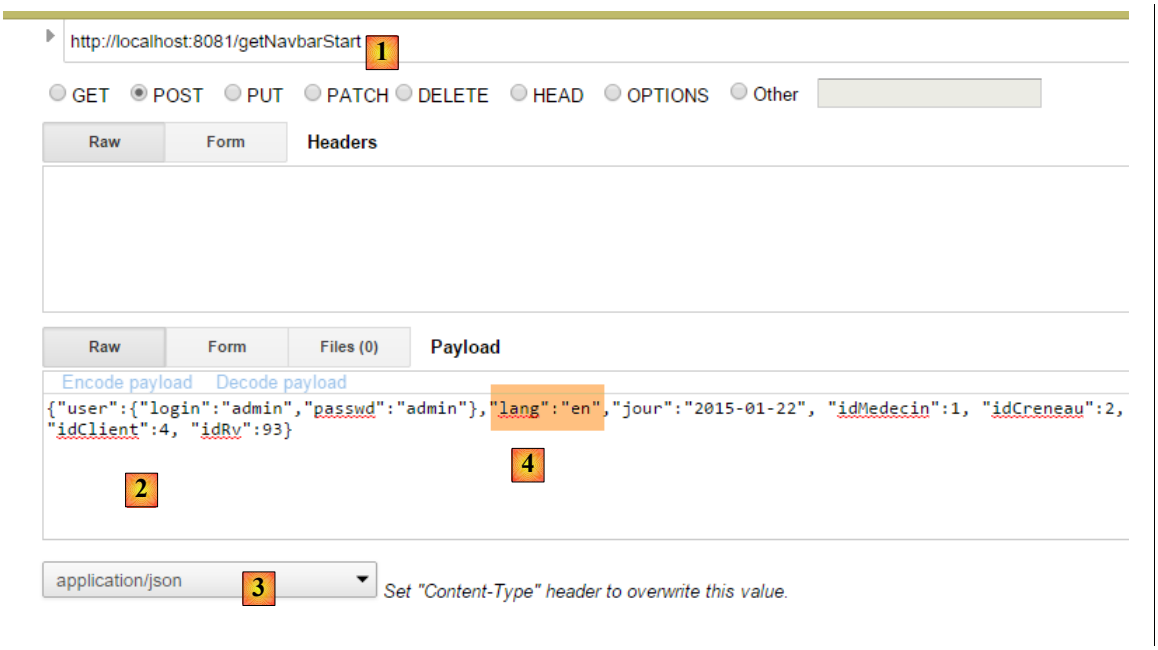
8.6.7.1 Configuration des tests

Toutes les actions attendent une valeur postée. Nous posterons des variantes de la chaîne jSON suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"en","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

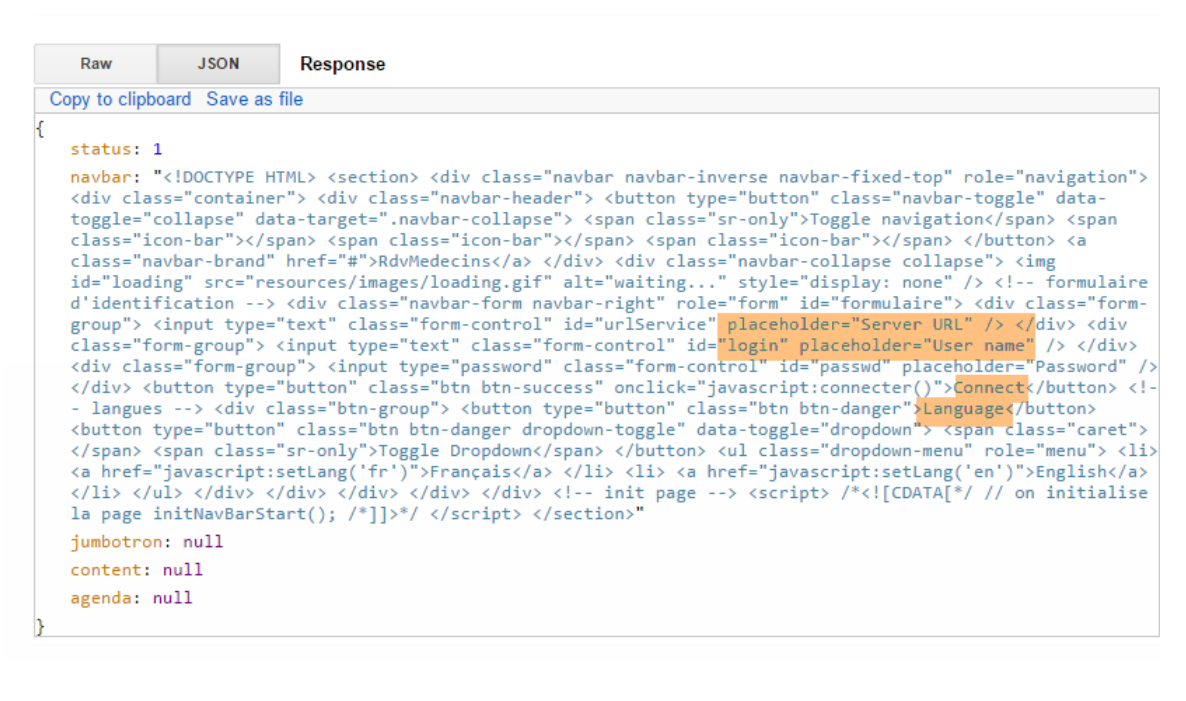
Cette valeur postée comprend des informations superflues pour la plupart des actions. Cependant, celles-ci sont ignorées par les actions qui les reçoivent et ne provoquent pas d'erreur. Cette valeur postée a l'avantage de couvrir les différentes valeurs à poster.

8.6.7.2 L'action [/getNavbarStart]



- en [1], l'action testée ;
- en [2], la valeur postée ;
- en [3], la valeur postée est une chaîne jSON ;
- en [4], la vue [navbar-start] est demandée en anglais ;

Le résultat obtenu est le suivant :



On a reçu la vue [navbar-start] en anglais (zones en surbrillance).

Maintenant, faisons une erreur. Nous mettons l'attribut [lang] de la valeur postée à *null*. Nous recevons le résultat suivant :

Nous avons reçu une réponse d'erreur (status 2) indiquant que le champ [lang] était obligatoire.

8.6.7.3 L'action [/getNavbarRun]

Nous demandons l'action [getNavbarRun] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :

8.6.7.4 L'action [/getJumbotron]

Nous demandons l'action [getJumbotron] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"en","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

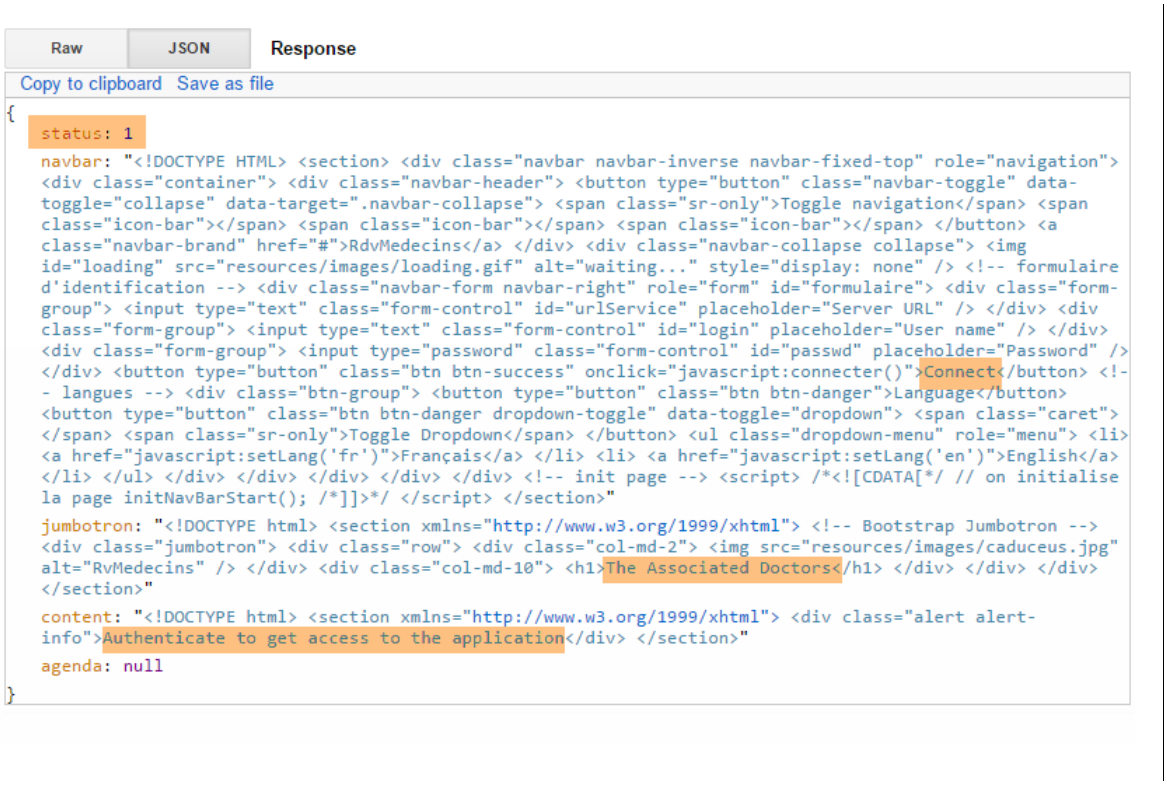
Le résultat obtenu est le suivant :

8.6.7.5 L'action [/getLogin]

Nous demandons l'action [getLogin] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"en","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :

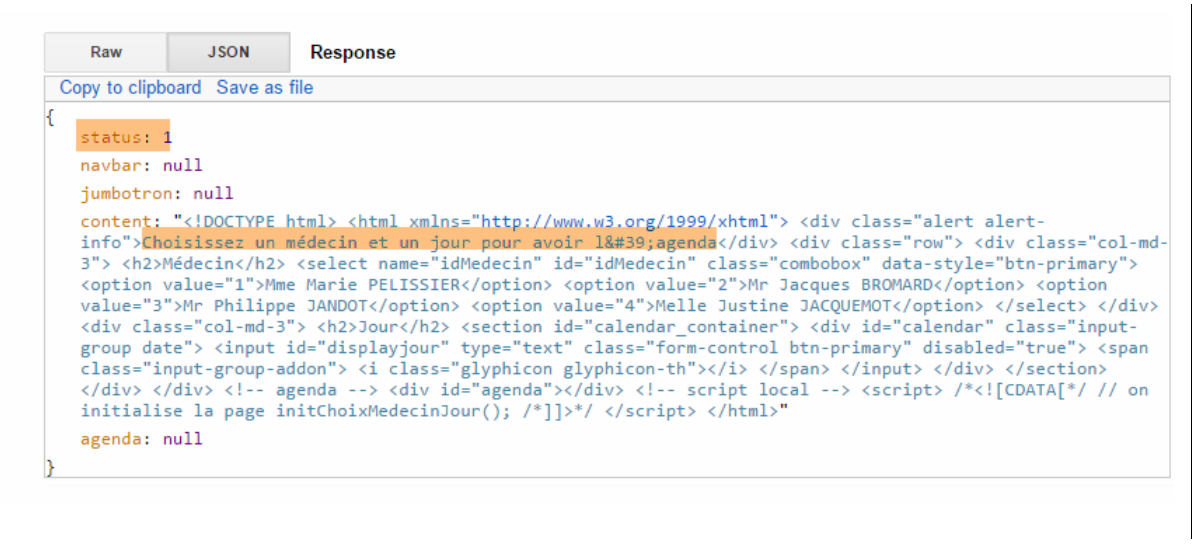


8.6.7.6 *L'action [/getAccueil]*

Nous demandons l'action [getAccueil] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :



Nous recommençons avec un utilisateur inconnu :

```
{"user":{"login":"x","passwd":"x"},"lang":"fr","jour":"2015-01-22", "idMedecin":1, "idCreneau":2, "idClient":4,
"idRv":93}
```

Le résultat obtenu est le suivant :

Nous recommençons avec un utilisateur existant mais pas autorisé à utiliser l'application:

```
{"user":{"login":"user","passwd":"user"},"lang":"en","jour":"2015-01-22", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

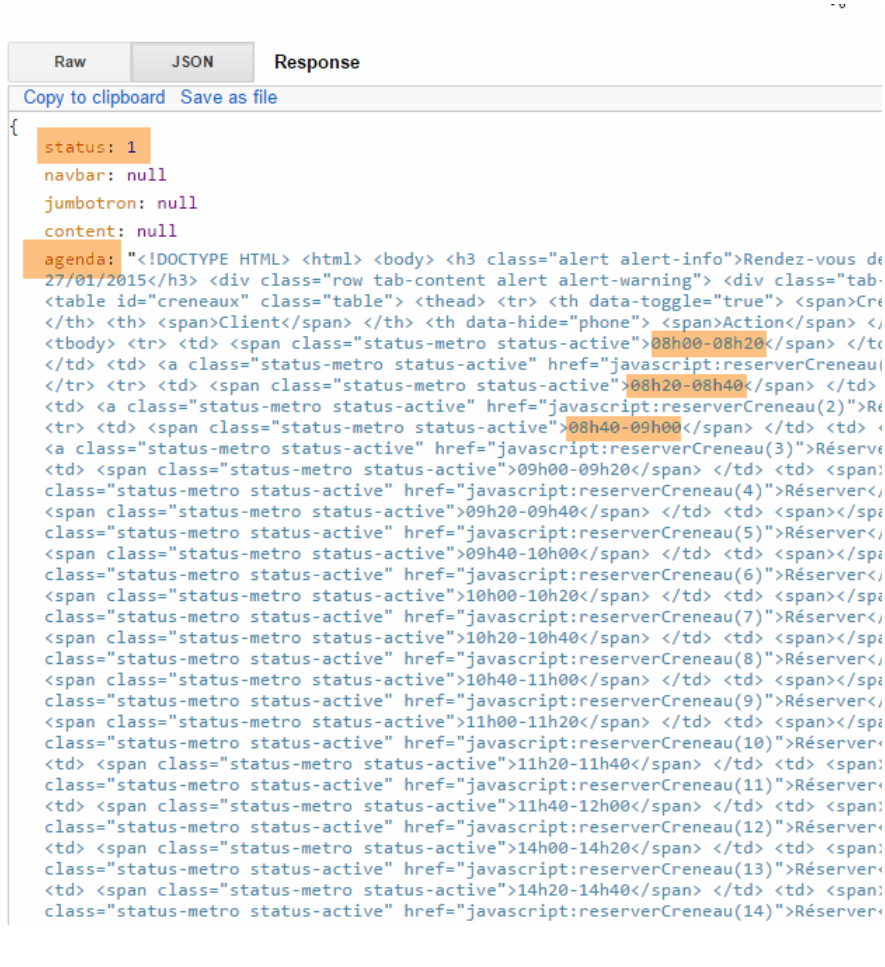
Le résultat obtenu est le suivant :

8.6.7.7 L'action [/getAgenda]

Nous demandons l'action [getAgenda] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-28", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :



Nous recommençons avec un jour antérieur à aujourd'hui :

Nous recommençons avec un médecin inexistant :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-28", "idMedecin":11, "idCreneau":2,
"idClient":4, "idRv":93}
```

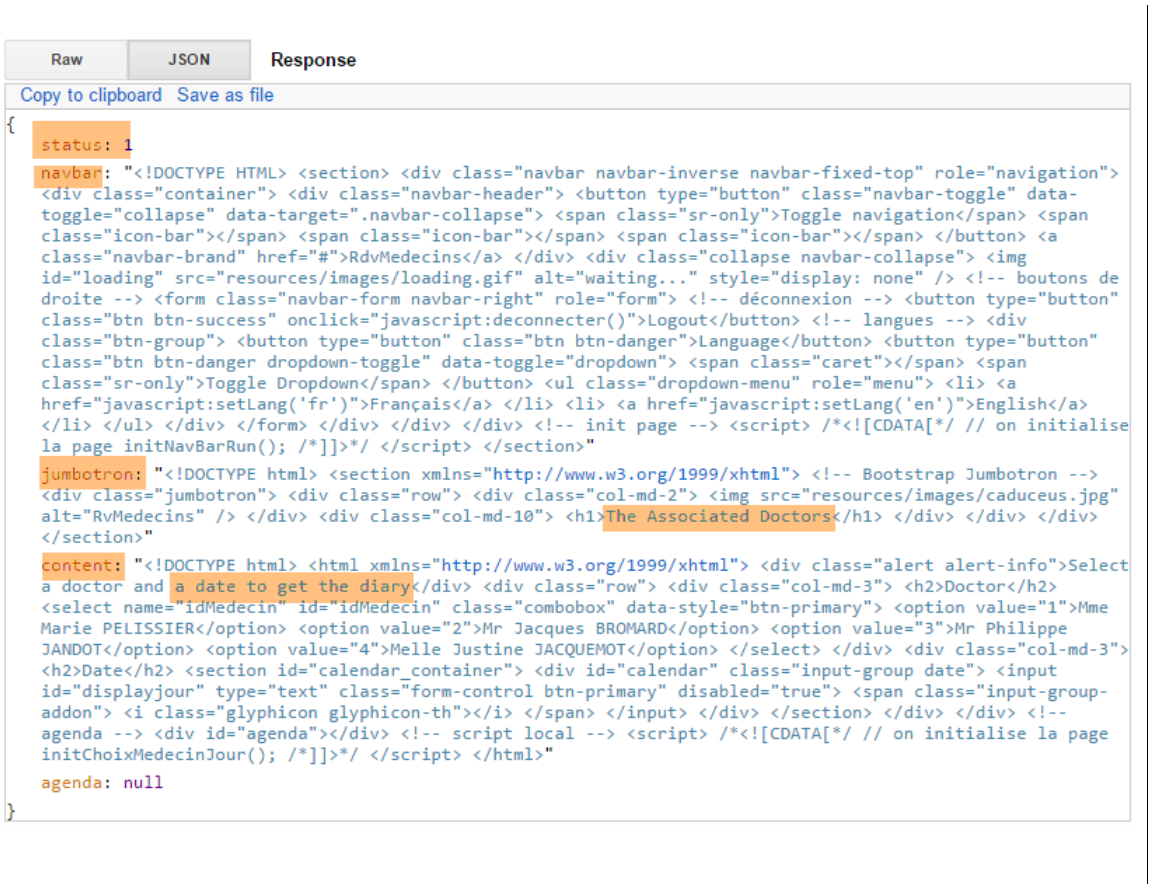
Le résultat obtenu est le suivant :

8.6.7.8 L'action [/getNavbarRunJumbotronAccueil]

Nous demandons l'action [getNavbarRunJumbotronAccueil] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"en","jour":"2015-01-28", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :



Même chose avec un utilisateur inconnu :

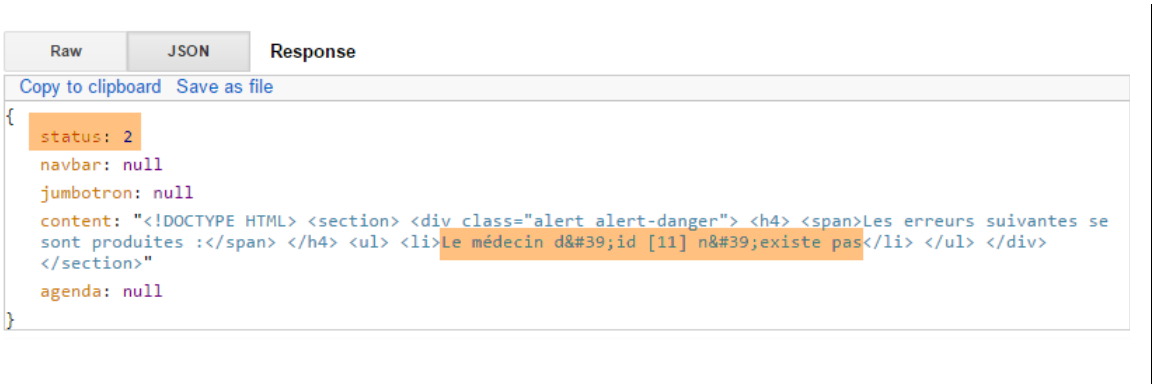
8.6.7.9 L'action [getNavbarRunJumbotronAccueilAgenda]

Nous demandons l'action [getNavbarRunJumbotronAccueilAgenda] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-28", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :

Nous mettons un médecin qui n'existe pas :



8.6.7.10 *L'action [/supprimerRv]*

Nous demandons l'action [supprimerRv] avec la valeur postée suivante :

```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-28", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le Rv de n° 93 n'existe pas. Le résultat obtenu est le suivant :

Avec un rendez-vous qui existe :

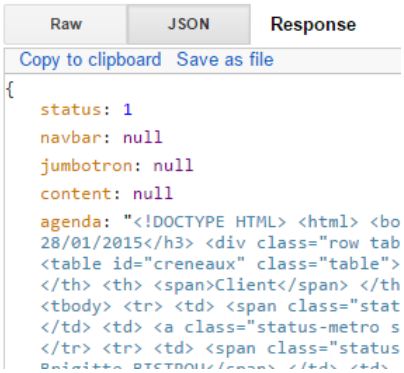
On peut vérifier en base que le rendez-vous a bien été supprimé. Le nouvel agenda est renvoyé.

8.6.7.11 *L'action [/validerRv]*

Nous demandons l'action [validerRv] avec la valeur postée suivante :

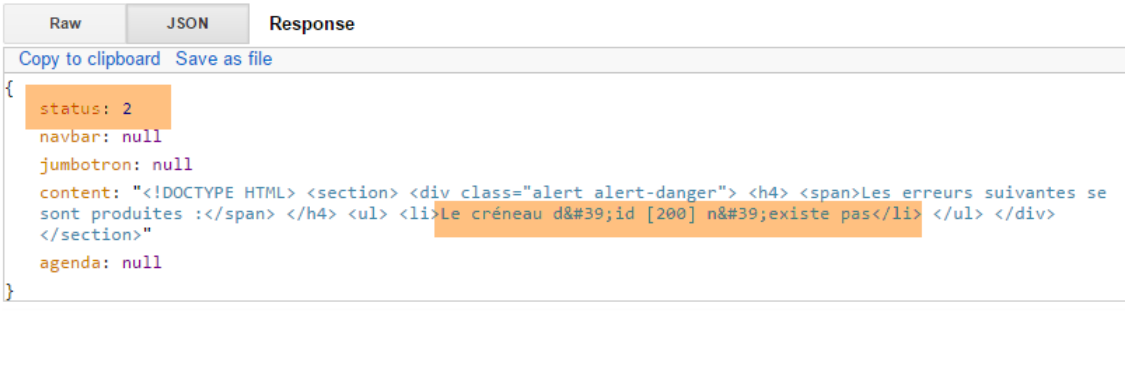
```
{"user":{"login":"admin","passwd":"admin"},"lang":"fr","jour":"2015-01-28", "idMedecin":1, "idCreneau":2,
"idClient":4, "idRv":93}
```

Le résultat obtenu est le suivant :

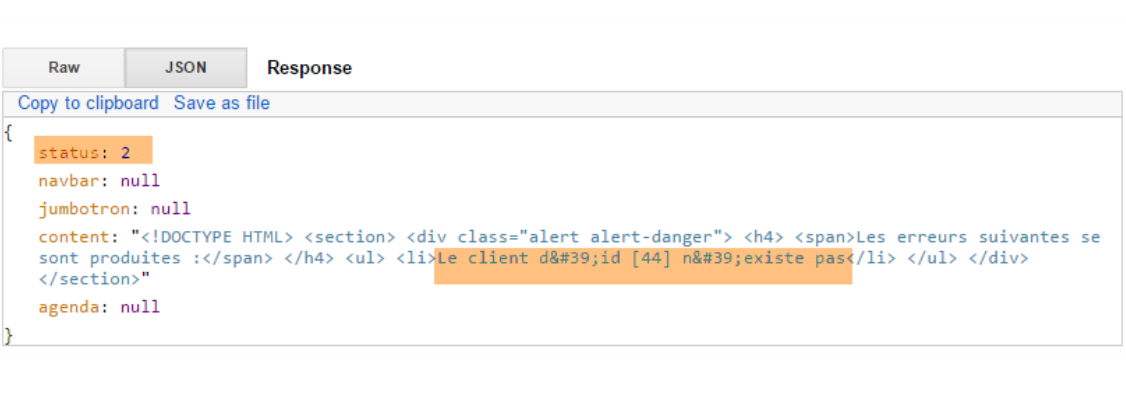


On peut vérifier en base que le rendez-vous a bien été créé. Le nouvel agenda a été renvoyé.

On fait la même chose avec un numéro de créneau inexistant :

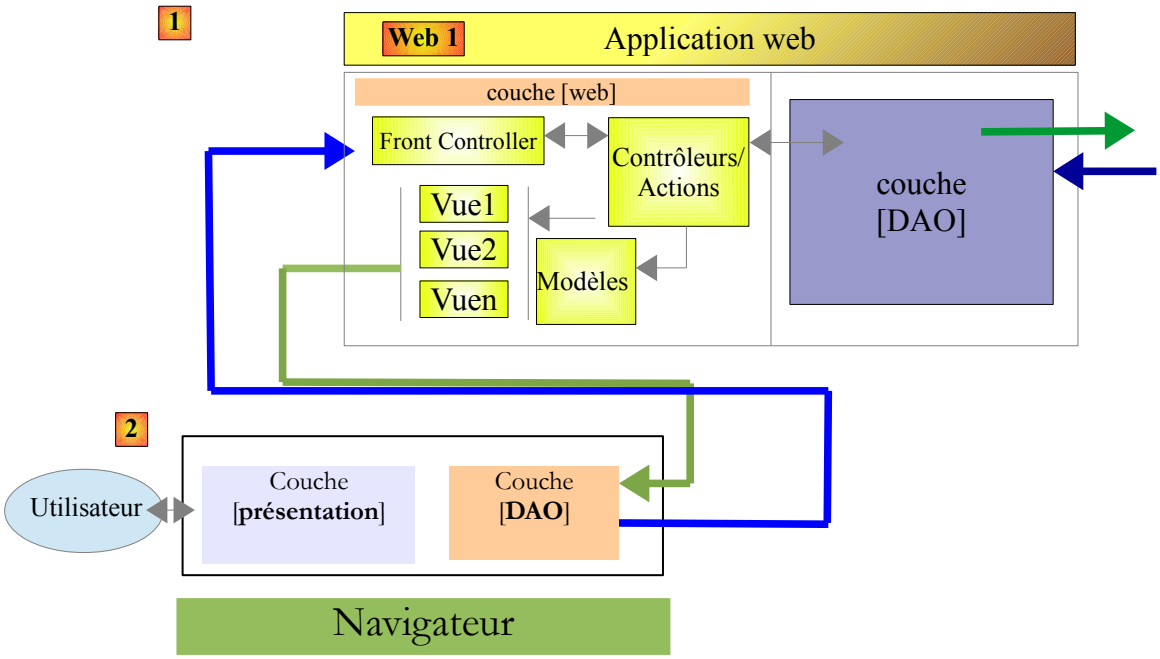


On fait la même chose avec un numéro de client inexistant :



8.6.8 étape 5 : Écriture du client Javascript

Revenons à l'architecture du serveur [Web1] :



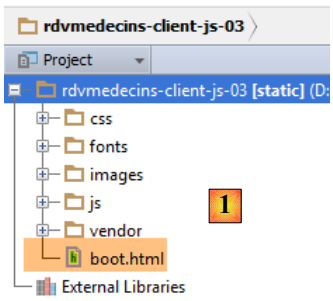
Le client [2] du serveur [Web1] est un client Javascript de type APU (Application à Page Unique) :

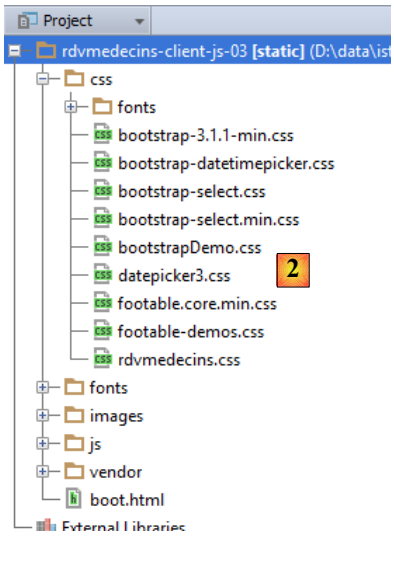
- le client demande la page de boot à un serveur web (pas forcément [Web1]) ;
- il demande les pages suivantes au serveur [Web1] via des appels Ajax ;

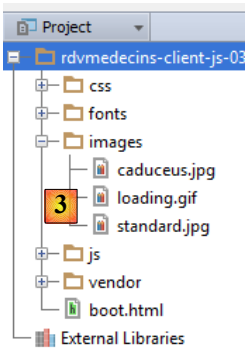
Pour construire ce client, nous allons utiliser l'outil [Webstorm] (cf paragraphe 9.8, page 609). Cet outil m'a semblé plus pratique que STS. Son principal avantage est qu'il offre l'auto-complétion dans la frappe du code ainsi que quelques options de *refactoring*. Cela évite de nombreuses erreurs.

8.6.8.1 Le projet JS

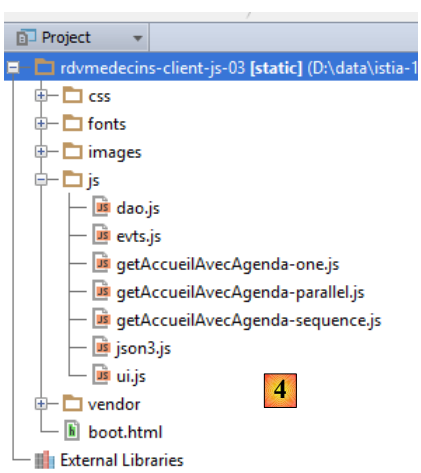
Le projet JS a l'arborescence suivante :

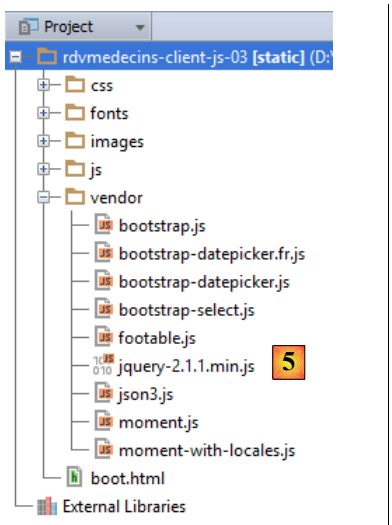






- en [1], le client JS dans son ensemble. [boot.html] est la page de démarrage. Ce sera l'unique page chargée par le navigateur ;
- en [2], les feuilles de style des composants Bootstrap ;
- en [3], les quelques images utilisées par l'application ;

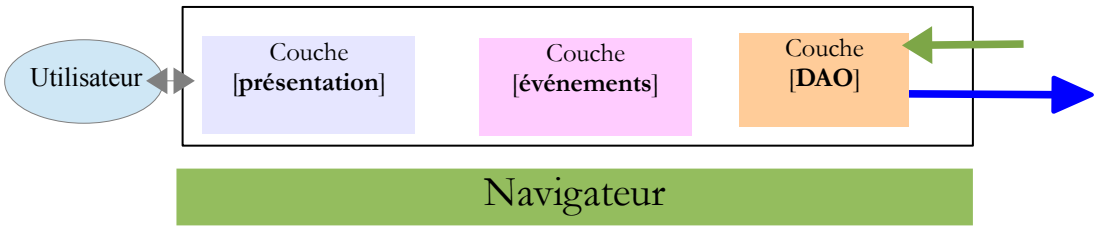




- en [4], les scripts JS. C'est là que se situe notre travail ;
- en [5], les bibliothèques JS utilisées : **jQuery** principalement, et celles des composants Bootstrap ;

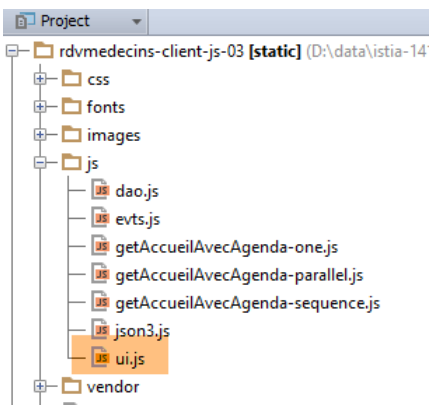
8.6.8.2 L'architecture du code

Le code a été fractionné en trois couches :



- la couche [présentation] rassemble les fonctions d'initialisation de la page [boot.xml] ainsi que celles des divers composants Bootstrap. Elle est implémentée par le fichier [**ui.js**] ;
- la couche [événements] rassemble toutes les gestionnaires des événements de la couche [présentation]. Elle est implémentée par le fichier [**evts.js**] ;
- la couche [DAO] fait les requêtes HTTP vers le serveur [Web1]. Elle est implémentée par le fichier [**dao.js**] ;

8.6.8.3 La couche [présentation]



La couche [présentation] est implémentée par le fichier [**ui.js**] suivant :

```
1.  //la couche [présentation]
2.  var ui = {
3.  // variables globales;
4.    "agenda": "",
5.    "resa": "",
6.    "langue": "",
7.    "urlService": "http://localhost:8081",
8.    "page": "login",
9.    "jourAgenda": "",
10.   "idMedecin": "",
11.   "user": {},
12.   "login": {},
13.   "exceptionTitle": {},
14.   "calendar_infos": {},
15.   "erreur": "",
16.   "idCreneau": "",
17.   "done": "",
18. // composants de la vue
19.   "body": "",
20.   "navbar": "",
21.   "jumbotron": "",
22.   "content": "",
23.   "exception": "",
24.   "exception_text": "",
25.   "exception_title": "",
26.   "loading": ""
27. };
28. // la couche des evts
29. var evts = {};
30. // la couche [dao]
31. var dao = {};
32.
33. // ------------ document ready
34. $(document).ready(function () {
35.   // initialisation document
36.   console.log("document.ready");
37.   // composants de la page
38.   ui.navbar = $("#navbar");
39.   ui.jumbotron = $("#jumbotron");
40.   ui.content = $("#content");
41.   ui.erreur = $("#erreur");
42.   ui.exception = $("#exception");
43.   ui.exception_text = $("#exception-text");
44.   ui.exception_title = $("#exception-title");
45.   // on mémorise la page de login pour pouvoir la restituer
46.   ui.login.lang = ui.langue;
47.   ui.login.navbar = ui.navbar.html();
48.   ui.login.jumbotron = ui.jumbotron.html();
49.   ui.login.content = ui.content.html();
50.   // URL du service
51.   $("#urlService").val(ui.urlService);
52. });
```

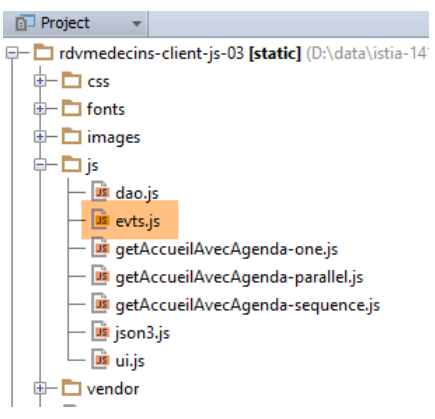
```
53.
54. // ------------------------ fonctions d'initialisation des composants Bootstrap
55. ui.initNavBarStart = function () {
56. ...
57. };
58.
59. ui.initNavBarRun = function () {
60. ...
61. };
62.
63. ui.initChoixMedecinJour = function () {
64. ...
65. };
66.
67. ui.updateCalendar = function (renew) {
68. ...
69. };
70.
71. // affiche le jour sélectionné
72. ui.displayJour = function () {
73. ...
74. };
75.
76. ui.initAgenda = function () {
77. ...
78. };
79.
80. ui.initResa = function () {
81.  ...
82. };
83.
```

- pour isoler les couches entre-elles, il a été décidé de les placer dans trois objets :
  - [ui] pour la couche [présentation] (lignes 2-27),
  - [evts] pour la couche de gestion des événements (ligne 29),
  - [dao] pour la couche [DAO] (ligne 31) ;

Cette séparation des couches dans trois objets permet d'éviter un certain nombre de conflits de noms de variables et fonctions. Chaque couche utilise des variables et fonctions préfixées par l'objet encapsulant la couche.

- lignes 38-44 : on mémorise les zones qui seront toujours présentes quelques soient les vues affichées. Cela évite de faire des recherches jQuery à répétition et inutiles ;
- lignes 46-49 : on mémorise localement la page de boot afin de pouvoir la restituer lorsque l'utilisateur se déconnecte et qu'il n'a pas changé de langue ;
- lignes 54-83 : fonctions d'initialisation des composants Bootstrap. Elles ont toutes été présentées dans l'étude de ceux-ci au paragraphe 8.6.4, page 505 ;

8.6.8.4 Les fonctions utilitaires de la couche [événements]



Les gestionnaires d'événements ont été placés dans le fichier [evts.js]. Plusieurs fonctions sont utilisées régulièrement par les gestionnaires d'événements. Nous les présentons maintenant :

```
1.  // début d'attente
2.  evts.beginWaiting = function () {
3.    // début attente
4.    ui.loading = $("#loading");
5.    ui.loading.show();
6.    ui.exception.hide();
7.    ui.erreur.hide();
8.    evts.travailEnCours = true;
9.  };
10.
11. // fin d'attente
12. evts.stopWaiting = function () {
13.   // fin attente
14.   evts.travailEnCours = false;
15.   ui.loading = $("#loading");
16.   ui.loading.hide();
17. };
18.
19. // affichage résultat
20. evts.showResult = function (result) {
21.   // on affiche les données reçues
22.   var data = result.data;
23.   // on analyse le status
24.   switch (result.status) {
25.     case 1:
26.       // erreur ?
27.       if (data.status == 2) {
28.         ui.erreur.html(data.content);
29.         ui.erreur.show();
30.       } else {
31.         if (data.navbar) {
32.           ui.navbar.html(data.navbar);
33.         }
34.         if (data.jumbotron) {
35.           ui.jumbotron.html(data.jumbotron);
36.         }
37.         if (data.content) {
38.           ui.content.html(data.content)
39.         }
40.         if (data.agenda) {
41.           ui.agenda = $("#agenda");
42.           ui.resa = $("#resa");
43.         }
44.       }
45.       break;
46.     case 2:
47.       // affichage erreur
48.       evts.showException(data);
49.       break;
50.   }
51. };
52.
53. // ------------ fonctions diverses
54. evts.showException = function (data) {
55.   // affichage erreur
56.   ui.exception.show();
57.   ui.exception_text.html(data);
58.   ui.exception_title.text(ui.exceptionTitle[ui.langue]);
59. };
```

- ligne 2 : la fonction [evts.beginwaiting] est appelée avant toute action [DAO] asynchrone ;
- lignes 4-5 : on affiche l'image animée de l'attente ;
- lignes 6-7 : on cache la zone d'affichage des erreurs et des exceptions (ce ne sont pas les mêmes) ;
- ligne 8 : on note qu'un travail asynchrone est en cours ;
- ligne 12 : la fonction [evts.stopwaiting] est appelée après qu'une action [DAO] asynchrone ait rendu son résultat ;
- ligne 14 : on note que le travail asynchrone est terminé ;
- lignes 15 : on cache l'image animée de l'attente ;
- ligne 20 : la fonction [evts.showResult] affiche le résultat [result] d'une action [DAO] asynchrone. Le résultat est un objet JS de la forme suivante **{'status':status,'data':data,'sendMeBack':sendMeBack}**.

- lignes 47-50 : utilisées si [result.status==2]. Cela arrive lorsque le serveur [Web1] envoie une réponse avec un entête HTTP d'erreur (par exemple 403 forbidden). Dans ce cas [data] est la chaîne jSON envoyée par le serveur pour signaler l'erreur ;
- ligne 25 : cas où on a reçu une réponse valide du serveur [Web1]. Le champ [data] contient alors la réponse du serveur : **{'status':status,'navbar':navbar,'jumbotron':jumbotron,'agenda':agenda,'content':content}** ;
- ligne 27 : cas où le serveur [Web1] a envoyé une réponse d'erreur **{'status':2,'navbar':null,'jumbotron':null,'agenda':null,'content':erreurs} ;**
- lignes 28-29 : la vue [erreurs] est affichée ;
- lignes 31-33 : affichage éventuel de la barre de navigation ;
- lignes 34-36 : affichage éventuel du jumbotron ;
- lignes 37-39 : affichage éventuel du champ [data.content]. Représente selon les cas l'une des vues [accueil, agenda] ;
- lignes 40-43 : si l'agenda a été régénéré on récupère certaines références sur ses composants afin de ne pas les rechercher à chaque fois qu'on en aura besoin ;
- ligne 54 : la fonction [evts.showException] a pour fonction d'afficher le texte de l'exception contenue dans son paramètre [data] ;
- lignes 57-58 : le texte de l'exception est affiché ;
- ligne 58 : le titre de l'exception dépend de la langue du moment ;

La fichier [evts.js] contient plus de 300 lignes de code que je ne vais pas commenter toutes. Je vais simplement prendre quelques exemples pour montrer l'esprit de cette couche.

8.6.8.5 Connexion d'un utilisateur

La connexion d'un utilisateur est assurée par la fonction suivante :

```
1.  // ----------------------- connexion
2.  evts.connecter = function () {
3.    // on récupère les valeurs à poster
4.    var login = $("#login").val().trim();
5.    var passwd = $("#passwd").val().trim();
6.    // on fixe l'URL du serveur
7.    ui.urlService = $("#urlService").val().trim();
8.    dao.setUrlService(ui.urlService);
9.    // paramètres de la requête
10.   var post = {
11.     "user": {
12.       "login": login,
13.       "passwd": passwd
14.     },
15.     "lang": ui.langue
16.   };
17.   var sendMeBack = {
18.     "user": {
19.       "login": login,
20.       "passwd": passwd
21.     },
22.     "caller": evts.connecterDone
23.   };
24.   // on fait la requête
25.   evts.execute([{
26.     "name": "accueil-sans-agenda",
27.     "post": post,
28.     "sendMeBack": sendMeBack
29.   }]);
30. };
```

- lignes 4-5 : on récupère le login et le mot de passe de l'utilisateur ;
- lignes 7-8 : on récupère l'URL du service [Web1]. Elle est mémorisée à la fois dans la couche [ui] et la couche [dao] ;
- lignes 10-16 : la valeur à poster : la langue du moment et l'utilisateur qui cherche à se connecter ;
- lignes 17-23 : l'objet [sendMeBack] est un objet qui est passé à la fonction [DAO] qui va être appelée et que celle-ci doit renvoyer à la fonction de la ligne 22. Ici l'objet [sendMeBack] encapsule l'utilisateur qui cherche à se connecter ;
- lignes 25-29 : la fonction [evts.execute] est capable d'exécuter une suite d'actions asynchrones. Ici, on passe une liste constituée d'une seule action. Les champs de celle-ci sont les suivants :
  - [name] : le nom de l'action asynchrone à exécuter,
  - [post] : la valeur à poster au serveur [Web1],
  - [sendMeBack] : la valeur que l'action asynchrone doit renvoyer avec son résultat ;

Avant de détailler la fonction [evts.execute], regardons la fonction [evts.connecterDone] de la ligne 22. C'est la fonction à laquelle la fonction [DAO] asynchrone appelée doit rendre son résultat :

```
1.  evts.connecterDone = function (result) {
2.    // affichage résultat
3.    evts.showResult(result);
4.    // connexion réussie ?
5.    if (result.status == 1 && result.data.status == 1) {
6.      // page
7.      ui.page = "accueil-sans-agenda";
8.      // on note l'utilisateur
9.      ui.user = result.sendMeBack.user;
10.   }
11. };
```

- ligne 3 : le résultat renvoyé par le serveur [Web1] est affiché ;
- ligne 5 : si ce résultat ne contient pas d'erreurs, alors on mémorise la nature de la nouvelle page (ligne 7) ainsi que l'utilisateur authentifié (ligne 9) ;

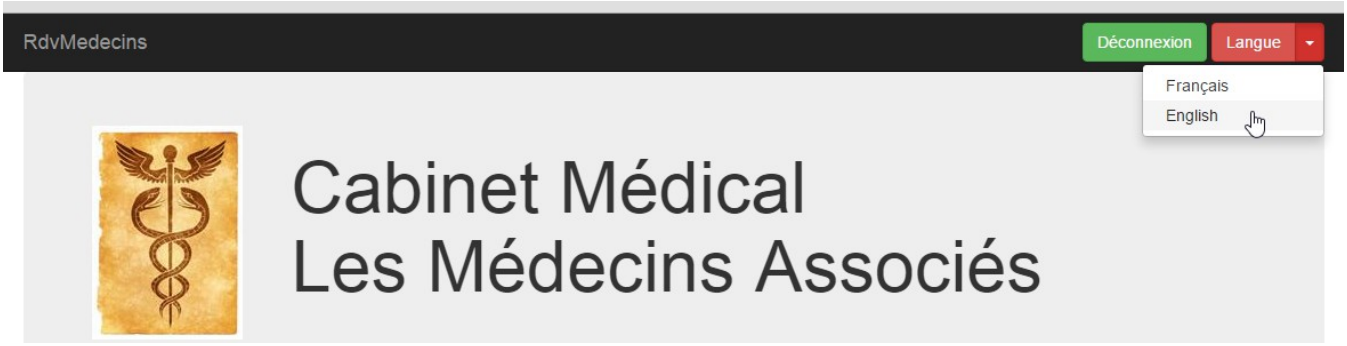
La fonction [evts.execute] exécute une suite d'actions asynchrones :

```
1.  // exécution d'une suite d'actions
2.  evts.execute = function (actions) {
3.    // travail en cours ?
4.    if (evts.travailEnCours) {
5.      // on ne fait rien
6.      return;
7.    }
8.    // attente
9.    evts.beginWaiting();
10.   // exécution des actions
11.   dao.doActions(actions, evts.stopWaiting);
12. };
```

- ligne 2 : le paramètre [actions] est une liste d'actions asynchrones à exécuter ;
- lignes 4-7 : l'exécution n'est acceptée que s'il n'y en a pas une autre déjà en cours ;
- ligne 9 : on met en route l'attente ;
- ligne 11 : on demande à la couche [DAO] d'exécuter la suite d'actions. Le second paramètre est le nom de la fonction à exécuter lorsque toutes les actions de la suite auront rendu leur résultat ;

Nous n'allons pas détailler maintenant la fonction [dao.doActions]. Nous allons examiner un autre événement.

8.6.8.6 Changement de langue

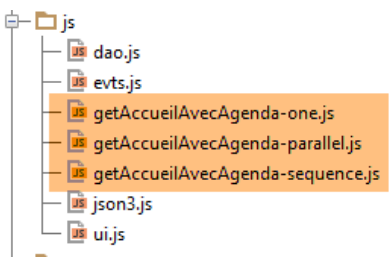


Le changement de langue est assuré par la fonction suivante :

```
1.  // ----------------------- changement de langue
2.  evts.setLang = function (lang) {
3.    // chgt de langue ?
4.    if (lang == ui.langue) {
5.      // on ne fait rien
6.      return;
7.    }
8.    // nouvelle langue
9.    ui.langue = lang;
10.   // quelle page faut-il traduire ?
11.   switch (ui.page) {
12.     case "login":
13.       evts.getLogin();
14.       break;
15.     case "accueil-sans-agenda":
16.       evts.getAccueilSansAgenda();
17.       break;
18.     case "accueil-avec-agenda":
19.       evts.getAccueilAvecAgenda(ui);
20.       break;
21.   }
22. };
```

- ligne 2 : le paramètre [lang] est la nouvelle langue : 'fr' ou 'en' ;
- lignes 4-7 : si la nouvelle langue est celle du moment, on ne fait rien ;
- ligne 9 : on mémorise la nouvelle langue ;
- lignes 12-20 : dans le cas d'un changement de langue, il faut régénérer la page actuellement affichée par le navigateur. Il y a trois pages possibles :
  - celle appelée [login] où la page affichée est celle de l'authentification,
  - celle appelée [accueil-sans-agenda] qui est la page affichée juste après une authentification réussie,
  - celle appelée [accueil-avec-agenda] qui est la page affichée dès qu'un premier agenda a été affiché. Ensuite, elle reste en permanence jusqu'à la déconnexion de l'utilisateur ;

Nous allons traiter le cas de la page [accueil-avec-agenda]. Il existe trois versions de cette fonction :



- la version [getAccueilAvecAgenda-one] fait exécuter une unique action asynchrone ;
- la version [getAccueilAvecAgenda-parallel] fait exécuter quatre actions asynchrones en parallèle ;
- la version [getAccueilAvecAgenda-sequence] fait exécuter quatre actions asynchrones l'une après l'autre ;

8.6.8.7 La fonction [getAccueilAvecAgenda-one]

C'est la fonction suivante :

```
1. // ------------------------- getAccueilAvecAgenda
2. evts.getAccueilAvecAgenda=function(ui) {
3.   // paramètres requête
4.   var post = {
5.     "user": ui.user,
6.     "lang": ui.langue,
7.     "idMedecin": ui.idMedecin,
8.     "jour": ui.jourAgenda
9.   };
10.  var sendMeBack = {
11.    "caller": evts.getAccueilAvecAgendaDone
12.  };
13.  // requête
14.  evts.execute([{
15.    "name": "accueil-avec-agenda",
16.    "post": post,
17.    "sendMeBack": sendMeBack
18.  }]);
19. };
```

- lignes 4-9 : la valeur à poster encapsule l'utilisateur connecté, la langue désirée, le n° du médecin dont on veut l'agenda, la journée de l'agenda désiré ;
- lignes 10-12 : l'objet [sendMeBack] est l'objet qui sera renvoyé à la fonction de la ligne 11. Ici, il n'embarque aucune information ;
- lignes 14-18 : exécution d'une suite d'une action asynchrone, celle nommée [accueil-avec-agenda] (ligne 15) ;
- ligne 11 : la fonction exécutée lorsque l'action asynchrone [accueil-avec-agenda] aura rendu son résultat ;

La fonction [**evts**.getAccueilAvecAgendaDone] de la ligne 11 affiche le résultat de la fonction asynchrone nommée [accueil-avec-agenda] :

```
1. evts.getAccueilAvecAgendaDone = function (result) {
2.   // affichage résultat
3.   evts.showResult(result);
4.   // nouvelle page ?
5.   if (result.status == 1 && result.data.status == 1) {
6.     ui.page = "accueil-avec-agenda";
7.   }
8. };
```

- ligne 1 : [result] est le résultat de la fonction asynchrone nommée [**accueil-avec-agenda**] ;
- ligne 3 : ce résultat est affiché ;
- ligne 5 : si c'est un résultat sans erreur, on note la nouvelle page (ligne 6) ;

8.6.8.8 La fonction [getAccueilAvecAgenda-parallel]

C'est la fonction suivante :

```
1. // ------------------------- getAccueilAvecAgenda
2. evts.getAccueilAvecAgenda=function(ui) {
3.   // actions [navbar-run, jumbotron, accueil, agenda] en //
4.   // navbar-run
5.   var navbarRun = {
6.     "name": "navbar-run"
7.   };
8.   navbarRun.post = {
9.     "lang": ui.langue
10.  };
11.  navbarRun.sendMeBack = {
12.    "caller": evts.showResult
```

```
13.   };
14.   // jumbotron
15.   var jumbotron = {
16.     "name": "jumbotron"
17.   };
18.   jumbotron.post = {
19.     "lang": ui.langue
20.   };
21.   jumbotron.sendMeBack = {
22.     "caller": evts.showResult
23.   };
24.   // accueil
25.   var accueil = {
26.     "name": "accueil"
27.   };
28.   accueil.post = {
29.     "lang": ui.langue,
30.     "user": ui.user
31.   };
32.   accueil.sendMeBack = {
33.     "caller": evts.showResult
34.   };
35.   // agenda
36.   var agenda = {
37.     "name": "agenda"
38.   };
39.   agenda.post = {
40.     "user": ui.user,
41.     "lang": ui.langue,
42.     "idMedecin": ui.idMedecin,
43.     "jour": ui.jourAgenda
44.   };
45.   agenda.sendMeBack = {
46.     'idMedecin': ui.idMedecin,
47.     'jour': ui.jourAgenda,
48.     "caller": evts.getAgendaDone
49.   };
50.   // exécution actions en //
51.   evts.execute([navbarRun, jumbotron, accueil, agenda])
52. };
```

- ligne 51 : on exécute cette fois quatre actions asynchrones. Elles vont être exécutées en parallèle ;
- lignes 5-13 : définition de l'action [navbarRun] qui récupère la barre de navigation [navbar-run] ;
- ligne 12 : la fonction à exécuter lorsque l'action asynchrone [navbarRun] aura rendu son résultat ;
- lignes 15-23 : définition de l'action [jumbotron] qui récupère la vue [jumbotron] ;
- ligne 22 : la fonction à exécuter lorsque l'action asynchrone [jumbotron] aura rendu son résultat ;
- lignes 25-34 : définition de l'action [accueil] qui récupère la vue [accueil] ;
- ligne 33 : la fonction à exécuter lorsque l'action asynchrone [accueil] aura rendu son résultat ;
- lignes 36-49 : définition de l'action [agenda] qui récupère la vue [jumbotron] ;
- ligne 48 : la fonction à exécuter lorsque l'action asynchrone [agenda] aura rendu son résultat ;

8.6.8.9 La fonction [getAccueilAvecAgenda-sequence]

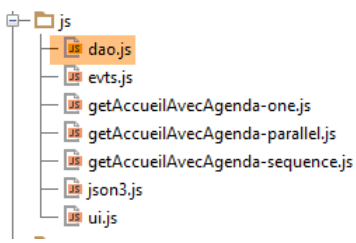
C'est la fonction suivante :

```
1.  // ------------------------- getAccueilAvecAgenda
2.  evts.getAccueilAvecAgenda=function(ui) {
3.    // actions [navbar-run, jumbotron, accueil, agenda] dans l'ordre
4.    // agenda
5.    var agenda = {
6.      "name" : "agenda"
7.    };
8.    agenda.post = {
```

```
9.      "user" : ui.user,
10.     "lang" : ui.langue,
11.     "idMedecin" : ui.idMedecin,
12.     "jour" : ui.jourAgenda
13.   };
14.   agenda.sendMeBack = {
15.     'idMedecin' : ui.idMedecin,
16.     'jour' : ui.jourAgenda,
17.     "caller" : evts.getAgendaDone
18.   };
19.   // accueil
20.   var accueil = {
21.     "name" : "accueil"
22.   };
23.   accueil.post = {
24.     "lang" : ui.langue,
25.     "user" : ui.user
26.   };
27.   accueil.sendMeBack = {
28.     "caller" : evts.showResult,
29.     "next" : agenda
30.   };
31.   // jumbotron
32.   var jumbotron = {
33.     "name" : "jumbotron"
34.   };
35.   jumbotron.post = {
36.     "lang" : ui.langue
37.   };
38.   jumbotron.sendMeBack = {
39.     "caller" : evts.showResult,
40.     "next" : accueil
41.   };
42.   // navbar-run
43.   var navbarRun = {
44.     "name" : "navbar-run"
45.   };
46.   navbarRun.post = {
47.     "lang" : ui.langue
48.   };
49.   navbarRun.sendMeBack = {
50.     "caller" : evts.showResult,
51.     "next" : jumbotron
52.   };
53.   // exécution actions en séquence
54.   evts.execute([ navbarRun ])
55. };
```

- ligne 54 : on exécute l'action [navbarRun]. Lorsqu'elle est terminée, on passe à la suivante : [jumbotron], ligne 51. Cette action est alors exécutée à son tour. Lorsqu'elle est terminée, on passe à la suivante : [accueil], ligne 40. Celle-ci est exécutée à son tour. Lorsqu'elle est terminée, on passe à la suivante : [agenda], ligne 29. Celle-ci est exécutée à son tour. Lorsqu'elle est terminée, on s'arrête car l'action [agenda] n'a pas d'action suivante.

8.6.8.10 La couche [DAO]



Le fichier [dao.js] rassemble toutes les fonctions de la couche [DAO]. Nous allons présenter celles-ci progressivement :

```
1.  // URL exposées par le serveur
2.  dao.urls = {
3.    "login": "/getLogin",
4.    "accueil": "/getAccueil",
5.    "jumbotron": "/getJumbotron",
6.    "agenda": "/getAgenda",
7.    "supprimerRv": "/supprimerRv",
8.    "validerRv": "/validerRv",
9.    "navbar-start": "/getNavbarStart",
10.   "navbar-run": "/getNavbarRun",
11.   "accueil-sans-agenda": "/getNavbarRunJumbotronAccueil",
12.   "accueil-avec-agenda": "/getNavbarRunJumbotronAccueilAgenda"
13. };
14. // --------------- interface
15. // url serveur
16. dao.setUrlService = function (urlService) {
17.   dao.urlService = urlService;
18. };
```

- lignes 16-18 : la fonction qui permet de fixer l'URL du service [Web1] ;
- lignes 2-13 : le dictionnaire reliant le nom d'une action asynchrone à l'URL du serveur [Web1] à interroger ;

```
1.  // ----------------- gestion générique des actions
2.  // exécution d'une suite d'actions asynchrones
3.  dao.doActions = function (actions, done) {
4.    // traitement des actions
5.    dao.actionsCount = actions.length;
6.    dao.actionIndex = 0;
7.    for (var i = 0; i < dao.actionsCount; i++) {
8.      // requête DAO asynchrone
9.      var deferred = $.Deferred();
10.     deferred.done(dao.actionDone);
11.     dao.doAction(deferred, actions[i], done);
12.   }
13. };
```

- ligne 3 : la fonction [***dao***.doActions] exécute une suite d'actions asynchrones [actions]. Le paramètre [done] est la fonction à exécuter lorsque toutes les actions ont rendu leur résultat ;
- lignes 7-12 : les actions asynchrones sont exécutées en parallèle. Cependant, dans le cas où l'une d'elles a une suivante, celle-ci est alors exécutée à la fin de l'action qui la précède ;
- ligne 9 : on objet [Deferred] dans l'état [pending] ;
- ligne 10 : lorsque cet objet passera dans l'état [resolved], la fonction [***dao***.actionDone] sera exécutée ;
- ligne 11 : l'action n° i de la liste est exécutée de façon asynchrone. Le paramètre [done] de la ligne 3 est passé en paramètre ;

La fonction [***dao***.actionDone] qui est exécutée à la fin de chaque action asynchrone est la suivante :

```
1.  // on a reçu un résultat
2.  dao.actionDone = function (result) {
3.    // caller ?
4.    var sendMeBack = result.sendMeBack;
```

```
5.    if (sendMeBack && sendMeBack.caller) {
6.      sendMeBack.caller(result);
7.    }
8.    // next ?
9.    if (sendMeBack && sendMeBack.next) {
10.     // requête DAO asynchrone
11.     var deferred = $.Deferred();
12.     deferred.done(dao.actionDone);
13.     dao.doAction(deferred, sendMeBack.next, sendMeBack.done);
14.   }
15.   // fini ?
16.   dao.actionIndex++;
17.   if (dao.actionIndex == dao.actionsCount) {
18.     // done ?
19.     if (sendMeBack && sendMeBack.done) {
20.       sendMeBack.done(result);
21.     }
22.   }
23. };
```

- ligne 2 : la fonction [***dao***.actionDone] reçoit le résultat [result] d'une des actions asynchrones de la liste des actions à exécuter ;
- lignes 4-7 : si l'action asynchrone terminée avait précisé une fonction à laquelle renvoyer le résultat, cette fonction est appelée ;
- lignes 9-14 : si l'action asynchrone terminée a une suivante, alors cette action est à son tour exécutée ;
- lignes 16 : une action est terminée. On augmente le compteur des actions terminées. Une action qui a un nombre indéterminé d'actions suivantes compte pour une action ;
- lignes 19-21 : si initialement, une fonction [done] avait été précisée pour être exécutée lorsque toutes les actions de la suite ont rendu leur résultat, alors cette fonction est maintenant exécutée ;

La méthode [***dao***.doAction] exécute une action asychrone :

```
1.  // exécution d'une action
2.  dao.doAction = function (deferred, action, done) {
3.    // fonction done à embarquer dans l'action
4.    if (action.sendMeBack) {
5.      action.sendMeBack.done = done;
6.    } else {
7.      action.sendMeBack = {
8.        "done": done
9.      };
10.   }
11.   // exécution action
12.   dao.executePost(deferred, action.sendMeBack, dao.urls[action.name], action.post)
13. };
```

- lignes 4-10 : on vient de le voir, la fonction qui va traiter le résultat de l'action asynchrone qui va être exécutée doit avoir accès à la fonction [done]. Pour cela, on met cette dernière dans l'objet [sendMeBack], objet qui fera partie du résultat de l'opération asynchrone ;
- ligne 12 : on exécute la fonction [***dao***.executePost] qui fait un appel HTTP au serveur [Web1]. L'URL cible est l'URL associée au nom de l'action à exécuter ;

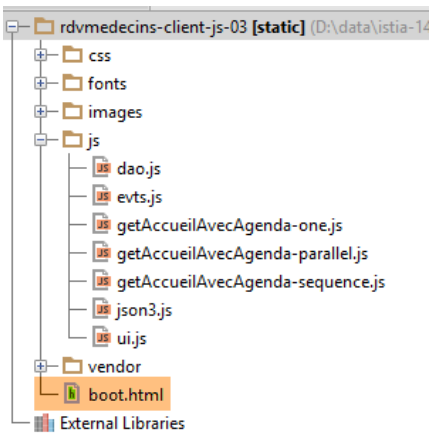
La fonction [***dao***.executePost] exécute un appel HTTP :

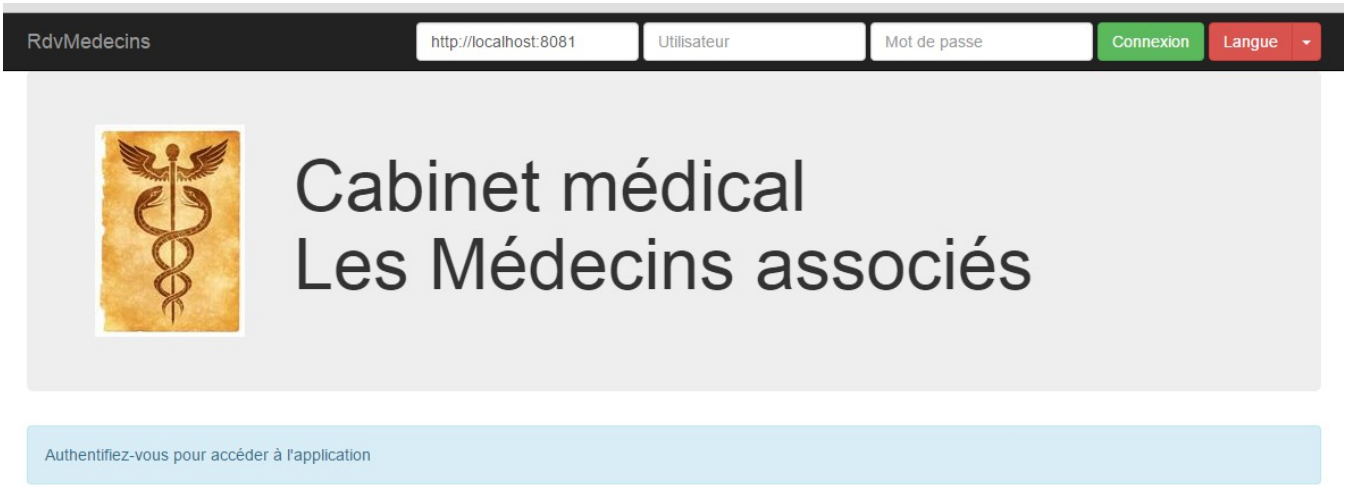
```
1.  // requête HTTP
2.  dao.executePost = function (deferred, sendMeBack, url, post) {
3.    // on fait un appel Ajax à la main
4.    $.ajax({
5.      headers: {
6.        'Accept': 'application/json',
7.        'Content-Type': 'application/json'
8.      },
9.      url: dao.urlService + url,
```

```
10.     type: 'POST',
11.     data: JSON3.stringify(post),
12.     dataType: 'json',
13.     success: function (data) {
14.       // on rend le résultat
15.       deferred.resolve({
16.         "status": 1,
17.         "data": data,
18.         "sendMeBack": sendMeBack
19.       });
20.     },
21.     error: function (jqXHR, textStatus, errorThrown) {
22.       var data;
23.       if (jqXHR.responseText) {
24.         data = jqXHR.responseText;
25.       } else {
26.         data = textStatus;
27.       }
28.       // on rend l'erreur
29.       deferred.resolve({
30.         "status": 2,
31.         "data": data,
32.         "sendMeBack": sendMeBack
33.       });
34.     }
35.   });
36. };
```

Nous avons déjà rencontré et commenté cette fonction. On notera simplement ligne 9 que l'URL cible est la concaténation de l'URL du serveur [Web1] avec l'URL associée au nom de l'action.

8.6.8.11 La page de boot





La page de boot [boot.html] affiche la vue ci-dessus. C'est l'unique page chargée directement par le navigateur. Les autres sont obtenues avec des appels Ajax. Son code est le suivant :

```
1.  <!DOCTYPE HTML>
2.  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
3.        xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
4.  <head>
5.    <meta name="viewport" content="width=device-width"/>
6.    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
7.    <title>RdvMedecins</title>
8.    <!-- Bootstrap core CSS -->
9.    <link rel="stylesheet" href="css/bootstrap-3.1.1-min.css"/>
10.   <link rel="stylesheet" type="text/css" href="css/bootstrap-select.min.css"/>
11.   <link rel="stylesheet" type="text/css" href="css/datepicker3.css"/>
12.   <link rel="stylesheet" type="text/css" href="css/footable.core.min.css"/>
13.   <!-- Custom styles for this template -->
14.   <link rel="stylesheet" type="text/css" href="css/rdvmedecins.css"/>
15.   <!-- Bootstrap core JavaScript ================================================= -->
16.   <script type="text/javascript" src="vendor/jquery-2.1.1.min.js"></script>
17.   <script type="text/javascript" src="vendor/bootstrap.js"></script>
18.   <script type="text/javascript" src="vendor/bootstrap-select.js"></script>
19.   <script type="text/javascript" src="vendor/moment-with-locales.js"></script>
20.   <script type="text/javascript" src="vendor/bootstrap-datepicker.js"></script>
21.   <script type="text/javascript" src="vendor/bootstrap-datepicker.fr.js"></script>
22.   <script type="text/javascript" src="vendor/footable.js"></script>
23.   <!-- scripts utilisateurs -->
24.   <script type="text/javascript" src="js/json3.js"></script>
25.   <script type="text/javascript" src="js/ui.js"></script>
26.   <script type="text/javascript" src="js/evts.js"></script>
27.   <script type="text/javascript" src="js/getAccueilAvecAgenda-sequence.js"></script>
28.   <script type="text/javascript" src="js/dao.js"></script>
29. </head>
30. <body id="body">
31. <div id="navbar">
32.   <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
33.     <div class="container">
34.       <div class="navbar-header">
35.         <button type="button" class="navbar-toggle" data-toggle="collapse" data-
    target=".navbar-collapse">
36.           <span class="sr-only">Toggle navigation</span> <span class="icon-bar"></span>
    <span class="icon-bar"></span>
37.           <span class="icon-bar"></span>
38.         </button>
39.         <a class="navbar-brand" href="#">RdvMedecins</a>
40.       </div>
41.       <div class="navbar-collapse collapse">
```

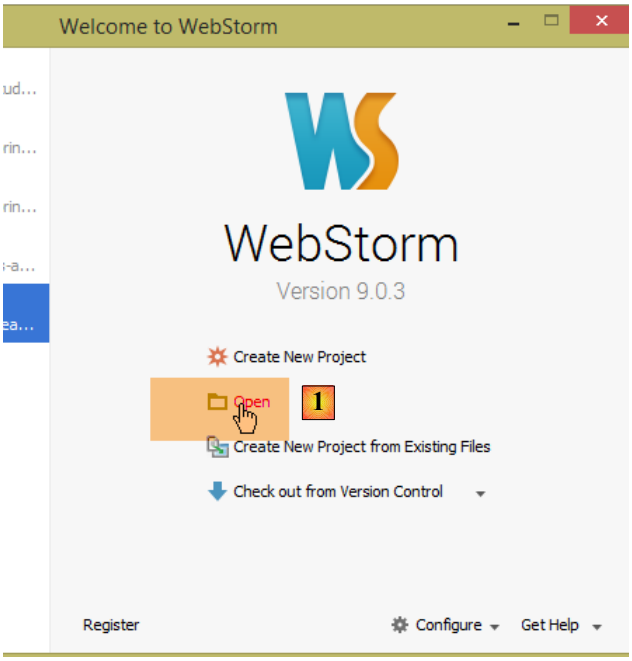
```
42.         <img id="loading" src="images/loading.gif" alt="waiting..." style="display: none"/>
43.         <!-- formulaire d'identification -->
44.         <div class="navbar-form navbar-right" role="form" id="formulaire">
45.           <div class="form-group">
46.             <input type="text" placeholder="URL du serveur" class="form-control"
    id="urlService"/>
47.           </div>
48.           <div class="form-group">
49.             <input type="text" placeholder="Utilisateur" class="form-control" id="login"/>
50.           </div>
51.           <div class="form-group">
52.             <input type="password" placeholder="Mot de passe" class="form-control"
    id="passwd"/>
53.           </div>
54.           <button type="button" class="btn btn-success"
    onclick="javascript:evts.connecter()">Connexion</button>
55.           <!-- langues -->
56.           <div class="btn-group">
57.             <button type="button" class="btn btn-danger">Langue</button>
58.             <button type="button" class="btn btn-danger dropdown-toggle" data-
    toggle="dropdown">
59.               <span class="caret"></span> <span class="sr-only">Toggle Dropdown</span>
60.             </button>
61.             <ul class="dropdown-menu" role="menu">
62.               <li><a href="javascript:evts.setLang('fr')">Français</a></li>
63.               <li><a href="javascript:evts.setLang('en')">English</a></li>
64.             </ul>
65.           </div>
66.         </div>
67.       </div>
68.     </div>
69.   </div>
70. </div>
71. <div class="container">
72.   <!-- Bootstrap Jumbotron -->
73.   <div id="jumbotron">
74.     <div class="jumbotron">
75.       <div class="row">
76.         <div class="col-md-2">
77.           <img src="images/caduceus.jpg" alt="RvMedecins"/>
78.         </div>
79.         <div class="col-md-10">
80.           <h1>
81.             Cabinet médical<br/>Les Médecins associés
82.           </h1>
83.         </div>
84.       </div>
85.     </div>
86.   </div>
87.   <!-- panneaux d'erreur -->
88.   <div id="erreur"></div>
89.   <div id="exception" class="alert alert-danger" style="display: none">
90.     <h3 id="exception-title"></h3>
91.     <span id="exception-text"></span>
92.   </div>
93.   <!-- contenu -->
94.   <div id="content">
95.     <div class="alert alert-info">Authentifiez-vous pour accéder à l'application</div>
96.   </div>
97. </div>
98. <!-- init page -->
99. <script>
100.     // on initialise la page
```

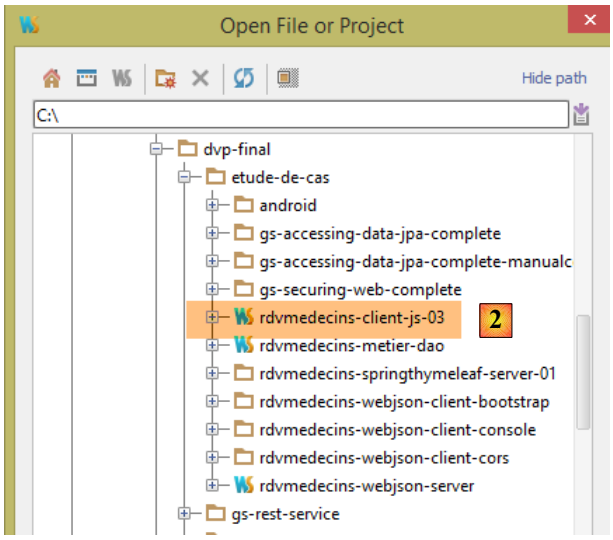
```
101.     ui.langue = 'fr';
102.     ui.exceptionTitle['fr'] = "L'erreur suivante s'est produite côté serveur :";
103.     ui.exceptionTitle['en'] = "The following server error was met:";
104.     ui.initNavBarStart();
105.   </script>
106.   </body>
107.   </html>
```

- nous avons déjà rencontré ce type de page dans le chapitre sur Bootstrap (paragraphe 8.6.4, page 505) ;
- lignes 99-105 : initialisation de certains éléments de la couche [présentation] ;
- ligne 27, le script [`getAccueilAvecAgenda-sequence.js`] est utilisé. En changeant le script de cette ligne on a trois comportements différents pour obtenir la page [accueil-avec-agenda] :
  - [`getAccueilAvecAgenda-one.js`] obtient la page avec un seul appel HTTP,
  - [`getAccueilAvecAgenda-parallel.js`] obtient la page avec quatre appels HTTP simultanés,
  - [`getAccueilAvecAgenda-sequence.js`] obtient la page avec quatre appels HTTP successifs ;

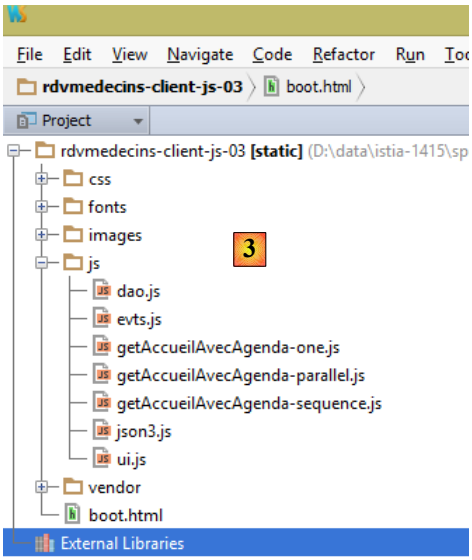
8.6.8.12 Tests

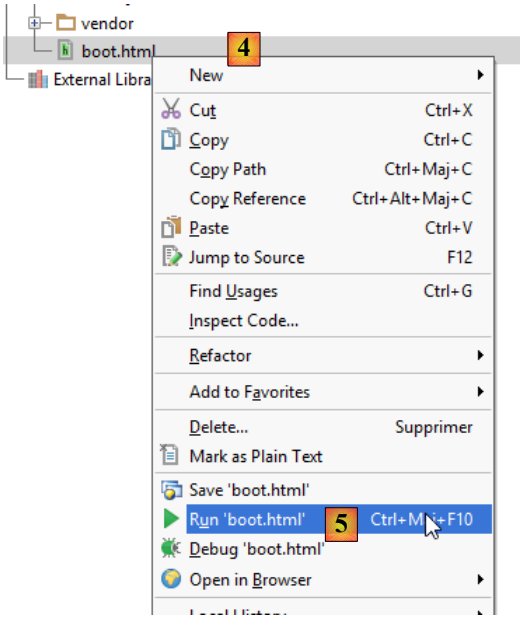
Il y a différentes façons de faire les tests. Nous allons utiliser ici l'outil [Webstorm] :





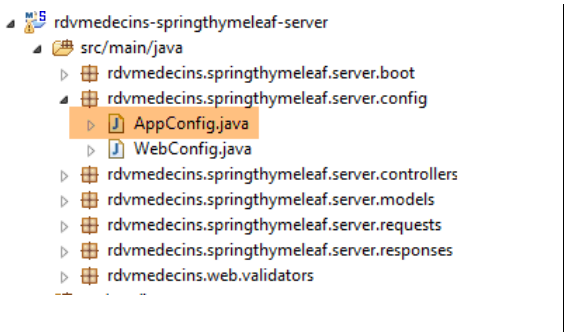
- en [1] on ouvre un projet. On désigne simplement le dossier [2] contenant l'arborescence statique (HTML, CSS, JS) du site à tester ;





- en [3], le site statique ;
- en [4-5], on charge la page [boot.html] ;

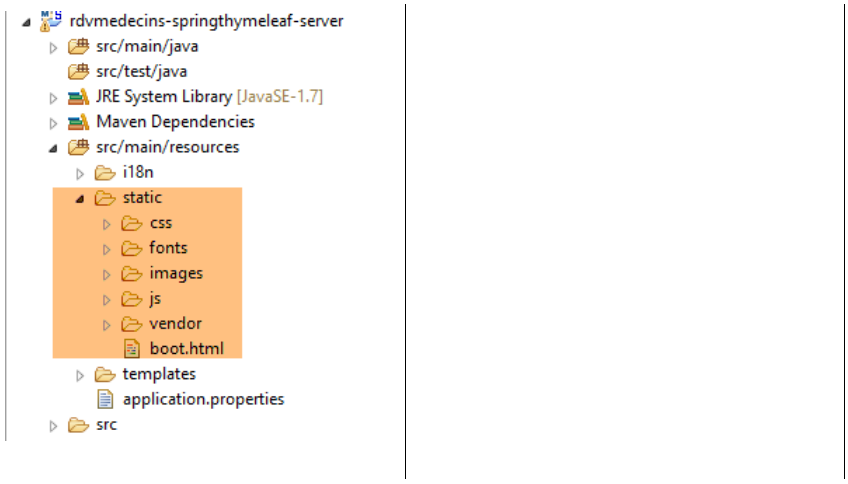
- en [5], on voit qu'un serveur embarqué par [Webstorm] a délivré la page [boot.html] à partir du port [63342]. C'est un point important à comprendre car cela veut dire que les scripts de la page [boot.html] vont faire des appels inter-domaines au serveur [Web1] qui lui travaille sur [localhost:8081]. Le navigateur qui a chargé [boot.html] sait qu'il l'a chargée à partir de [localhost:63342]. Il ne va donc pas accepter que cette page fasse des appels au site [localhost:8081] parce que ce n'est pas le même port. Il va donc mettre en oeuvre les appels inter-domaines décrits au paragraphe 8.4.13, page 452. Pour cette raison, il faut que l'application [Web1] soit configuré pour accepter ces appels inter-domaines. C'est dans le fichier [AppConfig] du serveur Spring / Thymeleaf que ça se décide :




```
1.  @EnableAutoConfiguration
2.  @ComponentScan(basePackages = { "rdvmedecins.springthymeleaf.server" })
3.  @Import({ WebConfig.class, DaoConfig.class })
4.  public class AppConfig {
5.
6.      // admin / admin
7.      private final String USER_INIT = "admin";
8.      private final String MDP_USER_INIT = "admin";
9.      // racine service web / json
10.     private final String WEBJSON_ROOT = "http://localhost:8080";
11.     // timeout en millisecondes
12.     private final int TIMEOUT = 5000;
13.     // CORS
14.     private final boolean CORS_ALLOWED=true;
15. ...
```

Nous laissons le lecteur faire les tests du client JS. Il doit être capable de reproduire les fonctionnalités décrites au paragraphe 8.6.3, page 498.

Une fois que le client JS a été déclaré correct, on peut le déployer dans le dossier du serveur [Web1] pour éviter d'avoir à autoriser les requêtes inter-domaines :



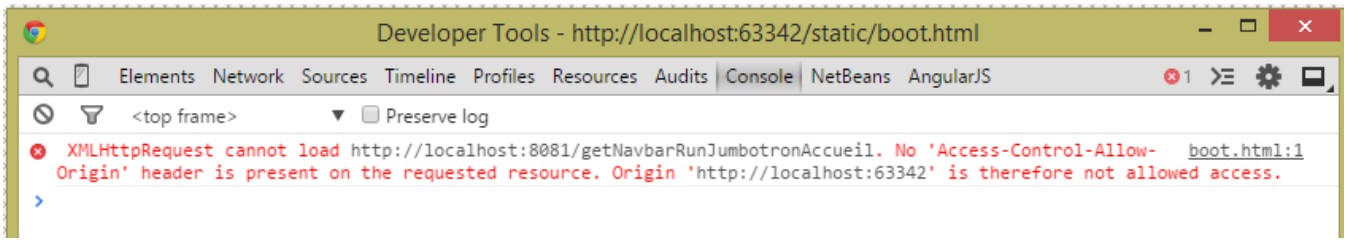
Ci-dessus, nous avons copié le site testé dans le dossier [src / main / resources / static]. Ensuite on peut demander l'URL [http://localhost:8081/boot.html] :

Maintenant nous n'avons plus besoin des requêtes inter-domaines et nous pouvons écrire dans le fichier de configuration [AppConfig] du serveur [Web1] :

```
  // CORS
  private final boolean CORS_ALLOWED=false;
```

L'application ci-dessus va continuer à fonctionner. Si on revient vers l'application [Webstorm], elle ne marche plus :

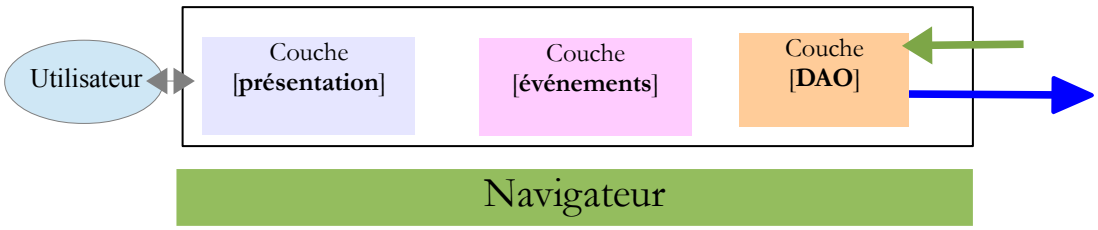
Si on va dans la console de développement (Ctrl-Maj-I) on a la cause de l'erreur :



C'est une erreur de requête inter-domaines non autorisée.

8.6.8.13 Conclusion

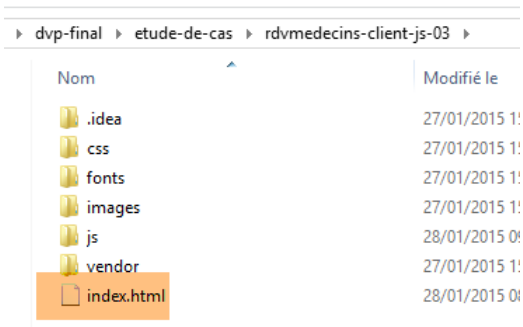
Nous avons réalisé l'architecture JS suivante :



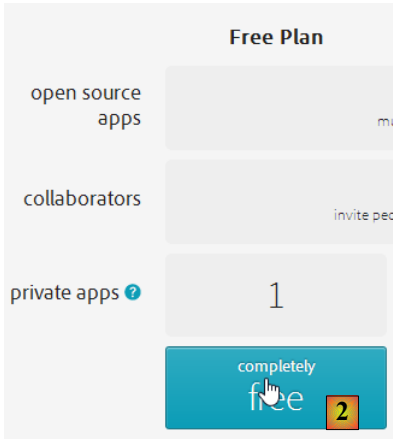
- les couches sont assez clairement séparées ;
- on a une application de type APU (Application à Page Unique). C'est cette caractéristique qui va maintenant nous permettre de générer une application native pour divers mobiles (Android, IoS, Windows Phone) ;
- on a créé un modèle capable d'exécuter des actions asynchrones en parallèle, en séquence ou un mix des deux ;

8.6.9 étape 6 : génération d'une application native pour Android

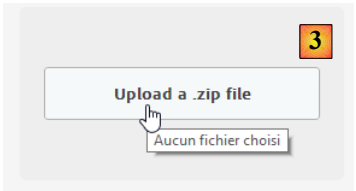
L'outil [Phonegap] [http://phonegap.com/] permet de produire un exécutable pour mobile (Android, IoS, Windows 8, ...) à partir d'une application HTML / JS / CSS. Il y a différentes façons d'arriver à ce but. Nous utilisons le plus simple : un outil présent en ligne sur le site de Phonegap [http://build.phonegap.com/apps]. Cet outil va 'uploader' le fichier zip du site statique à convertir. La page de boot doit s'appeler [index.html]. Nous renommons donc la page [boot.html] en [index.html] :

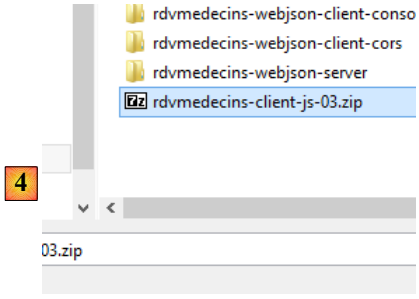


puis nous zippons le dossier, ici [rdvmedecins-client-js-03]. Ensuite nous allons sur le site de Phonegap [http://build.phonegap.com/apps] :

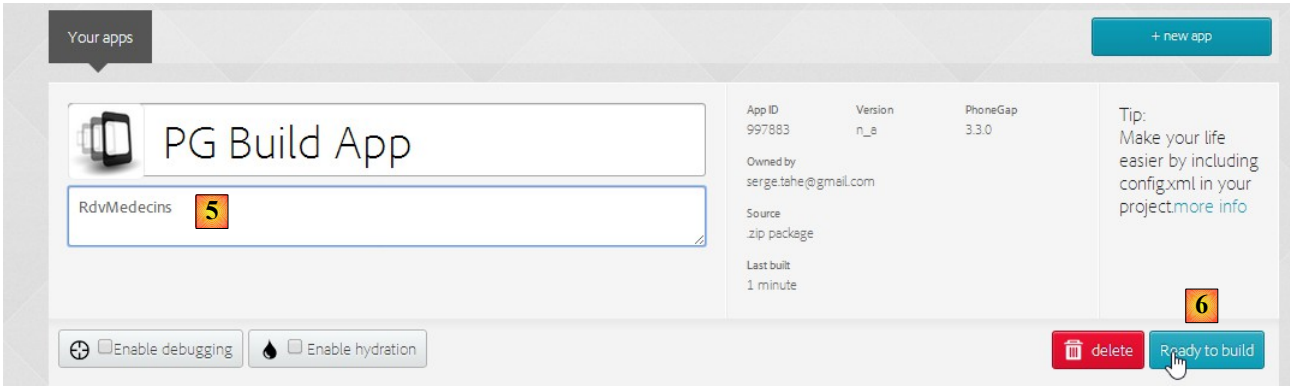


- avant [1], vous aurez peut-être à créer un compte ;
- en [1], on démarre ;
- en [2], on choisit un plan gratuit n'autorisant qu'une application Phonegap ;



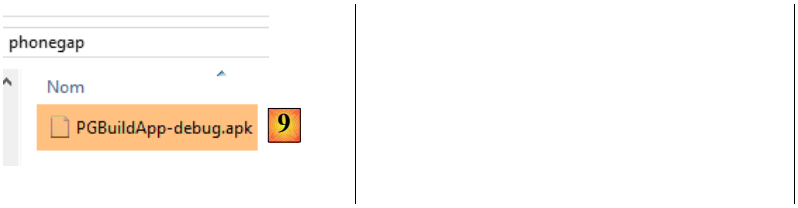


- en [3], on télécharge l'application zippée [4] ;



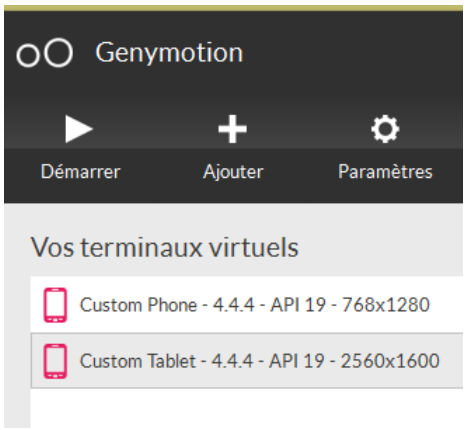
- en [5], on donne un nom à l'application ;
- en [6], on la construit. Cette opération peut prendre 1 minute. Patientez jusqu'à ce que les icônes des différentes plate-formes mobiles indiquent que la construction est terminée ;

- seuls les binaires Android [7] et Windows [8] ont été générés ;
- on clique sur [7] pour télécharger le binaire d'Android ;



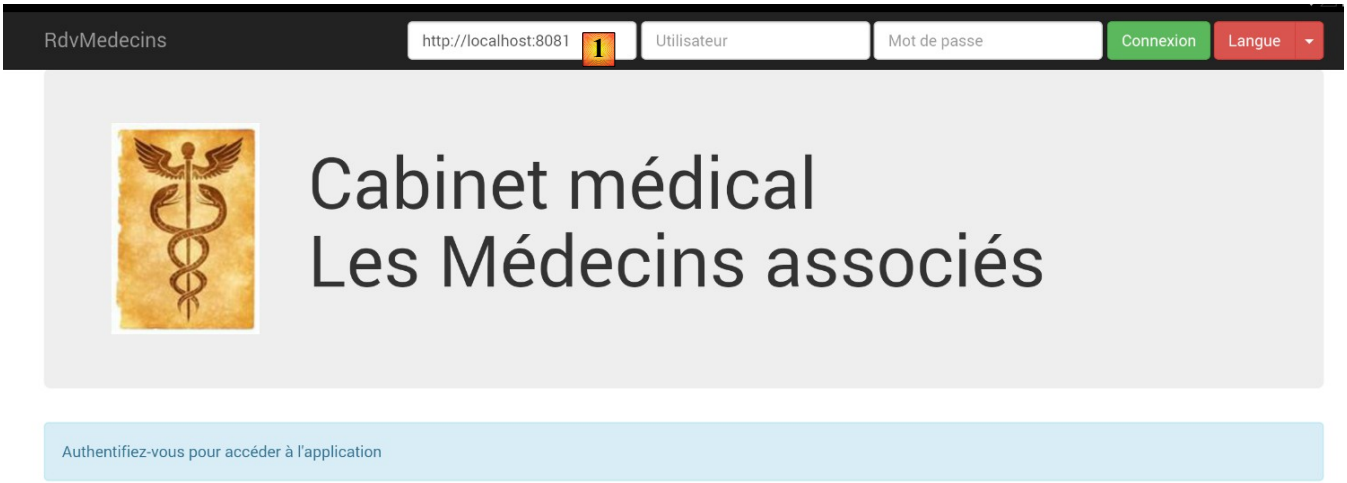
- en [9] le binaire [apk] téléchargé ;

Lancez un émulateur [GenyMotion] pour une tablette Android (voir paragraphe 9.9, page 612) :



Ci-dessus, on lance un émulateur de tablette avec l'API 19 d'Android. Une fois l'émulateur lancé,

- déverrouillez-le en tirant le verrou (s'il est présent) sur le côté puis en le lâchant ;
- avec la souris, tirez le fichier [PGBuildApp-debug.apk] que vous avez téléchargé et déposez-le sur l'émulateur. Il va être alors installé et exécuté ;



Il faut changer l'URL en [1]. Pour cela, dans une fenêtre de commande, tapez la commande [ipconfig] (ligne 1 ci-dessous) qui va afficher les différentes adresses IP de votre machine :

```
1.  C:\Users\Serge Tahé>ipconfig
2.
3.  Configuration IP de Windows
4.
5.
6.  Carte réseau sans fil Connexion au réseau local* 15 :
7.
8.     Statut du média. . . . . . . . . . . . : Média déconnecté
9.     Suffixe DNS propre à la connexion. . . :
10.
11. Carte Ethernet Connexion au réseau local :
12.
13.    Suffixe DNS propre à la connexion. . . : ad.univ-angers.fr
14.    Adresse IPv6 de liaison locale. . . . .: fe80::698b:455a:925:6b13%4
15.    Adresse IPv4. . . . . . . . . . . . . .: 172.19.81.34
16.    Masque de sous-réseau. . . . . . . . . : 255.255.0.0
17.    Passerelle par défaut. . . . . . . . . : 172.19.0.254
18.
19. Carte réseau sans fil Wi-Fi :
20.
21.    Statut du média. . . . . . . . . . . . : Média déconnecté
22.    Suffixe DNS propre à la connexion. . . :
```

```
23.
24. ...
```

Notez soit l'adresse IP Wifi (lignes 6-9), soit l'adresse IP sur le réseau local (lignes 11-17). Puis utilisez cette adresse IP dans l'URL du serveur web :

Ceci fait, connectez-vous au service web :

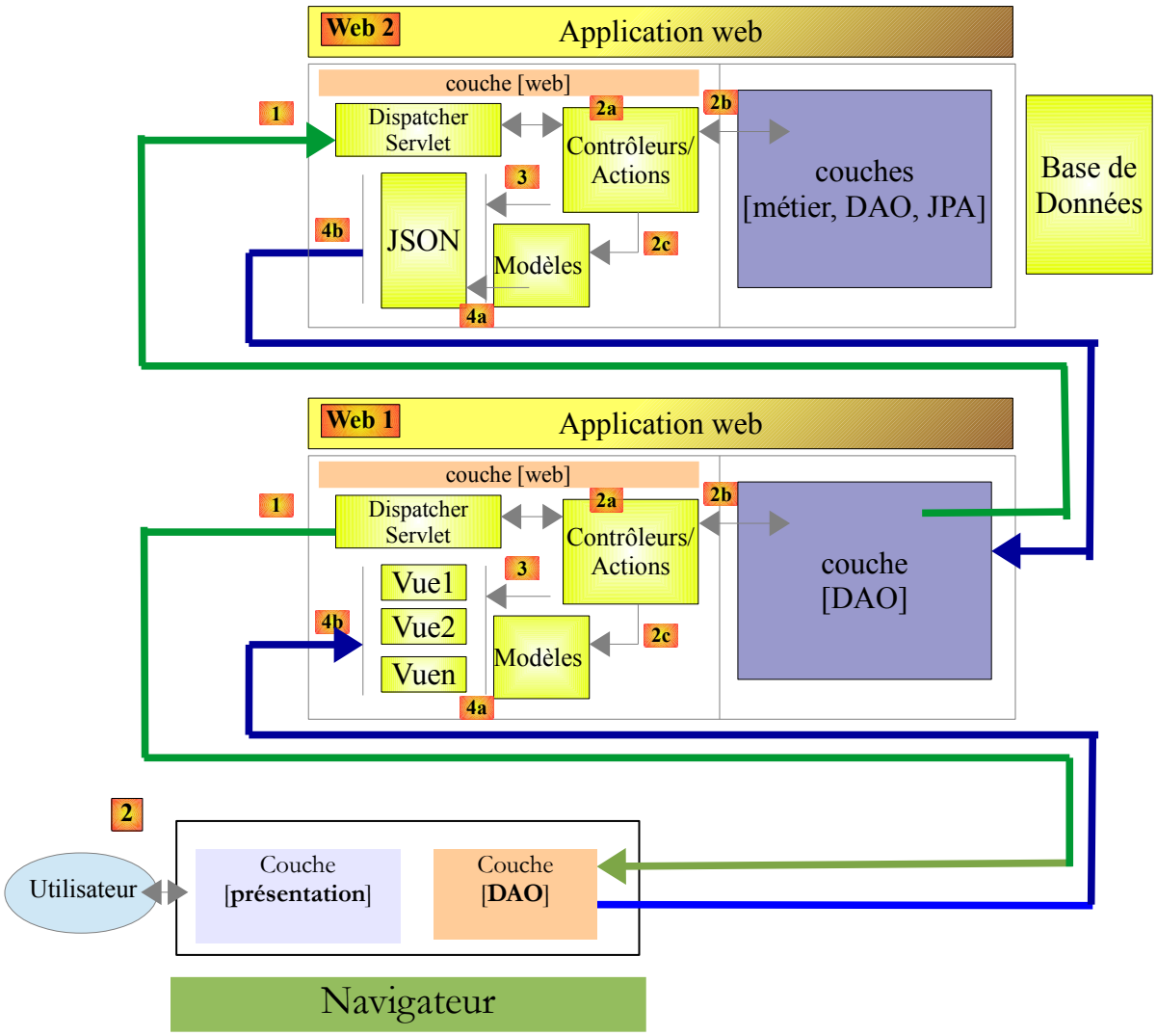
Testez l'application sur l'émulateur. Elle doit fonctionner. Côté serveur, on peut ou non autoriser les entêtes CORS dans la classe [ApplicationModel] :

```
  // CORS
  private final boolean CORS_ALLOWED=false;
```

Cela n'a pas d'importance pour l'application Android. Celle-ci ne s'exécute pas dans un navigateur. Or l'exigence des entêtes CORS vient du navigateur et non pas du serveur.

8.6.10 Conclusion de l'étude de cas

Nous avons développé l'architecture suivante :



C'est une architecture 3tier complexe. Elle visait à réutiliser la couche [Web2] qui était la couche serveur de l'application [AngularJS-Spring MVC] du document [Tutoriel AngularJS / Spring 4] à l'URL [http://tahe.developpez.com/angularjs-spring4/]. C'est uniquement pour cette raison qu'on a une architecture 3tier. Là où dans l'application [AngularJS-Spring MVC], le client de [Web2] était un client [AngularJS], ici le client de [Web2] est une architecture 2tier [jQuery] / [Spring MVC / Thymeleaf]. On a augmenté les couches donc on va perdre en performances.

L'application étudiée ici a été développée au cours du temps dans trois documents différents :

1. [Introduction aux frameworks JSF2, Primefaces et Primefaces mobile] à l'URL [http://tahe.developpez.com/java/primefaces/]. L'étude de cas avait alors été développée avec les frameworks JSF2 / Primefaces. Primefaces est une bibliothèque de composants ajaxifiés qui évite d'écrire du javascript. L'application développée alors, était moins complexe que celle étudiée ici. Elle avait une version web classique pour l'ordinateur et une version mobile pour les téléphones ;
2. [Tutoriel AngularJS / Spring 4] à l'URL [http://tahe.developpez.com/angularjs-spring4/]. L'application développée alors, avait les mêmes caractéristiques que celle étudiée dans ce document. L'application avait également été portée sur Android ;
3. le présent document ;

De ce travail, il ressort pour moi les points suivants :

- l'application [Primefaces] a été de loin la plus simple à écrire et sa version web mobile s'est révélée performante. Elle ne nécessite pas de connaissances Javascript. Il n'est pas possible de la porter nativement sur les OS des différents mobiles mais est-ce nécessaire ? Il semble difficile de changer le style de l'application. On travaille en effet avec les feuilles de style de Primefaces. Ce peut-être un inconvénient ;

- l'application [AngularJS-Spring MVC] a été complexe à écrire. Le framework [AngularJS] m'a semblé assez difficile à appréhender dès lors qu'on veut le maîtriser. L'architecture [client Angular] / [service web / jSON implémenté par Spring MVC] est particulièrement propre et performante. Cette architecture est reproductible pour toute application web. C'est l'architecture qui me paraît la plus prometteuse car elle met en jeu côté client et côté serveur des compétences différentes (JS+HTML+CSS côté client, Java ou autre chose côté serveur), ce qui permet de développer le client et le serveur en parallèle ;

- pour l'application développée dans ce document avec une architecture 3tier [client jQuery] / [serveur Web1 / Spring MVC / Thymeleaf] / [serveur Web2 / Spring MVC], il est possible que certains trouvent la technologie [jQuery+Spring MVC+Thymelaf] plus simple à appréhender que celle de [AngularJS]. La couche [DAO] du client Javascript que nous avons écrite est réutilisable dans d'autres applications ;

9 Annexes

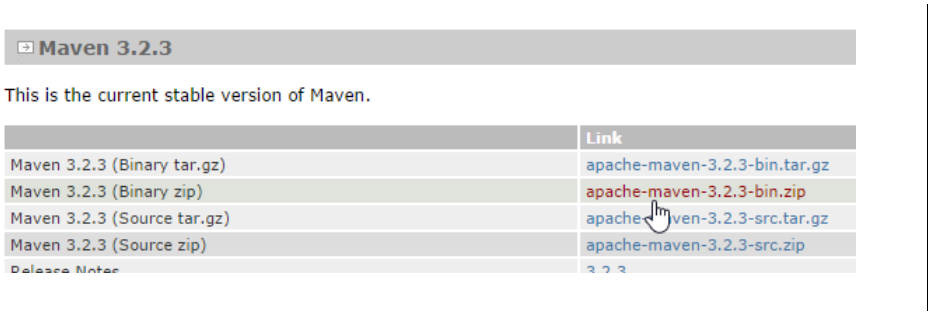
Nous présentons ici comment installer les outils utilisés dans ce document sur des machines windows 7 ou 8.

9.1 Installation d'un JDK

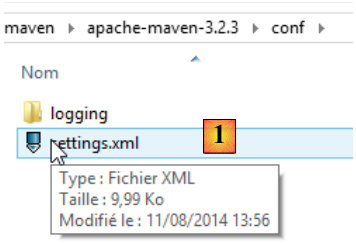
On trouvera à l'URL [http://www.oracle.com/technetwork/java/javase/downloads/index.html] (octobre 2014), le JDK le plus récent. On nommera par la suite <jdk-install> le dossier d'installation du JDK.

9.2 Installation de Maven

Maven est un outil de gestion des dépendances d'un projet Java et plus encore. Il est disponible à l'URL [http://maven.apache.org/download.cgi].



Téléchargez et dézippez l'archive. Nous appellerons <maven-install> le dossier d'installation de Maven.



- en [1], le fichier [conf / settings.xml] configure Maven ;

On y trouve les lignes suivantes :

```
1.    <!-- localRepository
2.     | The path to the local repository maven will use to store artifacts.
3.     |
4.     | Default: ${user.home}/.m2/repository
5.    <localRepository>/path/to/local/repo</localRepository>
6.  -->
```

La valeur par défaut de la ligne 4, si comme moi votre {user.home} a un espace dans son chemin (par exemple [C:\Users\Serge Tahé]), peut poser problème à certains logiciels dont IntellijIDEA. On écrira alors quelque chose comme :

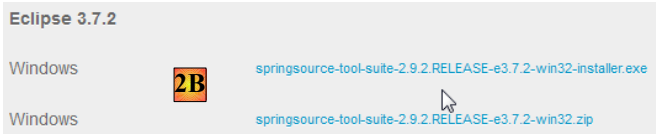
```
1.    <!-- localRepository
2.     | The path to the local repository maven will use to store artifacts.
3.     |
4.     | Default: ${user.home}/.m2/repository
5.    <localRepository>/path/to/local/repo</localRepository>
6.    -->
7.  <localRepository>D:\Programs\devjava\maven\.m2\repository</localRepository>
```

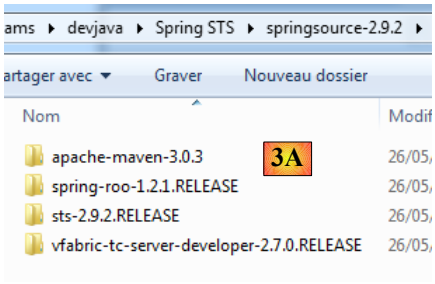
et on évitera, ligne 7, un chemin qui contient des espaces.

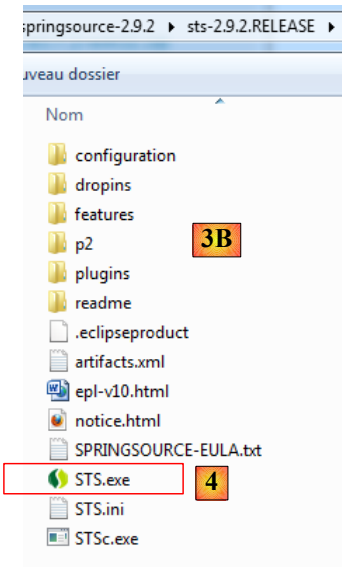
9.3 Installation de STS (Spring Tool Suite)

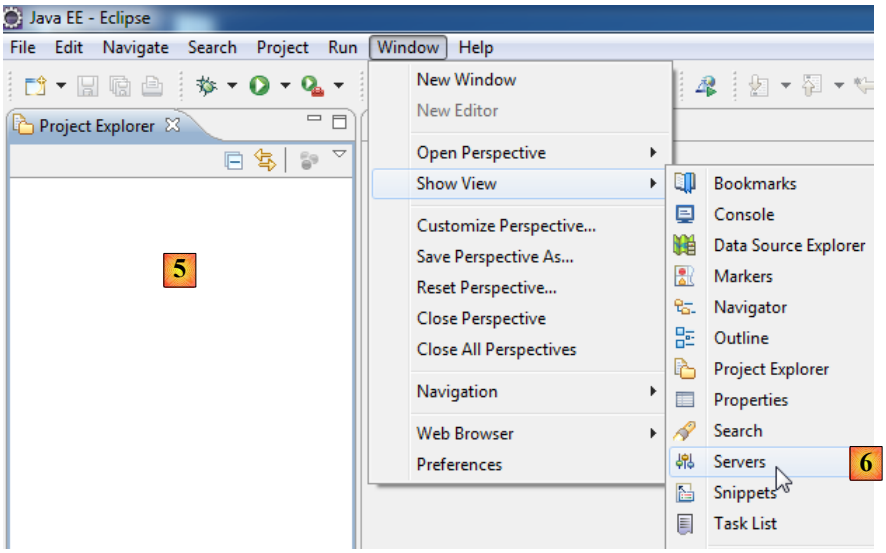
Nous allons installer **SpringSource Tool Suite** [http://www.springsource.com/developer/sts], un Eclipse pré-équipé avec de nombreux plugins liés au framework Spring et également avec une configuration Maven pré-installée.

- aller sur le site de **SpringSource Tool Suite** (STS) [1], pour télécharger la version courante de STS [2A] [2B],

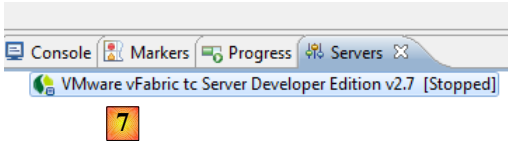






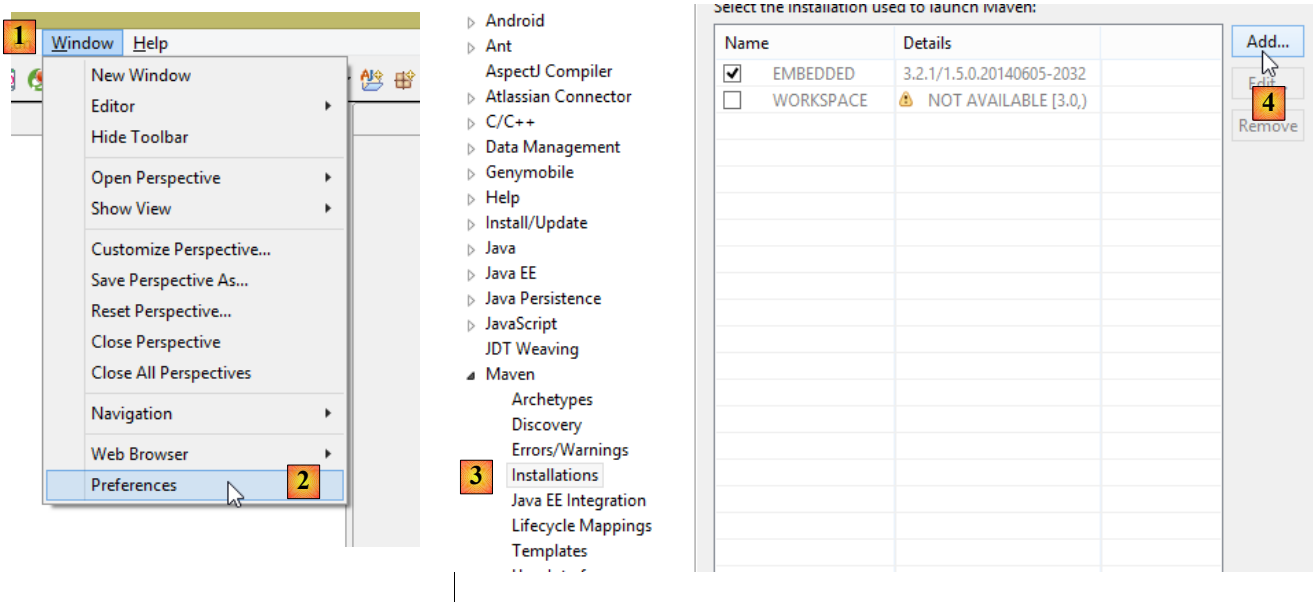


- le fichier téléchargé est un installateur qui crée l'arborescence de fichiers [3A] [3B]. En [4], on lance l'exécutable,
- en [5], la fenêtre de travail de l'IDE après avoir fermé la fenêtre de bienvenue. En [6], on fait afficher la fenêtre des serveurs d'applications,

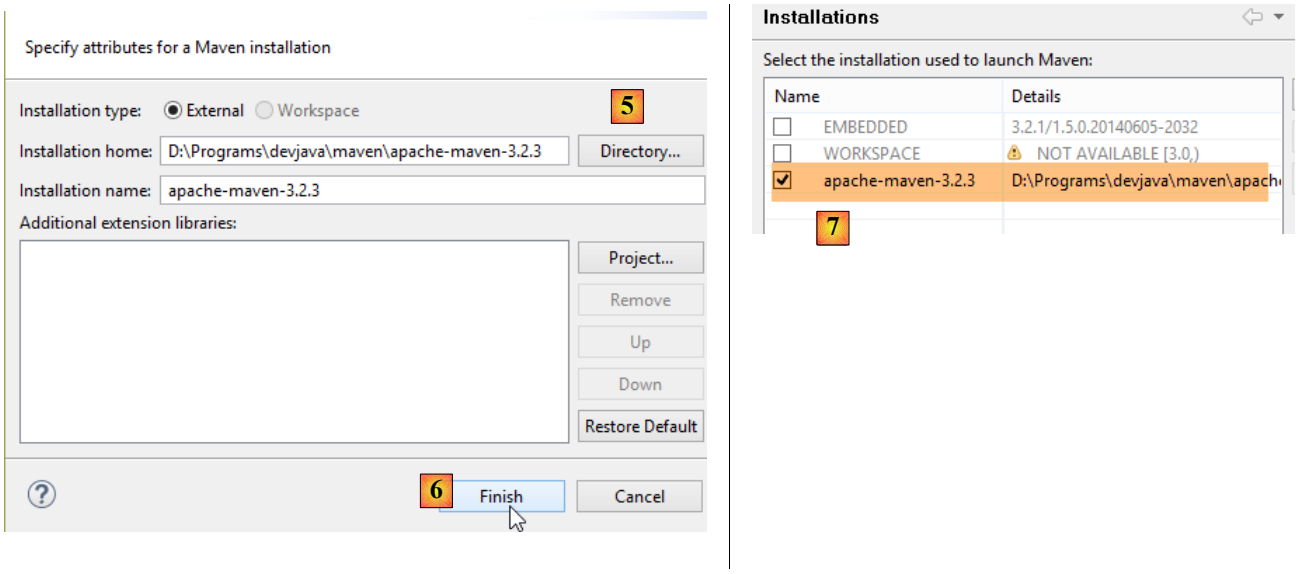


- en [7], la fenêtre des serveurs. Un serveur est enregistré. C'est un serveur VMware compatible Tomcat.

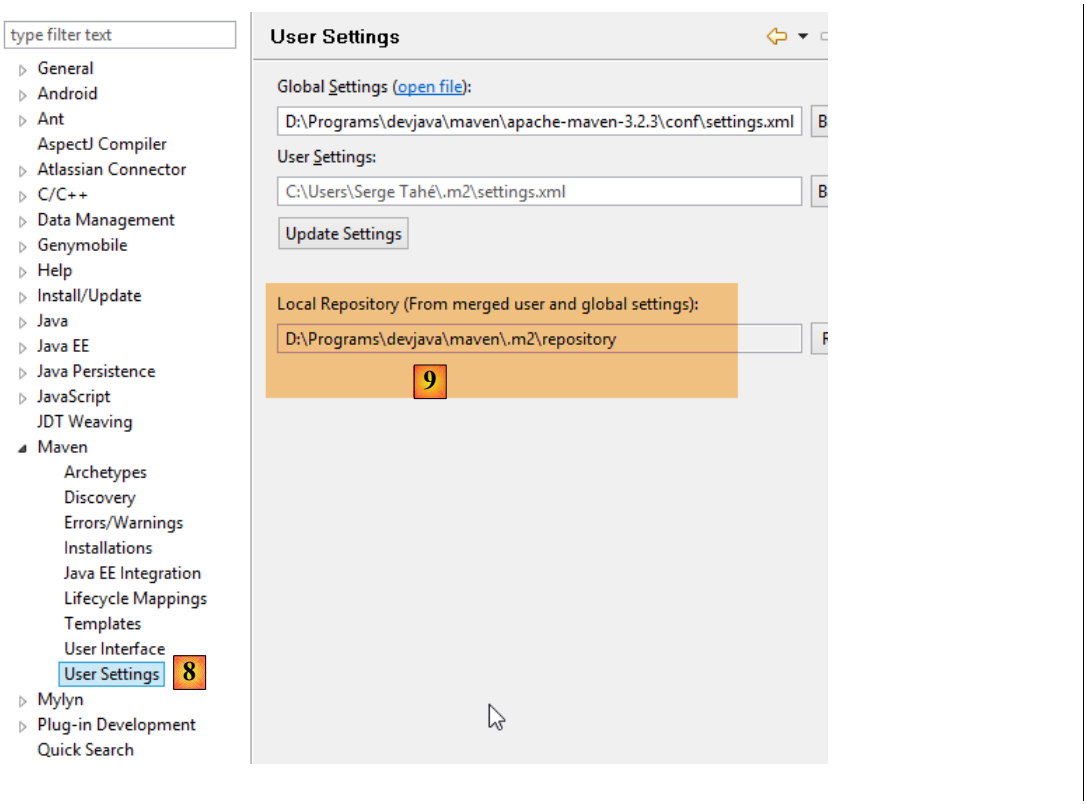
Il faut indiquer à STS le dossier d'installation de Maven :



- en [1-2], on configure STS ;
- en [3-4], on ajoute une nouvelle installation Maven ;



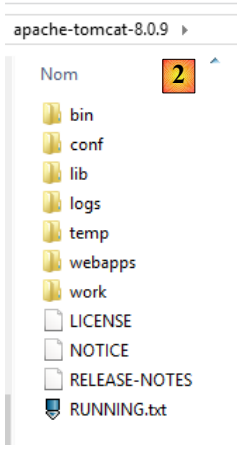
- en [5], on désigne le dossier d'installation de Maven ;
- en [6], on termine l'assistant ;
- en [7], on fait de la nouvelle installation Maven, l'installation par défaut ;



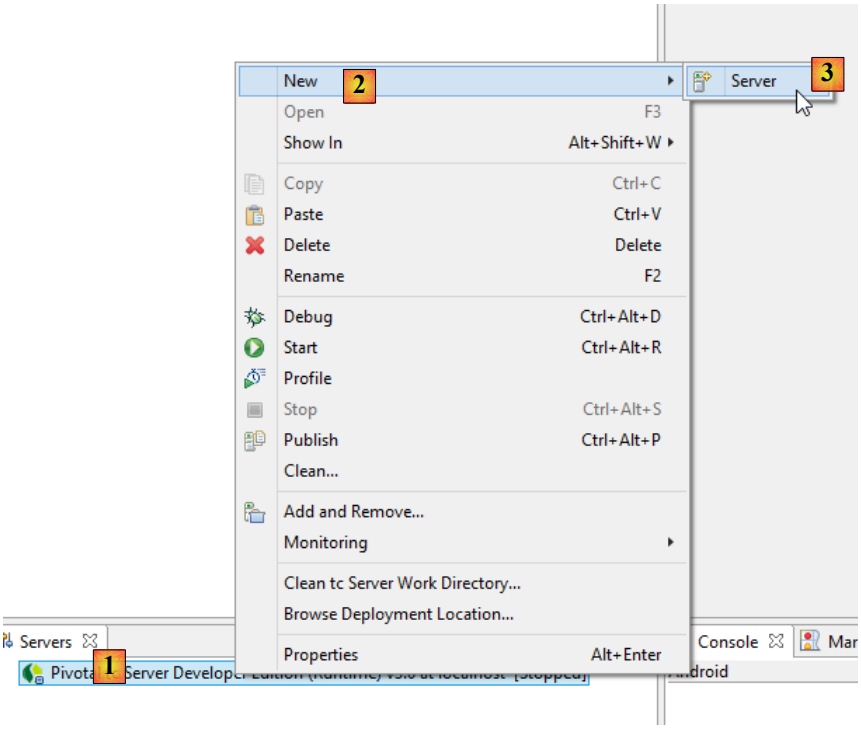
- en [8-9], on vérifie le dépôt local de Maven, le dossier où il mettra les dépendances qu'il téléchargera et où STS mettra les artifacts qui seront construits ;

9.4 Installation d'un serveur Tomcat

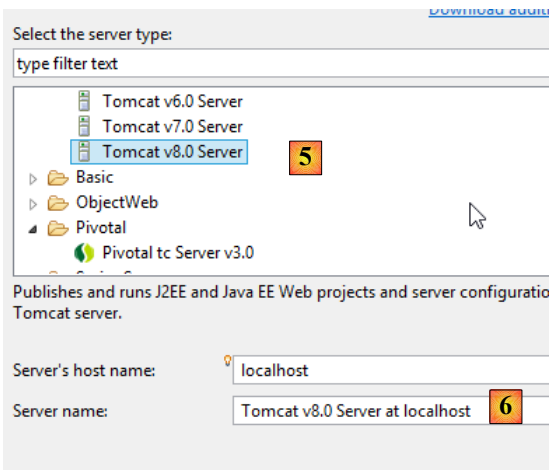
Les serveurs Tomcat sont disponibles à l'URL [http://tomcat.apache.org/download-80.cgi]. Les exemples de ce document ont été testés avec la version 8.0.9 disponible à l'URL [http://archive.apache.org/dist/tomcat/tomcat-8/v8.0.9/bin/].

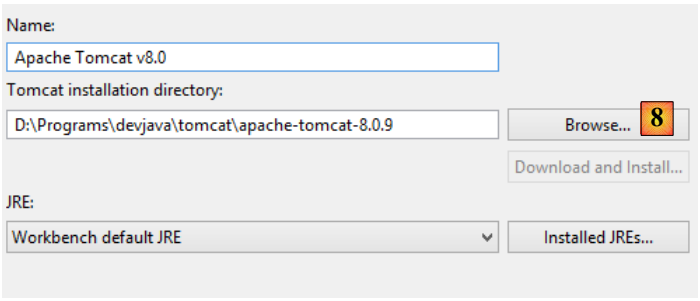


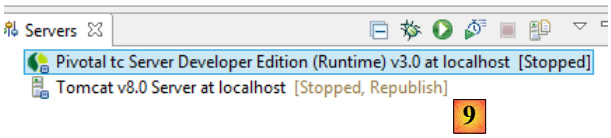
On télécharge [1] le zip qui convient au poste de travail. Une fois dézippé on obtient l'arborescence [2]. Ceci fait, on peut ajouter ce serveur aux serveurs de STS :



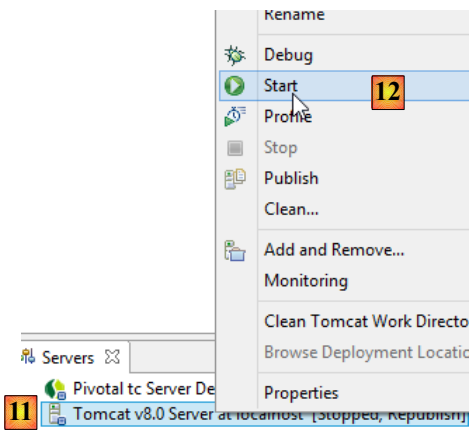
- en [1-3], on ajoute un nouveau serveur dans STS ; (pour avoir la feneêtre des serveurs faire Window / Show view / Other / Server / Servers) ;

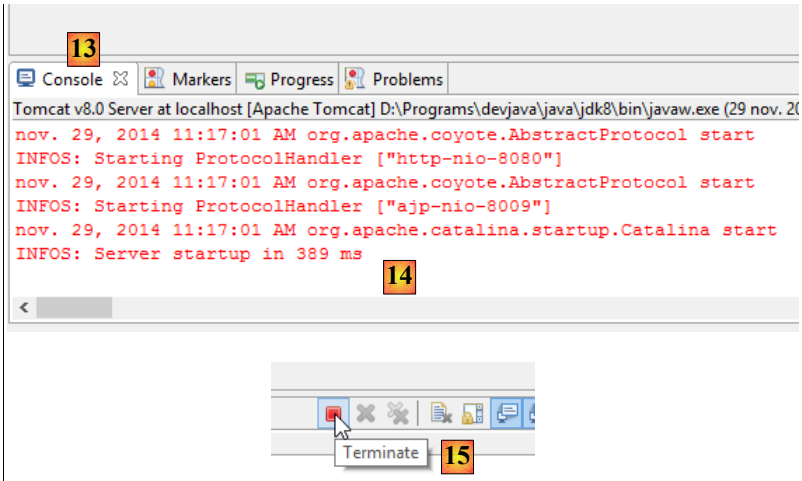






- en [5], choisir un serveur Tomcat 8 ;
- en [6], donner un nom à ce serveur ;
- en [8], indiquer le dossier d'installation du Tomcat téléchargé précédemment ;
- en [9], le nouveau serveur ;

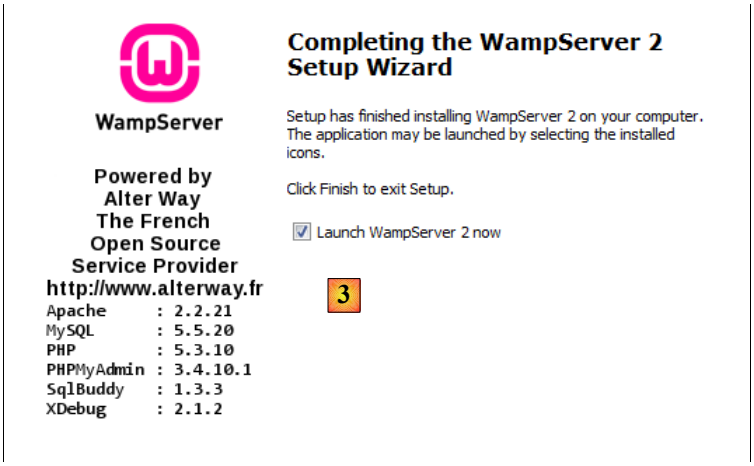




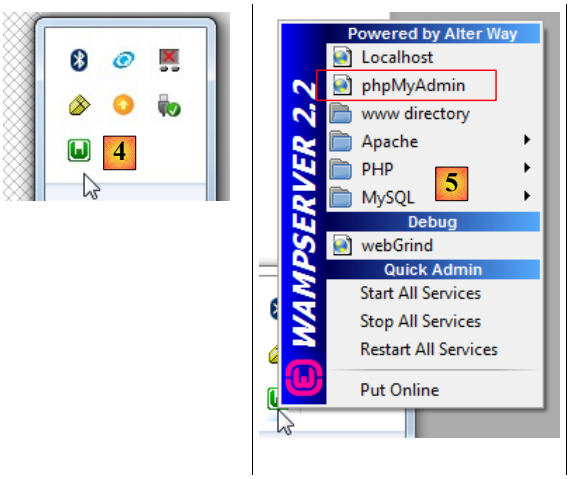
- en [11-12], lancer le serveur Tomcat 8 ;
- en [13-14], sa fenêtre de logs ;
- en [15], pour l'arrêter ;

9.5 Installation de [WampServer]

[WampServer] est un ensemble de logiciels pour développer en PHP / MySQL / Apache sur une machine Windows. Nous l'utiliserons uniquement pour le SGBD MySQL.



- sur le site de [WampServer] [1], choisir la version qui convient [2],
- l'exécutable téléchargé est un installateur. Diverses informations sont demandées au cours de l'installation. Elles ne concernent pas MySQL. On peut donc les ignorer. La fenêtre [3] s'affiche à la fin de l'installation. On lance [WampServer],



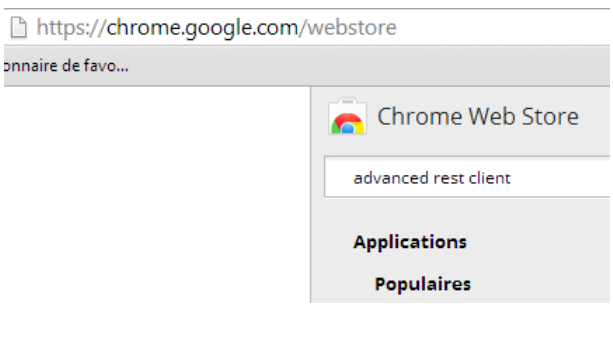
- en [4], l'icône de [WampServer] s'installe dans la barre des tâches en bas et à droite de l'écran [4],
- lorsqu'on clique dessus, le menu [5] s'affiche. Il permet de gérer le serveur Apache et le SGBD MySQL. Pour gérer celui-ci, on utiliser l'option [PhpPmyAdmin],
- on obtient alors la fenêtre ci-dessous,

Nous donnerons ici peu de détails sur l'utilisation de [PhpMyAdmin]. Le document donne les informations à connaître lorsque c'est nécessaire.

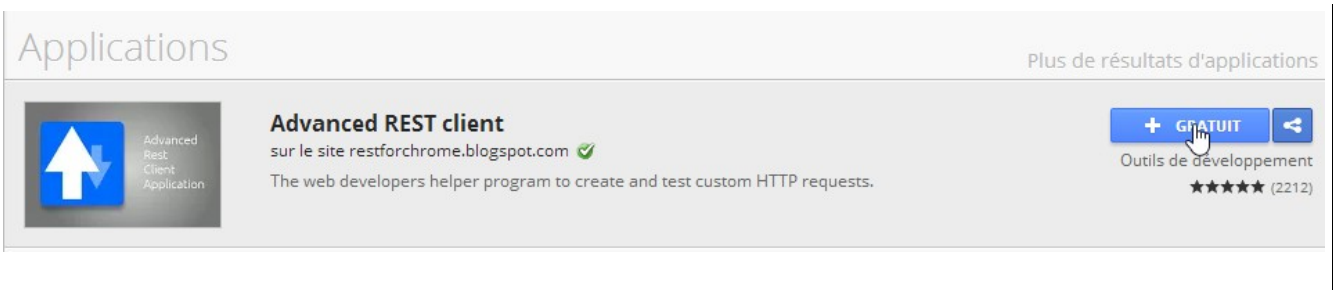
9.6 Installation du plugin Chrome [Advanced Rest Client]

Dans ce document, on utilise le navigateur **Chrome** de Google (http://www.google.fr/intl/fr/chrome/browser/). On lui ajoutera l'extension [Advanced Rest Client]. On pourra procéder ainsi :

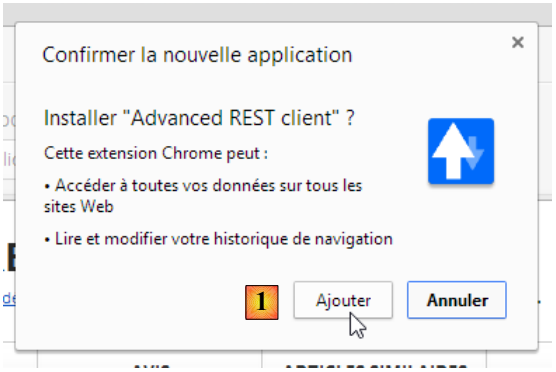
- aller sur le site de [Google Web store] (https://chrome.google.com/webstore) avec le navigateur Chrome ;
- chercher l'application [Advanced Rest Client] :

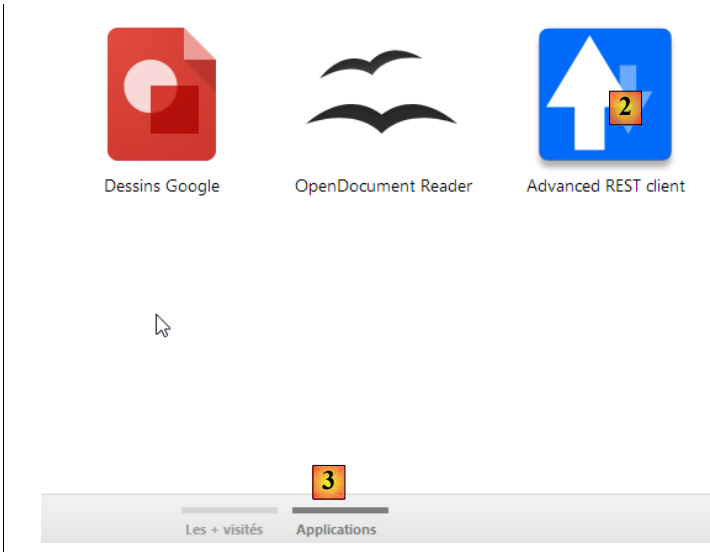


- l'application est alors disponible au téléchargement :



- pour l'obtenir, il vous faudra créer un compte Google. [Google Web Store] demande ensuite confirmation [1] :





- en [2], l'extension ajoutée est disponible dans l'option [Applications] [3]. Cette option est affichée sur chaque nouvel onglet que vous créez (CTRL-T) dans le navigateur.

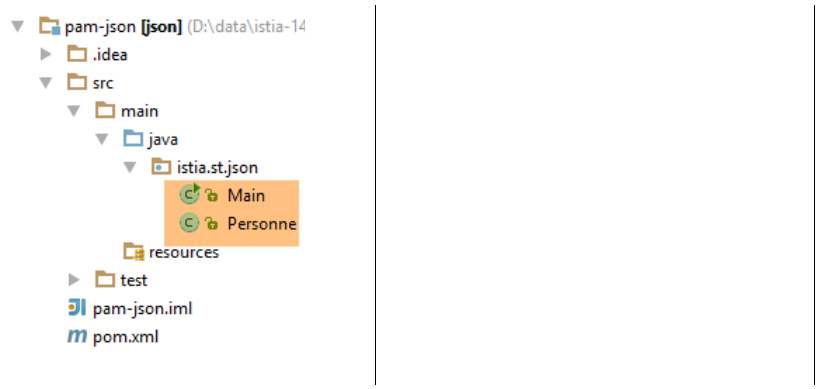
9.7 Gestion du jSON en Java

De façon transparente pour le développeur le framework [Spring MVC] utilise la bibliothèque jSON [Jackson]. Pour illustrer ce qu'est le jSON (JavaScript Object Notation), nous présentons ici un programme qui sérialise des objets en jSON et fait l'inverse en désérialisant les chaînes jSON produites pour recréer les objets initiaux.

La bibliothèque 'Jackson' permet de construire :

- la chaîne jSON d'un objet : *new ObjectMapper().writeValueAsString(object)* ;
- un objet à partir d'un chaîne jSON : *new ObjectMapper().readValue(jsonString, Object.class)*.

Les deux méthodes sont susceptibles de lancer une *IOException*. Voici un exemple.



Le projet ci-dessus est un projet Maven avec le fichier [pom.xml] suivant ;

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0"
3.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    4.0.0.xsd">
5.    <modelVersion>4.0.0</modelVersion>
6.
7.    <groupId>istia.st.pam</groupId>
8.    <artifactId>json</artifactId>
9.    <version>1.0-SNAPSHOT</version>
10.
11.   <dependencies>
12.     <dependency>
13.       <groupId>com.fasterxml.jackson.core</groupId>
14.       <artifactId>jackson-databind</artifactId>
15.       <version>2.3.3</version>
16.     </dependency>
17.   </dependencies>
18. </project>
```

- lignes 12-16 : la dépendance qui amène la bibliothèque 'Jackson' ;

La classe [Personne] est la suivante :

```
1.  package istia.st.json;
2.
3.  public class Personne {
4.     // data
5.     private String nom;
6.     private String prenom;
7.     private int age;
8.
9.     // constructeurs
10.    public Personne() {
11.
12.    }
13.
14.    public Personne(String nom, String prénom, int âge) {
15.       this.nom = nom;
16.       this.prenom = prénom;
17.       this.age = âge;
18.    }
19.
20.    // signature
21.    public String toString() {
22.       return String.format("Personne[%s, %s, %d]", nom, prenom, age);
23.    }
24.
25.    // getters et setters
26. ...
```

```
27. }
```

La classe [Main] est la suivante :

```
1.  package istia.st.json;
2.
3.  import com.fasterxml.jackson.databind.ObjectMapper;
4.
5.  import java.io.IOException;
6.  import java.util.HashMap;
7.  import java.util.Map;
8.
9.  public class Main {
10.   // l'outil de sérialisation / désérialisation
11.   static ObjectMapper mapper = new ObjectMapper();
12.
13.   public static void main(String[] args) throws IOException {
14.     // création d'une personne
15.     Personne paul = new Personne("Denis", "Paul", 40);
16.     // affichage jSON
17.     String json = mapper.writeValueAsString(paul);
18.     System.out.println("Json=" + json);
19.     // instanciation Personne à partir du Json
20.     Personne p = mapper.readValue(json, Personne.class);
21.     // affichage personne
22.     System.out.println("Personne=" + p);
23.     // un tableau
24.     Personne virginie = new Personne("Radot", "Virginie", 20);
25.     Personne[] personnes = new Personne[]{paul, virginie};
26.     // affichage Json
27.     json = mapper.writeValueAsString(personnes);
28.     System.out.println("Json personnes=" + json);
29.     // dictionnaire
30.     Map<String, Personne> hpersonnes = new HashMap<String, Personne>();
31.     hpersonnes.put("1", paul);
32.     hpersonnes.put("2", virginie);
33.     // affichage Json
34.     json = mapper.writeValueAsString(hpersonnes);
35.     System.out.println("Json hpersonnes=" + json);
36.   }
37. }
```

L'exécution de cette classe produit l'affichage écran suivant :

```
1.  Json={"nom":"Denis","prenom":"Paul","age":40}
2.  Personne=Personne[Denis, Paul, 40]
3.  Json personnes=[{"nom":"Denis","prenom":"Paul","age":40},
    {"nom":"Radot","prenom":"Virginie","age":20}]
4.  Json hpersonnes={"2":{"nom":"Radot","prenom":"Virginie","age":20},"1":
    {"nom":"Denis","prenom":"Paul","age":40}}
```

De l'exemple on retiendra :

- l'objet [ObjectMapper] nécessaire aux transformations jSON / Object : ligne 11 ;
- la transformation [Personne] --> jSON : ligne 17 ;
- la transformation jSON --> [Personne] : ligne 20 ;
- l'exception [IOException] lancée par les deux méthodes : ligne 13.

9.8 Installation de [Webstorm]

[WebStorm] (WS) est l'IDE de JetBrains pour développer des applications HTML / CSS / JS. Je l'ai trouvé parfait pour développer des applications Angular. Le site de téléchargement est [http://www.jetbrains.com/webstorm/download/]. C'est un IDE payant mais une version d'évaluation de 30 jours est téléchargeable. Il existe une version personnelle et une version étudiante peu onéreuses.

Pour installer des bibliothèques JS au sein d'une application, WS utilise un outil appelé [bower]. Cet outil est un module de [node.js], un ensemble de bibliothèques JS. Par ailleurs, les bibliothèques JS sont cherchées sur un site Git, nécessitant un client Git sur le poste qui télécharge.

9.8.1 Installation de [node.js]

Le site de téléchargement de [node.js] est [http://nodejs.org/]. Téléchargez l'installateur puis exécutez-le. Il n'y a rien de plus à faire pour le moment.

9.8.2 Installation de l'outil [bower]

L'installation de l'outil [bower] qui va permettre le téléchargement des bibliothèques Javascript peut se faire de différentes façons. Nous allons la faire à partir de la console :

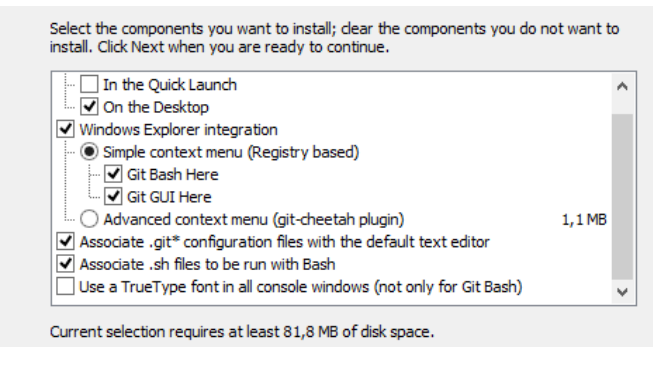
```
1.  C:\Users\Serge Tahé>npm install -g bower
2.  C:\Users\Serge Tahé\AppData\Roaming\npm\bower -> C:\Users\Serge
    Tahé\AppData\Roaming\npm\node_modules\bower\bin\bower
3.  bower@1.3.7 C:\Users\Serge Tahé\AppData\Roaming\npm\node_modules\bower
4.  ├── stringify-object@0.2.1
5.  ├── is-root@0.1.0
6.  ├── junk@0.3.0
7.  ...
8.  ├── insight@0.3.1 (object-assign@0.1.2, async@0.2.10, lodash.debounce@2.4.1, req
9.  uest@2.27.0, configstore@0.2.3, inquirer@0.4.1)
10. ├── mout@0.9.1
11. └── inquirer@0.5.1 (readline2@0.1.0, mute-stream@0.0.4, through@2.3.4, async@0.8
12. .0, lodash@2.4.1, cli-color@0.3.2)
```

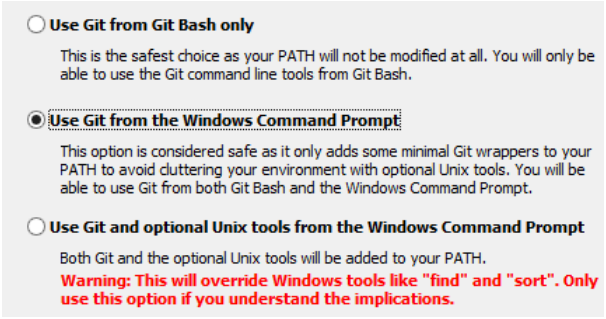
- ligne 1 : la commande [node.js] qui installe le module [bower]. Pour que la commande marche, il faut que l'exécutable [npm] soit dans le PATH de la machine (voir paragraphe ci-après) ;

9.8.3 Installation de [Git]

Git est un système de gestion de versions de logiciel. Il existe une version windows appelée [msysgit] et disponible à l'URL [http://msysgit.github.io/]. Nous n'allons pas utiliser [msysgit] pour gérer des versions de notre application mais simplement pour télécharger des bibliothèques JS qui se trouvent sur des sites de type [https://github.com] qui nécessitent un protocole d'accès spécial et qui est fourni par le client [msysgit]

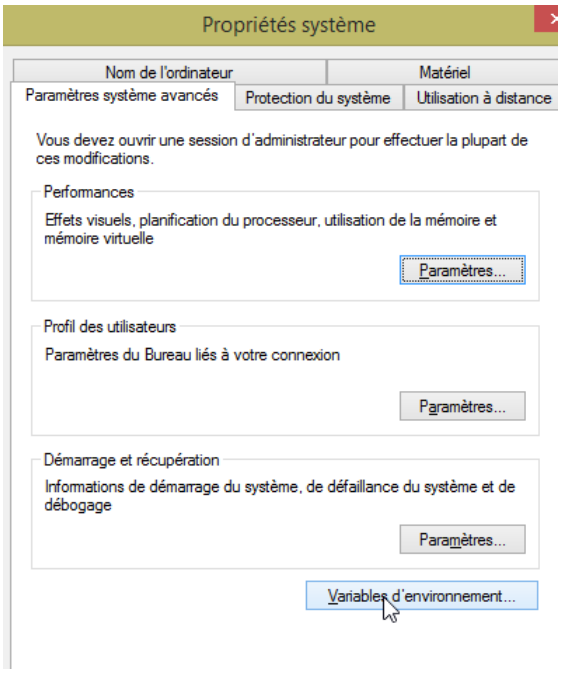
L'assistant d'installation propose différentes étapes dont les suivantes :

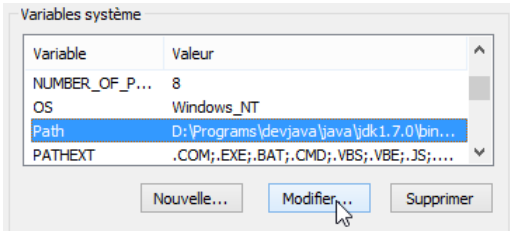




Pour les autres étapes de l'installation, vous pouvez accepter les valeurs par défaut proposées.

Une fois, l'installation de Git terminée, vérifiez que l'exécutable est dans le PATH de votre machine : [Panneau de configuration / Système et sécurité / Système / Paramètres systèmes avancés] :





La variable PATH ressemble à ceci :

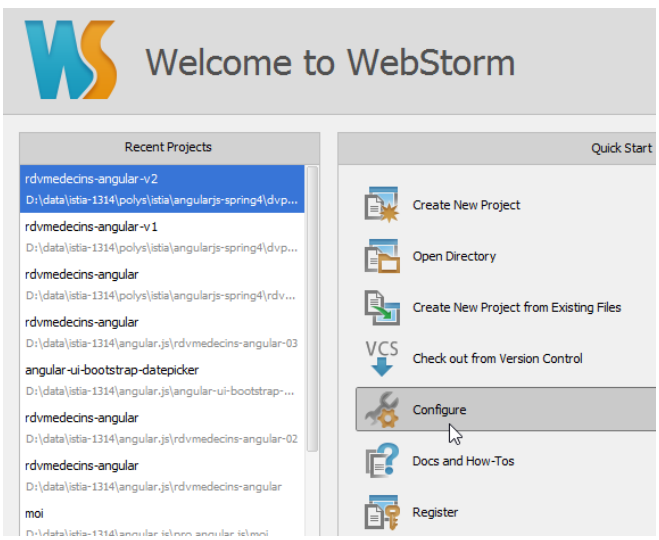
```
D:\Programs\devjava\java\jdk1.7.0\bin;D:\Programs\ActivePerl\Perl64\site\bin;D:\Programs\ActivePerl\Perl64\bin;D:\Programs\sgbd\OracleXE\app\oracle\product\11.2.0\client;D:\Programs\sgbd\OracleXE\app\oracle\product\11.2.0\client\bin;D:\Programs\sgbd\OracleXE\app\oracle\product\11.2.0\server\bin;...;D:\Programs\javascript\node.js\;D:\Programs\utilitaires\Git\cmd
```

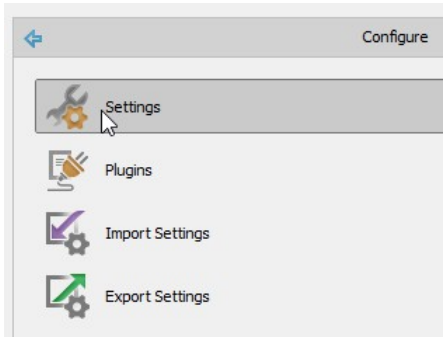
Vérifiez que :

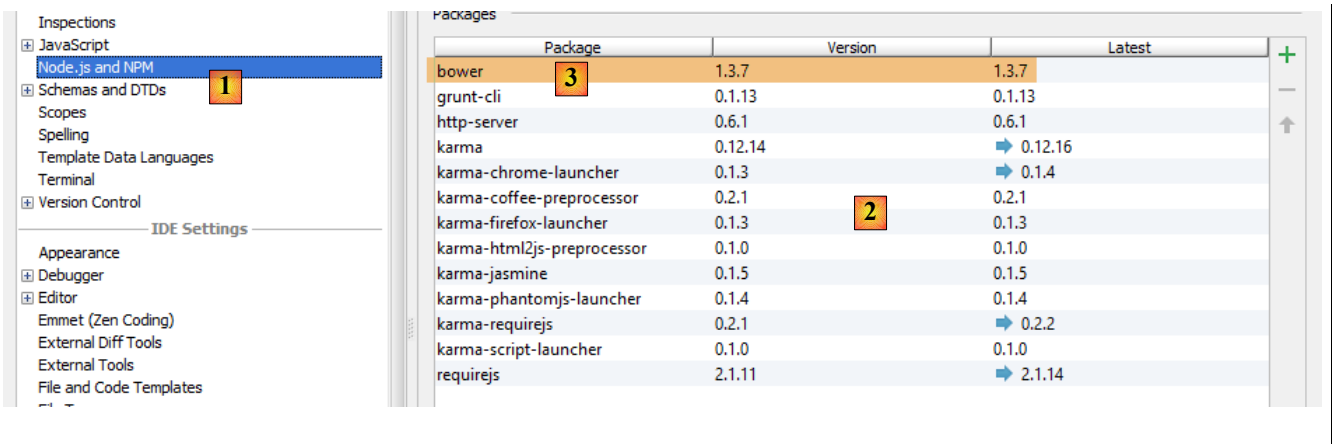
- le chemin du dossier d'installation de [node.js] est bien présent (ici D:\Programs\javascript\node.js) ;
- le chemin de l'excéutable du client Git est bien présent (ici D:\Programs\utilitaires\Git\cmd) ;

9.8.4 Configuration de [Webstorm]

Vérifions maintenant la configuration de [Webstorm]







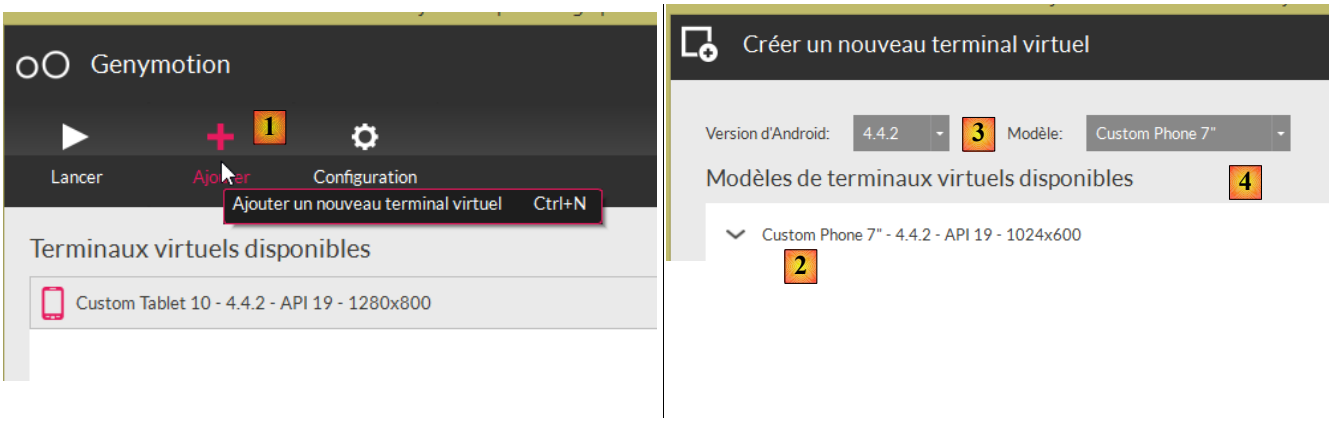
Ci-dessus, sélectionnez l'option [1]. La liste des modules [node.js] déjà installés apparaît en [2]. Cette liste ne devrait contenir que la ligne [3] du module [bower] si vous avez suivi le processus d'installation précédent.

9.9 Installation d'un émulateur pour Android

Les émulateurs fournis avec le SDK d'Android sont lents ce qui décourage de les utiliser. L'entreprise [Genymotion] offre un émulateur beaucoup plus performant. Celui-ci est disponible à l'URL [https://cloud.genymotion.com/page/launchpad/download/] (février 2014).

Vous aurez à vous enregistrer pour obtenir une version à usage personnel. Téléchargez le produit [Genymotion] avec la machine virtuelle VirtualBox ;

Installez puis lancez [Genymotion]. Téléchargez ensuite une image pour une tablette ou un téléphone :



- en [1], ajoutez un terminal virtuel ;
- en [2], choisissez un ou plusieurs terminaux à installer. Vous pouvez affiner la liste affichée en précisant la version d'Android désirée [3] ainsi que le modèle de terminal [4] ;

- une fois le téléchargement terminé, vous obtenez en [5] la liste des terminaux virtuels dont vous disposez pour tester vos applications Android ;