Introduction par l'exemple à Java Server Faces, PrimeFaces et PrimeFaces Mobile

serge.tahe at istia.univ-angers.fr, juin 2012

1 Introduction

1.1 Présentation

Nous nous proposons ici d'introduire, à l'aide d'exemples
- le framework **Java Server Faces 2** (JSF2),
- la bibliothèque de composants **PrimeFaces** pour JSF2, pour les applications de bureau et les mobiles.

Ce document est principalement destiné aux étudiants et développeurs intéressés par la bibliothèque de composants PrimeFaces [http://www.primefaces.org]. Ce framework propose plusieurs dizaines de composants ajaxifiés [http://www.primefaces.org/showcase/ui/home.jsf] permettant de construire des applications web ayant un aspect et un comportement analogues à ceux des applications de bureau. Primefaces s'appuie sur JSF2 d'où la nécessité de présenter d'abord ce framework. Primefaces propose de plus des composants spécifiques pour les mobiles. Nous les présenterons également.

Pour illustrer notre propos nous construirons une application de prise de rendez-vous dans différents environnements :
1. une application web classique avec les technologies JSF2 / EJB3 / JPA sur un serveur Glassfish 3.1,
2. la même application ajaxifiée, avec la technologie PrimeFaces,
3. et pour finir, une version mobile avec la technologie PrimeFaces Mobile.

A chaque fois qu'une version aura été construite, nous la porterons dans un environnement JSF2 / Spring / JPA sur un serveur Tomcat 7. Nous construirons donc six applications Java EE.

Le document ne présente **que ce qui est nécessaire pour construire ces applications**. Aussi ne doit-on pas y chercher l'exhaustivité. On ne la trouvera pas. Ce n'est pas non plus un recueil de bonnes pratiques. Des développeurs aguerris pourront ainsi trouver qu'ils auraient fait les choses différemment. J'espère simplement que ce document n'est pas non plus un recueil de mauvaises pratiques.

Pour approfondir JSF2 et PrimeFaces on pourra utiliser les références suivantes :

- **[ref1] : Java EE5 Tutorial** [http://java.sun.com/javaee/5/docs/tutorial/doc/index.HTML], la référence Sun pour découvrir Java EE5. On lira la partie II [Web Tier] pour découvrir la technologie web et notamment JSF. Les exemples du tutoriel peuvent être téléchargés. Ils viennent sous forme de projets Netbeans qu'on peut charger et exécuter,
- **[ref2] : Core JavaServer Faces** de David Geary et Cay S. Horstmann, troisième édition, aux éditions Mc Graw-Hill,
- **[ref3]** : la documentation de PrimeFaces [http://primefaces.googlecode.com/files/primefaces_users_guide_3_2.pdf],
- **[ref4]** : la documentation de PrimeFaces Mobile : [http://primefaces.googlecode.com/files/primefaces_mobile_users_guide_0_9_2.pdf].

Nous ferons parfois référence à **[ref2]** pour indiquer au lecteur qu'il peut approfondir un domaine avec ce livre.

Le document a été écrit de telle façon qu'il puisse être compris par le plus grand nombre. Les pré-requis nécessaires à sa compréhension sont les suivants :

- connaissance du langage Java,
- connaissances de base du développement Java EE.

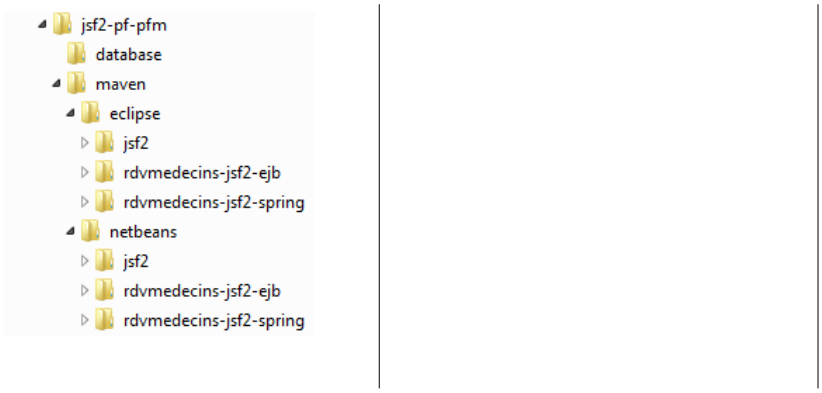
On trouvera sur le site [http://developpez.com] toutes les ressources nécessaires à ces pré-requis. On en trouvera certaines sur le site [http://tahe.developpez.com] :

- **[ref5] : Introduction à la programmation web en Java** (septembre 2002) [http://tahe.developpez.com/java/web/] : donne les bases de la programmation web en Java : **servlets** et **pages JSP,**
- **[ref6] : Les bases du développement web MVC en Java** (mai 2006) [http://tahe.developpez.com/java/baseswebmvc] : préconise le développement d'applications web avec des architectures à trois couches, où la couche web implémente le modèle de conception (Design Pattern) **MVC** (Modèle, Vue, Contrôleur).
- [**ref7**] : **Introduction à Java EE 5** (juin 2010) [http://tahe.developpez.com/java/javaee]. Ce document permet de découvrir **JSF 1** et les **EJB3**.
- [**ref8**] : **Persistance Java par la pratique** (juin 2007) [http://tahe.developpez.com/java/jpa]. Ce document permet de découvrir la persistance des données avec **JPA** (**Java Persistence API**).

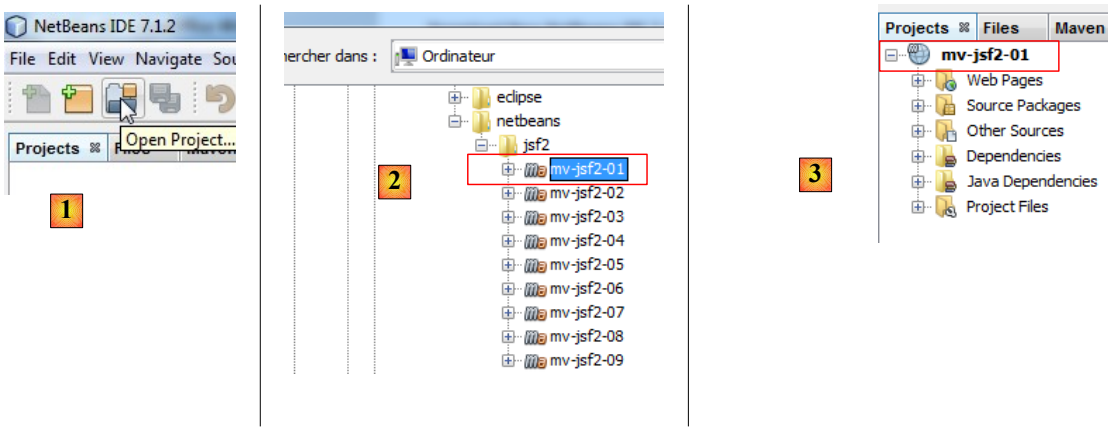
- [**ref9**] : **Construire un service web Java EE avec Netbeans et le serveur Glassfish** (janvier 2009) [http://tahe.developpez.com/java/webservice-jee]. Ce document étudie la construction d'un service web. L'application exemple qui est étudiée dans le présent document provient de [**ref9**].

1.2 Les exemples

Les exemples du présent document sont disponibles à l'URL [http://tahe.ftp-developpez.com/fichiers-archive/jsf2-pf-pfm.zip] sous la forme de projets Maven à la fois pour Netbeans et Eclipse :

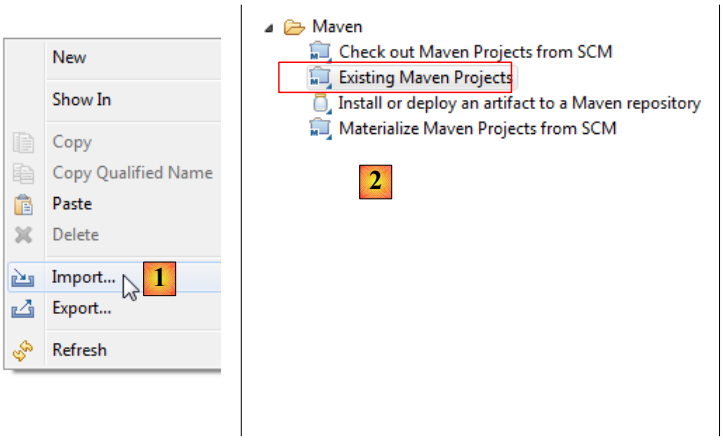


Pour importer un projet sous Netbeans :

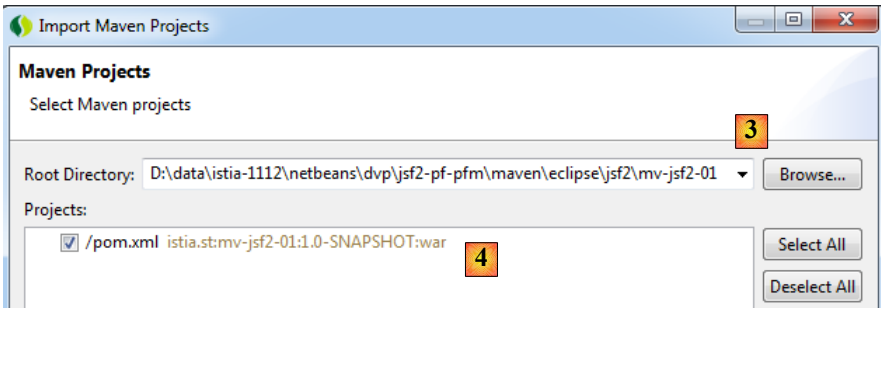


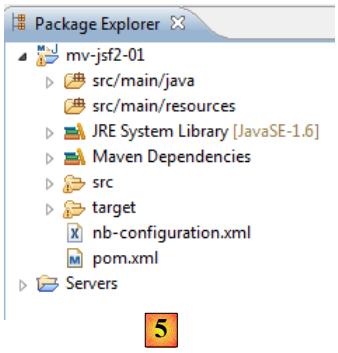
- en [1], on ouvre un projet,
- en [2], on désigne le projet à ouvrir dans le système de fichiers,
- en [3], il est ouvert.

Avec Eclipse,



- en [1], on importe un projet,
- en [2], le projet est un projet Maven existant,





- en [3] : on le désigne dans le système de fichiers,
- en [4] : Eclipse le reconnaît comme un projet Maven,
- en [5], le projet importé.

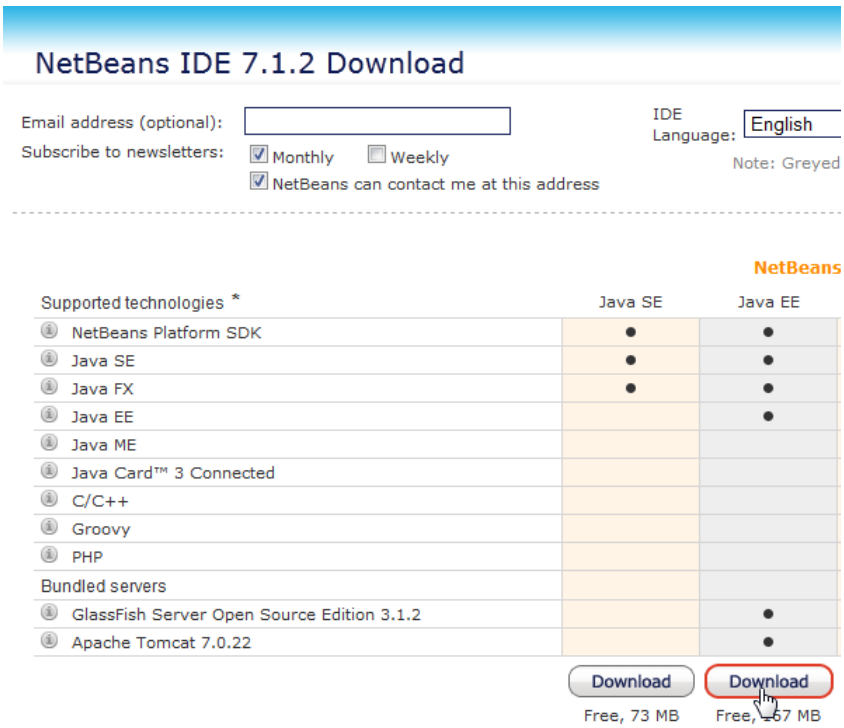
1.3 Les outils utilisés

Dans la suite, nous utilisons (mai 2012) :

- l'IDE (**I**ntegrated **D**evelopment **E**nvironment) **Netbeans** 7.1.2 disponible à l'URL [http://www.netbeans.org],
- l'IDE Eclipse Indigo [http://www.eclipse.org/] sous sa forme customisée **SpringSource Tool Suite** (STS) [http://www.springsource.com/developer/sts],
- **PrimeFaces** dans sa version 3.2 disponible à l'URL [http://primefaces.org/],
- **PrimeFaces Mobile** dans sa version 0.9.2 disponible à l'URL [http://primefaces.org/],
- **WampServer** pour le SGBD MySQL [http://www.wampserver.com/].

1.3.1 Installation de l'IDE Netbeans

Nous décrivons ici l'installation de Netbeans 7.1.2 disponible à l'URL [http://netbeans.org/downloads/].

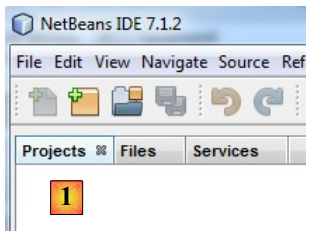


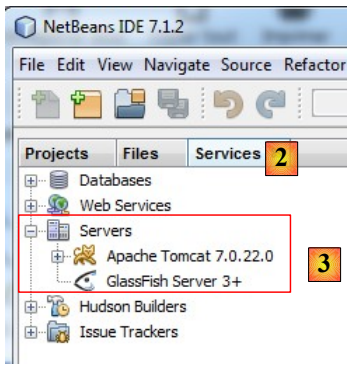
On pourra prendre ci-dessus, la version Java EE avec les deux serveurs Glassfish 3.1.2 et Apache Tomcat 7.0.22. Le premier permet de développer des applications Java EE avec des EJB3 (Enterprise Java Bean) et l'autre des applications Java EE sans EJB. Nous remplacerons alors les EJB par le framework Spring [http://www.springsource.com/].

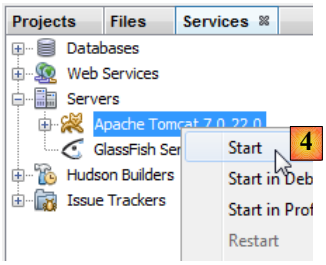
Une fois l'installateur de Netbeans téléchargé, on l'exécutera. Il faut pour cela avoir un JDK (Java Development Kit) [http://www.oracle.com/technetwork/java/javase/overview/index.HTML]. Outre Netbeans 7.1.2, on installera :

- le framework JUnit pour les tests unitaires,
- le serveur Glassfish 3.1.2,
- le serveur Tomcat 7.0.22.

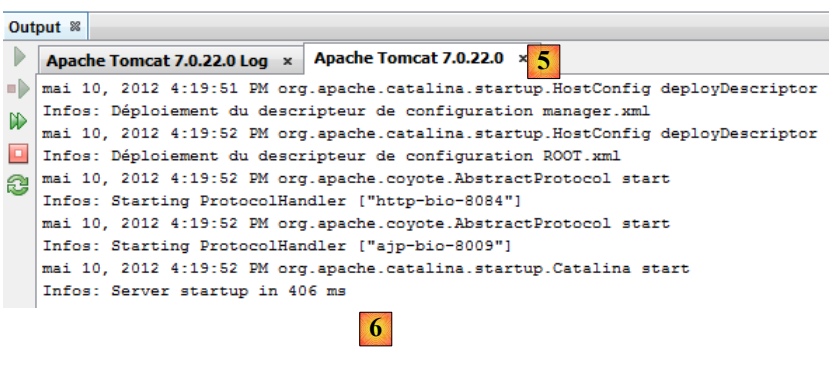
Une fois l'installation terminée, on lance Netbeans :

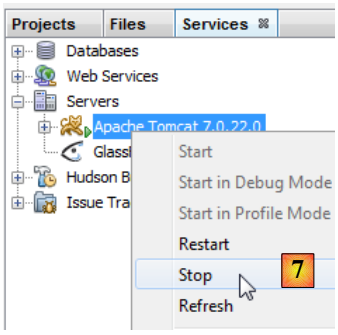






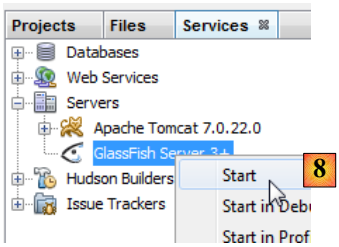
- en [1], Netbeans 7.1.2 avec trois onglets :
  - l'onglet [Projects] visualise les projets Netbeans ouverts ;
  - l'onglet [Files] visualise les différents fichiers qui composent les projets Netbeans ouverts ;
  - l'onglet [Services] regroupe un certain nombre d'outils utilisés par les projets Netbeans,
- en [2], on active l'onglet [Services] et en [3] la branche [Servers],
- en [4], on lance le serveur Tomcat pour vérifier sa bonne installation,

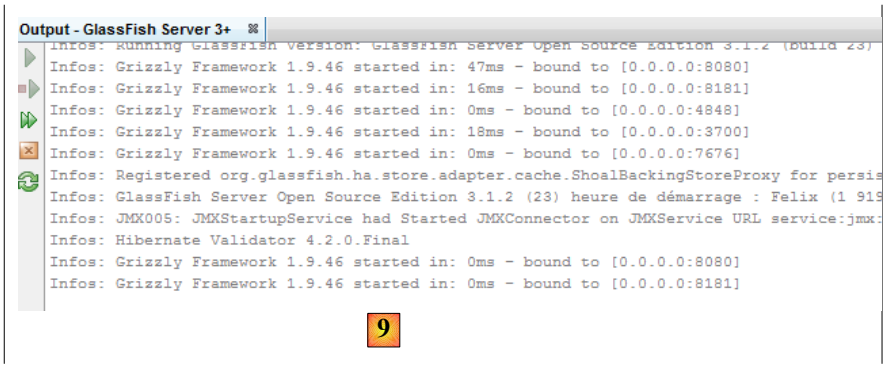




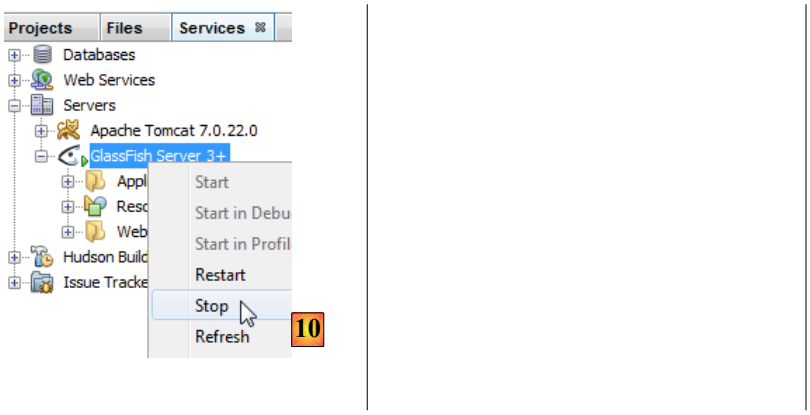
- en [6], dans l'onglet [Apache Tomcat] [5], on peut vérifier le lancement correct du serveur,
- en [7], on arrête le serveur Tomcat,

On procède de même pour vérifier la bonne installation du serveur Glassfish 3.1.2 :





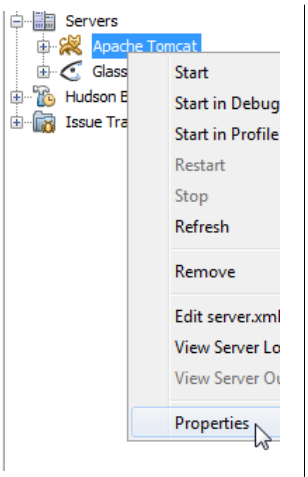
- en [8], on lance le serveur Glassfish,
- en [9] on vérifie qu'il s'est lancé correctement,

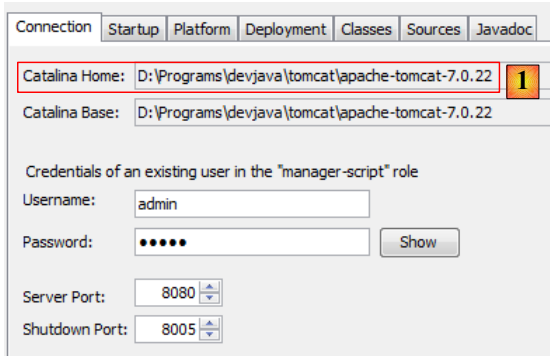


- en [10], on l'arrête.

Nous avons désormais un IDE opérationnel. Nous pouvons commencer à construire des projets. Selon ceux-ci nous serons amenés à détailler différentes caractéristiques de l'IDE.

Nous allons maintenant installer l'IDE Eclipse et nous allons au sein de celui-ci enregistrer les deux serveurs Tomcat 7 et Glassfish 3.1.2 que nous venons de télécharger. Pour cela, nous avons besoin de connaître les dossiers d'installation de ces deux serveurs. On trouve ces informations dans les propriétés de ces deux serveurs :





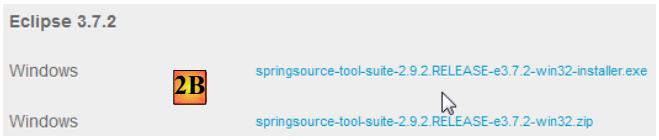
- en [1], le dossier d'installation de Tomcat 7,

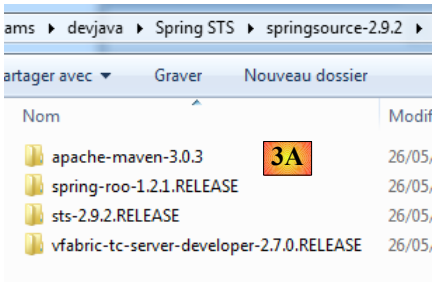
- en [2], le dossier des domaines du serveur Glassfish. Dans l'enregistrement de Glassfish dans Eclipse, il faudra donner seulement l'information suivante [D:\Programs\devjava\Glassfish\glassfish-3.1.2\glassfish].

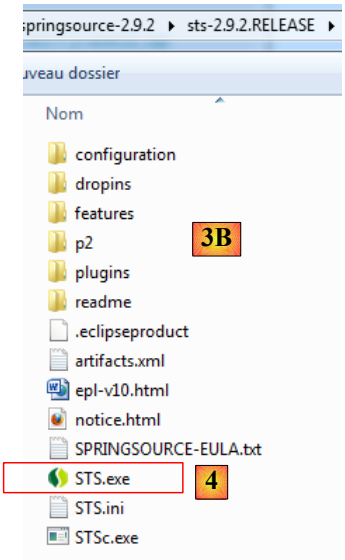
1.3.2 Installation de l'IDE Eclipse

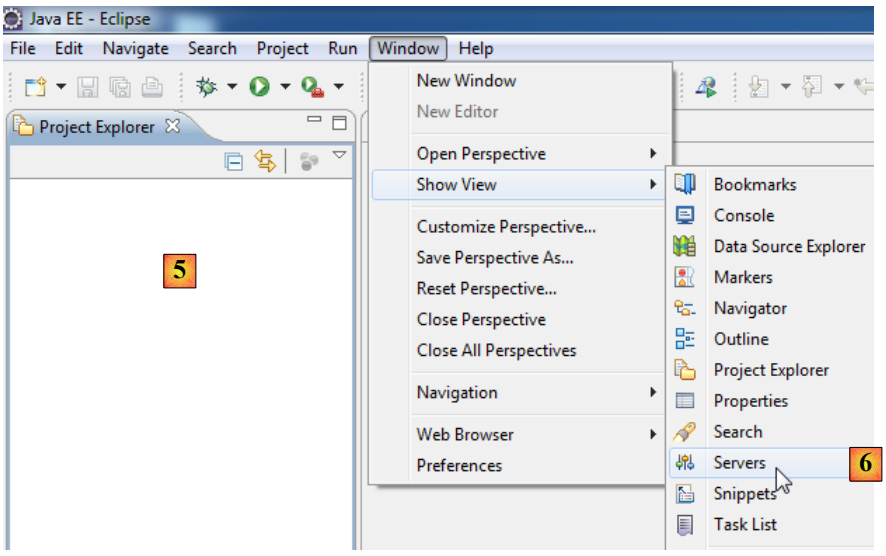
Bien qu'Eclipse ne soit pas l'IDE qui soit utilisé dans ce document, nous en présentons néanmoins l'installation. En effet, les exemples du document sont également livrés sous la forme de projets Maven pour Eclipse. Eclipse ne vient pas avec Maven pré-intégré. Il faut installer un plugin pour ce faire. Plutôt qu'installer un Eclipse brut, nous allons installer **SpringSource Tool Suite** [http://www.springsource.com/developer/sts], un Eclipse pré-équipé avec de nombreux plugins liés au framework Spring mais 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],

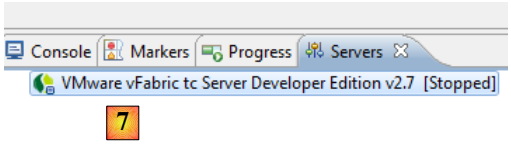




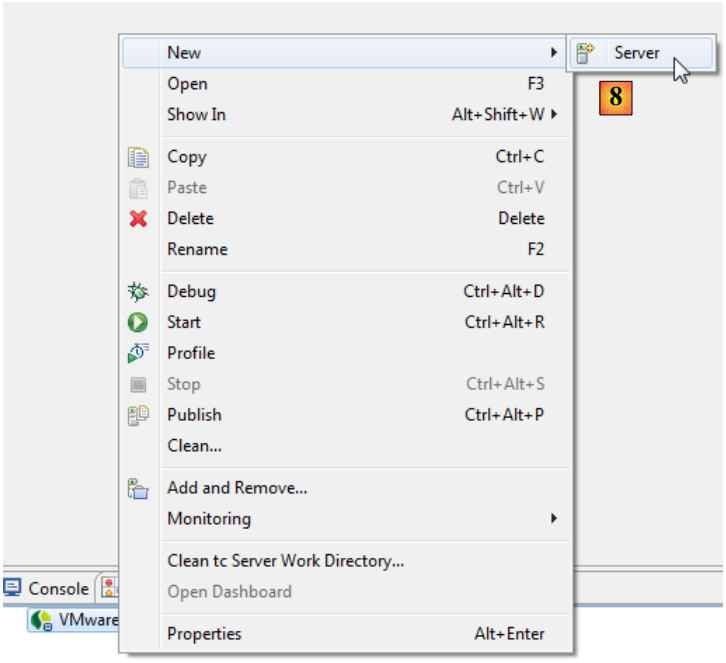




- le fichier téléchargé est un installateur qui crée l'arborescence de fichiers [3A] [3B]. En [4], on lance l'exécutable,
- en [5], la fenêtre de travail de l'IDE après avoir fermé la fenêtre de bienvenue. En [6], on fait afficher la fenêtre des serveurs d'applications,

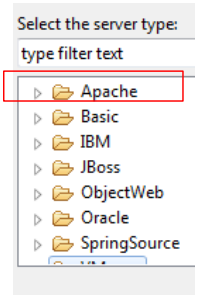


7

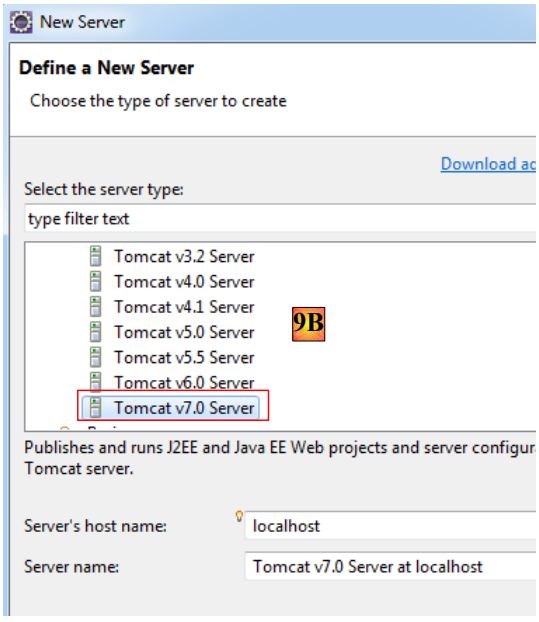


8

- en [7], la fenêtre des serveurs. Un serveur est enregistré. C'est un serveur VMware que nous n'utiliserons pas. En [8], on appelle l'assistant d'ajout d'un nouveau serveur,

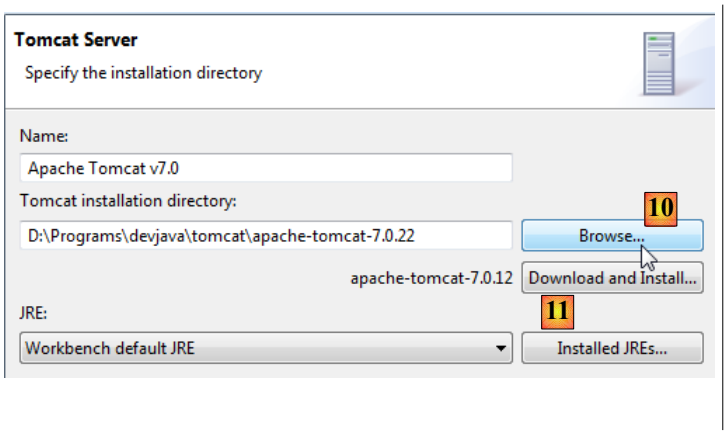


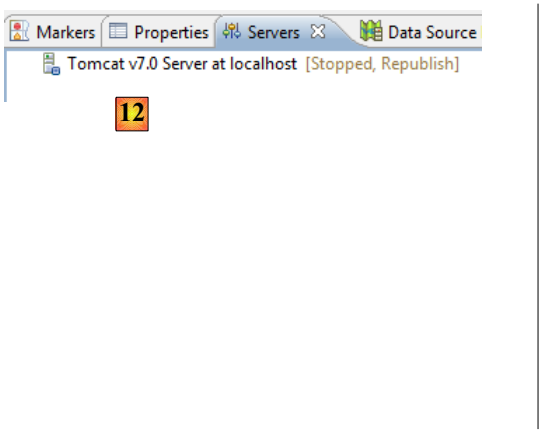
9A



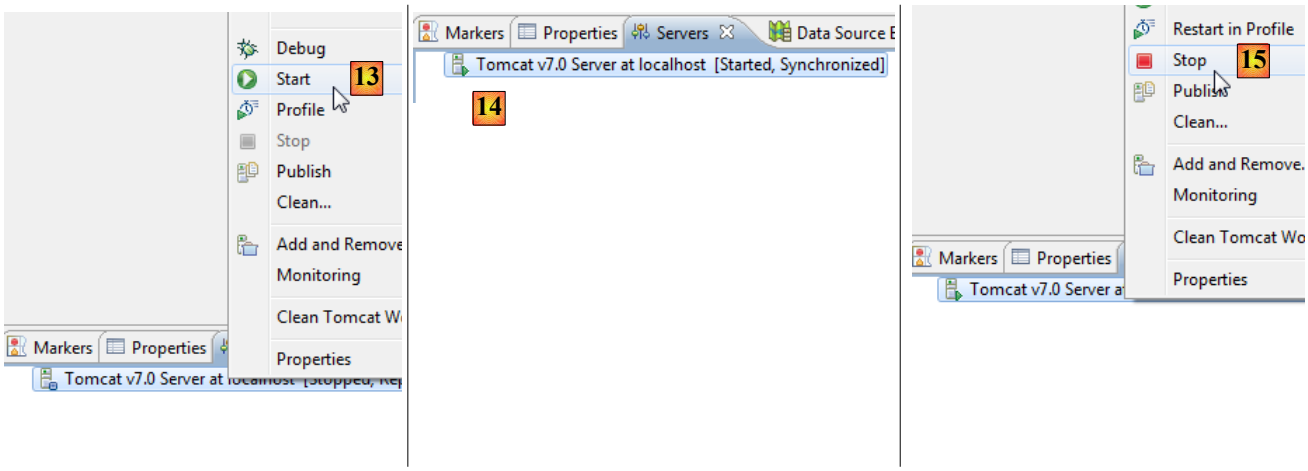
9B

- en [9A], divers serveurs nous sont proposés. Nous choisissons d'installer un serveur Tomcat 7 d'Apache [9B],





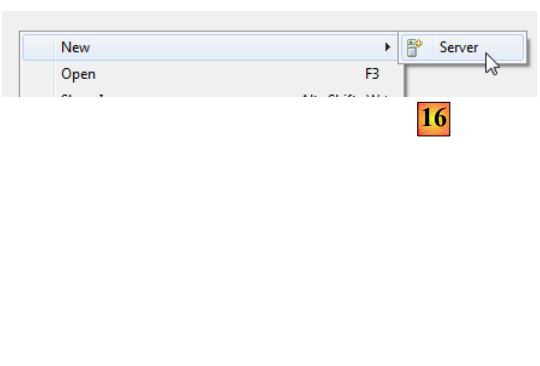
- en [10], nous désignons le dossier d'installation du serveur Tomcat 7 installé avec Netbeans. Si on n'a pas de serveur Tomcat, utiliser le bouton [11],
- en [12], le serveur Tomcat apparaît dans la fenêtre [Servers],



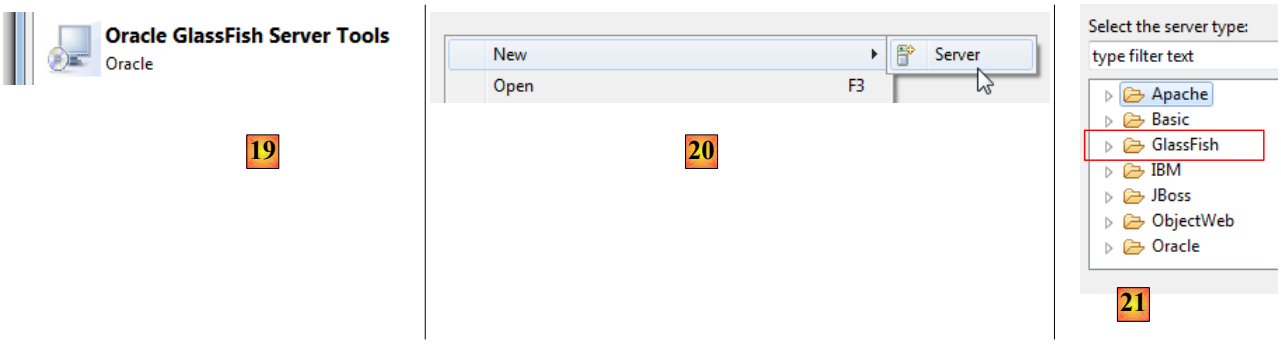
- en [13], nous lançons le serveur,
- en [14], il est lancé,
- en [15], on l'arrête.

Dans la fenêtre [Console], on obtient les logs suivants de Tomcat si tout va bien :

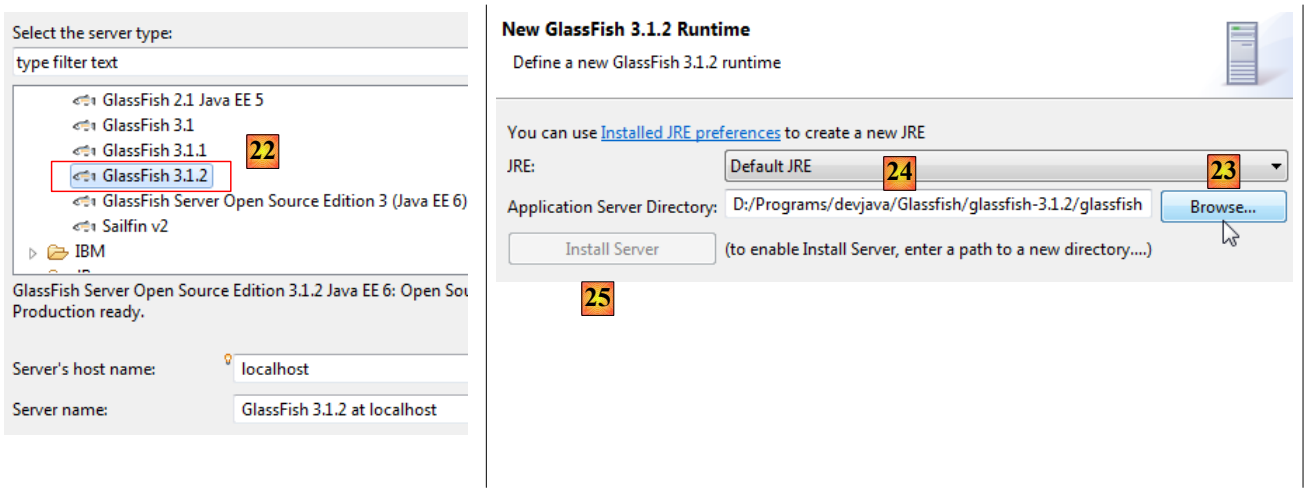
```
1.  mai 26, 2012 8:56:51 AM org.apache.catalina.core.AprLifecycleListener init
2.  Infos: The APR based Apache Tomcat Native library which allows optimal performance in production
    environments was not found on the java.library.path: ...
3.  mai 26, 2012 8:56:55 AM org.apache.coyote.AbstractProtocol init
4.  Infos: Initializing ProtocolHandler ["http-bio-8080"]
5.  mai 26, 2012 8:56:55 AM org.apache.coyote.AbstractProtocol init
6.  Infos: Initializing ProtocolHandler ["ajp-bio-8009"]
7.  mai 26, 2012 8:56:55 AM org.apache.catalina.startup.Catalina load
8.  Infos: Initialization processed in 4527 ms
9.  mai 26, 2012 8:56:55 AM org.apache.catalina.core.StandardService startInternal
10. Infos: Démarrage du service Catalina
11. mai 26, 2012 8:56:55 AM org.apache.catalina.core.StandardEngine startInternal
12. Infos: Starting Servlet Engine: Apache Tomcat/7.0.22
13. mai 26, 2012 8:56:57 AM org.apache.catalina.util.SessionIdGenerator createSecureRandom
14. Infos: Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [218]
    milliseconds.
15. mai 26, 2012 8:56:57 AM org.apache.coyote.AbstractProtocol start
16. Infos: Starting ProtocolHandler ["http-bio-8080"]
17. mai 26, 2012 8:56:57 AM org.apache.coyote.AbstractProtocol start
18. Infos: Starting ProtocolHandler ["ajp-bio-8009"]
19. mai 26, 2012 8:56:57 AM org.apache.catalina.startup.Catalina start
20. Infos: Server startup in 2252 ms
```



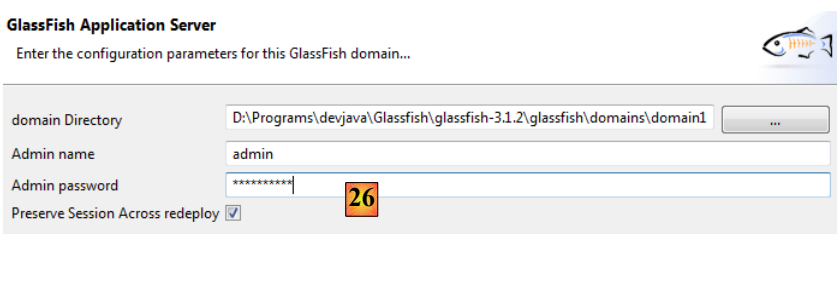
- toujours dans la fenêtre [Servers], on ajoute un nouveau serveur [16],
- en [17], le serveur Glassfish n'est pas proposé,
- dans ce cas, on utilise le lien [18],



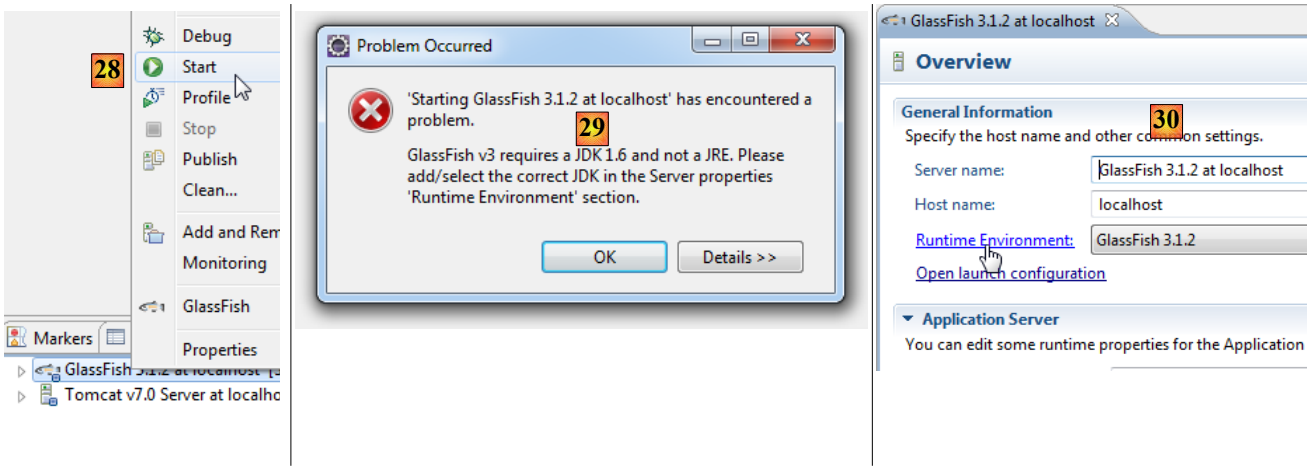
- en [19], on choisit d'ajouter un adaptateur pour le serveur Glassfish,
- celui est téléchargé et de retour dans la fenêtre [Servers], on ajoute un nouveau serveur,
- cette fois-ci en [21], les serveurs Glassfish sont proposés,



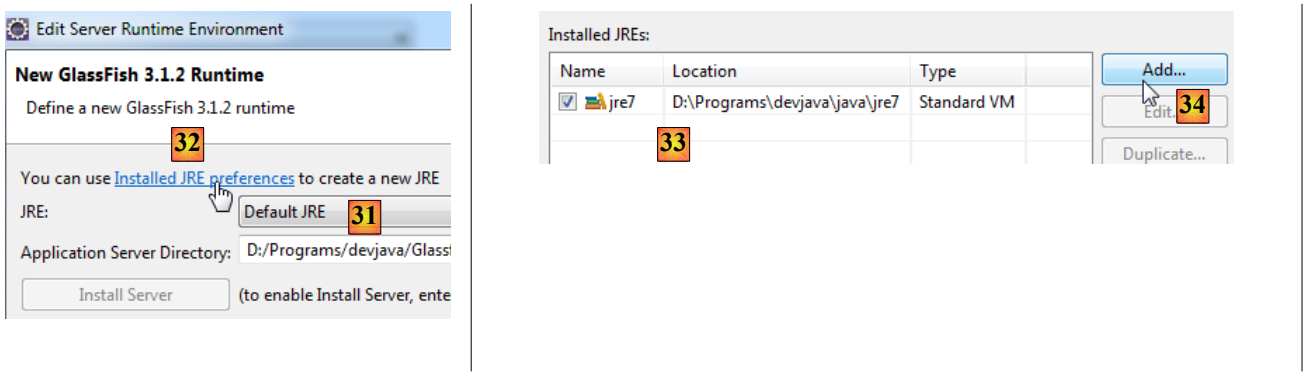
- en [22], on choisit le serveur Glassfish 3.1.2 téléchargé avec Netbeans,
- en [23], on désigne le dossier d'installation de Glassfish 3.1.2 (faire attention au chemin indiqué en [24]),
- si on n'a pas de serveur Glassfish, utiliser le bouton [25],



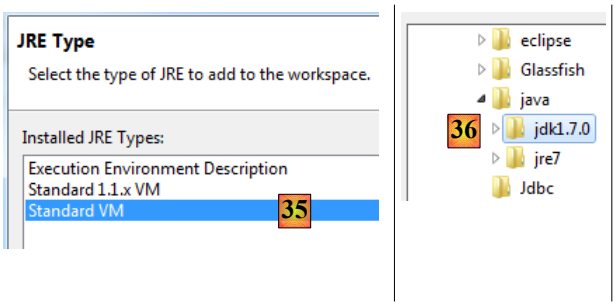
- en [26], l'assistant nous demande le mot de passe de l'administrateur du serveur Glassfish. Pour une première installation, c'est normalement *adminadmin*,
- lorsque l'assistant est terminé, dans la fenêtre [Servers], le serveur Glassfish apparaît [27],



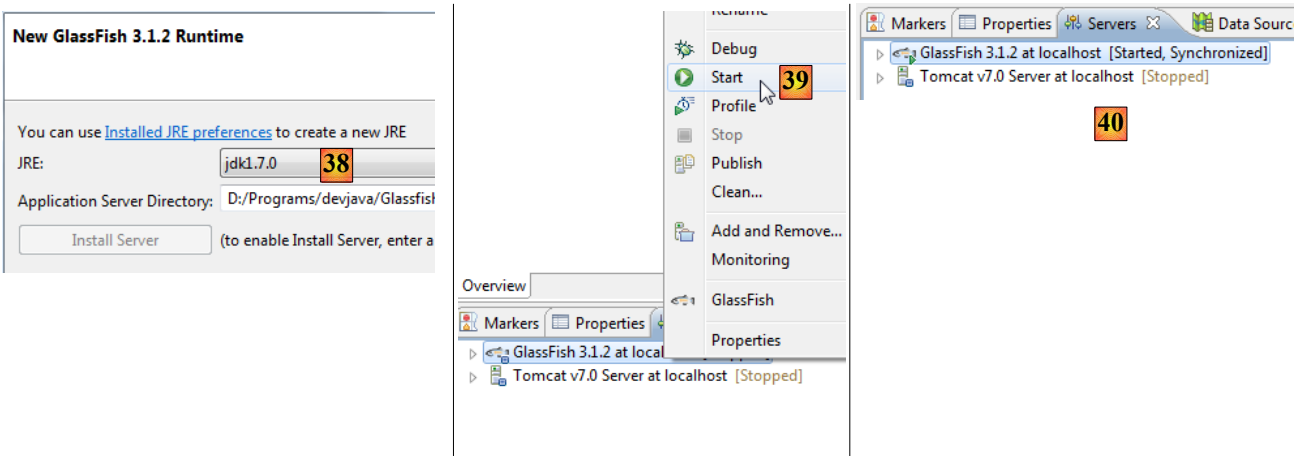
- en [28], on le lance,
- en [29] un problème peut survenir. Cela dépend des informations données à l'installation. Glassfish veut un JDK (Java Development Kit) et non pas un JRE (Java Runtime Environment),
- pour avoir accès aux propriétés du serveur Glassfish, on double-clique dessus dans la fenêtre [Servers],
- on obtient la fenêtre [20] dans laquelle on suit le lien [Runtime Environment],



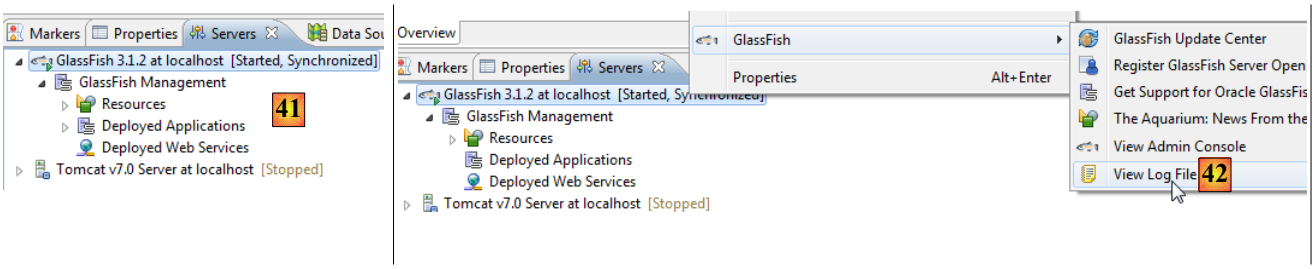
- en [31], on va remplacer le JRE utilisé par défaut par un JDK. Pour cela, on utilise le lien [32],
- en [33], les JRE installés sur la machine,
- en [34], on en ajoute un autre,



- en [35], on choisit [Standard VM] (Virtual Machine),
- en [36], on sélectionne d'un JDK (>=1.6),
- en [37], le nom donné au nouveau JRE,



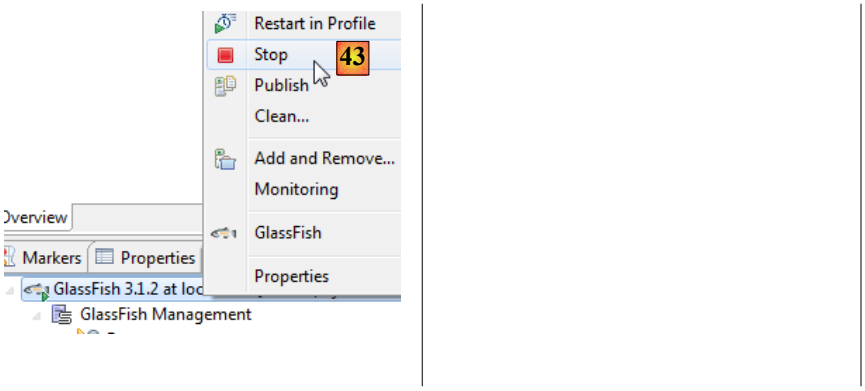
- revenu au choix du JRE pour Glassfish, nous choisissons le nouveau JRE déclaré [38],
- une fois l'assistant de configuration terminé, on relance [Glassfish] [39],
- en [40], il est lancé,



- en [41], l'arborescence du serveur,
- en [42], on accède aux logs de Glassfish :

```
1.  ...
2.  Infos: Running GlassFish Version: GlassFish Server Open Source Edition 3.1.2 (build 23)
3.
4.  Infos: Grizzly Framework 1.9.46 started in: 125ms - bound to [0.0.0.0:7676]
5.  Infos: Grizzly Framework 1.9.46 started in: 125ms - bound to [0.0.0.0:3700]
6.  Infos: Grizzly Framework 1.9.46 started in: 188ms - bound to [0.0.0.0:8080]
7.  Infos: Grizzly Framework 1.9.46 started in: 141ms - bound to [0.0.0.0:4848]
8.  Infos: Grizzly Framework 1.9.46 started in: 141ms - bound to [0.0.0.0:8181]
9.  Infos: Registered org.glassfish.ha.store.adapter.cache.ShoalBackingStoreProxy for persistence-type =
    replicated in BackingStoreFactoryRegistry
10.
11. Infos: GlassFish Server Open Source Edition 3.1.2 (23) heure de démarrage : Felix (13 790 ms), services de
    démarrage (2 028 ms), total (15 818 ms)
```

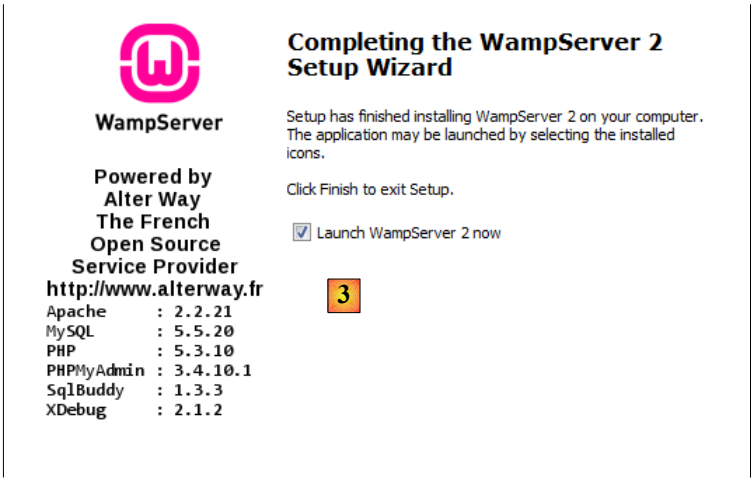
```
12.
13. Infos: JMX005: JMXStartupService had Started JMXConnector on JMXService URL
    service:jmx:rmi://Gportpers3.ad.univ-angers.fr:8686/jndi/rmi://Gportpers3.ad.univ-angers.fr:8686/jmxrmi
```



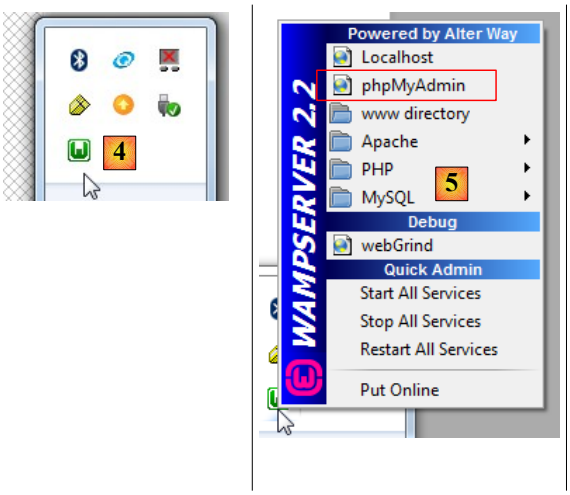
- en [43], nous arrêtons le serveur Glassfish.

1.3.3 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,

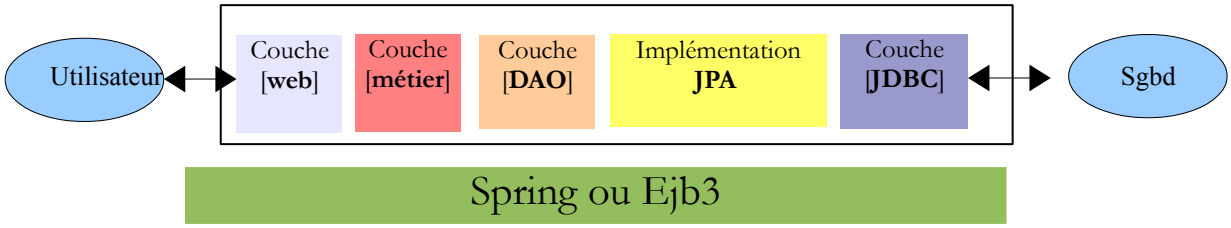
Nous donnerons peu de détails sur l'utilisation de [PhpMyAdmin]. Nous montrerons simplement comment l'utiliser pour créer la base de données de l'application exemple de ce tutoriel.

2 Java Server Faces

Nous présentons maintenant le framework **Java Server Faces**. Ce sera la version 2 qui sera utilisée mais les exemples présentent principalement des caractéristiques la version 1. Nous présenterons de la version 2, les seules caractéristiques nécessaires à l'application exemple qui suivra.

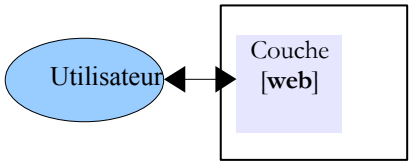
2.1 La place de JSF dans une application web

Tout d'abord, situons JSF dans le développement d'une application web. Le plus souvent, celle-ci sera bâtie sur une architecture multi-couche 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 JSF 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 [**jpa**] (Java Persistence Api) et le pilote JDBC gèrent l'accès aux données du Sgbd. Le couche [jpa] sert d'ORM (Object Relational Mapper). Elle fait un pont entre les objets manipulés par la couche [DAO] et les lignes et les colonnes des données d'une base de données relationnelle,
- l'intégration des couches peut être réalisée par un conteneur Spring ou EJB3 (Enterprise Java Bean).

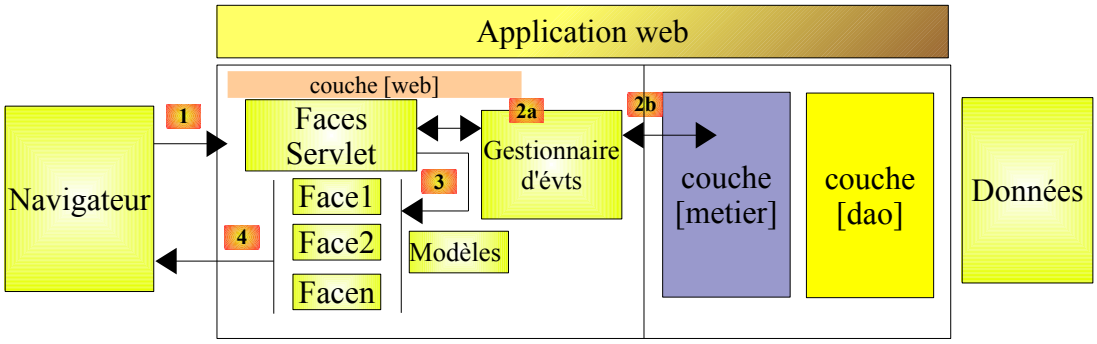
Les exemples donnés dans la suite pour illustrer JSF, n'utiliseront qu'une seule couche, la couche [web] :



Une fois les bases de JSF acquises, nous construirons des applications Java EE multi-couche.

2.2 Le modèle de développement MVC de JSF

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



Cette architecture implémente le Design Pattern **MVC** (Modèle, Vue, Contrôleur). Le traitement d'une demande d'un client se déroule selon les **quatre** étapes suivantes :

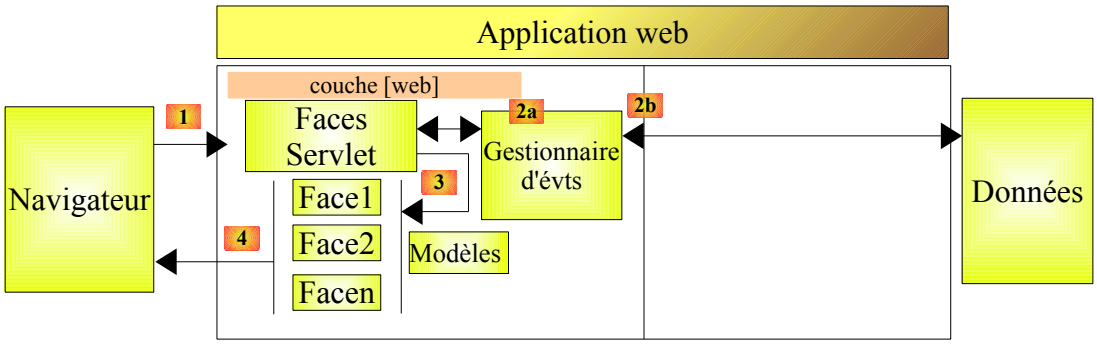
1. **demande** - le client navigateur fait une demande au contrôleur [**Faces Servlet**]. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC,
2. **traitement** - le contrôleur **C** traite cette demande. Pour ce faire, il se fait aider par des gestionnaires d'événements spécifiques à l'application écrite [2a]. Ces gestionnaires peuvent avoir besoin de l'aide 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'erreurs si la demande n'a pu être traitée correctement ;
    - une page de confirmation sinon,
3. **navigation** - le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :
    - choisir la Facelet qui va générer la réponse. C'est ce qu'on appelle la vue **V**, le **V** de MVC. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur ;
    - fournir à cette Facelet les données dont elle a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par le contrôleur. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC,

   L'étape 3 consiste donc en le choix d'une vue **V** et en la construction du modèle **M** nécessaire à celle-ci.
4. **réponse** - le contrôleur **C** demande à la Facelet choisie de s'afficher. Celle-ci utilise le modèle **M** préparé par le contrôleur **C** pour initialiser les parties dynamiques de la réponse qu'elle doit envoyer au client. La forme exacte de celle-ci peut être diverse : ce peut être un flux HTML, PDF, Excel, ...

Dans un projet JSF :

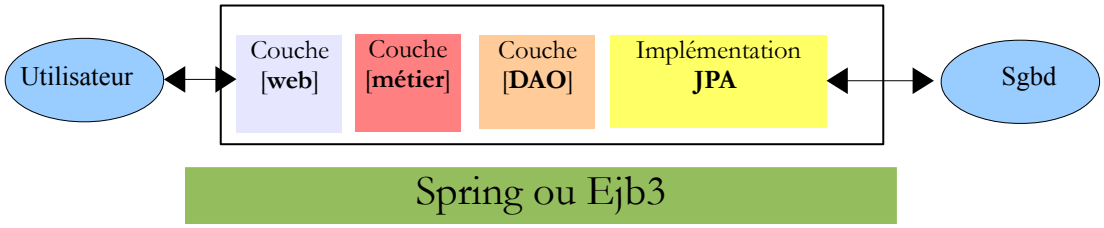
- le contrôleur **C** est la servlet [javax.faces.webapp.FacesServlet]. On trouve celle-ci dans la bibliothèque [javaee.jar],
- les vues **V** sont implémentées par des pages utilisant la technologie des Facelets,
- les modèles **M** et les **gestionnaires d'événements** sont implémentés par des classes Java souvent appelées "backing beans" ou plus simplement **beans**.

Maintenant, précisons le lien entre architecture web MVC et architecture en couches. Ce sont deux concepts différents qui sont parfois confondus. Prenons une application web JSF à une couche :



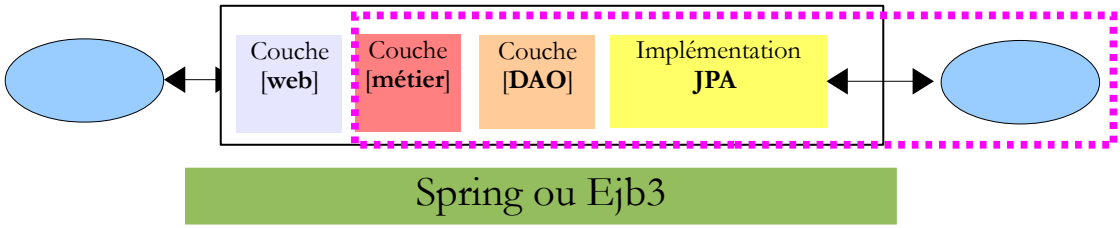
Si nous implémentons la couche [web] avec JSF, nous aurons bien une architecture web MVC mais pas une architecture multi-couche. Ici, la couche [web] s'occupera de tout : présentation, métier, accès aux données. Avec JSF, ce sont les beans qui feront ce travail.

Maintenant, considérons une architecture web multi-couche :



La couche [web] peut être implémentée sans framework et sans suivre le modèle MVC. On a bien alors une architecture multi-couche mais la couche web n'implémente pas le modèle 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 souvent donnée :



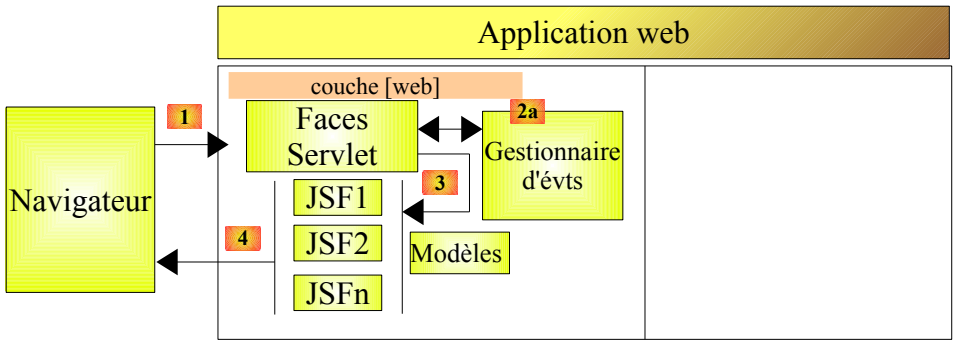
Beaucoup d'auteurs considèrent que ce qui est à droite de la couche [web] forme le modèle M du MVC. Pour éviter les ambigüités on parlera :

- 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.

2.3 Exemple mv-jsf2-01 : les éléments d'un projet JSF

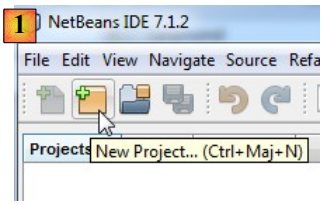
Les premiers exemples seront réduits à la seule couche web implémentée avec JSF 2 :

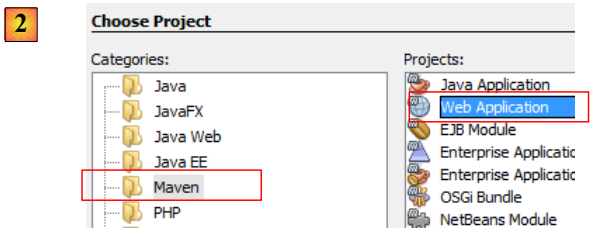


Lorsque les bases seront acquises, nous étudierons des exemples plus complexes avec des architectures multi-couche.

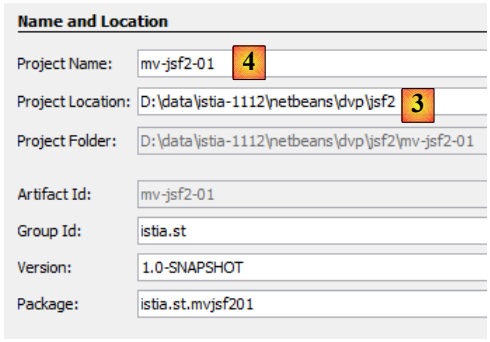
2.3.1 Génération du projet

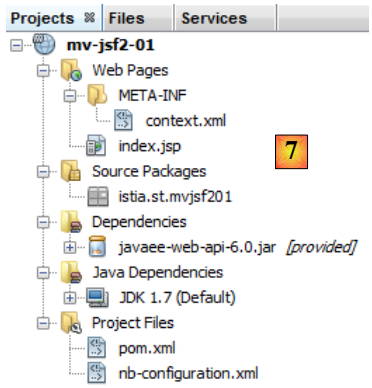
Nous générons notre premier projet JSF2 avec Netbeans 7.





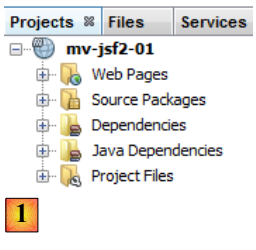
- en [1], créer un nouveau projet,
- en [2], choisir la catégorie [Maven] et le type de projet [Web Application],





- en [3], désigner le dossier parent du dossier du nouveau projet,
- en [4], donner un nom au projet,
- en [5], choisir un serveur. Avec Netbeans 7, on a le choix entre les serveurs Apache Tomcat et Glassfish. La différence entre les deux est que Glassfish supporte les EJB (Enterprise Java Bean) et Tomcat non. Nos exemples JSF ne vont pas utiliser d'EJB. Donc ici, on peut choisir n'importe quel serveur,
- en [6], on choisit la version Java EE 6 Web,
- en [7], le projet généré.

Examinons les éléments du projet et explicitons le rôle de chacun.



- en [1] : les différentes branches du projet :
  - [Web Pages] : contiendra les pages web (.xhtml, .jsp, .html), les ressources (images, documents divers), la configuration de la couche web ainsi que celle du framework JSF ;
  - [Source packages] : les classes Java du projet ;
  - [Dependencies] : les archives .jar nécessaires au projet et gérées par le framework Maven ;
  - [Java Dependencies] : les archives .jar nécessaires au projet et non gérées par le framework Maven ;
  - [Project Files] : fichier de configuration de Maven et Netbeans,

- en [2] : la branche [Web Pages],

Elle contient la page [index.jsp] suivante :

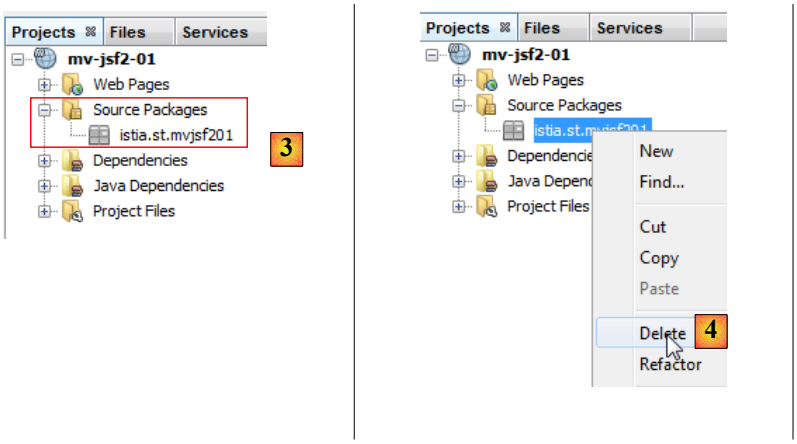
```
1.  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
3.     "http://www.w3.org/TR/HTML4/loose.dtd">
4.
5.  <html>
6.      <head>
7.          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8.          <title>JSP Page</title>
9.      </head>
10.     <body>
11.         <h1>Hello World!</h1>
12.     </body>
13. </html>
```

C'est une page web qui affiche la chaîne de caractères 'Hello World' en gros caractères.

Le fichier [META-INF/context.xml] est le suivant :

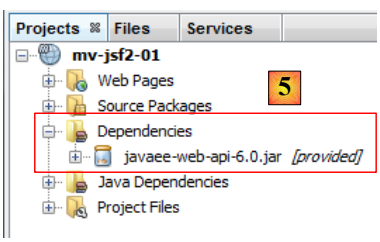
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <Context antiJARLocking="true" path="/mv-jsf2-01"/>
```

La ligne 2 indique que le contexte de l'application (ou son nom) est /mv-jsf2-01. Cela signifie que les pages web du projet seront demandées via une URL de la forme *http://machine:port/mv-jsf2-01/page*. Le contexte est par défaut le nom du projet. Nous n'aurons pas à modifier ce fichier.



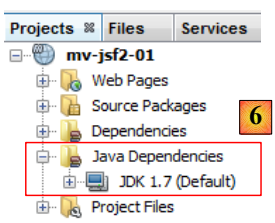
- en [3], la branche [Source Packages],

Cette branche contient les codes source des classes Java du projet. Ici nous n'avons aucune classe. Netbeans a généré un package par défaut qui peut être supprimé [4].

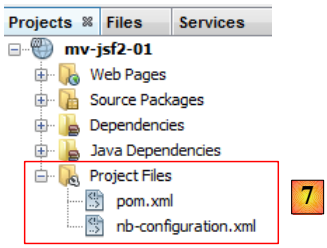


- en [5], la branche [Dependencies],

Cette branche affiche toute les bibliothèques nécessaires au projet et gérées par Maven. Toutes les bibliothèques listées ici vont être automatiquement téléchargées par Maven. C'est pourquoi un projet Maven a besoin d'un accès Internet. Les bibliothèques téléchargées vont être stockées en local. Si un autre projet a besoin d'une bibliothèque déjà présente en local, celle-ci ne sera alors pas téléchargée. Nous verrons que cette liste de bibliothèques ainsi que les dépôts où on peut les trouver sont définis dans le fichier de configuration du projet Maven.



- en [6], les bibliothèques nécessaires au projet et non gérées par Maven,



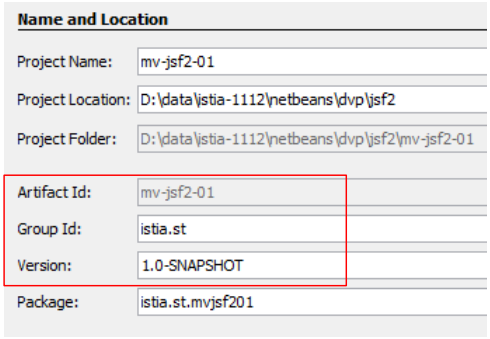
- en [7], les fichiers de configuration du projet Maven :
  - [nb-configuration.xml] est le fichier de configuration de Netbeans. Nous ne nous y intéresserons pas.
  - [pom.xml] : le fichier de configuration de Maven. POM signifie **P**roject **O**bject **M**odel. On sera parfois amené à intervenir directement sur ce fichier.

Le fichier [pom.xml] généré 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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-jsf2-01</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>war</packaging>
9.
10.   <name>mv-jsf2-01</name>
11.
12.   <properties>
13.     <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
```

```
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.   <dependencies>
18.     <dependency>
19.       <groupId>javax</groupId>
20.       <artifactId>javaee-web-api</artifactId>
21.       <version>6.0</version>
22.       <scope>provided</scope>
23.     </dependency>
24.   </dependencies>
25.
26.   <build>
27.     <plugins>
28.       <plugin>
29.         <groupId>org.apache.maven.plugins</groupId>
30.         <artifactId>maven-compiler-plugin</artifactId>
31.         <version>2.3.2</version>
32.         <configuration>
33.           <source>1.6</source>
34.           <target>1.6</target>
35.           <compilerArguments>
36.             <endorseddirs>${endorsed.dir}</endorseddirs>
37.           </compilerArguments>
38.         </configuration>
39.       </plugin>
40.       <plugin>
41.         <groupId>org.apache.maven.plugins</groupId>
42.         <artifactId>maven-war-plugin</artifactId>
43.         <version>2.1.1</version>
44.         <configuration>
45.           <failOnMissingWebXml>false</failOnMissingWebXml>
46.         </configuration>
47.       </plugin>
48.       <plugin>
49.         <groupId>org.apache.maven.plugins</groupId>
50.         <artifactId>maven-dependency-plugin</artifactId>
51.         <version>2.1</version>
52.         <executions>
53.           <execution>
54.             <phase>validate</phase>
55.             <goals>
56.               <goal>copy</goal>
57.             </goals>
58.             <configuration>
59.               <outputDirectory>${endorsed.dir}</outputDirectory>
60.               <silent>true</silent>
61.               <artifactItems>
62.                 <artifactItem>
63.                   <groupId>javax</groupId>
64.                   <artifactId>javaee-endorsed-api</artifactId>
65.                   <version>6.0</version>
66.                   <type>jar</type>
67.                 </artifactItem>
68.               </artifactItems>
69.             </configuration>
70.           </execution>
71.         </executions>
72.       </plugin>
73.     </plugins>
74.   </build>
75.
76. </project>
```

- les lignes 5-8 définissent l'objet (artifact) Java qui va être créé par le projet Maven. Ces informations proviennent de l'assistant qui a été utilisé lors de la création du projet :



Un objet Maven est défini par quatre propriétés :

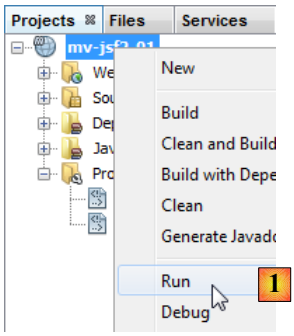
- [**groupId**] : une information qui ressemble à un nom de package. Ainsi les bibliothèques du framework Spring ont groupId=*org.springframework*, celles du framework JSF ont groupId=*javax.faces,*
- [**artifactId**] : le nom de l'objet Maven. Dans le groupe [org.springframework] on trouve ainsi les artifactId suivants : *spring-context*, *spring-core*, *spring-beans*, ... Dans le groupe [javax.faces], on trouve l'artifactId *jsf-api,*
- [**version**] : n° de version de l'artifact Maven. Ainsi l'artifact *org.springframework.spring-core* a les versions suivantes : 2.5.4, 2.5.5, 2.5.6, 2.5.6.SECO1, ...
- [**packaging**] : la forme prise par l'artifact, le plus souvent *war* ou *jar*.

Notre projet Maven va donc générer un [war] (ligne 8) dans le groupe [istia.st] (ligne 5), nommé [mv-jsf2-01] (ligne 6) et de version [1.0-SNAPSHOT] (ligne 7). Ces quatre informations doivent définir de façon unique un artifact Maven.

Les lignes 17-24 listent les dépendances du projet Maven, c'est à dire la liste des bibliothèques nécessaires au projet. Chaque bibliothèque est définie par les quatre informations (groupId, artifactId, version, packaging). Lorsque l'information *packaging* est absente comme ici, le *packaging* **jar** est utilisé. On y ajoute une autre information, **scope** qui fixe à quels moments de la vie du projet on a besoin de la bibliothèque. La valeur par défaut est **compile** qui indique que la bibliothèque est nécessaire à la compilation et à l'exécution. La valeur **provided** signifie que la bibliothèque est nécessaire lors de la compilation mais pas lors de l'exécution. Ici à l'exécution, elle sera fournie par le serveur Tomcat 7.

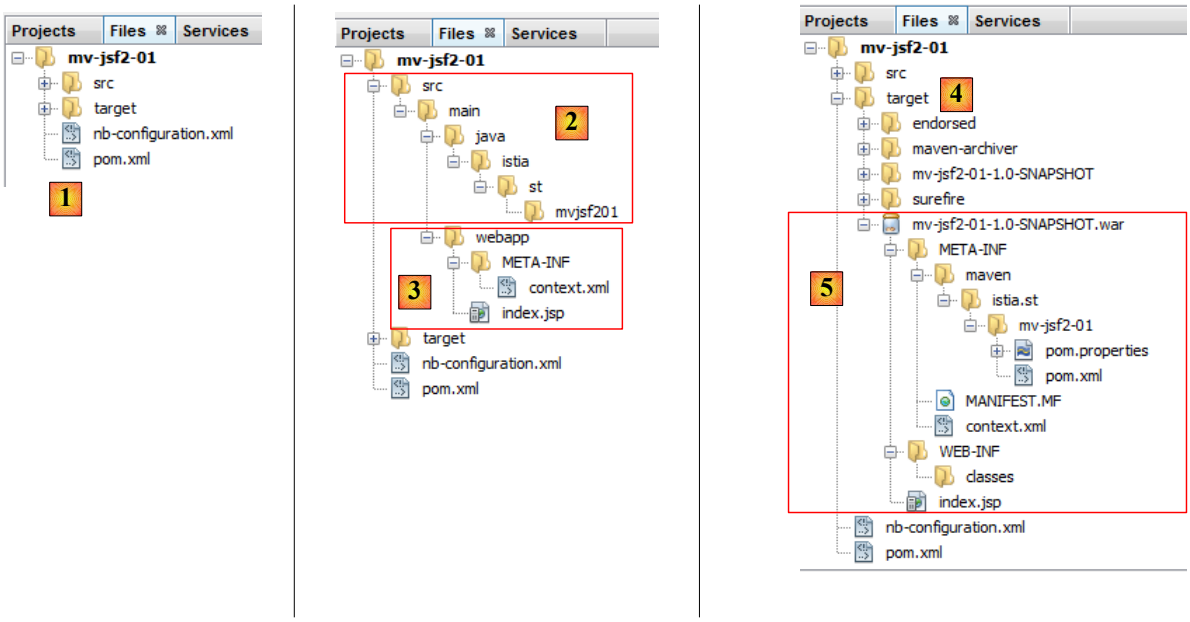
2.3.2 Exécution du projet

Nous exécutons le projet :



En [1], le projet Maven est exécuté. Le serveur Tomcat est alors lancé s'il ne l'était pas déjà. Un navigateur est lancé également et l'URL du contexte du projet est demandée [2]. Comme aucun document n'est demandé, la page *index.html*, *index.jsp*, *index.xhtml* est alors utilisée si elle existe. Ici, ce sera la page [index.jsp].

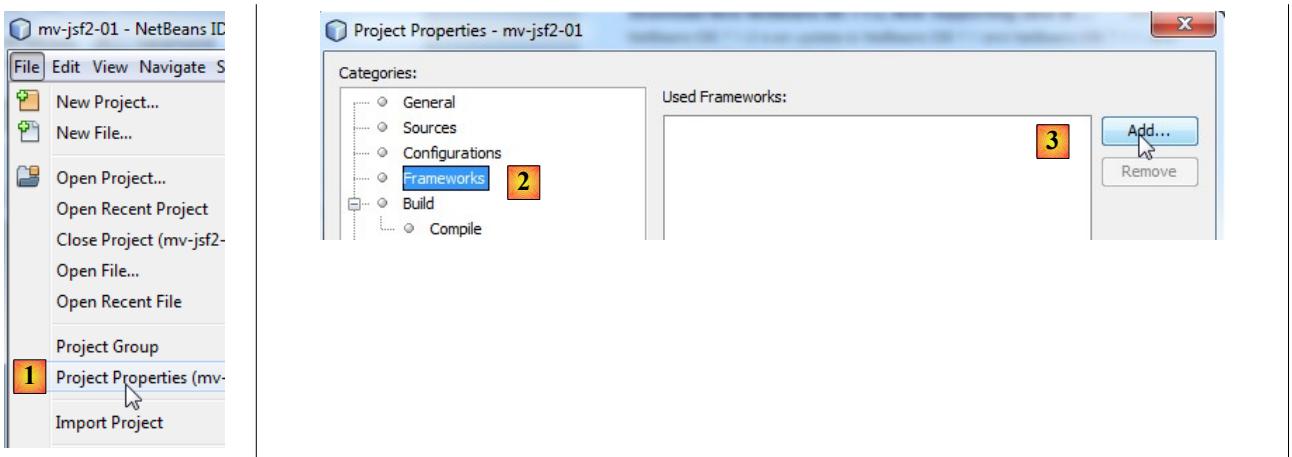
2.3.3 Le système de fichiers d'un projet Maven



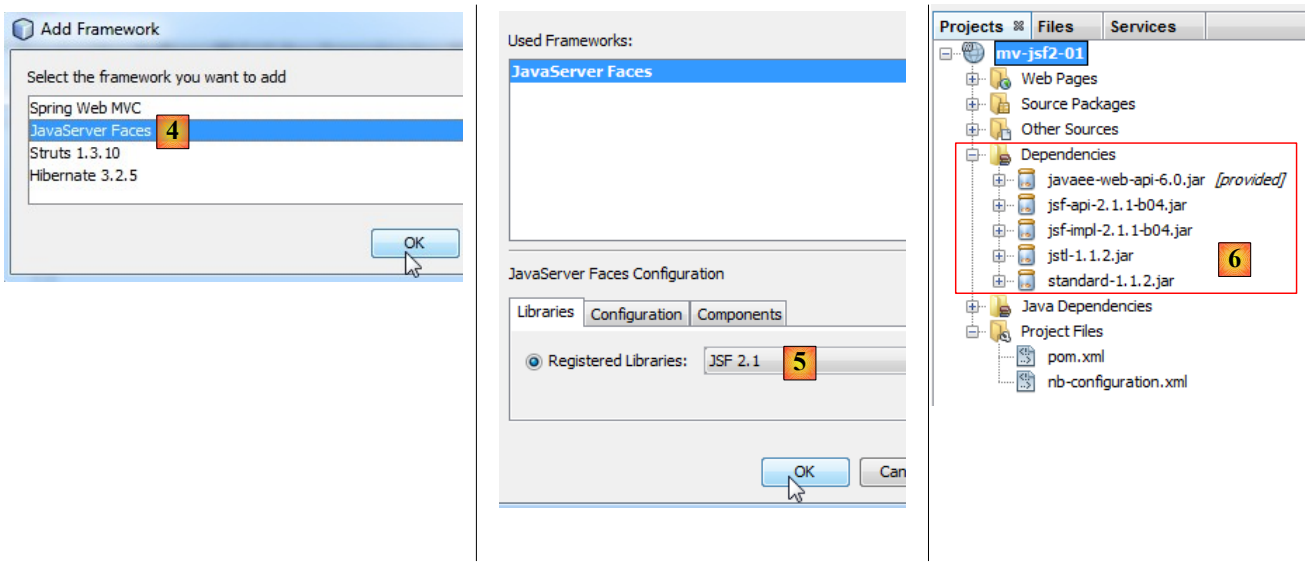
- [1] : le système de fichiers du projet est dans l'onglet [Files],
- [2] : les sources Java sont dans le dossier [src / main / java],
- [3] : les pages web sont dans le dossier [src / main / webapp],
- [4] : le dossier [target] est créé par la construction (build) du projet,
- [5] : ici, la construction du projet a créé une archive [mv-jsf2-01-1.0-SNAPSHOT.war]. C'est cette archive qui a été exécutée par le serveur Tomcat.

2.3.4 Configurer un projet pour JSF

Notre projet actuel n'est pas un projet JSF. Il lui manque les bibliothèques du framework JSF. Pour faire du projet courant, un projet JSF on procède de la façon suivante :



- en [1], on accède aux propriétés du projet,
- en [2], on choisit la catégorie [Frameworks],
- en [3], on ajoute un framework,



- en [4], on choisit Java Server Faces,
- en [5], Netbeans nous propose la version 2.1 du framework. On l'accepte,
- en [6], le projet s'enrichit alors de nouvelles dépendances.

Le fichier [pom.xml] a évolué pour refléter cette nouvelle configuration :

```
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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-jsf2-01</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>war</packaging>
9.
10.   <name>mv-jsf2-01</name>
11.
12.   ...
13.   <dependencies>
14.     <dependency>
15.       <groupId>com.sun.faces</groupId>
16.       <artifactId>jsf-api</artifactId>
17.       <version>2.1.1-b04</version>
18.     </dependency>
19.     <dependency>
20.       <groupId>com.sun.faces</groupId>
21.       <artifactId>jsf-impl</artifactId>
22.       <version>2.1.1-b04</version>
23.     </dependency>
24.     <dependency>
25.       <groupId>javax.servlet</groupId>
26.       <artifactId>jstl</artifactId>
27.       <version>1.1.2</version>
28.     </dependency>
29.     <dependency>
30.       <groupId>taglibs</groupId>
31.       <artifactId>standard</artifactId>
32.       <version>1.1.2</version>
33.     </dependency>
34.     <dependency>
35.       <groupId>javax</groupId>
36.       <artifactId>javaee-web-api</artifactId>
37.       <version>6.0</version>
38.       <scope>provided</scope>
39.     </dependency>
40.   </dependencies>
41.
42.   <build>
43.     ...
44.   </build>
```

```
45.   <repositories>
46.     <repository>
47.       <URL>http://download.java.net/maven/2/</URL>
48.       <id>jsf20</id>
49.       <layout>default</layout>
50.       <name>Repository for library Library[jsf20]</name>
51.     </repository>
52.     <repository>
53.       <URL>http://repo1.maven.org/maven2/</URL>
54.       <id>jstl11</id>
55.       <layout>default</layout>
56.       <name>Repository for library Library[jstl11]</name>
57.     </repository>
58.   </repositories>
59. </project>
```

Lignes 14-33, de nouvelles dépendances ont été ajoutées. Maven les télécharge automatiquement. Il va les chercher dans ce qu'on appelle des dépôts. Le dépôt central (Central Repository) est automatiquement utilisé. On peut ajouter d'autres dépôts grâce à la balise <**repository**>. Ici deux dépôts ont été ajoutés :

- lignes 46-51 : un dépôt pour la bibliothèque JSF 2,
- lignes 52-57 : un dépôt pour la bibliothèque JSTL 1.1.

Le projet s'est également enrichi d'une nouvelle page web :

La page [index.HTML] est la suivante :

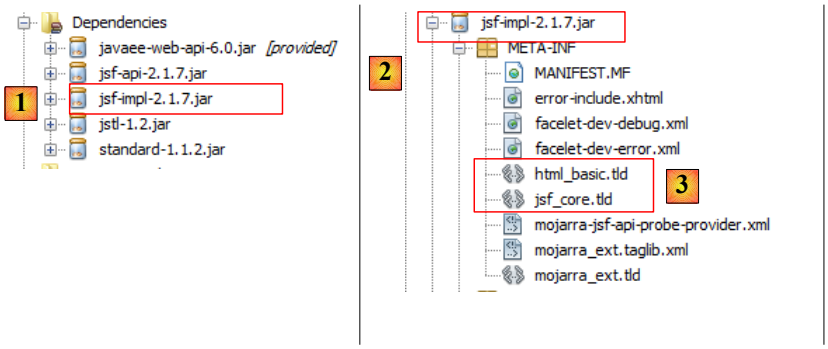
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <h:head>
6.      <title>Facelet Title</title>
7.    </h:head>
8.    <h:body>
9.      Hello from Facelets
10.   </h:body>
11. </html>
```

On a là un fichier XML (ligne 1). On y retrouve les balises du HTML mais au format XML. On appelle cela du XHTML. La technologie utilisée pour créer des pages web avec JSF 2 s'appelle **Facelets**. Aussi appelle-t-on parfois la page XHTML une page Facelet.

Les lignes 3-4 définissent la balise <html> avec des espaces de noms XML (xmlns=XML Name Space).

- la ligne 3 définit l'espace de noms principal `http://www.w3.org/1999/xhtml`,
- la ligne 4 définit l'espace de noms `http://java.sun.com/jsf/html` des balises HTML. Celles-ci seront préfixées par **h:** comme indiqué par `xmlns:h`. On trouve ces balises aux lignes 5, 7, 8 et 10.

A la rencontre de la déclaration d'un espace de noms, le serveur web va explorer les dossiers [META-INF] du *Classpath* de l'application, à la recherche de fichiers avec le suffixe **.tld** (TagLib Definition). Ici, il va les trouver dans l'archive [jsf-impl.jar] [1,2] :



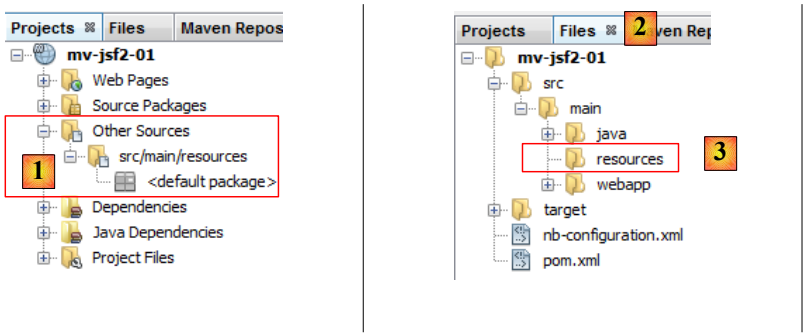
Examinons [3] le fichier [HTML_basic.tld] :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.
3.  <taglib xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd" version="2.1">
4.
5.  <!-- ============== Tag Library Description Elements ============= -->
6.
7.      <description>
8.          This tag library contains JavaServer Faces component tags for all
9.          UIComponent + HTML RenderKit Renderer combinations defined in the
10.         JavaServer Faces Specification.
11.     </description>
12.     <tlib-version>
13.         2.1
14.     </tlib-version>
15.     <short-name>
16.         h
17.     </short-name>
18.     <uri>
19.         http://java.sun.com/jsf/html
20.     </uri>
21.
22. <!-- ============== Tag Library Validator ============= -->
23. ...
```

- en ligne 19, **l'uri** de la bibliothèque de balises,
- en ligne 16, son nom court.

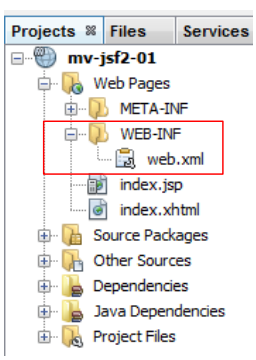
Les définitions des différentes balises **<h:xx>** sont trouvées dans ce fichier. Ces balises sont gérées par des classes Java qu'on trouve également dans l'artifact [jsf-impl.jar].

Revenons à notre projet JSF. Il s'est enrichi d'une nouvelle branche :



La branche [Other Sources] [1] contient les fichiers qui doivent être dans le Classpath du projet et qui ne sont pas du code Java. Ce sera le cas des fichiers de messages en JSF. Nous avons vu que sans l'ajout du framework JSF au projet, cette branche est absente. Pour la créer, il suffit de créer le dossier [src / main / resources] [3] dans l'onglet [Files] [2].

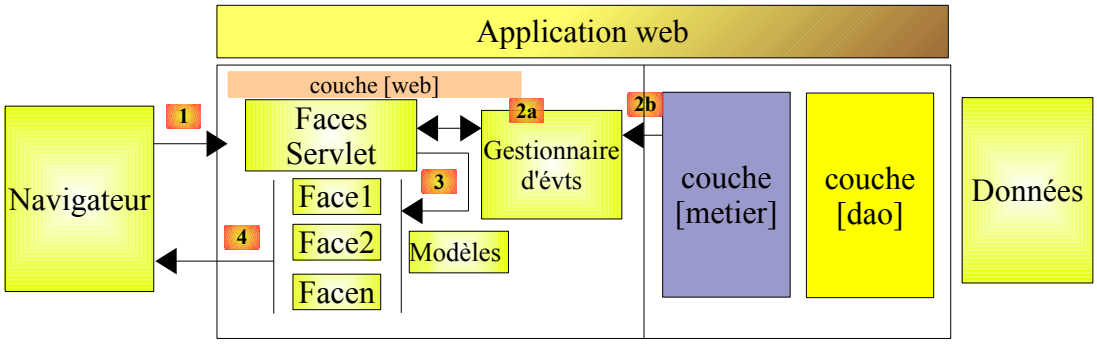
Enfin, un nouveau dossier est apparu dans la branche [Web Pages] :



Le dossier [WEB-INF] a été créé avec dedans le fichier [web.xml]. Celui-ci configure l'application web :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.      <context-param>
4.          <param-name>javax.faces.PROJECT_STAGE</param-name>
5.          <param-value>Development</param-value>
6.      </context-param>
7.      <servlet>
8.          <servlet-name>Faces Servlet</servlet-name>
9.          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.         <load-on-startup>1</load-on-startup>
11.     </servlet>
12.     <servlet-mapping>
13.         <servlet-name>Faces Servlet</servlet-name>
14.         <URL-pattern>/faces/*</URL-pattern>
15.     </servlet-mapping>
16.     <session-config>
17.         <session-timeout>
18.             30
19.         </session-timeout>
20.     </session-config>
21.     <welcome-file-list>
22.         <welcome-file>faces/index.xhtml</welcome-file>
23.     </welcome-file-list>
24. </web-app>
```

- les lignes 7-10 définissent une **servlet**, c.a.d. une classe Java capable de traiter les demandes des clients. Une application JSF fonctionne de la façon suivante :



Cette architecture implémente le Design Pattern **MVC** (Modèle, Vue, Contrôleur). Nous rappelons ce qui a déjà été écrit plus haut. Le traitement d'une demande d'un client se déroule selon les **quatre** étapes suivantes :

1 - demande - le client navigateur fait une demande au contrôleur [**Faces Servlet**]. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC,
2 - traitement - le contrôleur **C** traite cette demande. Pour ce faire, il se fait aider par des gestionnaires d'événements spécifiques à l'application écrite [2a]. Ces gestionnaires peuvent avoir besoin de l'aide 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'erreurs si la demande n'a pu être traitée correctement ;
- une page de confirmation sinon,

3 - navigation - le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :

- choisir la Facelet qui va générer la réponse. C'est ce qu'on appelle la vue **V**, le **V** de MVC. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur ;
- fournir à cette Facelet les données dont elle a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par le contrôleur. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC,

L'étape 3 consiste donc en le choix d'une vue **V** et en la construction du modèle **M** nécessaire à celle-ci.
4 - réponse - le contrôleur **C** demande à la Facelet choisie de s'afficher. Celle-ci utilise le modèle **M** préparé par le contrôleur **C** pour initialiser les parties dynamiques de la réponse qu'elle doit envoyer au client. La forme exacte de celle-ci peut être diverse : ce peut être un flux HTML, PDF, Excel, ...

Dans un projet JSF :

- le contrôleur **C** est la servlet [javax.faces.webapp.FacesServlet],
- les vues **V** sont implémentées par des pages utilisant la technologie des Facelets,
- les modèles **M** et les **gestionnaires d'événements** sont implémentés par des classes Java souvent appelées "backing beans" ou plus simplement Beans.

Revenons sur le contenu du fichier [web.xml) :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.      <context-param>
4.          <param-name>javax.faces.PROJECT_STAGE</param-name>
5.          <param-value>Development</param-value>
6.      </context-param>
7.      <servlet>
8.          <servlet-name>Faces Servlet</servlet-name>
9.          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.         <load-on-startup>1</load-on-startup>
11.     </servlet>
12.     <servlet-mapping>
13.         <servlet-name>Faces Servlet</servlet-name>
14.         <URL-pattern>/faces/*</URL-pattern>
15.     </servlet-mapping>
16.     <session-config>
17.         <session-timeout>
18.             30
19.         </session-timeout>
20.     </session-config>
21.     <welcome-file-list>
22.         <welcome-file>faces/index.xhtml</welcome-file>
23.     </welcome-file-list>
24. </web-app>
```

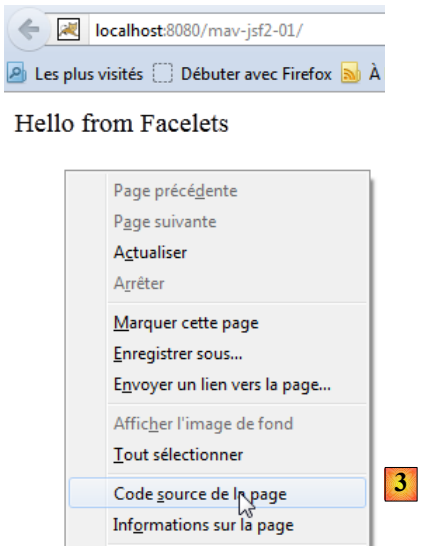
- lignes 12-15 : la balise <servlet-mapping> sert à associer une servlet à une URL demandée par le navigateur client. Ici, il est indiqué que les URL de la forme [/faces/*] doivent être traitées par la servlet de nom [Faces Servlet]. Celle-ci est définie lignes 7-10. Comme il n'y a pas d'autre balise <servlet-mapping> dans le fichier, cela signifie que la servlet [Faces Servlet] ne traitera que les URL de la forme [/faces/*]. Nous avons vu que le contexte de l'application s'appelait [/mv-jsf2-01]. Les URL des clients traitées par la servlet [Faces Servlet] auront donc la forme [http://machine:port/mv-jsf2-01/faces/*]. Les pages .html et .jsp seront traitées par défaut par le conteneur de servlets lui-même, et non par une servlet particulière. En effet, le conteneur de servlets sait comment les gérer,
- lignes 7-10 : définissent la servlet [Faces Servlet]. Comme toutes les URL acceptées sont dirigées vers elle, elle est le contrôleur C du modèle MVC,

- ligne 10 : indique que la servlet doit être chargée en mémoire dès le démarrage du serveur web. Par défaut, une servlet n'est chargée qu'à réception de la première demande qui lui est faite,
- lignes 3-6 : définissent un paramètre destiné à la servlet [Faces Servlet]. Le paramètre `javax.faces.PROJECT_STAGE` définit l'étape dans laquelle se trouve le projet exécuté. Au stade `Development` la servlet [Faces Servlet] fait afficher des messages d'erreur utiles au débogage. Au stade `Production` ces messages ne sont plus affichés,
- lignes 17-19 : durée en minutes d'une session. Un client dialogue avec l'application par une suite de **cycles demande / réponse**. Chaque cycle utilise une connexion TCP-IP qui lui est propre, nouvelle à chaque nouveau cycle. Aussi, si un client C fait deux demandes D1 et D2, le serveur S n'a pas les moyens de savoir que les deux demandes appartiennent au même client C. Le serveur S n'a pas la mémoire du client. C'est le protocole HTTP utilisé (HyperText Transport Protocol) qui veut ça : le client dialogue avec le serveur par une succession de cycles demande client / réponse serveur utilisant à chaque fois une nouvelle connexion TCP-IP. On parle de protocole **sans état**. Dans d'autres protocoles, comme par exemple FTP (File Transfer Protocol), le client C utilise la même connexion pendant la durée de son dialogue avec le serveur S. Une connexion est donc liée à un client particulier. Le serveur S sait toujours à qui il a affaire. Afin de pouvoir reconnaître qu'une demande appartient à un client donné, le serveur web peut utiliser la technique de la **session** :
  - lors de la première demande d'un client, le serveur S lui envoie la réponse attendue plus un **jeton**, une suite de caractères aléatoire, unique à ce client ;
  - lors de chaque demande suivante, le client C renvoie au serveur S le jeton qu'il a reçu, permettant ainsi au serveur S de le reconnaître.

  L'application a désormais la possibilité de demander au serveur de mémoriser des informations associées à un client donné. On parle de **session client**. La ligne 18 indique que la durée de vie d'une session est de 30 mn. Cela signifie que si un client C ne fait pas de nouvelle demande pendant 30 mn, sa session est détruite et les informations qu'elle contenait, perdues. Lors de sa prochaine demande, tout se passera comme s'il était un nouveau client et une nouvelle session démarrera,
- lignes 21-23 : la liste des pages à afficher lorsque l'utilisateur demande le contexte sans préciser de page, par exemple ici [http://machine:port/mv-jsf2-01]. Dans ce cas, le serveur web (pas la servlet) recherche si l'application a défini une balise <welcome-file-list>. Si oui, il affiche la première page trouvée dans la liste. Si elle n'existe pas, la deuxième page, et ainsi de suite jusqu'à trouver une page existante. Ici, lorsque le client demande l'URL [http://machine:port/mv-jsf2-01], c'est l'URL [http://machine:port/mv-jsf2-01/index.xhtml] qui lui sera servie.

2.3.5 Exécuter le projet

Lorsqu'on exécute le nouveau projet, le résultat obtenu dans le navigateur est suivant :



- en [1], le contexte a été demandé sans précision de document,
- en [2], comme il a été expliqué, c'est alors la page d'accueil (welcome-file) [index.xhtml] qui est servie.

On peut avoir la curiosité de regarder le code source reçu [3] :

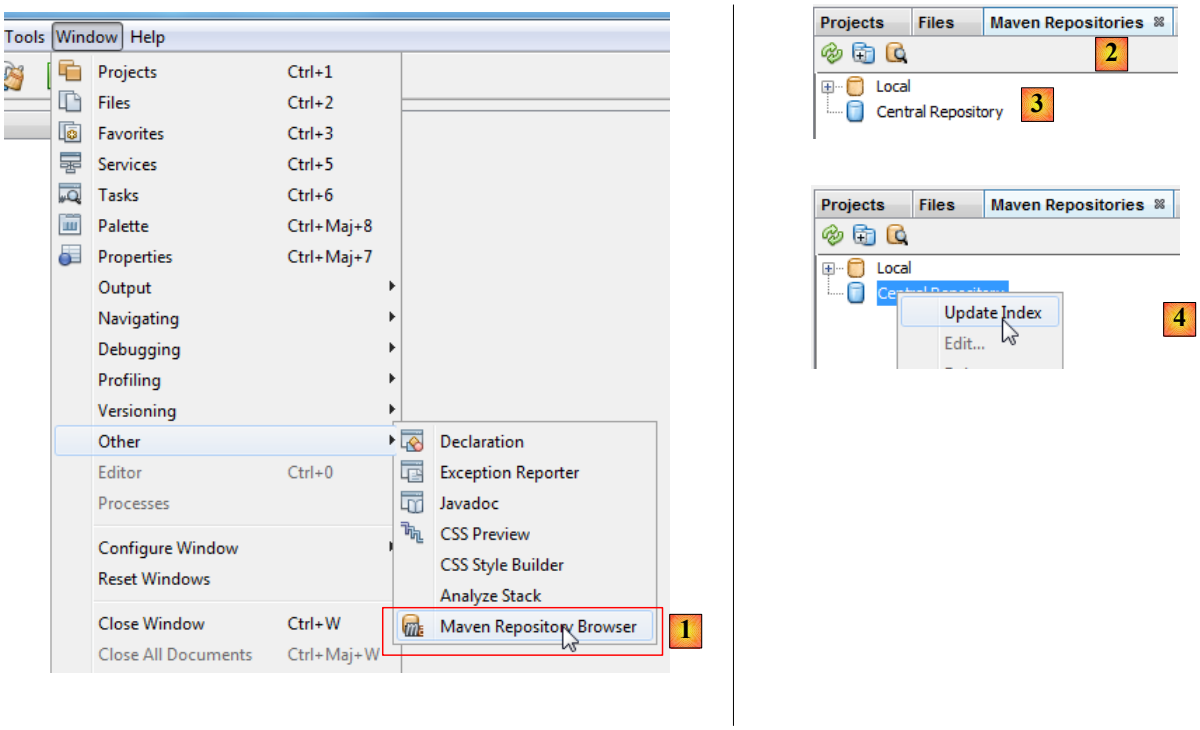
```
1. <?xml version='1.0' encoding='UTF-8' ?>
```

```
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"><head>
4.      <title>Facelet Title</title></head><body>
5.      Hello from Facelets
6.    </body>
7.  </html>
```

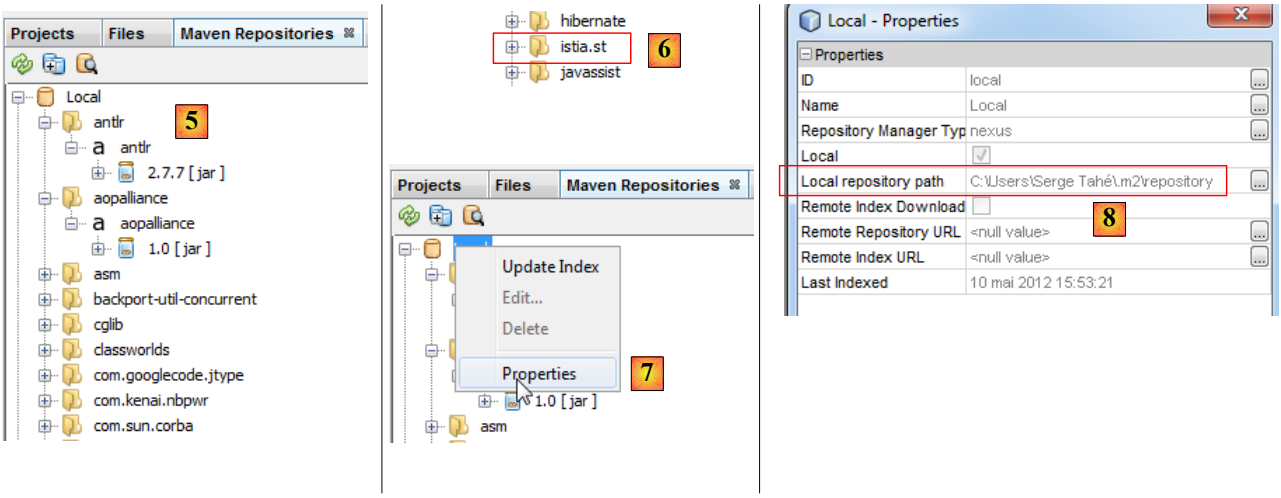
On a reçu du HTML. Toutes les balises <h:xx> de *index.xhtml* ont été traduites dans leurs correspondants HTML.

2.3.6 Le dépôt Maven local

Nous avons dit que Maven téléchargeait les dépendances nécessaires au projet et les stockait localement. On peut explorer ce dépôt local :



- en [1], on choisit l'option [Window / Other / Maven Repository Browser],
- en [2], un onglet [Maven Repositories] s'ouvre,
- en [3], il contient deux branches, une pour le dépôt local, l'autre pour le dépôt central. Ce dernier est gigantesque. Pour visualiser son contenu, il faut mettre à jour son index [4]. Cette mise à jour dure plusieurs dizaines de minutes.



- en [5], les bibliothèques du dépôt local,
- en [6], on y trouve une branche [istia.st] qui correspond au [groupId] de notre projet,
- en [7], on accède aux propriétés du dépôt local,
- en [8], on a le chemin du dépôt local. Il est utile de le connaître car parfois (rarement) Maven n'utilise plus la dernière version du projet. On fait des modifications et on constate qu'elles ne sont pas prises en compte. On peut alors supprimer manuellement la branche du dépôt local correspondant à notre [groupId]. Cela force Maven à recréer la branche à partir de la dernière version du projet.

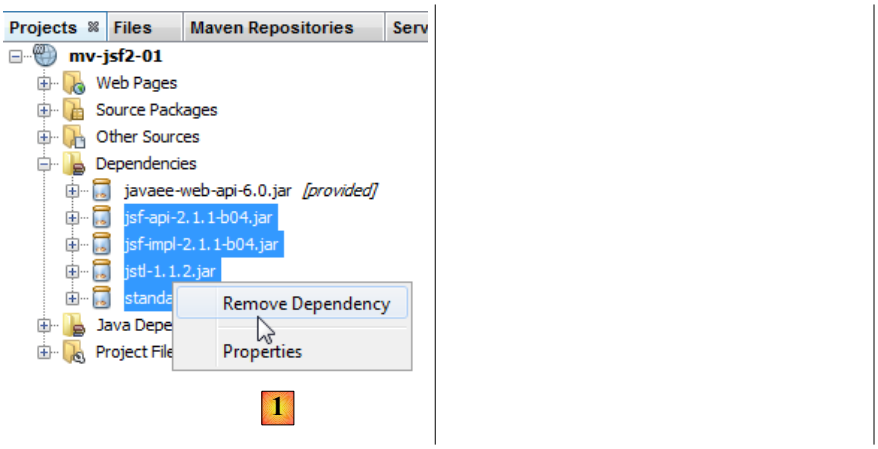
2.3.7 Chercher un artifact avec Maven

Apprenons maintenant à chercher un artifact avec Maven. Partons de la liste des dépendances actuelles du fichier [pom.xml] :

```
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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-jsf2-01</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>war</packaging>
9.
10.   <name>mv-jsf2-01</name>
11.
12.   ...
13.   <dependencies>
14.     <dependency>
15.       <groupId>com.sun.faces</groupId>
16.       <artifactId>jsf-api</artifactId>
17.       <version>2.1.1-b04</version>
18.     </dependency>
19.     <dependency>
20.       <groupId>com.sun.faces</groupId>
21.       <artifactId>jsf-impl</artifactId>
22.       <version>2.1.1-b04</version>
23.     </dependency>
24.     <dependency>
25.       <groupId>javax.servlet</groupId>
26.       <artifactId>jstl</artifactId>
27.       <version>1.1.2</version>
28.     </dependency>
29.     <dependency>
30.       <groupId>taglibs</groupId>
31.       <artifactId>standard</artifactId>
32.       <version>1.1.2</version>
33.     </dependency>
34.     <dependency>
35.       <groupId>javax</groupId>
36.       <artifactId>javaee-web-api</artifactId>
37.       <version>6.0</version>
38.       <scope>provided</scope>
39.     </dependency>
40.   </dependencies>
41.
42.   <build>
43.     ...
44.   </build>
45.   <repositories>
46.     <repository>
47.       <url>http://download.java.net/maven/2/</url>
48.       <id>jsf20</id>
49.       <layout>default</layout>
50.       <name>Repository for library Library[jsf20]</name>
51.     </repository>
52.     <repository>
53.       <url>http://repo1.maven.org/maven2/</url>
54.       <id>jstl11</id>
55.       <layout>default</layout>
56.       <name>Repository for library Library[jstl11]</name>
57.     </repository>
58.   </repositories>
```

```
59. </project>
```

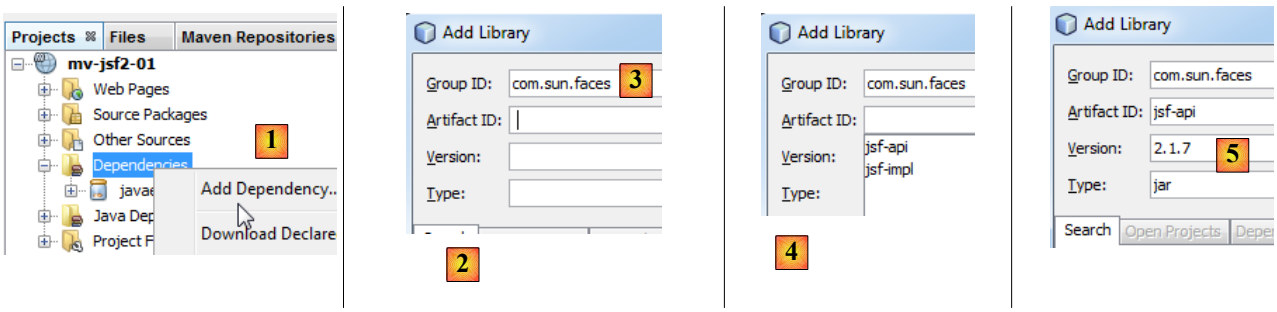
Les lignes 13-40 définissent des dépendances et les lignes 45-58 les dépôts où on peut les trouver, outre le dépôt central qui lui, est toujours utilisé. On va modifier les dépendances pour utiliser les bibliothèques dans leur version la plus récente.



Tout d'abord, nous supprimons les dépendances actuelles [1]. Le fichier [pom.xml] est alors modifié :

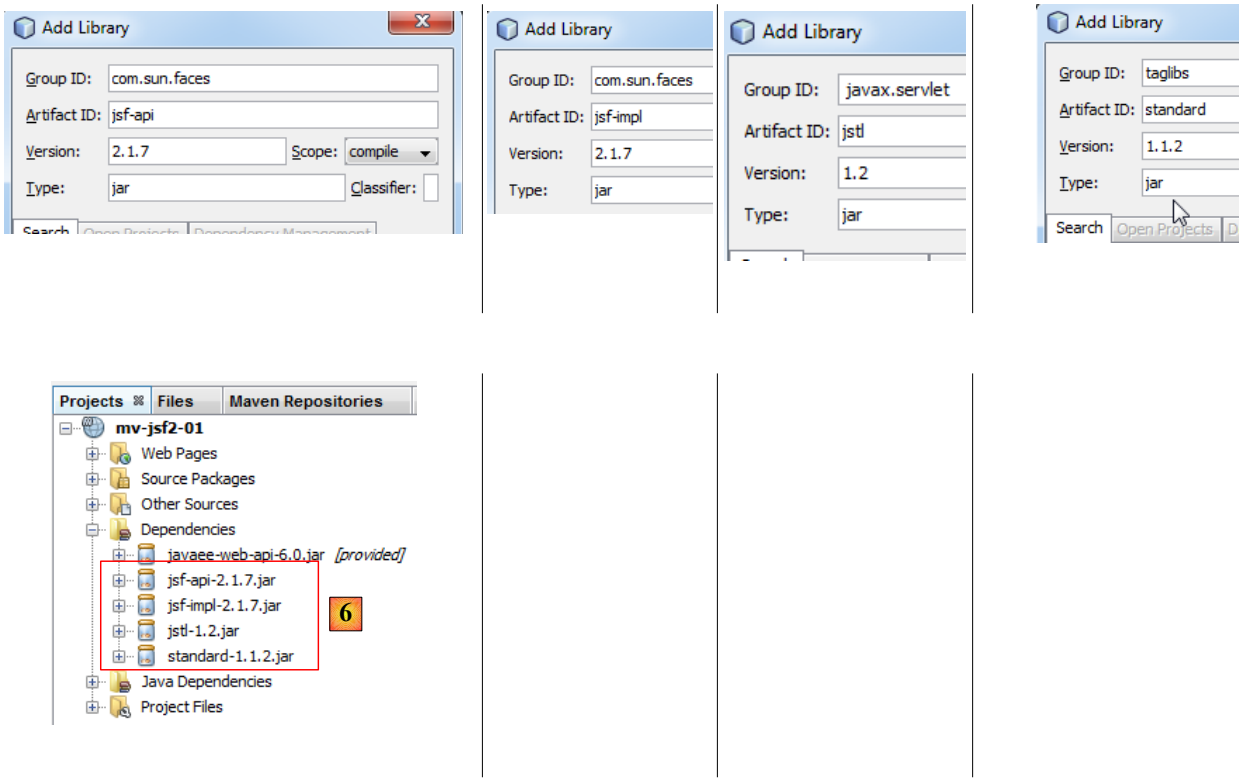
```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    4.0.0.xsd">
3.      <modelVersion>4.0.0</modelVersion>
4.  ...
5.      <dependencies>
6.          <dependency>
7.              <groupId>javax</groupId>
8.              <artifactId>javaee-web-api</artifactId>
9.              <version>6.0</version>
10.             <scope>provided</scope>
11.         </dependency>
12.     </dependencies>
13. ...
14.     <repositories>
15.         <repository>
16.             <url>http://download.java.net/maven/2/</url>
17.             <id>jsf20</id>
18.             <layout>default</layout>
19.             <name>Repository for library Library[jsf20]</name>
20.         </repository>
21.         <repository>
22.             <url>http://repo1.maven.org/maven2/</url>
23.             <id>jstl11</id>
24.             <layout>default</layout>
25.             <name>Repository for library Library[jstl11]</name>
26.         </repository>
27.     </repositories>
28. </project>
```

Lignes 5-12, les dépendances supprimées n'apparaissent plus dans [pom.xml]. Maintenant, recherchons-les dans les dépôts Maven.



- en [1], on aoute une dépendance au projet,
- en [2], on doit préciser des informations sur l'artifact cherché (groupId, artifactId, version, packaging (Type) et scope). Nous commençons par préciser le [groupId] [3],
- en [4], nous tapons [espace] pour faire afficher la liste des artifacts possibles. Ici [jsf-api] et [jsf-impl]. Nous choisissons [jsf-api],
- en [5], en procédant de la même façon, on choisit la version la plus récente. Le type de packaging est *jar*.

Nous procédons ainsi pour tous les artifacts :



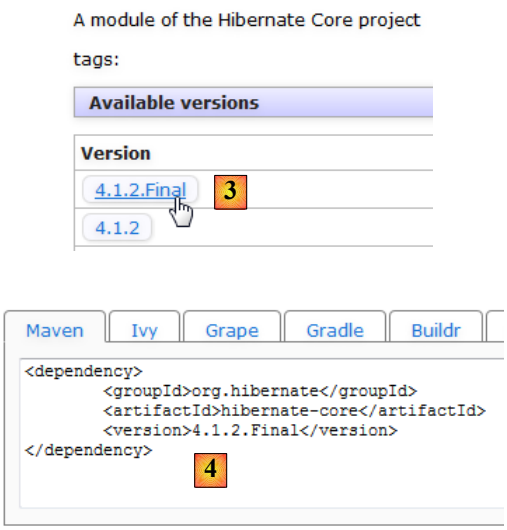
En [6], les dépendances ajoutées appraissent dans le projet. Le fichier [pom.xml] reflète ces changements :

```
1.  <dependencies>
2.          <dependency>
3.              <groupId>com.sun.faces</groupId>
4.              <artifactId>jsf-api</artifactId>
5.              <version>2.1.7</version>
6.              <type>jar</type>
7.          </dependency>
8.          <dependency>
9.              <groupId>com.sun.faces</groupId>
10.             <artifactId>jsf-impl</artifactId>
11.             <version>2.1.7</version>
12.             <type>jar</type>
13.         </dependency>
14.         <dependency>
15.             <groupId>javax.servlet</groupId>
16.             <artifactId>jstl</artifactId>
17.             <version>1.2</version>
18.             <type>jar</type>
19.         </dependency>
20.         <dependency>
21.             <groupId>taglibs</groupId>
22.             <artifactId>standard</artifactId>
23.             <version>1.1.2</version>
24.             <type>jar</type>
25.         </dependency>
26.         <dependency>
27.             <groupId>javax</groupId>
28.             <artifactId>javaee-web-api</artifactId>
```

```
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

Supposons maintenant qu'on ne connaisse pas le [groupId] de l'artifact que l'on désire. Par exemple, on veut utiliser Hibernate comme ORM (Object Relational Mapper) et c'est tout ce qu'on sait. On peut aller alors sur le site [http://mvnrepository.com/] :

En [1], on peut taper des mots clés. Tapons *hibernate* et lançons la recherche.

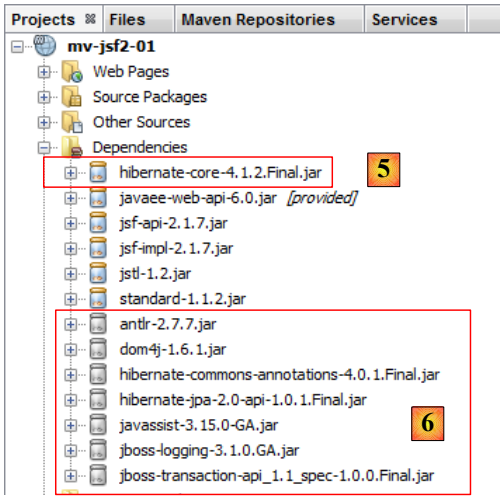


- en [2], choisissons le [groupId] *org.hibernate* et l'[artifactId] *hibernate-core*,
- en [3], choisissons la version *4.1.2-Final*,
- en [4], nous obtenons le code Maven à coller dans le fichier [pom.xml]. Nous le faisons.

```
<dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>4.1.2.Final</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
```

```
10.       <version>2.1.7</version>
11.       <type>jar</type>
12.     </dependency>
13.     ...
14.   </dependencies>
```

Nous sauvegardons le fichier [pom.xml]. Maven entreprend alors le téléchargement des nouvelles dépendances. Le projet évolue comme suit :



- en [5], la dépendance [hibernate-core-4.1.2-Final]. Dans le dépôt où il a été trouvé, cet [artifactId] est lui aussi décrit par un fichier [pom.xml]. Ce fichier a été lu et Maven a découvert que l'[artifactId] avait des dépendances. Il les télécharge également. Il fera cela pour chaque [artifactId] téléchargé. Au final, on trouve en [6] des dépendances qu'on n'avait pas demandées directement. Elles sont signalées par une icône différente de celle de l'[artifactId] principal.

Dans ce document, nous utilisons Maven principalement pour cette caractéristique. Cela nous évite de connaître toutes les dépendances d'une bibliothèque que l'on veut utiliser. On laisse Maven les gérer. Par ailleurs, en partageant un fichier [pom.xml] entre développeurs, on est assuré que chaque développeur utilise bien les mêmes bibliothèques.

Dans les exemples qui suivront, nous nous conterons de donner le fichier [pom.xml] utilisé. Le lecteur n'aura qu'à l'utiliser pour se trouver dans les mêmes conditions que le document. Par ailleurs les projets Maven sont reconnus par les principaux IDE Java (Eclipse, Netbeans, IntelliJ, Jdeveloper). Aussi le lecteur pourra-t-il utiliser son IDE favori pour tester les exemples.

2.4 Exemple mv-jsf2-02 : gestionnaire d'événement – internationalisation - navigation entre pages

2.4.1 L'application

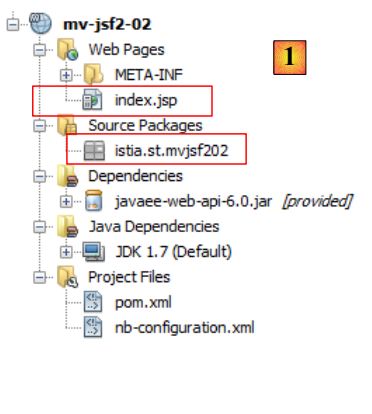
L'application est la suivante :

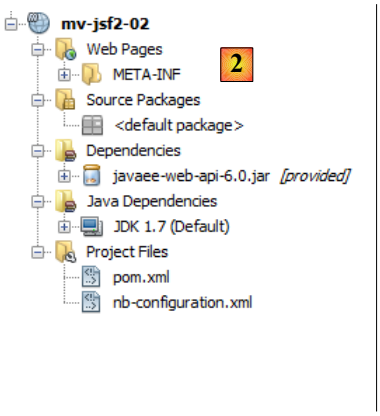
- en [1], la page d'accueil de l'application,
- en [2], deux liens pour changer la langue des pages de l'application,
- en [3], un lien de navigation vers une autre page,
- lorsqu'on clique sur [3], la page [4] est affichée,
- le lien [5] permet de revenir à la page d'accueil.

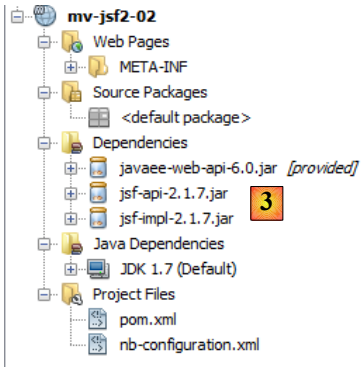
- sur la page d'accueil [1], les liens [2] permettent de changer de langue,
- en [3], la page d'accueil en anglais.

2.4.2 Le projet Netbeans

On génèrera un nouveau projet web comme expliqué au paragraphe 2.3.1, page 18. On le nommera **mv-jsf2-02** :



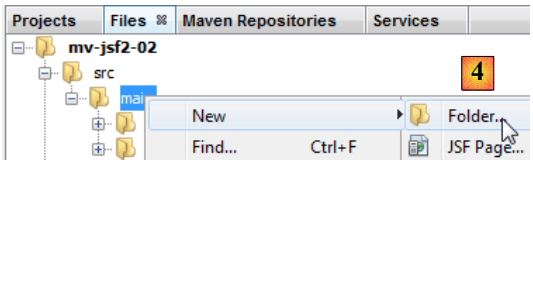


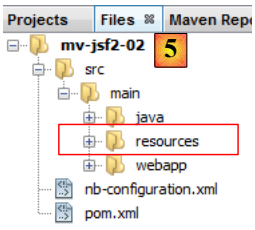


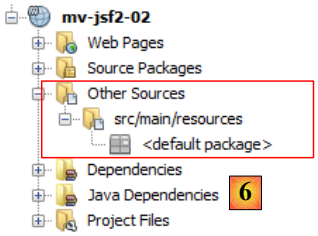
- en [1], le projet généré,
- en [2], on a supprimé le package [istia.st.mvjsf202] et le fichier [index.jsp],
- en [3], on a rajouté des dépendances Maven au moyen du fichier [pom.xml] suivant :

```
<dependencies>
        <dependency>
            <groupId>com.sun.faces</groupId>
            <artifactId>jsf-api</artifactId>
            <version>2.1.7</version>
        </dependency>
        <dependency>
            <groupId>com.sun.faces</groupId>
            <artifactId>jsf-impl</artifactId>
            <version>2.1.7</version>
        </dependency>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>6.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

Les dépendances ajoutées sont celles du framework JSF. Il suffit de copier les lignes ci-dessus dans le fichier [pom.xml] à la place des anciennes dépendances.





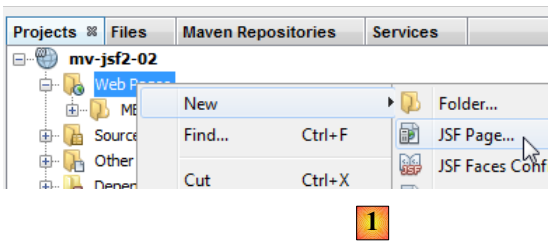


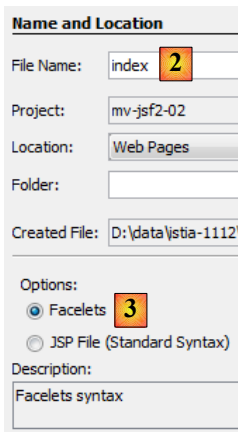
- en [4, 5] : on crée un dossier [src / main / resources] dans l'onglet [Files],
- en [6], dans l'onglet [Projects], cela a créé la branche [Other Sources].

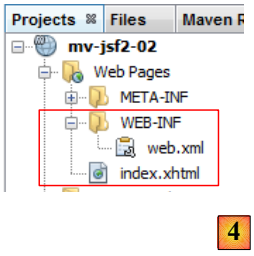
Nous avons désormais un projet JSF. Nous y créerons différents types de fichiers :

- des pages web au format XHTML,
- des classes Java,
- des fichiers de messages,
- le fichier de configuration du projet JSF.

Voyons comment créer chaque type de fichier :







- en [1], nous créons une page JSF

- en [2], nous créons une page [index.xhtml] au format [Facelets] [3],
- en [4], deux fichiers ont été créés : [index.xhtml] et [WEB-INF / web.xml].

Le fichier [web.xml] configure l'application JSF. C'est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.      <context-param>
4.          <param-name>javax.faces.PROJECT_STAGE</param-name>
5.          <param-value>Development</param-value>
6.      </context-param>
7.      <servlet>
8.          <servlet-name>Faces Servlet</servlet-name>
9.          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.         <load-on-startup>1</load-on-startup>
11.     </servlet>
12.     <servlet-mapping>
13.         <servlet-name>Faces Servlet</servlet-name>
14.         <URL-pattern>/faces/*</URL-pattern>
15.     </servlet-mapping>
16.     <session-config>
17.         <session-timeout>
18.             30
19.         </session-timeout>
20.     </session-config>
21.     <welcome-file-list>
22.         <welcome-file>faces/index.xhtml</welcome-file>
23.     </welcome-file-list>
24. </web-app>
```

Nous avons déjà commenté ce fichier en page 28 . Rappelons ses principales propriétés :

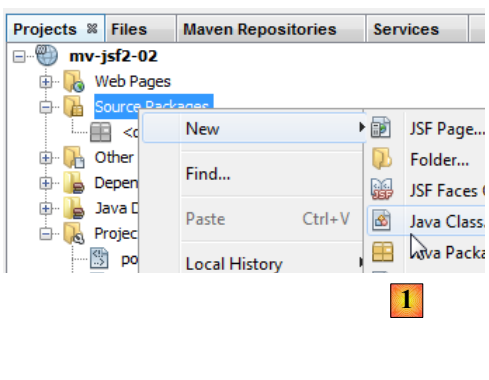
- toutes les URL du type *faces/** sont traitées par la servlet [`javax.faces.webapp.FacesServlet`],
- la page [index.xhtml] est la page d'accueil de l'application.

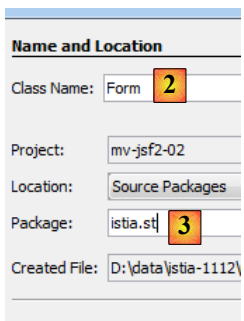
Le fichier [index.xhtml] créé est le suivant :

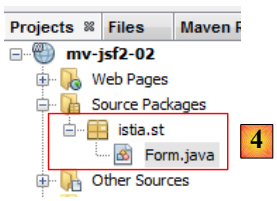
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <h:head>
6.      <title>Facelet Title</title>
7.    </h:head>
8.    <h:body>
9.      Hello from Facelets
10.   </h:body>
11. </html>
```

Nous avons déjà rencontré ce fichier page 26.

Créons maintenant une classe Java :





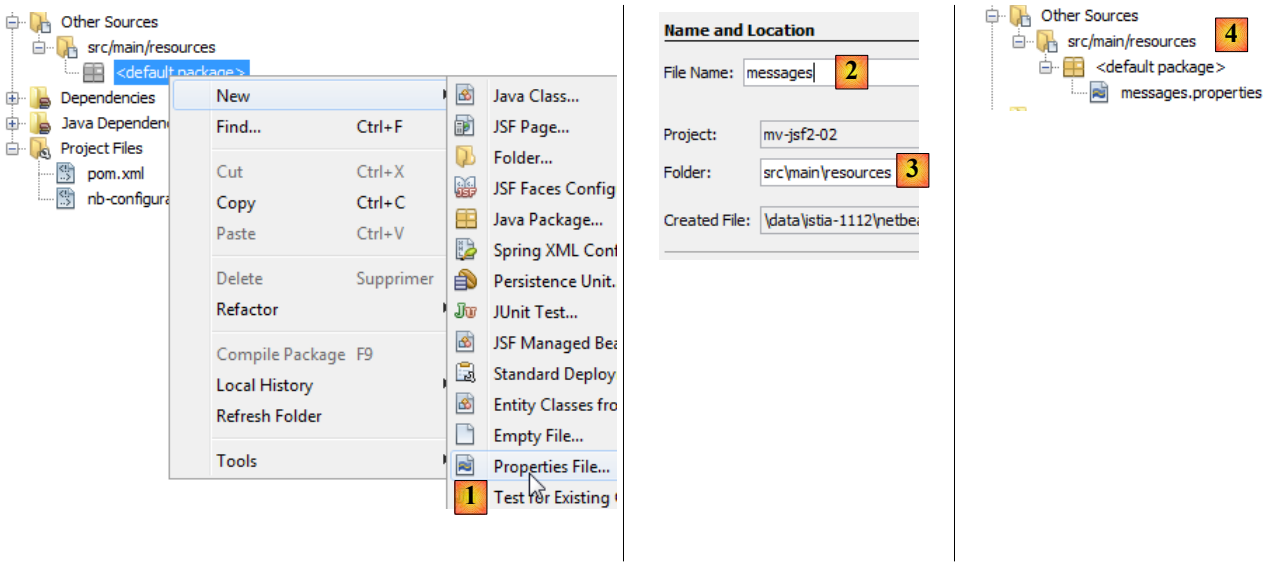


- en [1], on crée une classe Java dans la branche [Source Packages],
- en [2], on lui donne un nom et on la met dans un package [3],
- en [4], la classe créée apparaît dans le projet.

Le code de la classe créée est un squelette de classe :

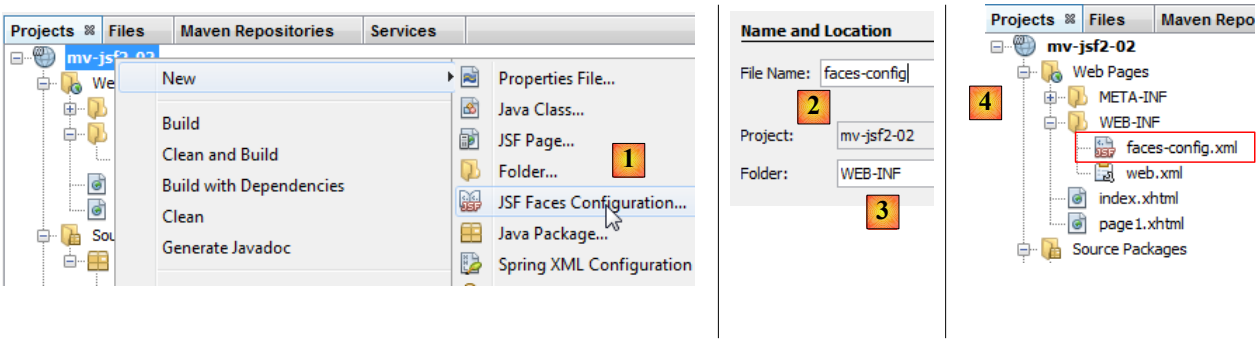
```
1.  /*
2.   * To change this template, choose Tools | Templates
3.   * and open the template in the editor.
4.   */
5.  package istia.st;
6.
7.  /**
8.   *
9.   * @author Serge Tahé
10.  */
11. public class Form {
12.
13. }
```

Enfin, créons un fichier de messages :



- en [1], création d'un fichier [Properties],
- en [2], on donne le nom du fichier et en [3] son dossier,
- en [4], le fichier [messages.properties] a été créé.

Parfois, il est nécessaire de créer le fichier [WEB-INF/faces-config.xml] pour configurer le projet JSF. Ce fichier était obligatoire avec JSF 1. Il est facultatif avec JSF 2. Il est cependant nécessaire si le site JSF est internationalisé. Ce sera le cas par la suite. Aussi montrons-nous maintenant comment créer ce fichier de configuration.



- en [1], nous créons le fichier de configuration JSF,

- en [2], on donne son nom et en [3] son dossier,
- en [4], le fichier créé.

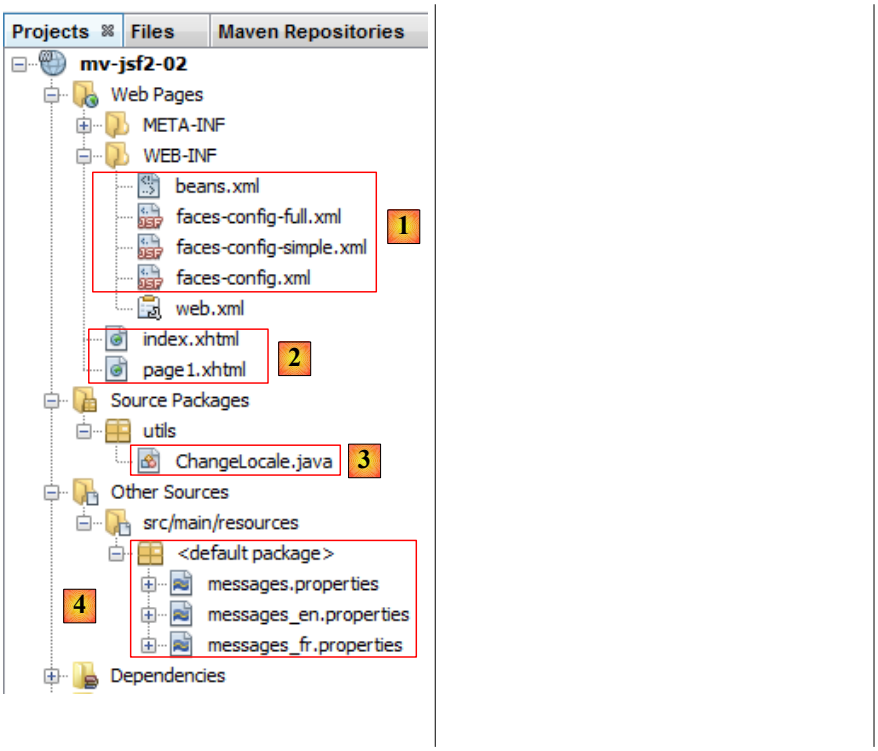
Le fichier [faces-config.xml] créé est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.      xmlns="http://java.sun.com/xml/ns/javaee"
7.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.
11. </faces-config>
```

La balise racine est <**faces-config**>. Le corps de cette balise est vide. Nous serons amenés à le compléter.

Nous avons désormais tous les éléments pour créer un projet JSF. Dans les exemples qui vont suivre, nous présentons le projet JSF complet et nous en détaillons ensuite les éléments un à un. Nous présentons maintenant un projet pour expliquer les notions :
- de gestionnaire d'événements d'un formulaire,
- d'internationalisation des pages d'un site JSF,
- de navigation entre pages.

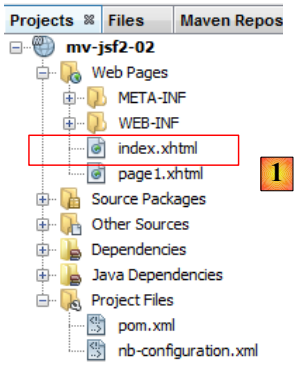
Le projet [mv-jsf2-02] devient le suivant. Le lecteur peut le trouver sur le site des exemples (cf paragraphe 1.2, page 3).



- en [1], des fichiers de configuration du projet JSF,
- en [2], les pages JSF du projet,
- en [3], l'unique classe Java,
- en [4], les fichiers de messages.

2.4.3 La page [index.xhtml]

Le fichier [index.xhtml] [1] envoie la page [2] au navigateur client :



Le code qui produit cette page est le suivant :

```
5.  <?xml version='1.0' encoding='UTF-8' ?>
6.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
7.  <html xmlns="http://www.w3.org/1999/xhtml"
8.        xmlns:h="http://java.sun.com/jsf/html"
9.        xmlns:f="http://java.sun.com/jsf/core">
10.   <f:view locale="#{changeLocale.locale}">
11.     <head>
12.       ...
13.     </head>
14.     <body>
15.       ....
16.     </body>
17.   </f:view>
18. </html>
```

- lignes 7-9 : les espaces de noms / bibliothèques de balises utilisées par la page. Les balises préfixées par **h** sont des balises HTML alors que les balises préfixées par **f** sont des balises propres à JSF,
- ligne 10 : la balise <**f:view**> sert à délimiter le code que le moteur JSF doit traiter, celui où apparaîssent les balises **<f:xx>**. L'attribut **locale** permet de préciser une langue d'affichage pour la page. Ici, nous en utiliserons deux, l'anglais et le français. La valeur de l'attribut **locale** est exprimée sous la forme d'une expression EL (Expression Language) **#{expression}**. La forme de **expression** peut être diverse. Nous l'exprimerons le plus souvent sous la forme **bean['clé']** ou **bean.champ**. Dans nos exemples, **bean** sera soit une classe Java soit un fichier de messages. Avec JSF 1, ces beans devaient être déclarés dans le fichier [faces-config.xml]. Avec JSF 2, ce n'est plus obligatoire pour les classes Java. On peut désormais utiliser des annotations qui font d'une classe Java, un bean connu de JSF 2. Le fichier des messages doit lui être déclaré dans le fichier de configuration [faces-config.xml].

2.4.4 Le bean [changeLocale]

Dans l'expression EL *#{changeLocale.locale}* :

- **changeLocale** est le nom d'un bean, ici la classe Java *ChangeLocale*,
- **locale** est un champ de la classe *ChangeLocale*. L'expression est évaluée par *[ChangeLocale].getLocale()*. De façon générale l'expression #{*bean.champ*} est évaluée comme *[Bean].getChamp()*, où *[Bean]* est une instance de la classe Java à qui on a attribué le nom *bean* et *getChamp*, le getter associé au champ *champ* du bean.

La classe *ChangeLocale* est la suivante :

```
1.  package utils;
2.
3.  import java.io.Serializable;
4.  import javax.faces.bean.ManagedBean;
5.  import javax.faces.bean.SessionScoped;
6.
7.  @ManagedBean
8.  @SessionScoped
9.  public class ChangeLocale implements Serializable{
```

```
10.   // la locale des pages
11.   private String locale="fr";
12.
13.   public ChangeLocale() {
14.   }
15.
16.   ...
17.   public String getLocale() {
18.     return locale;
19.   }
20.
21. }
```

- ligne 11 : le champ **locale**,
- ligne 17 : son getter,
- ligne 7 : l'annotation **ManagedBean** fait de la classe Java *ChangeLocale* un bean reconnu par JSF. Un bean est identifié par un nom. Celui-ci peut être fixé par l'attribut *name* de l'annotation : *@ManagedBean(name= "xx ")*. En l'absence de l'attribut *name*, le nom de la classe est utilisé en passant son premier caractère en minuscule. Le nom du bean ***C**hangeLocale* est donc ***c**hangeLocale*. On prêtera attention au fait que l'annotation **ManagedBean** appartient au package `javax.faces.bean.ManagedBean` et non au package `javax.annotations.ManagedBean`.
- ligne 8 : l'annotation **SessionScoped** fixe la portée du bean. Il y en a plusieurs. Nous utiliserons couramment les trois suivantes :
  - **RequestScoped** : la durée de vie du bean est celle du cycle demande navigateur / réponse serveur. Si pour traiter une nouvelle requête du même navigateur ou d'un autre, ce bean est de nouveau nécessaire, il sera instancié de nouveau,
  - **SessionScoped** : la durée de vie du bean est celle de la session d'un client particulier. Le bean est créé initialement pour les besoins de l'une des requêtes de ce client. Il restera ensuite en mémoire dans la session de ce client. Un tel bean mémorise en général des données propres à un client donné. Il sera détruit lorsque la session du client sera détruite,
  - **ApplicationScoped** : la durée de vie du bean est celle de l'application elle-même. Un bean avec cette durée de vie est le plus souvent partagé par tous les clients de l'application. Il est en général initialisé au début de l'application.

Ces annotations existent dans deux packages : `javax.enterprise.context` et `javax.faces.bean`. Avec l'annotation @ManagedBean il faut utiliser ce dernier package. Les annotations du package `javax.enterprise.context` peuvent elles être utilisées en combinaison avec l'annotation @Named (au lieu de @ManagedBean). Elles nécessitent un conteneur Java EE6 et sont donc plus exigeantes. On notera que la classe [ChangeLocale] implémente l'interface [Serializable]. C'est obligatoire pour les beans de portée *Session* que le serveur web peut être amené à sérialiser dans des fichiers. Nous reviendrons ultérieurement sur le bean [ChangeLocale].

2.4.5 Le fichier des messages

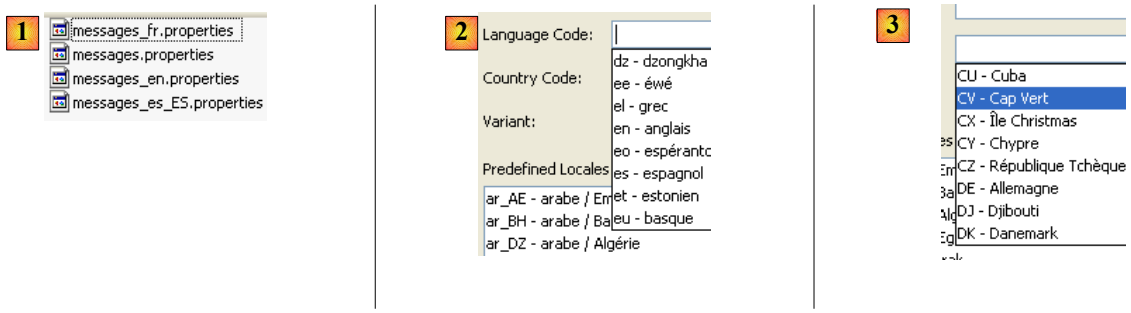
Revenons au fichier [index.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['welcome.titre']}" /></title>
9.      </head>
10.     <body>
11.     ...
12.     </body>
13.   </f:view>
14. </html>
```

- ligne 8 : la balise <**h:outputText**> affiche la valeur d'une expression EL **#{msg['welcome.titre']}** de la forme #{bean['champ']}. **bean** est soit le nom d'une classe Java soit celui d'un fichier de messages. Ici, c'est celui d'un fichier de messages. Ce dernier doit être déclaré dans le fichier de configuration [faces-config.xml]. Le bean **msg** est déclaré comme suit :

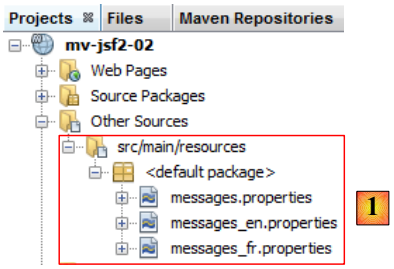
```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- =========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.
11.   <application>
12.     <resource-bundle>
13.       <base-name>
14.         messages
15.       </base-name>
16.       <var>msg</var>
17.     </resource-bundle>
18.   </application>
19. </faces-config>
```

- lignes 11-18 : la balise <**application**> sert à configurer l'application JSF,
- lignes 12-17 : la balise <**resource-bundle**> sert à définir des ressources pour l'application, ici un fichier de messages,
- lignes 13-15 : la balise <**base-name**> définit le nom du fichier de messages,
- ligne 14 : le fichier s'appellera **messages***[_CodeLangue][_CodePays].properties*. La balise <**base-name**> ne définit que la première partie du nom. Le reste est implicite. Il peut exister plusieurs fichiers de messages, un par langue :



- en [1], on voit quatre fichiers de messages correspondant au nom de base **messages** défini dans [faces-config.xml],
  - **messages_fr.properties** : contient les messages en français (code fr) ;
  - **messages_en.properties** : contient les messages en anglais (code en) ;
  - **messages_es_ES.properties** : contient les messages en espagnol (code es) de l'Espagne (code ES). Il existe d'autres types d'espagnol, par exemple celui de Bolivie (es_BO) ;
  - **messages.properties** : est utilisé par le serveur lorsque la langue de la machine sur laquelle il s'exécute n'a aucun fichier de messages qui lui est associé. Il serait utilisé par exemple, si l'application s'exécutait sur une machine en Allemagne où la langue par défaut serait l'allemand (de). Comme il n'existe pas de fichier [**messages_de.properties**], l'application utiliserait le fichier [**messages.properties**],
- en [2] : les codes des langues font l'objet d'un standard international,
- en [3] : idem pour les codes des pays.

Le nom du fichier des messages est défini ligne 14. Il sera cherché dans le *Classpath* du projet. S'il est à l'intérieur d'un paquetage, celui-ci doit être défini ligne 14, par exemple *ressources.messages*, si le fichier [messages.properties] se trouve dans le dossier [ressources] du *Classpath*. Le nom, ligne 14, ne comportant pas de paquetage, le fichier [messages.properties] doit être placé à la racine du dossier [src / main / resources] :



En [1], dans l'onglet [Projects] du projet Netbeans, le fichier [messages.properties] est présenté comme une liste des différentes versions de messages définies. Les versions sont identifiées par une suite d'un à trois codes [codeLangue_codePays_codeVariante]. En [1], seul le code [codeLangue] a été utilisé : **en** pour l'anglais, **fr** pour le français. Chaque version fait l'objet d'un fichier séparé dans le système de fichiers.

Dans notre exemple, le fichier de messages en français [**messages_fr.properties**] contiendra les éléments suivants :

```
1.  welcome.titre=Tutoriel JSF (JavaServer Faces)
2.  welcome.langue1=Fran\u00e7ais
3.  welcome.langue2=Anglais
4.  welcome.page1=Page 1
5.  page1.titre=page1
6.  page1.entete=Page 1
7.  page1.welcome=Page d'accueil
```

Le fichier [messages_en.properties] lui, sera le suivant :

```
1.  welcome.titre=JSF (JavaServer Faces) Tutorial
2.  welcome.langue1=French
3.  welcome.langue2=English
4.  welcome.page1=Page 1
5.  page1.titre=page1
6.  page1.entete=Page 1
7.  page1.welcome=Welcome page
```

Le fichier [messages.properties] est identique au fichier [messages_en.properties]. Au final, le navigateur client aura le choix entre des pages en Français et des pages en Anglais.

Revenons au fichier [faces-config.xml] qui déclare le fichier des messages :

```
1.  ...
2.
3.    <application>
4.      <resource-bundle>
5.        <base-name>
6.          messages
7.        </base-name>
8.        <var>msg</var>
9.      </resource-bundle>
10.   </application>
11. </faces-config>
```

La ligne 8 indique qu'une ligne du fichier des messages sera référencée par l'identificateur **msg** dans les pages JSF. Cet identificateur est utilisé dans le fichier [index.xhtml] étudié :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['welcome.titre']}" /></title>
9.      </head>
10.     <body>
```

```
11.       ...
12.     </body>
13.   </f:view>
14. </html>
```

La balise <**h:outputText**> de la ligne 8, va afficher la valeur du message (présence de l'identificateur **msg**) de clé **welcome.titre**. Ce message est cherché et trouvé dans le fichier [messages.properties] de la langue active du moment. Par exemple, pour le français :

```
welcome.titre=Tutoriel JSF (JavaServer Faces)
```

Un message est de la forme **clé=valeur**. La ligne 8 du fichier [index.xhtml] devient la suivante après évaluation de l'expression **#{msg['welcome.titre']}** :

```
      <title><h:outputText value="Tutoriel JSF (JavaServer Faces)" /></title>
```

Ce mécanisme des fichiers de messages permet de changer facilement la langue des pages d'un projet JSF. On parle d'**internationalisation** du projet ou plus souvent de son abréviation **i18n**, parce que le mot **internationalisation** commence par **i** et finit par **n** et qu'il y a **18** lettres entre le **i** et le **n**.

2.4.6 Le formulaire

Continuons à explorer le contenu du fichier [index.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['welcome.titre']}" /></title>
9.      </head>
10.     <body>
11.       <h:form id="formulaire">
12.         <h:panelGrid columns="2">
13.           <h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
14.           <h:commandLink value="#{msg['welcome.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
15.         </h:panelGrid>
16.         <h1><h:outputText value="#{msg['welcome.titre']}" /></h1>
17.         <h:commandLink value="#{msg['welcome.page1']}" action="page1"/>
18.       </h:form>
19.     </body>
20.   </f:view>
21. </html>
```

- lignes 11-18 : la balise <**h:form**> introduit un formulaire. Un formulaire est généralement constitué de :
  - balises de champs de saisie (texte, boutons radio, cases à cocher, listes d'éléments, ...) ;
  - balises de validation du formulaire (boutons, liens). C'est via un bouton ou un lien que l'utilisateur envoie ses saisies au serveur qui les traitera,

  Toute balise JSF peut être identifiée par un attribut **id**. Le plus souvent, on peut s'en passer et c'est ce qui a été fait dans la plupart des balises JSF utilisées ici. Néanmoins, cet attribut est utile dans certains cas. Ligne 17, le formulaire est identifié par l'id **formulaire**. Dans cet exemple, l'id du formulaire ne sera pas utilisé et aurait pu être omis.
- lignes 18-21 : la balise <**h:panelGrid**> définit ici un tableau HTML à deux colonnes. Elle donne naissance à la balise HTML <**table**>,
- le formulaire dispose de trois liens déclenchant son traitement, en lignes 19, 20 et 23. La balise <**h:commandLink**> a au moins deux attributs :
  - **value** : le texte du lien ;
  - **action** : soit une chaîne de caractères **C**, soit la référence d'une méthode qui après exécution rend la chaîne de caractères **C**. Cette chaîne de caractères **C** peut être :
    - soit le nom d'une page JSF du projet,
    - soit un nom défini dans les règles de navigation du fichier [faces-config.xml] et associé à une page JSF du projet ;

    Dans les deux cas, la page JSF est affichée, une fois que l'action définie par l'attribut **action** a été exécutée.

Examinons la mécanique du traitement des formulaires avec l'exemple du lien de la ligne 13 :

```
<h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>}"/>
```

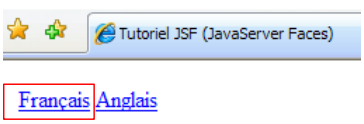
Tout d'abord, le fichier des messages est exploité pour remplacer l'expression **#{msg['welcome.langue1']}** par sa valeur. Après évaluation, la balise devient :

```
<h:commandLink value="Français" action="#{changeLocale.setFrenchLocale}"/>}"/>
```

La traduction HTML de cette balise JSF va être la suivante :

```
<a href="#" onclick="mojarra.jsfcljs(document.getElementById('formulaire'),
{'formulaire:j_idt8':'formulaire:j_idt8'},'');return false">Français</a>
```

ce qui donnera l'apparence visuelle qui suit :

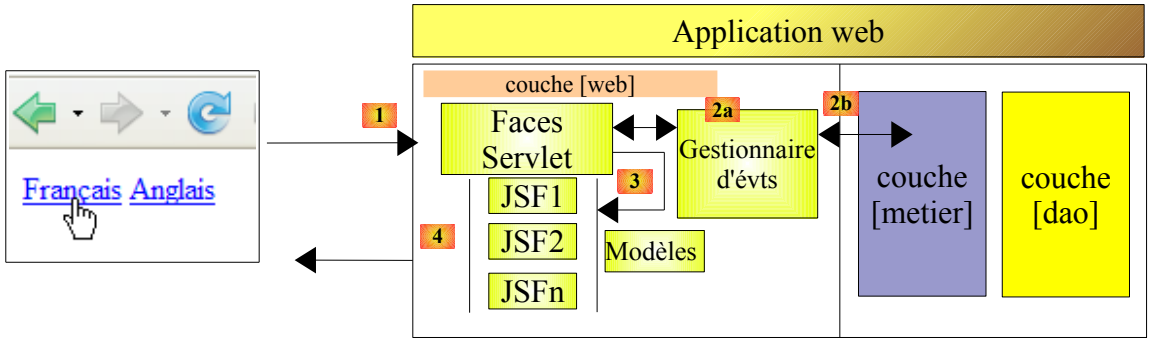


On notera l'attribut **onclick** de la balise HTML **<a>**. Lorsque l'utilisateur va cliquer sur le lien [Français], du code Javascript va être exécuté. Celui-ci est embarqué dans la page que le navigateur a reçue et c'est le **navigateur qui l'exécute**. Le code Javascript est largement utilisé dans les technologies JSF et AJAX (**A**synchronous **J**avascript **A**nd **X**ml). Il a en général pour but d'améliorer l'ergonomie et la réactivité des applications web. Il est le plus souvent généré de façon automatique par des outils logiciels et il n'est pas alors utile de le comprendre. Mais parfois un développeur peut être amené à ajouter du code Javascript dans ses pages JSF. La connaissance de Javascript est alors nécessaire.

Il est inutile ici de comprendre le code Javascript généré pour la balise JSF <**h:commandLink**>. On peut cependant noter deux points :

- le code Javascript utilise l'identifiant **formulaire** que nous avons donné à la balise JSF <**h:form**>,
- JSF génère des identifiants automatiques pour toutes les balises où l'attribut **id** n'a pas été défini. On en voit un exemple ici : j_idt8. Donner un identifiant clair aux balises permet de mieux comprendre le code Javascript généré si cela devient nécessaire. C'est notamment le cas lorsque le développeur doit lui-même ajouter du code Javascript qui manipule les composants de la page. Il a alors besoin de connaître les identifiants **id** de ses composants.

Que va-t-il se passer lorsque l'utilisateur va cliquer sur le lien [Français] de la page ci-dessus ? Considérons l'architecture d'une application JSF :



Le contrôleur [Faces Servlet] va recevoir la requête du navigateur client sous la forme HTTP suivante :

```
1.  POST /mv-jsf2-02/faces/index.xhtml HTTP/1.1
2.  Host: localhost:8080
3.  Content-Type: application/x-www-form-URLencoded
4.  Content-Length: 126
5.
6.  formulaire=formulaire&javax.faces.ViewState=-9139703055324497810%3A8197824608762605653&formulaire
    %3Aj_idt8=formulaire%3Aj_idt8
```

- lignes 1-2 : le navigateur demande l'URL [http://localhost:8080/mv-jsf2-02/faces/index.xhtml]. **C'est toujours ainsi** : les saisies faites dans un formulaire JSF initialement obtenu avec l'URL *URLFormulaire* sont envoyées à cette même URL. Le navigateur a deux moyens pour envoyer les valeurs saisies : **GET** et **POST**. Avec la méthode **GET**, les valeurs saisies sont envoyées par le navigateur dans l'URL qui est demandée. Ci-dessus, le navigateur aurait pu envoyer la première ligne suivante :

```
GET /mv-jsf2-02/faces/index.xhtml?formulaire=formulaire&javax.faces.ViewState=-
9139703055324497810%3A8197824608762605653&formulaire%3Aj_idt8=formulaire%3Aj_idt8 HTTP/1.1
```

Avec la méthode POST utilisée ici, le navigateur envoie au serveur les valeurs saisies au moyen de la ligne 6.

- ligne 3 : indique la forme d'encodage des valeurs du formulaire,
- ligne 4 : indique la taille en octets de la ligne 6,
- ligne 5 : ligne vide qui indique la fin des entêtes HTTP et le début des 126 octets des valeurs du formulaire,
- ligne 6 : les valeurs du formulaire sous la forme **element1=valeur1&element2=valeur2& ...**, la forme d'encodage définie par la ligne 3. Dans cette forme de codage, certains caractères sont remplacés par leur valeur hexadécimale. C'est le cas dans le dernier élément :

```
formulaire=formulaire&javax.faces.ViewState=...&formulaire%3Aj_idt8=formulaire%3Aj_idt8
```

où %3A représente le caractère :. C'est donc la chaîne **formulaire:j_idt8=formulaire:j_idt8** qui est envoyée au serveur. On se rappelle peut-être que nous avons déjà rencontré l'identifiant **j_idt8** lorsque nous avons examiné le code HTML généré pour la balise

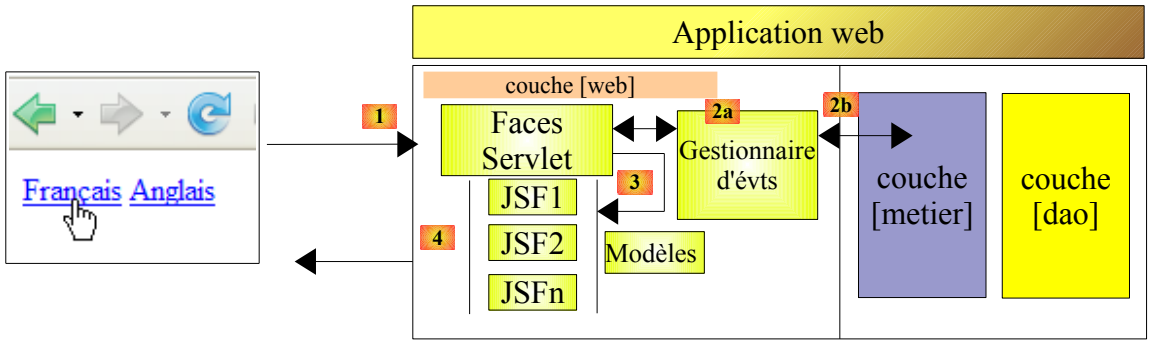
```
<h:commandLink value="#{msg['welcome.Langue1']}" action="#{changeLocale.setFrenchLocale}"/>
```

Il avait été généré de façon automatique par JSF. Ce qui nous importe ici, c'est que la présence de cet identifiant dans la chaîne des valeurs envoyées par le navigateur client permet à JSF de savoir que le lien [Français] a été cliqué. Il va alors utiliser l'attribut **action** ci-dessus, pour décider comment traiter la chaîne reçue. L'attribut `action="#{changeLocale.setFrenchLocale}"` indique à JSF que la requête du client doit être traitée par la méthode [**setFrenchLocale**] d'un objet appelé **changeLocale**. On se rappelle que ce bean a été défini par des annotations dans la classe Java [ChangeLocale] :

```
@ManagedBean
@SessionScoped
public class ChangeLocale implements Serializable{
```

Le nom d'un bean est défini par l'attribut **name** de l'annotation **@ManagedBean**. En l'absence de cet attribut, c'est le nom de la classe qui est utilisé comme nom de bean avec le premier caractère mis en minuscule.

Revenons à la requête du navigateur :



et à la balise <*h:commandLink*> qui a généré le lien [Français] sur lequel on a cliqué :

```
<h:commandLink value="#{msg['welcome.Langue1']}" action="#{changeLocale.setFrenchLocale}"/>
```

Le contrôleur va transmettre la requête du navigateur au gestionnaire d'événements défini par l'attribut **action** de la balise <h:commandLink>. Le gestionnaire d'événements **M** référencé par l'attribut **action** d'une commande <**h:commandLink**> doit avoir la signature suivante :

```
public String M();
```

- il ne reçoit aucun paramètre. Nous verrons qu'il peut néanmoins avoir accès à la requête du client,
- il doit rendre un résultat **C** de type *String*. Cette chaîne de caractères **C** peut être :
  - soit le nom d'une page JSF du projet ;
  - soit un nom défini dans les règles de navigation du fichier [faces-config.xml] et associé à une page JSF du projet ;
  - soit un pointeur null, si le navigateur client ne doit pas changer de page,

  Dans l'architecture JSF ci-dessus, le contrôleur [Faces Servlet] utilisera la chaîne **C** rendue par le gestionnaire d'événements et éventuellement son fichier de configuration [faces-config.xml] pour déterminer quelle page JSF, il doit envoyer en réponse au client [4].

Dans la balise

```
<h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
```

le gestionnaire de l'événement *clic sur le lien [Français]* est la méthode [*changeLocale.setFrenchLocale*] où **changeLocale** est une instance de la classe [utils.ChangeLocale] déjà étudiée :

```
1.  package utils;
2.
3.  import java.io.Serializable;
4.  import javax.faces.bean.SessionScoped;
5.  import javax.faces.bean.ManagedBean;
6.
7.  @ManagedBean
8.  @SessionScoped
9.  public class ChangeLocale implements Serializable{
10.   // la locale des pages
11.   private String locale="fr";
12.
13.   public ChangeLocale() {
14.   }
15.
16.   public String setFrenchLocale(){
17.     locale="fr";
18.     return null;
19.   }
20.
21.   public String setEnglishLocale(){
22.     locale="en";
23.     return null;
24.   }
25.
26.   public String getLocale() {
27.     return locale;
28.   }
29. }
```

La méthode *setFrenchLocale* a bien la signature des gestionnaires d'événements. Rappelons-nous que le gestionnaire d'événements doit traiter la requête du client. Puisqu'il ne reçoit pas de paramètres, comment peut-il avoir accès à celle-ci ? Il existe diverses façons de faire :

- le bean **B** qui contient le gestionnaire d'événements de la page JSF **P** est aussi souvent celui qui contient le modèle **M** de cette page. Cela signifie que le bean **B** contient des champs qui seront initialisés par les valeurs saisies dans la page **P**. Cela sera fait par le contrôleur [Faces Servlet] **avant** que le gestionnaire d'événements du bean **B** ne soit appelé. Ce gestionnaire aura donc accès, via les champs du bean **B** auquel il appartient, aux valeurs saisies par le client dans le formulaire et pourra les traiter.
- la méthode statique [FacesContext.getCurrentInstance()] de type [FacesContext] donne accès au contexte d'exécution de la requête JSF courante qui est un objet de type [FacesContext]. Le contexte d'exécution de la requête ainsi obtenu, permet d'avoir accès aux paramètres postés au serveur par le navigateur client avec la méthode suivante :

  ```
  Map FacesContext.getCurrentInstance().getExternalContext().getRequestParameterMap()
  ```

  Si les paramètres postés (POST) par le navigateur client sont les suivants :

  ```
  formulaire=formulaire&javax.faces.ViewState=...&formulaire%3Aj_id_id21=formulaire%3Aj_id_id21
  ```

  la méthode *getRequestParameterMap*() rendra le dictionnaire suivant :

clé	valeur
formulaire	formulaire
javax.faces.ViewState	...
formulaire:j_id_id21	formulaire:j_id_id21

Dans la balise

```
<h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
```

qu'attend-on du gestionnaire d'événements ***locale***.*setFrenchLocale* ? On veut qu'il fixe la langue utilisée par l'application. Dans le jargon Java, on appelle cela " localiser " l'application. Cette localisation est utilisée par la balise <**f:view**> de la page JSF [index.xhtml] :

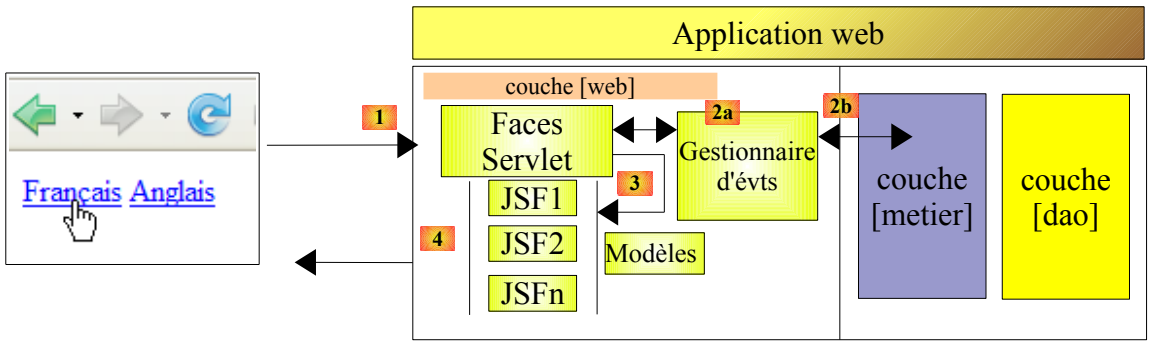
```
1.    <f:view locale="#{changeLocale.locale}">
2.      ...
3.  </f:view>
```

Pour passer la page en Français, il suffit que l'attribut **locale** ait la valeur **fr**. Pour la passer en anglais, il faut lui donner la valeur **en**. La valeur de l'attribut **locale** est obtenue par l'expression *[ChangeLocale].getLocale()*. Cette expression rend la valeur du champ *locale* de la classe [ChangeLocale]. On en déduit le code de la méthode *[ChangeLocale].setFrenchLocale()* qui doit passer les pages en Français :

```
1.    public String setFrenchLocale(){
2.      locale="fr";
3.      return null;
4.  }
```

Nous avons expliqué qu'un gestionnaire d'événements devait rendre une chaîne de caractères **C** qui sera utilisée par [Faces Servlet] afin de trouver la page JSF à envoyer en réponse au navigateur client. Si la page à renvoyer est la même que celle en cours de traitement, le gestionnaire d'événements peut se contenter de renvoyer la valeur **null**. C'est ce qui est fait ici ligne 3 : on veut renvoyer la même page [index.xhtml] mais dans une langue différente.

Revenons à l'architecture de traitement de la requête :



Le gestionnaire d'événements *changeLocale.setFrenchLocale* a été exécuté et a rendu la valeur *null* au contrôleur [Faces Servlet]. Celui-ci va donc réafficher la page [index.xhtml]. Revoyons celle-ci :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['welcome.titre']}" /></title>
9.      </head>
10.     <body>
11.       <h:form id="formulaire">
```

```
12.         <h:panelGrid columns="2">
13.           <h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
14.           <h:commandLink value="#{msg['welcome.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
15.         </h:panelGrid>
16.         <h1><h:outputText value="#{msg['welcome.titre']}" /></h1>
17.         <h:commandLink value="#{msg['welcome.page1']}" action="page1"/>
18.       </h:form>
19.     </body>
20.   </f:view>
21. </html>
```

A chaque fois qu'une valeur de type #{msg['...']} est évaluée, l'un des fichiers des messages [messages.properties] est utilisé. Celui utilisé est celui qui correspond à la " localisation " de la page (ligne 6). Le gestionnaire d'événements *changeLocale.setFrenchLocale* définissant cette localisation à *fr*, c'est le fichier [messages_fr.properties] qui sera utilisé. Un clic sur le lien [Anglais] (ligne 14) changera la localisation en *en* (cf méthode *changeLocale.setEnglishLocale*). Ce sera alors le fichier [messages_en.properties] qui sera utilisé et la page apparaîtra en anglais :

A chaque fois que la page [index.xhtml] est affichée, la balise *<f:view>* est exécutée :

```
<f:view locale="#{changeLocale.locale}">
```

et donc la méthode *[ChangeLocale].getLocale()* est réexécutée. Comme nous avons donné la portée **Session** à notre bean :

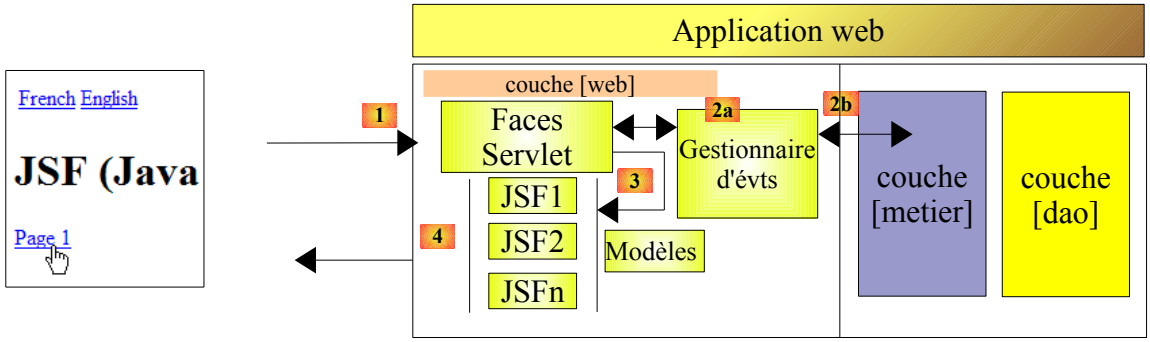
```
@ManagedBean
@SessionScoped
public class ChangeLocale implements Serializable{
```

la localisation faite lors d'une requête est conservée pour les requêtes suivantes.

Il nous reste un dernier élément de la page [index.xhtml] à étudier :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['welcome.titre']}" /></title>
9.      </head>
10.     <body>
11.       <h:form id="formulaire">
12.         <h:panelGrid columns="2">
13.           <h:commandLink value="#{msg['welcome.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
14.           <h:commandLink value="#{msg['welcome.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
15.         </h:panelGrid>
16.         <h1><h:outputText value="#{msg['welcome.titre']}" /></h1>
17.         <h:commandLink value="#{msg['welcome.page1']}" action="page1"/>
18.       </h:form>
19.     </body>
20.   </f:view>
21. </html>
```

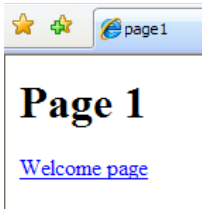
La balise <h:commandLink> de la ligne 17, a un attribut **action** égal à une chaîne de caractères. Dans ce cas, aucun gestionnaire d'événements n'est appelé pour traiter la page. On passe tout de suite à la page [page1.xhtml]. Examinons le fonctionnement de l'application dans ce cas d'utilisation :



L'utilisateur clique sur le lien [Page 1]. Le formulaire est posté au contrôleur [Faces Servlet]. Celui reconnaît dans la requête qu'il reçoit, le fait que le lien [Page 1] a été cliqué. Il examine la balise correspondante :

```
<h:commandLink value="#{msg['welcome.page1']}" action="page1"/>
```

Il n'y a pas de gestionnaire d'événements associé au lien. Le contrôleur [Faces Servlet] passe tout de suite à l'étape [3] ci-dessus et affiche la page [page1.xhtml] :



2.4.7 La page JSF [page1.xhtml]

La page [page1.xhtml] envoie le flux suivant au navigateur client :

Le code qui produit cette page est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.    <f:view locale="#{changeLocale.locale}">
7.      <head>
8.        <title><h:outputText value="#{msg['page1.titre']}"/></title>
```

```
9.      </head>
10.     <body>
11.       <h1><h:outputText value="#{msg['page1.entete']}"/></h1>
12.       <h:form>
13.         <h:commandLink value="#{msg['page1.welcome']}" action="index"/>
14.       </h:form>
15.     </body>
16.   </f:view>
17. </html>
```

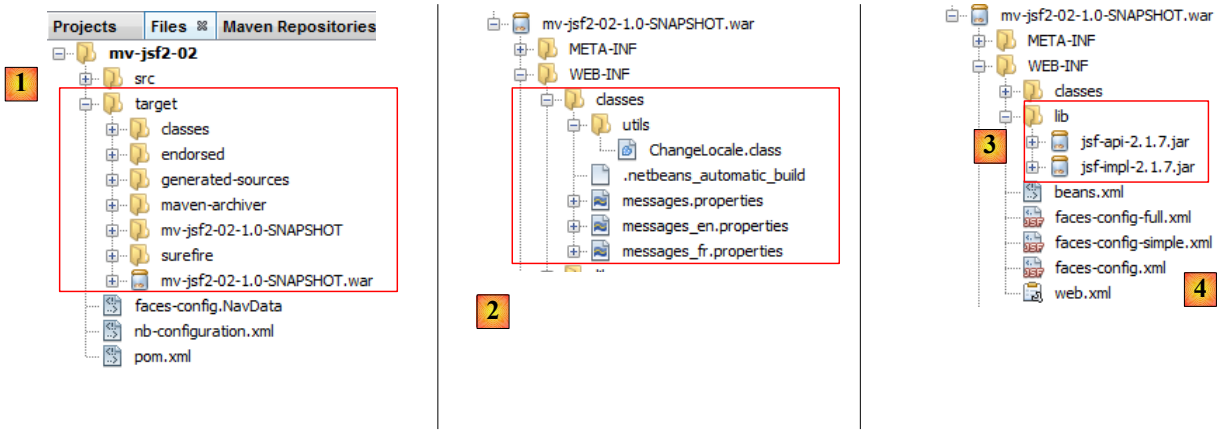
Il n'y a dans cette page rien qui n'ait déjà été expliqué. Le lecteur fera la correspondance entre le code JSF et la page envoyée au navigateur client. Le lien de retour à la page d'accueil :

```
<h:commandLink value="#{msg['page1.welcome']}" action="index"/>
```

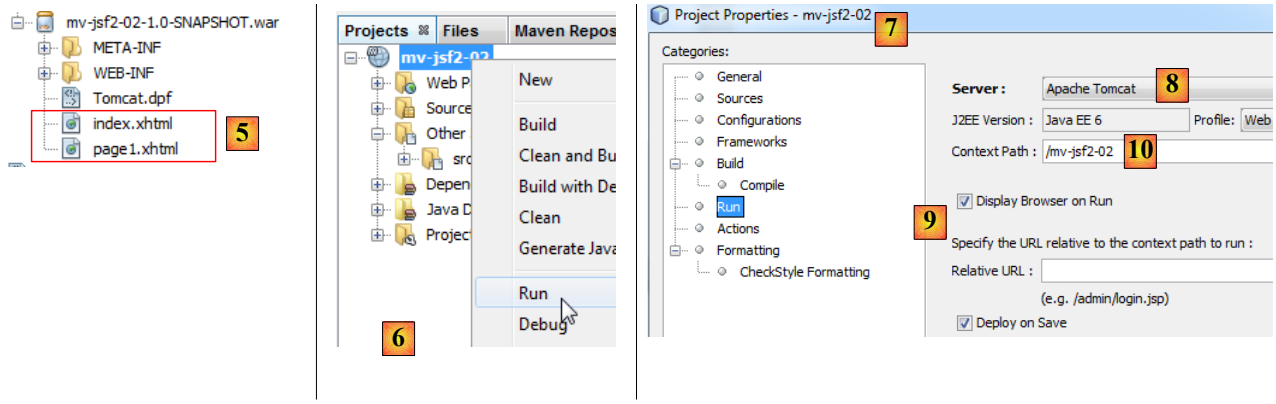
fera afficher la page [index.xhtml].

2.4.8 Exécution du projet

Notre projet est désormais complet. Nous pouvons le construire (Clean and Build) :



- la construction du projet crée dans l'onglet [Files] le dossier [target]. Dans celui-ci, on trouve l'archive [mv-jsf2-02-1.0-SNAPSHOT.war] du projet. C'est cette archive qui est déployée sur le serveur,
- dans [WEB-INF / classes] [2], on trouve les classes compilées du dossier [Source Packages] du projet ainsi que les fichiers qui se trouvaient dans la branche [Other Sources], ici les fichiers de messages,
- dans [WEB-INF / lib] [3], on trouve les bibliothèques du projet,
- à la racine de [WEB-INF] [4], on trouve les fichiers de configuration du projet,



- à la racine de l'archive [5], on trouve les pages JSF qui étaient dans la branche [Web Pages] du projet,
- une fois le projet construit, il peut être exécuté [6]. Il va être exécuté selon sa configuration d'exécution [7],
- le serveur Tomcat va être lancé s'il n'était pas déjà lancé [8],

- l'archive [mv-jsf2-02-1.0-SNAPSHOT.war] va être chargée sur le serveur. On appelle cela le déploiement du projet sur le serveur d'application,
- en [9], il est demandé de lancer un navigateur à l'exécution. Celui-ci va demander le contexte de l'application [10], c.a.d. l'URL [http://localhost:8080/mv-jsf2-02]. D'après les règles du fichier [web.xml] (cf page 39), c'est le fichier [faces/index.xhtml] qui va être servi au navigateur client. Puisque l'URL est de la forme [/faces/*], elle va être traitée par le contrôleur [Faces Servlet] (cf [web.xml] page 39). Celui-ci va traiter la page et envoyer le flux HTML suivant :

- le contrôleur [Faces Servlet] traitera alors les événements qui vont se produire à partir de cette page.

2.4.9 Le fichier de configuration [faces-config.xml]

Nous avons utilisé le fichier [faces-config.xml] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
9.
10.
11.    <application>
12.      <resource-bundle>
13.        <base-name>
14.          messages
15.        </base-name>
16.        <var>msg</var>
17.      </resource-bundle>
18.    </application>
19. </faces-config>
```

C'est le fichier minimal pour une application JSF 2 internationalisée. Nous avons utilisé ici des possibilités nouvelles de JSF 2 vis à vis de JSF 1 :

- déclarer des beans et leur portée avec les annotations *@ManagedBean*, *@RequestScoped*, *@SessionScoped*, *@ApplicationScoped*,
- naviguer entre des pages en utilisant comme clés de navigation, les **noms** des pages XHTML sans leur suffixe xhtml.

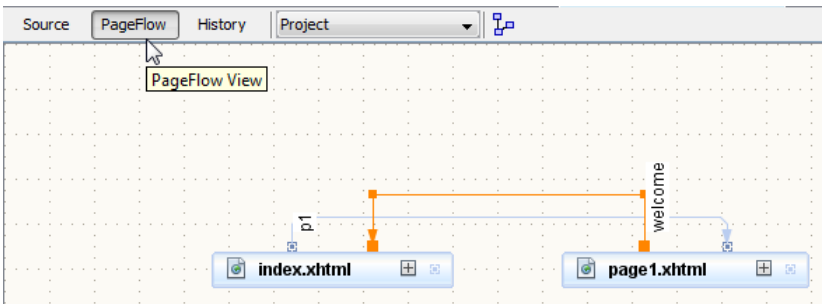
On peut vouloir ne pas utiliser ces possibilités et déclarer ces éléments du projet JSF dans [faces-config.xml] comme dans JSF 1. Dans ce cas, le fichier [faces-config.xml] pourrait être le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
```

```
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.  <!-- application -->
10.   <application>
11.     <resource-bundle>
12.       <base-name>
13.         messages
14.       </base-name>
15.       <var>msg</var>
16.     </resource-bundle>
17.   </application>
18.
19.   <!-- managed beans -->
20.   <managed-bean>
21.     <managed-bean-name>changeLocale</managed-bean-name>
22.     <managed-bean-class>utils.ChangeLocale</managed-bean-class>
23.     <managed-bean-scope>session</managed-bean-scope>
24.   </managed-bean>
25.
26.    <!-- navigation -->
27.   <navigation-rule>
28.     <description/>
29.     <from-view-id>/index.xhtml</from-view-id>
30.     <navigation-case>
31.       <from-outcome>p1</from-outcome>
32.       <to-view-id>/page1.xhtml</to-view-id>
33.     </navigation-case>
34.   </navigation-rule>
35.
36.   <navigation-rule>
37.     <description/>
38.     <from-view-id>/page1.xhtml</from-view-id>
39.     <navigation-case>
40.       <from-outcome>welcome</from-outcome>
41.       <to-view-id>/index.xhtml</to-view-id>
42.     </navigation-case>
43.   </navigation-rule>
44.
45.
46. </faces-config>
```

- lignes 20-24 : déclaration du bean *changeLocale* :
  - ligne 21 : nom du bean ;
  - ligne 22 : nom complet de la classe associée au bean ;
  - ligne 23 : portée du bean. Les valeurs possibles sont *request*, *session*, *application*,
- lignes 27-34 : déclaration d'une règle de navigation :
  - ligne 28 : on peut décrire la règle. Ici, on ne l'a pas fait ;
  - ligne 29 : la page à parir de laquelle on navigue (point de départ) ;
  - lignes 30-33 : un cas de navigation. Il peut y en avoir plusieurs ;
  - ligne 31 : la clé de navigation ;
  - ligne 32 : la page vers laquelle on navigue.

Les règles de navigation peuvent être affichées d'une façon plus visuelle. Lorsque le fichier [faces-config.xml] est édité, on peut utiliser l'onglet [PageFlow] :



Supposons que nous utilisions le fichier [faces-config.xml] précédent. Comment évoluerait notre application ?

- dans la classe [ChangeLocale], les annotations *@ManagedBean* et *@SessionScoped* disparaîtraient puisque désormais le bean est déclaré dans [faces-config],
- la navigation de [index.xhtml] vers [page1.xhtml] par un lien deviendrait :

```
<h:commandLink value="#{msg['welcome.page1']}" action="p1"/>
```

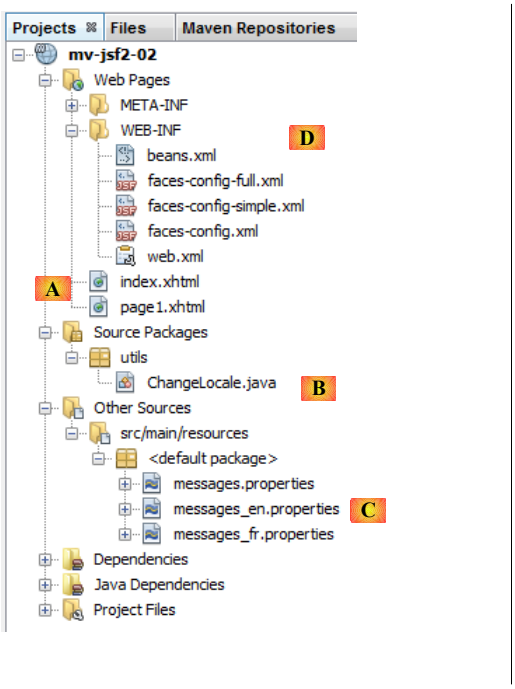
  A l'attribut **action**, on affecte la clé de navigation **p1** définie dans [faces-config],
- la navigation de [page1.xhtml] vers [index.xhtml] par un lien deviendrait :

```
<h:commandLink value="#{msg['page1.welcome']}" action="welcome"/>
```

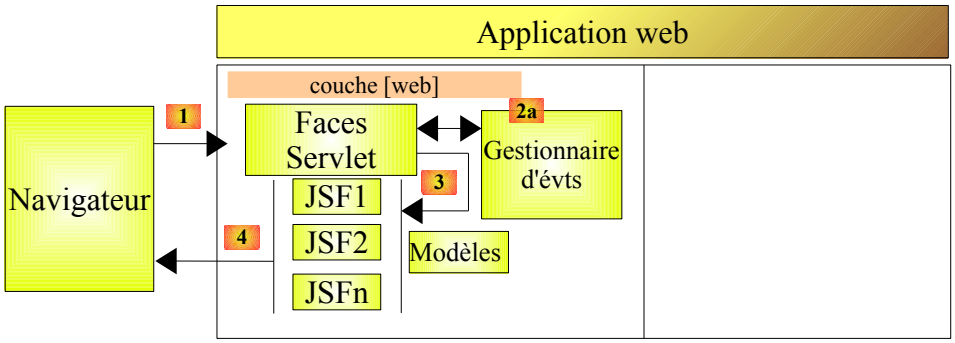
  A l'attribut **action**, on affecte la clé de navigation **welcome** définie dans [faces-config],
- les méthodes *setFrenchLocale* et *setEnglishLocale* qui doivent rendre une clé de navigation, n'ont pas à être modifiées car elles rendaient **null** pour indiquer qu'on restait sur la même page.

2.4.10 Conclusion

Revenons sur le projet Netbeans que nous avons écrit :



Ce projet recouvre l'architecture suivante :



Dans chaque projet JSF, nous trouverons les éléments suivants :

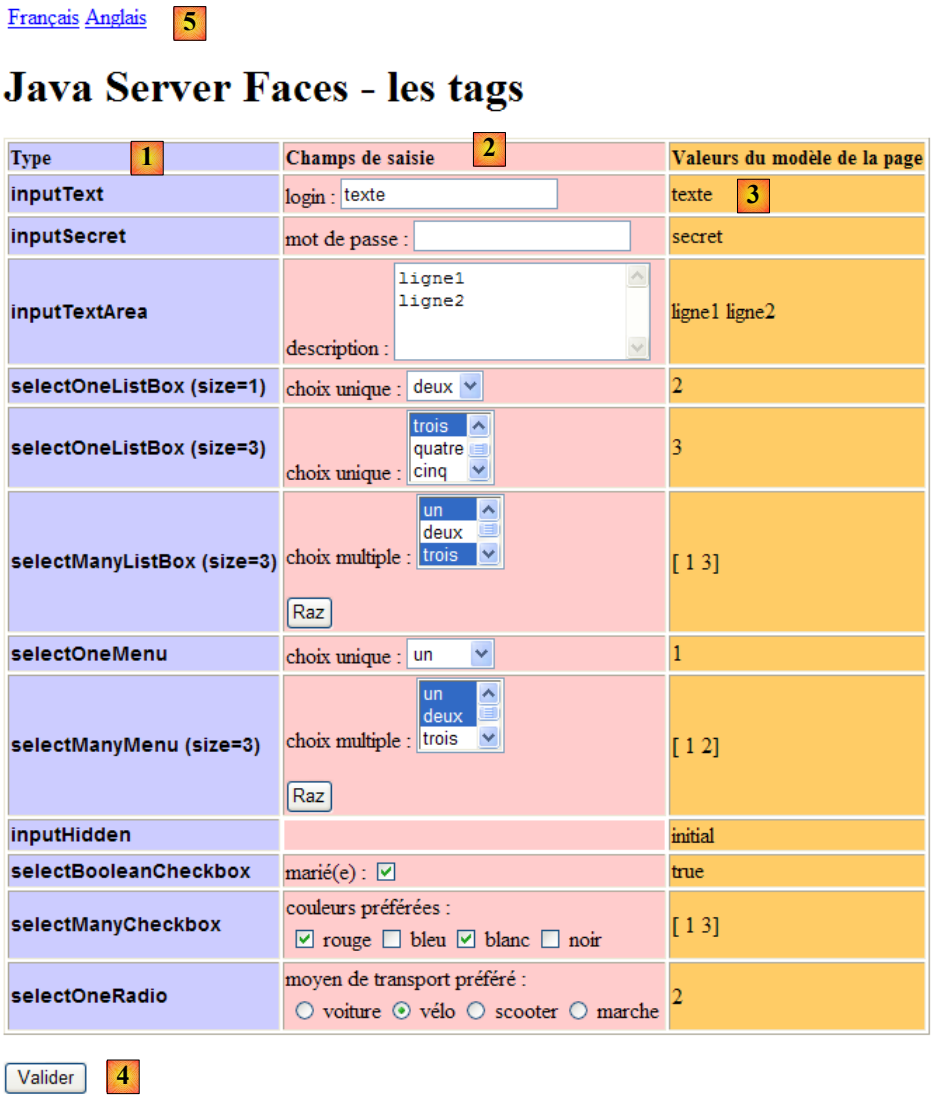
- des pages JSF [A] qui sont envoyées [4] aux navigateurs clients par le contrôleur [Faces Servlet] [3],
- des fichiers de messages [C] qui permettent de changer la langue des pages JSF,
- des classes Java [B] qui traitent les événements qui se produisent sur le navigateur client [2a, 2b] et / ou qui servent de modèles aux pages JSF [3]. Le plus souvent, les couches [métier] et [DAO] sont développées et testées séparément. La couche [web] est alors testée avec une couche [métier] fictive. Si les couches [métier] et [DAO] sont disponibles, on travaille le plus souvent avec leurs archives .jar.
- des fichiers de configuration [D] pour lier ces divers éléments entre-eux. Le fichier [web.xml] a été décrit page 39, et sera peu souvent modifié. De même pour [faces-config] où nous utiliserons toujours la version simplifiée.

2.5 Exemple mv-jsf2-03 : formulaire de saisie - composants JSF

A partir de maintenant, nous ne montrerons plus la construction du projet. Nous présentons des projets tout faits et nous en expliquons le fonctionnement. Le lecteur peut récupérer l'ensemble des exemples sur le site de ce document (cf paragraphe 1.2).

2.5.1 L'application

L'application a une unique vue :

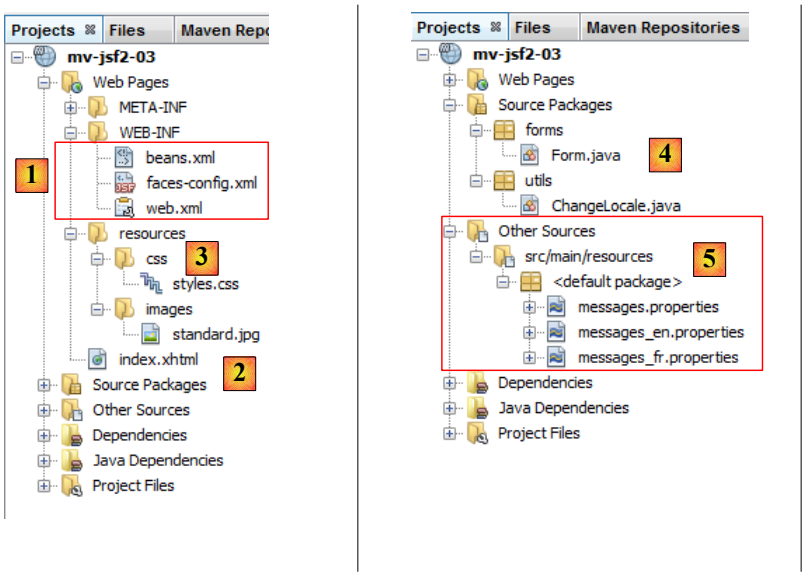


L'application présente les principaux composants JSF utilisables dans un formulaire de saisies :

- la colonne [1] indique le nom de la balise JSF / HTML utilisée,
- la colonne [2] présente un exemple de saisie pour chacune des balises rencontrées,
- la colonne [3] affiche les valeurs du bean servant de modèle à la page,
- les saisies faites en [2] sont validées par le bouton [4]. Cette validation ne fait que mettre à jour le bean modèle de la page. La même page est ensuite renvoyée. Aussi après validation, la colonne [3] présente-t-elle les nouvelles valeurs du bean modèle permettant ainsi à l'utilisateur de vérifier l'impact de ses saisies sur le modèle de la page.

2.5.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



- en [1], les fichiers de configuration du projet JSF,
- en [2], l'unique page du projet : **index.xhtml,**
- en [3], une feuille de style [styles.css] pour configurer l'aspect de la page [index.xhtml]
- en [4], les classes Java du projet,
- en [5], le fichier des messages de l'application en deux langues : français et anglais.

2.5.3 Le fichier [pom.xml]

Nous ne présentons que les dépendances :

```
1.      <dependencies>
2.          <dependency>
3.              <groupId>com.sun.faces</groupId>
4.              <artifactId>jsf-api</artifactId>
5.              <version>2.1.7</version>
6.          </dependency>
7.          <dependency>
8.              <groupId>com.sun.faces</groupId>
9.              <artifactId>jsf-impl</artifactId>
10.             <version>2.1.7</version>
11.         </dependency>
12.         <dependency>
13.             <groupId>javax</groupId>
14.             <artifactId>javaee-web-api</artifactId>
15.             <version>6.0</version>
16.             <scope>provided</scope>
17.         </dependency>
18. </dependencies>
```

Ce sont les dépendances nécessaires à un projet JSF. Dans les exemples qui suivront, ce fichier ne sera présenté que lorsqu'il changera.

2.5.4 Le fichier [web.xml]

Le fichier [web.xml] a été configuré pour que la page [index.xhtml] soit la page d'accueil du projet :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.    <context-param>
4.      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
5.      <param-value>client</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.PROJECT_STAGE</param-name>
9.      <param-value>Development</param-value>
10.   </context-param>
11.   <context-param>
12.     <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
13.     <param-value>true</param-value>
14.   </context-param>
15.   <servlet>
16.     <servlet-name>Faces Servlet</servlet-name>
17.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
18.     <load-on-startup>1</load-on-startup>
19.   </servlet>
20.   <servlet-mapping>
21.     <servlet-name>Faces Servlet</servlet-name>
22.     <url-pattern>/faces/*</url-pattern>
23.   </servlet-mapping>
24.   <session-config>
25.     <session-timeout>
26.       30
27.     </session-timeout>
28.   </session-config>
29.   <welcome-file-list>
30.     <welcome-file>faces/index.xhtml</welcome-file>
31.   </welcome-file-list>
32. </web-app>
```

- ligne 30 : la page [index.xhtml] est page d'accueil,
- lignes 11-14 : un paramètre pour la servlet [Faces Servlet]. Elle demande à ce que les commentaires dans une facelet du genre :

```
<!-- langues -->
```

  soient ignorés. Sans ce paramètre, les commentaires posent des problèmes peu compréhensibles,
- lignes 3-6 : un paramètre pour la servlet [Faces Servlet] qui sera expliqué un peu plus loin.

2.5.5 Le fichier [faces-config.xml]

Le fichier [faces-config.xml] de l'application est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <resource-bundle>
12.       <base-name>
13.         messages
14.       </base-name>
```

```
15.       <var>msg</var>
16.     </resource-bundle>
17.   </application>
18. </faces-config>
```

- lignes 11-16 : configurent le fichier des messages de l'application.

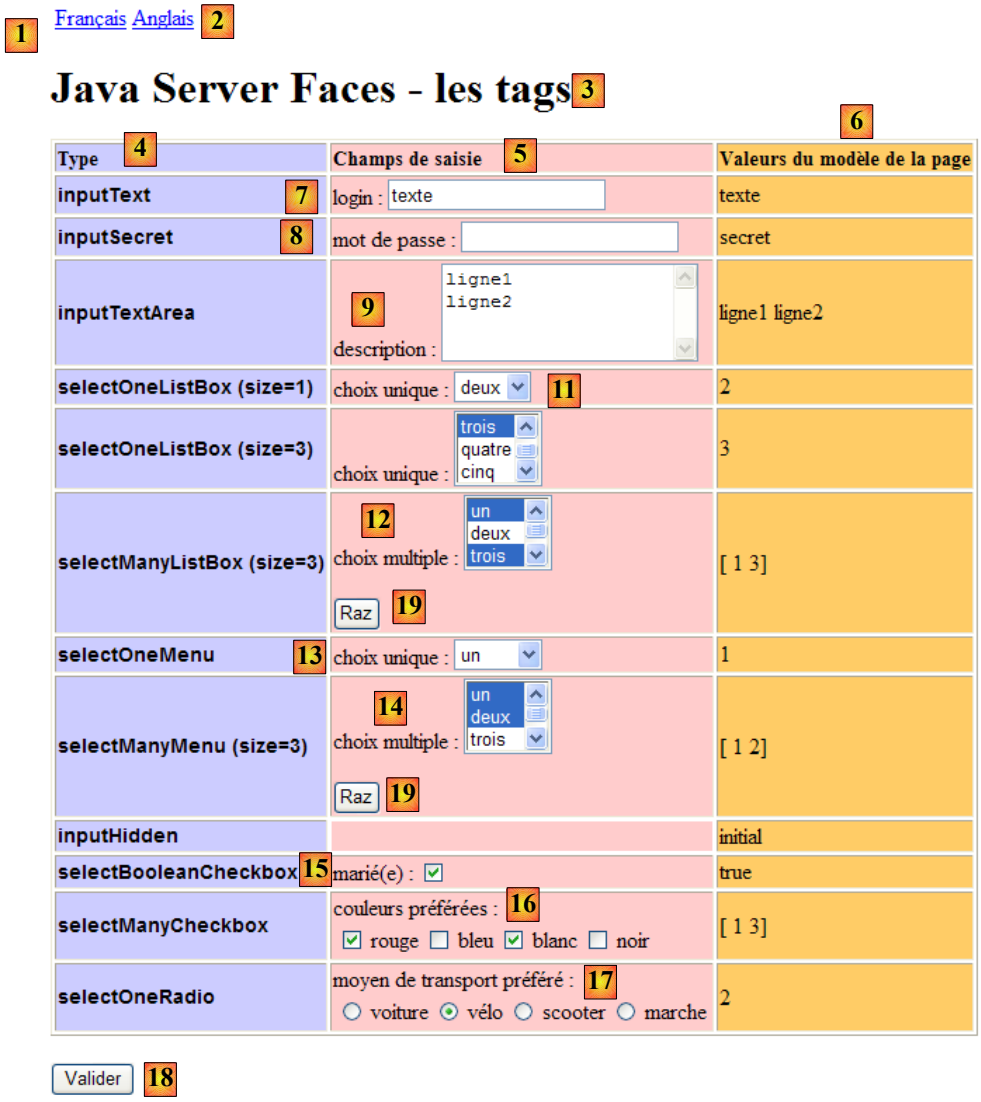
2.5.6 Le fichier des messages [messages.properties]

Les fichiers des messages (cf [5] dans la copie d'écran du projet) sont les suivants :

[**messages_fr.properties**]

```
1.  form.langue1=Fran\u00e7ais
2.  form.langue2=Anglais
3.  form.titre=Java Server Faces - les tags
4.  form.headerCol1=Type
5.  form.headerCol2=Champs de saisie
6.  form.headerCol3=Valeurs du modèle de la page
7.  form.loginPrompt=login :
8.  form.passwdPrompt=mot de passe :
9.  form.descPrompt=description :
10. form.selectOneListBox1Prompt=choix unique :
11. form.selectOneListBox2Prompt=choix unique :
12. form.selectManyListBoxPrompt=choix multiple :
13. form.selectOneMenuPrompt=choix unique :
14. form.selectManyMenuPrompt=choix multiple :
15. form.selectBooleanCheckboxPrompt=marié(e) :
16. form.selectManyCheckboxPrompt=couleurs préférées :
17. form.selectOneRadioPrompt=moyen de transport préféré :
18. form.submitText=Valider
19. form.buttonRazText=Raz
```

Ces messages sont affichés aux endroits suivants de la page :

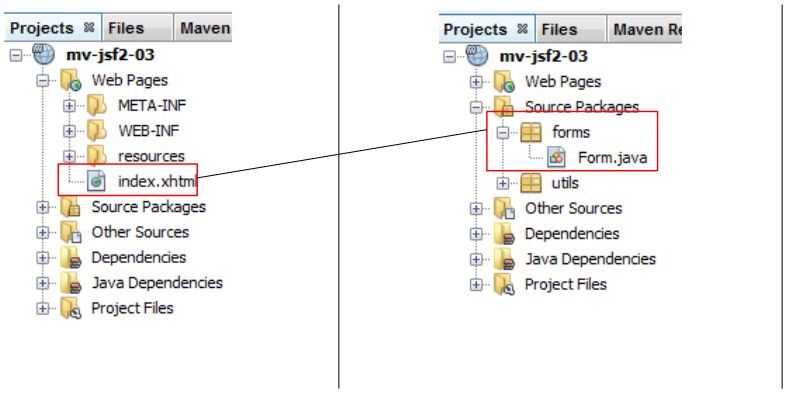


La version anglaise des messages est la suivante :

[messages_en.properties]

```
1.  form.langue1=French
2.  form.langue2=English
3.  form.titre=Java Server Faces - the tags
4.  form.headerCol1=Input Type
5.  form.headerCol2=Input Fields
6.  form.headerCol3=Page Model Values
7.  form.loginPrompt=login :
8.  form.passwdPrompt=password :
9.  form.descPrompt=description :
10. form.selectOneListBox1Prompt=unique choice :
11. form.selectOneListBox2Prompt=unique choice :
12. form.selectManyListBoxPrompt=multiple choice :
13. form.selectOneMenuPrompt=unique choice :
14. form.selectManyMenuPrompt=multiple choice :
15. form.selectBooleanCheckboxPrompt=married :
16. form.selectManyCheckboxPrompt=preferred colors :
17. form.selectOneRadioPrompt=preferred transport means :
18. form.submitText=Submit
19. form.buttonRazText=Reset
```

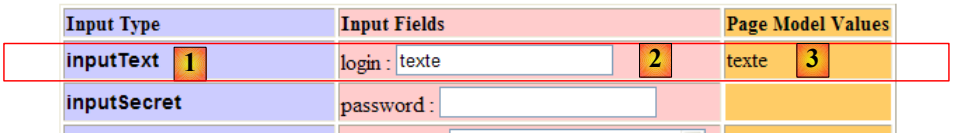
2.5.7 Le modèle [Form.java] de la page [index.xhtml]



Dans le projet ci-dessus, la classe [Form.java] va servir de modèle ou **backing bean** à la page JSF [index.xhtml]. Illustrons cette notion de modèle avec un exemple tiré de la page [index.xhtml] :

```
1.  <!-- ligne 1 -->
2.            <h:outputText value="inputText"  styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.loginPrompt']}"/>
5.              <h:inputText id="inputText" value="#{form.inputText}"/>
6.            </h:panelGroup>
7.            <h:outputText value="#{form.inputText}"/>
```

A la demande initiale de la page [index.xhtml], le code ci-dessus génère la ligne 2 du tableau des saisies :



La ligne 2 affiche la zone [1], les lignes 3-6 : la zone [2], la ligne 7 : la zone [3].

Les lignes 5 et 7 utilisent une expression EL faisant intervenir le bean **form** défini dans la classe [Form.java] de la façon suivante :

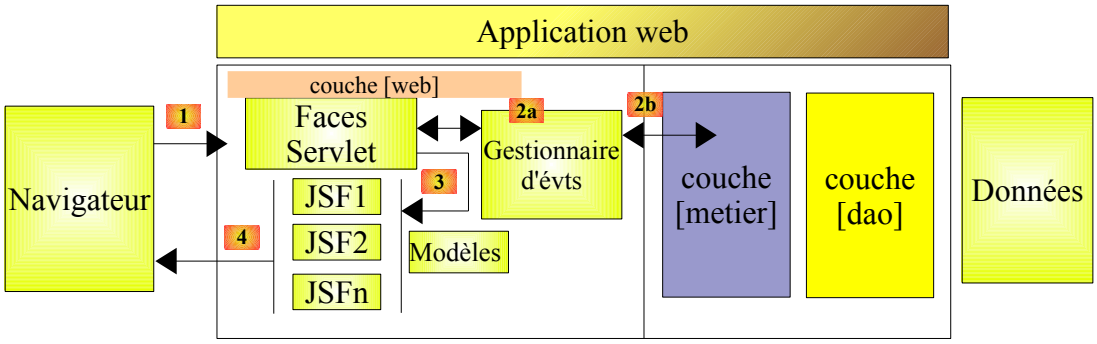
```
1.  package forms;
2.  import javax.faces.bean.RequestScoped;
3.  import javax.faces.bean.ManagedBean;
4.
5.
6.  @ManagedBean
7.  @RequestScoped
8.  public class Form {
```

- la ligne 7 définit un bean sans nom. Celui-ci sera donc le nom de la classe commençant par une minuscule : **form**,
- le bean est de portée **request**. Cela signifie que dans un cycle demande client / réponse serveur, il est instancié lorsque la requête en a besoin et supprimé lorsque la réponse au client a été rendue.

Dans le code ci-dessous de la page [index.xhtml] :

```
1.  <!-- ligne 1 -->
2.            <h:outputText value="inputText"  styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.loginPrompt']}"/>
5.              <h:inputText id="inputText" value="#{form.inputText}"/>
6.            </h:panelGroup>
7.  <h:outputText value="#{form.inputText}"/>
```

les lignes 5 et 7 utilisent la valeur **inputText** du bean **form**. Pour comprendre les liens qui unissent une page **P** à son modèle **M**, il faut revenir au cycle demande client / réponse serveur qui caractérise une application web :



Il faut distinguer le cas où la page **P** est envoyée en réponse au navigateur (étape 4), par exemple lors de la demande initiale de la page, du cas où l'utilisateur ayant provoqué un événement sur la page **P**, celui-ci est traité par le contrôleur [Faces Servlet] (étape 1).

On peut distinguer ces deux cas en les regardant du point de vue du navigateur :

1. lors de la demande initiale de la page, le navigateur fait une opération GET sur l'URL de la page,
2. lors de la soumission des valeurs saisies dans la page, le navigateur fait une opération POST sur l'URL de la page.

Dans les deux cas, c'est la même URL qui est demandée. Selon la nature de la demande GET ou POST du navigateur, le traitement de la requête va différer.

[cas 1 – demande initiale de la page P]

Le navigateur demande l'URL de la page avec un GET. Le contrôleur [Faces Servlet] va passer directement à l'étape [4] de rendu de la réponse et la page [index.xhtml] va être envoyée au client. Le contrôleur JSF va demander à chaque balise de la page de s'afficher. Prenons l'exemple de la ligne 5 du code de [index.xhtml] :

```
<h:inputText id="inputText" value="#{form.inputText}"/>
```

La balise JSF **<h:inputText value="valeur"/>** donne naissance à la balise HTML **<input type="text" value="valeur"/>**. La classe chargée de traiter cette balise rencontre l'expression **#{form.inputText}** qu'elle doit évaluer :

- si le bean **form** n'existe pas encore, il est créé par instanciation de la classe **forms.Form,**
- l'expression **#{form.inputText}** est évaluée par appel à la méthode **form.getInputText()**,
- le texte **<input id="formulaire:inputText" type="text" name="formulaire:inputText" value="texte" />** est inséré dans le flux HTML qui va être envoyé au client si on imagine que la méthode **form.getInputText()** a rendu la chaîne **"texte"**. JSF va par ailleurs donner un nom (name) au composant HTML mis dans le flux. Ce nom est construit à partir des identifiants **id** du composant JSF analysé et ceux de ses composants parents, ici la balise <h:form id="formulaire"/>.

On retiendra que si dans une page **P**, on utilise l'expression **#{M.champ}** où **M** est le bean modèle de la page **P**, celui-ci doit disposer de la méthode publique **getChamp()**. Le type rendu par cette méthode doit pouvoir être converti en type **String**. Un modèle **M** possible et fréquent est le suivant :

```
1.  private T champ;
2.  public T getChamp(){
3.          return champ;
4.  }
```

où *T* est un type qui peut être converti en type *String*, éventuellement avec une méthode **toString**.

Toujours dans le cas de l'affichage de la page **P**, le traitement de la ligne :

```
<h:outputText value="#{form.inputText}"/>
```

sera analogue et le flux HTML suivant sera créé :

```
texte
```

De façon interne au serveur, la page **P** est représentée comme un arbre de composants, image de l'arbre des balises de la page envoyée au client. Nous appellerons **vue** ou **état de la page**, cet arbre. Cet état est mémorisé. Il peut l'être de deux façons selon une configuration faite dans le fichier [web.xml] de l'application :

```
1.  <web-app ...>
2.  ...
3.    <context-param>
4.      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
5.      <param-value>client</param-value>
6.    </context-param>
7.    <servlet>
8.      <servlet-name>Faces Servlet</servlet-name>
9.      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.     <load-on-startup>1</load-on-startup>
11.   </servlet>
12. ...
13. </web-app>
```

Les lignes 7-11 définissent le contrôleur [Faces Servlet]. Celui-ci peut être configuré par différentes balises <context-param> dont celle des lignes 3-6 qui indique que l'état d'une page doit être sauvegardé sur le client (le navigateur). L'autre valeur possible, ligne 5, est **server** pour indiquer une sauvegarde sur le serveur. C'est la valeur par défaut.

Lorsque l'état d'une page est sauvegardé sur le client, le contrôleur JSF ajoute à chaque page HTML qu'il envoie, un champ caché dont la valeur est l'état actuel de la page. Ce champ caché a la forme suivante :

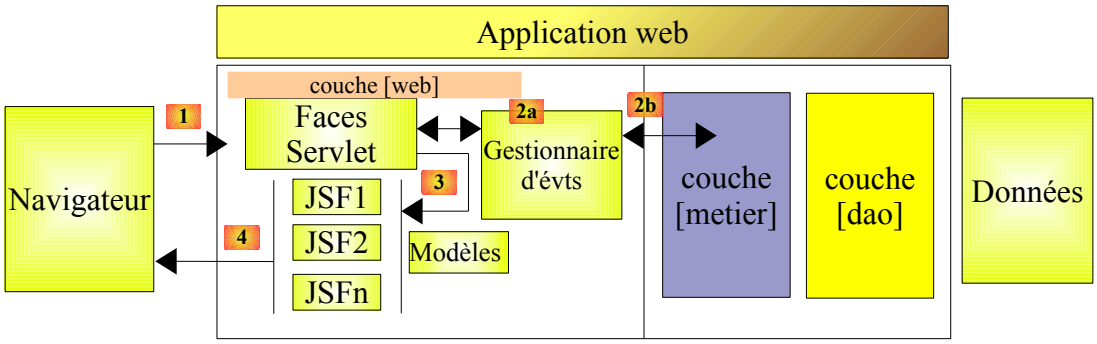
```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="H4sIAAAAAAAAANV...Bnoz8dqAAA=" />
```

Sa valeur représente sous forme codée, l'état de la page envoyée au client. Ce qu'il est important de comprendre, c'est que ce champ caché fait partie du formulaire de la page et fera donc partie des valeurs postées par le navigateur lors de la validation du formulaire. A partir de ce champ caché, le contrôleur JSF est capable de restaurer la vue **telle qu'elle a été envoyée au client**.

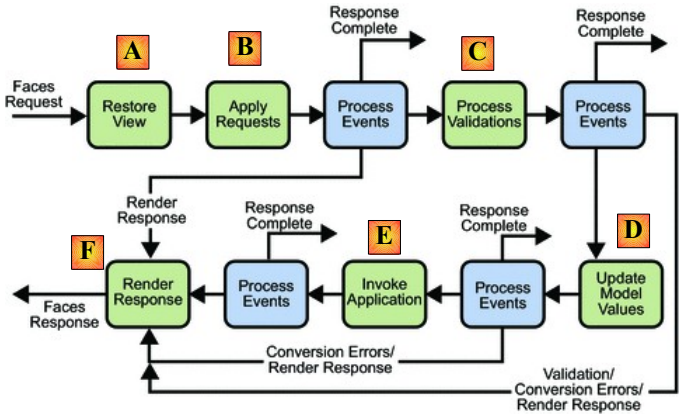
Lorsque l'état d'une page est sauvegardé sur le serveur, l'état de la page envoyée au client est sauvegardé dans la **session** de celui-ci. Lorsque le navigateur client va poster les valeurs saisies dans le formulaire, il va envoyer également son jeton de session. A partir de celui-ci, le contrôleur JSF retrouvera l'état de la page envoyée au client et la restaurera.

L'état d'une page JSF peut nécessiter plusieurs centaines d'octets pour être codé. Comme cet état est maintenu pour chaque utilisateur de l'application, on peut rencontrer des problèmes mémoire s'il y a un grand nombre d'utilisateurs. Pour cette raison, nous avons choisi ici de sauvegarder l'état de la page sur le client (cf [web.xml] paragraphe 2.5.4, page 59).

[**cas 2 – traitement de la page P**]



On est à l'étape [1] ci-dessus où le contrôleur [Faces Servlet] va recevoir une requête POST du navigateur client à qui il a envoyé précédemment la page [index.xhtml]. On est en présence du traitement d'un événement de la page. Plusieurs étapes vont se dérouler avant même que l'événement ne puisse être traité en [2a]. Le cycle de traitement d'une requête POST par le contrôleur JSF est le suivant :



- en [A], grâce au champ caché *javax.faces.ViewState* la vue initialement envoyée au navigateur client est reconstituée. Ici, les composants de la page retrouvent la valeur qu'ils avaient dans la page envoyée. Notre composant **inputText** retrouve sa valeur "**texte**",
- en [B], les valeurs postées par le navigateur client sont utilisées pour mettre à jour les composants de la vue. Ainsi si dans le champ de saisie HTML nommé **inputText**, l'utilisateur a tapé "jean", la valeur "jean" remplace la valeur "texte". Désormais la vue reflète la page telle que **l'a modifiée l'utilisateur** et non plus telle qu'elle a été envoyée au navigateur,
- en [C], les valeurs postées sont vérifiées. Supposons que le composant **inputText** précédent soit le champ de saisie d'un âge. Il faudra que la valeur saisie soit un nombre entier. Les valeurs postées par le navigateur sont toujours de type *String*. Leur type final dans le modèle **M** associé à la page **P** peut être tout autre. Il y a alors **conversion** d'un type *String* vers un autre type *T*. Cette conversion peut échouer. Dans ce cas, le cycle demande / réponse est terminé et la page **P** construite en [B] est renvoyée au navigateur client avec des messages d'erreur si l'auteur de la page **P** les a prévus. On notera que l'utilisateur retrouve la page telle qu'il l'a saisie, sans effort de la part du développeur. Dans une autre technologie, telle que JSP, le développeur doit reconstruire lui-même la page **P** avec les valeurs saisies par l'utilisateur. La valeur d'un composant peut subir également un processus de **validation**. Toujours avec l'exemple du composant **inputText** qui est le champ de saisie d'un âge, la valeur saisie devra être non seulement un nombre entier mais un nombre entier compris dans un intervalle [1,N]. Si la valeur saisie passe l'étape de la **conversion**, elle peut ne pas passer l'étape de la **validation**. Dans ce cas, là également le cycle demande / réponse est terminé et la page **P** construite en [B] est renvoyée au navigateur client,
- en [D], si tous les composants de la page **P** passent l'étape de conversion et de validation, leurs valeurs vont être affectées au modèle **M** de la page **P**. Si la valeur du champ de saisie généré à partir de la balise suivante :

```
<h:inputText value="#{form.inputText}"/>
```

  est "jean", alors cette valeur sera affectée au modèle **form** de la page par exécution du code *form.setInputText("jean")*. On retiendra que dans le modèle **M** de la page **P**, les champs privés de **M** qui mémorisent la valeur d'un champ de saisie de **P** doivent avoir une méthode **set,**
- une fois le modèle **M** de la page **P** mis à jour par les valeurs postées, l'événement qui a provoqué le POST de la page **P** peut être traité. C'est l'étape [E]. On notera que si le gestionnaire de cet événement appartient au bean **M**, il a accès aux valeurs du formulaire **P** qui ont été stockées dans les champs de ce même bean.
- l'étape [E] va rendre au contrôleur JSF, une clé de navigation. Dans nos exemples, ce sera toujours le nom de la page XHTML à afficher, sans le suffixe *.xhtml*. C'est l'étape [F]. Une autre façon de faire est de renvoyer une clé de navigation qui sera recherchée dans le fichier [faces-config.xml]. Nous avons décrit ce cas.

Nous retiendrons de ce qui précède que :

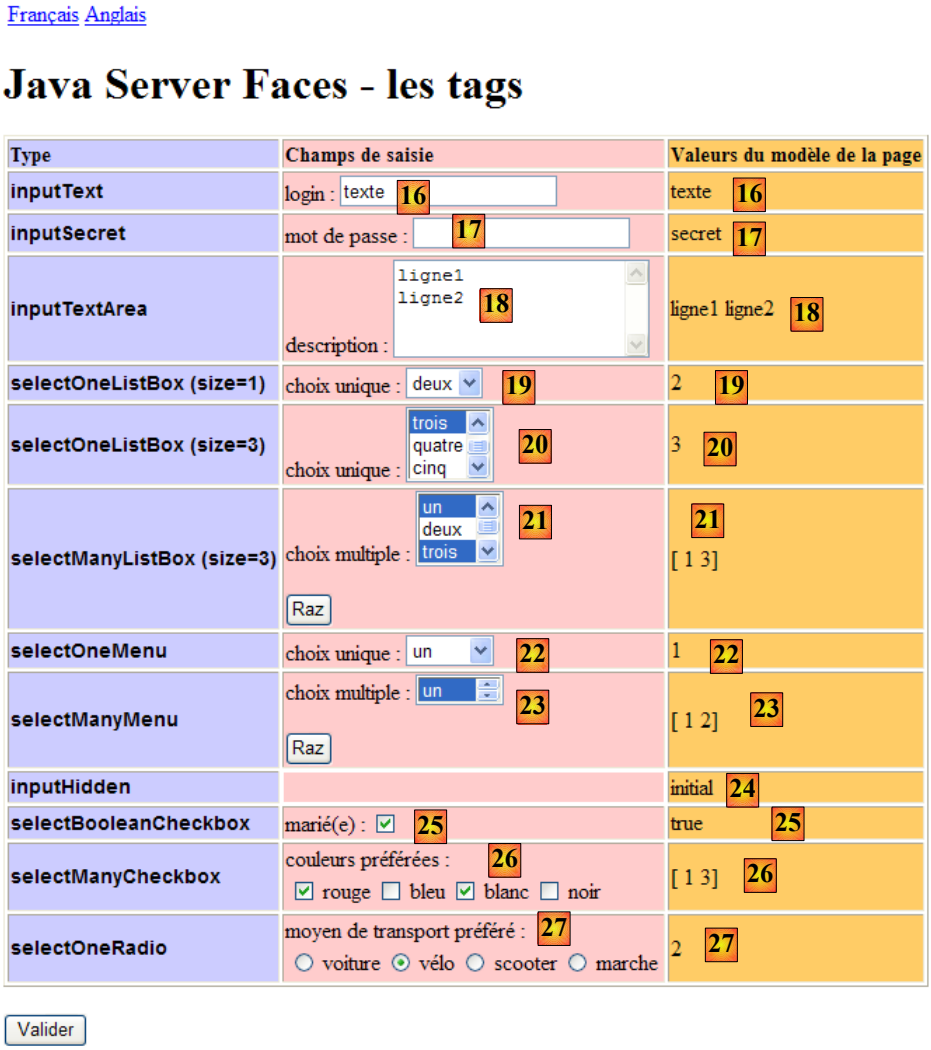
- une page **P** affiche les champs **C** de son modèle **M** avec des méthodes **[M].getC(),**
- les champs **C** du modèle **M** d'une page **P** sont initialisés avec les valeurs saisies dans la page **P** à l'aide des méthodes **[M].setC(saisie)**. Dans cette étape, peuvent intervenir des processus de **conversion** et de **validation** susceptibles d'échouer. Dans ce cas, l'événement qui a provoqué le POST de la page **P** n'est pas traité et la page est renvoyée de nouveau au client telle que celui-ci l'a saisie.

Le modèle [Form.java] de la page [index.xhtml] sera le suivant :

```
1.  package forms;
2.
3.  import javax.faces.bean.RequestScoped;
4.  import javax.faces.bean.ManagedBean;
5.
6.
7.  @ManagedBean
```

```
8.  @RequestScoped
9.  public class Form {
10.
11.   /** Creates a new instance of Form */
12.   public Form() {
13.   }
14.
15.   // champs du formulaire
16.   private String inputText="texte";
17.   private String inputSecret="secret";
18.   private String inputTextArea="ligne1\nligne2\n";
19.   private String selectOneListBox1="2";
20.   private String selectOneListBox2="3";
21.   private String[] selectManyListBox=new String[]{"1","3"};
22.   private String selectOneMenu="1";
23.   private String[] selectManyMenu=new String[]{"1","2"};
24.   private String inputHidden="initial";
25.   private boolean selectBooleanCheckbox=true;
26.   private String[] selectManyCheckbox=new String[]{"1","3"};
27.   private String selectOneRadio="2";
28.
29.   // événements
30.   public String submit(){
31.     return null;
32.   }
33.
34.   // getters et setters
35.   ...
36. }
```

Les champs des lignes 16-27 sont utilisés aux endroits suivants du formulaire :



2.5.8 La page [index.xhtml]

La page [index.xhtml] qui génère la vue précédente est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.
7.    <f:view locale="#{changeLocale.locale}">
8.      <h:head>
9.        <title>JSF</title>
10.       <h:outputStylesheet library="css" name="styles.css"/>
11.     </h:head>
12.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
13.       <h:form id="formulaire">
14.         <!-- langues -->
15.         <h:panelGrid columns="2">
16.           <h:commandLink value="#{msg['form.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
17.           <h:commandLink value="#{msg['form.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
18.         </h:panelGrid>
19.         <h1><h:outputText value="#{msg['form.titre']}"/></h1>
20.         <h:panelGrid columnClasses="col1,col2,col3" columns="3" border="1">
21.           <!-- entêtes -->
22.           <h:outputText value="#{msg['form.headerCol1']}" styleClass="entete"/>
23.           <h:outputText value="#{msg['form.headerCol2']}" styleClass="entete"/>
```

```
24.           <h:outputText value="#{msg['form.headerCol3']}" styleClass="entete"/>
25.           <!-- ligne 1 -->
26.           ...
27.           <!-- ligne 2 -->
28.           ...
29.           <!-- ligne 3 -->
30.           ...
31.           <!-- ligne 4 -->
32.           ...
33.           <!-- ligne 5 -->
34.           ...
35.           <!-- ligne 6 -->
36.           ...
37.           <!-- ligne 7 -->
38.           ...
39.           <!-- ligne 8 -->
40.           ...
41.           <!-- ligne 9 -->
42.           ...
43.           <!-- ligne 10 -->
44.           ...
45.           <!-- ligne 11 -->
46.           ...
47.           <!-- ligne 12 -->
48.           ...
49.         </h:panelGrid>
50.         <p>
51.           <h:commandButton type="submit" id="submit" value="#{msg['form.submitText']}"/>
52.         </p>
53.       </h:form>
54.     </h:body>
55.   </f:view>
56. </html>
```

Nous allons étudier successivement les principaux composants de cette page. On notera la structure générale d'un formulaire JSF :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.
7.    <f:view ...>
8.      <h:head>
9.        ...
10.     </h:head>
11.     <h:body ...>
12.       <h:form id="formulaire">
13.         ...
14.         <h:commandButton type="submit" id="submit" value="#{msg['form.submitText']}"/>
15.         ...
16.       </h:form>
17.     </h:body>
18.   </f:view>
19. </html>
```

Les composants d'un formulaire doivent être à l'intérieur d'une balise <**h:form**> (lignes 12-16). La balise <**f:view**> (lignes 7-18) est nécessaire si on internationalise l'application. Par ailleurs, un formulaire doit disposer d'un moyen d'être posté (POST), souvent un lien ou un bouton comme dans la ligne 14. Il peut être posté également par de nombreux événements (changement d'une sélection dans une liste, changement de champ actif, frappe d'un caractère dans un champ de saisie, ...).

2.5.9 Le style du formulaire

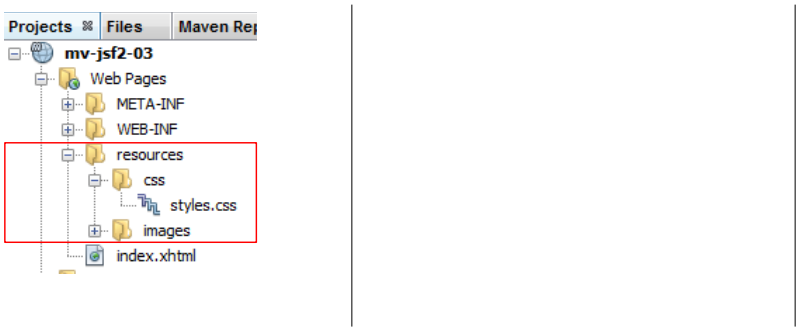
Afin de rendre plus lisibles les colonnes du tableau du formulaire, celui est accompagné d'une feuille de style :

```
1.    <f:view locale="#{changeLocale.locale}">
2.      <h:head>
3.        <title>JSF</title>
4.        <h:outputStylesheet library="css" name="styles.css"/>
5.  </h:head>
```

- ligne 4 : la feuille de style de la page est définie à l'intérieur de la balise HTML <**head**>, par la balise :

```
<h:outputStylesheet library="css" name="styles.css"/>
```

La feuille de style sera cherchée dans le dossier [resources] :



Dans la balise :

```
<h:outputStylesheet library="css" name="styles.css"/>
```

- **library** est le nom du dossier contenant la feuille de style,
- **name** est le nom de la feuille de style.

Voyons une utilisation de cette feuille de style :

```
<h:panelGrid columnClasses="col1,col2,col3" columns="3" border="1">
```

La balise <**h:panelGrid columns="3"**/> définit un tableau à trois colonnes. L'attribut **columnClasses** permet de donner un style à ces colonnes. Les valeurs col1,col2,col3 de l'attribut **columnClasses** désignent les styles respectifs des colonnes 1, 2 et 3 du tableau. Ces styles sont cherchés dans le feuille de style de la page :

```
1.  .info{
2.     font-family: Arial,Helvetica,sans-serif;
3.     font-size: 14px;
4.     font-weight: bold
5.  }
6.
7.  .col1{
8.     background-color: #ccccff
9.  }
10.
11. .col2{
12.    background-color: #ffcccc
13. }
14.
15. .col3{
16.    background-color: #ffcc66
17. }
18.
19. .entete{
20.    font-family: 'Times New Roman',Times,serif;
21.    font-size: 14px;
22.    font-weight: bold
23. }
```

- lignes 7-9 : le style nommé **col1,**
- lignes 11-13 : le style nommé **col2,**
- lignes 15-17 : le style nommé **col3,**

Ces trois styles définissent la couleur de fond de chacune des colonnes.

- lignes 19-23 : le style **entete** sert à définir le style des textes de la première ligne du tableau :

```
            <!-- entêtes -->
  <h:outputText value="#{msg['form.headerCol1']}" styleClass="entete"/>
```

```
<h:outputText value="#{msg['form.headerCol2']}" styleClass="entete"/>
<h:outputText value="#{msg['form.headerCol3']}" styleClass="entete"/>
```

- lignes 1-5 : le style **info** sert à définir le style des textes de la première colonne du tableau :

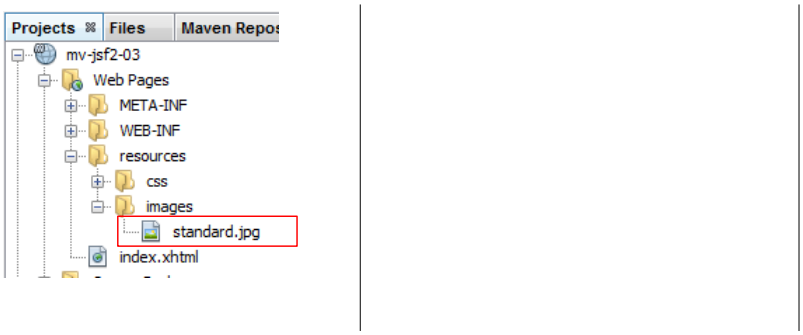
```
  <!-- ligne 1 -->
<h:outputText value="inputText"  styleClass="info"/>
```

Nous insisterons peu sur l'utilisation des feuilles de style car celles-ci méritent à elles seules un livre et que par ailleurs leur élaboration en est souvent confiée à des spécialistes. Néanmoins, nous avons souhaité en utiliser une, minimaliste, afin de rappeler que leur usage est indispensable.

Regardons maintenant comment a été définie l'image de fond de la page :

```
<h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
```

L'image de fond est fixée par l'attribut **style** de la balise <**h:body**>. Cet attribut permet de fixer des éléments de style. L'image de fond est dans le dossier [*resources/images/standard.jpg*] :



Cette image est obtenue via l'URL [/mv-jsf2-03/resources/images/standard.jpg]. On pourrait donc écrire :

```
<h:body style="background-image: url('mv-jsf2-03/resources/images/standard.jpg');">
```

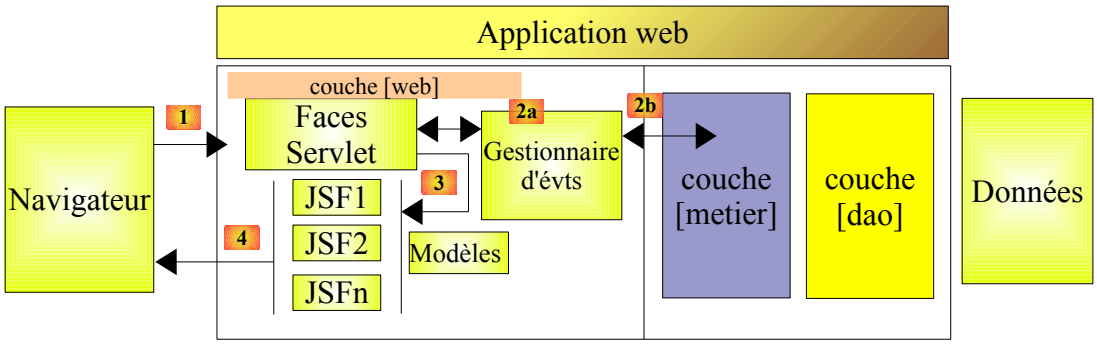
/mv-jsf2-03 est le contexte de l'application. Ce contexte est fixé par l'administrateur du serveur web et peut donc changer. Ce contexte peut être obtenu par l'expression EL *${request.contextPath}*. Aussi préfèrera-t-on l'attribut **style** suivant :

```
style="background-image: url('${request.contextPath}/resources/images/standard.jpg');"
```

qui sera valide quelque soit le contexte.

2.5.10 Les deux cycles demande client / réponse serveur d'un formulaire

Revnons sur ce qui a déjà été expliqué page 62 dans un cas général et appliquons-le au formulaire étudié. Celui-ci sera testé dans l'environnement JSF classique :



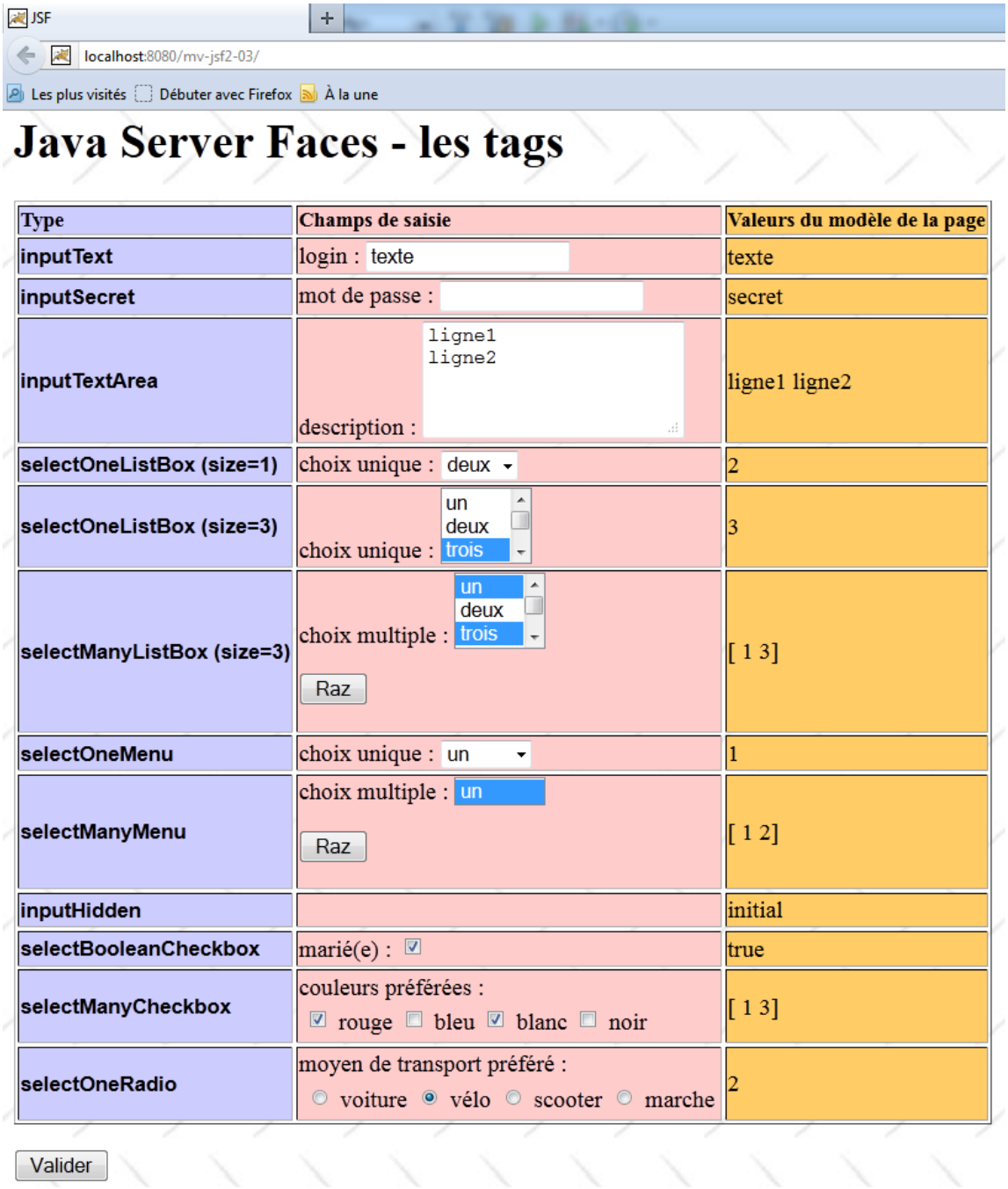
Ici, il n'y aura pas de gestionnaires d'événements ni de couche [métier]. Les étapes [2x] n'existeront donc pas. On distinguera le cas où le formulaire **F** est demandée initialement par le navigateur du cas où l'utilisateur ayant provoqué un événement dans le formulaire **F**, celui-ci est traité par le contrôleur [Faces Servlet]. Il y a deux cycles demande client / réponse serveur qui sont différents.

- le premier correspondant à la demande initiale de la page est provoqué par une opération **GET** du navigateur sur l'URL du formulaire,
- le second correspondant à la soumission des valeurs saisies dans la page est provoqué par une opération **POST** sur cette même URL.

Selon la nature de la demande GET ou POST du navigateur, le traitement de la requête par le contrôleur [Faces Servlet] diffère.

[**cas 1 – demande initiale du formulaire F**]

Le navigateur demande l'URL de la page avec un GET. Le contrôleur [Faces Servlet] va passer directement à l'étape [4] de rendu de la réponse. Le formulaire [index.xhtml] va être initialisé par son modèle [Form.java] et être envoyé au client qui reçoit la vue suivante :



Les échanges HTTP client / serveur sont les suivants à cette occasion :

Demande HTTP du client :

```
1.  GET /mv-jsf2-03/ HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0
4.  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5.  Accept-Language: fr,fr-fr;q=0.8,en;q=0.6,en-us;q=0.4,es;q=0.2
6.  Accept-Encoding: gzip, deflate
7.  DNT: 1
8.  Connection: keep-alive
```

Ligne 1, on voit le GET du navigateur.

Réponse HTTP du serveur :

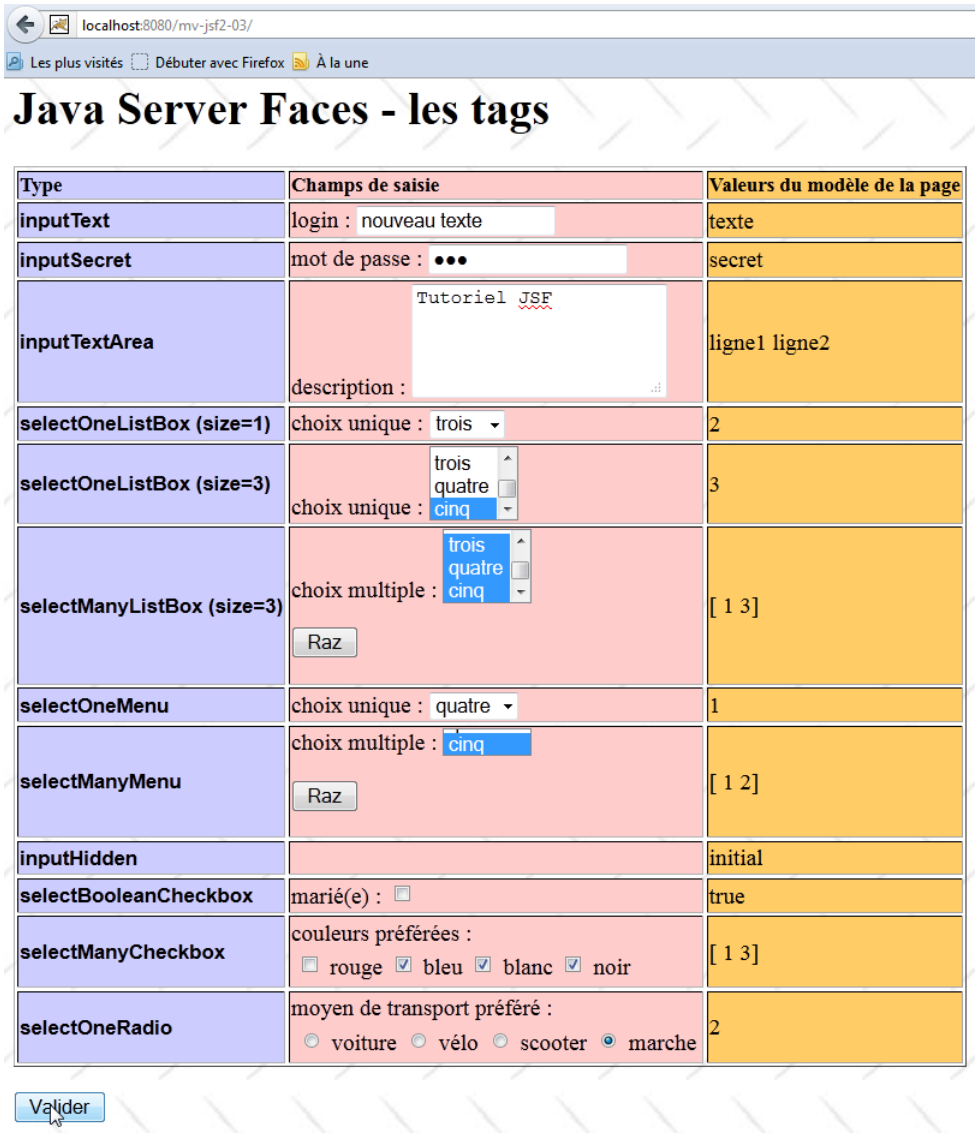
```
1.  HTTP/1.1 200 OK
```

```
2.  Server: Apache-Coyote/1.1
3.  X-Powered-By: JSF/2.0
4.  Set-Cookie: JSESSIONID=F6E66136BF00EEE026ADAB1BBEBFD587; Path=/mv-jsf2-03/; HTTPOnly
5.  Content-Type: text/html;charset=UTF-8
6.  Content-Length: 7371
7.  Date: Tue, 15 May 2012 09:04:57 GMT
```

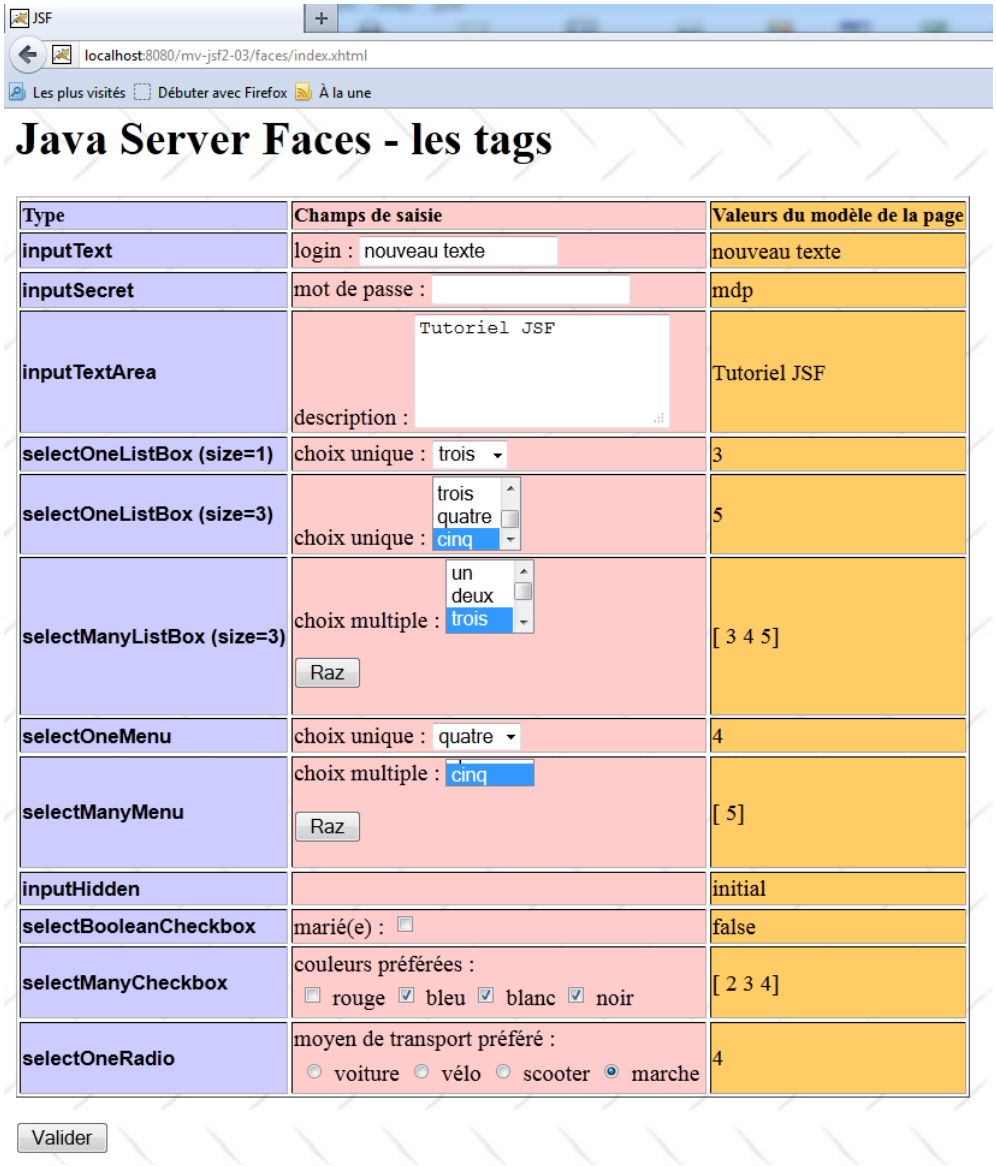
Non montré ici, la ligne 7 est suivie d'une ligne vide et du code HTML du formulaire. C'est ce code que le navigateur interprète et affiche.

[**cas 2 – traitement des valeurs saisies dans le formulaire F**]

L'utilisateur remplit le formulaire et le valide par le bouton [Valider]. Le navigateur demande alors l'URL du formulaire avec un POST. Le contrôleur [Faces Servlet] traite cette requête, met à jour le modèle [Form.java] du formulaire [index.xhtml], et renvoie de nouveau le formulaire [index.xhtml] mis à jour par ce nouveau modèle. Examinons ce cycle sur un exemple :



Ci-dessus, l'utilisateur a fait ses saisies et les valide. Il reçoit en réponse la vue suivante :



Les échanges HTTP client / serveur sont les suivants à cette occasion :

Demande HTTP du client :

```
1.  POST /mv-jsf2-03/faces/index.xhtml HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0
4.  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5.  Accept-Language: fr,fr-fr;q=0.8,en;q=0.6,en-us;q=0.4,es;q=0.2
6.  Accept-Encoding: gzip, deflate
7.  DNT: 1
8.  Connection: keep-alive
9.  Referer: http://localhost:8080/mv-jsf2-03/faces/index.xhtml
10. Cookie: JSESSIONID=374CC5F1D2ACAC182A5747A443651E36
11. Content-Type: application/x-www-form-URLencoded
12. Content-Length: 1543
13.
14. formulaire=formulaire&formulaire%3AinputText=nouveau+texte&formulaire%3AinputSecret=mdp&formulaire
    %3AinputTextArea=Tutoriel+JSF%0D%0A&formulaire%3AselectOneListBox1=3&formulaire
    %3AselectOneListBox2=5&formulaire%3AselectManyListBox=3&formulaire
    %3AselectManyListBox=4&formulaire%3AselectManyListBox=5&formulaire%3AselectOneMenu=4&formulaire
    %3AselectManyMenu=5&formulaire%3AinputHidden=initial&formulaire%3AselectManyCheckbox=2&formulaire
    %3AselectManyCheckbox=3&formulaire%3AselectManyCheckbox=4&formulaire%3AselectOneRadio=4&formulaire
    %3Asubmit=Valider&javax.faces.ViewState=H4sIAAAAAAAAAJVUT0g...P4BKm1E4F0FAAA
```

En ligne 1, le POST fait par le navigateur. En ligne 14, les valeurs saisies par l'utilisateur. On peut par exemple y découvrir le texte mis dans le champ de saisie :

```
formulaire%3AinputText=nouveau+texte
```

Dans la ligne 14, le champ caché *javax.faces.ViewState* a été posté. Ce champ représente, sous forme codée, l'état du formulaire tel qu'il a été envoyé initialement au navigateur lors de son GET initial.

Réponse HTTP du serveur :

```
1.  HTTP/1.1 200 OK
2.  Server: Apache-Coyote/1.1
3.  X-Powered-By: JSF/2.0
4.  Content-Type: text/html;charset=UTF-8
5.  Content-Length: 7299
6.  Date: Tue, 15 May 2012 09:37:17 GMT
```

Non montré ici, la ligne 6 est suivie d'une ligne vide et du code HTML du formulaire mis à jour par son nouveau modèle issu du POST.

Nous examinons maintenant les différentes composantes de ce formulaire.

2.5.11 Balise <h:inputText>

La balise <**h:inputText**> génère une balise HTML <**input type="text" ...**>.

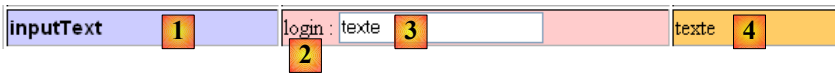
Considérons le code suivant :

```
1.          <!-- ligne 1 -->
2.          <h:outputText value="inputText"  styleClass="info"/>
3.          <h:panelGroup>
4.            <h:outputText value="#{msg['form.loginPrompt']}"/>
5.            <h:inputText id="inputText" value="#{form.inputText}"/>
6.          </h:panelGroup>
7.  <h:outputText value="#{form.inputText}"/>
```

et son modèle [Form.java] :

```
1.    private String inputText="texte";
2.
3.    public String getInputText() {
4.      return inputText;
5.    }
6.
7.    public void setInputText(String inputText) {
8.      this.inputText = inputText;
9.  }
```

Lorsque la page [index.html] est demandée la première fois, la page obtenue est la suivante :



- la ligne 2 du code XHTML génère [1],
- la balise <*h:panelGroup*> (lignes 3-6) permet de regrouper plusieurs éléments dans une même cellule du tableau généré par la balise <*h:panelGrid*> de la ligne 20 du code complet de la page, (cf page 67). Le texte [2] est généré par la ligne 4. Le champ de saisie [3] est généré par la ligne [5]. Ici, la méthode **getInputText** de [Form.java] (lignes 3-5 du code Java) a été utilisée pour générer le texte du champ de saisie,
- la ligne 7 du code XHTML génère [4]. C'est de nouveau la méthode **getInputText** de [Form.java] qui est utilisée pour générer le texte [4].

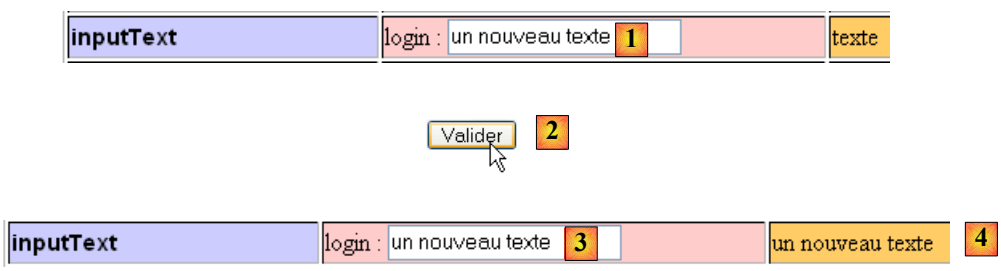
Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">inputText</span></td>
3.  <td class="col2">login : <input id="formulaire:inputText" type="text" name="formulaire:inputText"
    value="texte" /></td>
```

```
4.  <td class="col3">texte</td>
5.  </tr>
```

Les balises HTML <tr> et <td> sont générées par la balise <*h:panelGrid*> utilisée pour générer le tableau du formulaire.

Maintenant, ci-dessous, saisissons une valeur dans le champ de saisie [1] et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3, 4] :



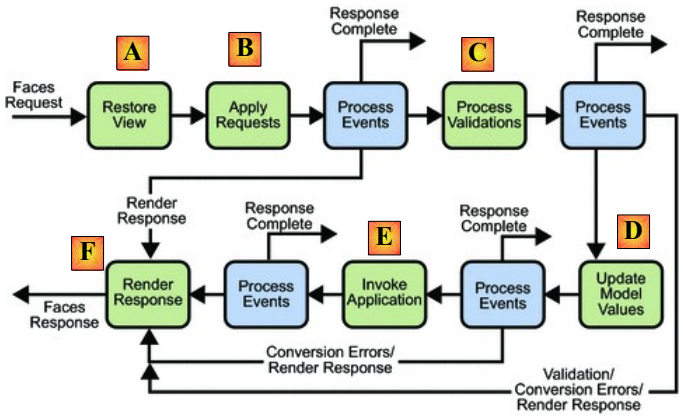
La valeur du champ [1] est postée de la façon suivante :

```
formulaire%3AinputText=nouveau+texte
```

En [2], le formulaire est validé avec le bouton suivant :

```
          <h:commandButton id="submit" type="submit" value="#{msg['form.submitText']}"/>
```

La balise <*h:commandButton*> n'a pas d'attribut **action**. Dans ce cas, aucun gestionnaire d'événement n'est invoqué ni aucune règle de navigation. Après traitement, la même page est renvoyée. Revoyons son cycle de traitement :



- en [A] la page **P** est restaurée telle qu'elle avait été envoyée. Cela signifie que le composant d'id **inputText** est restauré avec sa valeur initiale **"texte",**
- en [B], les valeurs postées par le navigateur (saisies par l'utilisateur) sont affectées aux composants de la page P. Ici, le composant d'id **inputText** reçoit la valeur **"un nouveau texte",**
- en [C], les conversions et validations ont lieu. Ici, il n'y en a aucune. Dans le modèle **M**, le champ associé au composant d'id **inputText** est le suivant :

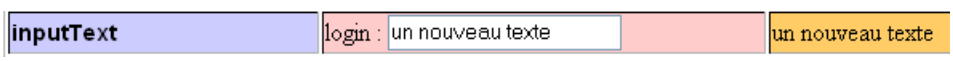
  ```
  private String inputText="texte";
  ```

  Comme les valeurs saisies sont de type *String*, il n'y a pas de conversion à faire. Par ailleurs, aucune règle de validation n'a été créée. Nous en construirons ultérieurement.
- en [D], les valeurs saisies sont affectées au modèle. Le champ **inputText** de [Form.java] reçoit la valeur **"un nouveau texte"**,
- en [E], rien n'est fait car aucun gestionnaire d'événement n'a été associé au bouton [Valider].

- en [F], la page **P** est de nouveau envoyée au client car le bouton [Valider] n'a pas d'attribut **action**. Les lignes suivantes de [index.xhtml] sont alors exécutées :

```
1.            <!-- ligne 1 -->
2.          <h:outputText value="inputText"  styleClass="info"/>
3.          <h:panelGroup>
4.            <h:outputText value="#{msg['form.loginPrompt']}"/>
5.            <h:inputText id="inputText" value="#{form.inputText}"/>
6.          </h:panelGroup>
7.  <h:outputText value="#{form.inputText}"/>
```

Les lignes 5 et 7 utilisent la valeur du champ **inputText** du modèle qui est désormais "un nouveau texte". D'où l'affichage obtenu :



2.5.12 Balise <h:inputSecret>

La balise <**h:inputSecret**> génère une balise HTML <**input type="password" ...**>. C'est un champ de saisie analogue à celui de la balise JSF <h:inputText> si ce n'est que chaque caractère tapé par l'utilisateur est remplacé visuellement par un caractère *.

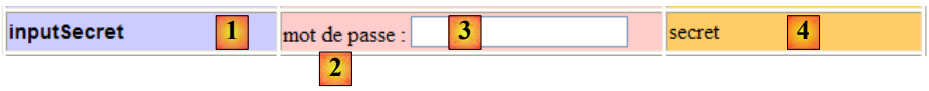
Considérons le code suivant :

```
1.            <!-- ligne 2 -->
2.          <h:outputText value="inputSecret"  styleClass="info"/>
3.          <h:panelGroup>
4.            <h:outputText value="#{msg['form.passwdPrompt']}"/>
5.            <h:inputSecret id="inputSecret" value="#{form.inputSecret}"/>
6.          </h:panelGroup>
7.  <h:outputText value="#{form.inputSecret}"/>
```

et son modèle dans [Form.java] :

```
1.  private String inputSecret="secret";
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



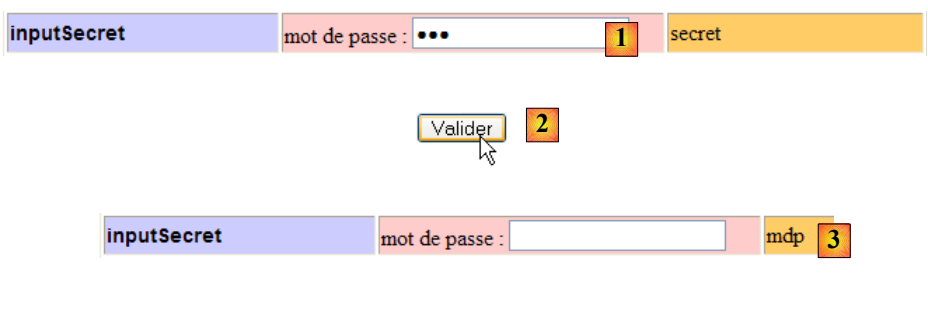
- la ligne 2 du code XHTML génère [1]
- le texte [2] est généré par la ligne 4. Le champ de saisie [3] est généré par la ligne [5]. Normalement, la méthode **getInputSecret** de [Form.java] aurait du être utilisée pour générer le texte du champ de saisie. Il y a une exception lorsque celui-ci est de type " mot de passe ". La balise <*h:inputSecret*> ne sert qu'à lire une saisie, pas à l'afficher.
- la ligne 7 du code XHTML génère [4]. Ici la méthode **getInputSecret** de [Form.java] a été utilisée pour générer le texte [4] (cf ligne 1 du code Java).

Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">inputSecret</span></td>
3.  <td class="col2">mot de passe : <input id="formulaire:inputSecret" type="password"
    name="formulaire:inputSecret" value="" /></td>
4.  <td class="col3">secret</td>
5.  </tr>
```

- ligne 3 : la balise HTML <**input type= "password " .../**> générée par la balise JSF <**h:inputSecret**>

Maintenant, ci-dessous, saisissons une valeur dans le champ de saisie [1] et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3] :



La valeur du champ [1] est postée de la façon suivante :

```
formulaire%3AinputSecret=mdp
```

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. Le champ **inputSecret** de [Form.java] a reçu alors la valeur **mdp**. Parce que le formulaire [index.xhtml] n'a défini aucune règle de navigation, ni aucun gestionnaire d'événement, il est réaffiché après la mise à jour de son modèle. On retombe alors dans l'affichage fait à la demande initiale de la page [index.xhtml] où simplement le champ **inputSecret** du modèle a changé de valeur [3].

2.5.13 Balise <h:inputTextArea>

La balise <**h:inputTextArea**> génère une balise HTML **<textarea ...>texte</textarea>**. C'est un champ de saisie analogue à celui de la balise JSF <*h:inputText*> si ce n'est qu'ici, on peut taper plusieurs lignes de texte.

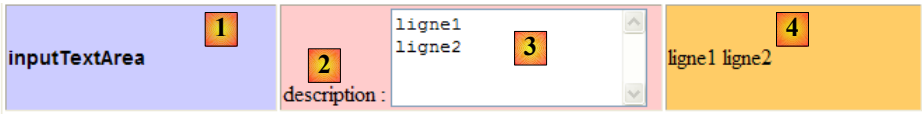
Considérons le code suivant :

```
1.            <!-- ligne 3 -->
2.            <h:outputText value="inputTextArea" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.descPrompt']}"/>
5.              <h:inputTextarea id="inputTextArea" value="#{form.inputTextArea}" rows="4"/>
6.            </h:panelGroup>
7.  <h:outputText value="#{form.inputTextArea}"/>
```

et son modèle dans [Form.java] :

```
1.  private String inputTextArea="ligne1\nligne2\n";
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



- la ligne 2 du code XHTML génère [1],
- le texte [2] est généré par la ligne 4. Le champ de saisie [3] est généré par la ligne [5]. Son contenu a été généré par appel à la méthode **getInputTextArea** du modèle, qui a rendu la valeur définie en ligne 1 du code Java ci-dessus,
- la ligne 7 du code XHTML génère [4]. Ici la méthode **getInputTextArea** de [Form.java] a été de nouveau utilisée. La chaîne " ligne1\nligne2 " contenait des sauts de ligne \n. Ils y sont toujours. Mais insérés dans un flux HTML, ils sont affichés comme des espaces par les navigateurs. La balise HTML <**textarea**> qui affiche [3], elle, interprète correctement les sauts de ligne.

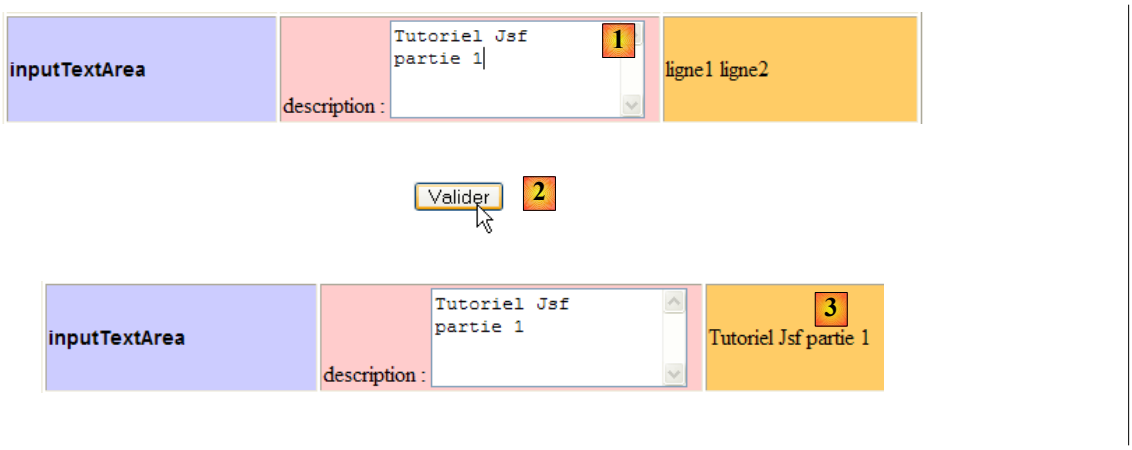
Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">inputTextArea</span></td>
```

```
3.  <td class="col2">description : <textarea id="formulaire:inputTextArea"
    name="formulaire:inputTextArea" rows="4">ligne1
4.  ligne2
5.  </textarea></td>
6.  <td class="col3">ligne1
7.  ligne2
8.  </td>
9.  </tr>
```

- lignes 3-5 : la balise HTML *<textarea>...</textarea>* générée par la balise JSF *<h:inputTextArea>*

Maintenant, ci-dessous, saisissons une valeur dans le champ de saisie [1] et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3] :



La valeur du champ [1] postée est la suivante :

```
formulaire%3AinputTextArea=Tutoriel+JSF%0D%0Apartie+1%0D%0A
```

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. Le champ **textArea** de [Form.java] a reçu alors la valeur **" Tutoriel JSF\npartie1 "**. Le réaffichage de [index.xhtml] montre que champ **textArea** du modèle a bien été mis à jour [3].

2.5.14 Balise <h:selectOneListBox>

La balise <**h:selectOneListBox**> génère une balise HTML **<select>...</select>**. Visuellement, elle génère une liste déroulante ou une liste avec ascenseur.

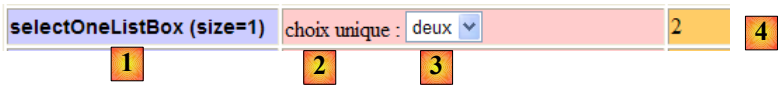
Considérons le code suivant :

```
1.  <!-- ligne 4 -->
2.            <h:outputText value="selectOneListBox (size=1)" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectOneListBox1Prompt']}"/>
5.              <h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}"
    size="1">
6.                <f:selectItem itemValue="1" itemLabel="un"/>
7.                <f:selectItem itemValue="2" itemLabel="deux"/>
8.                <f:selectItem itemValue="3" itemLabel="trois"/>
9.              </h:selectOneListbox>
10.           </h:panelGroup>
11.           <h:outputText value="#{form.selectOneListBox1}"/>
```

et son modèle dans [Form.java] :

```
1.  private String selectOneListBox1="2";
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



- la ligne 2 du code XHTML génère [1]
- le texte [2] est généré par la ligne 4. La liste déroulante [3] est générée par les lignes [5-9]. C'est la valeur de l'attribut **size= "1"** qui fait que la liste n'affiche qu'un élément. Si cet attribut est manquant, la valeur par défaut de l'attribut **size** est **1**. Les éléments de la liste ont été générés par les balises <**f:selectItem**> des lignes 6-8. Ces balises ont la syntaxe suivante :

```
<f:selectItem itemValue="valeur" itemLabel="texte"/>
```

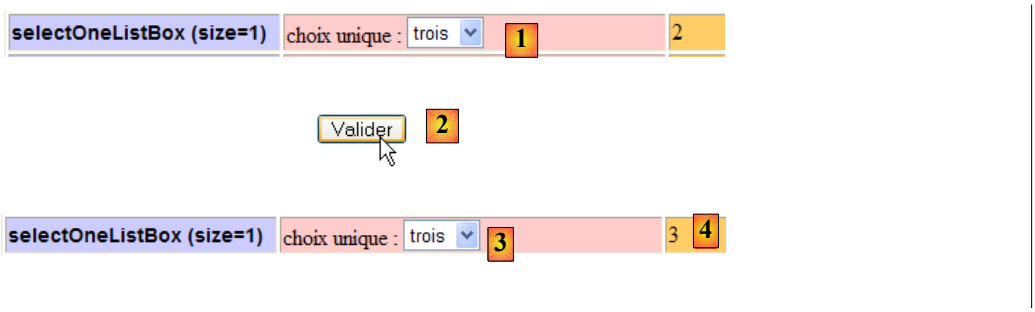
  La valeur de l'attribut **itemLabel** est ce qui est affiché dans la liste. La valeur de l'attribut **itemValue** est la **valeur** de l'élément. C'est cette valeur qui sera envoyée au contrôleur [Faces Servlet] si l'élément est sélectionné dans la liste déroulante.
  L'élément affiché en [3] a été déterminé par appel à la méthode **getSelectOneListBox1()** (ligne 5). Le résultat "2" obtenu (ligne 1 du code Java) a fait que l'élément de la ligne 7 de la liste déroulante a été affiché, ceci parce que son attribut **itemValue** vaut "2",
- la ligne 11 du code XHTML génère [4]. Ici la méthode **getSelectOneListBox1** de [Form.java] a été de nouveau utilisée.

Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectOneListBox (size=1)</span></td>
3.  <td class="col2">choix unique : <select id="formulaire:selectOneListBox1"
    name="formulaire:selectOneListBox1" size="1">
4.          <option value="1">un</option>
5.          <option value="2" selected="selected">deux</option>
6.          <option value="3">trois</option>
7.  </select></td>
8.  <td class="col3">2</td>
9.  </tr>
```

- lignes 3 et 7 : la balise HTML <**select** ...>...</**select**> générée par la balise JSF <h:**selectOneListBox**>,
- lignes 4-6 : les balises HTML <**option** ...> ... </**option**> générées par les balises JSF <f:**selectItem**>,
- ligne 5 : le fait que l'élément de valeur="2" soit sélectionné dans la liste se traduit par la présence de l'attribut **selected**="selected".

Maintenant, ci-dessous, choisissons [1] une nouvelle valeur dans la liste et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3] :



La valeur du champ [1] postée est la suivante :

```
formulaire%3AselectOneListBox1=3
```

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. L'élément HTML

```
 <option value="3">trois</option>
```

a été sélectionné. Le navigateur a envoyé la chaîne "3" comme valeur du composant JSF ayant produit la liste déroulante :

```
<h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
```

Le contrôleur JSF va utiliser la méthode **setSelectOneListBox1**("3") pour mettre à jour le modèle de la liste déroulante. Aussi après cette mise à jour, le champ du modèle [Form.java]

```
private String selectOneListBox1;
```

contient désormais la valeur "3".

Lorsque la page [index.xhtml] est réaffichée à l'issue de son traitement, cette valeur provoque l'affichage [3,4] ci-dessus :

- elle détermine l'élément de la liste déroulante qui doit être affiché [3],
- la valeur du champ *selectOneListBox1* est affichée en [4].

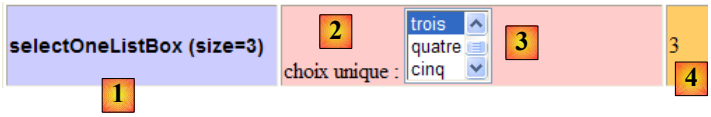
Considérons une variante de la balise <**h:selectOneListBox**> :

```
1.  <!-- ligne 5 -->
2.            <h:outputText value="selectOneListBox (size=3)" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectOneListBox2Prompt']}"/>
5.              <h:selectOneListbox id="selectOneListBox2" value="#{form.selectOneListBox2}"
    size="3">
6.                <f:selectItem itemValue="1" itemLabel="un"/>
7.                <f:selectItem itemValue="2" itemLabel="deux"/>
8.                <f:selectItem itemValue="3" itemLabel="trois"/>
9.                <f:selectItem itemValue="4" itemLabel="quatre"/>
10.               <f:selectItem itemValue="5" itemLabel="cinq"/>
11.             </h:selectOneListbox>
12.           </h:panelGroup>
13.           <h:outputText value="#{form.selectOneListBox2}"/>
```

Le modèle dans [Form.java] de la balise <**h:selectOneListBox**> de la ligne 5 est le suivant :

```
1.    private String selectOneListBox2="3";
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



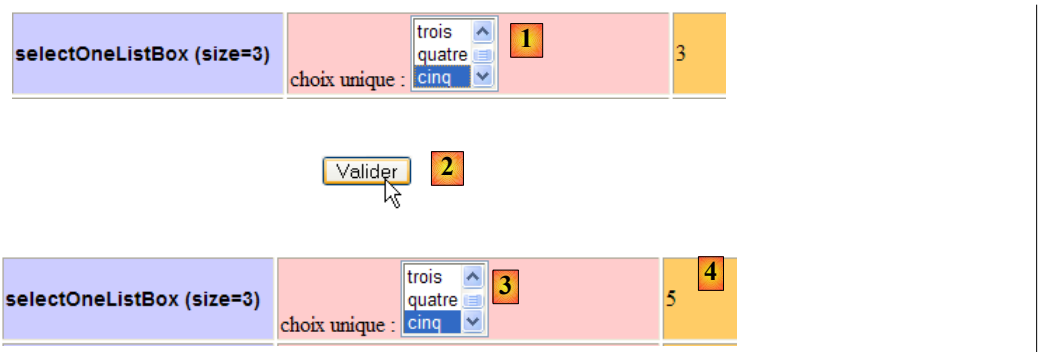
- la ligne 2 du code XHTML génère [1],
- le texte [2] est généré par la ligne 4. La liste avec ascenseur [3] est générée par les lignes [5-11]. C'est la valeur de l'attribut **size= "3 "** qui fait qu'on a une liste avec ascenseur plutôt qu'une liste déroulante. Les éléments de la liste ont été générés par les balises <**f:selectItem**> des lignes 6-8,
  L'élément sélectionné en [3] a été déterminé par appel à la méthode **getSelectOneListBox2()** (ligne 5). Le résultat "3" obtenu (ligne 1 du code Java) a fait que l'élément de la ligne 8 de la liste a été affiché, ceci parce que son attribut **itemValue** vaut "3",
- la ligne 13 du code XHTML génère [4]. Ici la méthode **getSelectOneListBox2** de [Form.java] a été de nouveau utilisée.

Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectOneListBox (size=3)</span></td>
3.  <td class="col2">choix unique : <select id="formulaire:selectOneListBox2"
    name="formulaire:selectOneListBox2" size="3">
4.          <option value="1">un</option>
5.          <option value="2">deux</option>
6.          <option value="3" selected="selected">trois</option>
7.          <option value="4">quatre</option>
8.          <option value="5">cinq</option>
9.  </select></td>
10. <td class="col3">3</td>
11. </tr>
```

- ligne 6 : le fait que l'élément de valeur="3" soit sélectionné dans la liste se traduit par la présence de l'attribut **selected**="selected".

Maintenant, ci-dessous, choisissons [1] une nouvelle valeur dans la liste et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3] :



La valeur postée pour le champ [1] est la suivante :

```
formulaire%3AselectOneListBox2=5
```

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. L'élément HTML

```
 <option value="5">cinq</option>
```

a été sélectionné. Le navigateur a envoyé la chaîne "5" comme valeur du composant JSF ayant produit la liste déroulante :

```
            <h:selectOneListbox id="selectOneListBox2" value="#{form.selectOneListBox2}" size="3">
```

Le contrôleur JSF va utiliser la méthode **setSelectOneListBox2**("5") pour mettre à jour le modèle de la liste. Aussi après cette mise à jour, le champ

```
        private String selectOneListBox2;
```

contient-il désormais la valeur "5".

Lorsque la page [index.xhtml] est réaffichée à l'issue de son traitement, cette valeur provoque l'affichage [3,4] ci-dessus :

- elle détermine l'élément de la liste qui doit être sélectionné [3],
- la valeur du champ *selectOneListBox2* est affiché en [4].

2.5.15 Balise <h:selectManyListBox>

La balise <**h:selectmanyListBox**> génère une balise HTML **<select multiple= "multiple ">...</select>** qui permet à l'utilisateur de sélectionner plusieurs éléments dans une liste.

Considérons le code suivant :

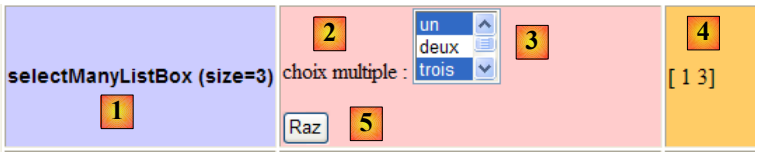
```
1.  <!-- ligne 6 -->
2.            <h:outputText value="selectManyListBox (size=3)"  styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectManyListBoxPrompt']}"/>
5.              <h:selectManyListbox id="selectManyListBox" value="#{form.selectManyListBox}"
    size="3">
6.                <f:selectItem itemValue="1" itemLabel="un"/>
7.                <f:selectItem itemValue="2" itemLabel="deux"/>
8.                <f:selectItem itemValue="3" itemLabel="trois"/>
9.                <f:selectItem itemValue="4" itemLabel="quatre"/>
10.               <f:selectItem itemValue="5" itemLabel="cinq"/>
11.             </h:selectManyListbox>
12.             <p><input type="button" value="#{msg['form.buttonRazText']}"
    onclick="this.form['formulaire:selectManyListBox'].selectedIndex=-1;" /></p>
```

```
13.           </h:panelGroup>
14.           <h:outputText value="#{form.selectManyListBoxValue}"/>
```

et son modèle dans [Form.java] :

```
1.  private String[] selectManyListBox=new String[]{"1","3"};
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



- la ligne 2 du code XHTML génère [1]
- le texte [2] est généré par la ligne 4. La liste [3] est générée par les lignes [5-11]. L'attribut **size="3"** fait que la liste affiche à un moment donné trois de ces éléments. Les éléments sélectionnés dans la liste ont été déterminés par appel à la méthode **getSelectManyListBox()** (ligne 5) du modèle Java. Le résultat {"1","3"} obtenu (ligne 1 du code Java) est un tableau d'éléments de type *String*. Chacun de ces éléments sert à sélectionner l'un des éléments de la liste. Ici, les éléments des lignes 6 et 10 ayant leur attribut **itemValue** dans le tableau {"1","3"} seront sélectionnés. C'est ce que montre [3].
- la ligne 14 du code XHTML génère [4]. Il est ici fait appel, non pas à la méthode **getSelectManyListBox** du modèle Java de la liste mais à la méthode **getSelectManyListBoxValue** suivante :

```
1.  private String[] selectManyListBox=new String[]{"1","3"};
2.    ...
3.    // getters et setters
4.
5.    public String getSelectManyListBoxValue(){
6.      return getValue(selectManyListBox);
7.    }
8.
9.    private String getValue(String[] chaines){
10.     String value="[";
11.     for(String chaine : chaines){
12.       value+=" "+chaine;
13.     }
14.     return value+"]";
15.   }
```

Si on avait fait appel à la méthode **getSelectManyListBox**, on aurait obtenu un tableau de *String*. Pour inclure cet élément dans le flux HTML, le contrôleur aurait appel à sa méthode *toString*. Or celle-ci pour un tableau ne fait que rendre le " hashcode " de celui-ci et non pas la liste de ses éléments comme nous le souhaitons. Aussi utilise-t-on la méthode **getSelectManyListBoxValue** ci-dessus pour obtenir une chaîne de caractères représentant le contenu du tableau,

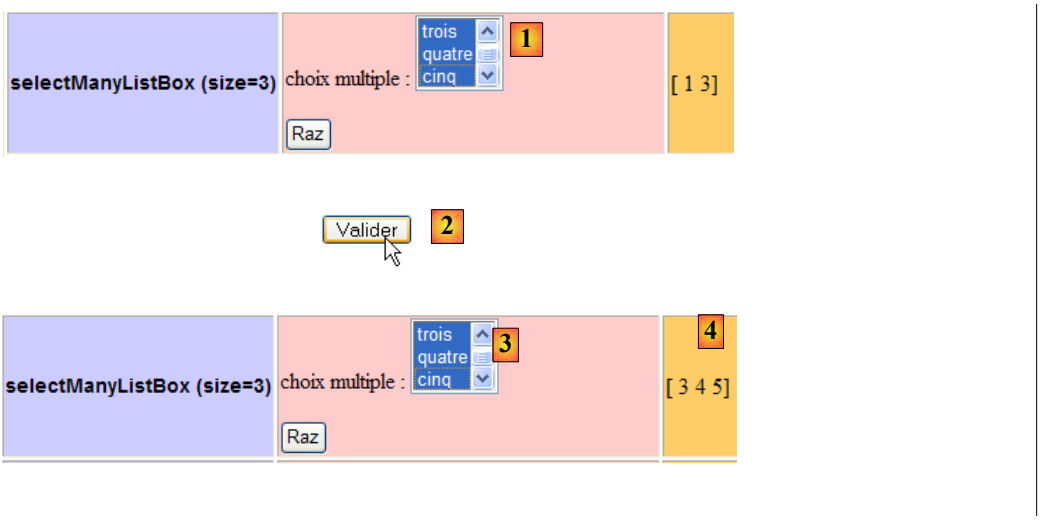
- la ligne 12 du code XHTML génère le bouton [5]. Lorsque ce bouton est cliqué, le code Javascript de l'attribut **onclick** est exécuté. Il sera embarqué au sein de la page HTML qui va être générée par le code JSF. Pour le comprendre, nous avons besoin de connaître la nature exacte de celle-ci.

Le flux HTML généré par la page XHTML est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectManyListBox (size=3)</span></td>
3.  <td class="col2">choix multiple : <select id="formulaire:selectManyListBox"
    name="formulaire:selectManyListBox" multiple="multiple" size="3">
4.          <option value="1" selected="selected">un</option>
5.          <option value="2">deux</option>
6.          <option value="3" selected="selected">trois</option>
7.          <option value="4">quatre</option>
8.          <option value="5">cinq</option>
9.  </select>
10.             <p><input type="button" value="Raz"
    onclick="this.form['formulaire:selectManyListBox'].selectedIndex=-1;" /></p>
11.           </td>
12. <td class="col3">[ 1 3]</td>
13. </tr>
```

- lignes 3 et 9 : la balise HTML <**select multiple**= "multiple "...>...</**select**> générée par la balise JSF <h:**selectManyListBox**>. C'est la présence de l'attribut **multiple** qui indique qu'on a affaire à une liste à sélection multiple,
- le fait que le modèle de la liste soit le tableau de String {"1","3"} fait que les éléments de la liste des lignes 4 (value="1") et 6 (value="3") ont l'attribut **selected**="selected",
- ligne 10 : lorsqu'on clique sur le bouton [Raz], le code Javascript de l'attribut **onclick** s'exécute. La page est représentée dans le navigateur par un arbre d'objets souvent appelé DOM (Document Object Model). Chaque objet de l'arbre est accessible au code Javascript via son attribut **name**. La liste de la ligne 3 du code HTML ci-dessus s'appelle *formulaire:selectManyListBox*. Le formulaire lui-même peut être désigné de diverses façons. Ici, il est désigné par la notation **this.form** où **this** désigne le bouton [Raz] et **this.form** le formulaire dans lequel se trouve ce bouton. La liste *formulaire:selectManyListBox* se trouve dans ce même formulaire. Aussi la notation **this.form['formulaire:selectManyListBox']** désigne-t-elle l'emplacement de la liste dans l'arbre des composants du formulaire. L'objet représentant une liste a un attribut **selectedIndex** qui a pour valeur le n° de l'élément sélectionné dans la liste. Ce n° commence à 0 pour désigner le premier élément de la liste. La valeur -1 indique qu'aucun élément n'est sélectionné dans la liste. Le code Javascript affectant la valeur -1 à l'attribut **selectedIndex** a pour effet de désélectionner tous les éléments de la liste s'il y en avait.

Maintenant, ci-dessous, choisissons [1] de nouvelles valeurs dans la liste (pour sélectionner plusieurs éléments dans la liste, maintenir la touche Ctrl appuyée en cliquant) et validons le formulaire avec le bouton [Valider] [2]. Nous obtenons en réponse la page [3,4] :



La valeur du champ [1] postée est la suivante :

```
formulaire%3AselectManyListBox=3&formulaire%3AselectManyListBox=4&formulaire%3AselectManyListBox=5
```

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. Les éléments HTML

```
<option value="3">trois</option>
<option value="4">quatre</option>
<option value="5">cinq</option>
```

ont été sélectionnés. Le navigateur a envoyé les trois chaînes "3", "4", "5" comme valeurs du composant JSF ayant produit la liste déroulante :

```
<h:selectManyListbox id="selectManyListBox" value="#{form.selectManyListBox}" size="3">
```

La méthode **setSelectManyListBox** du modèle va être utilisée pour mettre à ce jour ce modèle avec les valeurs envoyées par le navigateur :

```
1.    private String[] selectManyListBox;
2.  ....
3.    public void setSelectManyListBox(String[] selectManyListBox) {
4.      this.selectManyListBox = selectManyListBox;
5.  }
```

Ligne 3, on voit que le paramètre de la méthode est un tableau de *String*. Ici ce sera le tableau {"3","4","5"}. Après cette mise à jour, le champ

```
        private String[] selectManyListBox;
```

contient désormais le tableau {"3","4","5"}.

Lorsque la page [index.xhtml] est réaffichée à l'issue de son traitement, cette valeur provoque l'affichage [3,4] ci-dessus :

- elle détermine les éléments de la liste qui doivent être sélectionnés [3],
- la valeur du champ *selectManyListBox* est affichée en [4].

2.5.16 Balise <h:selectOneMenu>

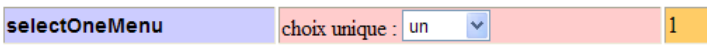
La balise <**h:selectOneMenu**> est identique à la balise <**h:selectOneListBox size="1"**>. Dans l'exemple, le code JSF exécuté est le suivant :

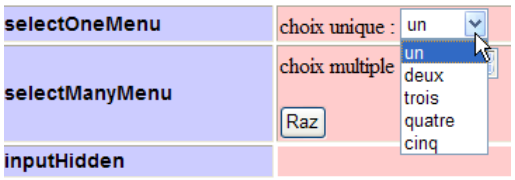
```
1.  <!-- ligne 7 -->
2.            <h:outputText value="selectOneMenu" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectOneMenuPrompt']}"/>
5.              <h:selectOneMenu id="selectOneMenu" value="#{form.selectOneMenu}">
6.                <f:selectItem itemValue="1" itemLabel="un"/>
7.                <f:selectItem itemValue="2" itemLabel="deux"/>
8.                <f:selectItem itemValue="3" itemLabel="trois"/>
9.                <f:selectItem itemValue="4" itemLabel="quatre"/>
10.               <f:selectItem itemValue="5" itemLabel="cinq"/>
11.             </h:selectOneMenu>
12.           </h:panelGroup>
13.           <h:outputText value="#{form.selectOneMenu}"/>
```

Le modèle de la balise <**h:selectOneMenu**> dans [Form.java] est le suivant :

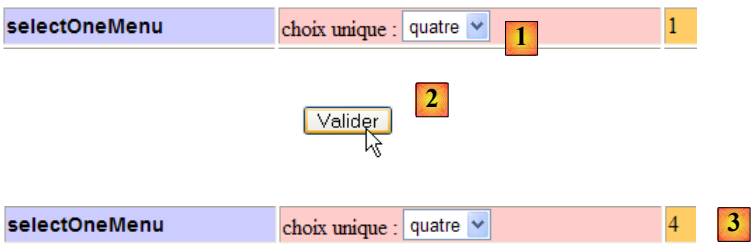
```
  private String selectOneMenu="1";
```

A la demande initiale de la page [index.xhtml], le code précédent génère la vue :





Un exemple d'exécution pourrait être celui qui suit :



La valeur postée pour le champ [1] est la suivante :

```
formulaire%3AselectOneMenu=4
```

2.5.17 Balise <h:selectManyMenu>

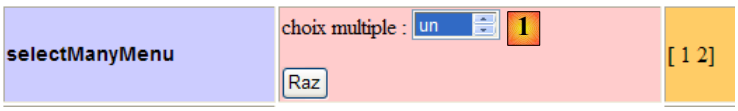
La balise <**h:selectManyMenu**> est identique à la balise <**h:selectManyListBox size="1"**>. Le code JSF exécuté dans l'exemple est le suivant :

```
1.  <!-- ligne 8 -->
2.            <h:outputText value="selectManyMenu" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectManyMenuPrompt']}" styleClass="prompt" />
5.              <h:selectManyMenu id="selectManyMenu" value="#{form.selectManyMenu}" >
6.                <f:selectItem itemValue="1" itemLabel="un"/>
7.                <f:selectItem itemValue="2" itemLabel="deux"/>
8.                <f:selectItem itemValue="3" itemLabel="trois"/>
9.                <f:selectItem itemValue="4" itemLabel="quatre"/>
10.               <f:selectItem itemValue="5" itemLabel="cinq"/>
11.             </h:selectManyMenu>
12.             <p><input type="button" value="#{msg['form.buttonRazText']}"
    onclick="this.form['formulaire:selectManyMenu'].selectedIndex=-1;" /></p>
13.           </h:panelGroup>
14.           <h:outputText value="#{form.selectManyMenuValue}" styleClass="prompt"/>
```

Le modèle de la balise <**h:selectManyMenu**> dans [Form.java] est le suivant :

```
    private String[] selectManyMenu=new String[]{"1","2"};
```

A la demande initiale de la page [index.xhtml], le code précédent génère la page :



La liste [1] contient les textes "un", ..., "cinq" avec les éléments "un" et "deux" sélectionnés. Le code HTML généré est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectManyMenu</span></td>
3.  <td class="col2"><span class="prompt">choix multiple : </span><select
    id="formulaire:selectManyMenu" name="formulaire:selectManyMenu" multiple="multiple" size="1">
4.          <option value="1" selected="selected">un</option>
5.          <option value="2" selected="selected">deux</option>
6.          <option value="3">trois</option>
7.          <option value="4">quatre</option>
8.          <option value="5">cinq</option>
9.  </select>
10.
11.
12.             <p><input type="button" value="Raz"
    onclick="this.form['formulaire:selectManyMenu'].selectedIndex=-1;" /></p>
13.           </td>
14. <td class="col3"><span class="prompt">[ 1 2]</span></td>
15. </tr>
```

On voit ci-dessus, en lignes 4 et 5 que les éléments "un" et "deux" sont sélectionnés (présence de l'attribut *selected*).
Il est difficile de donner une copie d'écran d'un exemple d'exécution car on ne peut pas montrer les éléments sélectionnés dans le menu. Le lecteur est invité à faire le test lui-même (pour sélectionner plusieurs éléments dans la liste, maintenir la touche Ctrl appuyée en cliquant).

2.5.18 Balise <h:inputHidden>

La balise <**h:inputHidden**> n'a pas de représentation visuelle. Elle ne sert qu'à insérer une balise HTML <**input type="hidden" value="..."**/> dans le flux HTML de la page. Inclus à l'intérieur d'une balise <h:form>, leurs valeurs font partie des valeurs

envoyées au serveur lorsque le formulaire est posté. Parce que ce sont des champs de formulaire et que l'utilisateur ne les voie pas, on les appelle des **champs cachés**. L'intérêt de ces champs est de garder de la mémoire entre les différents cycles demande / réponse d'un même client :

- le client demande un formulaire F. Le serveur le lui envoie et met une information I dans un champ caché C, sous la forme *<h:inputHidden id="C" value="I"/>*,
- lorsque le client a rempli le formulaire F et qu'il le poste au serveur, la valeur I du champ C est renvoyée au serveur. Celui-ci peut alors retrouver l'information I qu'il avait stockée dans la page. On a ainsi créé une mémoire entre les deux cycles demande / réponse,
- JSF utilise lui-même cette technique. L'information I qu'il stocke dans le formulaire F est la valeur de tous les composants de celui-ci. Il utilise le champ caché suivant pour cela :

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="H4sIAAAAAAAAANV...8PswawAA" />
```

Le champ caché s'appelle **javax.faces.ViewState** et sa valeur est une chaîne qui représente sous forme codée la valeur de tous les composants de la page envoyée au client. Lorsque celui-ci renvoie la page après avoir fait des saisies dans le formulaire, le champ caché **javax.faces.ViewState** est renvoyé avec les valeurs saisies. C'est ce qui permet au contrôleur JSF de reconstituer la page telle qu'elle avait été envoyée initialement. Ce mécanisme a été expliqué page 63.

Le code JSF de l'exemple est le suivant :

```
1.  <!-- ligne 9 -->
2.            <h:outputText value="inputHidden"  styleClass="info"/>
3.            <h:inputHidden id="inputHidden" value="#{form.inputHidden}"/>
4.            <h:outputText value="#{form.inputHidden}"/>
```

Le modèle de la balise *<h:inputHidden>* dans [Form.java] est le suivant :

```
  private String inputHidden="initial";
```

Ce qui donne l'affichage suivant lors de la demande initiale de la page [index.xhtml] :

- la ligne 2 génère [1], la ligne 4 [2]. La ligne 3 ne génère aucun élément visuel.

Le code HTML généré est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">inputHidden</span></td>
3.  <td class="col2"><input id="formulaire:inputHidden" type="hidden" name="formulaire:inputHidden"
    value="initial" /></td>
4.  <td class="col3">initial</td>
5.  </tr>
```

Au moment du POST du formulaire, la valeur "initial" du champ nommé *formulaire:inputHidden* de la ligne 3 sera postée avec les autres valeurs du formulaire. Le champ

```
  private String inputHidden;
```

sera mis à jour avec cette valeur, qui est celle qu'il avait déjà initialement. Cette valeur sera intégrée dans la nouvelle page renvoyée au client. On obtient donc toujours la copie d'écran ci-dessus.

La valeur postée pour le champ caché est la suivante :

```
formulaire%3AinputHidden=initial
```

2.5.19 Balise <h:selectBooleanCheckBox>

La balise **<h:selectBooleanCheckBox>** génère une balise HTML **<input type="checkbox" ...>**.

Considérons le code JSF suivant :

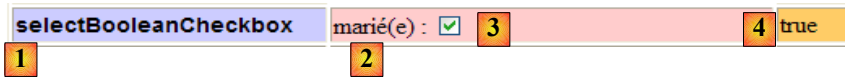
```
1.  <!-- ligne 10 -->
```

```
2.    <h:outputText value="selectBooleanCheckbox" styleClass="info"/>
3.    <h:panelGroup>
4.      <h:outputText value="#{msg['form.selectBooleanCheckboxPrompt']}" styleClass="prompt" />
5.      <h:selectBooleanCheckbox id="selectBooleanCheckbox" value="#{form.selectBooleanCheckbox}"/>
6.    </h:panelGroup>
7.    <h:outputText value="#{form.selectBooleanCheckbox}"/>
```

Le modèle de la balise <**h:selectBooleanCheckbox**> de ligne 5 ci-dessus dans [Form.java] est le suivant:

```
private boolean selectBooleanCheckbox=true;
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



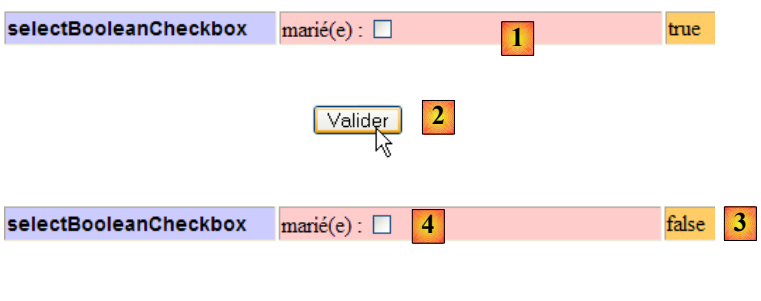
- la ligne 2 du code XHTML génère [1],
- le texte [2] est généré par la ligne 4. La case à cocher [3] est générée par la ligne [5]. Ici, la méthode **getSelectBooleanCheckbox** de [Form.java] a été utilisée pour cocher ou non la case. La méthode rendant le booléen **true** (cf code Java), la case a été cochée,
- la ligne 7 du code XHTML génère [4]. C'est de nouveau la méthode **getSelectBooleanCheckbox** de [Form.java] qui est utilisée pour générer le texte [4].

Le flux HTML généré par le code JSF précédent est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectBooleanCheckbox</span></td>
3.  <td class="col2"><span class="prompt">mari&eacute;(e) : </span>
4.  <input id="formulaire:selectBooleanCheckbox" type="checkbox"
    name="formulaire:selectBooleanCheckbox" checked="checked" /></td>
5.  <td class="col3">true</td>
6.  </tr>
```

En [4], on voit la balise HTML <**input type="checkbox"**> qui a été générée. La valeur *true* du modèle associé a fait que l'attribut **checked="checked"** a été ajouté à la balise. Ce qui fait que la case est cochée.

Maintenant, ci-dessous, décochons la case [1], validons le formulaire [2] et regardons le résultat obtenu [3, 4] :



Parce que la case est décochée, il n'y a pas de valeur postée pour le champ [1].

La validation du formulaire par [2] a provoqué la mise à jour du modèle [Form.java] par la saisie [1]. Le champ **selectBooleanCheckbox** de [Form.java] a reçu alors la valeur **false**. Le réaffichage de [index.xhtml] montre que le champ **selectBooleanCheckbox** du modèle a bien été mis à jour [3] et [4]. Il est intéressant de noter ici que c'est grâce au champ caché **javax.faces.ViewState** que JSF a été capable de dire que la case à cocher initialement cochée avait été décochée par l'utilisateur. En effet, la valeur d'une case décochée ne fait pas partie des valeurs postées par le navigateur. Grâce à l'arbre des composants stocké dans le champ caché **javax.faces.ViewState** JSF retrouve le fait qu'il y avait une case à cocher nommée "selectBooleanCheckbox" dans le formulaire et que sa valeur ne fait pas partie des valeurs postées par le navigateur client. Il peut en conclure qu'elle était décochée dans le formulaire posté, ce qui lui permet d'affecter le booléen *false* au modèle Java associé :

```
private boolean selectBooleanCheckbox;
```

2.5.20 Balise <h:selectManyCheckBox>

La balise <**h:selectManyCheckBox**> génère un groupe de cases à cocher et donc plusieurs balises HTML <**input type="checkbox" ...**>. Cette balise est le pendant de la balise <**h:selectManyListBox**>, si ce n'est que les éléments à sélectionner sont présentés sous forme de cases à cocher contigües plutôt que sous forme de liste. Ce qui a été dit pour la balise <**h:selectManyListBox**> reste valide ici.

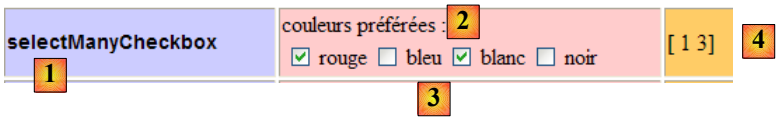
Considérons le code JSF suivant :

```
1.           <!-- ligne 11 -->
2.           <h:outputText value="selectManyCheckbox" styleClass="info"/>
3.           <h:panelGroup>
4.             <h:outputText value="#{msg['form.selectManyCheckboxPrompt']}" styleClass="prompt" />
5.             <h:selectManyCheckbox id="selectManyCheckbox" value="#{form.selectManyCheckbox}">
6.               <f:selectItem itemValue="1" itemLabel="rouge"/>
7.               <f:selectItem itemValue="2" itemLabel="bleu"/>
8.               <f:selectItem itemValue="3" itemLabel="blanc"/>
9.               <f:selectItem itemValue="4" itemLabel="noir"/>
10.            </h:selectManyCheckbox>
11.          </h:panelGroup>
12. <h:outputText value="#{form.selectManyCheckboxValue}"/>
```

Le modèle de la balise <**h:selectManyCheckbox**> de ligne 5 ci-dessus dans [Form.java] est le suivant:

```
private String[] selectManyCheckbox=new String[]{"1","3"};
```

Lorsque la page [index.xhtml] est demandée la première fois, la page obtenue est la suivante :



- la ligne 2 du code XHTML génère [1],
- le texte [2] est généré par la ligne 4. Les cases à cocher [3] sont générées par les lignes 5-10. Pour chacune d'elles :
  - l'attribut **itemLabel** définit le texte affiché près de la case à cocher ;
  - l'attribut **itemvalue** définit la valeur qui sera postée au serveur si la case est cochée,

  Le modèle des quatre cases est le champ Java suivant :

  ```
  private String[] selectManyCheckbox=new String[]{"1","3"};
  ```

  Ce tableau définit :
  - lorsque la page est affichée, les cases qui doivent être cochées. Ceci est fait via leur valeur, c.a.d. leur champ *itemValue*. Ci-dessus, les cases ayant leurs valeurs dans le tableau {"1","3"} seront cochées. C'est ce qui est vu sur la copie d'écran ci-dessus ;
  - lorsque la page est postée, le modèle *selectManyCheckbox* reçoit le tableau des valeurs des cases que l'utilisateur a cochées. C'est ce que nous allons voir prochainement,
- la ligne 12 du code XHTML génère [4]. C'est la méthode *getSelectManyCheckboxValue* suivante qui a généré [4] :

```
1.    public String getSelectManyCheckboxValue(){
2.      return getValue(getSelectManyCheckbox());
3.    }
4.
5.    private String getValue(String[] chaines){
6.      String value="[";
7.      for(String chaine : chaines){
8.        value+=" "+chaine;
9.      }
10.     return value+"]";
11. }
```

Le flux HTML généré par le code JSF précédent est le suivant :

```
1.          <tr>
2.  <td>
```

```
3.  <input name="formulaire:selectManyCheckbox" id="formulaire:selectManyCheckbox:0" value="1"
    type="checkbox" checked="checked" /><label for="formulaire:selectManyCheckbox:0">
    rouge</label></td>
4.  <td>
5.  <input name="formulaire:selectManyCheckbox" id="formulaire:selectManyCheckbox:1" value="2"
    type="checkbox" /><label for="formulaire:selectManyCheckbox:1"> bleu</label></td>
6.  <td>
7.  <input name="formulaire:selectManyCheckbox" id="formulaire:selectManyCheckbox:2" value="3"
    type="checkbox" checked="checked" /><label for="formulaire:selectManyCheckbox:2">
    blanc</label></td>
8.  <td>
9.  <input name="formulaire:selectManyCheckbox" id="formulaire:selectManyCheckbox:3" value="4"
    type="checkbox" /><label for="formulaire:selectManyCheckbox:3"> noir</label></td>
10.         </tr>
11. </table></td>
12. <td class="col3">[ 1 3]</td>
13. </tr>
```

Quatre balises HTML **<input type="checkbox" ...>** ont été générées. Les balises des lignes 3 et 7 ont l'attribut **checked="checked"** qui font qu'elles apparaissent cochées. On notera qu'elles ont toutes le même attribut **name="formulaire:selectManyCheckbox"**, autrement dit les quatre champs HTML ont le même nom. Si les cases des lignes 5 et 9 sont cochées par l'utilisateur, le navigateur enverra les valeurs des quatre cases à cocher sous la forme :

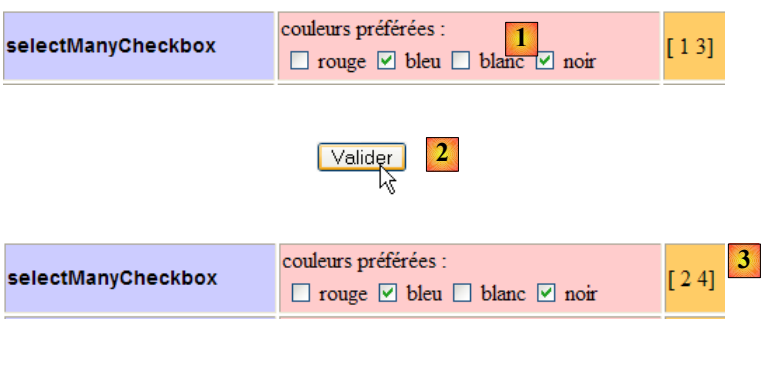
```
formulaire:selectManyCheckbox=2&formulaire:selectManyCheckbox=4
```

et le modèle des quatre cases

```
private String[] selectManyCheckbox=new String[]{"1","3"};
```

recevra le tableau {"2","4"}.

Vérifions-le ci-dessous. En [1], on fait le changement, en [2] on valide le formulaire. En [3] le résultat obtenu :



Les valeurs postées pour les champs [1] sont les suivantes :

```
formulaire%3AselectManyCheckbox=2&formulaire%3AselectManyCheckbox=4
```

2.5.21 Balise <h:selectOneRadio>

La balise **<h:selectOneRadio>** génère un groupe de boutons radio exclusifs les uns des autres.

Considérons le code JSF suivant :

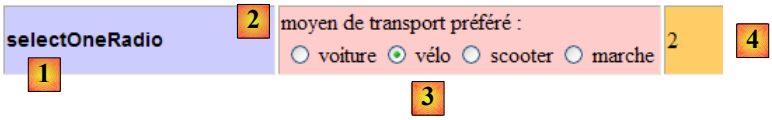
```
1.  <!-- ligne 12 -->
2.            <h:outputText value="selectOneRadio" styleClass="info"/>
3.            <h:panelGroup>
4.              <h:outputText value="#{msg['form.selectOneRadioPrompt']}" />
5.              <h:selectOneRadio id="selectOneRadio" value="#{form.selectOneRadio}">
6.                <f:selectItem itemValue="1" itemLabel="voiture"/>
7.                <f:selectItem itemValue="2" itemLabel="vélo"/>
8.                <f:selectItem itemValue="3" itemLabel="scooter"/>
```

```
9.                <f:selectItem itemValue="4" itemLabel="marche"/>
10.             </h:selectOneRadio>
11.           </h:panelGroup>
12.           <h:outputText value="#{form.selectOneRadio}"/>
```

Le modèle de la balise <*h:selectOneRadio*> de la ligne 5 ci-dessus est le suivant dans [Form.java] :

```
private String selectOneRadio="2";
```

Lorsque la page [index.xhtml] est demandée la première fois, la vue obtenue est la suivante :



- la ligne 2 du code XHTML génère [1],
- le texte [2] est généré par la ligne 4. Les boutons radio [3] sont générés par les lignes 5-10. Pour chacun d'eux :
  - l'attribut **itemLabel** définit le texte affiché près du bouton radio ;
  - l'attribut **itemvalue** définit la valeur qui sera postée au serveur si le bouton est coché,

  Le modèle des quatre boutons radio est le champ Java suivant :

  ```
  private String selectOneRadio="2";
  ```

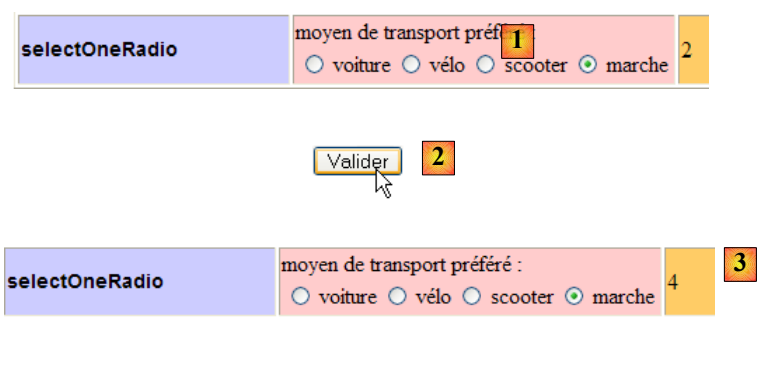
  Ce modèle définit :
  - lorsque la page est affichée, l'unique bouton radio qui doit être coché. Ceci est fait via leur valeur, c.a.d. leur champ *itemValue*. Ci-dessus, le bouton radio ayant la valeur " 2 " sera cochée. C'est ce qui est vu sur la copie d'écran ci-dessus ;
  - lorsque la page est postée, le modèle *selectOneRadio* reçoit la valeur du bouton radio qui a été coché. C'est ce que nous allons voir prochainement,
- la ligne 12 du code XHTML génère [4].

Le flux HTML généré par le code JSF précédent est le suivant :

```
1.  <tr>
2.  <td class="col1"><span class="info">selectOneRadio</span></td>
3.  <td class="col2">moyen de transport pr&eacute;f&eacute;r&eacute; : <table
    id="formulaire:selectOneRadio">
4.    <tr>
5.  <td>
6.  <input type="radio" name="formulaire:selectOneRadio" id="formulaire:selectOneRadio:0" value="1"
    /><label for="formulaire:selectOneRadio:0"> voiture</label></td>
7.  <td>
8.  <input type="radio" checked="checked" name="formulaire:selectOneRadio"
    id="formulaire:selectOneRadio:1" value="2" /><label for="formulaire:selectOneRadio:1">
    v&eacute;lo</label></td>
9.  <td>
10. <input type="radio" name="formulaire:selectOneRadio" id="formulaire:selectOneRadio:2" value="3"
    /><label for="formulaire:selectOneRadio:2"> scooter</label></td>
11. <td>
12. <input type="radio" name="formulaire:selectOneRadio" id="formulaire:selectOneRadio:3" value="4"
    /><label for="formulaire:selectOneRadio:3"> marche</label></td>
13. </tr>
```

Quatre balises HTML **<input type="radio" ...>** ont été générées. La balise de la ligne 8 a l'attribut **checked="checked"** qui fait que le bouton radio correspondant apparaît coché. On notera que les balises ont toutes le même attribut **name="formulaire:selectOneRadio"**, autrement dit les quatre champs HTML ont le même nom. C'est la condition pour avoir un groupe de boutons radio exclusifs : lorsque l'un est coché, les autres ne le sont pas.

Ci-dessous, en [1], on coche un des boutons radio, en [2] on valide le formulaire, en [3] le résultat obtenu :



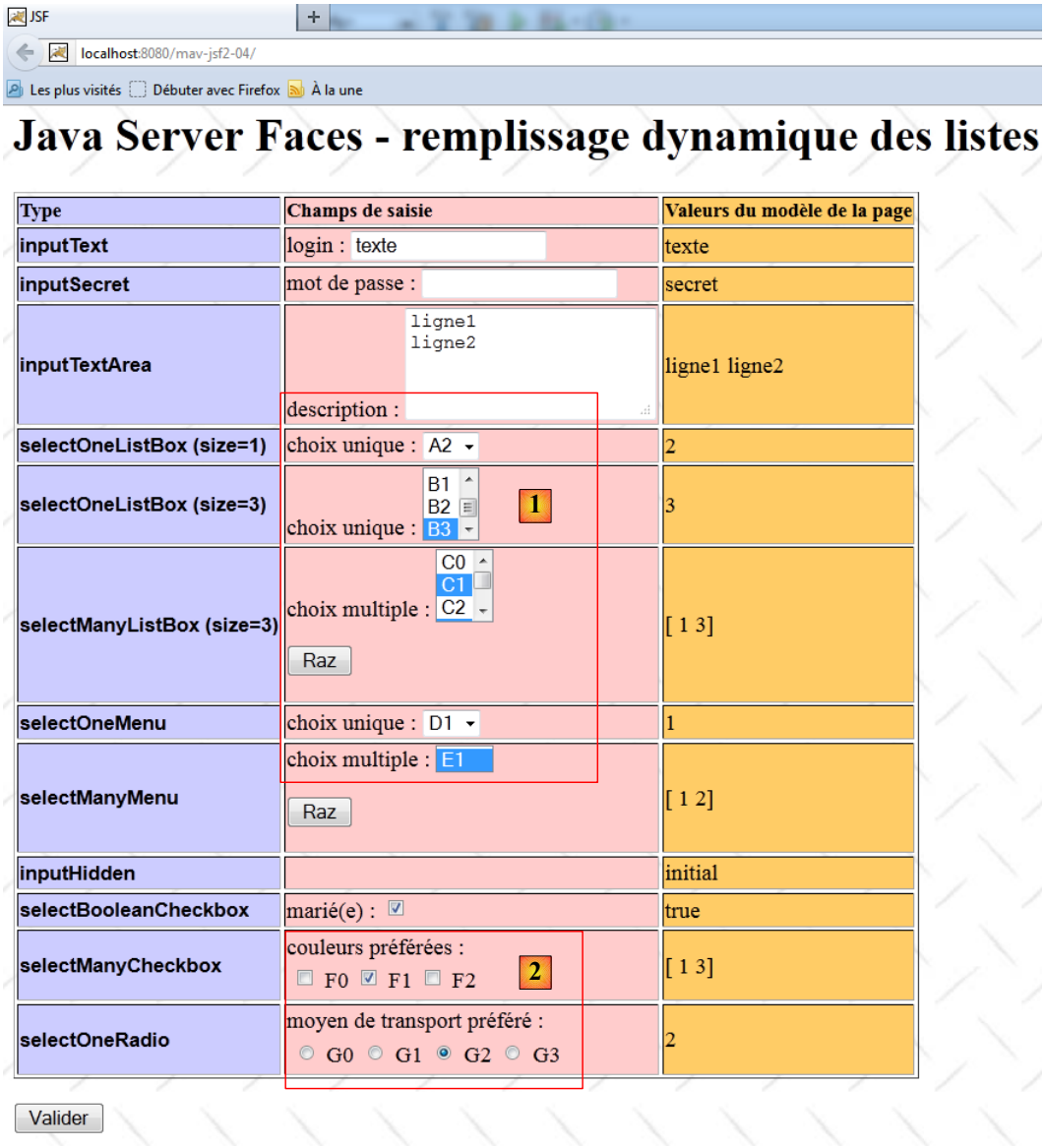
La valeur postée pour le champ [1] est la suivante :

```
formulaire%3AselectOneRadio=4
```

2.6 Exemple mv-jsf2-04 : listes dynamiques

2.6.1 L'application

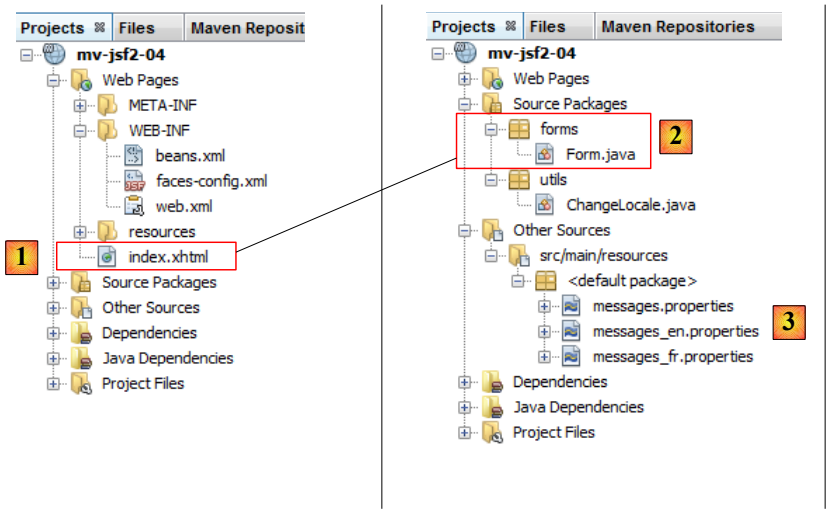
L'application est la même que précédemment :



Les seuls changements proviennent de la façon dont sont générés les éléments des listes des zones [1] et [2]. Ils sont ici générés dynamiquement par du code Java alors que dans la version précédente ils étaient écrits "en dur" dans le code de la page JSF.

2.6.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



Le projet [mv-jsf2-04] est identique au projet [mv-jsf2-03] aux différences près suivantes :

- en [1], dans la page JSF, les éléments des listes ne vont plus être écrites "en dur" dans le code,
- en [2], le modèle de la page JSF [1] va être modifié,
- en [3], l'un des messages va être modifié.

2.6.3 La page [index.xhtml] et son modèle [Form.java]

La page JSF [index.xhtml] devient la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core">
6.
7.    <f:view locale="#{changeLocale.locale}">
8.      <h:head>
9.        <title>JSF</title>
10.       <h:outputStylesheet library="css" name="styles.css"/>
11.     </h:head>
12.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
13.       <h:form id="formulaire">
14.         <!-- langues -->
15.         <h:panelGrid columns="2">
16.           <h:commandLink value="#{msg['form.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
17.           <h:commandLink value="#{msg['form.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
18.         </h:panelGrid>
19.         <h1><h:outputText value="#{msg['form.titre']}"/></h1>
20.         <h:panelGrid columnClasses="col1,col2,col3" columns="3" border="1">
21. ...
22.           <!-- ligne 4 -->
23.           <h:outputText value="selectOneListBox (size=1)" styleClass="info"/>
24.           <h:panelGroup>
25.             <h:outputText value="#{msg['form.selectOneListBox1Prompt']}"/>
26.             <h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
27.               <f:selectItems value="#{form.selectOneListbox1Items}"/>
28.             </h:selectOneListbox>
29.           </h:panelGroup>
30.           <h:outputText value="#{form.selectOneListBox1}"/>
31.           <!-- ligne 5 -->
32.           <h:outputText value="selectOneListBox (size=3)" styleClass="info"/>
33.           <h:panelGroup>
34.             <h:outputText value="#{msg['form.selectOneListBox2Prompt']}"/>
35.             <h:selectOneListbox id="selectOneListBox2" value="#{form.selectOneListBox2}" size="3">
36.               <f:selectItems value="#{form.selectOneListbox2Items}"/>
37.             </h:selectOneListbox>
38.           </h:panelGroup>
39.           <h:outputText value="#{form.selectOneListBox2}"/>
40.           <!-- ligne 6 -->
41.           <h:outputText value="selectManyListBox (size=3)"  styleClass="info"/>
42.           <h:panelGroup>
43.             <h:outputText value="#{msg['form.selectManyListBoxPrompt']}"/>
44.             <h:selectManyListbox id="selectManyListBox" value="#{form.selectManyListBox}" size="3">
45.               <f:selectItems value="#{form.selectManyListBoxItems}"/>
46.             </h:selectManyListbox>
47.             <p><input type="button" value="#{msg['form.buttonRazText']}"
    onclick="this.form['formulaire:selectManyListBox'].selectedIndex=-1;" /></p>
```

```
48.           </h:panelGroup>
49.           <h:outputText value="#{form.selectManyListBoxValue}"/>
50.           <!-- ligne 7 -->
51.           <h:outputText value="selectOneMenu" styleClass="info"/>
52.           <h:panelGroup>
53.             <h:outputText value="#{msg['form.selectOneMenuPrompt']}"/>
54.             <h:selectOneMenu id="selectOneMenu" value="#{form.selectOneMenu}">
55.               <f:selectItems value="#{form.selectOneMenuItems}"/>
56.             </h:selectOneMenu>
57.           </h:panelGroup>
58.           <h:outputText value="#{form.selectOneMenu}"/>
59.           <!-- ligne 8 -->
60.           <h:outputText value="selectManyMenu" styleClass="info"/>
61.           <h:panelGroup>
62.             <h:outputText value="#{msg['form.selectManyMenuPrompt']}" styleClass="prompt" />
63.             <h:selectManyMenu id="selectManyMenu" value="#{form.selectManyMenu}" >
64.               <f:selectItems value="#{form.selectManyMenuItems}"/>
65.             </h:selectManyMenu>
66.             <p><input type="button" value="#{msg['form.buttonRazText']}"
    onclick="this.form['formulaire:selectManyMenu'].selectedIndex=-1;" /></p>
67.           </h:panelGroup>
68.           <h:outputText value="#{form.selectManyMenuValue}" styleClass="prompt"/>
69. ...
70.           <!-- ligne 11 -->
71.           <h:outputText value="selectManyCheckbox" styleClass="info"/>
72.           <h:panelGroup>
73.             <h:outputText value="#{msg['form.selectManyCheckboxPrompt']}" styleClass="prompt" />
74.             <h:selectManyCheckbox id="selectManyCheckbox" value="#{form.selectManyCheckbox}">
75.               <f:selectItems value="#{form.selectManyCheckboxItems}"/>
76.             </h:selectManyCheckbox>
77.           </h:panelGroup>
78.           <h:outputText value="#{form.selectManyCheckboxValue}"/>
79.           <!-- ligne 12 -->
80.           <h:outputText value="selectOneRadio" styleClass="info"/>
81.           <h:panelGroup>
82.             <h:outputText value="#{msg['form.selectOneRadioPrompt']}" />
83.             <h:selectOneRadio id="selectOneRadio" value="#{form.selectOneRadio}">
84.               <f:selectItems value="#{form.selectOneRadioItems}"/>
85.             </h:selectOneRadio>
86.           </h:panelGroup>
87.           <h:outputText value="#{form.selectOneRadio}"/>
88.         </h:panelGrid>
89.         <p>
90.           <h:commandButton type="submit" id="submit" value="#{msg['form.submitText']}"/>
91.         </p>
92.       </h:form>
93.     </h:body>
94.   </f:view>
95. </html>
```

Les modifications apportées sont illustrées par les lignes 26-28. Là où auparavant, on avait le code :

```
1.  <h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
2.                  <f:selectItem itemValue="1" itemLabel="un"/>
3.                  <f:selectItem itemValue="2" itemLabel="deux"/>
4.                  <f:selectItem itemValue="3" itemLabel="trois"/>
5.  </h:selectOneListbox>
```

on a désormais celui-ci :

```
a)  <h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
b)                  <f:selectItems value="#{form.selectOneListbox1Items}"/>
c)  </h:selectOneListbox>
```

Les trois balises <**f:selectItem**> des lignes 2-4 ont été remplacées par l'unique balise <**f:selectItems**> de la ligne b. Cette balise a un attribut **value** dont la valeur est une collection d'éléments de type *javax.faces.model.SelectItem*. Ci-dessus, la valeur de l'attribut **value** va être obtenue par appel de la méthode [form].**getSelectOneListbox1Items** suivante :

```
1.    public SelectItem[] getSelectOneListbox1Items() {
2.      return getItems("A",3);
3.    }
4.
5.    private SelectItem[] getItems(String label, int qte) {
6.      SelectItem[] items=new SelectItem[qte];
7.      for(int i=0;i<qte;i++){
8.        items[i]=new SelectItem(i,label+i);
9.      }
10.     return items;
11. }
```

- ligne 1, la méthode **getSelectOneListbox1Items** rend un tableau d'éléments de type *javax.faces.model.SelectItem* construit par la méthode privée **getItems** de la ligne 5. On notera que la méthode **getSelectOneListbox1Items** n'est pas le *getter* d'un champ privé *selectOneListBox1Items*,
- la classe *javax.faces.model.SelectItem* a divers constructeurs.

Constructor Summary
SelectItem() Construct a SelectItem with no initialized property values.
SelectItem(java.lang.Object value) Construct a SelectItem with the specified value.
SelectItem(java.lang.Object value, java.lang.String label) Construct a SelectItem with the specified value and label.
SelectItem(java.lang.Object value, java.lang.String label, java.lang.String description) Construct a SelectItem instance with the specified value, label and description.
SelectItem(java.lang.Object value, java.lang.String label, java.lang.String description, boolean disabled) Construct a SelectItem instance with the specified property values.

Nous utilisons ligne 8 de la méthode **getItems**, le constructeur *SelectItem(Object value, String label)* qui correspond à la balise JSF

<f:selectItem itemValue="value" labelValue="label"/>

- lignes 5-10 : la méthode **getItems(String label, int qte)** construit un tableau de **qte** éléments de type **SelectItem**, où l'élément **i** est obtenu par le constructeur **SelectItem(i, label+i)**.

Le code JSF

```
<h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
              <f:selectItems value="#{form.selectOneListbox1Items}"/>
</h:selectOneListbox>
```

devient alors fonctionnellement équivalent au code JSF suivant :

```
<h:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}" size="1">
              <f:selectItem itemValue="0" itemLabel="A0"/>
              <f:selectItem itemValue="1" itemLabel="A1"/>
              <f:selectItem itemValue="2" itemLabel="A2"/>
</h:selectOneListbox>
```

Il est fait de même pour toutes les autres listes de la page JSF. On trouve ainsi dans le modèle [Form.java] les nouvelles méthodes suivantes :

```
1.    public SelectItem[] getSelectOneListbox1Items() {
2.      return getItems("A",3);
3.    }
4.
5.    public SelectItem[] getSelectOneListbox2Items() {
6.      return getItems("B",4);
7.    }
8.
9.    public SelectItem[] getSelectManyListBoxItems() {
10.     return getItems("C",5);
11.   }
12.
13.   public SelectItem[] getSelectOneMenuItems() {
14.     return getItems("D",3);
15.   }
16.
17.   public SelectItem[] getSelectManyMenuItems() {
18.    return getItems("E",4);
19.    }
20.
21.   public SelectItem[] getSelectManyCheckboxItems() {
22.    return getItems("F",3);
23.    }
24.
25.   public SelectItem[] getSelectOneRadioItems() {
```

```
26.    return getItems("G",4);
27.    }
28.
29.   private SelectItem[] getItems(String label, int qte) {
30.     SelectItem[] items=new SelectItem[qte];
31.     for(int i=0;i<qte;i++){
32.       items[i]=new SelectItem(i,label+i);
33.     }
34.     return items;
35. }
```

2.6.4 Le fichier des messages

Un seul message est modifié :

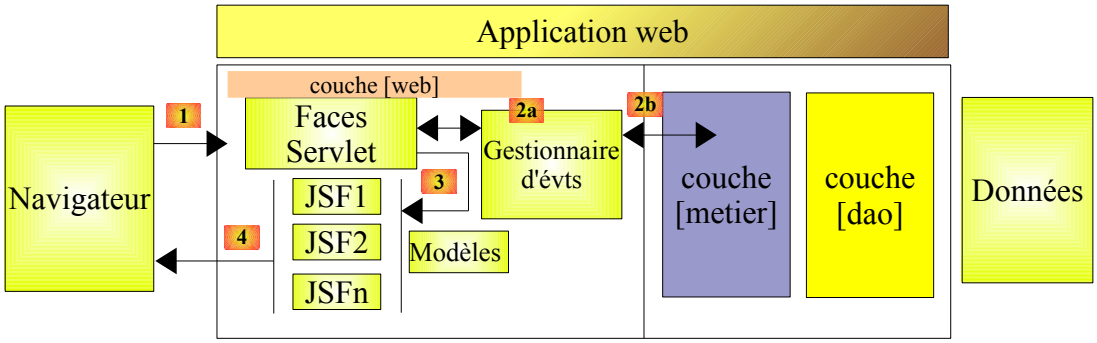
[messages_fr.properties]
```
form.titre=Java Server Faces - remplissage dynamique des listes
```

[messages_en.properties]
```
form.titre=Java Server Faces - dynamic filling of lists of elements
```

2.6.5 Tests

Le lecteur est invité à tester cette nouvelle version.

Le plus souvent les éléments dynamiques d'un formulaire sont les résultats d'un traitement métier ou proviennent d'une base de données :



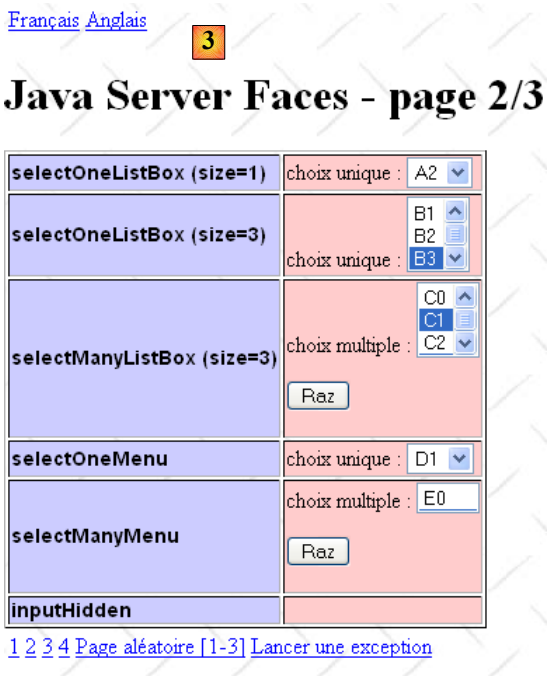
Etudions la demande initiale de la page JSF [index.xhtml] par un GET du navigateur :

- la page JSF est demandée [1],
- le contrôleur [Faces Servlet] demande son affichage en [3]. Le moteur JSF qui traite la page fait appel au modèle [Form.java] de celle-ci, par exemple à la méthode **getSelectOneListBox1Items**. Cette méthode pourrait très bien rendre un tableau d'éléments de type *SelectItem*, à partir d'informations enregistrées dans une base de données. Pour cela, elle ferait appel à la couche [métier] [2b].

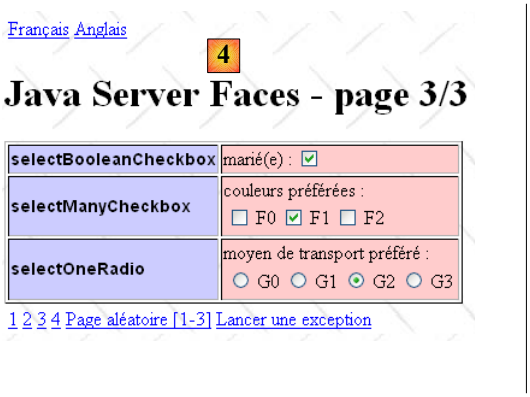
2.7 Exemple mv-jsf2-05 : navigation – session – gestion des exceptions

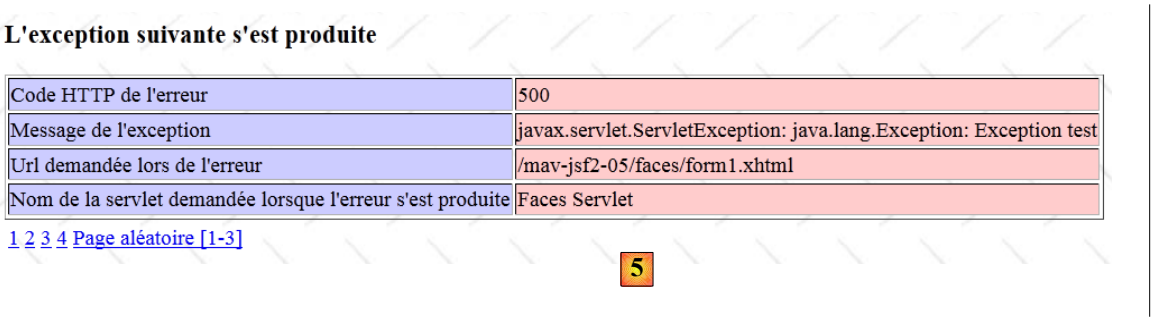
2.7.1 L'application

L'application est la même que précédemment si ce n'est que le formulaire se présente désormais sous la forme d'un assistant à plusieurs pages :



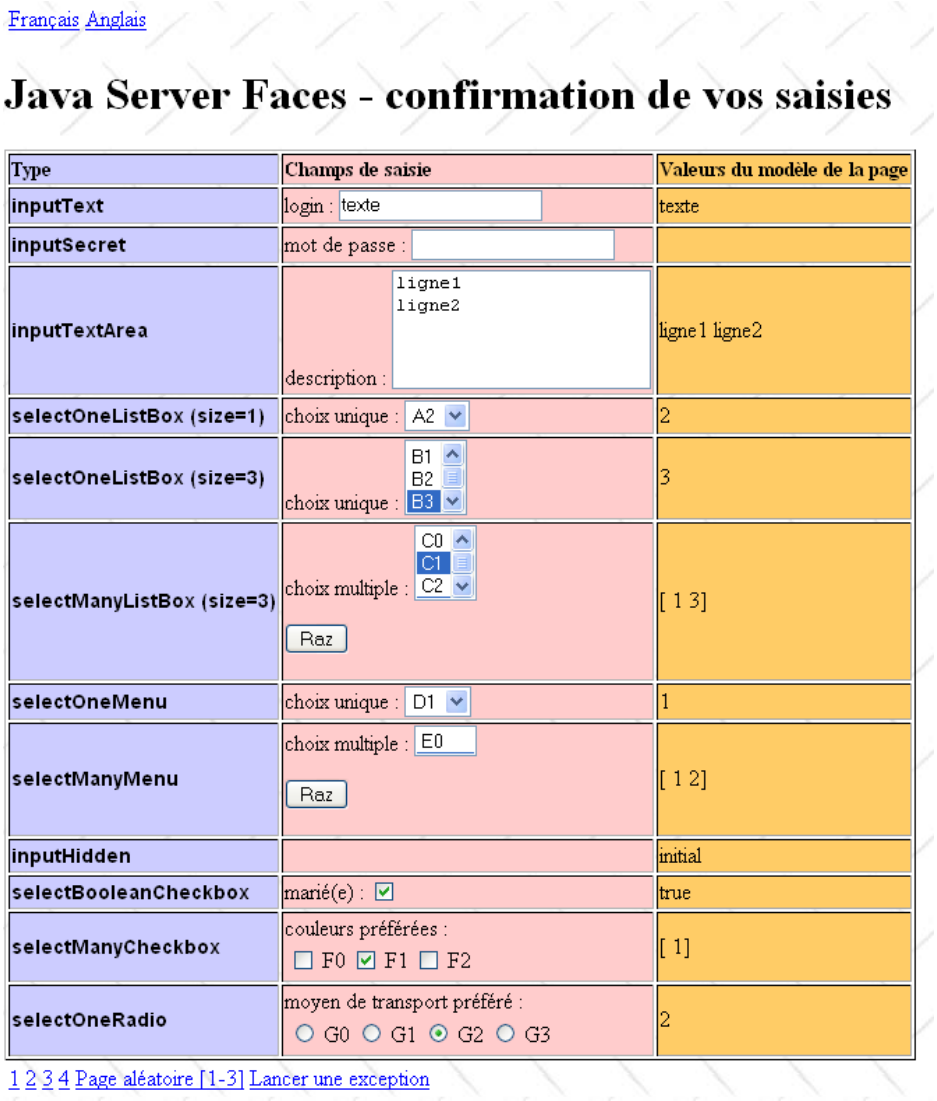
- en [1], la page 1 du formulaire - peut être obtenue également par le lien **1** de [2]
- en [2], un groupe de 5 liens.
- en [3], la page 2 du formulaire obtenue par le lien **2** de [2]





- en [4], la page 3 du formulaire obtenue par le lien **3** de [2]
- en [5], la page obtenue par le lien **Lancer une exception** de [2]

6



Français Anglais

Java Server Faces - confirmation de vos saisies

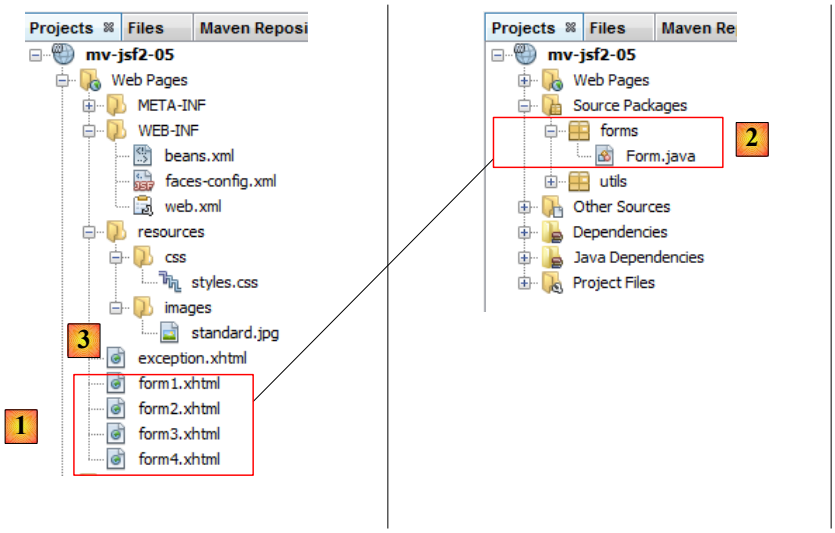
Type	Champs de saisie	Valeurs du modèle de la page
inputText	login : texte	texte
inputSecret	mot de passe :	
inputTextArea	description : ligne1 ligne2	ligne1 ligne2
selectOneListBox (size=1)	choix unique : A2	2
selectOneListBox (size=3)	choix unique : B1 B2 B3	3
selectManyListBox (size=3)	choix multiple : C0 C1 C2 Raz	[1 3]
selectOneMenu	choix unique : D1	1
selectManyMenu	choix multiple : E0 Raz	[1 2]
inputHidden		initial
selectBooleanCheckbox	marié(e) :	true
selectManyCheckbox	couleurs préférées : F0 F1 F2	[1]
selectOneRadio	moyen de transport préféré : G0 G1 G2 G3	2

1 2 3 4 Page aléatoire [1-3] Lancer une exception

- en [6], la page obtenue par le lien **4** de [2]. Elle récapitule les saisies faites dans les pages 1 à 3.

2.7.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



Le projet [mv-jsf2-05] introduit deux nouveautés :

1. en [1], la page JSF [index.xhtml] est scindée en trois pages [form1.xhtml, form2.xhtml, form3.xhtml] sur lesquelles ont été réparties les saisies. La page [form4.xhtml] est une copie de la page [index.xhtml] du projet précédent. En [2], la classe [Form.java] reste inchangée. Elle va servir de modèle aux quatre pages JSF précédentes,
2. en [3], une page [exception.xhtml] est ajoutée : elle sera utilisée lorsque se produira une exception dans l'application.

2.7.3 Les pages [form.xhtml] et leur modèle [Form.java]

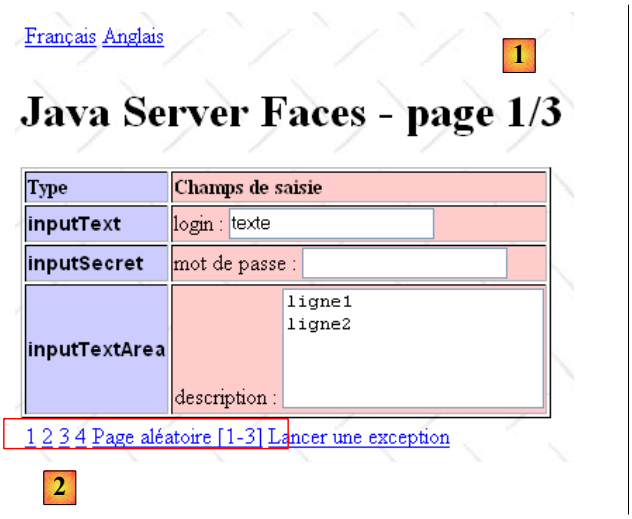
2.7.3.1 Le code des pages XHTML

La page JSF [form1.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:f="http://java.sun.com/jsf/core">
7.    <f:view locale="#{changeLocale.locale}">
8.      <h:head>
9.        <title>JSF</title>
10.       <h:outputStylesheet library="css" name="styles.css"/>
11.     </h:head>
12.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
13.     <h:form id="formulaire">
14.         <!-- liens -->
15.         <h:panelGrid columns="2">
16.           <h:commandLink value="#{msg['form.langue1']}" action="#{changeLocale.setFrenchLocale}"/>
17.           <h:commandLink value="#{msg['form.langue2']}" action="#{changeLocale.setEnglishLocale}"/>
18.         </h:panelGrid>
19.         <h1><h:outputText value="#{msg['form1.titre']}"/></h1>
20.         <h:panelGrid columnClasses="col1,col2" columns="2" border="1">
21.           <h:outputText value="#{msg['form.headerCol1']}" styleClass="entete"/>
22.           <h:outputText value="#{msg['form.headerCol2']}" styleClass="entete"/>
23.           <!-- ligne 1 -->
24.           <h:outputText value="inputText"  styleClass="info"/>
25.           <h:panelGroup>
26.             <h:outputText value="#{msg['form.loginPrompt']}"/>
27.             <h:inputText id="inputText" value="#{form.inputText}"/>
28.           </h:panelGroup>
29.           <!-- ligne 2 -->
30.           <h:outputText value="inputSecret"  styleClass="info"/>
31.           <h:panelGroup>
32.             <h:outputText value="#{msg['form.passwdPrompt']}"/>
33.             <h:inputSecret id="inputSecret" value="#{form.inputSecret}"/>
34.           </h:panelGroup>
35.           <!-- ligne 3 -->
36.           <h:outputText value="inputTextArea" styleClass="info"/>
```

```
37.           <h:panelGroup>
38.             <h:outputText value="#{msg['form.descPrompt']}"/>
39.             <h:inputTextarea id="inputTextArea" value="#{form.inputTextArea}" rows="4"/>
40.           </h:panelGroup>
41.         </h:panelGrid>
42.         <!-- liens -->
43.         <h:panelGrid columns="6">
44.           <h:commandLink value="1" action="form1"/>
45.           <h:commandLink value="2" action="#{form.doAction2}"/>
46.           <h:commandLink value="3" action="form3"/>
47.           <h:commandLink value="4" action="#{form.doAction4}"/>
48.           <h:commandLink value="#{msg['form.pagealeatoireLink']}" action="#{form.doAlea}"/>
49.           <h:commandLink value="#{msg['form.exceptionLink']}" action="#{form.throwException}"/>
50.         </h:panelGrid>
51.       </h:form>
52.       </h:body>
53.   </f:view>
54. </html>
```

et correspond à l'affichage suivant :



On notera les points suivants :

- ligne 16, le tableau qui avait auparavant trois colonnes, n'en a plus que deux. La colonne 3 qui affichait les valeurs du modèle a été supprimée. C'est [form4.xhtml] qui les affichera,
- lignes 40-46 : un tableau de six liens. Les liens des lignes 44 et 46 ont une navigation statique : leur attribut **action** est codé en dur. Les autres liens ont une navigation dynamique : leur attribut **action** pointe sur une méthode du bean **form** chargée de rendre la clé de navigation. Les méthodes référencées dans [Form.java] sont les suivantes :

```
1.  // événements
2.    public String doAction2(){
3.      return "form2";
4.    }
5.
6.    public String doAction4(){
7.      return "form4";
8.    }
9.
10.   public String doAlea(){
11.     // un nombre aléatoire entre 1 et 3
12.     int i=1+(int)(3*Math.random());
13.     // on rend la clé de navigation
14.     return "form"+i;
15.   }
16.
17.   public String throwException() throws java.lang.Exception{
18.     throw new Exception("Exception test");
19. }
```

Nous ignorerons pour l'instant la méthode *throwException* de la ligne 17. Nous y reviendrons ultérieurement. Les méthodes *doAction2* et *doAction4* se contentent de rendre la clé de navigation sans faire de traitement. On aurait donc tout ausi bien pu écrire :

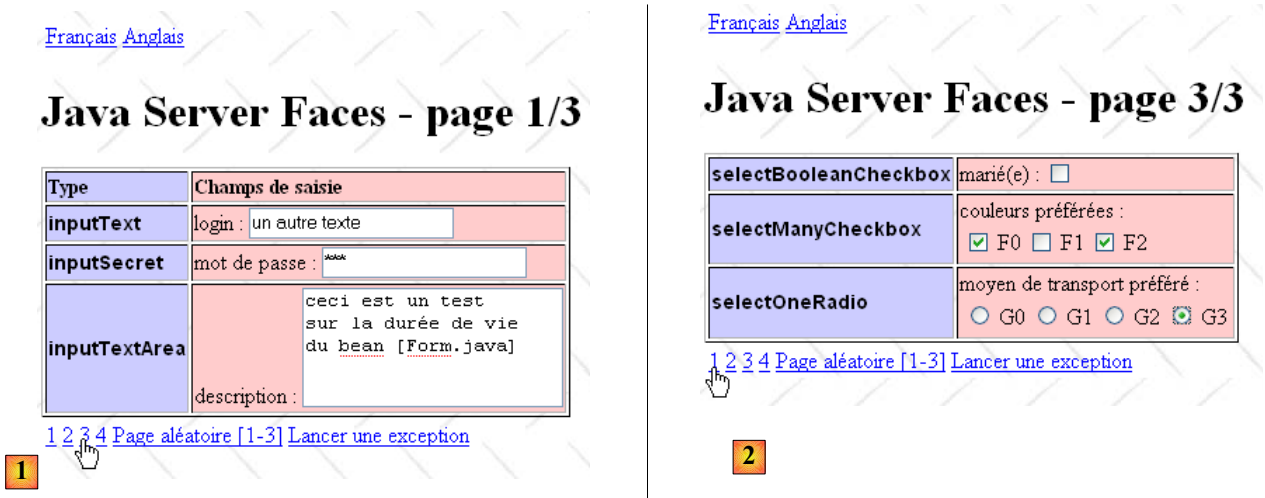
```
1.          <h:commandLink value="1" action="form1"/>
2.                  <h:commandLink value="2" action="form2"/>
3.                  <h:commandLink value="3" action="form3"/>
4.                  <h:commandLink value="4" action="form4"/>
5.                  <h:commandLink value="#{msg['form.pagealeatoireLink']}" action="#{form.doAlea}"/>
6.                  <h:commandLink value="#{msg['form.exceptionLink']}" action="#{form.throwException}"/>
```

La méthode **doAlea** génère, elle, une clé de navigation aléatoire qui prend sa valeur dans l'ensemble {"form1","form2","form3"}.

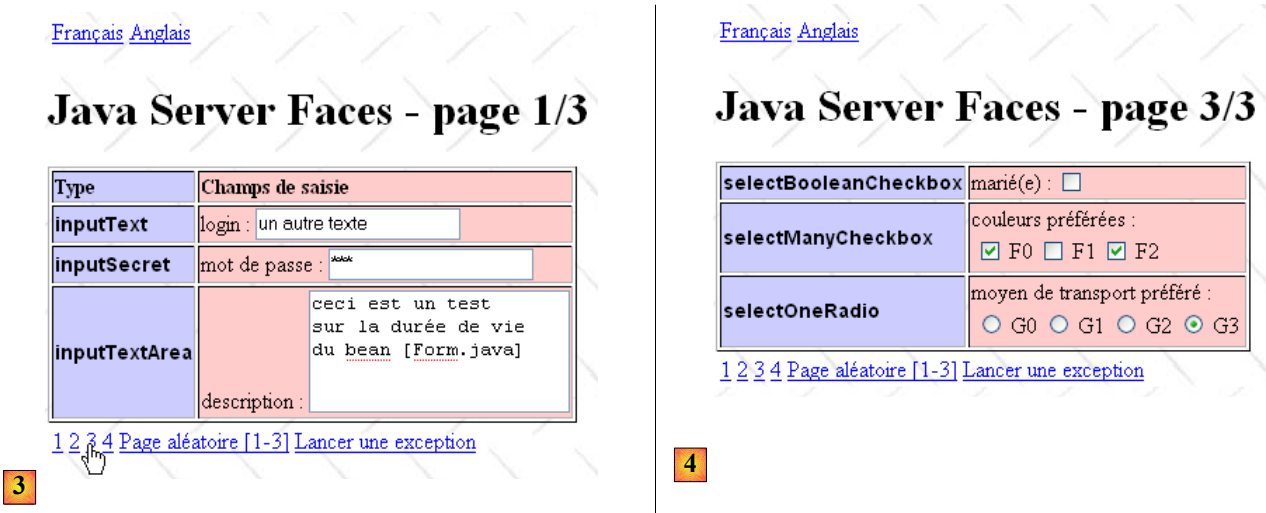
Le code des pages [form2.xhtml, form3.xhtml, form3.xhtml] est analogue à celle de la page [form1.xhtml].

2.7.3.2 Durée de vie du modèle [Form.java] des pages [form.xhtml]*

Considérons la séquence d'actions suivantes :



- en [1], on remplit la page 1 et on passe à la page 3,
- en [2], on remplit la page 3 et on revient à la page 1,



- en [3], on retrouve la page 1 telle qu'on l'a saisie. On revient alors à la page 3,
- en [4], la page 3 est retrouvée telle qu'elle a été saisie.

Le mécanisme du champ caché [javax.faces.ViewState] ne suffit pas à expliquer ce phénomène.

Lors du passage de [1] à [2], plusieurs étapes ont lieu :
- le modèle [Form.java] est mis à jour avec le POST de [form1.jsp]. Notamment, le champ **inputText** reçoit la valeur "*un autre texte*",
- la clé de navigation "**form3**" provoque l'affichage de [form3.xhtml]. Le *ViewState* embarqué dans [form3.xhtml] est l'état des seuls composants de [form3.xhtml] pas ceux de [form1.xhtml].

Lors du passage de [2] à [3] :
- le modèle [Form.java] est mis à jour avec le POST de [form3.xhtml]. Si la durée de vie du modèle [Form.java] est **request**, un objet [Form.java] tout neuf est créé avant d'être mis à jour par le POST de [form3.xhtml]. Dans ce cas, le champ **inputText** du modèle retrouve sa valeur par défaut :

```
private String inputText="texte";
```

  et la garde : en effet dans le POST de [form3.xhtml], rien ne vient mettre à jour le champ **inputText** qui fait partie du modèle de [form1.xhtml] et non de celui de [form3.xhtml],
- la clé de navigation "**form1**" provoque l'affichage de [form1.xhtml]. La page affiche son modèle. Dans notre cas, le champ de saisie *login* lié au modèle *inputText* affichera *texte* et non pas la valeur "*un autre texte*" saisie en [1]. Pour que le champ **inputText** garde la valeur saisie en [1], la durée de vie du modèle [Form.java] doit être **session** et non **request**. Dans ce cas,
  - à l'issue du POST de [form1.xhtml], le modèle sera placé dans la session du client. Le champ **inputText** aura la valeur "*un autre texte*",
  - au moment du POST de [form3.xhtml], le modèle sera recherché dans cette session et mis à jour par le POST de [form3.xhtml]. Le champ **inputText** ne sera pas mis à jour par ce POST mais gardera la valeur "*un autre texte*" acquise à l'issue du POST de [form1.xhtml] [1].

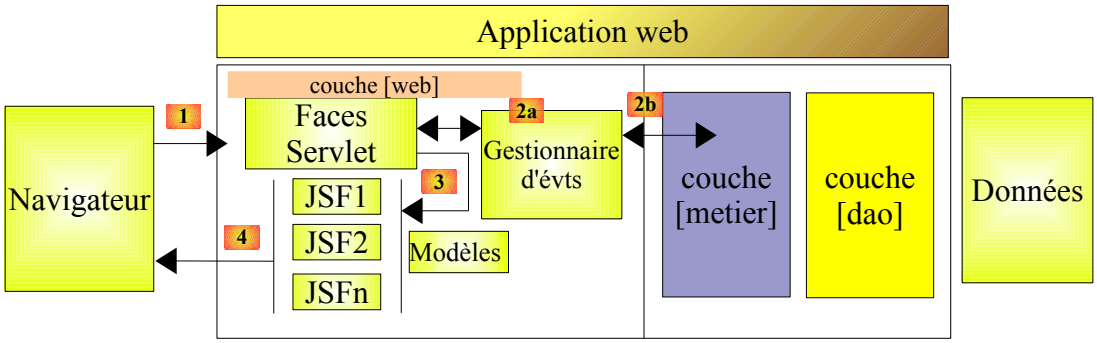
La déclaration du bean [Form.java] est donc la suivante :

```
1.  package forms;
2.
3.  import javax.faces.bean.SessionScoped;
4.  import javax.faces.bean.ManagedBean;
5.  import javax.faces.model.SelectItem;
6.
7.  @ManagedBean
8.  @SessionScoped
9.  public class Form {
```

La ligne 8 donne au bean une portée **session**.

2.7.4 Gestion des exceptions

Revenons sur l'architecture générale d'une application JSF :



Que se passe-t-il lorsque un gestionnaire d'événement ou bien un modèle récupère une exception provenant de la couche métier, une déconnexion imprévue d'avec une base de données par exemple ?

- les gestionnaires d'événements [2a] peuvent intercepter toute exception remontant de la couche [métier] et rendre au contrôleur [Faces Servlet] une clé de navigation vers une page d'erreur spécifique à l'exception,
- pour les modèles, cette solution n'est pas utilisable car lorsqu'ils sont sollicités [3,4], on est dans la phase de rendu d'une page XHTML précise et plus dans la phase de choix de celle-ci. Comment faire pour changer de page alors même qu'on est dans la phase de rendu de l'une d'elles ? Une solution simple mais qui ne convient pas toujours est de ne pas gérer l'exception qui va alors remonter jusqu'au conteneur de servlets qui exécute l'application. Celle-ci peut être configurée pour afficher une page particulière lorsqu'une exception remonte jusqu'au conteneur de servlets. Cette solution est toujours utilisable et nous l'examinons maintenant.

2.7.4.1 Configuration de l'application web pour la gestion des exceptions

La configuration d'une application web pour la gestion des exceptions se fait dans son fichier [web.xml] :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.    <context-param>
4.      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
5.      <param-value>client</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.PROJECT_STAGE</param-name>
9.      <param-value>Development</param-value>
10.   </context-param>
11.   <context-param>
12.     <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
13.     <param-value>true</param-value>
14.   </context-param>
15.   <servlet>
16.     <servlet-name>Faces Servlet</servlet-name>
17.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
18.     <load-on-startup>1</load-on-startup>
19.   </servlet>
20.   <servlet-mapping>
21.     <servlet-name>Faces Servlet</servlet-name>
22.     <url-pattern>/faces/*</url-pattern>
23.   </servlet-mapping>
24.   <session-config>
25.     <session-timeout>
26.       30
27.     </session-timeout>
28.   </session-config>
29.   <welcome-file-list>
30.     <welcome-file>faces/form1.xhtml</welcome-file>
31.   </welcome-file-list>
32.   <error-page>
33.     <error-code>500</error-code>
34.     <location>/faces/exception.xhtml</location>
35.   </error-page>
36.   <error-page>
37.     <exception-type>java.lang.Exception</exception-type>
38.     <location>/faces/exception.xhtml</location>
39.   </error-page>
40. </web-app>
```

Lignes 32-39, on trouve la définition de deux pages d'erreur. On peut avoir autant de balises <**error-page**> que nécessaires. La balise <**location**> indique la page à afficher en cas d'erreur. Le type de l'erreur associée à la page peut être défini de deux façons :

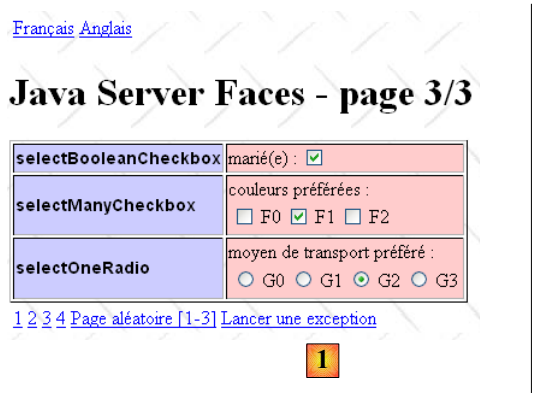
- par la balise <**exception-type**> qui définit le type Java de l'exception gérée. Ainsi la balise <error-page> des lignes 36-39 indique que si le conteneur de servlets récupère une exception de type [java.lang.Exception] ou dérivé (ligne 37) au cours de l'exécution de l'application, alors il doit faire afficher la page [/faces/exception.xhtml] (ligne 38). En prenant ici, le type d'exception le plus générique [java.lang.Exception], on s'assure de gérer toutes les exceptions,
- par la balise <**error-code**> (ligne 33) qui définit un code HTTP d'erreur. Par exemple, si un navigateur demande l'URL [http://machine:port/**contexte**/P] et que la page P n'existe pas dans l'application **contexte**, celle-ci n'intervient pas dans la réponse. C'est le conteneur de servlets qui génère cette réponse en envoyant une page d'erreur par défaut. La première ligne du flux HTTP de sa réponse contient un code d'erreur 404 indiquant que la page P demandée n'existe pas. On peut vouloir générer une réponse qui par exemple respecte la charte graphique de l'application ou qui donne des liens pour

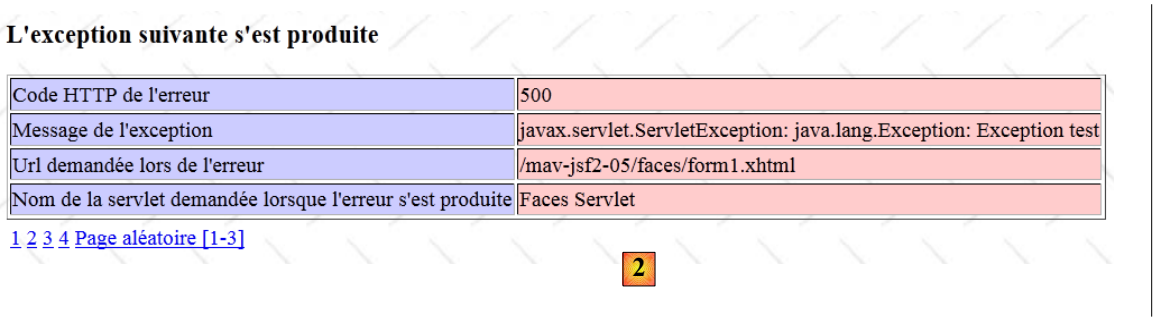
résoudre le problème. Dans ce cas, on utilisera une balise **<error-page>** avec une balise **<error-code>404</error-code>**.

Ci-dessus, le code HTTP d'erreur 500 est le code renvoyé en cas de " plantage " de l'application. C'est le code qui serait renvoyé si une exception remontait jusqu'au conteneur de servlets. Les deux balises <error-page> des lignes 28-35 sont donc probablement redondantes. On les a mises toutes les deux pour illustrer les deux façons de gérer une erreur.

2.7.4.2 La simulation de l'exception

Une exception est produite artificiellement par le lien [Lancer une exception] :





Un clic sur le lien [Lancer une exception] [1] provoque l'affichage de la page [2].

Dans le code des pages [formx.xhtml], le lien [Lancer une exception] est généré de la façon suivante :

```
1.  <!-- liens -->
2.          <h:panelGrid columns="6">
3.            <h:commandLink value="1" action="form1"/>
4.  ...
5.            <h:commandLink value="#{msg['form.exceptionLink']}" action="#{form.throwException}"/>
6.          </h:panelGrid>
```

Ligne 5, on voit que lors d'un clic sur le lien, la méthode **[form].throwException** va être exécutée. Celle-ci est la suivante :

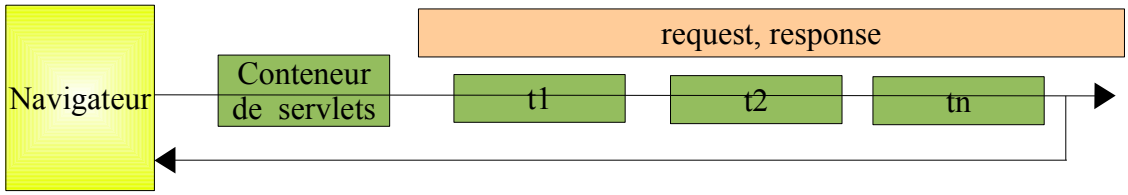
```
1.    public String throwException() throws java.lang.Exception{
2.      throw new Exception("Exception test");
3.  }
```

On y lance une exception de type [java.lang.Exception]. Elle va remonter jusqu'au conteneur de servlets qui va alors afficher la page [/faces/exception.xhtml].

2.7.4.3 Les informations liées à une exception

Lorsqu'une exception remonte jusqu'au conteneur de servlets, celui-ci va faire afficher la page d'erreur correspondante en transmettant à celle-ci des informations sur l'exception. Celles-ci sont placées comme **nouveaux attributs** de la requête en cours de traitement. La requête d'un navigateur et la réponse qu'il va recevoir sont encapsulées dans des objets Java de type

[HttpServletRequest request] et [HttpServletResponse response]. Ces objets sont disponibles à tous les étapes de traitement de la requête du navigateur.



A réception de la requête HTTP du navigateur, le conteneur de servlets encapsule celle-ci dans l'objet Java [HttpServletRequest request] et crée l'objet [HttpServletResponse response] qui va permettre de générer la réponse. Dans cet objet, on trouve notamment le canal TCP-IP à utiliser pour le flux HTTP de la réponse. Tous les couches t1, t2, ..., tn qui vont intervenir dans le traitement de l'objet *request*, ont accès à ces deux objets. Chacune d'elles peut avoir accès aux éléments de la requête initiale *request*, et préparer la réponse en enrichissant l'objet *response*. Une couche de *localisation* pourra par exemple fixer la *localisation* de la réponse par la méthode *response.setLocale(Locale l)*.

Les différentes couches **ti** peuvent se passer des informations via l'objet *request*. Celui-ci a un dictionnaire d'attributs, vide à sa création, qui peut être enrichi par les couches de traitement successives. Celles-ci peuvent mettre dans les attributs de l'objet *request* des informations nécessaires à la couche de traitement suivante. Il existe deux méthodes pour gérer les attributs de l'objet *request* :

- *void setAttribute(String s, Object o)* qui permet d'ajouter aux attributs un objet **o** identifié par la chaîne **s,**
- *Object getAttribute(String s)* qui permet d'obtenir l'attribut **o** identifié par la chaîne **s.**

Lorsqu'une exception remonte jusqu'au conteneur de servlets, ce dernier met les attributs suivants dans la requête en cours de traitement :

clé	valeur
javax.servlet.error.status_code	le code d'erreur HTTP qui va être renvoyé au client
javax.servlet.error.exception	le type Java de l'exception accompagné du message d'erreur.
javax.servlet.error.request_uri	l'URL demandée lorsque l'exception s'est produite
javax.servlet.error.servlet_name	la servlet qui traitait la requête lorsque l'exception s'est produite

Nous utiliserons ces attributs de la requête dans la page [exception.xhtml] pour les afficher.

2.7.4.4 La page d'erreur [exception.xhtml]

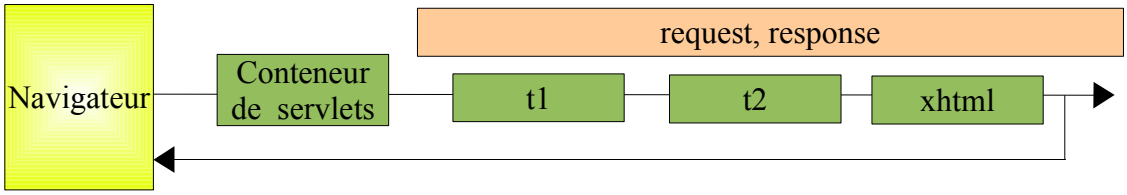
Son contenu est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:f="http://java.sun.com/jsf/core">
7.    <f:view locale="#{changeLocale.locale}">
8.      <h:head>
9.        <title>JSF</title>
10.       <h:outputStylesheet library="css" name="styles.css"/>
11.     </h:head>
12.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
13.       <h:form id="formulaire">
14.         <h3><h:outputText value="#{msg['exception.header']}"/></h3>
15.         <h:panelGrid columnClasses="col1,col2" columns="2" border="1">
16.           <h:outputText value="#{msg['exception.httpCode']}"/>
17.           <h:outputText value="#{requestScope['javax.servlet.error.status_code']}"/>
18.           <h:outputText value="#{msg['exception.message']}"/>
19.           <h:outputText value="#{requestScope['javax.servlet.error.exception']}"/>
20.           <h:outputText value="#{msg['exception.requestUri']}"/>
21.           <h:outputText value="#{requestScope['javax.servlet.error.request_uri']}"/>
22.           <h:outputText value="#{msg['exception.servletName']}"/>
23.           <h:outputText value="#{requestScope['javax.servlet.error.servlet_name']}"/>
```

```
24.         </h:panelGrid>
25.         <!-- liens -->
26.         <h:panelGrid columns="6">
27.           <h:commandLink value="1" action="form1"/>
28.           <h:commandLink value="2" action="#{form.doAction2}"/>
29.           <h:commandLink value="3" action="form3"/>
30.           <h:commandLink value="4" action="#{form.doAction4}"/>
31.           <h:commandLink value="#{msg['form.pagealeatoireLink']}" action="#{form.doAlea}"/>
32.         </h:panelGrid>
33.       </h:form>
34.     </h:body>
35.   </f:view>
36. </html>
```

2.7.4.4.1 Les expressions de la page d'exception

Dans la chaîne de traitement de la requête du client, la page XHTML est normalement le dernier maillon de la chaîne :



Tous les éléments de la chaîne sont des classes Java, y compris la page XHTML. Celle-ci est en effet transformée en servlet par le conteneur de servlets, c.a.d. en une classe Java normale. Plus précisément, la page XHTML est transformée en code Java qui s'exécute au sein de la méthode suivante :

```
1.  public void _jspService(HttpServletRequest request, HttpServletResponse response)
2.  throws java.io.IOException, ServletException {
3.
4.  JspFactory _jspxFactory = null;
5.  PageContext pageContext = null;
6.  HTTPSession session = null;
7.  ServletContext application = null;
8.  ServletConfig config = null;
9.  JspWriter out = null;
10. Object page = this;
11. JspWriter _jspx_out = null;
12. PageContext _jspx_page_context = null;
13. ...
14. ...code de la page XHTML
15.
```

A partir de la ligne 14, on trouvera le code Java image de la page XHTML. Ce code va disposer d'un certain nombre d'objets initialisés par la méthode **_jspService**, ligne 1 ci-dessus :

- ligne 1 : **HttpServletRequest request** : la requête en cours de traitement,
- ligne 1 : **HttpServletResponse response** : la réponse qui va être envoyée au client,
- ligne 7 : **ServletContext application** : un objet qui représente l'application web elle-même. Comme l'objet *request*, l'objet *application* peut avoir des attributs. Ceux-ci sont partagés par toutes les requêtes de tous les clients. Ce sont en général des attributs en lecture seule,
- ligne 6 : **HTTPSession session** : représente la session du client. Comme les objets *request* et *application*, l'objet *session* peut avoir des attributs. Ceux-ci sont partagés par toutes les requêtes d'un même client,
- ligne 9 : **JspWriter out** : un flux d'écriture vers le navigateur client. Cet objet est utile pour le débogage d'une page XHTML. Tout ce qui est écrit via *out.println(texte)* sera affiché dans le navigateur client.

Lorsque dans la page JSF, on écrit **#{expression}**, **expression** peut être **la clé** d'un attribut des objets **request**, **session** ou **application** ci-dessus. L'attribut correspondant est cherché successivement dans ces trois objets. Ainsi **#{clé}** est évaluée de la façon suivante :

1. **request.getAttribute(clé)**

2. **session.getAttribute(clé)**
3. **application.getAttribute(clé)**

Dès qu'une valeur non *null* est obtenue, l'évaluation de **#{clé}** est arrêtée. On peut vouloir être plus précis en indiquant le contexte dans lequel l'attribut doit être cherché :

- **#{requestScope['clé']}** pour chercher l'attribut dans l'objet **request,**
- **#{sessionScope['clé']}** pour chercher l'attribut dans l'objet **session,**
- **#{applicationScope['clé']}** pour chercher l'attribut dans l'objet **application.**

C'est ce qui a été fait dans la page [exception.xhtml] page 106. Les attributs utilisés sont les suivants :

clé	domaine	valeur
`javax.servlet.error.status_code`	request	cf paragraphe 2.7.4.3, page 106.
`javax.servlet.error.exception`	idem	idem
`javax.servlet.error.request_uri`	idem	idem
`javax.servlet.error.servlet_name`	idem	idem

Les différents messages nécessaires à la page JSF [exception.xhtml] ont été rajoutés aux fichiers de messages déjà existants :

[**messages_fr.properties**]

```
exception.header=L'exception suivante s'est produite
exception.httpCode=Code HTTP de l'erreur
exception.message=Message de l'exception
exception.requestUri=URL demandée lors de l'erreur
exception.servletName=Nom de la servlet demandée lorsque l'erreur s'est produite
```

[**messages_en.properties**]

```
exception.header=The following error occurred
exception.httpCode=HTTP error code
exception.message=Exception message
exception.requestUri=URL requested when error occurred
exception.servletName=Servlet requested when error occurred
```

2.8 Exemple mv-jsf2-06 : validation et conversion des saisies

2.8.1 L'application

L'application présente un formulaire de saisies. A la validation de celui-ci, le même formulaire est renvoyé en réponse, avec d'éventuels messages d'erreurs si les saisies ont été trouvées incorrectes.

1

Jsf - validations et conversions

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle
1-Nombre entier de type int	x		0
2-Nombre entier de type int	x		0
3-Nombre entier de type int	x		0
4-Nombre entier de type int dans l'intervalle [1,10]	100		0
5-Nombre réel de type double	x		0.0
6-Nombre réel>=0 de type double	-4		0.0
7-Booléen	true		true
8-Date au format jj/mm/aaaa	x		15/10/2007
9-Chaîne de 4 caractères	x		
10-Nombre entier de type int <1 ou >7	5		0
11-Nombre entier de type int	10		0
12-Nombre entier de type int	10		0

Valider Annuler

2

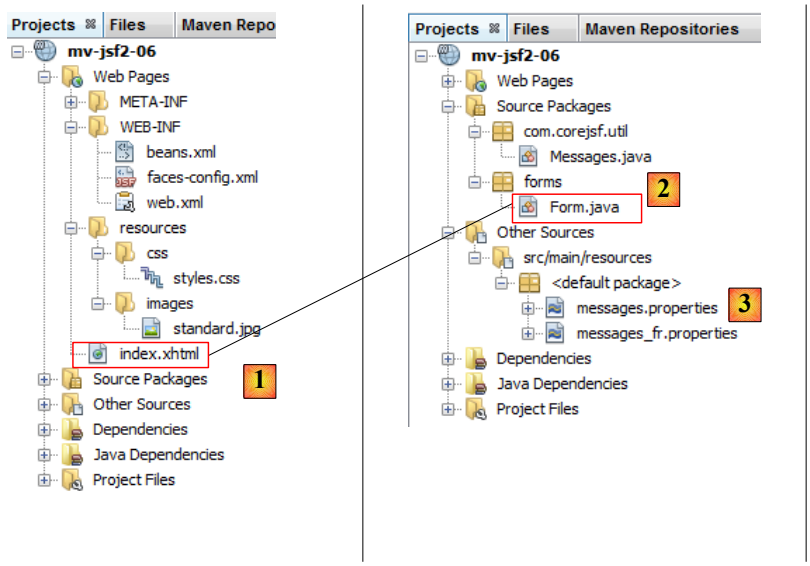
Jsf - validations et conversions

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	x	formulaire:saisie1: 'x' must be a number between -2147483648 and 2147483647 Example: 9346	0
2-Nombre entier de type int	x	formulaire:saisie2: 'x' must be a number consisting of one or more digits.	0
3-Nombre entier de type int	x	Vous devez entrer un nombre entier	0
4-Nombre entier de type int dans l'intervalle [1,10]	100	4-Vous devez entrer un nombre entier dans l'intervalle [1,10]	0
5-Nombre réel de type double	x	Vous devez entrer un nombre	0.0
6-Nombre réel>=0 de type double	-4	6-Vous devez entrer un nombre >=0	0.0
7-Booléen	true		true
8-Date au format jj/mm/aaaa	x	8-Vous devez entrer une date valide au format jj/mm/aaaa	11/10/2007
9-Chaîne de 4 caractères	x	9-Vous devez entrer une chaîne de 4 caractères exactement	
10-Nombre entier de type int <1 ou >7	5	10-Vous devez entrer un nombre entier <1 ou >7	0
11-Nombre entier de type int	10		0
12-Nombre entier de type int	10		0

Valider Annuler

2.8.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



Le projet [mv-jsf2-06] repose de nouveau sur une unique page [index.html] [1] et son modèle [Form.java] [2]. Il continue à utiliser des messages tirés de [messages.properties] mais uniquement en français [3]. L'option de changement de langue n'est pas offerte.

2.8.3 L'environnement de l'application

Nous donnons ici la teneur des fichiers qui configurent l'application sans donner d'explications particulières. Ces fichiers permettent de mieux comprendre ce qui suit.

[**faces-config.xml**]

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <resource-bundle>
12.       <base-name>
13.         messages
14.       </base-name>
15.       <var>msg</var>
16.     </resource-bundle>
17.     <message-bundle>messages</message-bundle>
18.   </application>
19. </faces-config>
```

La ligne 17 est nouvelle. Elle sera expliquée ultérieurement.

Le fichier des messages [**messages_fr.properties**]

```
1.  form.titre=Jsf - validations et conversions
2.  saisie1.prompt=1-Nombre entier de type int
3.  saisie2.prompt=2-Nombre entier de type int
4.  saisie3.prompt=3-Nombre entier de type int
5.  data.required=Vous devez entrer une donn\u00e9e
6.  integer.required=Vous devez entrer un nombre entier
```

```
saisie4.prompt=4-Nombre entier de type int dans l'intervalle [1,10]
saisie4.error=4-Vous devez entrer un nombre entier dans l'intervalle [1,10]
saisie5.prompt=5-Nombre r\u00e9el de type double
double.required=Vous devez entrer un nombre
saisie6.prompt=6-Nombre r\u00e9el>=0  de type double
saisie6.error=6-Vous devez entrer un nombre >=0
saisie7.prompt=7-Bool\u00e9en
saisie7.error=7-Vous devez entrer un bool\u00e9en
saisie8.prompt=8-Date au format jj/mm/aaaa
saisie8.error=8-Vous devez entrer une date valide au format jj/mm/aaaa
date.required=Vous devez entrer une date
saisie9.prompt=9-Cha\u00eene de 4 caract\u00e8res
saisie9.error=9-Vous devez entrer une cha\u00eene de 4 caract\u00e8res exactement
saisie9B.prompt=9B-Heure au format hh:mm
saisie9B.error=La cha\u00eene saisie ne respecte pas le format hh:mm
submit=Valider
cancel=Annuler
saisie.type=Type de la saisie
saisie.champ=Champ de saisie
saisie.erreur=Erreur de saisie
bean.valeur=Valeurs du mod\u00e8le du formulaire
saisie10.prompt=10-Nombre entier de type int <1 ou >7
saisie10.incorrecte=10-Saisie n\u00b0 10 incorrecte
saisie10.incorrecte_detail=10-Vous devez entrer un nombre entier <1 ou >7
saisies11et12.incorrectes=La propri\u00e9t\u00e9 saisie11+saisie12=10 n'est pas v\u00e9rifi\u00e9e
saisies11et12.incorrectes_detail=La propri\u00e9t\u00e9 saisie11+saisie12=10 n'est pas v\u00e9rifi\u00e9e
saisie11.prompt=11-Nombre entier de type int
saisie12.prompt=12-Nombre entier de type int
error.sign="!"
error.sign_detail="!"
```

La feuille de style [**styles.css**] est la suivante :

```
.info{
   font-family: Arial,Helvetica,sans-serif;
   font-size: 14px;
   font-weight: bold
}

.col1{
   background-color: #ccccff
}

.col2{
   background-color: #ffcccc
}

.col3{
   background-color: #ffcc66
}

.col4{
   background-color: #ccffcc
}

.error{
   color: #ff0000
}

.saisie{
   background-color: #ffcccc;
   border-color: #000000;
   border-width: 5px;
   color: #cc0033;
   font-family: cursive;
   font-size: 16px
}

.entete{
   font-family: 'Times New Roman',Times,serif;
   font-size: 14px;
   font-weight: bold
}
```

2.8.4 La page [index.xhtml] et son modèle [Form.java]

La page [index.xhtml] est la suivante :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>JSF</title>
    <h:outputStylesheet library="css" name="styles.css"/>
  </h:head>
  <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
    <h2><h:outputText value="#{msg['form.titre']}"/></h2>
    <h:form id="formulaire">
      <h:messages globalOnly="true" />
      <h:panelGrid columns="4" columnClasses="col1,col2,col3,col4" border="1">
        <!-- ligne 1 -->
        <h:outputText value="#{msg['saisie.type']}" styleClass="entete"/>
        <h:outputText value="#{msg['saisie.champ']}" styleClass="entete"/>
        <h:outputText value="#{msg['saisie.erreur']}" styleClass="entete"/>
        <h:outputText value="#{msg['bean.valeur']}" styleClass="entete"/>
        <!-- ligne 2 -->
        <h:outputText value="#{msg['saisie1.prompt']}"/>
        <h:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
        <h:message for="saisie1" styleClass="error"/>
        <h:outputText value="#{form.saisie1}"/>
        <!-- ligne 3 -->
        <h:outputText value="#{msg['saisie2.prompt']}" />
        <h:inputText id="saisie2" value="#{form.saisie2}"  styleClass="saisie"/>
        <h:message for="saisie2" showSummary="true" showDetail="false" styleClass="error"/>
        <h:outputText value="#{form.saisie2}"/>
        <!-- ligne 4 -->
        <h:outputText value="#{msg['saisie3.prompt']}" />
        <h:inputText id="saisie3" value="#{form.saisie3}"  styleClass="saisie" required="true" requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"/>
        <h:message for="saisie3" styleClass="error"/>
        <h:outputText value="#{form.saisie3}"/>
        <!-- ligne 5 -->
        <h:outputText value="#{msg['saisie4.prompt']}" />
        <h:inputText id="saisie4" value="#{form.saisie4}"  styleClass="saisie" required="true" requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}" validatorMessage="#{msg['saisie4.error']}">
          <f:validateLongRange minimum="1" maximum="10" />
        </h:inputText>
        <h:message for="saisie4" styleClass="error"/>
        <h:outputText value="#{form.saisie4}"/>
        <!-- ligne 6 -->
        ...
        <!-- ligne 7 -->
        ...
        <!-- ligne 8 -->
        ...
        <!-- ligne 9 -->
        ...
        <!-- ligne 10 -->
        ...
        <!-- ligne 11 -->
        ...
        <!-- ligne 12 -->
        ...
        <!-- ligne 13 -->
        ...
      </h:panelGrid>
      <!-- boutons de commande -->
      <h:panelGrid columns="2">
        <h:commandButton value="#{msg['submit']}" action="#{form.submit}"/>
        <h:commandButton value="#{msg['cancel']}" immediate="true" action="#{form.cancel}"/>
      </h:panelGrid>
    </h:form>
  </h:body>
```

```
67. </html>
```

La principale nouveauté vient de la présence de balises :

- pour afficher des messages d'erreurs <**h:messages**>(ligne 14), <**h:message**> (lignes 24, 29, 34),
- qui posent des contraintes de validité sur les saisies <**f:validateLongRange**> (ligne 39), <**f:validateDoubleRange**>, <**f:validateLength**>, <**f:validateRegex**>,
- qui définissent un convertisseur entre la saisie et son modèle comme <**f:convertDateTime**>.

Le modèle de cette page est la classe [Form.java] suivante :

```
package forms;

import com.corejsf.util.Messages;
import java.util.Date;
import javax.faces.bean.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

@ManagedBean
@RequestScoped
public class Form {

  public Form() {
  }
  // saisies
  private Integer saisie1 = 0;
  private Integer saisie2 = 0;
  private Integer saisie3 = 0;
  private Integer saisie4 = 0;
  private Double saisie5 = 0.0;
  private Double saisie6 = 0.0;
  private Boolean saisie7 = true;
  private Date saisie8 = new Date();
  private String saisie9 = "";
  private Integer saisie10 = 0;
  private Integer saisie11 = 0;
  private Integer saisie12 = 0;
  private String errorSaisie11 = "";
  private String errorSaisie12 = "";

  // actions
  public String submit() {
    ...
  }

  public String cancel() {
    ...
  }

  // validateurs
  public void validateSaisie10(FacesContext context, UIComponent component, Object value) {
    ...
  }
  // getters et setters
  ...
}
```

La nouveauté ici est que les champs du modèle ne sont plus uniquement de type *String* mais de types divers.

2.8.5 Les différentes saisies du formulaire

Nous étudions maintenant successivement les différentes saisies du formulaire.

2.8.5.1 Saisies 1 à 4 : saisie d'un nombre entier

La page [index.xhtml] présente la saisie 1 sous la forme suivante :

```
1.  <!-- ligne 2 -->
```

```
2.          <h:outputText value="#{msg['saisie1.prompt']}"/>
3.          <h:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
4.          <h:message for="saisie1" styleClass="error"/>
5.          <h:outputText value="#{form.saisie1}"/>
```

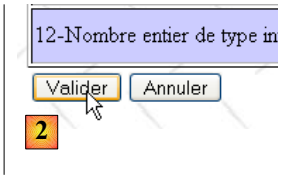
Le modèle *form.saisie1* est défini comme suit dans [Form.java] :

```
1.  private Integer saisie1 = 0;
```

Sur un GET du navigateur , la page [index.xhtml] associée à son modèle [Form.java] produit visuellement ce qui suit :

- la ligne 2 produit [1],
- la ligne 3 produit [2],
- la ligne 4 produit [3],
- la ligne 5 produit [4].

Supposons que la saisie suivante soit faite puis validée :

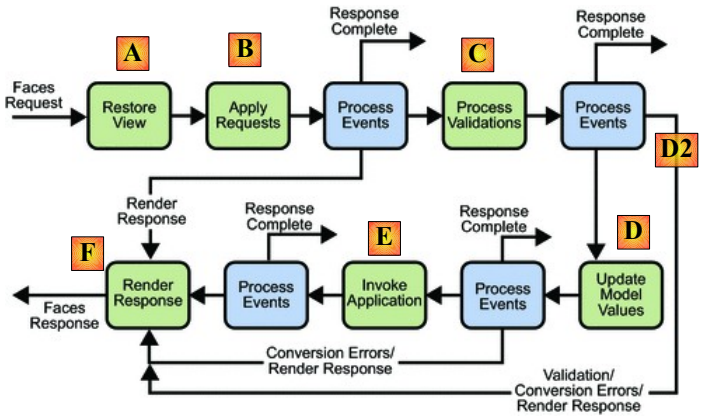


On obtient alors le résultat suivant dans le formulaire renvoyé par l'application :

Type de la saisie	Champ de saisie	Erreur de saisie 2	Valeurs du modèle du formulaire
1-Nombre entier de type int	x 1	formulaire:saisie1 : «x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0 3

- en [1], la saisie erronée,
- en [2], le message d'erreur le signalant,
- en [3], on voit que la valeur du champ *Integer saisie1* du modèle n'a pas changé.

Expliquons ce qui s'est passé. Pour cela revenons au cycle de traitement d'une page JSF :



Nous examinons ce cycle pour le composant :

```
<h:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
```

et son modèle :

```
private Integer saisie1 = 0;
```

- en [A], la page [index.xhtml] envoyée lors du GET du navigateur est restaurée. En [A], la page est telle que l'utilisateur l'a reçue. Le composant *id="saisie1"* retrouve sa valeur initiale "0",
- en [B], les composants de la page reçoivent pour valeurs, les valeurs postées par le navigateur. En [B], la page est telle que l'utilisateur l'a saisie et validée. Le composant *id="saisie1"* reçoit pour valeur, la valeur postée "x",
- en [C], si la page contient des validateurs et des convertisseurs explicites, ceux-ci sont exécutés. Des convertisseurs implicites sont également exécutés si le type du champ associé au composant n'est pas de type *String*. C'est le cas ici, où le champ *form.saisie1* est de type *Integer*. JSF va essayer de transformer la valeur "x" du composant *id="saisie1"* en un type *Integer*. Ceci va provoquer une erreur qui va arrêter le cycle de traitement [A-F]. Cette erreur sera associée au composant *id="saisie1"*. Via [D2], on passe ensuite directement à la phase de rendu de la réponse. La même page [index.xhtml] est renvoyée,
- la phase [D] n'a lieu que si tous les composants d'une page ont passé la phase de conversion / validation. C'est dans cette phase, que la valeur du composant *id="saisie1"* sera affectée à son modèle *form.saisie1*.

Si la phase [C] échoue, la page est réaffichée et le code suivant est exécuté de nouveau :

```
1.  <!-- ligne 2 -->
2.          <h:outputText value="#{msg['saisie1.prompt']}"/>
3.          <h:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
4.          <h:message for="saisie1" styleClass="error"/>
5.  <h:outputText value="#{form.saisie1}"/>
```

Type de la saisie	Champ de saisie	Erreur de saisie 2	Valeurs du modèle du formulaire
1-Nombre entier de type int	x 1	formulaire:saisie1 : «x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0 3

Le message affiché en [2] vient de la ligne 4 de [index.xhtml]. La balise **<h:message for="idComposant"/>** affiche le message d'erreur associé au composant désigné par l'attribut **for**, si erreur il y a. Le message affiché en [2] est standard et se trouve dans le fichier [javax/faces/Messages.properties] de l'archive [jsf-api.jar] :

En [2], on voit que le fichier des messages existe en plusieurs variantes. Examinons le contenu de [Messages_fr.properties] :

```
1.  ...
2.  # ========================================================================
3.  # Component Errors
4.  # ========================================================================
5.  javax.faces.component.UIInput.CONVERSION={0} : une erreur de conversion est survenue.
6.  javax.faces.component.UIInput.REQUIRED={0} : erreur de validation. Vous devez indiquer une valeur.
7.  javax.faces.component.UIInput.UPDATE={0} : une erreur est survenue lors du traitement des
    informations que vous avez soumises.
8.  javax.faces.component.UISelectOne.INVALID={0} : erreur de validation. La valeur est incorrecte.
9.  javax.faces.component.UISelectMany.INVALID={0} : erreur de validation. La valeur est incorrecte.
10.
11. # ========================================================================
12. # Converter Errors
13. # ========================================================================
14. ...
```

```
15. javax.faces.converter.FloatConverter.FLOAT={2} : «{0}» doit être un nombre composé d'un ou de
    plusieurs chiffres.
16. javax.faces.converter.FloatConverter.FLOAT_detail={2} : «{0}» doit être un nombre compris entre
    1.4E-45 et 3.4028235E38. Exemple : {1}
17. javax.faces.converter.IntegerConverter.INTEGER={2} : «{0}» doit être un nombre composé d'un ou de
    plusieurs chiffres.
18. javax.faces.converter.IntegerConverter.INTEGER_detail={2} : «{0}» doit être un nombre compris
    entre -2147483648 et 2147483647. Exemple : {1}
19. ...
20.
21.
22. # ========================================================================
23. # Validator Errors
24. # ========================================================================
25. javax.faces.validator.DoubleRangeValidator.MAXIMUM={1} : erreur de validation. La valeur est
    supérieure à la valeur maximale autorisée, "{0}".
26. javax.faces.validator.DoubleRangeValidator.MINIMUM={1} : erreur de validation. La valeur est
    inférieure à la valeur minimale autorisée, "{0}".
27. javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE={2} : erreur de validation. L'attribut
    spécifié n'est pas compris entre les valeurs attendues {0} et {1}.
28. javax.faces.validator.DoubleRangeValidator.TYPE={0} : erreur de validation. La valeur n'est pas du
    type correct.
29. ...
```

Le fichier contient des messages divisés en catégories :

- erreurs sur un composant, ligne 3,
- erreurs de conversion entre un composant et son modèle, ligne 12
- erreurs de validation lorsque des validateurs sont présents dans la page, ligne 23.

L'erreur qui s'est produite sur le composant *id="saisie1"* est de type *erreur de conversion* d'un type *String* vers un type *Integer*. Le message d'erreur associé est celui de la ligne 18 du fichier des messages.

```
javax.faces.converter.IntegerConverter.INTEGER_detail={2} : «{0}» doit être un nombre compris entre
-2147483648 et 2147483647. Exemple : {1}
```

Le message d'erreur qui a été affiché est reproduit ci-dessous :

Type de la saisie	Champ de saisie	Erreur de saisie 2	Valeurs du modèle du formulaire
1-Nombre entier de type int	x 1	formulaire:saisie1 : «x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0 3

On voit que dans le message :

- le paramètre {2} a été remplacé par l'identifiant du composant pour lequel l'erreur de conversion s'est produite,
- la paramètre {0} a été remplacé par la saisie faite en [1] pour le composant,
- le paramètre {1} a été remplacé par le nombre 9346.

La plupart des messages liés aux composants ont deux versions : une version résumée (summary) et une version détaillée (detail). C'est le cas des lignes 16-18 :

```
1.  javax.faces.converter.IntegerConverter.INTEGER={2} : «{0}» doit être un nombre composé d'un ou de
    plusieurs chiffres.
2.  javax.faces.converter.IntegerConverter.INTEGER_detail={2} : «{0}» doit être un nombre compris
    entre -2147483648 et 2147483647. Exemple : {1}
```

Le message ayant la clé *_detail* (ligne 2) est le message dit **détaillé**. L'autre est le message dit **résumé**. La balise <h:message> affiche par défaut le message détaillé. Ce comportement peut être changé par les attributs **showSummary** et **showDetail**. C'est ce qui est fait pour le composant d'id *saisie2 :*

```
1.          <!-- ligne 3 -->
2.          <h:outputText value="#{msg['saisie2.prompt']}" />
3.          <h:inputText id="saisie2" value="#{form.saisie2}"  styleClass="saisie"/>
4.          <h:message for="saisie2" showSummary="true" showDetail="false" styleClass="error"/>
5.  <h:outputText value="#{form.saisie2}"/>
```

Ligne 2, le composant *saisie2* est lié au champ *form.saisie2* suivant :

```
  private Integer saisie2 = 0;
```

Le résultat obenu est le suivant :

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	0x	formulaire:saisie1 : «0x» doit être un nombre compris entre -2147483648 et 2147[1]647. Exemple : 9346	0
2-Nombre entier de type int	0x	formulaire:saisie2 : «0x» doit être un nombre composé d'u[2] de plusieurs chiffres.	0

- en [1], le message détaillé, en [2], le message résumé.

La balise <**h:messages**> affiche sous la forme d'une liste tous les messages d'erreurs **résumés** de tous les composants ainsi que les messages d'erreurs **non liés** à un composant. Là encore des attributs peuvent changer ce comportement par défaut :

- **showDetail** : true / false pour demander ou non les messages détaillés,
- **showSummary** : true / false pour demander ou non les messages résumés,
- **globalOnly** : true / false pour demander à ne voir ou non que les messages d'erreurs non liées à des composants. Un tel message pourrait, par exemple, être créé par le développeur.

Le message d'erreur associé à une conversion peut être changé de diverses façons. Tout d'abord, on peut indiquer à l'application d'utiliser un autre fichier de messages. Cette modification se fait dans [faces-config.xml] :

```
1.  <faces-config ...">
2.    <application>
3.      <resource-bundle>
4.        <base-name>
5.          messages
6.        </base-name>
7.        <var>msg</var>
8.      </resource-bundle>
9.      <message-bundle>messages</message-bundle>
10.   </application>
11. ...
12. </faces-config>
```

Les lignes 3-8 définissent un fichier des messages mais ce n'est pas celui-ci qui est utilisé par les balises **<h:message>** et **<h:messages>**. Il faut utiliser la balise **<message-bundle>** de la ligne 9 pour le définir. La ligne 9 indique aux balises <h:message(s)> que le fichier [messages.properties] doit être exploré avant le fichier [javax.faces.Messages.properties]. Ainsi si on ajoute les lignes suivantes au fichier [messages_fr.properties] :

```
1.  # conversions
2.  javax.faces.converter.IntegerConverter.INTEGER=erreur
3.  javax.faces.converter.IntegerConverter.INTEGER_detail=erreur d\u00e9taill\u00e9e
```

l'erreur renvoyée pour les composants *saisie1* et *saisie2* devient :

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	0x	erreur détaillée	0
2-Nombre entier de type int	0x	erreur	0

L'autre façon de modifier le message d'erreur de conversion est d'utiliser l'attribut **converterMessage** du composant comme ci-dessous pour le composant *saisie3* :

```
1.          <!-- ligne 4 -->
2.          <h:outputText value="#{msg['saisie3.prompt']}" />
3.          <h:inputText id="saisie3" value="#{form.saisie3}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"/>
4.          <h:message for="saisie3" styleClass="error"/>
```

```
5.  <h:outputText value="#{form.saisie3}"/>
```

Le composant *saisie3* est lié au champ *form.saisie3* suivant :

```
  private Integer saisie3 = 0;
```

- ligne 3, l'attribut **converterMessage** fixe explicitement le message à afficher lors d'une erreur de conversion,
- ligne 3, l'attribut **required="true"** indique que la saisie est obligatoire. Le champ ne peut rester vide. Un champ est considéré comme vide s'il ne contient aucun caractère ou s'il contient une suite d'espaces. Là encore, il existe dans [javax.faces.Messages.properties] un message par défaut :

```
javax.faces.component.UIInput.REQUIRED={0} : erreur de validation. Vous devez indiquer une valeur.
```

L'attribut **requiredMessage** permet de remplacer ce message par défaut. Si le fichier [messages.properties] contient les messages suivants :

```
1.  ...
2.  data.required=Vous devez entrer une donnée
3.  integer.required=Vous devez entrer un nombre entier
4.
```

on pourra obtenir le résultat suivant :

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	0x	formulaire:saisie1 : «0x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0
2-Nombre entier de type int	0x	formulaire:saisie2 : «0x» doit être un nombre composé d'un ou de plusieurs chiffres.	0
3-Nombre entier de type int		Vous devez entrer une donnée	0

ou encore celui-ci :

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	0x	formulaire:saisie1 : «0x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0
2-Nombre entier de type int	0x	formulaire:saisie2 : «0x» doit être un nombre composé d'un ou de plusieurs chiffres.	0
3-Nombre entier de type int	x	Vous devez entrer un nombre entier	0

Vérifier qu'une saisie correspond bien à un nombre entier n'est pas toujours suffisant. Il faut parfois vérifier que le nombre saisi appartient à un intervalle donné. On utilise alors un **validateur**. La saisie n° 4 en donne un exemple. Son code dans [index.xhtml] est le suivant :

```
1.          <!-- ligne 5 -->
2.          <h:outputText value="#{msg['saisie4.prompt']}" />
3.          <h:inputText id="saisie4" value="#{form.saisie4}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"
    validatorMessage="#{msg['saisie4.error']}">
4.            <f:validateLongRange minimum="1" maximum="10" />
5.          </h:inputText>
6.          <h:message for="saisie4" styleClass="error"/>
7.  <h:outputText value="#{form.saisie4}"/>
```

Ligne 3, le composant *saisie4* est lié au modèle *form.saisie4* suivant :

```
  private Integer saisie4 = 0;
```

Lignes 3-5, la balise **<h:inputText>** a une balise enfant **<f:validateLongRange>** qui admet deux attributs facultatifs **minimum** et **maximum**. Cette balise, appelée également **validateur**, permet d'ajouter une contrainte à la valeur de la saisie : ce doit être non

seulement un entier, mais un entier dans l'intervalle [*minimum*, *maximum*] si les deux attributs *minimum* et *maximum* sont présents, supérieure ou égale à *minimum* si seul l'attribut *minimum* est présent, inférieure ou égale à *maximum* si seul l'attribut *maximum* est présent. Le validateur <**f:validateLongRange**> a des messages d'erreur par défaut dans [javax.faces.Messages.properties] :

```
1. javax.faces.validator.LongRangeValidator.MINIMUM={1} : erreur de validation. La valeur est
   inférieure à la valeur minimale autorisée, "{0}".
2. javax.faces.validator.LongRangeValidator.NOT_IN_RANGE={2} : erreur de validation. L'attribut
   spécifié n'est pas compris entre les valeurs attendues {0} et {1}.
3. javax.faces.validator.LongRangeValidator.TYPE={0} : erreur de validation. La valeur n'est pas du
   type correct.
```

De nouveau, il est possible de remplacer ces messages par d'autres. Il existe un attribut **validatorMessage** qui permet de définir un message spécifique pour le composant. Ainsi avec le code JSF suivant :

```
1.          <h:inputText id="saisie4" value="#{form.saisie4}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"
    validatorMessage="#{msg['saisie4.error']}">
2.            <f:validateLongRange minimum="1" maximum="10" />
3.  </h:inputText>
```

et le message suivant dans [messages.properties] :

```
saisie4.error=4-Vous devez entrer un nombre entier dans l'intervalle [1,10]
```

on obtient le résultat suivant :

Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	x	formulaire:saisie1 : «x» doit être un nombre compris entre -2147483648 et 2147483647. Exemple : 9346	0
2-Nombre entier de type int	x	formulaire:saisie2 : «x» doit être un nombre composé d'un ou de plusieurs chiffres.	0
3-Nombre entier de type int	x	Vous devez entrer un nombre entier	0
4-Nombre entier de type int dans l'intervalle [1,10]	20	4-Vous devez entrer un nombre entier dans l'intervalle [1,10]	0

2.8.5.2 Saisies 5 et 6 : saisie d'un nombre réel

La saisie des nombres réels obéit à des règles similaires à celles de la saisie des nombres entiers. Le code XHTML des saisies 5 et 6 est le suivant :

```
1.  <!-- ligne 6 -->
2.          <h:outputText value="#{msg['saisie5.prompt']}" />
3.          <h:inputText id="saisie5" value="#{form.saisie5}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['double.required']}"/>
4.          <h:message for="saisie5" styleClass="error"/>
5.          <h:outputText value="#{form.saisie5}"/>
6.          <!-- ligne 7 -->
7.          <h:outputText value="#{msg['saisie6.prompt']}"/>
8.          <h:inputText id="saisie6" value="#{form.saisie6}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['double.required']}"
    validatorMessage="#{msg['saisie6.error']}">
9.            <f:validateDoubleRange minimum="0.0"/>
10.         </h:inputText>
11.         <h:message for="saisie6" styleClass="error"/>
12.         <h:outputText value="#{form.saisie6}"/>
```

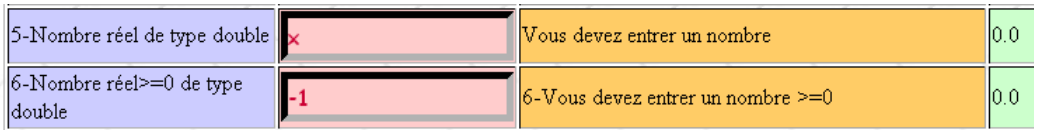
Les éléments du modèle [Form.java] liés aux composants *saisie5* et *saisie6* :

```
private Double saisie5 = 0.0;
private Double saisie6 = 0.0;
```

Les messages d'erreurs associés aux convertisseurs et validateurs des composants *saisie5* et *saisie6*, dans [messages.properties] :

```
1.  double.required=Vous devez entrer un nombre
2.  saisie6.error=6-Vous devez entrer un nombre >=0
```

Voici un exemple d'exécution :



2.8.5.3 Saisie 7 : saisie d'un booléen

La saisie d'un booléen devrait être normalement faite avec une case à cocher. Si elle est faite avec un champ de saisie, la chaîne "**true**" est convertie en booléen **true** et tout autre chaîne en booléen **false**.

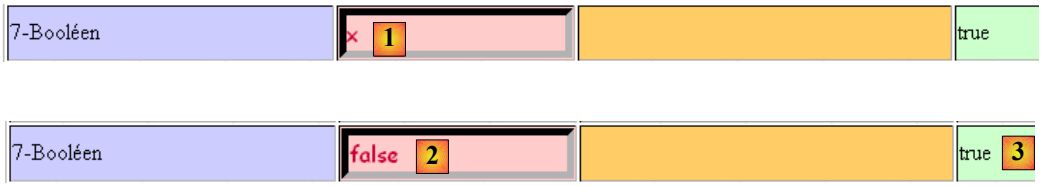
Le code XHTML de l'exemple :

```
1.  <!-- ligne 8 -->
2.          <h:outputText value="#{msg['saisie7.prompt']}"/>
3.          <h:inputText id="saisie7" value="#{form.saisie7}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['double.required']}"/>
4.          <h:message for="saisie7" styleClass="error"/>
5.          <h:outputText value="#{form.saisie7}"/>
```

Le modèle du composant *saisie7* :

```
private Boolean saisie7 = true;
```

Voici un exemple de saisie et sa réponse :



En [1], la valeur saisie. Par conversion, cette chaîne "x" devient le booléen **false**. C'est ce que montre [2]. La valeur [3] du modèle n'a pas changé. Elle ne change que lorsque toutes les conversions et validations de la page ont réussi. Ce n'était pas le cas dans cet exemple.

2.8.5.4 Saisie 8 : saisie d'une date

La saisie d'une date est faite dans l'exemple avec le code XHTML suivant :

```
1.  <!-- ligne 9 -->
2.          <h:outputText value="#{msg['saisie8.prompt']}"/>
3.          <h:inputText id="saisie8" value="#{form.saisie8}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['date.required']}" converterMessage="#{msg['saisie8.error']}">
4.            <f:convertDateTime pattern="dd/MM/yyyy"/>
5.          </h:inputText>
6.          <h:message for="saisie8" styleClass="error"/>
7.          <h:outputText value="#{form.saisie8}">
8.            <f:convertDateTime pattern="dd/MM/yyyy"/>
9.          </h:outputText>
```

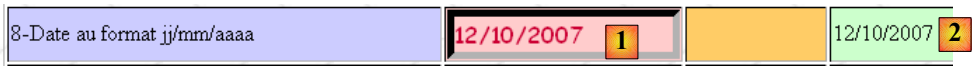
Le composant *saisie8* de la ligne 3 utilise un convertisseur *java.lang.String* <--> *java.util.Date*. Le modèle *form.saisie8* associé au composant *saisie8* est le suivant :

```
private Date saisie8 = new Date();
```

Le composant défini par les lignes 7-9 utilise également un convertisseur mais uniquement dans le sens *java.util.Date* --> *java.lang.String*.

Le convertisseur **<f:convertDateTime>** admet divers attributs dont l'attribut **pattern** qui fixe la forme de la chaîne de caractères qui doit être transformée en date où avec laquelle une date doit être affichée.

Lors de la demande initiale de la page [index.xhtml], la ligne 8 précédente s'affiche comme suit :



Les champs [1] et [2] affichent tous deux la valeur du modèle *form.saisie8* :

```
  private Date saisie8 = new Date();
```

où *saisie8* prend pour valeur la date du jour. Le convertisseur utilisé dans les deux cas pour l'affichage de la date est le suivant :

```
            <f:convertDateTime pattern="dd/MM/yyyy"/>
```

où dd (day) désigne le n° du jour, MM (Month) le n° du mois et yyyy (year) l'année. En [1], le convertisseur est utilisé pour la conversion inverse *java.lang.String --> java.util.Date*. La date saisie devra donc suivre le modèle "dd/MM/yyyy" pour être valide.

Il existe des messages par défaut pour les dates non valides dans [javax.faces.Messages.properties] :

```
1. javax.faces.converter.DateTimeConverter.DATE={2} : «{0}» n'a pas pu être interprété en tant que
   date.
2. javax.faces.converter.DateTimeConverter.DATE_detail={2} : «{0}» n'a pas pu être interprété en tant
   que date. Exemple : {1}
```

qu'on peut remplacer par ses propres messages. Ainsi dans l'exemple :

```
1. <h:inputText     id="saisie8"     value="#{form.saisie8}"         styleClass="saisie"     required="true"
   requiredMessage="#{msg['date.required']}" converterMessage="#{msg['saisie8.error']}">
2.   <f:convertDateTime pattern="dd/MM/yyyy"/>
3. </h:inputText>
```

le message affiché en cas d'erreur de conversion sera le message de clé *saisie8.error* suivant :

```
saisie8.error=8-Vous devez entrer une date valide au format jj/mm/aaaa
```

Voici un exemple :

2.8.5.5 Saisie 9 : saisie d'une chaîne de longueur contrainte

La saisie 9 montre comment imposer à une chaîne saisie d'avoir un nombre de caractères compris dans un intervalle :

```
1.  <!-- ligne 10 -->
2.          <h:outputText value="#{msg['saisie9.prompt']}"/>
3.          <h:inputText id="saisie9" value="#{form.saisie9}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" validatorMessage="#{msg['saisie9.error']}">
4.            <f:validateLength minimum="4" maximum="4"/>
5.          </h:inputText>
6.          <h:message for="saisie9" styleClass="error"/>
7.          <h:outputText value="#{form.saisie9}"/>
```

Ligne 4, le validateur **<f:validateLength minimum="4" maximum="4"/>** impose à la chaîne saisie d'avoir exactement 4 caractères. On peut n'utiliser que l'un des attributs : **minimum** pour un nombre minimal de caractères, **maximum** pour un nombre maximal.

Le modèle *form.saisie9* du composant *saisie9* de la ligne 3 est le suivant :

```
  private String saisie9 = "";
```

Il existe des messages d'erreur par défaut pour ce type de validation :

```
1.  javax.faces.validator.LengthValidator.MAXIMUM={1} : erreur de validation. La longueur est
    supérieure à la valeur maximale autorisée, "{0}".
2.  javax.faces.validator.LengthValidator.MINIMUM={1} : erreur de validation. La longueur est
    inférieure à la valeur minimale autorisée, "{0}".
```

qu'on peut remplacer en utilisant l'attribut **validatorMessage** comme dans la ligne 3 ci-dessus. Le message de clé *saisie9.error* est le suivant :

```
saisie9.error=9-Vous devez entrer une chaîne de 4 caractères exactement
```

Voici un exemple d'exécution :

2.8.5.6 Saisie 9B : saisie d'une chaîne devant se conformer à un modèle

La saisie 9B montre comment imposer à une chaîne saisie d'avoir un nombre de caractères compris dans un intervalle :

```
1.  <!-- ligne 10B -->
2.          <h:outputText value="#{msg['saisie9B.prompt']}"/>
3.          <h:inputText id="saisie9B" value="#{form.saisie9B}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" validatorMessage="#{msg['saisie9B.error']}">
4.            <f:validateRegex pattern="^\s*\d{2}:\d{2}\s*$"/>
5.          </h:inputText>
6.          <h:message for="saisie9B" styleClass="error"/>
7.          <h:outputText value="#{form.saisie9B}"/>
```

Ligne 4, le validateur `<f:validateRegex pattern="^\s*\d{2}:\d{2}\s*$"/>` impose à la chaîne saisie de correspondre au modèle d'une expression régulière, ici : une suite de 0 ou davantage d'espaces, 2 chiffres, le signe :, 2 chiffres, une suite de 0 ou davantage d'espaces.

Le modèle *form.saisie9B* du composant *saisie9B* de la ligne 3 est le suivant :

```
private String saisie9B;
```

Il existe des messages d'erreur par défaut pour ce type de validation :

```
1.  javax.faces.validator.RegexValidator.PATTERN_NOT_SET=Le modèle d'expression régulière doit être
    défini.
2.  javax.faces.validator.RegexValidator.PATTERN_NOT_SET_detail=La valeur définie du modèle
    d'expression régulière ne peut pas être vide.
3.  javax.faces.validator.RegexValidator.NOT_MATCHED=Discordance du modèle d'expression régulière.
4.  javax.faces.validator.RegexValidator.NOT_MATCHED_detail=Discordance du modèle d'expression
    régulière «{0}».
5.  javax.faces.validator.RegexValidator.MATCH_EXCEPTION=Erreur dans l'expression régulière.
6.  javax.faces.validator.RegexValidator.MATCH_EXCEPTION_detail=Erreur dans l'expression régulière,
    «{0}»
```

qu'on peut remplacer en utilisant l'attribut **validatorMessage** comme dans la ligne 3 ci-dessus. Le message de clé *saisie9.error* est le suivant :

```
saisie9B.error=La cha\u00eene saisie ne respecte pas le format hh:mm
```

Voici un exemple d'exécution :

2.8.5.7 *Saisie 10 : écrire une méthode de validation spécifique*

Résumons : JSF permet de vérifier parmi les valeurs saisies, la validité des nombres (entiers, réels), des dates, la longueur des chaînes et la conformité d'une saisie vis à vis d'une expression régulière. JSF permet d'ajouter aux validateurs et convertisseurs existants ses propres validateurs et convertisseurs. Ce point n'est pas abordé ici mais on pourra lire **[ref2]** pour l'approfondir.

Nous présentons ici une autre méthode : celle qui consiste à valider une donnée saisie par une méthode du modèle du formulaire. C'est l'exemple suivant :

```
1.  <!-- ligne 11 -->
2.          <h:outputText value="#{msg['saisie10.prompt']}"/>
3.          <h:inputText id="saisie10" value="#{form.saisie10}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" validator="#{form.validateSaisie10}"/>
4.          <h:message for="saisie10" styleClass="error"/>
5.          <h:outputText value="#{form.saisie10}"/>
```

Le modèle *form.saisie10* associé au composant *saisie10* de la ligne 3 est le suivant :

```
  private Integer saisie10 = 0;
```

On veut que le nombre saisi soit <1 ou >7. On ne peut le vérifier avec les validateurs de base de JSF. On écrit alors sa propre méthode de validation du composant *saisie10*. On l'indique avec l'attribut **validator** du composant à valider :

```
          <h:inputText id="saisie10" value="#{form.saisie10}"  styleClass="saisie" required="true"
requiredMessage="#{msg['data.required']}" validator="#{form.validateSaisie10}"/>
```

Le composant *saisie10* est validé par la méthode *form.validateSaisie10*. Celle-ci est la suivante :

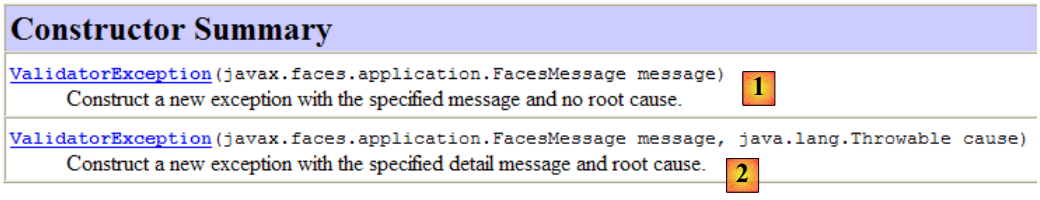
```
1.    public void validateSaisie10(FacesContext context, UIComponent component, Object value) {
2.      int saisie = (Integer) value;
3.      if (!(saisie < 1 || saisie > 7)) {
4.        FacesMessage message = Messages.getMessage(null, "saisie10.incorrecte", null);
5.        message.setSeverity(FacesMessage.SEVERITY_ERROR);
6.        throw new ValidatorException(message);
7.      }
8.  }
```

La signature d'une méthode de validation est obligatoirement celle de la ligne 1 :

- **FacesContext context** : contexte d'exécution de la page - donne accès à diverses informations, notamment aux objets *HttpServletRequest request* et *HttpServletResponse response,*
- **UIComponent component** : le composant qu'il faut valider. La balise <*h:inputText*> est représenté par un composant de type *UIInput* dérivé de *UIComponent*. Ici c'est ce composant *UIInput* qui est reçu en second paramètre,
- **Object value** : la valeur saisie à vérifier, transformée dans le type de son modèle. Il est important de comprendre ici que si la conversion *String -> type du modèle* a échoué, alors la méthode de validation n'est pas exécutée. Lorsqu'on arrive dans la méthode *validateSaisie10*, c'est que la conversion *String --> Integer* a réussi. Le troisième paramètre est alors de type *Integer*.

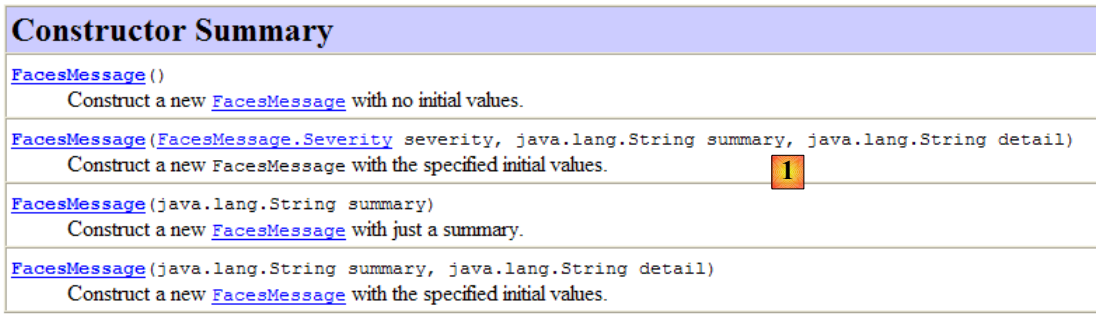
- ligne 2 : la valeur saisie est transformée en type *int,*
- ligne 3 : on vérifie que la valeur saisie est <1 ou >7. Si c'est le cas, la validation est terminée. Si ce n'est pas le cas, le validateur doit signaler l'erreur en lançant une exception de type **ValidatorException**.

La classe *ValidatorException* a deux constructeurs :



- le constructeur [1] a pour paramètre un message d'erreur de type *FacesMessage*. Ce type de message est celui affiché par les balises <*h:messages*> et <*h:message*>,
- le constructeur [2] permet de plus d'encapsuler la cause de type *Throwable* ou dérivé de l'erreur.

Il nous faut construire un message de type *FacesMessage*. Cette classe a divers constructeurs :



Constructor Summary

`FacesMessage()`
Construct a new `FacesMessage` with no initial values.

`FacesMessage(FacesMessage.Severity severity, java.lang.String summary, java.lang.String detail)`
Construct a new `FacesMessage` with the specified initial values. 1

`FacesMessage(java.lang.String summary)`
Construct a new `FacesMessage` with just a summary.

`FacesMessage(java.lang.String summary, java.lang.String detail)`
Construct a new `FacesMessage` with the specified initial values.

Le constructeur [1] définit les propriétés d'un objet *FacesMessage* :

- *FacesMessage.Severity severity* : un niveau de gravité pris dans l'énumération suivante : SEVERITY_ERROR, SEVERITY_FATAL, SEVERITY_INFO, SEVERITY_WARN,
- *String summary* : la version résumée du message d'erreur - est affichée par les balises <*h:message showSummary="true"*> et <*h:messages*>,
- *String detail* : la version détaillée du message d'erreur - est affichée par les balises <*h:message*> et <*h:messages showDetail="true"*>.

N'importe lequel des constructeurs peut-être utilisé, les paramètres manquants pouvant être fixés ultérieurement par des méthodes *set*.

Le constructeur [1] ne permet pas de désigner un message qui serait dans un fichier de messages internationalisé. C'est évidemment dommage. **David Geary** et **Cay Horstmann [ref2]** comblent cette lacune dans leur livre "Core JavaServer Faces" avec la classe utilitaire *com.corejsf.util.Messages*. C'est cette classe qui est utilisée ligne 4 du code Java pour créer le message d'erreur. Elle ne contient que des méthodes statiques dont la méthode *getMessage* utilisée ligne 4 :

```
public static FacesMessage getMessage(String bundleName, String resourceId, Object[] params)
```

La méthode *getMessage* admet trois paramètres :

- *String bundleName* : le nom d'un fichier de messages sans son suffixe *.properties* mais avec son nom de paquetage. Ici, notre premier paramètre pourrait être *messages* pour désigner le fichier [messages.properties]. Avant d'utiliser le fichier désigné par le premier paramètre, *getMessage* essaie d'utiliser le fichier des messages de l'application, s'il y en a un. Ainsi si dans [faces-config.xml] on a déclaré un fichier de messages avec la balise :

```
1.    <application>
2.  ...
3.       <message-bundle>messages</message-bundle>
4.  </application>
```

  on peut passer *null* comme premier paramètre à la méthode *getMessage*. C'est ce qui a été fait ici (cf [web.xm], page 110),

- *String resourceId* : la clé du message à exploiter dans le fichier des messages. Nous avons vu qu'un message pouvait avoir à la fois une version résumée et une version détaillée. *resourceId* est l'identifiant de la version résumée. La version détaillée sera recherchée automatiquement avec la clé *resourceId_detail*. Ainsi, aurons-nous deux messages dans [messages.properties] pour l'erreur sur la saisie n° 10 :

```
saisie10.incorrecte=10-Saisie n° 10 incorrecte
saisie10.incorrecte_detail=10-Vous devez entrer un nombre entier <1 ou >7
```

  Le message de type *FacesMessage* produit par la méthode *Messages.getMessage* inclut à la fois les versions résumée et détaillée si elles ont été trouvées. **Les deux versions doivent être présentes** sinon on récupère une exception de type [NullPointerException],

- *Object[] params* : les paramètres effectifs du message si celui-ci a des paramètres formels {0}, {1}, ... Ces paramètres formels seront remplacés par les éléments du tableau *params*.

Revenons au code de la méthode de validation du composant *saisie10* :

```
1.    public void validateSaisie10(FacesContext context, UIComponent component, Object value) {
2.      int saisie = (Integer) value;
3.      if (!(saisie < 1 || saisie > 7)) {
4.        FacesMessage message = Messages.getMessage(null, "saisie10.incorrecte", null);
5.        message.setSeverity(FacesMessage.SEVERITY_ERROR);
6.        throw new ValidatorException(message);
7.      }
8.  }
```

- en [4], le message de type *FacesMessage* est créé à l'aide de la méthode statique *Messages.getMessage,*
- en [5], on fixe le niveau de gravité du message,
- en [6], on lance une exception de type *ValidatorException* avec le message construit précédemment. La méthode de validation a été appelée par le code XHTML suivant :

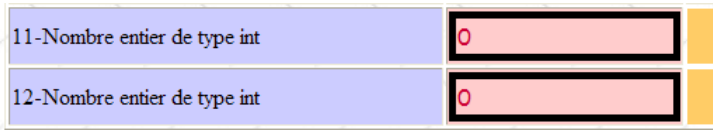
```
1.  <!-- ligne 11 -->
2.          <h:outputText value="#{msg['saisie10.prompt']}"/>
3.          <h:inputText id="saisie10" value="#{form.saisie10}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" validator="#{form.validateSaisie10}"/>
4.          <h:message for="saisie10" styleClass="error"/>
5.  <h:outputText value="#{form.saisie10}"/>
```

Ligne 3, la méthode de validation est exécutée pour le composant d'id *saisie10*. Aussi le message d'erreur produit par la méthode *validateSaisie10* est-il associé à ce composant et donc affiché par la ligne 4 (attribut *for="saisie10"*). C'est la version détaillée qui est affichée par défaut par la balise <*h:message*>.

Voici un exemple d'exécution :

2.8.5.8 Saisies 11 et 12 : validation d'un groupe de composants

Jusqu'à maintenant, les méthodes de validation rencontrées ne validaient qu'un unique composant. Comment faire si la validation souhaitée concerne plusieurs composants ? C'est ce que nous voyons maintenant. Dans le formulaire :



nous voulons que les saisies 11 et 12 soient deux nombres entiers dont la somme soit égale à 10.

Le code JSF sera le suivant :

```
1.  <!-- ligne 12 -->
2.          <h:outputText value="#{msg['saisie11.prompt']}"/>
3.          <h:inputText id="saisie11" value="#{form.saisie11}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"/>
4.          <h:panelGroup>
5.            <h:message for="saisie11" styleClass="error"/>
6.            <h:outputText value="#{form.errorSaisie11}" styleClass="error"/>
7.          </h:panelGroup>
8.          <h:outputText value="#{form.saisie11}"/>
9.          <!-- ligne 13 -->
10.         <h:outputText value="#{msg['saisie12.prompt']}"/>
11.         <h:inputText id="saisie12" value="#{form.saisie12}" styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"/>
12.         <h:panelGroup>
13.           <h:message for="saisie12" styleClass="error"/>
14.           <h:outputText value="#{form.errorSaisie12}" styleClass="error"/>
15.         </h:panelGroup>
16.         <h:outputText value="#{form.saisie12}"/>
```

et le modèle associé :

```
1.    private Integer saisie11 = 0;
2.    private Integer saisie12 = 0;
3.    private String errorSaisie11 = "";
4.  private String errorSaisie12 = "";
```

Ligne 3 du code JSF, on utilise les techniques déjà présentées pour vérifier que la valeur saisie pour le composant *saisie11* est bien un entier. Il en est de même, ligne 11, pour le composant *saisie12*. Pour vérifier que *saisie11* + *saisie12* =10, on pourrait construire un validateur spécifique. C'est la solution à préférer. De nouveau on lira **[ref2]** pour la découvrir. Nous suivons ici une autre démarche.

La page [index.xhtml] est validée par un bouton [Valider] dont le code JSF est le suivant :

```
1.  <!-- boutons de commande -->
2.        <h:panelGrid columns="2">
3.          <h:commandButton value="#{msg['submit']}" action="#{form.submit}"/>
4.          ...
5.        </h:panelGrid>
```

où le message *msg['submit']* est le suivant :

```
submit=Valider
```

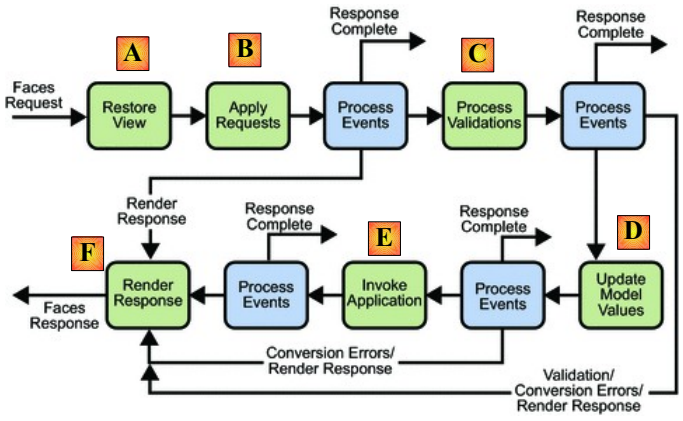
On voit ligne 3, que la méthode *form.submit* va être exécutée pour traiter le clic sur le bouton [Valider]. Celle-ci est la suivante :

```
1.    // actions
2.    public String submit() {
3.      // dernières validations
4.      validateForm();
5.      // on renvoie le même formulaire
6.      return null;
7.    }
8.
9.    // validations globales
10.   private void validateForm() {
11.     if ((saisie11 + saisie12) != 10) {
12. ...
13. }
```

Il est important de comprendre que lorsque la méthode **submit** s'exécute :

- tous les validateurs et convertisseurs du formulaire ont été exécutés et réussis,
- les champs du modèle [Form.java] ont reçu les valeurs postées par le client.

En effet, revenons au cycle de traitement d'un POST JSF :



La méthode *submit* est un gestionnaire d'événement. Elle gère l'événement *clic* sur le bouton [Valider]. Comme tous les gestionnaires d'événement, elle s'exécute dans la phase [E], une fois que tous les validateurs et convertisseurs ont été exécutés et réussis [C] et que le modèle a été mis à jour avec les valeurs postées [D]. Il n'est donc plus question ici de lancer des exceptions de type [ValidatorException] comme nous l'avons fait précédemment. Nous nous contenterons de renvoyer le formulaire avec des messages d'erreur :

En [1], nous alerterons l'utilisateur et en [2] et [3], nous mettrons un signe d'erreur. Dans le code JSF, le message [1] sera obtenu de la façon suivante :

```
1.  <h:form id="formulaire">
2.        <h:messages globalOnly="true" />
3.        <h:panelGrid columns="4" columnClasses="col1,col2,col3,col4" border="1">
4.          <!-- ligne 1 -->
5.          ...
```

En ligne 2, la balise **<h:messages>** affiche par défaut la version résumée des messages d'erreurs de toutes les saisies erronées de composants du formulaire ainsi que tous les messages d'erreur non liés à des composants. L'attribut *globalOnly="true"* limite l'affichage à ces derniers.

Les messages [2] et [3] sont affichés avec de simples balises **<h:outputText>** :

```
1.  <!-- ligne 12 -->
2.          <h:outputText value="#{msg['saisie11.prompt']}"/>
3.          <h:inputText id="saisie11" value="#{form.saisie11}"  styleClass="saisie" required="true"
    requiredMessage="#{msg['data.required']}" converterMessage="#{msg['integer.required']}"/>
4.          <h:panelGroup>
5.            <h:message for="saisie11" styleClass="error"/>
6.            <h:outputText value="#{form.errorSaisie11}" styleClass="error"/>
7.          </h:panelGroup>
8.          <h:outputText value="#{form.saisie11}"/>
9.          <!-- ligne 13 -->
10.         ...
11.           <h:outputText value="#{form.errorSaisie12}" styleClass="error"/>
12.         ...
```

Lignes 4-7, le composant *saisie11* a deux messages d'erreur possibles :

- celui indiquant une conversion erronée ou une absence de donnée. Ce message généré par JSF lui-même sera contenu dans un type *FacesMessage* et affiché par la balise **<h:message>** de la ligne 5,
- celui que nous allons générer si *saisie11* + *saisie12* n'est pas égal à 10. Il sera affiché par la ligne 6. Le message d'erreur sera contenu dans le modèle *form.errorSaisie11*.

Les deux messages correspondent à des erreurs qui ne peuvent se produire en même temps. La vérification *saisie11* + *saisie12* = *10* est faite dans la méthode *submit* qui ne s'exécute que s'il ne reste aucune erreur dans le formulaire. Lorsqu'elle s'exécutera, le composant *saisie11* aura été vérifié et son modèle *form.saisie11* aura reçu sa valeur. Le message de la ligne 5 ne pourra plus être affiché. Inversement, si le message de la ligne 5 est affiché, il reste alors au moins une erreur dans le formulaire et la méthode *submit* ne s'exécutera pas. Le message de la ligne 6 ne sera pas affiché. Afin que les deux messages d'erreur possibles soient dans la même colonne du tableau, ils ont été rassemblés dans une balise **<h:panelGroup>** (lignes 4 et 7).

La méthode *submit* est la suivante :

```
1.    // actions
2.    public String submit() {
3.      // dernières validations
4.      validateForm();
5.      // on renvoie le même formulaire
6.      return null;
7.    }
8.
9.    // validations globales
10.   private void validateForm() {
11.     if ((saisie11 + saisie12) != 10) {
12.       // msg global
```

```
13.       FacesMessage message = Messages.getMessage(null, "saisies11et12.incorrectes", null);
14.       message.setSeverity(FacesMessage.SEVERITY_ERROR);
15.       FacesContext context = FacesContext.getCurrentInstance();
16.       context.addMessage(null, message);
17.       // msg liés aux champs
18.       message = Messages.getMessage(null, "error.sign", null);
19.       setErrorSaisie11(message.getSummary());
20.       setErrorSaisie12(message.getSummary());
21.     } else {
22.       setErrorSaisie11("");
23.       setErrorSaisie12("");
24.     }
25. }
```

- ligne 4 : la méthode *submit* appelle la méthode *validateForm* pour faire les dernières validations,
- ligne 11 : on vérifie si *saisie11+saisie12*=10,
- si ce n'est pas le cas, lignes 13-14, on crée un message de type *FacesMessage* avec le message d'id *saisies11et12.incorrectes*. Celui-ci est le suivant :

```
saisies11et12.incorrectes=La propriété saisie11+saisie12=10 n'est pas vérifiée
```

- le message ainsi construit est ajouté (lignes 15-16) à la liste des messages d'erreur de l'application. Ce message n'est pas lié à un composant particulier. C'est un message global de l'application. Il sera affiché par la balise <*h:messages globalOnly="true"*/> présentée plus haut,
- ligne 18 : on crée un nouveau message de type *FacesMessage* avec le message d'id *error.sign*. Celui-ci est le suivant :

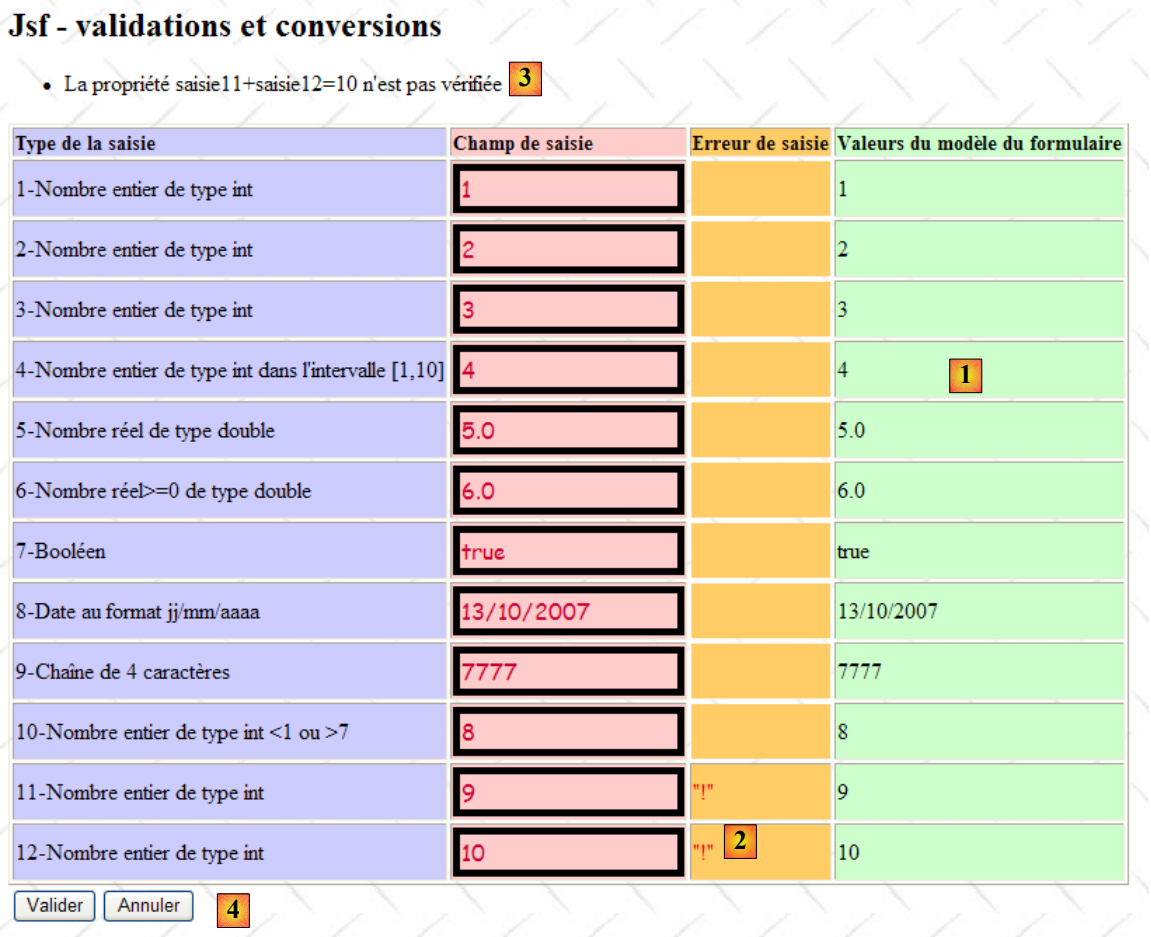
```
error.sign="!"
```

Nous avons dit que la méthode statique [Messages.getMessage] construisait un message de type *FacesMessage* avec une version résumée et une version détaillée si elles existaient. Ici, seule la version résumée du message *error.sign* existe. On obtient la version résumée d'un message *m*, par *m.getSummary()*. Lignes 19 et 20, la version résumée du message *error.sign* est mise dans les champs *errorSaisie11* et *errorSaisie12* du modèle. Ils seront affichés par les balises JSF suivantes :

```
1.            <h:outputText value="#{form.saisie11}"/>
2.            ...
3.            <h:outputText value="#{form.saisie12}"/>
```

- lignes 22-23 : si la propriété *saisie11+saisie12=10* est vérifiée, alors les deux champs *errorSaisie11* et *errorSaisie12* du modèle sont vidés afin qu'un éventuel message d'erreur précédent soit effacé. Il faut se rappeler ici que le modèle est conservé entre les requêtes, dans la session du client.

Voici un exemple d'exécution :



On remarquera dans la colonne [1] que le modèle a reçu les valeurs postées, ce qui montre que toutes les opérations de validation et de conversion entre les valeurs postées et le modèle ont réussi. Le gestionnaire d'événement *form.submit* qui gère le clic sur le bouton [Valider] a pu ainsi s'exécuter. C'est lui qui a produit les messages affichés en [2] et [3]. On voit que le modèle a été mis à jour alors même que le formulaire a été refusé et renvoyé au client. On pourrait vouloir que le modèle ne soit pas à mis à jour dans un tel cas. En effet, en imaginant que l'utilisateur annule sa mise à jour avec le bouton [Annuler] [4], on ne pourra pas revenir au modèle initial sauf à l'avoir mémorisé.

2.8.5.9 POST d'un formulaire sans vérification des saisies

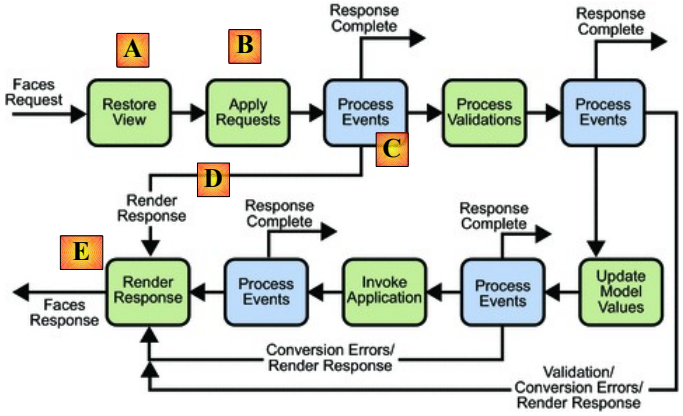
Considérons le formulaire ci-dessus et supposons que l'utilisateur ne comprenant pas ses erreurs veuille abandonner la saisie du formulaire. Il va alors utiliser le bouton [Annuler] généré par le code JSF suivant :

```
1.  <!-- boutons de commande -->
2.        <h:panelGrid columns="2">
3.          <h:commandButton value="#{msg['submit']}" action="#{form.submit}"/>
4.          <h:commandButton value="#{msg['cancel']}" immediate="true" action="#{form.cancel}"/>
5.        </h:panelGrid>
```

Ligne 4, le message *msg['cancel']* est le suivant :

```
cancel=Annuler
```

La méthode *form.cancel* associée au bouton [Annuler] ne sera exécutée que si le formulaire est valide. C'est ce que nous avons montré pour la méthode *form.submit* associée au bouton [Valider]. Si l'utilisateur veut annuler la saisie du formulaire, il est bien sûr inutile de vérifier la validité de ses saisies. Ce résultat est obtenu avec l'attribut *immediate="true"* qui indique à JSF d'exécuter la méthode *form.cancel* sans passer par la phase de validation et de conversion. Revenons au cycle de traitement du POST JSF :

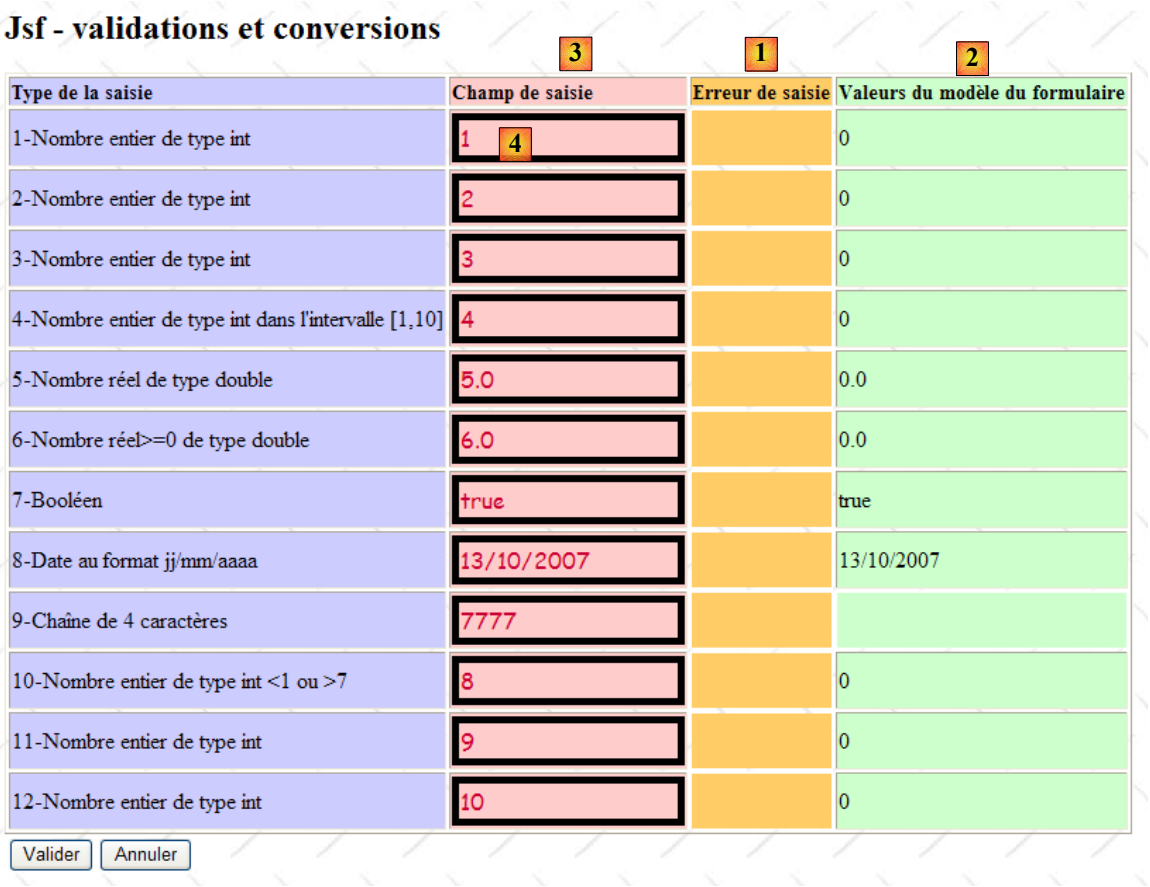


Les événements des composants d'action <**h:commandButton**> et <**h:commandLink**> ayant l'attribut **immediate="true"** sont traités dans la phase [C] puis le cycle JSF passe directement à la phase [E] de rendu de la réponse.

La méthode *form.cancel* est la suivante :

```
1.    public String cancel() {
2.      saisie1 = 0;
3.      saisie2 = 0;
4.      saisie3 = 0;
5.      saisie4 = 0;
6.      saisie5 = 0.0;
7.      saisie6 = 0.0;
8.      saisie7 = true;
9.      saisie8 = new Date();
10.     saisie9 = "";
11.     saisie10 = 0;
12.     return null;
13. }
```

Si on utilise le bouton [Annuler] dans le formulaire précédent, on obtient en retour la page suivante :



- on obtient de nouveau le formulaire car le gestionnaire d'événement *form.cancel* rend la clé de navigation *null*. La page [index.xhtml] est donc renvoyée,
- le modèle [Form.java] a été modifié par la méthode *form.cancel*. Ceci est reflété par la colonne [2] qui affiche ce modèle,
- la colonne [3], elle, reflète la valeur postée pour les composants.

Revenons sur le code JSF du composant *saisie1* [4] ;

```
1.            <!-- ligne 1 -->
2.            <h:outputText value="#{msg['saisie1.prompt']}"/>
3.            <h:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
4.            <h:message for="saisie1" styleClass="error"/>
5.  <h:outputText value="#{form.saisie1}"/>
```

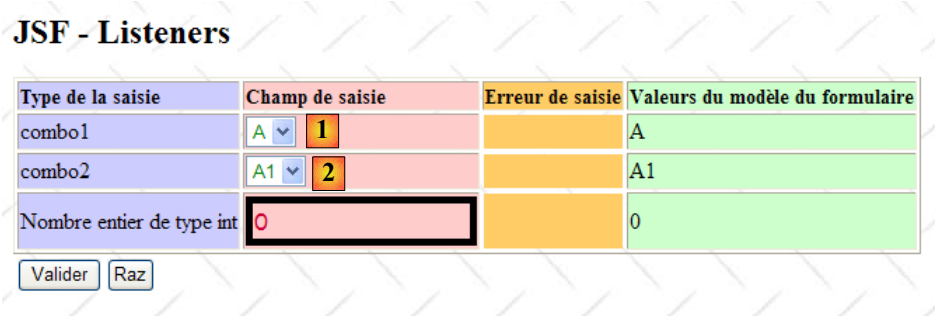
Ligne 4, la valeur du composant *saisie1* est liée au modèle *form.saisie1*. Cela entraîne plusieurs choses :

- lors d'un GET de [index.xhtml], le composant *saisie1* affichera la valeur du modèle *form.saisie1,*
- lors d'un POST de [index.xhtml], la valeur postée pour le composant *saisie1* n'est affectée au modèle *form.saisie1* que si l'ensemble des validations et conversions du formulaire réussissent. Que le modèle ait été mis à jour ou non par les valeurs postées, si le formulaire est renvoyé à l'issue du POST, les composants affichent la valeur qui a été postée et non pas la valeur du modèle qui leur est associé. C'est ce que montre la copie d'écran ci-dessus, où les colonnes [2] et [3] n'ont pas les mêmes valeurs.

2.9 Exemple mv-jsf2-07 : événements liés au changement d'état de composants JSF

2.9.1 L'application

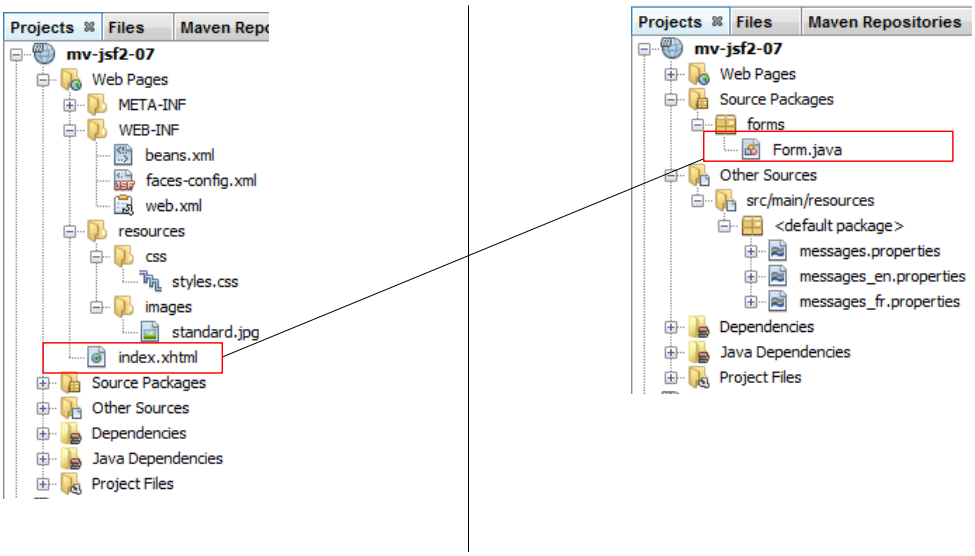
L'application montre un exemple de POST réalisé sans l'aide d'un bouton ou d'un lien. Le formulaire est le suivant :



Le contenu de la liste *combo2* [2] est liée à l'élément sélectionné dans le *combo1* [1]. Lorsqu'on change la sélection dans [1], un POST du formulaire est réalisé au cours duquel le contenu de *combo2* est modifié pour refléter l'élément sélectionné dans [1], puis le formulaire renvoyé. Au cours de ce POST, aucune validation n'est faite.

2.9.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



On a un unique formulaire [index.xhtml] avec son modèle [Form.java].

2.9.3 L'environnement de l'application

Le fichier des messages [**messages_fr.properties**] :

```
1.  app.titre=intro-07
2.  app.titre2=JSF - Listeners
3.  combo1.prompt=combo1
4.  combo2.prompt=combo2
5.  saisie1.prompt=Nombre entier de type int
6.  submit=Valider
7.  raz=Raz
8.  data.required=Donnée requise
9.  integer.required=Entrez un nombre entier
10. saisie.type=Type de la saisie
11. saisie.champ=Champ de saisie
12. saisie.erreur=Erreur de saisie
13. bean.valeur=Valeurs du modèle du formulaire
```

La feuille de style [**styles.css**] :

```
1.  .info{
2.     font-family: Arial,Helvetica,sans-serif;
```

```
    font-size: 14px;
    font-weight: bold
}

.col1{
    background-color: #ccccff
}

.col2{
    background-color: #ffcccc
}

.col3{
    background-color: #ffcc66
}

.col4{
    background-color: #ccffcc
}

.error{
    color: #ff0000
}

.saisie{
    background-color: #ffcccc;
    border-color: #000000;
    border-width: 5px;
    color: #cc0033;
    font-family: cursive;
    font-size: 16px
}

.combo{
  color: green;
}

.entete{
    font-family: 'Times New Roman',Times,serif;
    font-size: 14px;
    font-weight: bold
}
```

2.9.4 Le formulaire [index.xhtml]

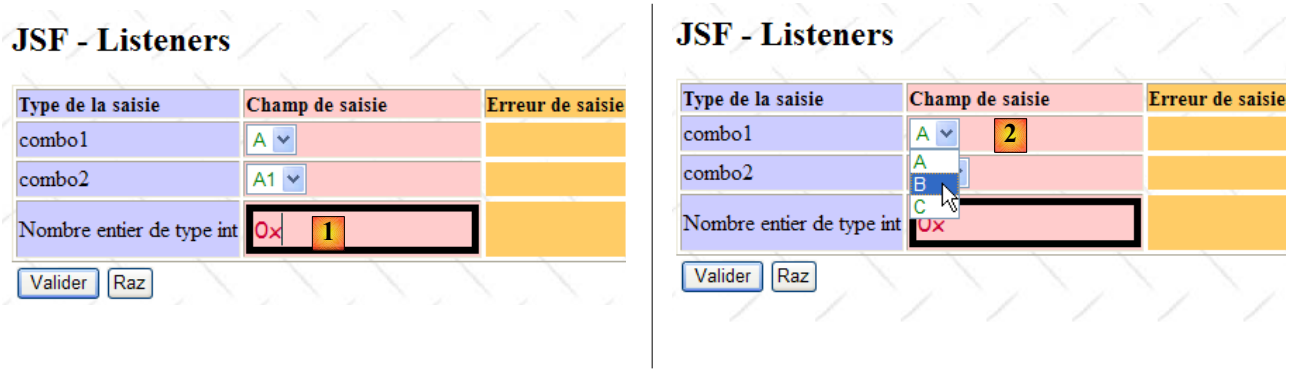
Le formulaire [index.xhtml] est le suivant :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>JSF</title>
    <h:outputStylesheet library="css" name="styles.css"/>
    ...
  </h:head>
  <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
    <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
    <h:form id="formulaire">
      <h:messages globalOnly="true"/>
      <h:panelGrid columns="4" border="1" columnClasses="col1,col2,col3,col4">
        <!-- headers -->
        <h:outputText value="#{msg['saisie.type']}" styleClass="entete"/>
        <h:outputText value="#{msg['saisie.champ']}" styleClass="entete"/>
        <h:outputText value="#{msg['saisie.erreur']}" styleClass="entete"/>
        <h:outputText value="#{msg['bean.valeur']}" styleClass="entete"/>
        <!-- ligne 1 -->
        <h:outputText value="#{msg['combo1.prompt']}"/>
        <h:selectOneMenu id="combo1" value="#{form.combo1}" immediate="true" onchange="submit();" valueChangeListener="#{form.combo1ChangeListener}" styleClass="combo">
```

```
25.           <f:selectItems value="#{form.combo1Items}"/>
26.         </h:selectOneMenu>
27.         <h:panelGroup></h:panelGroup>
28.         <h:outputText value="#{form.combo1}"/>
29.         <!-- ligne 2 -->
30.         <h:outputText value="#{msg['combo2.prompt']}"/>
31.         <h:selectOneMenu id="combo2" value="#{form.combo2}" styleClass="combo">
32.           <f:selectItems value="#{form.combo2Items}"/>
33.         </h:selectOneMenu>
34.         <h:panelGroup></h:panelGroup>
35.         <h:outputText value="#{form.combo2}"/>
36.         <!-- ligne 3 -->
37.         <h:outputText value="#{msg['saisie1.prompt']}"/>
38.         <h:inputText id="saisie1" value="#{form.saisie1}" required="true"
    requiredMessage="#{msg['data.required']}" styleClass="saisie"
    converterMessage="#{msg['integer.required']}"/>
39.         <h:message for="saisie1" styleClass="error"/>
40.         <h:outputText value="#{form.saisie1}"/>
41.       </h:panelGrid>
42.       <!-- boutons de commande -->
43.       <h:panelGrid columns="2" border="0">
44.         <h:commandButton value="#{msg['submit']}"/>
45.         ...
46.       </h:panelGrid>
47.     </h:form>
48.   </h:body>
49. </html>
```

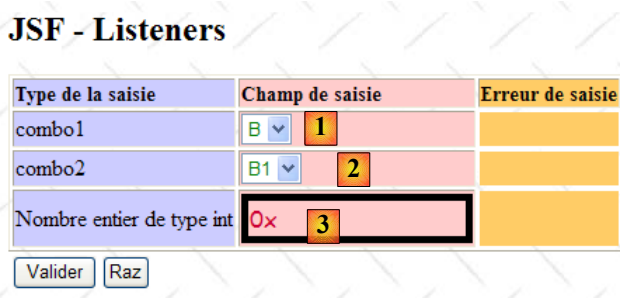
La nouveauté réside dans le code de la liste *combo1*, lignes 24-26. De nouveaux attributs apparaissent :

- **onchange** : attribut HTML - déclare une fonction ou du code Javascript qui doit être exécuté lorsque l'élément sélectionné dans *combo1* change. Ici, le code Javascript *submit()* poste le formulaire au serveur,
- **valueChangeListener** : attribut JSF - déclare le nom de la méthode à exécuter côté serveur lorsque l'élément sélectionné dans *combo1* change. Au total, il y a deux méthodes exécutées : l'une côté **client**, l'autre côté **serveur**,
- **immediate=true** : attribut JSF - fixe le moment où doit être exécuté le gestionnaire d'événement côté serveur : après que le formulaire ait été reconstitué comme l'utilisateur l'a saisi mais avant les contrôles de validité des saisies. On veut ici, remplir la liste *combo2* en fonction de l'élément sélectionné dans la liste *combo1*, même si par ailleurs dans le formulaire il peut y avoir des saisies erronées. Voici un exemple :



- en [1], une première saisie,
- en [2], on passe l'élément sélectionné de *combo1* de A à B.

Le résultat obtenu est le suivant :



Le POST a eu lieu. Le contenu de *combo2* [2] a été adapté à l'élément sélectionné dans *combo1* [1] bien que la saisie [3] était incorrecte. C'est l'attribut *immediate=true* qui a fait que la méthode *form.combo1ChangeListener* a été exécutée avant les contrôles de validité. Sans cet attribut, elle n'aurait pas été exécutée car le cycle de traitement se serait arrêté aux contrôles de validité à cause de l'erreur en [3].

Les messages associés au formulaire sont les suivants dans [messages.properties] :

```
1.  app.titre=intro-07
2.  app.titre2=JSF - Listeners
3.  combo1.prompt=combo1
4.  combo2.prompt=combo2
5.  saisie1.prompt=Nombre entier de type int
6.  submit=Valider
7.  raz=Raz
8.  data.required=Donnée requise
9.  integer.required=Entrez un nombre entier
10. saisie.type=Type de la saisie
11. saisie.champ=Champ de saisie
12. saisie.erreur=Erreur de saisie
13. bean.valeur=Valeurs du modèle du formulaire
```

La durée de vie de [Form.java] est fixée à **session** :

```
1.  package forms;
2.
3.  ...
4.
5.  @ManagedBean
6.  @SessionScoped
7.  public class Form {
```

Ligne 6, on fixe la portée du bean à **session**. La première version du bean avait la portée **request**. Des erreurs incompréhensibles apparaissaient alors. Le changement de portée les a fait disparaître. Je n'ai pas trouvé d'explications à ces erreurs.

2.9.5 Le modèle [Form.java]

Le modèle [Form.java] est le suivant :

```
1.  package forms;
2.
3.  import java.util.logging.Logger;
4.  import javax.faces.bean.SessionScoped;
5.  import javax.faces.bean.ManagedBean;
6.  import javax.faces.context.FacesContext;
7.  import javax.faces.event.ValueChangeEvent;
8.  import javax.faces.model.SelectItem;
9.
10. @ManagedBean
11. @SessionScoped
12. public class Form {
13.
14.   public Form() {
15.   }
16.
17. // champs du formulaire
18.   private String combo1="A";
```

```
19.   private String combo2="A1";
20.   private Integer saisie1=0;
21.
22.   // champs de travail
23.   final private String[] combo1Labels={"A","B","C"};
24.   private String combo1Label="A";
25.   private static final Logger logger=Logger.getLogger("forms.Form");
26.
27.   // méthodes
28.   public SelectItem[] getCombo1Items(){
29.     // init combo1
30.     SelectItem[] combo1Items=new SelectItem[combo1Labels.length];
31.     for(int i=0;i<combo1Labels.length;i++){
32.       combo1Items[i]=new SelectItem(combo1Labels[i],combo1Labels[i]);
33.     }
34.     return combo1Items;
35.   }
36.
37.   public SelectItem[] getCombo2Items(){
38.     // init combo2 en fonction de combo1
39.     SelectItem[] combo2Items=new SelectItem[5];
40.     for(int i=1;i<=combo2Items.length;i++){
41.       combo2Items[i-1]=new SelectItem(combo1Label+i,combo1Label+i);
42.     }
43.     return combo2Items;
44.   }
45.
46.   // listeners
47.   public void combo1ChangeListener(ValueChangeEvent event){
48.     // suivi
49.     logger.info("combo1ChangeListener");
50.     // on récupère la valeur postée de combo1
51.     combo1Label=(String)event.getNewValue();
52.     // on rend la réponse car on veut court-circuiter les validations
53.     FacesContext.getCurrentInstance().renderResponse();
54.   }
55.
56.   public String raz(){
57.     // suivi
58.     logger.info("raz");
59.     // raz du formulaire
60.     combo1Label="A";
61.     combo1="A";
62.     combo2="A1";
63.     saisie1=0;
64.     return null;
65.   }
66.
67. // getters - setters
68.   ...
69. }
```

Relions le formulaire [index.xhtml] à son modèle [Form.java] :

La liste *combo1* est générée par le code JSF suivant :

```
1.          <h:selectOneMenu id="combo1" value="#{form.combo1}" immediate="true" onchange="submit();"
    valueChangeListener="#{form.combo1ChangeListener}" styleClass="combo">
2.            <f:selectItems value="#{form.combo1Items}"/>
3.  </h:selectOneMenu>
```

Elle obtient ses éléments par la méthode *getCombo1Items* de son modèle (ligne 2). Celle-ci est définie lignes 28-35 du code Java. Elle génère une liste de trois éléments {"A","B","C"}.

La liste *combo2* est générée par le code JSF suivant :

```
1.          <h:selectOneMenu id="combo2" value="#{form.combo2}" styleClass="combo">
2.            <f:selectItems value="#{form.combo2Items}"/>
3.  </h:selectOneMenu>
```

Elle obtient ses éléments par la méthode *getCombo2Items* de son modèle (ligne 2). Celle-ci est définie lignes 37-44 du code Java. Elle génère une liste de cinq éléments {"X1","X2","X3","X4","X5"} où X est l'élément *combo1Label* de la ligne 16. Donc lors de la génération initiale du formulaire, la liste *combo2* contient les éléments {"A1","A2","A3","A4","A5"}.

Lorsque l'utilisateur va changer l'élément sélectionné dans la liste *combo1*,

- l'événement *onchange="submit();"* va être traité par le navigateur client. Le formulaire va donc être posté au serveur,
- côté serveur, JSF va détecter que le composant *combo1* a changé de valeur. La méthode *combo1ChangeListener* des lignes 47-54 va être exécutée. Une méthode de type *ValueChangeListener* reçoit en paramètre un objet de type *javax.faces.event.ValueChangeEvent*. Cet objet permet d'obtenir l'ancienne et la nouvelle valeur du composant qui a changé de valeur avec les méthodes suivantes :

java.lang.Object	getNewValue() Return the current local value of the source UIComponent.
java.lang.Object	getOldValue() Return the previous local value of the source UIComponent.

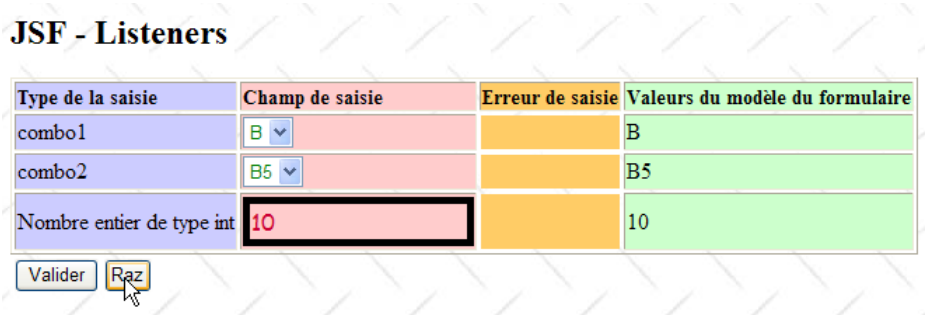
Ici le composant est la liste *combo1* de type *UISelectOne*. Sa valeur est de type *String*.

- ligne 51 du modèle Java : la nouvelle valeur de *combo1* est mémorisée dans *combo1Label* qui sert à générer les éléments de la liste *combo2*,
- ligne 53 : on rend la réponse. Il faut se rappeler ici que le gestionnaire *combo1ChangeListener* est exécuté avec l'attribut *immediate="true"*. Il est donc exécuté **après** la phase où l'arbre des composants de la page a été mis à jour avec les valeurs postées et **avant** le processus de validation des valeurs postées. Or on veut éviter ce processus de validation parce que la liste *combo2* doit être mise à jour même si par ailleurs dans le formulaire, il reste des saisies erronées. On demande donc à ce que la réponse soit envoyée tout de suite sans passer par la phase de validation des saisies.
- le formulaire va être renvoyé tel qu'il a été saisi. Cependant, les éléments des listes *combo1* et *combo2* ne sont pas des valeurs postées. Elles vont être générées de nouveau par appel aux méthodes *getCombo1Items* et *getCombo2Items*. Cette dernière méthode va alors utiliser la nouvelle valeur de *combo1Label* fixée par *combo1ChangeListener* et les éléments de la liste *combo2* vont changer.

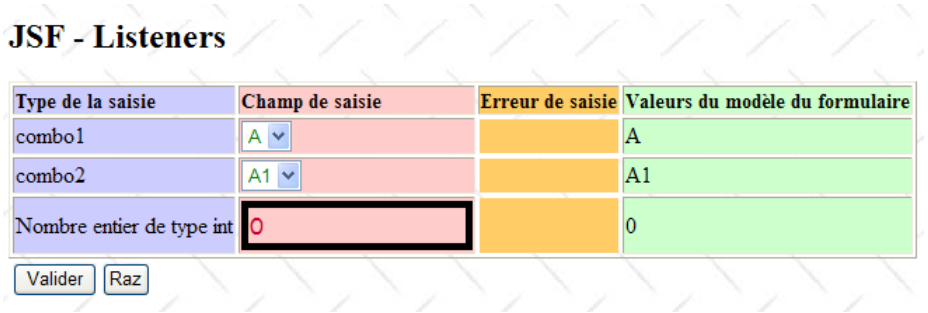
2.9.6 Le bouton [Raz]

Nous voulons avec le bouton [Raz] remettre le formulaire dans un état initial comme montré ci-dessous :

1



2



En [1], le formulaire avant le POST du bouton [Raz], en [2] le résultat du POST.

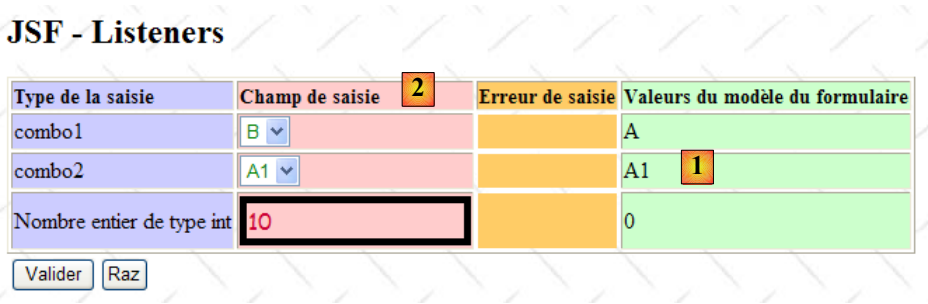
Bien que fonctionnellement simple, la gestion de ce cas d'utilisation s'avère assez complexe. On peut essayer diverses solutions, notamment celle utilisée pour le bouton [Annuler] de l'exemple précédent :

```
<h:commandButton value="#{msg['raz']}" immediate="true" action="#{form.raz}"/>
```

où la méthode *form.raz* est la suivante :

```
1.    public String raz(){
2.      // raz du formulaire
3.      combo1Label="A";
4.      combo1="A";
5.      combo2="A1";
6.      saisie1=0;
7.      return null;
8.  }
```

Le résultat obtenu par le bouton [Raz] dans l'exemple précédent est alors le suivant :



La colonne [1] montre que la méthode *form.raz* a été exécutée. Cependant la colonne [1] continue à afficher les valeurs postées :

- pour *combo1*, la valeur postée était "B". Cet élément est donc sélectionné dans la liste,
- pour *combo2*, la valeur postée était "B5". Du fait de l'exécution de *form.raz*, les éléments {"B1", ..., "B5"} de *combo2* ont été changés en {"A1", ..., "A5"}. L'élément "B5" n'existe plus et ne peut donc être sélectionné. C'est alors le premier élément de la liste qui est affiché,
- pour *saisie1*, la valeur postée était 10.

Ceci est le fonctionnement normal avec l'attribut **immediate="true"**. Pour avoir un résultat différent, il faut poster les valeurs qu'on veut voir dans le nouveau formulaire même si l'utilisateur a lui saisi d'autres valeurs. On réalise cela avec un peu de code Javascript côté client. Le formulaire devient le suivant :

```
1.  <script language="javascript">
2.    function raz(){
3.      document.forms['formulaire'].elements['formulaire:combo1'].value="A";
4.      document.forms['formulaire'].elements['formulaire:combo2'].value="A1";
5.      document.forms['formulaire'].elements['formulaire:saisie1'].value=0;
6.      //document.forms['formulaire'].submit();
7.    }
8.  </script>
9.  ...
10. <h:commandButton value="#{msg['raz']}" onclick='raz()' immediate="true" action="#{form.raz}"/>
```

- ligne 10, l'attribut **onclick='raz()'** indique d'exécuter la fonction Javascript *raz* lorsque l'utilisateur clique sur le bouton [Raz],
- ligne 3 : on affecte la valeur "A" à l'élément HTML de nom '**formulaire:combo1**'. Les différents éléments de la ligne 3 sont les suivants :
  - **document** : page affichée par le navigateur,
  - **document.forms** : ensemble des formulaires du document,
  - **document.forms['formulaire']** : le formulaire avec l'attribut *name="formulaire"*,
  - **documents.forms['formulaire'].elements** : ensemble des éléments du formulaire ayant l'attribut *name="formulaire"*,
  - **document.forms['formulaire'].elements['formulaire:combo1']** : élément du formulaire ayant l'attribut *name="formulaire:combo1"*
  - **document.forms['formulaire'].elements['formulaire:combo1'].value** : valeur qui sera postée par l'élément du formulaire ayant l'attribut *name="formulaire:combo1"*.

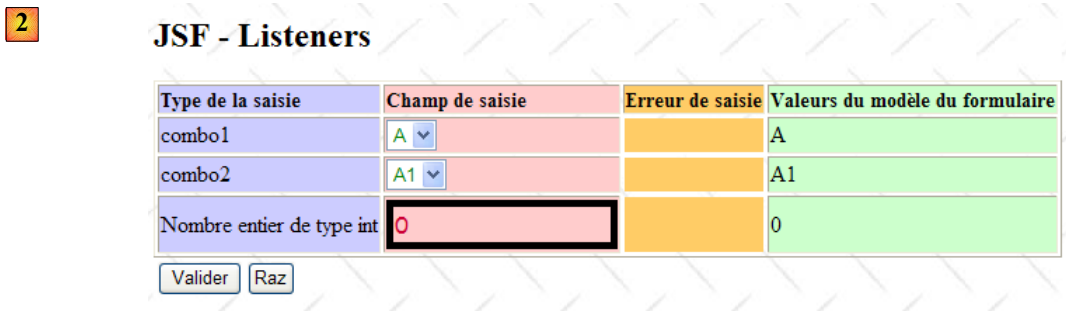
Pour connaître les attributs *name* des différents éléments de la page affichée par le navigateur, on pourra consulter son code source (ci-dessous avec IE7) :


```
<form id="formulaire" name="formulaire" ...>
...
<select id="formulaire:combo1" name="formulaire:combo1" ...>
```

Ceci expliqué, on peut comprendre que dans le code Javascript de la fonction *raz* :

- la ligne 3 fait que la valeur postée pour le composant *combo1* sera la chaîne **A,**
- la ligne 4 fait que la valeur postée pour le composant *combo2* sera la chaîne **A1,**
- la ligne 5 fait que la valeur postée pour le composant *saisie1* sera la chaîne **0.**

Ceci fait, le POST du formulaire, associé à tout bouton de type <*h:commandButton*> (ligne 10) va se produire. La méthode *form.raz* va être exécutée et le formulaire renvoyé tel qu'il a été posté. On obtient alors le résultat suivant :



Ce résultat cache beaucoup de choses. Les valeurs "A","A1","0" des composants *combo1*, *combo2*, *saisie1* sont postées au serveur. Supposons que la valeur précédente de *combo1* était "B". Alors il y a un changement de valeur du composant *combo1* et la méthode *form.combo1ChangeListener* devrait être exécutée elle aussi. On a deux gestionnaires d'événements ayant l'attribut *immediate="true"*. Vont-ils être exécutés tous les deux ? Si oui, dans quel ordre ? L'un seulement ? Si oui lequel ?

Pour en savoir plus, nous créons des logs dans l'application :

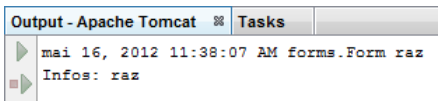
```
1.  package forms;
2.
3.  import java.util.logging.Logger;
4.  ...
5.  public class Form {
6.
7.  ...
8.  // champs du formulaire
9.    private String combo1="A";
10.   private String combo2="A1";
11.   private Integer saisie1=0;
12.
13.   // champs de travail
```

```
14.   final private String[] combo1Labels={"A","B","C"};
15.   private String combo1Label="A";
16.   private static final Logger logger=Logger.getLogger("forms.Form");
17.
18.   // listener
19.   public void combo1ChangeListener(ValueChangeEvent event){
20.     // suivi
21.     logger.info("combo1ChangeListener");
22.     // on récupère la valeur postée de combo1
23.     combo1Label=(String)event.getNewValue();
24.     // on rend la réponse car on veut court-circuiter les validations
25.     FacesContext.getCurrentInstance().renderResponse();
26.   }
27.
28.   public String raz(){
29.     // suivi
30.     logger.info("raz");
31.     // raz du formulaire
32.     combo1Label="A";
33.     combo1="A";
34.     combo2="A1";
35.     saisie1=0;
36.     return null;
37.   }
38. ...
39.}
```

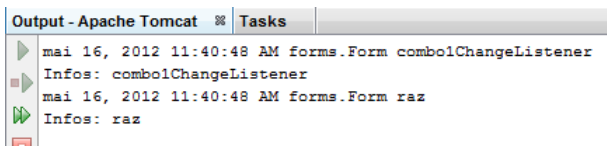
- ligne 16 : un générateur de logs est créé. Le paramètre de *getLogger* permet de différencier les origines des logs. Ici, le logueur s'appelle *forms.Form*,
- ligne 21 : on logue le passage dans la méthode *combo1ChangeListener*,
- ligne 30 : on logue le passage dans la méthode *raz*.

Quels sont les logs produits par le bouton [Raz] ou le changement de valeur de *combo1* ? Considérons divers cas :

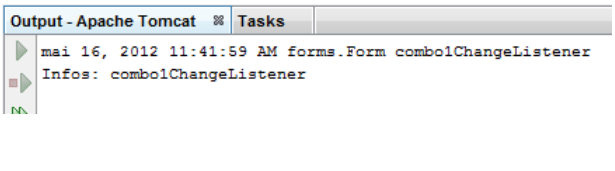
- on utilise le bouton [Raz] alors que l'élément sélectionné dans *combo1* est "A". "A" est donc la dernière valeur du composant *combo1*. Nous avons vu que le bouton [Raz] exécutait une fonction Javascript qui postait la valeur "A" pour le composant *combo1*. Le composant *combo1* ne change donc pas de valeur. Les logs montrent alors que seule la méthode *form.raz* est exécutée :



- on utilise le bouton [Raz] alors que l'élément sélectionné dans *combo1* n'est pas "A". Le composant *combo1* change donc de valeur : sa dernière valeur n'était pas "A" et le bouton [Raz] va lui poster la valeur "A". Les logs montrent alors que deux méthodes sont exécutées. Dans l'ordre : *combo1ChangeListener*, *raz* :



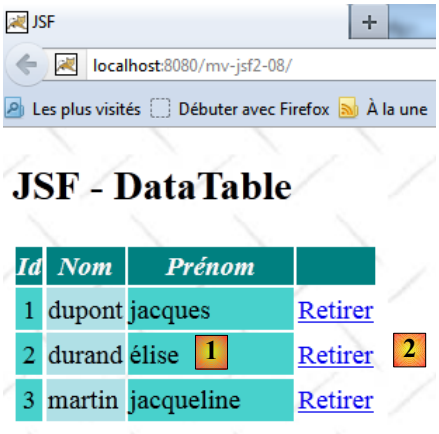
- on change la valeur du *combo1* sans utiliser le bouton [Raz]. Les logs montrent que seule la méthode *combo1ChangeListener* est exécutée :



2.10 Exemple mv-jsf2-08 : la balise <h:dataTable>

2.10.1 L'application

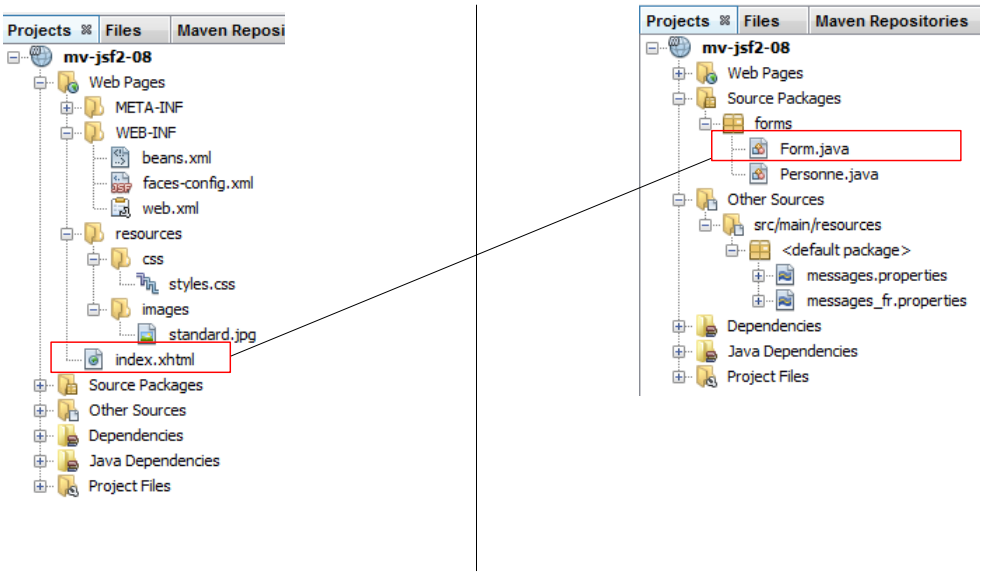
L'application montre une liste de personnes avec la possibilité d'en supprimer :



- en [1], une liste de personnes,
- en [2], les liens qui permettent de les supprimer.

2.10.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



On a un unique formulaire [index.xhtml] avec son modèle [Form.java].

2.10.3 L'environnement de l'application

Le fichier de configuration [**faces-config.xml**] :

```
<?xml version='1.0' encoding='UTF-8'?>

<!-- ========== FULL CONFIGURATION FILE ================================== -->

<faces-config version="2.0"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">

  <application>
    <resource-bundle>
      <base-name>
        messages
      </base-name>
      <var>msg</var>
    </resource-bundle>
    <message-bundle>messages</message-bundle>
  </application>
</faces-config>
```

Le fichier des messages [**messages_fr.properties**] :

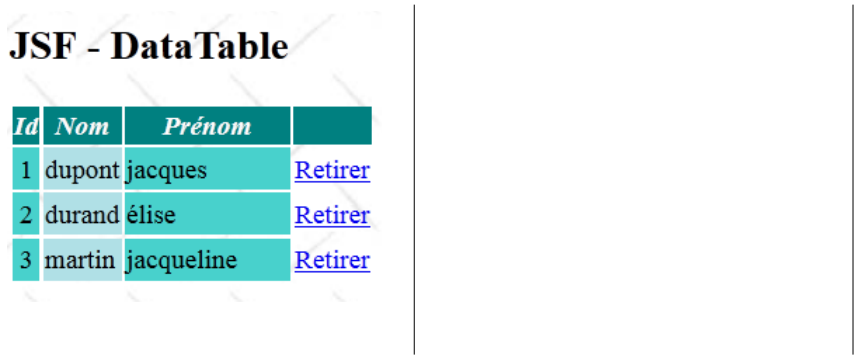
```
app.titre=intro-08
app.titre2=JSF - DataTable
submit=Valider
personnes.headers.id=Id
personnes.headers.nom=Nom
personnes.headers.prenom=Pr\u00e9nom
```

La feuille de style [**styles.css**] :

```
.headers {
   text-align: center;
   font-style: italic;
   color: Snow;
   background: Teal;
}

.id {
   height: 25px;
   text-align: center;
   background: MediumTurquoise;
}

.nom {
   text-align: left;
   background: PowderBlue;
}
.prenom {
   width: 6em;
   text-align: left;
   color: Black;
   background: MediumTurquoise;
}
```

2.10.4 Le formulaire [index.xhtml] et son modèle [Form.java]

Rappelons la vue associée à la page [index.xhtml] :



Le formulaire [index.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:f="http://java.sun.com/jsf/core">
7.    <h:head>
8.      <title>JSF</title>
9.      <h:outputStylesheet library="css" name="styles.css"/>
10.   </h:head>
11.   <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
12.     <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
13.     <h:form id="formulaire">
14.       <h:dataTable value="#{form.personnes}" var="personne" headerClass="headers"
    columnClasses="id,nom,prenom">
15.   ......................
16.       </h:dataTable>
17.     </h:form>
18.   </h:body>
19. </html>
```

Ligne 14, la balise <**h:dataTable**> utilise le champ **#{form.personnes}** comme source de données. C'est le suivant :

```
private List<Personne> personnes;
```

La classe [Personne] est la suivante :

```
1.  package forms;
2.
3.  public class Personne {
4.    // data
5.    private int id;
6.    private String nom;
7.    private String prénom;
8.
9.    // constructeurs
10.   public Personne(){
11.
12.   }
13.
14.   public Personne(int id, String nom, String prénom){
15.     this.id=id;
16.     this.nom=nom;
17.     this.prénom=prénom;
18.   }
19.
20.   // toString
21.   public String toString(){
22.     return String.format("Personne[%d,%s,%s]", id,nom,prénom);
23.   }
24.
25.   // getter et setters
26. ...
```

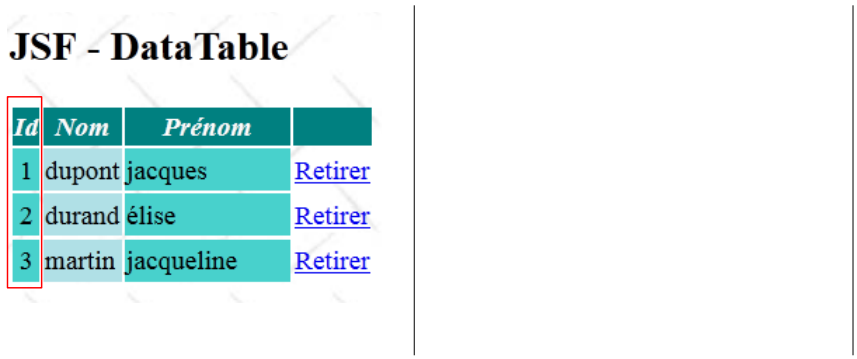
```
27. }
```

Revenons au contenu de la balise <**h:dataTable**> :

```
1. <h:dataTable value="#{form.personnes}" var="personne" headerClass="headers"
   columnClasses="id,nom,prenom">
2. ...
3. </h:dataTable>
```

- l'attribut **var="personne"** fixe le nom de la variable représentant la personne courante à l'intérieur de la balise <**h:datatable**>,
- l'attribut **headerClass="headers"** fixe le style des titres des colonnes du tableau,
- l'attribut **columnClasses="...."** fixe le style de chacune des colonnes du tableau.

Examinons l'une des colonnes du tableau et voyons comment elle est construite :



Le code XHTML de la colonne *Id* est le suivant :

```
1. <h:dataTable value="#{form.personnes}" var="personne" headerClass="headers"
   columnClasses="id,nom,prenom">
2.         <h:column>
3.           <f:facet name="header">
4.             <h:outputText value="#{msg['personnes.headers.id']}"/>
5.           </f:facet>
6.           <h:outputText value="#{personne.id}"/>
7.         </h:column>
8.         ...
9.       </h:dataTable>
```

- lignes 3-5 : la balise <**f:facet name="header"**> définit le titre de la colonne,
- ligne 4 : le titre de la colonne est pris dans le fichier des messages,
- ligne 6 : **personne** fait référence à l'attribut **var** de la balise <**h:dataTable** ...> (ligne 1). On écrit donc l'id de la personne courante.

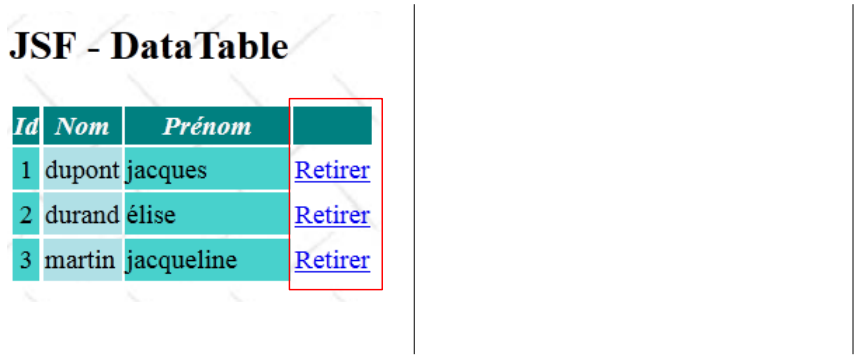
L'ensemble des colonnes est affiché par le code suivant :

```
1. <h:dataTable value="#{form.personnes}" var="personne" headerClass="headers"
   columnClasses="id,nom,prenom">
2.         <h:column>
3.           <f:facet name="header">
4.             <h:outputText value="#{msg['personnes.headers.id']}"/>
5.           </f:facet>
6.           <h:outputText value="#{personne.id}"/>
7.         </h:column>
8.         <h:column>
9.           <f:facet name="header">
10.            <h:outputText value="#{msg['personnes.headers.nom']}"/>
11.          </f:facet>
12.          <h:outputText value="#{personne.nom}"/>
13.        </h:column>
14.        <h:column>
15.          <f:facet name="header">
```

```
16.             <h:outputText value="#{msg['personnes.headers.prenom']}"/>
17.           </f:facet>
18.           <h:outputText value="#{personne.prénom}"/>
19.         </h:column>
20. ...
21.       </h:dataTable>
```

- lignes 3-7 : la colonne **id** du tableau,
- lignes 8-13 : la colonne **nom** du tableau,
- lignes 14-19 : la colonne **prénom** du tableau.

Maintenant, examinons la colonne des liens [Retirer] :



Cette colonne est générée par le code suivant :

```
1.  <h:dataTable value="#{form.personnes}" var="personne" headerClass="headers"
    columnClasses="id,nom,prenom">
2.  ...
3.          <h:column>
4.            <h:commandLink value="Retirer" action="#{form.retirerPersonne}">
5.              <f:setPropertyActionListener target="#{form.personneId}" value="#{personne.id}"/>
6.            </h:commandLink>
7.          </h:column>
8.        </h:dataTable>
```

Le lien [Retirer] est généré par les lignes 4-6. Lorsque le lien est cliqué, la méthode *[Form].retirerPersonne* va être exécutée. Il est temps d'examiner la classe [Form.java] :

```
1.  package forms;
2.
3.
4.  import java.util.ArrayList;
5.  import java.util.List;
6.  import javax.faces.bean.ManagedBean;
7.  import javax.faces.bean.SessionScoped;
8.
9.  @ManagedBean
10. @SessionScoped
11. public class Form {
12.
13.   // modèle
14.   private List<Personne> personnes;
15.   private int personneId;
16.
17.   // constructeur
18.   public Form() {
19.     // initialisation de la liste des personnes
20.     personnes = new ArrayList<Personne>();
21.     personnes.add(new Personne(1, "dupont", "jacques"));
22.     personnes.add(new Personne(2, "durand", "élise"));
23.     personnes.add(new Personne(3, "martin", "jacqueline"));
24.   }
25.
26.   public String retirerPersonne() {
27.     // on recherche la personne sélectionnée
```

```
28.     int i = 0;
29.     for (Personne personne : personnes) {
30.       // personne courante = personne sélectionnée ?
31.       if (personne.getId() == personneId) {
32.         // on supprime la personne courante de la liste
33.         personnes.remove(i);
34.         // on a fini
35.         break;
36.       } else {
37.         // personne suivante
38.         i++;
39.       }
40.     }
41.     // on teste sur la même page
42.     return null;
43.   }
44.
45.   // getters et setters
46. ...
47. }
```

- lignes 18-24 : le contructeur initialise la liste des personnes de la ligne 14,
- ligne 10 : parce que cette liste doit vivre au fil des requêtes, la portée du bean est la session.

Lorsque la méthode [retirerPersonne] de la ligne 26 est exécutée, le champ de la ligne 15 a été initialisé par l'id de la personne dont on a cliqué le lien [Retirer] :

```
1.             <h:commandLink value="Retirer" action="#{form.retirerPersonne}">
2.               <f:setPropertyActionListener target="#{form.personneId}" value="#{personne.id}"/>
3.   </h:commandLink>
```

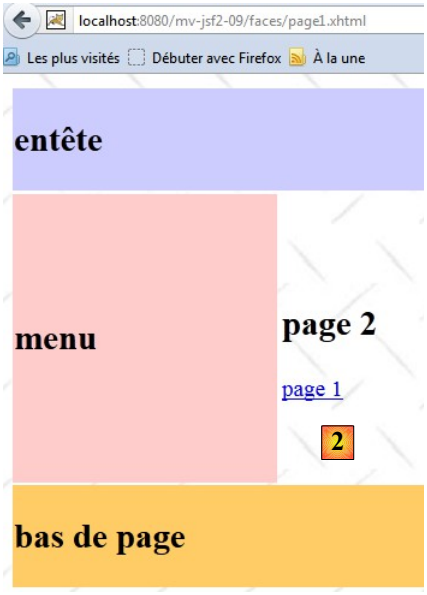
La balise `<f:setPropertyActionListener>` permet de transférer des informations au modèle. Ici la valeur de l'attribut **value** est copiée dans le champ du modèle identifié par l'attribut **target**. Ainsi l'id de la personne courante, celle qu'on doit supprimer de la liste des personnes, est copiée dans le champ **[Form].personneId** via le getter de ce champ. Ceci est fait avant l'exécution de la méthode référencée par l'attribut **action** de la ligne 1.

Lignes 26-43, la méthode [supprimerPersonne] supprime la personne dont l'*id* est égal à *personneId*.

2.11 Exemple mv-jsf2-09 : mise en page d'une application JSF

2.11.1 L'application

L'application montre comment mettre en page d'une application JSF à deux vues :



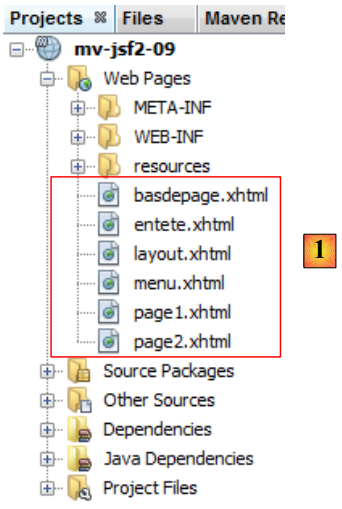
L'application a deux vues :

- en [1], la page 1,
- en [2], la page 2.

On peut naviguer entre les deux pages. Ce qu'on veut montrer ici, c'est que les pages 1 et 2 partagent une mise en forme commune comme il apparaît sur les copies d'écran ci-dessus.

2.11.2 Le projet Netbeans

Le projet Netbeans de l'application est le suivant :



L'application n'a que des pages XHTML. Il n'y a pas de modèle Java associé.

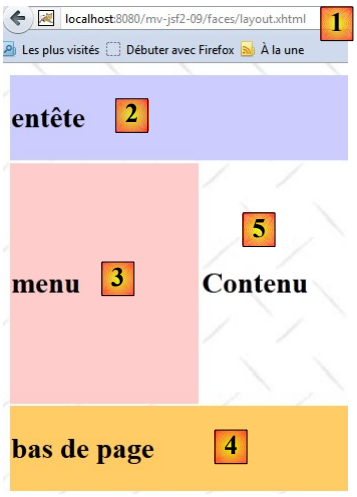
2.11.3 La page [layout.xhtml]

La page [layout.xhtml] fixe la forme des pages de l'application :

```
27. <?xml version='1.0' encoding='UTF-8' ?>
28. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
29.
30. <html xmlns="http://www.w3.org/1999/xhtml"
31.       xmlns:h="http://java.sun.com/jsf/html"
32.       xmlns:f="http://java.sun.com/jsf/core"
33.       xmlns:ui="http://java.sun.com/jsf/facelets">
34.   <h:head>
35.     <title>JSF</title>
36.     <h:outputStylesheet library="css" name="styles.css"/>
37.   </h:head>
38.   <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
39.     <h:form id="formulaire">
40.       <table style="width: 400px">
41.         <tr>
42.           <td colspan="2" bgcolor="#ccccff">
43.             <ui:include src="entete.xhtml"/>
44.           </td>
45.         </tr>
46.         <tr style="height: 200px">
47.           <td bgcolor="#ffcccc">
48.             <ui:include src="menu.xhtml"/>
49.           </td>
50.           <td>
51.             <ui:insert name="contenu" >
52.               <h2>Contenu</h2>
53.             </ui:insert>
54.           </td>
55.         </tr>
56.         <tr bgcolor="#ffcc66">
57.           <td colspan="2">
58.             <ui:include src="basdepage.xhtml"/>
59.           </td>
60.         </tr>
61.       </table>
62.     </h:form>
63.   </h:body>
64. </html>
```

Ligne 7, apparaît un nouvel espace de noms **ui**. Cet espace de noms contient les balises qui permettent de mettre en forme les pages d'une application. Les balises de cet espace sont utilisées aux lignes 17, 22, 25, 32.

La page [layout.xhtml] affiche des informations dans un tableau HTML (ligne 14). On peut demander cette page avec un navigateur :



- en [1], l'URL demandée.

La zone [2] a été générée par le code XHTML suivant :

```
1.    <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
2.      <h:form id="formulaire">
3.        <table style="width: 400px">
4.          <tr>
5.            <td colspan="2" bgcolor="#ccccff">
6.              <ui:include src="entete.xhtml"/>
7.            </td>
8.          </tr>
9.  ...
10.       </table>
11.     </h:form>
12. </h:body>
```

La balise **<ui:include>** de la ligne 6 permet d'inclure dans la page, un code XHTML externe. Le fichier [entete.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <body>
6.      <h2>entête</h2>
7.    </body>
8.  </html>
```

C'est tout le code des lignes 3-8 qui va être inséré dans [layout.xhtml]. Ainsi des balises <*html*> et <*body*> vont être insérées dans une balise <*td*>. Ca ne provoque pas d'erreurs. Donc, les pages insérées par <**ui:include**> sont des pages XHTML complètes. D'un point de vue visuel, seule la ligne 6 va avoir un effet. Les balises <*html*> et <*body*> sont présentes pour des raisons syntaxiques.

La zone [3] a été générée par le code XHTML suivant :

```
1.  <h:form id="formulaire">
2.        <table style="width: 400px">
3.          <tr style="height: 200px">
4.            <td bgcolor="#ffcccc">
5.              <ui:include src="menu.xhtml"/>
6.            </td>
7.  ...
8.          </tr>
9.  ...
10.       </table>
11.     </h:form>
```

La balise **<ui:include>** de la ligne 5 inclut le fichier [menu.xhtml] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <body>
6.      <h2>menu</h2>
7.    </body>
8.  </html>
```

La zone [4] a été générée par le code XHTML suivant :

```
1.  <h:form id="formulaire">
2.        <table style="width: 400px">
3.  ...
4.          <tr bgcolor="#ffcc66">
5.            <td colspan="2">
6.              <ui:include src="basdepage.xhtml"/>
7.            </td>
8.          </tr>
9.        </table>
10.     </h:form>
```

La balise **<ui:include>** de la ligne 6 inclut le fichier [basdepage.xhtml] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <body>
6.      <h2>bas de page</h2>
7.    </body>
8.  </html>
```

La zone [5] a été générée par le code XHTML suivant :

```
1.      <h:form id="formulaire">
2.  ...
3.            <td>
4.              <ui:insert name="contenu" >
5.                <h2>Contenu</h2>
6.              </ui:insert>
7.            </td>
8.   ...
9.        </table>
10. </h:form>
```

La balise **<ui:insert>** de la ligne 5 définit une zone appelée **contenu**. C'est une zone qui peut recevoir un contenu variable. Nous verrons comment. Lorsque nous avons demandé la page [layout.xhtml], aucun contenu n'a été défini pour la zone appelée **contenu**. Dans ce cas, le contenu de la balise <**ui:insert**> des lignes 4-6 est utilisé. La ligne 5 est donc affichée.

2.11.4 La page [page1.xhtml]

La page [layout.xhtml] n'est pas destinée à être visualisée. Elle sert de modèle aux pages [page1.xhtml] et [page2.xhtml]. On parle de **template** de pages. La page [page1.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.    <ui:composition template="layout.xhtml">
8.      <ui:define name="contenu">
9.        <h2>page 1</h2>
10.       <h:commandLink value="page 2" action="page2"/>
11.     </ui:define>
12.   </ui:composition>
13. </html>
```

- ligne 6, on utilise l'espace de noms **ui**,
- ligne 7, on indique que la page est associée au template [layout.xhtml] à l'aide d'une balise **<ui:composition>,**
- ligne 8, cette association fait que chaque balise <**ui:define**> sera associée à une balise <**ui:insert**> du template utilisé, ici [layout.xhtml]. Le lien se fait par l'attribut **name** des deux balises. Ils doivent être identiques.

La page affichée est [layout.xhtml] où le contenu de chaque balise <**ui:insert**> est remplacé par le contenu de la balise <**ui:define**> de la page demandée. Ici, tout se passe comme si la page affichée était :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <h:head>
9.      <title>JSF</title>
```

```
10.     <h:outputStylesheet library="css" name="styles.css"/>
11.   </h:head>
12.   <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
13.     <h:form id="formulaire">
14.       <table style="width: 400px">
15.         <tr>
16.           <td colspan="2" bgcolor="#ccccff">
17.             <ui:include src="entete.xhtml"/>
18.           </td>
19.         </tr>
20.         <tr style="height: 200px">
21.           <td bgcolor="#ffcccc">
22.             <ui:include src="menu.xhtml"/>
23.           </td>
24.           <td>
25.             <h2>page 1</h2>
26.             <h:commandLink value="page 2" action="page2"/>
27.           </td>
28.         </tr>
29.         <tr bgcolor="#ffcc66">
30.           <td colspan="2">
31.             <ui:include src="basdepage.xhtml"/>
32.           </td>
33.         </tr>
34.       </table>
35.     </h:form>
36.   </h:body>
37. </html>
```

Les lignes 25-26 de [page1.xhtml] ont été insérées en lieu et place de la balise <**ui:insert**> de [layout.xml].

La page [page2.xhtml] est analogue à [page1.xhtml] :

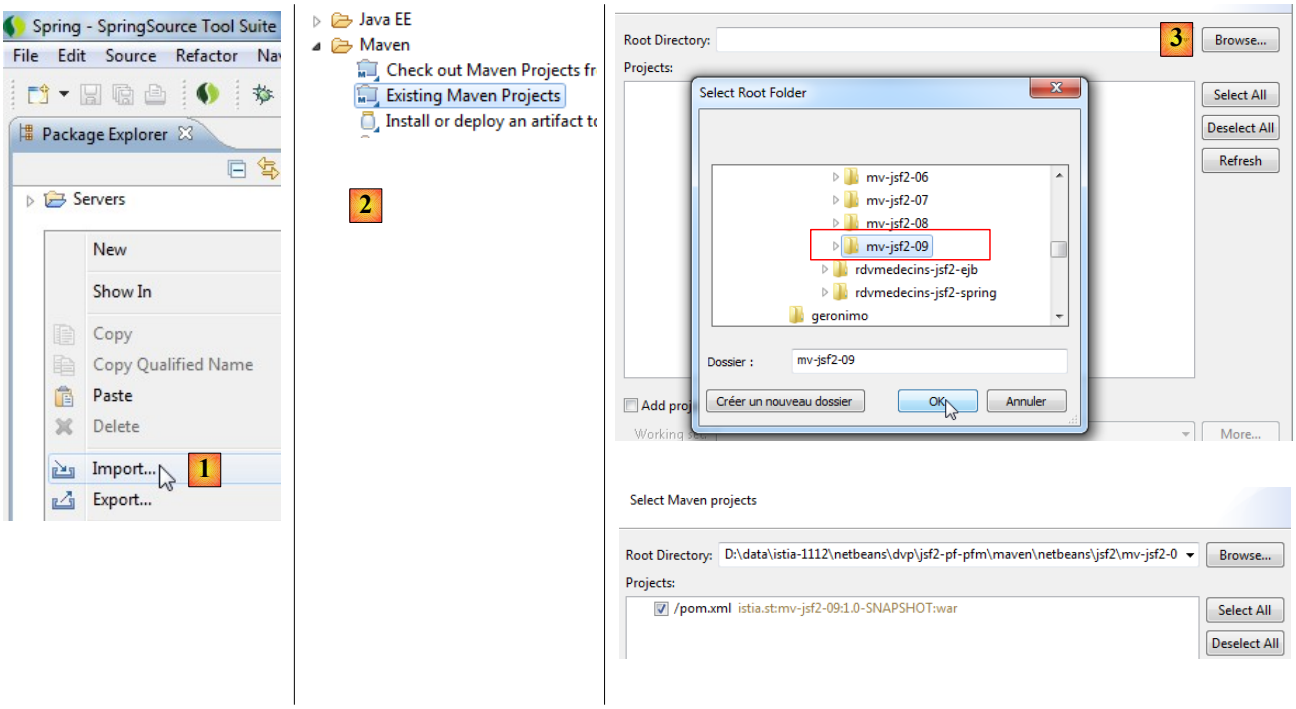
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.    <ui:composition template="layout.xhtml">
8.      <ui:define name="contenu">
9.        <h2>page 2</h2>
10.       <h:commandLink value="page 1" action="page1"/>
11.     </ui:define>
12.   </ui:composition>
13. </html>
```

2.12 Conclusion

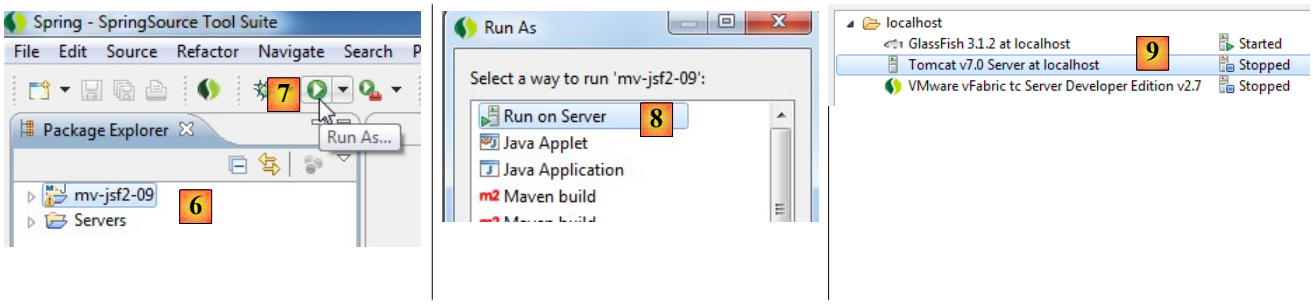
L'étude qui vient d'être faite de JSF 2 est loin d'être exhaustive. Mais elle est suffisante pour comprendre les exemples qui vont suivre. Pour approfondir, on lira [**ref2**].

2.13 Les tests avec Eclipse

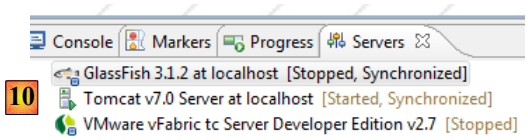
Montrons comment procéder aux tests des projets Maven avec **SpringSource Tool Suite** :

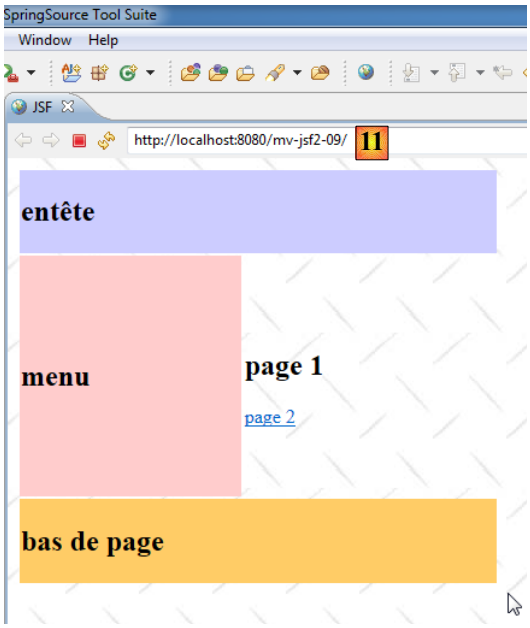


- en [1], on importe un projet Maven [2] qu'on désigne avec le bouton [3]. On prend ici le projet Maven [mv-jsf2-09] pour Eclipse
- en [4], le projet importé a bien été reconnu comme un projet Maven [5],



- en [6], le projet importé dans l'explorateur de projets,
- en [7], on l'exécute sur un serveur [8] Tomcat [9],





- en [10], Tomcat 7 a été lancé,
- en [11], la page d'accueil du projet [mv-jsf2-09] [11] s'affiche dans un navigateur interne à Eclipse.

3 Application exemple – 01 : rdvmedecins-jsf2-ejb

Le texte qui suit fait référence aux documents suivants :

- [**ref7**] : **Introduction à Java EE 5** (juin 2010) [http://tahe.developpez.com/java/javaee]. Ce document permet de découvrir **JSF 1** et les **EJB3**.
- [**ref8**] : **Persistance Java par la pratique** (juin 2007) [http://tahe.developpez.com/java/jpa]. Ce document permet de découvrir la persistence des données avec **JPA** (**Java Persistence API**).
- [**ref9**] : **Construire un service web Java EE avec Netbeans et le serveur Glassfish** (janvier 2009) [http://tahe.developpez.com/java/webservice-jee]. Ce document étudie la construction d'un service web.

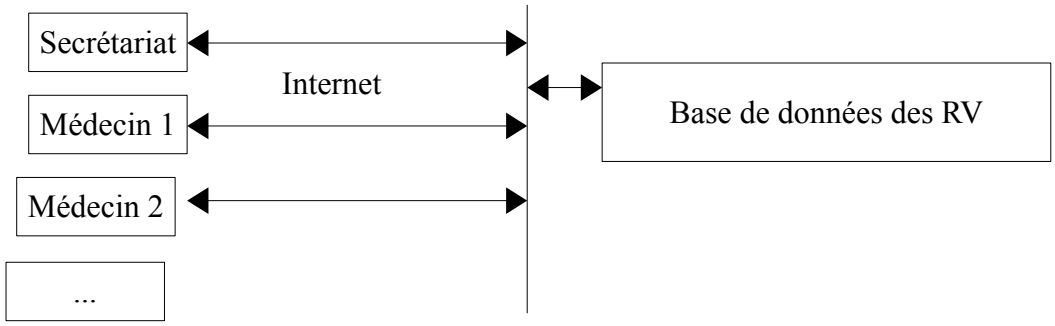
L'application exemple qui va être étudiée provient de [**ref9**].

3.1 L'application

Une société de services en informatique [ISTIA-AGI] désire proposer un service de prise de rendez-vous. Le premier marché visé est celui des médecins travaillant seuls. Ceux-ci n'ont en général pas de secrétariat. Les clients désirant prendre rendez-vous téléphonent alors directement au médecin. Celui-ci est ainsi dérangé fréquemment au cours d'une journée ce qui diminue sa disponibilité à ses patients. La société [ISTIA-AGI] souhaite leur proposer un service de prise de rendez-vous fonctionnant sur le principe suivant :

- un secrétariat assure les prises de RV pour un grand nombre de médecins. Ce secrétariat peut être réduit à une unique personne. Le salaire de celle-ci est mutualisé entre tous les médecins utilisant le service de RV.
- le secrétariat et tous les médecins sont reliés à Internet
- les RV sont enregistrés dans une base de données centralisée, accessible par Internet, par le secrétariat et les médecins
- la prise de RV est normalement faite par le secrétariat. Elle peut être faite également par les médecins eux-mêmes. C'est le cas notamment lorsqu'à la fin d'une consultation, le médecin fixe lui-même un nouveau RV à son patient.

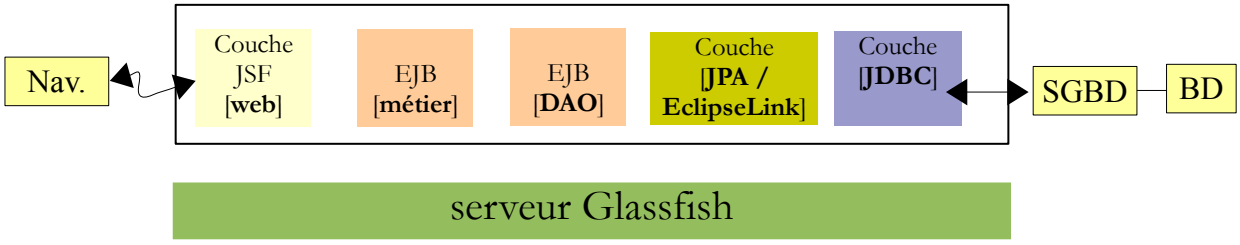
L'architecture du service de prise de RV est le suivant :



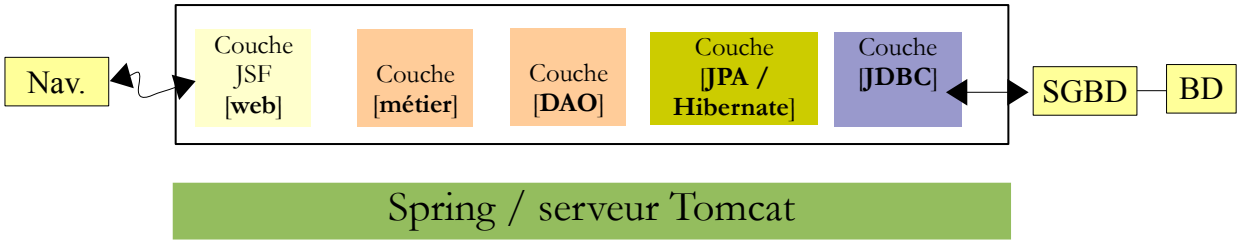
Les médecins gagnent en efficacité s'ils n'ont plus à gérer les RV. S'ils sont suffisamment nombreux, leur contribution aux frais de fonctionnement du secrétariat sera faible.

La société [ISTIA-AGI] décide de réaliser l'application en deux versions :

- une version JSF / EJB3 / JPA EclipseLink / serveur Glassfish :



- puis une version JSF / Spring / JPA Hibernate / serveur Tomcat :



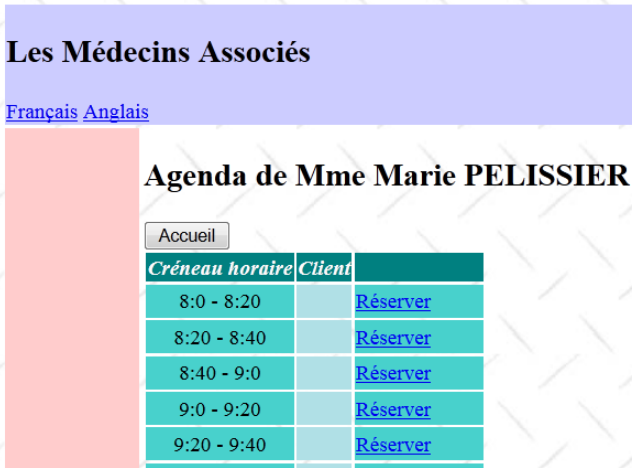
3.2 Fonctionnement de l'application

Nous appellerons [RdvMedecins] l'application. Nous présentons ci-dessous des copies d'écran de son fonctionnement.

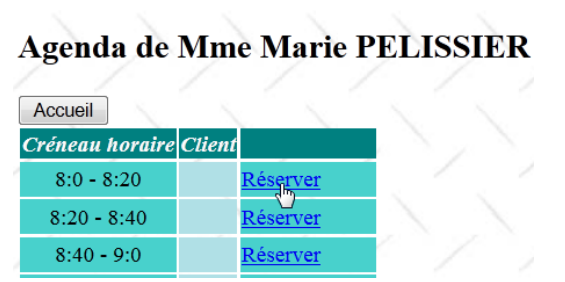
La page d'accueil de l'application est la suivante :

A partir de cette première page, l'utilisateur (Secrétariat, Médecin) va engager un certain nombre d'actions. Nous les présentons ci-dessous. La vue de gauche présente la vue à partir de laquelle l'utilisateur fait une **demande**, la vue de droite la **réponse** envoyée par le serveur.

L'utilisateur a sélectionné un médecin et a saisi un jour de RV



On obtient la liste (vue partielle ici) des RV du médecin sélectionné pour le jour indiqué.



On réserve

On obtient une page à renseigner

On la renseigne

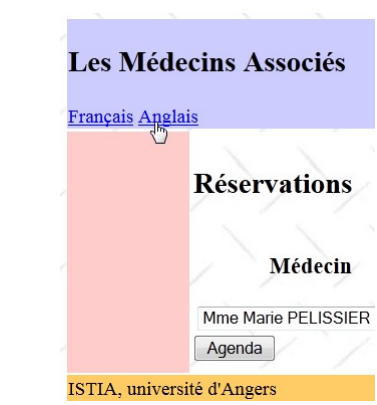
Le nouveau RV apparaît dans la liste

L'utilisateur peut supprimer un RV

Le RV supprimé a disparu de la liste des RV

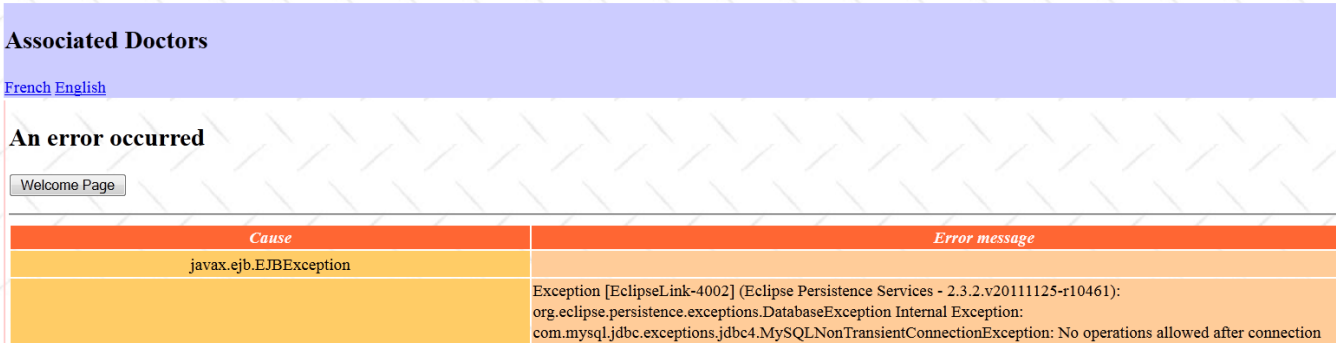
Une fois l'opération de prise de RV ou d'annulation de RV faite, l'utilisateur peut revenir à la page d'accueil

Il la retrouve dans son dernier état



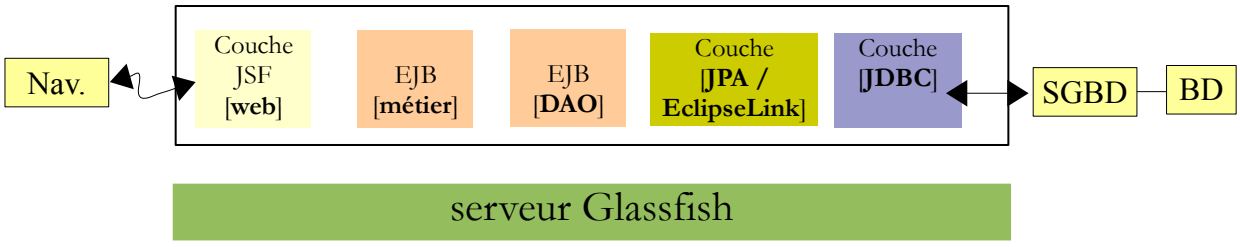
On peut changer la langue	La langue a été changée

Enfin, on peut également obtenir une page d'erreurs :

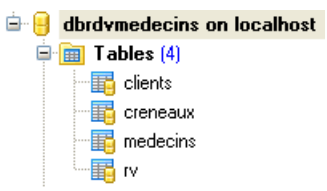


3.3 La base de données

Revenons à l'architecture de l'application à construire :

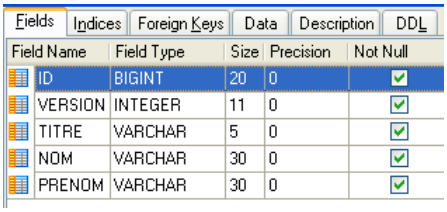


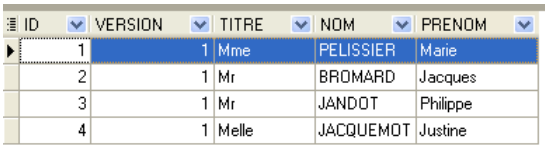
La base de données qu'on appellera [dbrdvmedecins2] est une base de données MySQL5 avec quatre tables :



3.3.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].



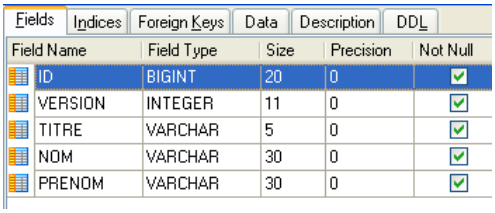


ID	VERSION	TITRE	NOM	PRENOM
1	1	Mme	PELISSIER	Marie
2	1	Mr	BROMARD	Jacques
3	1	Mr	JANDOT	Philippe
4	1	Melle	JACQUEMOT	Justine

- 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)

3.3.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :

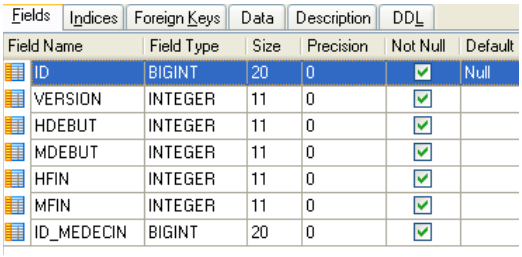


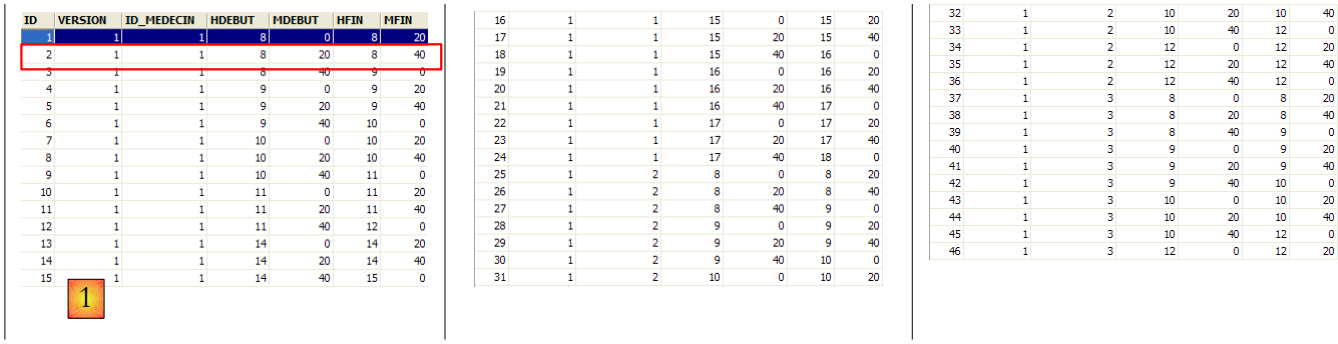
ID	VERSION	TITRE	NOM	PRENOM
1	1	Mr	MARTIN	Jules
2	1	Mme	GERMAN	Christine
3	1	Mr	JACQUARD	Jules
4	1	Melle	BISTROU	Brigitte

- 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)

3.3.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :



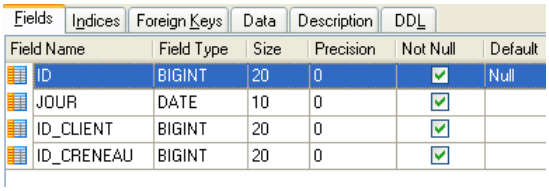


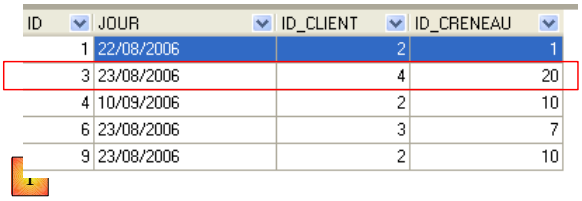
- ID : n° identifiant le créneau horaire - clé primaire de la table (ligne 8)
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – 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).

3.3.4 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) :

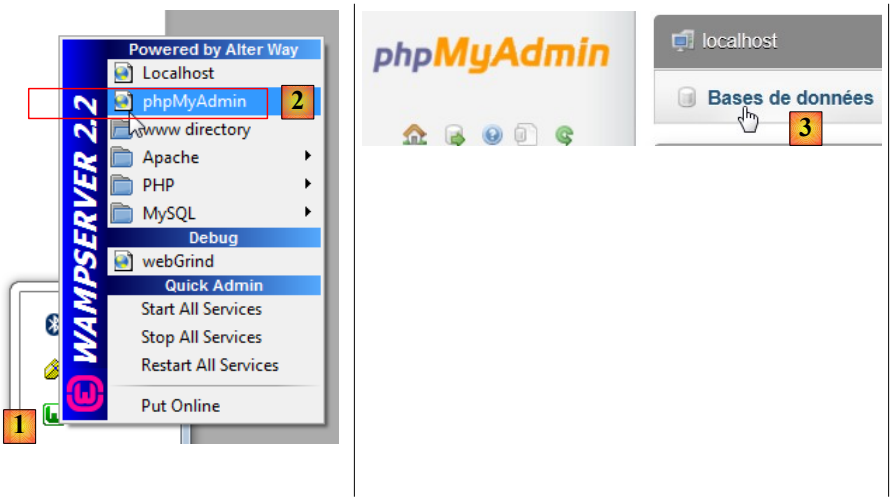
```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

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.

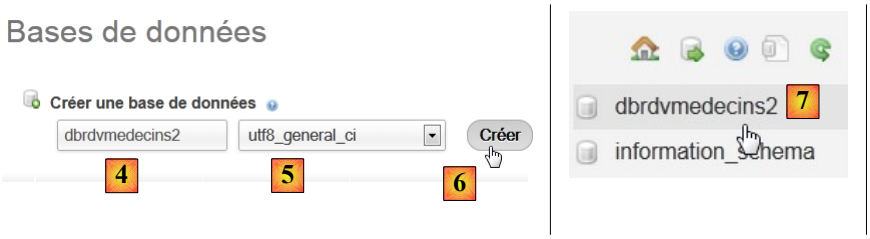
La ligne d'*id* égal à 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTROU.

3.3.5 Génération de la base

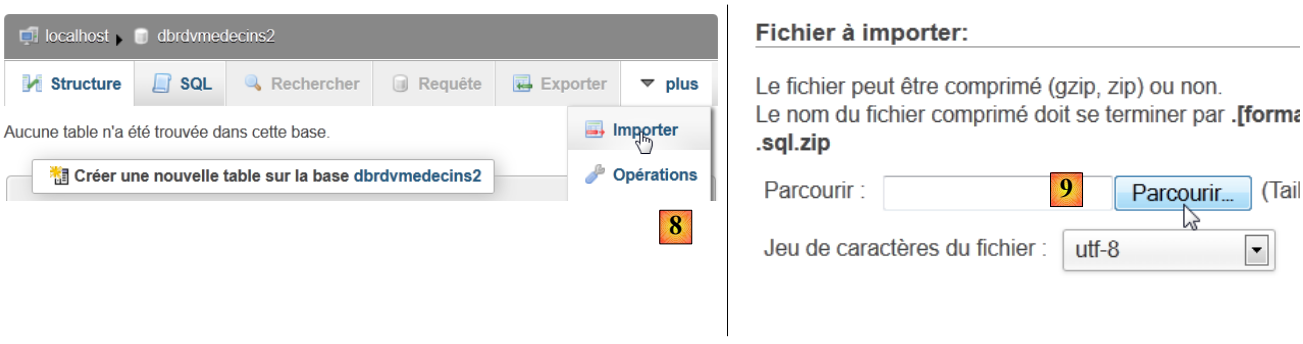
Pour créer les tables et les remplir on pourra utiliser le script [dbrdvmedecins2.sql] qu'on trouvera sur le site des exemples. Avec [WampServer] (cf paragraphe 1.3.3, page 14), on pourra procéder comme suit :



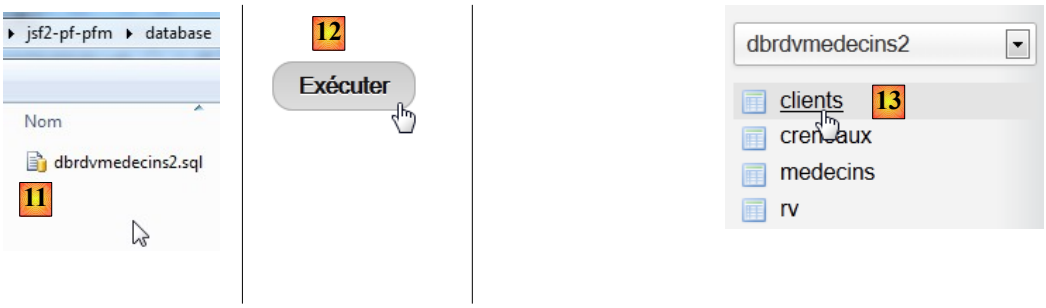
- en [1], on clique sur l'icône de [WampServer] et on choisit l'option [PhpMyAdmin] [2],
- en [3], dans la fenêtre sui s'est ouverte, on sélectionne le lien [Bases de données],



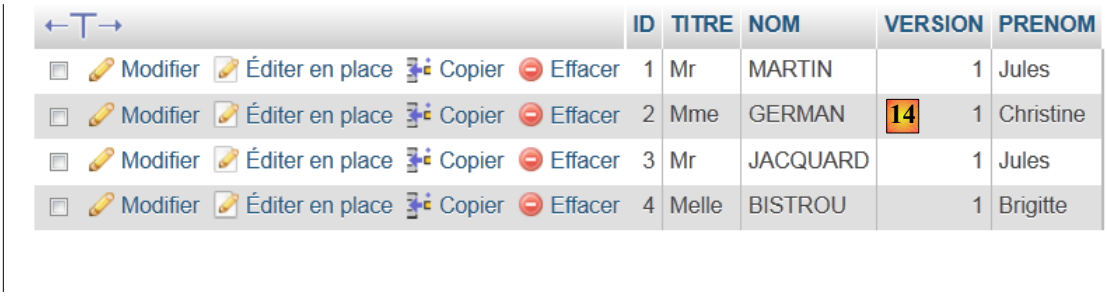
- en [2], on crée une base de données dont on a donné le nom [4] et l'encodage [5],
- en [7], la base a été créée. On clique sur son lien,



- en [8], on importe un fichier SQL,
- qu'on désigne dans le système de fichiers avec le bouton [9],



- en [11], on sélectionne le script SQL et en [12] on l'exécute,
- en [13], les quatre tables de la base ont été créées. On suit l'un des liens,

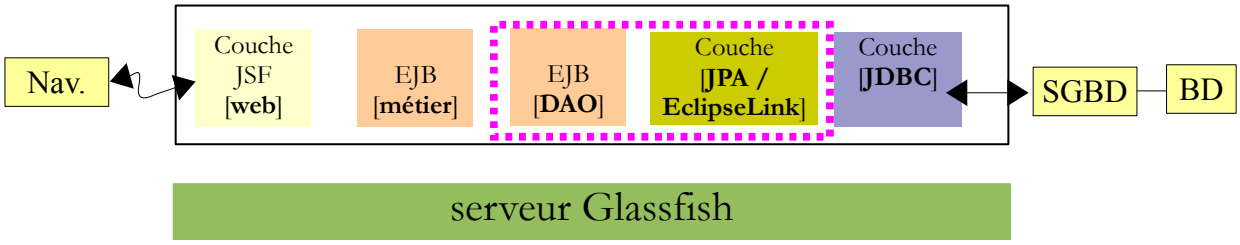


- en [14], le contenu de la table.

Par la suite, nous ne reviendrons plus sur cette base. Mais le lecteur est invité à suivre son évolution au fil des programmes surtout lorsque ça ne marche pas.

3.4 Les couches [DAO] et [JPA]

Revenons à l'architecture que nous devons construire :



Nous allons construire quatre projets Maven :

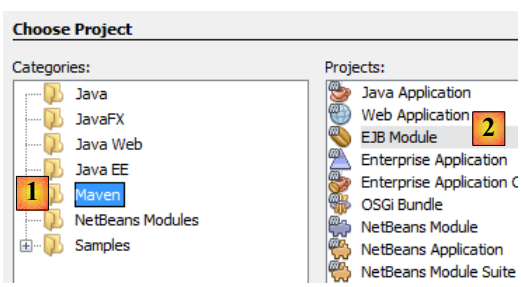
- un projet pour les couches [DAO] et [JPA],
- un projet pour la couche [métier],
- un projet pour la couche [web],
- un projet d'entreprise qui va rassembler les trois projets précédents.

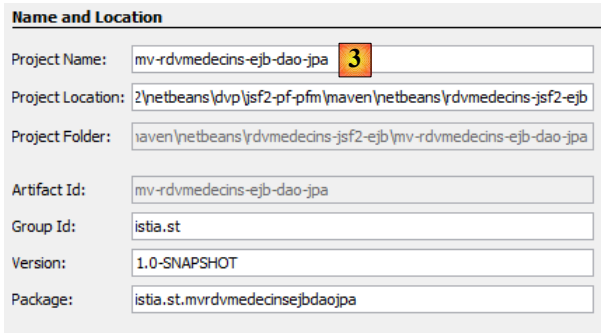
Nous construisons maintenant le projet Maven des couches [DAO] et [JPA].

Note : la compréhension des couches [métier], [DAO], [JPA] nécessite des connaissances en Java EE. Pour cela, on pourra lire [**ref7**] (cf page 154).

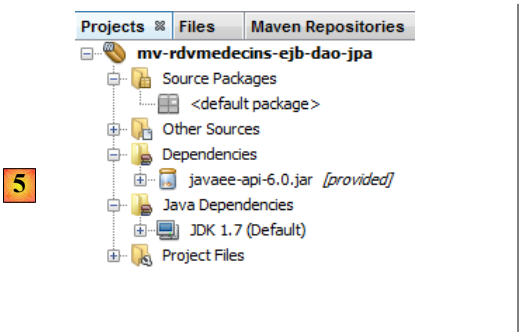
3.4.1 Le projet Netbeans

C'est le suivant :





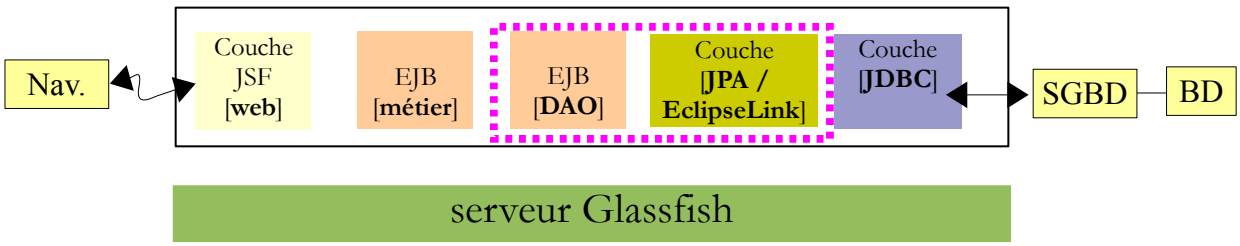
- en [1], on construit un projet Maven de type [EJB Module] [2],
- en [3], on donne un nom au projet,



- en [4], on choisit comme serveur le serveur Glassfish,
- en [5], le projet généré.

3.4.2 Génération de la couche [JPA]

Revenons à l'architecture que nous devons construire :

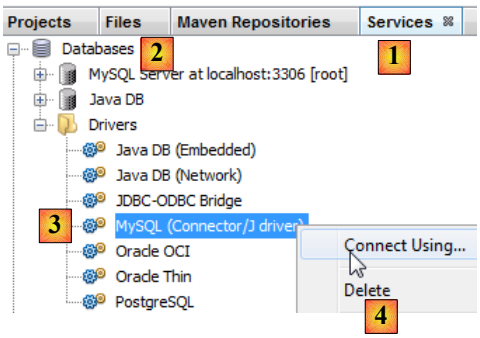


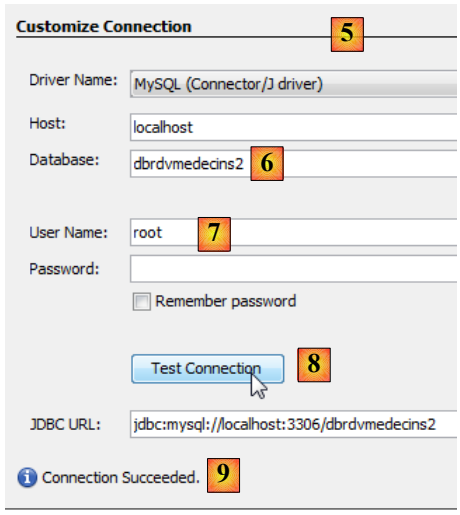
Avec Netbeans, il est possible de générer automatiquement la couche [JPA] et la couche [EJB] qui contrôle l'accès aux entités JPA générées. Il est intéressant de connaître ces méthodes de génération automatique car le code généré donne de précieuses indications sur la façon d'écrire des entités JPA ou le code EJB qui les utilise.

Nous décrivons maintenant certains de ces outils de génération automatique. Pour comprendre le code généré, il faut avoir de bonnes notions sur les entités JPA [**ref8**] et les EJB [**ref7**] (cf page 154).

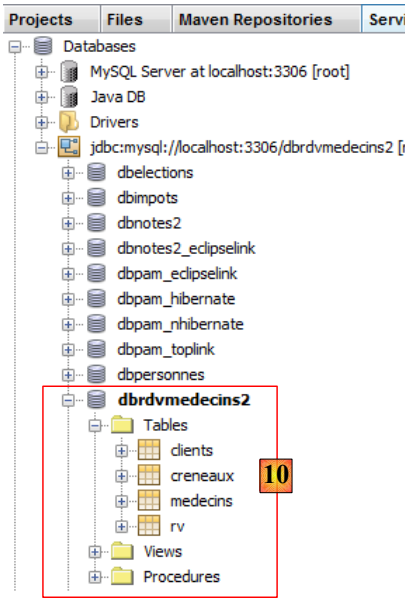
3.4.2.1 *Création d'une connexion Netbeans à la base de données*

- lancer le SGBD MySQL 5 afin que la BD soit disponible,
- créer une connexion Netbeans sur la base [dbrdvmedecins2],





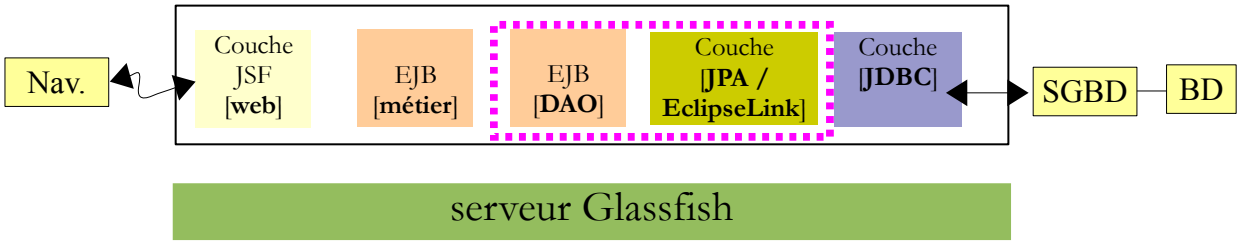
- dans l'onglet [Services] [1], dans la branche [Databases] [2], sélectionner le pilote JDBC MySQL [3],
- puis sélectionner l'option [4] "Connect Using" permettant de créer une connexion avec une base MySQL,
- en [5], donner les informations qui sont demandées. En [6], le nom de la base, en [7] l'utilisateur de la base et son mot de passe,
- en [8], on peut tester les informations qu'on a fournies,
- en [9], le message attendu lorsque celles-ci sont bonnes,



- en [10], la connexion est créée. On y voit les quatre tables de la base de données connectée.

3.4.2.2 *Création d'une unité de persistance*

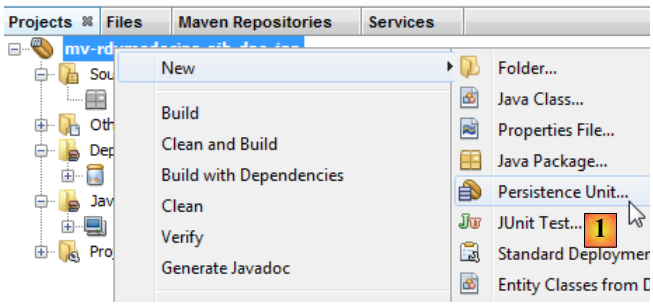
Revenons à l'architecture en cours de construction :

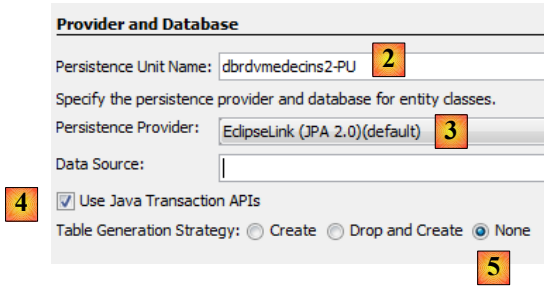


Nous sommes en train de construire la couche [JPA]. La configuration de celle-ci est faite dans un fichier [persistence.xml] dans lequel on définit des unités de persistance. Chacune d'elles a besoin des informations suivantes :

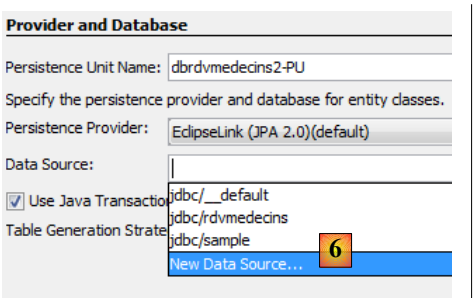
- les caractéristiques JDBC d'accès à la base (URL, utilisateur, mot de passe),
- les classes qui seront les images des tables de la base de données,
- l'implémentation JPA utilisée. En effet, JPA est une spécification implémentée par divers produits. Ici, nous utiliserons EclipseLink qui est l'implémentation par défaut utilisée par le serveur Glassfish. Cela nous évite d'adjoindre à Glassfish les bibliothèques d'une autre implémentation.

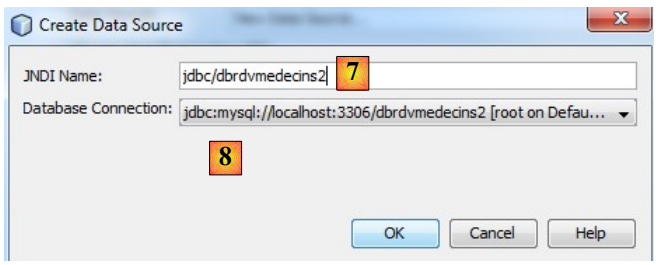
Netbeans peut générer ce fichier de persistance via l'utilisation d'un assistant.





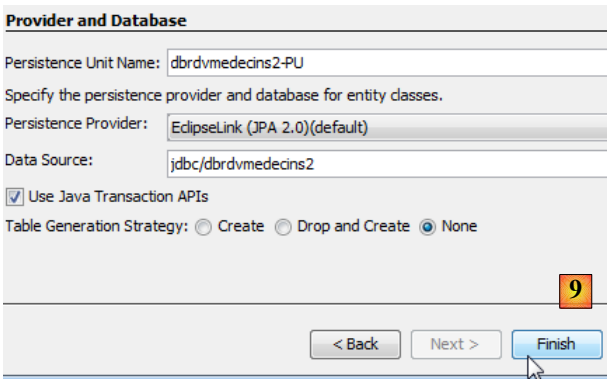
- cliquer droit sur le projet et choisir la création d'une unité de persistance [1],
- en [2], donner un nom à l'unité de persistance que l'on crée,
- en [3], choisir l'implémentation JPA EclipseLink (JPA 2.0),
- en [4], indiquer que les transactions avec la base de données seront gérées par le conteneur EJB du serveur Glassfish,
- en [5], indiquer que les tables de la BD sont déjà créées et que donc on ne les crée pas,

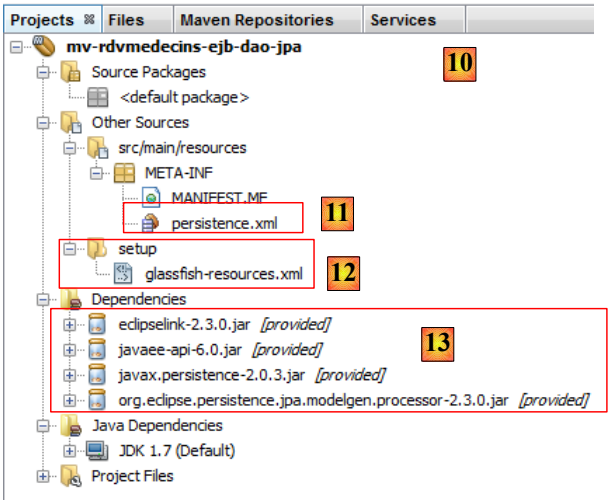




- en [6], créer une nouvelle source de données pour le serveur Glassfish,
- en [7], donner un nom JNDI (Java Naming Directory Interface),

- en [8], relier ce nom à la connexion MySQL créée à l'étape précédente,





- en [9], terminer l'assistant,
- en [10], le nouveau projet,
- en [11], le fichier [persistence.xml] a été généré dans le dossier [META-INF],
- en [12], un dossier [setup] a été généré,
- en [13], de nouvelles dépendances ont été ajoutées au projet Maven.

Le fichier [META-INF/persistence.xml] généré est le suivant :

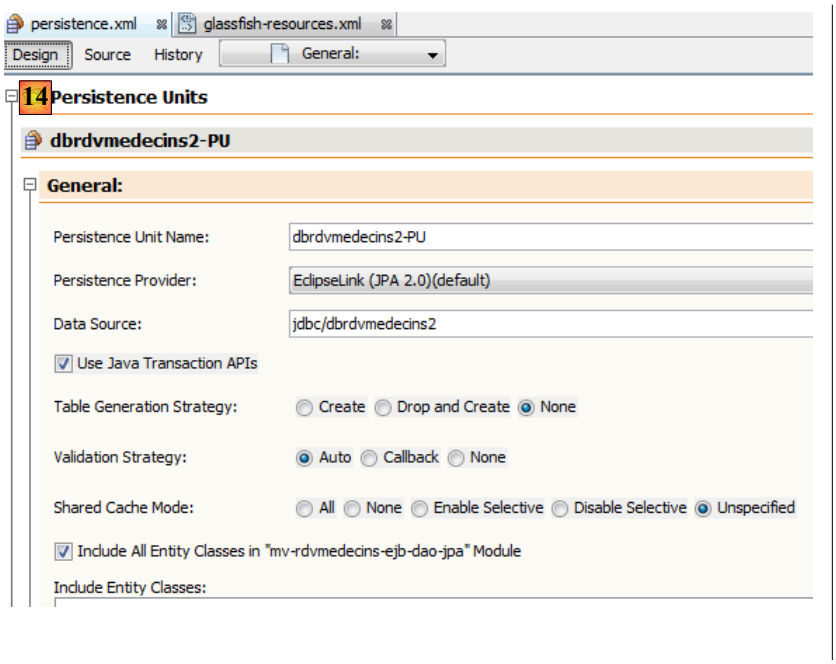
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
3.    <persistence-unit name="dbrdvmedecins2-PU" transaction-type="JTA">
4.      <jta-data-source>jdbc/dbrdvmedecins2</jta-data-source>
5.      <exclude-unlisted-classes>false</exclude-unlisted-classes>
6.      <properties/>
7.    </persistence-unit>
8.  </persistence>
```

Il reprend les informations données dans l'assistant :

- ligne 3 : le nom de l'unité de persistance,
- ligne 3 : le type de transactions avec la base de données, ici des transactions JTA (Java Transaction Api) gérées par le conteneur EJB3 du serveur Glassfish,
- ligne 4 : le nom JNDI de la source de données.

Normalement, on trouve dans ce fichier le type d'implémentation JPA utilisée. Dans l'assistant, nous avons indiqué EclipseLink. Comme c'est l'implémentation JPA utilisée par défaut par le serveur Glassfish, elle n'est pas mentionnée dans le fichier [persistence.xml].

Dans l'onglet [Design], on peut avoir une vue globale du fichier [persistence.xml] :



Pour avoir des logs d'EclipseLink, nous utiliserons le fichier [persistence.xml] suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
3.    <persistence-unit name="dbrdvmedecins2-PU" transaction-type="JTA">
4.      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
5.      <jta-data-source>jdbc/dbrdvmedecins2</jta-data-source>
6.      <exclude-unlisted-classes>false</exclude-unlisted-classes>
7.      <properties>
8.        <property name="eclipselink.logging.level" value="FINE"/>
9.      </properties>
10.   </persistence-unit>
11. </persistence>
```

- ligne 4 : on indique qu'on utilise l'implémentation JPA d'EclipseLink,
- lignes 7-9 : rassemblent les propriétés de configuration du provider JPA, ici EclipseLink,
- ligne 8 : cette propriété permet de loguer les ordres SQL que va émettre EclipseLink.

Le fichier [glassfish-resources.xml] qui a été créé est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE resources PUBLIC "-//GlassFish.org//DTD GlassFish Application Server 3.1 Resource
    Definitions//EN" "http://glassfish.org/dtds/glassfish-resources_1_5.dtd">
3.  <resources>
4.      <jdbc-connection-pool allow-non-component-callers="false" ... steady-pool-size="8" validate-atmost-
    once-period-in-seconds="0" wrap-jdbc-objects="false">
5.          <property name="serverName" value="localhost"/>
6.          <property name="portNumber" value="3306"/>
7.          <property name="databaseName" value="dbrdvmedecins2"/>
8.          <property name="User" value="root"/>
9.          <property name="Password" value=""/>
10.         <property name="URL" value="jdbc:mysql://localhost:3306/dbrdvmedecins2"/>
11.         <property name="driverClass" value="com.mysql.jdbc.Driver"/>
12.     </jdbc-connection-pool>
13.     <jdbc-resource enabled="true" jndi-name="jdbc/dbrdvmedecins2" object-type="user" pool-
    name="mysql_dbrdvmedecins2_rootPool"/>
14. </resources>
```

Ce fichier reprend les informations que nous avons données dans les deux assistants utilisés précédemment :

- lignes 5-11 : les caractéristiques JDBC de la base de données MySQL5 [dbrdvmedecins2],
- ligne 13 : le nom JNDI de la source de données.

Ce fichier va être utilisé pour créer la source de données JNDI [jdbc/dbrdvmedecins2] du serveur Glassfish. C'est complètement propriétaire à ce serveur. Pour un autre serveur, il faudrait s'y prendre autrement, généralement par un outil d'administration. Celui-ci existe également pour Glassfish.

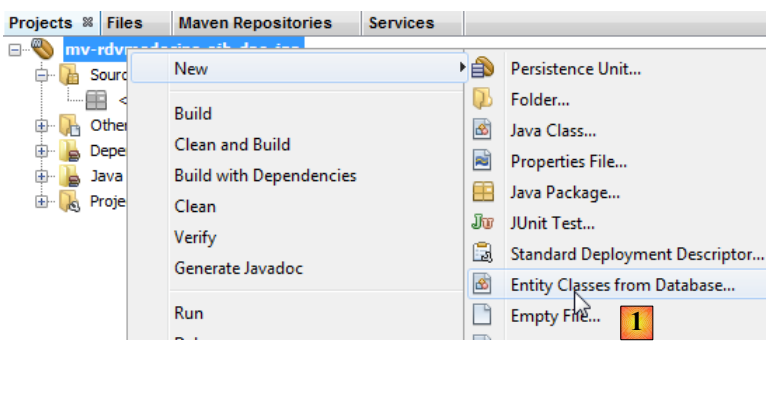
Enfin, des dépendances ont été ajoutées au projet. Le fichier [pom.xml] est le suivant :

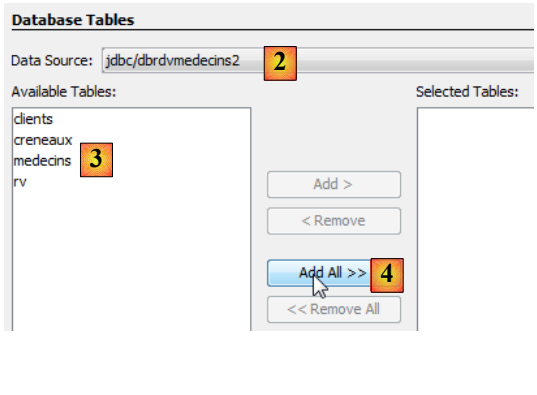
```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    4.0.0.xsd">
3.      <modelVersion>4.0.0</modelVersion>
4.
5.      <groupId>istia.st</groupId>
6.      <artifactId>mv-rdvmedecins-ejb-dao-jpa</artifactId>
7.      <version>1.0-SNAPSHOT</version>
8.      <packaging>ejb</packaging>
9.
10.     <name>mv-rdvmedecins-ejb-dao-jpa</name>
11.
12.     ...
13.     <dependencies>
14.         <dependency>
15.             <groupId>org.eclipse.persistence</groupId>
16.             <artifactId>eclipselink</artifactId>
17.             <version>2.3.0</version>
18.             <scope>provided</scope>
19.         </dependency>
20.         <dependency>
21.             <groupId>org.eclipse.persistence</groupId>
22.             <artifactId>javax.persistence</artifactId>
23.             <version>2.0.3</version>
24.             <scope>provided</scope>
25.         </dependency>
26.         <dependency>
27.             <groupId>org.eclipse.persistence</groupId>
28.             <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
29.             <version>2.3.0</version>
30.             <scope>provided</scope>
31.         </dependency>
32.         <dependency>
33.             <groupId>javax</groupId>
34.             <artifactId>javaee-api</artifactId>
35.             <version>6.0</version>
36.             <scope>provided</scope>
37.         </dependency>
38.     </dependencies>
39.
40. ...
41.     <repositories>
42.         <repository>
43.             <url>http://download.eclipse.org/rt/eclipselink/maven.repo/</url>
44.             <id>eclipselink</id>
45.             <layout>default</layout>
46.             <name>Repository for library Library[eclipselink]</name>
47.         </repository>
48.     </repositories>
49. </project>
```

- ligne 32-37, une couche [JPA] nécessite l'artifact [javaee-api],
- lignes 16, 22, 28, les artifacts nécessités par l'implémentation JPA / EclipseLink utilisée ici.
- lignes 18, 24, 30, 36 : tous les artifacts ont l'attribut *provided*. On rappelle que cela veut dire qu'ils sont nécessaires pour la compilation mais pas pour l'exécution. En effet, lors de l'exécution, ils sont fournis (*provided*) par le serveur Glassfish,
- lignes 41-48 : définissent un nouveau dépôt d'artifacts Maven, celui où les artifacts EclipseLink peuvent être trouvés.

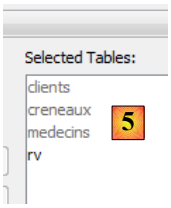
3.4.2.3 Génération des entités JPA

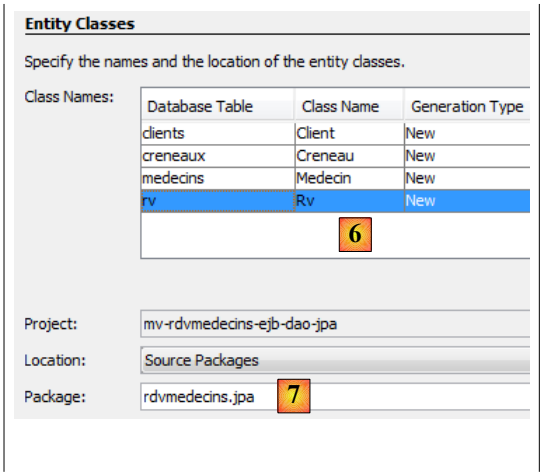
Les entités JPA peuvent être générées par un assistant de Netbeans :

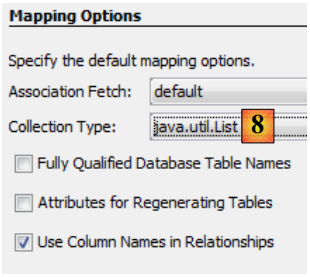




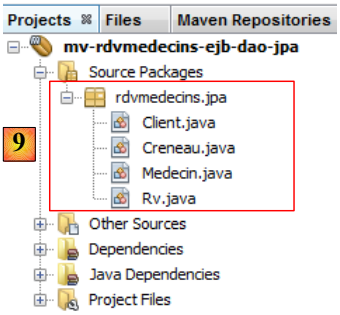
- en [1], on crée des entités JPA à partir d'une base de données,
- en [2], on sélectionne la source de données [jdbc / dbrdvmedecins2] créée précédemment,
- en [3], la liste des tables de cette source de données,
- en [4], on les prend toutes,







- en [5], les tables sélectionnées,
- en [6], on donne un nom aux classes Java associées aux quatre tables,
- ainsi qu'un nom de paquetage [7],
- en [8], JPA rassemble des lignes de tables de BD dans des collections. Nous choisissons la liste comme collection,



- en [9], les classes Java créées par l'assistant.

3.4.2.4 *Les entités JPA générées*

L'entité [Medecin] est l'image de la table [medecins]. La classe Java est truffée d'annotations qui rendent le code peu lisible au premier abord. Si on ne garde que ce qui est essentiel à la compréhension du rôle de l'entité, on obtient le code suivant :

```
1.  package rdvmedecins.jpa;
2.
3.  ...
4.  @Entity
5.  @Table(name = "medecins")
6.  public class Medecin implements Serializable {
7.
8.  @Id
9.    @GeneratedValue(strategy = GenerationType.IDENTITY)
10.   @Column(name = "ID")
11.   private Long id;
12.
13.   @Column(name = "TITRE")
14.   private String titre;
15.
16.   @Column(name = "NOM")
17.   private String nom;
18.
19.   @Column(name = "VERSION")
20.   private int version;
21.
22.   @Column(name = "PRENOM")
23.   private String prenom;
24.
25.   @OneToMany(cascade = CascadeType.ALL, mappedBy = "idMedecin")
26.   private List<Creneau> creneauList;
27.
28. // constructeurs
29. ....
30.
31.   // getters et setters
32. ....
33.
34.   @Override
35.   public int hashCode() {
36.   ...
37.   }
38.
39.   @Override
40.   public boolean equals(Object object) {
41.   ...
42.   }
43.
44.   @Override
45.   public String toString() {
46.     ...
47.   }
48.
49. }
```

- ligne 4, l'annotation **@Entity** fait de la classe [Medecin], une entité JPA, c.a.d. une classe liée à une table de BD via l'API JPA,
- ligne 5, le nom de la table de BD associée à l'entité JPA. Chaque champ de la table fait l'objet d'un champ dans la classe Java,
- ligne 6, la classe implémente l'interface **Serializable**. Ceci est nécessaire dans les applications client / serveur, où les entités sont sérialisées entre le client et le serveur.
- lignes 10-11 : le champ **id** de la classe [Medecin] correspond au champ [ID] (ligne 10) de la table [medecins],
- lignes 13-14 : le champ **titre** de la classe [Medecin] correspond au champ [TITRE] (ligne 13) de la table [medecins],
- lignes 16-17 : le champ **nom** de la classe [Medecin] correspond au champ [NOM] (ligne 16) de la table [medecins],
- lignes 19-20 : le champ **version** de la classe [Medecin] correspond au champ [VERSION] (ligne 19) de la table [medecins]. Ici, l'assistant ne reconnaît pas le fait que la colonne est en fait un colonne de **version** qui doit être incrémentée à chaque modification de la ligne à laquelle elle appartient. Pour lui donner ce rôle, il faut ajouter l'annotation **@Version**. Nous le ferons dans une prochaine étape,
- lignes 22-23 : le champ **prenom** de la classe [Medecin] correspond au champ [PRENOM] de la table [medecins],
- lignes 10-11 : le champ **id** correspond à la clé primaire [ID] de la table. Les annotations des lignes 8-9 précisent ce point,

- ligne 8 : l'annotation **@Id** indique que le champ annoté est associé à la **clé primaire** de la table,
- ligne 9 : la couche [JPA] va générer la clé primaire des lignes qu'elle insèrera dans la table [Medecins]. Il y a plusieurs stratégies possibles. Ici la stratégie *GenerationType.IDENTITY* indique que la couche JPA va utiliser le mode *auto_increment* de la table MySQL,
- lignes 25-26 : la table [creneaux] a une clé étrangère sur la table [medecins]. Un créneau appartient à un médecin. Inversement, un médecin a plusieurs créneaux qui lui sont associés. On a donc une relation **un** (médecin) **à plusieurs** (créneaux), une relation qualifiée par l'annotation **@OneToMany** par JPA (ligne 25). Le champ de la ligne 26 contiendra tous les créneaux du médecin. Ceci sans programmation. Pour comprendre totalement la ligne 25, il nous faut présenter la classe [Creneau].

Celle-ci est la suivante :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
4.  import java.util.List;
5.  import javax.persistence.*;
6.  import javax.validation.constraints.NotNull;
7.
8.  @Entity
9.  @Table(name = "creneaux")
10. public class Creneau implements Serializable {
11.   @Id
12.   @GeneratedValue(strategy = GenerationType.IDENTITY)
13.   @Column(name = "ID")
14.   private Long id;
15.
16.   @Column(name = "MDEBUT")
17.   private int mdebut;
18.
19.   @Column(name = "HFIN")
20.   private int hfin;
21.
22.   @Column(name = "HDEBUT")
23.   private int hdebut;
24.
25.   @Column(name = "MFIN")
26.   private int mfin;
27.
28.   @Column(name = "VERSION")
29.   private int version;
30.
31.   @JoinColumn(name = "ID_MEDECIN", referencedColumnName = "ID")
32.   @ManyToOne(optional = false)
33.   private Medecin idMedecin;
34.
35.   @OneToMany(cascade = CascadeType.ALL, mappedBy = "idCreneau")
36.   private List<Rv> rvList;
37.
38. // constructeurs
39. ...
40. // getters et setters
41. ...

42.   @Override
43.   public int hashCode() {
44.     ...
45.   }
46.
47.   @Override
48.   public boolean equals(Object object) {
49.     ...
50.   }
51.
52.   @Override
53.   public String toString() {
54.     ...
55.   }
56.
57. }
```

Nous ne commentons que les nouvelles annotations :

- nous avons dit que la table [creneaux] avait une clé étrangère vers la table [medecins] : un créneau est associé à un médecin. Plusieurs créneaux peuvent être asssociés au même médecin. On a une relation de la table [creneaux] vers la table [medecins] qui est qualifiée de **plusieurs** (créneaux) **à un** (médecin). C'est l'annotation **@ManyToOne** de la ligne 32 qui sert à qualifier la clé étrangère,
- la ligne 31 avec l'annotation **@JoinColumn** précise la relation de clé étrangère : la colonne [ID_MEDECIN] de la table [creneaux] est clé étrangère sur la colonne [ID] de la table [medecins],
- ligne 33 : une référence sur le médecin propriétaire du créneau. On l'obtient là encore sans programmation.

Le lien de clé étrangère entre l'entité [Creneau] et l'entité [Medecin] est donc matérialisé par deux annotations :

- dans l'entité [Creneau] :

```
1.  @JoinColumn(name = "ID_MEDECIN", referencedColumnName = "ID")
2.    @ManyToOne(optional = false)
3.  private Medecin idMedecin;
```

- dans l'entité [Medecin] :

```
1.  @OneToMany(cascade = CascadeType.ALL, mappedBy = "idMedecin")
2.  private List<Creneau> creneauList;
```

Les deux annotations reflètent la même relation : celle de la clé étrangère de la table [creneaux] vers la table [medecins]. On dit qu'elles sont **inverses** l'une de l'autre. Seule la relation **@ManyToOne** est indispensable. Elle qualifie sans ambiguïté la relation de clé étrangère. La relation **@OneToMany** est facultative. Si elle est présente, elle se contente de référencer la relation **@ManyToOne** à laquelle elle est associée. C'est le sens de l'attribut **mappedBy** de la ligne 1 de l'entité [Medecin]. La valeur de cet attribut est le nom du champ de l'entité [Creneau] qui a l'annotation **@ManyToOne** qui spécifie la clé étrangère. Toujours dans cette même ligne 1 de l'entité [Medecin], l'attribut **cascade=CascadeType.ALL** fixe le comportement de l'entité [Medecin] vis à vis de l'entité [Creneau] :

- si on insère une nouvelle entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être insérées elles-aussi,
- si on modifie une entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être modifiées elles-aussi,
- si on supprime une entité [Medecin] dans la base, alors les entités [Creneau] du champ de la ligne 2 doivent être supprimées elles-aussi.

Nous donnons le code des deux autres entités sans commentaires particuliers puisqu'elles n'introduisent pas de nouvelles notations.

L'entité [**Client**]

```
1.  package rdvmedecins.jpa;
2.
3.  ...
4.  @Entity
5.  @Table(name = "clients")
6.  public class Client implements Serializable {
7.    @Id
8.    @GeneratedValue(strategy = GenerationType.IDENTITY)
9.    @Column(name = "ID")
10.   private Long id;
11.
12.   @Column(name = "TITRE")
13.   private String titre;
14.
15.   @Column(name = "NOM")
16.   private String nom;
17.
18.   @Column(name = "VERSION")
19.   private int version;
20.
21.   @Column(name = "PRENOM")
22.   private String prenom;
23.
24.   @OneToMany(cascade = CascadeType.ALL, mappedBy = "idClient")
25.   private List<Rv> rvList;
26.
27. // constructeurs
28. ...
```

```
29. // getters et setters
30. ...
31.
32.   @Override
33.   public int hashCode() {
34.     ...
35.   }
36.
37.   @Override
38.   public boolean equals(Object object) {
39.     ...
40.   }
41.
42.   @Override
43.   public String toString() {
44.     ...
45.   }
46.
47. }
```

- les lignes 24-25 reflètent la relation de clé étrangère entre la table [rv] et la table [clients].

L'entité [**Rv**] :

```
1.  package rdvmedecins.jpa;
2.
3.  ...
4.  @Entity
5.  @Table(name = "rv")
6.  public class Rv implements Serializable {
7.    @Id
8.    @GeneratedValue(strategy = GenerationType.IDENTITY)
9.    @Column(name = "ID")
10.   private Long id;
11.
12.   @Column(name = "JOUR")
13.   @Temporal(TemporalType.DATE)
14.   private Date jour;
15.
16.   @JoinColumn(name = "ID_CRENEAU", referencedColumnName = "ID")
17.   @ManyToOne(optional = false)
18.   private Creneau idCreneau;
19.
20.   @JoinColumn(name = "ID_CLIENT", referencedColumnName = "ID")
21.   @ManyToOne(optional = false)
22.   private Client idClient;
23.
24.   // constructeurs
25. ...
26.
27.   // getters et setters
28. ...
29.
30.   @Override
31.   public int hashCode() {
32.     ...
33.   }
34.
35.   @Override
36.   public boolean equals(Object object) {
37.     ...
38.   }
39.
40.   @Override
41.   public String toString() {
42.     ...
43.   }
44.
45. }
```

- la ligne 13 qualifie le champ jour de type Java *Date*. On indique que dans la table [rv], la colonne [JOUR] (ligne 12) est de type date (sans heure),
- lignes 16-18 : qualifient la relation de clé étrangère qu'a la table [rv] vers la table [creneaux],

- lignes 20-22 : qualifient la relation de clé étrangère qu'a la table [rv] vers la table [clients].

La génération automatique des entités JPA nous permet d'obtenir une base de travail. Parfois elle est suffisante, parfois pas. C'est le cas ici :

- il faut ajouter l'annotation **@Version** aux différents champs **version** des entités,
- il faut écrire des méthodes **toString** plus explicites que celles générées,
- les entités [Medecin] et [Client] sont analogues. On va les faire dériver d'une classe [**Personne**],
- on va supprimer les relations **@OneToMany** inverses des relations **@ManyToOne**. Elles ne sont pas indispensables et elles amènent des complications de programmation,
- on supprime la validation **@NotNull** sur les clés primaires. Lorsqu'on persiste une entité JPA avec MySQL, l'entité au départ a une clé primaire *null*. Ce n'est qu'après persistance dans la base, que la clé primaire de l'élément persisté a une valeur.

Avec ces spécifications, les différentes classes deviennent les suivantes :

La classe **Personne** est utilisée pour représenter les médecins et les clients :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
4.  import javax.persistence.*;
5.  import javax.validation.constraints.NotNull;
6.  import javax.validation.constraints.Size;
7.
8.  @MappedSuperclass
9.  public class Personne implements Serializable {
10.   private static final long serialVersionUID = 1L;
11.   @Id
12.   @GeneratedValue(strategy = GenerationType.IDENTITY)
13.   @Column(name = "ID")
14.   private Long id;
15.
16.   @Basic(optional = false)
17.   @Size(min = 1, max = 5)
18.   @Column(name = "TITRE")
19.   private String titre;
20.
21.   @Basic(optional = false)
22.   @NotNull
23.   @Size(min = 1, max = 30)
24.   @Column(name = "NOM")
25.   private String nom;
26.
27.   @Basic(optional = false)
28.   @NotNull
29.   @Column(name = "VERSION")
30.   @Version
31.   private int version;
32.
33.   @Basic(optional = false)
34.   @NotNull
35.   @Size(min = 1, max = 30)
36.   @Column(name = "PRENOM")
37.   private String prenom;
38. // constructeurs
39. ...
40.
41. // getters et setters
42.   ...
43.
44.   @Override
45.   public String toString() {
46.     return String.format("[%s,%s,%s,%s,%s]", id, version, titre, prenom, nom);
47.   }
48.
49. }
```

- ligne 8 : on notera que la classe [Personne] n'est pas elle-même une entité (@Entity). Elle va être la classe parent d'entités. L'annotation **@MappedSuperClass** désigne cette situation.

L'entité [**Client**] encapsule les lignes de la table [clients]. Elle dérive de la classe [Personne] précédente :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
4.  import javax.persistence.*;
5.
6.  @Entity
7.  @Table(name = "clients")
8.  public class Client extends Personne implements Serializable {
9.    private static final long serialVersionUID = 1L;
10.
11. // constructeurs
12. ...
13.
14.   @Override
15.   public int hashCode() {
16. ...
17.   }
18.
19.   @Override
20.   public boolean equals(Object object) {
21.   ...
22.   }
23.
24.   @Override
25.   public String toString() {
26.     return String.format("Client[%s,%s,%s,%s]", getId(), getTitre(), getPrenom(), getNom());
27.   }
28.
29. }
```

- ligne 6 : la classe [Client] est une entité Jpa,
- ligne 7 : elle est associée à la table [clients],
- ligne 8 : elle dérive de la classe [Personne].

L'entité [**Medecin**] qui encapsule les lignes de la table [medecins] suit le même modèle :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
4.  import javax.persistence.*;
5.
6.  @Entity
7.  @Table(name = "medecins")
8.  public class Medecin extends Personne implements Serializable {
9.    private static final long serialVersionUID = 1L;
10.
11.   // constructeurs
12. ...
13.
14.   @Override
15.   public int hashCode() {
16.     ...
17.   }
18.
19.   @Override
20.   public boolean equals(Object object) {
21.     ...
22.   }
23.
24.   @Override
25.   public String toString() {
26.     return String.format("Médecin[%s,%s,%s,%s]", getId(), getTitre(), getPrenom(), getNom());
27.   }
28.
29. }
```

L'entité [**Creneau**] encapsule les lignes de la table [creneaux] :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
```

```
4.  import java.util.List;
5.  import javax.persistence.*;
6.  import javax.validation.constraints.NotNull;
7.
8.  @Entity
9.  @Table(name = "creneaux")
10. public class Creneau implements Serializable {
11.
12.   private static final long serialVersionUID = 1L;
13.   @Id
14.   @GeneratedValue(strategy = GenerationType.IDENTITY)
15.   @Basic(optional = false)
16.   @Column(name = "ID")
17.   private Long id;
18.
19.   @Basic(optional = false)
20.   @NotNull
21.   @Column(name = "MDEBUT")
22.   private int mdebut;
23.
24.   @Basic(optional = false)
25.   @NotNull
26.   @Column(name = "HFIN")
27.   private int hfin;
28.
29.   @Basic(optional = false)
30.   @NotNull
31.   @Column(name = "HDEBUT")
32.   private int hdebut;
33.
34.   @Basic(optional = false)
35.   @NotNull
36.   @Column(name = "MFIN")
37.   private int mfin;
38.
39.   @Basic(optional = false)
40.   @NotNull
41.   @Column(name = "VERSION")
42.   @Version
43.   private int version;
44.
45.   @JoinColumn(name = "ID_MEDECIN", referencedColumnName = "ID")
46.   @ManyToOne(optional = false)
47.   private Medecin medecin;
48.
49.   // constructeurs
50.   ...
51.
52.   // getters et setters
53.   ...
54.
55.   @Override
56.   public int hashCode() {
57.     ...
58.   }
59.
60.   @Override
61.   public boolean equals(Object object) {
62.     // TODO: Warning - this method won't work in the case the id fields are not set
63.     ...
64.   }
65.
66.   @Override
67.   public String toString() {
68.     return String.format("Creneau [%s, %s, %s:%s, %s:%s,%s]", id, version, hdebut, mdebut, hfin, mfin,
    medecin);
69.   }
70. }
```

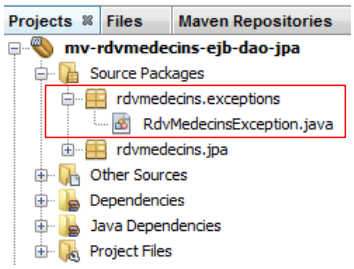
- les lignes 45-47 modélisent la relation "plusieurs à un" qui existe entre la table [creneaux] et la table [medecins] de la base de données : un médecin a plusieurs créneaux, un créneau appartient à un seul médecin.

L'entité [**Rv**] encapsule les lignes de la table [rv] :

```
1.  package rdvmedecins.jpa;
2.
3.  import java.io.Serializable;
4.  import java.util.Date;
5.  import javax.persistence.*;
6.  import javax.validation.constraints.NotNull;
7.
8.  @Entity
9.  @Table(name = "rv")
10. public class Rv implements Serializable {
11.
12.   private static final long serialVersionUID = 1L;
13.   @Id
14.   @GeneratedValue(strategy = GenerationType.IDENTITY)
15.   @Basic(optional = false)
16.   @Column(name = "ID")
17.   private Long id;
18.
19.   @Basic(optional = false)
20.   @NotNull
21.   @Column(name = "JOUR")
22.   @Temporal(TemporalType.DATE)
23.   private Date jour;
24.
25.   @JoinColumn(name = "ID_CRENEAU", referencedColumnName = "ID")
26.   @ManyToOne(optional = false)
27.   private Creneau creneau;
28.
29.   @JoinColumn(name = "ID_CLIENT", referencedColumnName = "ID")
30.   @ManyToOne(optional = false)
31.   private Client client;
32.
33.   // constructeurs
34. ...
35.
36.   // getters et setters
37. ...
38.
39.   @Override
40.   public int hashCode() {
41.     ...
42.   }
43.
44.   @Override
45.   public boolean equals(Object object) {
46.     ...
47.   }
48.
49.   @Override
50.   public String toString() {
51.     return String.format("Rv[%s, %s, %s]", id, creneau, client);
52.   }
53. }
```

- les lignes 29-31 modélisent la relation "plusieurs à un" qui existe entre la table [rv] et la table [clients] (un client peut apparaître dans plusieurs Rv) de la base de données et les lignes 25-27 la relation "plusieurs à un" qui existe entre la table [rv] et la table [creneaux] (un créneau peut apparaître dans plusieurs Rv).

3.4.3 La classe d'exception

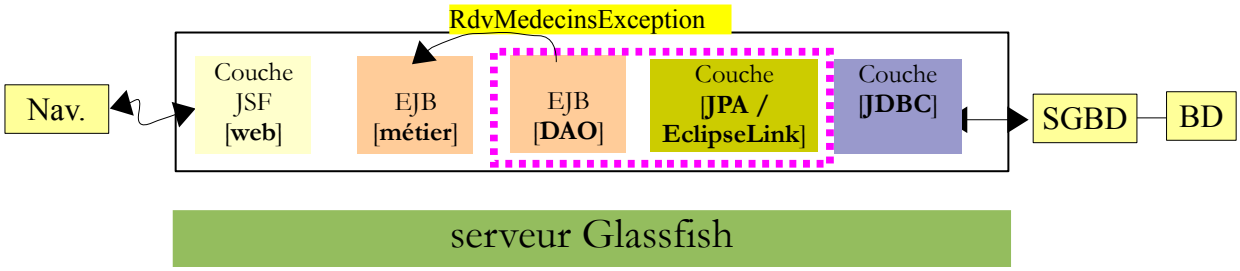


La classe d'exception [RdvMedecinsException] de l'application est la suivante :

```
1.  package rdvmedecins.exceptions;
2.
3.  import java.io.Serializable;
4.  import javax.ejb.ApplicationException;
5.
6.  @ApplicationException(rollback=true)
7.  public class RdvMedecinsException extends RuntimeException implements Serializable{
8.
9.    // champs privés
10.   private int code = 0;
11.
12.   // constructeurs
13.   public RdvMedecinsException() {
14.     super();
15.   }
16.
17.   public RdvMedecinsException(String message) {
18.     super(message);
19.   }
20.
21.   public RdvMedecinsException(String message, Throwable cause) {
22.     super(message, cause);
23.   }
24.
25.   public RdvMedecinsException(Throwable cause) {
26.     super(cause);
27.   }
28.
29.   public RdvMedecinsException(String message, int code) {
30.     super(message);
31.     setCode(code);
32.   }
33.
34.   public RdvMedecinsException(Throwable cause, int code) {
35.     super(cause);
36.     setCode(code);
37.   }
38.
39.   public RdvMedecinsException(String message, Throwable cause, int code) {
40.     super(message, cause);
41.     setCode(code);
42.   }
43.
44.   // getters - setters
45.   public int getCode() {
46.     return code;
47.   }
48.
49.   public void setCode(int code) {
50.     this.code = code;
51.   }
52. }
```

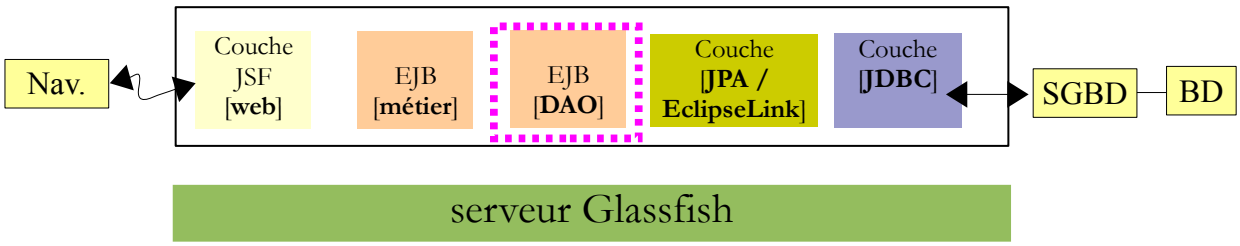
- ligne 7 : la classe dérive de la classe [RuntimeException]. Le compilateur ne force donc pas à la gérer avec des try / catch.
- ligne 6 : l'annotation **@ApplicationException** fait que l'exception ne sera pas "avalée" par une exception de type [EjbException].

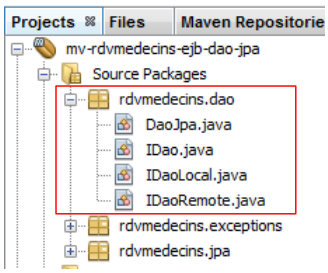
Pour comprendre l'annotation **@ApplicationException,** revenons à l'architecture utilisée côté serveur :



L'exception de type [RdvMedecinsException] sera lancée par les méthodes de l'EJB de la couche [DAO] à l'intérieur du conteneur EJB3 et interceptée par celui-ci. Sans l'annotation @**ApplicationException** le conteneur EJB3 encapsule l'exception survenue, dans une exception de type [EjbException] et relance celle-ci. On peut ne pas vouloir de cette encapsulation et laisser sortir du conteneur EJB3 une exception de type [RdvMedecinsException]. C'est ce que permet l'annotation @**ApplicationException**. Par ailleurs, l'attribut (**rollback=true**) de cette annotation indique au conteneur EJB3 que si l'exception de type [RdvMedecinsException] se produit à l'intérieur d'une méthode exécutée au sein d'une transaction avec un SGBD, celle-ci doit être annulée. En termes techniques, cela s'appelle faire un *rollback* de la transaction.

3.4.4 L'EJB de la couche [DAO]





L'interface java [IDao] de la couche [DAO] est la suivante :

```
1.  package rdvmedecins.dao;
2.
3.
4.  import java.util.Date;
5.  import java.util.List;
6.  import rdvmedecins.jpa.Client;
7.  import rdvmedecins.jpa.Creneau;
8.  import rdvmedecins.jpa.Medecin;
9.  import rdvmedecins.jpa.Rv;
10.
11. public interface IDao {
12.
13.   // liste des clients
14.   public List<Client> getAllClients();
15.   // liste des Médecins
16.   public List<Medecin> getAllMedecins();
17.   // liste des créneaux horaires d'un médecin
```

```
18.   public List<Creneau> getAllCreneaux(Medecin medecin);
19.   // liste des Rv d'un médecin, un jour donné
20.   public List<Rv> getRvMedecinJour(Medecin medecin, Date jour);
21.   // trouver un client identifié par son id
22.   public Client getClientById(Long id);
23.   // trouver un client identifié par son id
24.   public Medecin getMedecinById(Long id);
25.   // trouver un Rv identifié par son id
26.   public Rv getRvById(Long id);
27.   // trouver un créneau horaire identifié par son id
28.   public Creneau getCreneauById(Long id);
29.   // ajouter un RV
30.   public Rv ajouterRv(Date jour, Creneau creneau, Client client);
31.   // supprimer un RV
32.   public void supprimerRv(Rv rv);
33. }
```

Cette interface a été construite après avoir identifié les besoins de la couche [web] :

- ligne 14 : la liste des clients. Nous en aurons besoin pour alimenter la liste déroulante des clients,
- ligne 16 : la liste des médecins. Nous en aurons besoin pour alimenter la liste déroulante des médecins,
- ligne 18 : la liste des créneaux horaires d'un médecin. On en aura besoin pour afficher l'agenda du médecin pour un jour donné,
- ligne 20 : la liste des rendez-vous d'un médecin pour un jour donné. Combinée avec la méthode précédente, elle nous permettra d'afficher l'agenda du médecin pour un jour donné avec ses créneaux déjà réservés,
- ligne 22 : permet de retrouver un client à partir de son n°. La méthode nous permettra de retrouver un client à partir d'un choix dans la liste déroulante des clients,
- ligne 24 : idem pour les médecins,
- ligne 26 : retrouve un rendez-vous par son n°. Peut être utilisé lorsqu'on supprime un rendez-vous pour vérifier auparavant qu'il existe bien,
- ligne 28 : retrouve un créneau horaire à partir de son n°. Permet d'identifier le créneau qu'un utilisateur veut ajouter ou supprimer,
- ligne 30 : pour ajouter un rendez-vous,
- ligne 32 : pour supprimer un rendez-vous.

L'interface locale [IDaoLocal] de l'EJB se contente de dériver l'interface [IDao] précédente :

```
1.  package rdvmedecins.dao;
2.
3.  import javax.ejb.Local;
4.
5.  @Local
6.  public interface IDaoLocal extends IDao{
7.
8.  }
```

Il en est de même pour l'interface distante [IDaoRemote] :

```
1.  package rdvmedecins.dao;
2.
3.  import javax.ejb.Remote;
4.
5.  @Remote
6.  public interface IDaoRemote extends IDao{
7.
8.  }
```

L'EJB [DaoJpa] implémente les deux interfaces, **locale** et **distante** :

```
1.  package rdvmedecins.dao;
2.
3.  ...
4.
5.  @Singleton (mappedName="rdvmedecins.dao")
6.  @TransactionAttribute(TransactionAttributeType.REQUIRED)
7.  public class DaoJpa implements IDaoLocal, IDaoRemote, Serializable {
```

- la ligne 5 indique que l'EJB distant porte le nom "rdvmedecins.dao". Par ailleurs, l'annotation @**Singleton** (Java EE6) fait qu'il n'y aura qu'une instance de l'EJB créée. L'annotation @**Stateless** (Java EE5) définit un EJB qui peut être créé en de multiples instances pour alimenter un pool d'EJB,
- la ligne 6 indique que toutes les méthodes de l'EJB se déroulent au sein d'une transaction gérée par le conteneur EJB3,
- la ligne 7 montre que l'EJB implémente les interfaces **locale** et **distante** et est également sérialisable.

Le code complet de l'EJB est le suivant :

```
1.  package rdvmedecins.dao;
2.
3.  import java.io.Serializable;
4.  import java.util.Date;
5.  import java.util.List;
6.  import javax.ejb.Singleton;
7.  import javax.ejb.TransactionAttribute;
8.  import javax.ejb.TransactionAttributeType;
9.  import javax.persistence.EntityManager;
10. import javax.persistence.PersistenceContext;
11. import rdvmedecins.exceptions.RdvMedecinsException;
12. import rdvmedecins.jpa.Client;
13. import rdvmedecins.jpa.Creneau;
14. import rdvmedecins.jpa.Medecin;
15. import rdvmedecins.jpa.Rv;
16.
17. @Singleton (mappedName="rdvmedecins.dao")
18. @TransactionAttribute(TransactionAttributeType.REQUIRED)
19. public class DaoJpa implements IDaoLocal, IDaoRemote, Serializable {
20.
21.   @PersistenceContext
22.   private EntityManager em;
23.
24.   // liste des clients
25.   public List<Client> getAllClients() {
26.     try {
27.       return em.createQuery("select rc from Client rc").getResultList();
28.     } catch (Throwable th) {
29.       throw new RdvMedecinsException(th, 1);
30.     }
31.   }
32.
33.   // liste des médecins
34.   public List<Medecin> getAllMedecins() {
35.     try {
36.       return em.createQuery("select rm from Medecin rm").getResultList();
37.     } catch (Throwable th) {
38.       throw new RdvMedecinsException(th, 2);
39.     }
40.   }
41.
42.   // liste des créneaux horaires d'un médecin donné
43.   // medecin : le médecin
44.   public List<Creneau> getAllCreneaux(Medecin medecin) {
45.     try {
46.       return em.createQuery("select rc from Creneau rc join rc.medecin m where
    m.id=:idMedecin").setParameter("idMedecin", medecin.getId()).getResultList();
47.     } catch (Throwable th) {
48.       throw new RdvMedecinsException(th, 3);
49.     }
50.   }
51.
52.   // liste des Rv d'un médecin donné, un jour donné
53.   // medecin : le médecin
54.   // jour : le jour
55.   public List<Rv> getRvMedecinJour(Medecin medecin, Date jour) {
56.     try {
57.       return em.createQuery("select rv from Rv rv join rv.creneau c join c.idMedecin m where
    m.id=:idMedecin and rv.jour=:jour").setParameter("idMedecin", medecin.getId()).setParameter("jour",
    jour).getResultList();
58.     } catch (Throwable th) {
59.       throw new RdvMedecinsException(th, 3);
60.     }
61.   }
62.
63.   // ajout d'un Rv
```

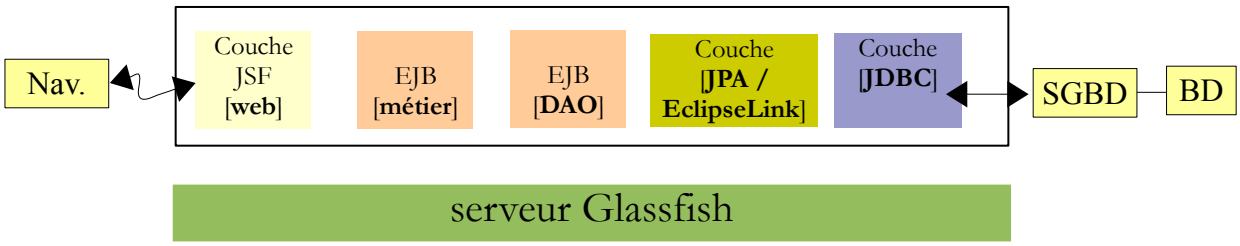
```
64.   // jour : jour du Rv
65.   // creneau : créneau horaire du Rv
66.   // client : client pour lequel est pris le Rv
67.   public Rv ajouterRv(Date jour, Creneau creneau, Client client) {
68.     try {
69.       Rv rv = new Rv(null, jour);
70.       rv.setClient(client);
71.       rv.setCreneau(creneau);
72.       em.persist(rv);
73.       return rv;
74.     } catch (Throwable th) {
75.       throw new RdvMedecinsException(th, 4);
76.     }
77.   }
78.
79.   // suppression d'un Rv
80.   // rv : le Rv supprimé
81.   public void supprimerRv(Rv rv) {
82.     try {
83.       em.remove(em.merge(rv));
84.     } catch (Throwable th) {
85.       throw new RdvMedecinsException(th, 5);
86.     }
87.   }
88.
89.   // récupérer un client donné
90.   public Client getClientById(Long id) {
91.     try {
92.       return (Client) em.find(Client.class, id);
93.     } catch (Throwable th) {
94.       throw new RdvMedecinsException(th, 6);
95.     }
96.   }
97.
98.   // récupérer un médecin donné
99.   public Medecin getMedecinById(Long id) {
100.    try {
101.      return (Medecin) em.find(Medecin.class, id);
102.    } catch (Throwable th) {
103.      throw new RdvMedecinsException(th, 6);
104.    }
105.  }
106.
107.  // récupérer un Rv donné
108.  public Rv getRvById(Long id) {
109.    try {
110.      return (Rv) em.find(Rv.class, id);
111.    } catch (Throwable th) {
112.      throw new RdvMedecinsException(th, 6);
113.    }
114.  }
115.
116.  // récupérer un créneau donné
117.  public Creneau getCreneauById(Long id) {
118.    try {
119.      return (Creneau) em.find(Creneau.class, id);
120.    } catch (Throwable th) {
121.      throw new RdvMedecinsException(th, 6);
122.    }
123.  }
124.}
```

- ligne 22 : l'objet **EntityManager** qui gère l'accès au contexte de persistance. A l'instanciation de la classe, ce champ sera initialisé par le conteneur EJB grâce à l'annotation @**PersistenceContext** de la ligne 21,
- ligne 27 : requête JPQL (Java Persistence Query Language) qui retourne toutes les lignes de la table [clients] sous la forme d'une liste d'objets [Client],
- ligne 36 : requête analogue pour les médecins,
- ligne 46 : une requête JPQL réalisant une jointure entre les tables [creneaux] et [medecins]. Elle est paramétrée par l'**id** du médecin,
- ligne 57 : une requête JPQL réalisant une jointure entre les tables [rv], [creneaux] et [medecins] et ayant deux paramètres : l'**id** du médecin et le **jour** du Rv,
- lignes 69-73 : création d'un Rv puis persistance de celui-ci en base de données,
- ligne 83 : suppression d'un Rv en base de données,

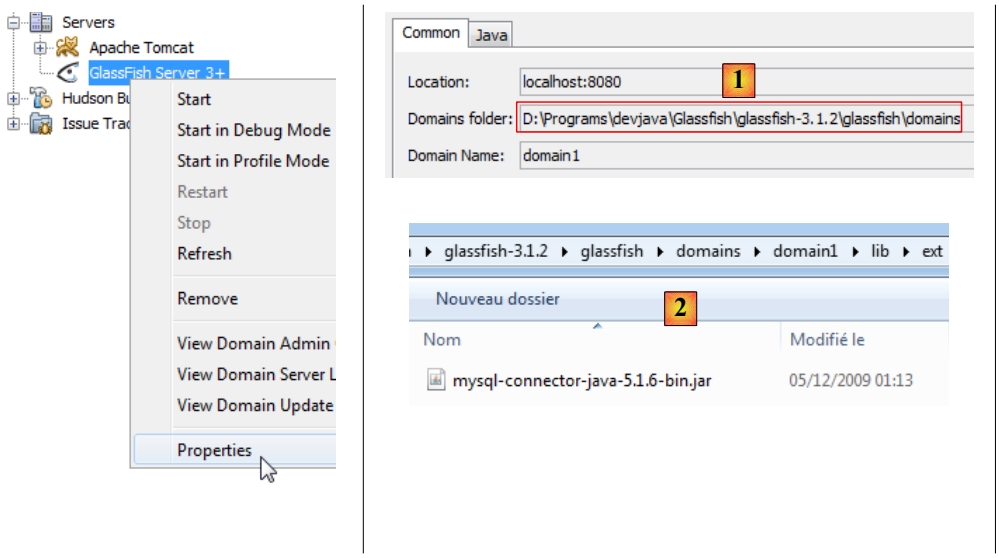
- ligne 92 : réalise un *select* sur la base de données pour trouver un client donné,
- ligne 101 : idem pour un médecin,
- ligne 110 : idem pour un Rv,
- ligne 119 : idem pour un créneau horaire,
- toutes les opérations avec le contexte de persistance **em** de la ligne 22 sont susceptibles de rencontrer un problème avec la base de données. Aussi sont-elles toutes entourées par un try / catch. L'éventuelle exception est encapsulée dans l'exception "maison" **RdvMedecinsException**.

3.4.5 Mise en place du pilote JDBC de MySQL

Dans l'architecture ci-dessous :



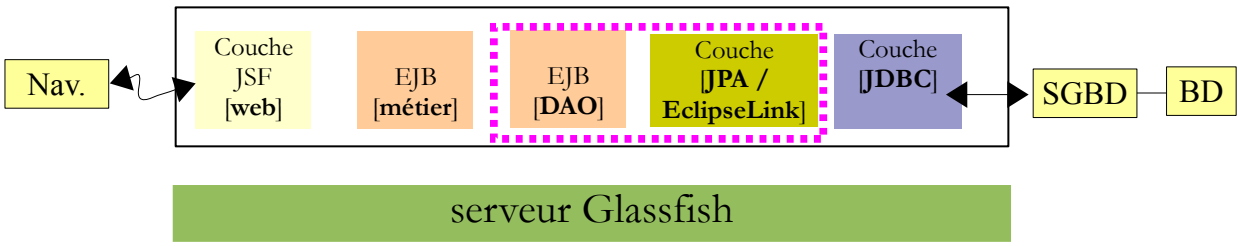
EclipseLink a besoin du pilote JDBC de MySQL. Il faut installer celui-ci dans les bibliothèques du serveur Glassfish dans le dossier <glassfish> /domains /domain1 /lib /ext où <glassfish> est le dossier d'installation du serveur Glassfish. On peut obtenir celui-ci de la façon suivante :



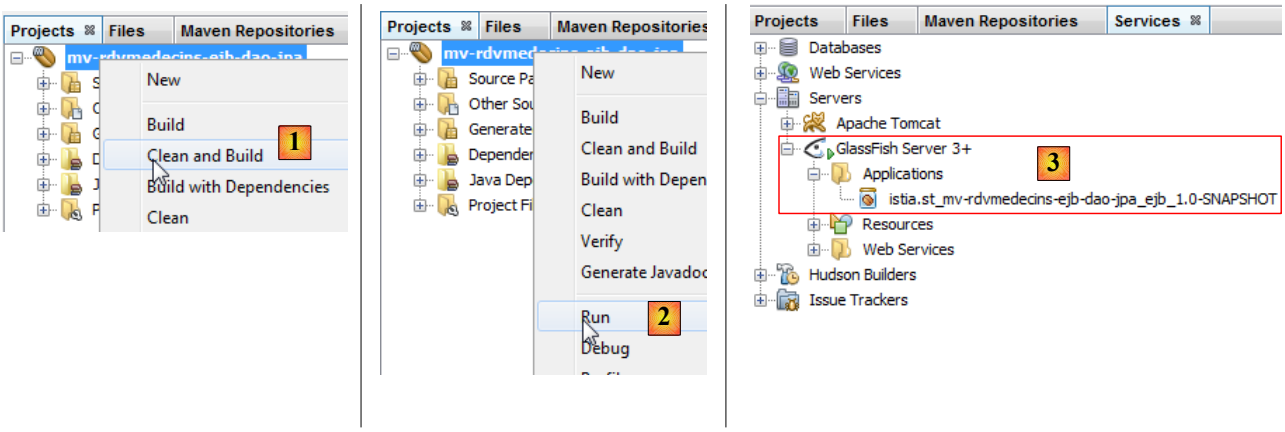
Le dossier où mettre le pilote JDBC de MySQL est <Domains folder>[1]/domain1/lib/ext [2]. Ce pilote est disponible à l'URL [http://www.mysql.fr/downloads/connector/j/]. Une fois celui-ci installé, il faut relancer le serveur Glassfish pour qu'il prenne en compte cette nouvelle bibliothèque.

3.4.6 Déploiement de l'EJB de la couche [DAO]

Revenons à l'architecture construite pour le moment :

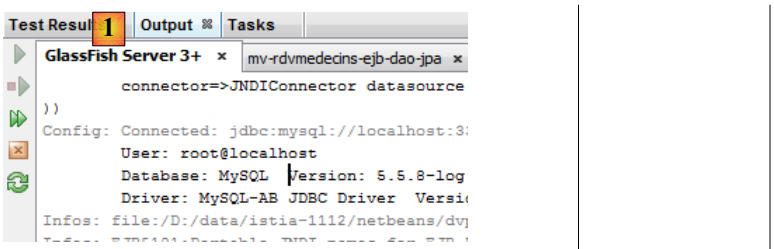


L'ensemble [web, métier, DAO, JPA] doit être déployé sur le serveur Glassfish. Nous le faisons :



- en [1], on construit le projet Maven,
- en [2], on l'exécute,
- en [3], il a été déployé sur le serveur Glassfish (onglet [Services])

On peut avoir la curiosité de regarder les logs de Glassfish :



En [1], les logs de Glassfish sont disponibles dans l'onglet [Output / Glassfish Server 3+]. Ce sont les suivants :

```
1.  Config: The access type for the persistent class [class rdvmedecins.jpa.Personne] is set to
    [FIELD].
2.  Config: The access type for the persistent class [class rdvmedecins.jpa.Client] is set to [FIELD].
3.  Config: The access type for the persistent class [class rdvmedecins.jpa.Rv] is set to [FIELD].
4.  Config: The target entity (reference) class for the many to one mapping element [field client] is
    being defaulted to: class rdvmedecins.jpa.Client.
5.  Config: The target entity (reference) class for the many to one mapping element [field creneau] is
    being defaulted to: class rdvmedecins.jpa.Creneau.
6.  Config: The access type for the persistent class [class rdvmedecins.jpa.Medecin] is set to
    [FIELD].
7.  Config: The access type for the persistent class [class rdvmedecins.jpa.Creneau] is set to
    [FIELD].
8.  Config: The target entity (reference) class for the many to one mapping element [field medecin] is
    being defaulted to: class rdvmedecins.jpa.Medecin.
9.  Config: The alias name for the entity class [class rdvmedecins.jpa.Client] is being defaulted to:
    Client.
10. Config: The alias name for the entity class [class rdvmedecins.jpa.Rv] is being defaulted to: Rv.
```

```
11. Config: The alias name for the entity class [class rdvmedecins.jpa.Medecin] is being defaulted to:
    Medecin.
12. Config: The alias name for the entity class [class rdvmedecins.jpa.Creneau] is being defaulted to:
    Creneau.
13. Infos: rdvmedecins.jpa.Creneau actually got transformed
14. Infos: rdvmedecins.jpa.Medecin actually got transformed
15. Infos: rdvmedecins.jpa.Personne actually got transformed
16. Infos: rdvmedecins.jpa.Client actually got transformed
17. Infos: rdvmedecins.jpa.Rv actually got transformed
18. Infos: EclipseLink, version: Eclipse Persistence Services - 2.3.2.v20111125-r10461
19. Précis: Detected database platform: org.eclipse.persistence.platform.database.MySQLPlatform
20. Config: connecting(DatabaseLogin(
21.    platform=>DatabasePlatform
22.    user name=> ""
23.    connector=>JNDIConnector datasource name=>null
24. ))
25. Config: Connected: jdbc:mysql://localhost:3306/dbrdvmedecins2
26.    User: root@localhost
27.    Database: MySQL  Version: 5.5.8-log
28.    Driver: MySQL-AB JDBC Driver  Version: mysql-connector-java-5.1.6 ( Revision: ${svn.Revision} )
29. Config: connecting(DatabaseLogin(
30.    platform=>MySQLPlatform
31.    user name=> ""
32.    connector=>JNDIConnector datasource name=>null
33. ))
34. Config: Connected: jdbc:mysql://localhost:3306/dbrdvmedecins2
35.    User: root@localhost
36.    Database: MySQL  Version: 5.5.8-log
37.    Driver: MySQL-AB JDBC Driver  Version: mysql-connector-java-5.1.6 ( Revision: ${svn.Revision} )
38. Infos: file:/D:/data/istia-1112/netbeans/dvp/jsf2-pf-pfm/maven/netbeans/rdvmedecins-jsf2-ejb/mv-
    rdvmedecins-ejb-dao-jpa/target/classes/_dbrdvmedecins2-PU login successful
39. Infos: EJB5181:Portable JNDI names for EJB DaoJpa: [java:global/istia.st_mv-rdvmedecins-ejb-dao-
    jpa_ejb_1.0-SNAPSHOT/DaoJpa!rdvmedecins.dao.IDaoLocal, java:global/istia.st_mv-rdvmedecins-ejb-
    dao-jpa_ejb_1.0-SNAPSHOT/DaoJpa!rdvmedecins.dao.IDaoRemote]
40. Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB DaoJpa:
    [rdvmedecins.dao#rdvmedecins.dao.IDaoRemote, rdvmedecins.dao]
41. Infos: istia.st_mv-rdvmedecins-ejb-dao-jpa_ejb_1.0-SNAPSHOT a été déployé en 270 ms.
```

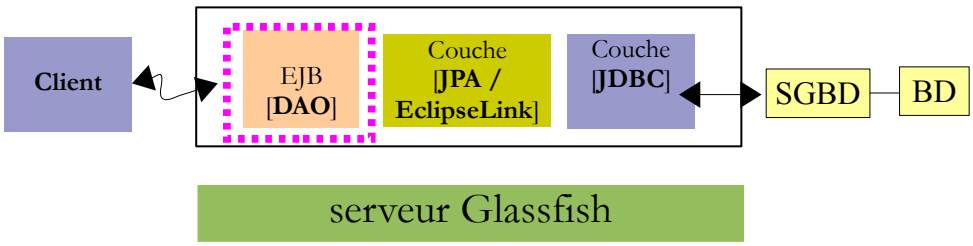
Les lignes identifiées par [Config], [Précis] sont les logs d'EclipseLink, celles identifiées par [Infos] proviennent de Glassfish.

- lignes 1-12 : EclipseLink traite les entités JPA qu'il a découvertes,
- lignes 13-17 : infos traduisant le fait que le traitement des entités JPA s'est fait normalement,
- ligne 18 : EclipseLink se signale,
- ligne 19 : EclipseLink reconnaît qu'il a affaire au SGBD MySQL,
- lignes 20-24 : EclipseLink essaie de se connecter à la BD,
- lignes 25-28 : il a réussi,
- lignes 29-33 : il essaie de se reconnecter cette fois en utilisant spécifiquement une plate-forme MySQL (ligne 30),
- lignes 34-37 : là également réussite,
- ligne 38 : confirmation que l'unité de persistance [dbrdvmedecins-PU] a pu être instanciée,
- ligne 39 : les noms portables des interfaces distante et locale de l'EJB [DaoJpa], " portable " voulant dire reconnues par tous les serveurs d'application Java EE 6,
- ligne 40 : les noms des interfaces distante et locale de l'EJB [DaoJpa], spécifiques à Glassfish. Nous utiliserons dans le test à venir, le nom " rdvmedecins.dao ".

Les lignes 39 et 40 sont importantes. Lorsqu'on écrit le client d'un EJB sur Glassfish, il est nécessaire de les connaître.

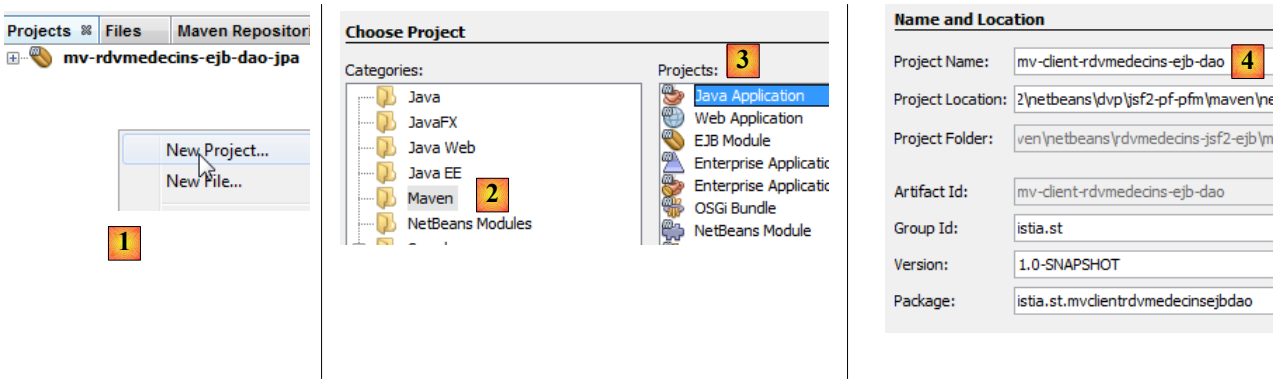
3.4.7 Tests de l'EJB de la couche [DAO]

Maintenant que l'EJB de la couche [DAO] de notre application a été déployée, nous pouvons le tester. Nous allons le faire dans le cadre d'une application client / serveur :

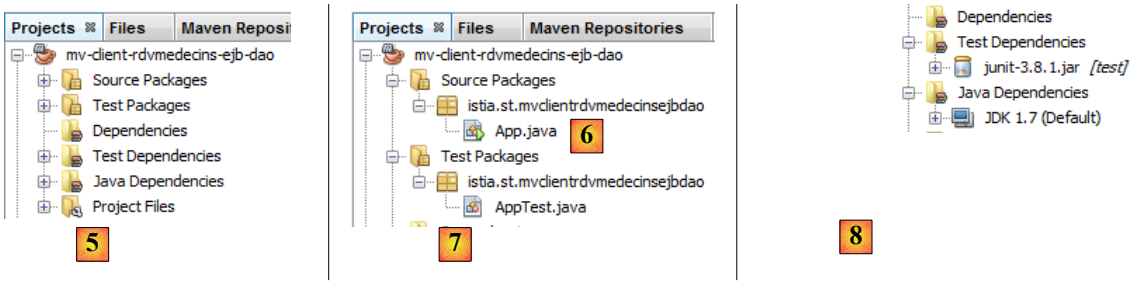


Le client va tester l'interface distante de l'EJB [DAO] déployé sur le serveur Glassfish.

Nous commençons par créer un nouveau projet Maven :



- en [1], nous créons un nouveau projet,
- en [2,3], nous créons un projet Maven de type [Java Application],
- en [4], nous lui donnons un nom et nous le mettons dans le même dossier que l'EJB [DAO],

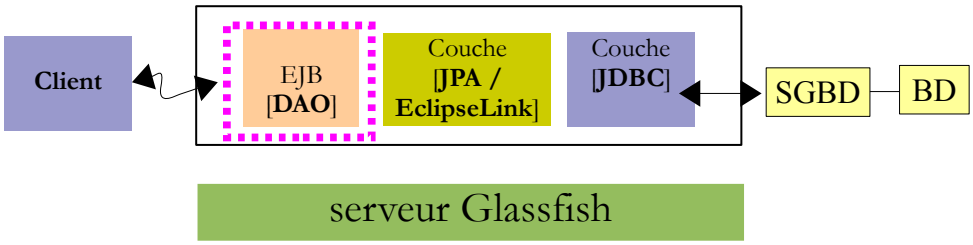


- en [5], le projet généré,
- en [6], une classe [App.java] a été générée. On la supprimera,
- en [7], une branche [Source Packages] a été générée. Nous ne l'avions pas encore rencontrée. On peut mettre des tests JUnit dans cette branche. Nous le ferons. Nous ne garderons pas la classe de test [AppTest] générée,
- en [8], les dépendances du projet Maven. La branche [Dependencies] est vide. Nous serons amenés à y mettre de nouvelles dépendances. La branche [Test Dependencies] rassemble les dépendances nécessaires aux tests. Ici, la bibliothèque utilisée est celle du framework JUnit 3.8. Nous serons amenés à la changer.

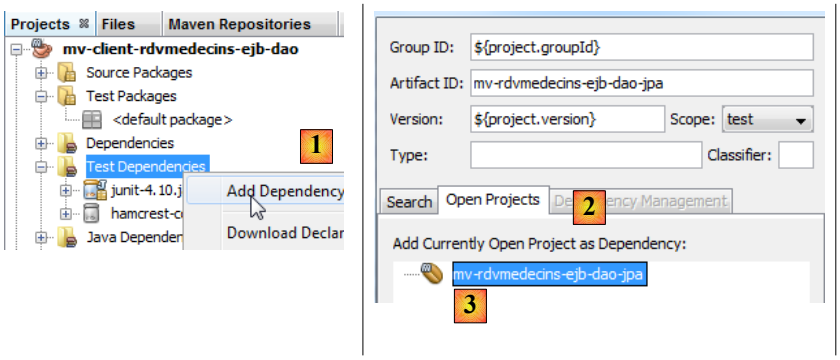
Le projet évolue de la façon suivante :

- en [1], le projet où les deux classes générées ont été supprimées ainsi que la dépendance JUnit.

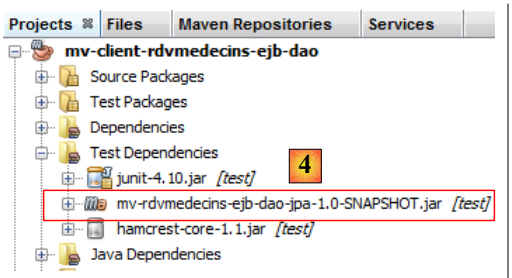
Revenons à l'architecture client / serveur qui va servir au test :



Le client a besoin de connaître l'interface distante offerte par l'EJB [DAO]. Par ailleurs, il va échanger avec l'EJB des entités JPA. Il a donc besoin de la définition de celles-ci. Afin que le projet de test de l'EJB ait accès à ces informations, on va ajouter le projet de l'EJB [DAO] comme dépendance au projet :

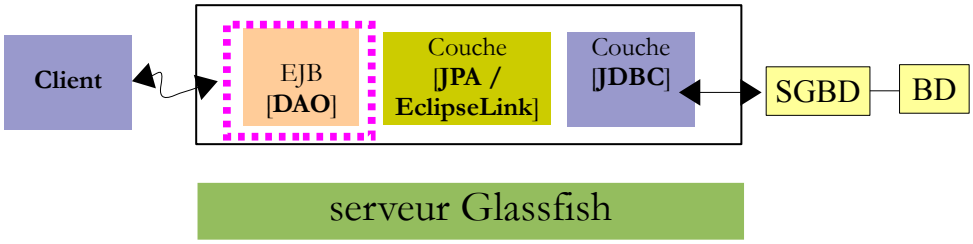


- en [1], on ajoute une dépendance à la branche [Test Dependencies],
- en [2], on choisit l'onglet [Open Projects],
- en [3], on choisit le projet Maven de l'EJB [DAO],

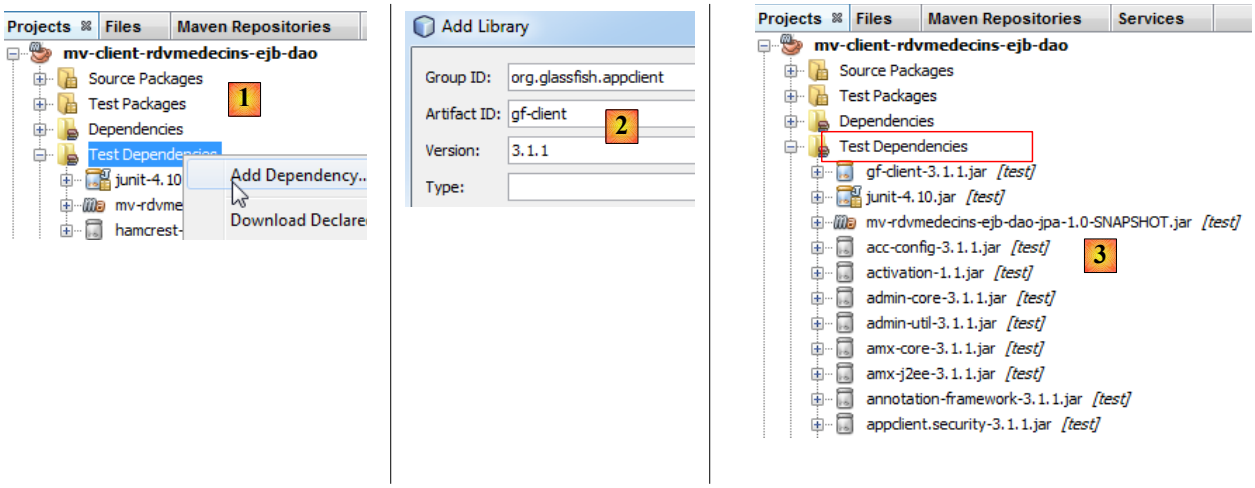


- en [4], la dépendance ajoutée.

Revenons à l'architecture client / serveur du test :

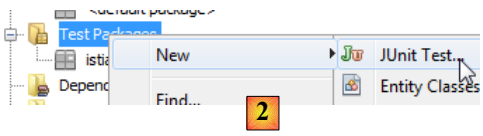


A l'exécution, le client et le serveur communiquent via le réseau TCP-IP. Nous n'allons pas programmer ces échanges. Pour chaque serveur d'applications, il existe une bibliothèque à intégrer aux dépendances du client. Celle pour Glassfish s'appelle [gf-client]. Nous l'ajoutons :

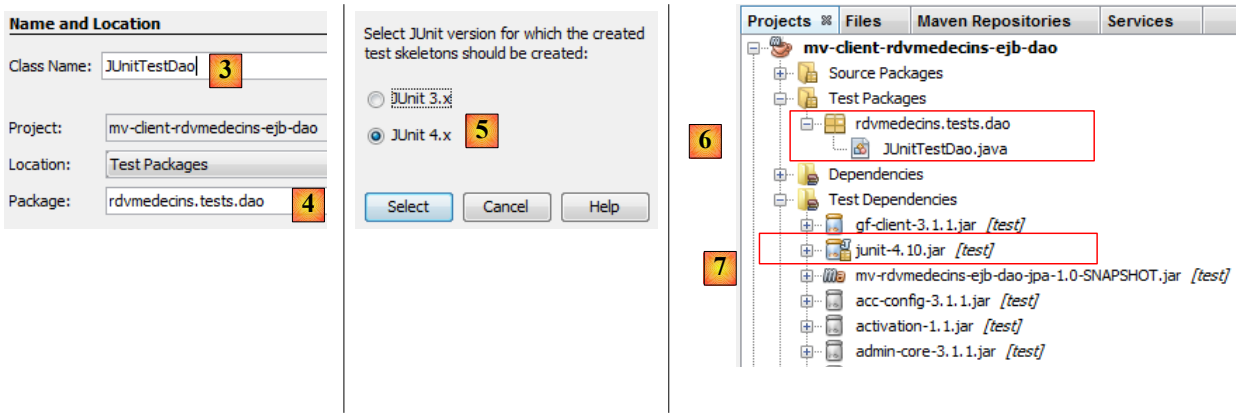


- en [1], on ajoute une dépendance,
- en [2], on donne les caractéristiques de l'artifact désiré,
- en [3], de très nombreuses dépendances sont ajoutées. Maven va les télécharger. Ceci peut prendre plusieurs minutes. Elles sont ensuites stockées dans le dépôt local de Maven.

Nous pouvons désormais créer le test JUnit :



- en [2], on clique droit sur [Test Packages] pour créer un nouveau test JUnit,



- en [3], on donne un nom à la classe de test ainsi qu'un paquetage pour celle-ci [4],
- en [5], on choisit le framework JUnit 4.x,
- en [6], la classe de test générée,
- en [7], les nouvelles dépendances du projet Maven.

Le fichier [pom.xml] est alors 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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-client-rdvmedecins-ejb-dao</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>mv-client-rdvmedecins-ejb-dao</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <repositories>
14.     <repository>
15.       <url>http://download.eclipse.org/rt/eclipselink/maven.repo/</url>
16.       <id>eclipselink</id>
17.       <layout>default</layout>
18.       <name>Repository for library Library[eclipselink]</name>
19.     </repository>
20.     <repository>
21.       <url>http://repo1.maven.org/maven2/</url>
22.       <id>junit_4</id>
23.       <layout>default</layout>
24.       <name>Repository for library Library[junit_4]</name>
25.     </repository>
26.   </repositories>
27.
28.   <properties>
29.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
30.   </properties>
31.
32.   <dependencies>
33.     <dependency>
34.       <groupId>junit</groupId>
35.       <artifactId>junit</artifactId>
36.       <version>4.10</version>
37.       <scope>test</scope>
38.     </dependency>
39.     <dependency>
40.       <groupId>${project.groupId}</groupId>
41.       <artifactId>mv-rdvmedecins-ejb-dao-jpa</artifactId>
42.       <version>${project.version}</version>
43.       <scope>test</scope>
```

```
44.     </dependency>
45.     <dependency>
46.       <groupId>org.glassfish.appclient</groupId>
47.       <artifactId>gf-client</artifactId>
48.       <version>3.1.1</version>
49.       <scope>test</scope>
50.     </dependency>
51.   </dependencies>
52. </project>
```

On notera :

- lignes 32-51, les dépendances du projet,
- lignes 13-26, deux dépôts Maven ont été définis, l'un pour EclipseLink (lignes 14-19), l'autre pour JUnit4 (lignes 20-25).

La classe de test sera la suivante :

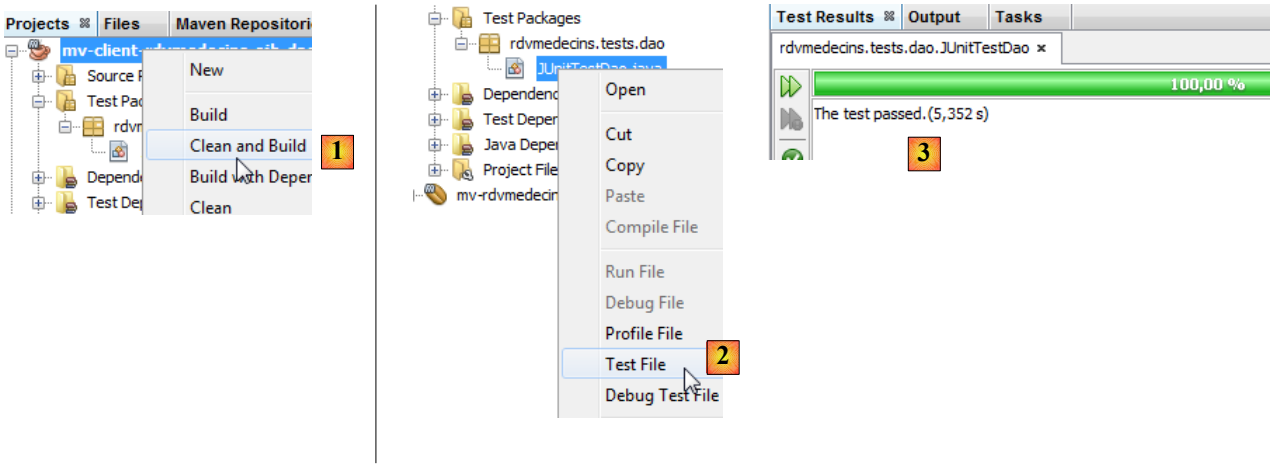
```
1.  package rdvmedecins.tests.dao;
2.
3.  import java.util.Date;
4.  import java.util.List;
5.  import javax.naming.InitialContext;
6.  import javax.naming.NamingException;
7.  import junit.framework.Assert;
8.  import org.junit.BeforeClass;
9.  import org.junit.Test;
10. import rdvmedecins.dao.IDaoRemote;
11. import rdvmedecins.jpa.Client;
12. import rdvmedecins.jpa.Creneau;
13. import rdvmedecins.jpa.Medecin;
14. import rdvmedecins.jpa.Rv;
15.
16. public class JUnitTestDao {
17.
18.   // couche [dao] testée
19.   private static IDaoRemote dao;
20.   // date du jour
21.   Date jour = new Date();
22.
23.   @BeforeClass
24.   public static void init() throws NamingException {
25.     // initialisation environnement JNDI
26.     InitialContext initialContext = new InitialContext();
27.     // instanciation couche dao
28.     dao = (IDaoRemote) initialContext.lookup("rdvmedecins.dao");
29.   }
30.
31.   @Test
32.   public void test1() {
33.     // affichage clients
34.     List<Client> clients =dao.getAllClients();
35.     display("Liste des clients :", clients);
36.     // affichage médecins
37.     List<Medecin> medecins =dao.getAllMedecins();
38.     display("Liste des médecins :", medecins);
39.     // affichage créneaux d'un médecin
40.     Medecin medecin = medecins.get(0);
41.     List<Creneau> creneaux = dao.getAllCreneaux(medecin);
42.     display(String.format("Liste des créneaux du médecin %s", medecin), creneaux);
43.     // liste des Rv d'un médecin, un jour donné
44.     display(String.format("Liste des créneaux du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, jour));
45.     // ajouter un RV
46.     Rv rv = null;
47.     Creneau creneau = creneaux.get(2);
48.     Client client = clients.get(0);
49.     System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s", jour,
    creneau, client));
50.     rv = dao.ajouterRv(jour, creneau, client);
51.     System.out.println("Rv ajouté");
52.     display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, jour));
53.     // ajouter un RV dans le même créneau du même jour
54.     // doit provoquer une exception
```

```
55.     System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s", jour,
    creneau, client));
56.     Boolean erreur = false;
57.     try {
58.       rv = dao.ajouterRv(jour, creneau, client);
59.       System.out.println("Rv ajouté");
60.     } catch (Exception ex) {
61.       Throwable th = ex;
62.       while (th != null) {
63.         System.out.println(ex.getMessage());
64.         th = th.getCause();
65.       }
66.       // on note l'erreur
67.       erreur=true;
68.     }
69.     // on vérifie qu'il y a eu une erreur
70.     Assert.assertTrue(erreur);
71.     // liste des RV
72.     display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, jour));
73.     // supprimer un RV
74.     System.out.println("Suppression du Rv ajouté");
75.     dao.supprimerRv(rv);
76.     System.out.println("Rv supprimé");
77.     display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, jour));
78.   }
79.
80.   // méthode utilitaire - affiche les éléments d'une collection
81.   private static void display(String message, List elements) {
82.     System.out.println(message);
83.     for (Object element : elements) {
84.       System.out.println(element);
85.     }
86.   }
87. }
```

- lignes 23-29 : la méthode taguée @**BeforeClass** est exécutée avant toutes les autres. Ici, on crée une référence sur l'interface distante de l'EJB [DaoJpa]. On se rappelle qu'on lui avait donné le nom JNDI "**rdvmedecins.dao**",
- lignes 34-35 : affichent la liste des clients,
- lignes 37-38 : affichent la liste des médecins,
- lignes 40-42 : affichent les créneaux horaires du premier médecin,
- ligne 44 : affichent les rendez-vous du premier médecin pour le jour de la ligne 21,
- lignes 46-51 : ajoutent un rendez-vous au premier médecin, pour son créneau n° 2 et le jour de la ligne 21,
- ligne 52 : affichent pour vérification les rendez-vous du premier médecin pour le jour de la ligne 21. Il doit y en avoir au moins un, celui qu'on vient d'ajouter,
- lignes 55-70 : on rajoute le même rendez-vous. Comme la table [RV] a une contrainte d'unicité, cet ajout doit provoquer une exception. On s'en assure ligne 70,
- ligne 72 : affichent pour vérification les rendez-vous du premier médecin pour le jour de la ligne 21. Celui qu'on voulait ajouter ne doit pas y être,
- lignes 74-76 : on supprime l'unique rendez-vous qui a été ajouté,
- ligne 77 : affichent pour vérification les rendez-vous du premier médecin pour le jour de la ligne 21. Celui qu'on vient de supprimer ne doit pas y être.

Ce test est un faux test JUnit. On n'y trouve qu'une assertion (ligne 70). C'est un test visuel avec les défauts qui vont avec.

Si tout va bien, les tests doivent passer :



- en [1], on construit le projet de test,
- en [2], on exécute le test,
- en [3], le test a été réussi.

Regardons de plus près les affichages du test :

```
1.  Liste des clients :
2.  Client[1,Mr,Jules,MARTIN]
3.  Client[2,Mme,Christine,GERMAN]
4.  Client[3,Mr,Jules,JACQUARD]
5.  Client[4,Melle,Brigitte,BISTROU]
6.  Liste des médecins :
7.  Médecin[1,Mme,Marie,PELISSIER]
8.  Médecin[2,Mr,Jacques,BROMARD]
9.  Médecin[3,Mr,Philippe,JANDOT]
10. Médecin[4,Melle,Justine,JACQUEMOT]
11. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER]
12. Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]]
13. Creneau [2, 1, 8:20, 8:40,Médecin[1,Mme,Marie,PELISSIER]]
14. Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]]
15. Creneau [4, 1, 9:0, 9:20,Médecin[1,Mme,Marie,PELISSIER]]
16. Creneau [5, 1, 9:20, 9:40,Médecin[1,Mme,Marie,PELISSIER]]
17. Creneau [6, 1, 9:40, 10:0,Médecin[1,Mme,Marie,PELISSIER]]
18. Creneau [7, 1, 10:0, 10:20,Médecin[1,Mme,Marie,PELISSIER]]
19. Creneau [8, 1, 10:20, 10:40,Médecin[1,Mme,Marie,PELISSIER]]
20. Creneau [9, 1, 10:40, 11:0,Médecin[1,Mme,Marie,PELISSIER]]
21. Creneau [10, 1, 11:0, 11:20,Médecin[1,Mme,Marie,PELISSIER]]
22. Creneau [11, 1, 11:20, 11:40,Médecin[1,Mme,Marie,PELISSIER]]
23. Creneau [12, 1, 11:40, 12:0,Médecin[1,Mme,Marie,PELISSIER]]
24. Creneau [13, 1, 14:0, 14:20,Médecin[1,Mme,Marie,PELISSIER]]
25. Creneau [14, 1, 14:20, 14:40,Médecin[1,Mme,Marie,PELISSIER]]
26. Creneau [15, 1, 14:40, 15:0,Médecin[1,Mme,Marie,PELISSIER]]
27. Creneau [16, 1, 15:0, 15:20,Médecin[1,Mme,Marie,PELISSIER]]
28. Creneau [17, 1, 15:20, 15:40,Médecin[1,Mme,Marie,PELISSIER]]
29. Creneau [18, 1, 15:40, 16:0,Médecin[1,Mme,Marie,PELISSIER]]
30. Creneau [19, 1, 16:0, 16:20,Médecin[1,Mme,Marie,PELISSIER]]
31. Creneau [20, 1, 16:20, 16:40,Médecin[1,Mme,Marie,PELISSIER]]
32. Creneau [21, 1, 16:40, 17:0,Médecin[1,Mme,Marie,PELISSIER]]
33. Creneau [22, 1, 17:0, 17:20,Médecin[1,Mme,Marie,PELISSIER]]
34. Creneau [23, 1, 17:20, 17:40,Médecin[1,Mme,Marie,PELISSIER]]
35. Creneau [24, 1, 17:40, 18:0,Médecin[1,Mme,Marie,PELISSIER]]
36. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 15:34:15 CEST 2012]
37. Ajout d'un Rv le [Wed May 23 15:34:15 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
38. Rv ajouté
39. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 15:34:15 CEST 2012]
40. Rv[242, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
41. Ajout d'un Rv le [Wed May 23 15:34:15 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
42. java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
43.    org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
44. java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
45.    org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
46. java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
```

```
47.    org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
48. java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
49.    org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
50. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 15:34:15 CEST 2012]
51. Rv[242, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
52. Suppression du Rv ajouté
53. Rv supprimé
54. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 15:34:15 CEST 2012]
```

Le lecteur est invité à lire ces logs en même temps que le code qui les a produits. On va s'attarder à l'exception qui s'est produite à l'ajout d'un rendez-vous déjà existant, lignes 41-49. La pile d'exceptions est reproduite lignes 42-48. Elle est inattendue. Revenons au code de la méthode d'ajout d'un rendez-vous :

```
1.    // ajout d'un Rv
2.    // jour : jour du Rv
3.    // creneau : créneau horaire du Rv
4.    // client : client pour lequel est pris le Rv
5.    public Rv ajouterRv(Date jour, Creneau creneau, Client client) {
6.      try {
7.        Rv rv = new Rv(null, jour);
8.        rv.setClient(client);
9.        rv.setCreneau(creneau);
10.       System.out.println(String.format("avant persist : %s",rv));
11.       em.persist(rv);
12.       System.out.println(String.format("après persist : %s",rv));
13.       return rv;
14.     } catch (Throwable th) {
15.       throw new RdvMedecinsException(th, 4);
16.     }
17. }
```

Regardons les logs de Glassfish lors de l'ajout des deux rendez-vous :

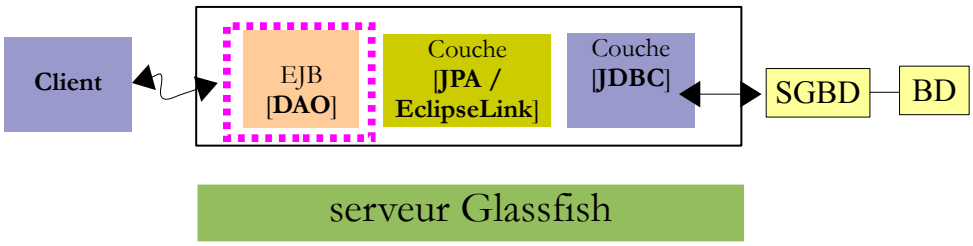
```
1.  ...
2.  Infos: avant persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
3.  Infos: après persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
4.  Précis: INSERT INTO rv (JOUR, ID_CLIENT, ID_CRENEAU) VALUES (?, ?, ?)
5.     bind => [3 parameters bound]
6.  Précis: SELECT LAST_INSERT_ID()
7.  Précis: SELECT t1.ID, t1.JOUR, t1.ID_CLIENT, t1.ID_CRENEAU FROM creneaux t0, rv t1 WHERE
    (((t0.ID_MEDECIN = ?) AND (t1.JOUR = ?)) AND (t0.ID = t1.ID_CRENEAU))
8.     bind => [2 parameters bound]
9.  Infos: avant persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
10. Infos: après persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
11. Précis: INSERT INTO rv (JOUR, ID_CLIENT, ID_CRENEAU) VALUES (?, ?, ?)
12.    bind => [3 parameters bound]
13. Précis: SELECT 1
14. Avertissement: Local Exception Stack:
15. Exception [EclipseLink-4002] (Eclipse Persistence Services - 2.3.2.v20111125-r10461):
    org.eclipse.persistence.exceptions.DatabaseException
16. Internal Exception: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
    Duplicate entry '2012-05-23-3' for key 'UNQ1_RV'
17. Error Code: 1062
18. ...
```

- ligne 2 : avant le premier *persist*,
- ligne 3 : après le premier *persist*,
- ligne 4 : l'ordre INSERT qui va être exécuté. On notera qu'il n'a pas lieu en même temps que l'opération *persist*. Si c'était le cas, ce log serait apparu avant la ligne 2. L'opération INSERT a alors normalement lieu à la fin de la transaction dans laquelle s'exécute la méthode,
- ligne 6 : EclipseLink demande à MySQL quelle est la dernière clé primaire utilisée. Il va obtenir la clé primaire du rendez-vous ajouté. Cette valeur va alimenter le champ **id** de l'entité [Rv] persistée,
- lignes 7-8 : la requête SELECT qui va afficher les rendez-vous du médecin,
- lignes 9-10 : les affichages écran du second *persist*,

- lignes 11-12 : l'ordre INSERT qui va être exécuté. Il doit provoquer une exception. Celle-ci apparaît aux lignes 15-16 et elle est claire. Elle est lancée initialement par le pilote JDBC de MySQL pour violation de la contrainte d'unicité des rendez-vous. On en déduit qu'on devrait voir ces exceptions dans les logs du test JUnit. Or ce n'est pas le cas :

```
1.  java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
2.     org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
3.  java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
4.     org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
5.  java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
6.     org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
7.  java.rmi.MarshalException: CORBA MARSHAL 1330446347 Maybe; nested exception is:
8.     org.omg.CORBA.MARSHAL: Avertissement: IOP00810011: Exception from readValue on ValueHandler in
    CDRInputStream  vmcid: OMG  minor code: 11 completed: Maybe
```

Rappelons l'architecture client / serveur du test :

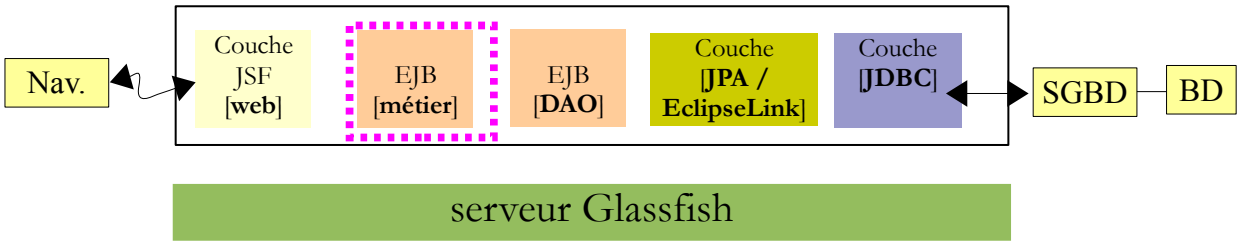


Lorsque l'EJB [DAO] lance une exception, celle-ci doit être sérialisée pour parvenir au client. C'est probablement cette opération qui a échoué pour une raison que je n'ai pas comprise. Comme notre application complète ne fonctionnera pas en client / serveur, nous pouvons ignorer ce problème.

Maintenant que l'EJB de la couche [DAO] est opérationnel, on peut passer à l'EJB de la couche [métier].

3.5 La couche [métier]

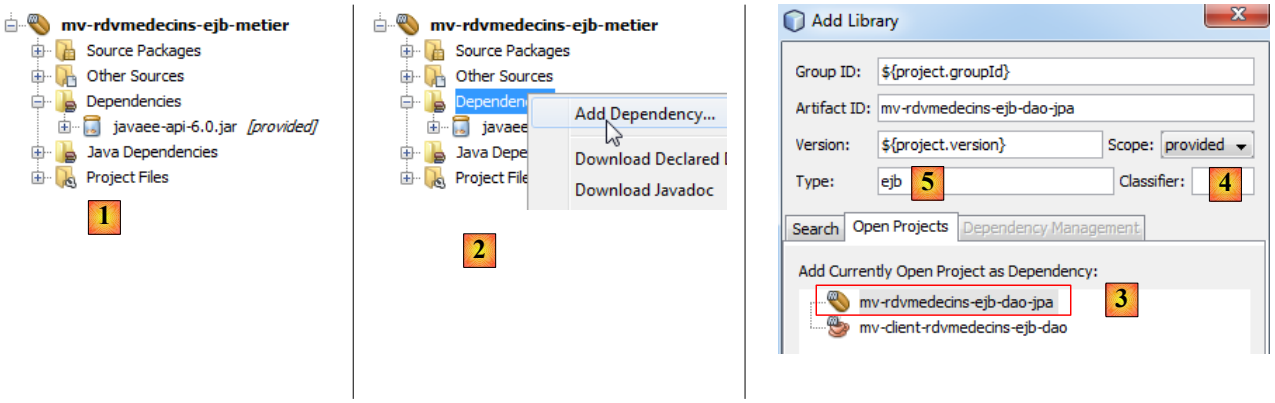
Revenons à l'architecture de l'application en cours de construction :



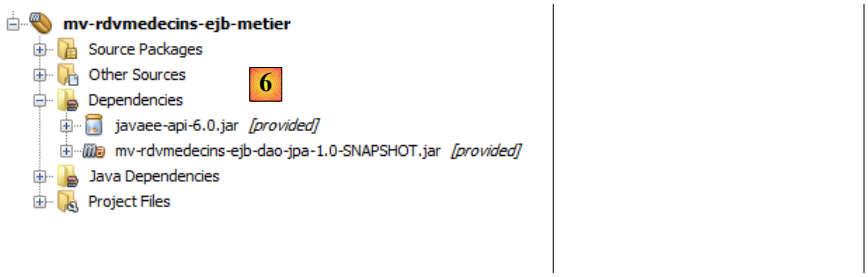
Nous allons construire un nouveau projet Maven pour l'EJB [métier]. Comme on le voit ci-dessus, il aura une dépendance sur le projet Maven qui a été construit pour les couches [DAO] et [JPA].

3.5.1 Le projet Netbeans

Nous construisons un nouveau projet Maven de type EJB. Pour cela, il suffit de suivre la procédure déjà utilisée et décrite page 163.

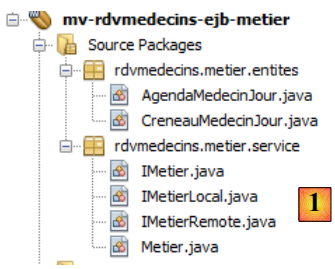


- en [1], le projet Maven de la couche [métier],
- en [2], on ajoute une dépendance,
- en [3], on choisit le projet Maven des couches [DAO] et [JPA],
- en [4], on choisit la portée [provided]. On rappelle que cela veut dire qu'on en a besoin pour la compilation mais pas pour l'exécution du projet. En effet, l'EJB de la couche [métier] va être déployé sur le serveur Glassfish avec l'EJB des couches [DAO] et [JPA]. Donc lorsqu'il s'exécutera, l'EJB des couches [DAO] et [JPA] sera déjà présent,



- en [6], le nouveau projet avec sa dépendance.

Présentons maintenant les codes source de la couche [métier] :

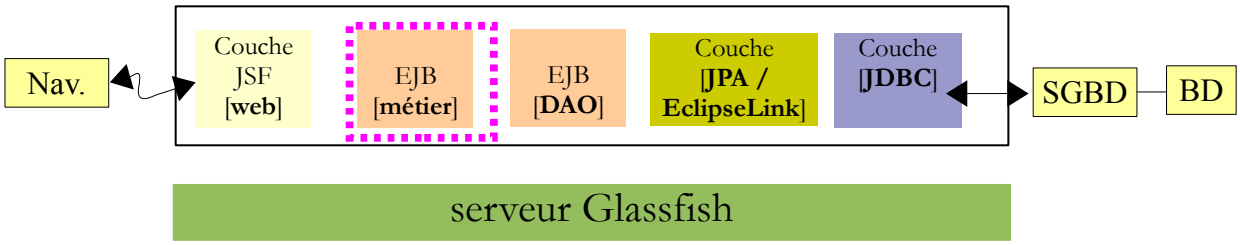


L'EJB [Metier] aura l'interface [IMetier] suivante :

```
1.  package rdvmedecins.metier.service;
2.
3.  import java.util.Date;
4.  import java.util.List;
5.
6.  import rdvmedecins.jpa.Client;
7.  import rdvmedecins.jpa.Creneau;
8.  import rdvmedecins.jpa.Medecin;
9.  import rdvmedecins.jpa.Rv;
10. import rdvmedecins.metier.entites.AgendaMedecinJour;
11.
12. public interface IMetier {
13.
```

```
14.         // couche dao
15.         // liste des clients
16.         public List<Client> getAllClients();
17.
18.         // liste des Médecins
19.         public List<Medecin> getAllMedecins();
20.
21.         // liste des créneaux horaires d'un médecin
22.         public List<Creneau> getAllCreneaux(Medecin medecin);
23.
24.         // liste des Rv d'un médecin, un jour donné
25.         public List<Rv> getRvMedecinJour(Medecin medecin, Date jour);
26.
27.         // trouver un client identifié par son id
28.         public Client getClientById(Long id);
29.
30.         // trouver un client idenbtifié par son id
31.         public Medecin getMedecinById(Long id);
32.
33.         // trouver un Rv identifié par son id
34.         public Rv getRvById(Long id);
35.
36.         // trouver un créneau horaire identifié par son id
37.         public Creneau getCreneauById(Long id);
38.
39.         // ajouter un RV
40.         public Rv ajouterRv(Date jour, Creneau creneau, Client client);
41.
42.         // supprimer un RV
43.         public void supprimerRv(Rv rv);
44.
45.         // metier
46.    public AgendaMedecinJour getAgendaMedecinJour(Medecin medecin, Date jour);
47.
48. }
```

Pour comprendre cette interface, il faut se rappeler l'architecture du projet :



Nous avons défini l'interface de la couche [DAO] (paragraphe 3.4.4, page 179) et avons indiqué que celle-ci répondait à des besoins de la couche [web], des besoins utilisateur. La couche [web] ne communique avec la couche [DAO] que via la couche [métier]. Ceci explique qu'on retrouve dans la couche [métier] toutes les méthodes de la couche [DAO]. Ces méthodes ne feront que déléguer la demande de la couche [web] à la couche [DAO]. Rien de plus.

Lors de l'étude de l'application, il apparaît un besoin : être capable d'afficher sur une page web l'agenda d'un médecin pour un jour donné afin de connaître les créneaux occupés et libres du jour. C'est typiquement le cas lorsque la secrétaire répond à une demande au téléphone. On lui demande un rendez-vous pour tel jour avec tel médecin. Pour répondre à ce besoin, la couche [métier] offre la méthode de la ligne 46.

```
        // metier
public AgendaMedecinJour getAgendaMedecinJour(Medecin medecin, Date jour);
```

On peut se demander où placer cette méthode :

- on pourrait la placer dans la couche [DAO]. Cependant, cette méthode ne répond pas vraiment à un besoin d'accès aux données mais plutôt à un besoin métier,
- on pourrait la placer dans la couche [web]. Ce serait là une mauvaise idée. Car si on change la couche [web] en une couche [Swing], on perdra la méthode alors que le besoin est toujours présent.

La méthode reçoit en paramètres le médecin et le jour pour lequel on veut l'agenda des réservations. Elle rend un objet [AgendaMedecinJour] qui représente l'agenda du médecin et du jour :

```
1.  package rdvmedecins.metier.entites;
2.
3.  import java.io.Serializable;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Date;
6.  import rdvmedecins.jpa.Medecin;
7.
8.  public class AgendaMedecinJour implements Serializable {
9.
10.         private static final long serialVersionUID = 1L;
11.         // champs
12.         private Medecin medecin;
13.         private Date jour;
14.         private CreneauMedecinJour[] creneauxMedecinJour;
15.
16.         // constructeurs
17.         public AgendaMedecinJour() {
18.
19.         }
20.
21.         public AgendaMedecinJour(Medecin medecin, Date jour, CreneauMedecinJour[] creneauxMedecinJour) {
22.                 this.medecin = medecin;
23.                 this.jour = jour;
24.                 this.creneauxMedecinJour = creneauxMedecinJour;
25.         }
26.
27.         public String toString() {
28.                 StringBuffer str = new StringBuffer("");
29.                 for (CreneauMedecinJour cr : creneauxMedecinJour) {
30.                         str.append(" ");
31.                         str.append(cr.toString());
32.                 }
33.                 return String.format("Agenda[%s,%s,%s]", medecin, new
    SimpleDateFormat("dd/MM/yyyy").format(jour), str.toString());
34.         }
35.
36.         // getters et setters
37. ...
38.
39. }
```

- ligne 12 : le médecin dont c'est l'agenda,
- ligne 13 : le jour de l'agenda,
- ligne 14 : les créneaux horaires du médecin pour ce jour.
- la classe présente des constructeurs (lignes 17, 21) ainsi qu'une méthode *toString* adaptée (ligne 27).

La classe [CreneauMedecinJour] (ligne 14) est la suivante :

```
1.  package rdvmedecins.metier.entites;
2.
3.  import java.io.Serializable;
4.  import rdvmedecins.jpa.Creneau;
5.
6.  import rdvmedecins.jpa.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.   ...
34. }
```

- ligne 12 : un créneau horaire du médecin,
- ligne 13 : le rendez-vous associé, *null* si le créneau est libre.

On voit ainsi que le champ *creneauxMedecinJour* de la ligne 14 de la classe [AgendaMedecinJour] nous permet d'avoir tous les créneaux horaires du médecin avec l'information " occupé " ou " libre " pour chacun d'eux. C'était le but de la nouvelle méthode [getAgendaMedecinJour] de l'interface [IMetier].

Notre Ejb [Metier] aura une interface locale et une interface distante qui se contenteront de dériver l'interface principale [IMetier] :

```
1.  package rdvmedecins.metier.service;
2.  import javax.ejb.Local;
3.
4.  @Local
5.  public interface IMetierLocal extends IMetier{
6.
7.  }
```

```
1.  package rdvmedecins.metier.service;
2.  import javax.ejb.Remote;
3.
4.  @Remote
5.  public interface IMetierRemote extends IMetier{
6.
7.  }
```

L'EJB [Metier] implémente ces interfaces de la façon suivante :

```
1.  package rdvmedecins.metier.service;
2.
3.  import java.io.Serializable;
4.  import java.util.Date;
5.  import java.util.Hashtable;
6.  import java.util.List;
7.  import java.util.Map;
8.
9.  import javax.ejb.EJB;
10. import javax.ejb.Singleton;
11. import javax.ejb.TransactionAttribute;
12. import javax.ejb.TransactionAttributeType;
13.
14. import rdvmedecins.dao.IDaoLocal;
15. import rdvmedecins.jpa.Client;
16. import rdvmedecins.jpa.Creneau;
17. import rdvmedecins.jpa.Medecin;
18. import rdvmedecins.jpa.Rv;
19. import rdvmedecins.metier.entites.AgendaMedecinJour;
20. import rdvmedecins.metier.entites.CreneauMedecinJour;
21.
22. @Singleton
23. @TransactionAttribute(TransactionAttributeType.REQUIRED)
24. public class Metier implements IMetierLocal, IMetierRemote, Serializable {
25.
26.   // couche dao
27.   @EJB
28.   private IDaoLocal dao;
29.
30.   public Metier() {
31.   }
32.
33.   @Override
```

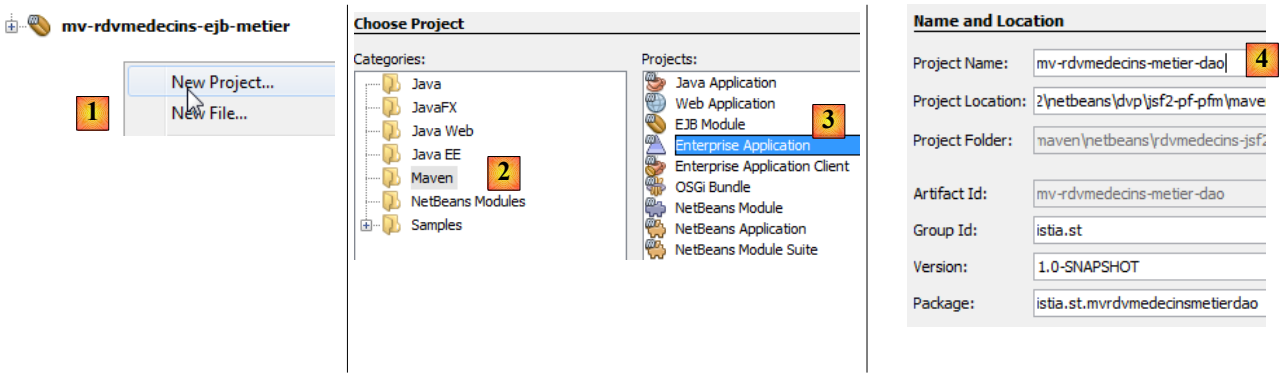
```
34.   public List<Client> getAllClients() {
35.     return dao.getAllClients();
36.   }
37.
38.   @Override
39.   public List<Medecin> getAllMedecins() {
40.     return dao.getAllMedecins();
41.   }
42.
43.   @Override
44.   public List<Creneau> getAllCreneaux(Medecin medecin) {
45.     return dao.getAllCreneaux(medecin);
46.   }
47.
48.   @Override
49.   public List<Rv> getRvMedecinJour(Medecin medecin, Date jour) {
50.     return dao.getRvMedecinJour(medecin, jour);
51.   }
52.
53.   @Override
54.   public Client getClientById(Long id) {
55.     return dao.getClientById(id);
56.   }
57.
58.   @Override
59.   public Medecin getMedecinById(Long id) {
60.     return dao.getMedecinById(id);
61.   }
62.
63.   @Override
64.   public Rv getRvById(Long id) {
65.     return dao.getRvById(id);
66.   }
67.
68.   @Override
69.   public Creneau getCreneauById(Long id) {
70.     return dao.getCreneauById(id);
71.   }
72.
73.   @Override
74.   public Rv ajouterRv(Date jour, Creneau creneau, Client client) {
75.     return dao.ajouterRv(jour, creneau, client);
76.   }
77.
78.   @Override
79.   public void supprimerRv(Rv rv) {
80.     dao.supprimerRv(rv);
81.   }
82.
83.   @Override
84.   public AgendaMedecinJour getAgendaMedecinJour(Medecin medecin, Date jour) {
85.     // liste des créneaux horaires du médecin
86.     List<Creneau> creneauxHoraires = dao.getAllCreneaux(medecin);
87.     // liste des réservations de ce même médecin pour ce même jour
88.     List<Rv> reservations = dao.getRvMedecinJour(medecin, jour);
89.     // on crée un dictionnaire à partir des Rv pris
90.     Map<Long, Rv> hReservations = new Hashtable<Long, Rv>();
91.     for (Rv resa : reservations) {
92.       hReservations.put(resa.getCreneau().getId(), resa);
93.     }
94.     // on crée l'agenda pour le jour demandé
95.     AgendaMedecinJour agenda = new AgendaMedecinJour();
96.     // le médecin
97.     agenda.setMedecin(medecin);
98.     // le jour
99.     agenda.setJour(jour);
100.    // les créneaux de réservation
101.    CreneauMedecinJour[] creneauxMedecinJour = new CreneauMedecinJour[creneauxHoraires.size()];
102.    agenda.setCreneauxMedecinJour(creneauxMedecinJour);
103.    // remplissage des créneaux de réservation
104.    for (int i = 0; i < creneauxHoraires.size(); i++) {
105.      // ligne i agenda
106.      creneauxMedecinJour[i] = new CreneauMedecinJour();
107.      // id du créneau
108.      creneauxMedecinJour[i].setCreneau(creneauxHoraires.get(i));
109.      // le créneau est-il libre ou réservé ?
```

```
110.       if (hReservations.containsKey(creneauxHoraires.get(i).getId())) {
111.         // le créneau est occupé - on note la resa
112.         Rv resa = hReservations.get(creneauxHoraires.get(i).getId());
113.         creneauxMedecinJour[i].setRv(resa);
114.       }
115.     }
116.     // on rend le résultat
117.     return agenda;
118.   }
119. }
```

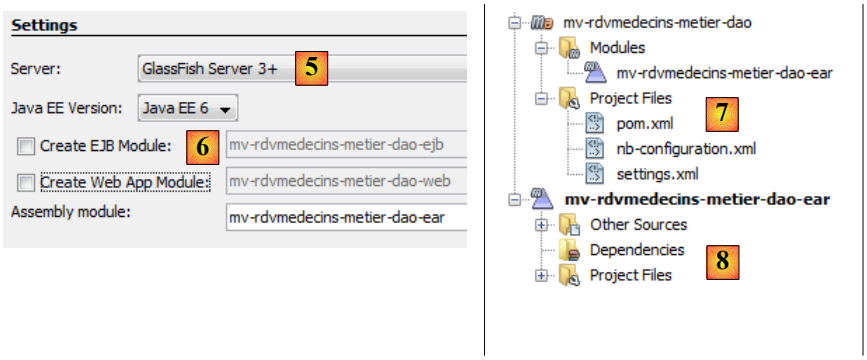
- ligne 22, la classe [Metier] est un EJB singleton,
- ligne 23, chaque méthode de l'EJB se déroule au sein d'une transaction. Cela veut dire que la transaction démarre au début de la méthode, dans la couche [métier]. Celle-ci va appeler des méthodes de la couche [DAO]. Celles-ci se dérouleront au sein de la même transaction,
- ligne 24, l'EJB implémente ses interfaces locale et distante et est de plus sérialisable,
- ligne 27 : une référence sur l'EJB de la couche [DAO],
- ligne 29 : celle-ci sera injectée par le conteneur EJB du serveur Glassfish, grâce à l'annotation @EJB. Donc lorsque les méthodes de la classe [Metier] s'exécutent, la référence sur l'EJB de la couche [DAO] a été initialisée,
- lignes 33-81 : cette référence est utilisée pour déléguer à la couche [DAO] l'appel fait à la couche [métier],
- ligne 84 : la méthode *getAgendaMedecinJour* qui permet d'avoir l'agenda d'un médecin pour un jour donné. Nous laissons le lecteur suivre les commentaires.

3.5.2 Déploiement de la couche [métier]

La couche [métier] a une dépendance sur la couche [DAO]. Chaque couche a été implémentée avec un EJB. Pour tester l'EJB [métier], il nous faut déployer les deux EJB. Pour cela, nous avons besoin d'un projet d'entreprise.



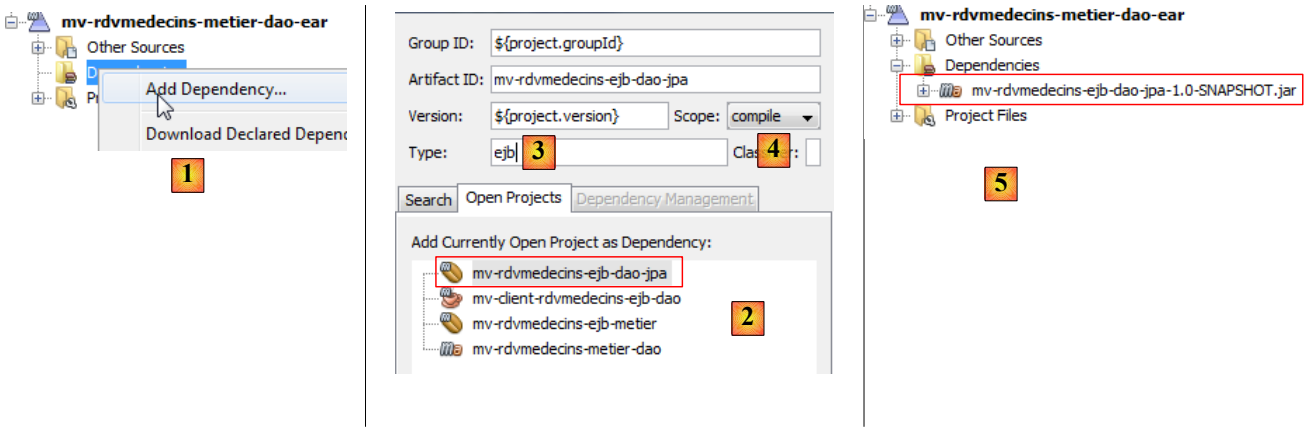
- [1], on crée un nouveau projet,
- de type Maven [2] et Application d'entreprise [3],
- on lui donne un nom [4]. Le suffixe *ear* lui sera automatiquement ajouté,



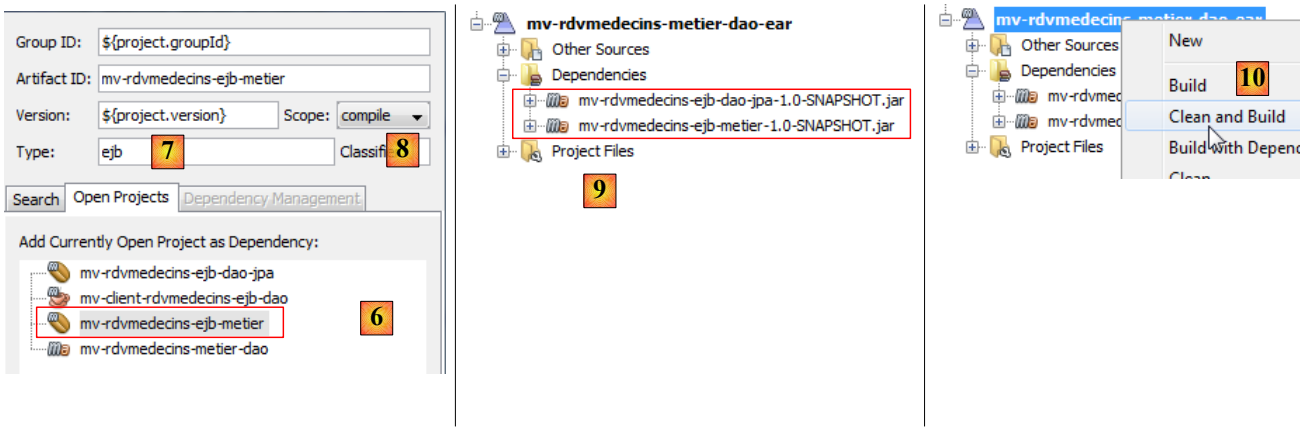
- en [5], on choisit le serveur Glassfish et Java EE 6,

- en [6], une application d'entreprise contient des modules, en général des modules EJB et des modules web. Ici, l'application d'entreprise va contenir les modules des deux EJB que nous avons construits. Comme ces modules existent, on ne coche pas les cases,
- en [7,8], deux projets on été créés. [8] est le projet d'entreprise que nous allons utiliser. [7] est un projet dont j'ignore le rôle. Je n'ai pas eu à l'utiliser et n'ayant pas approfondi Maven, je ne sais pas à quoi il peut servir. Donc nous l'ignorerons.

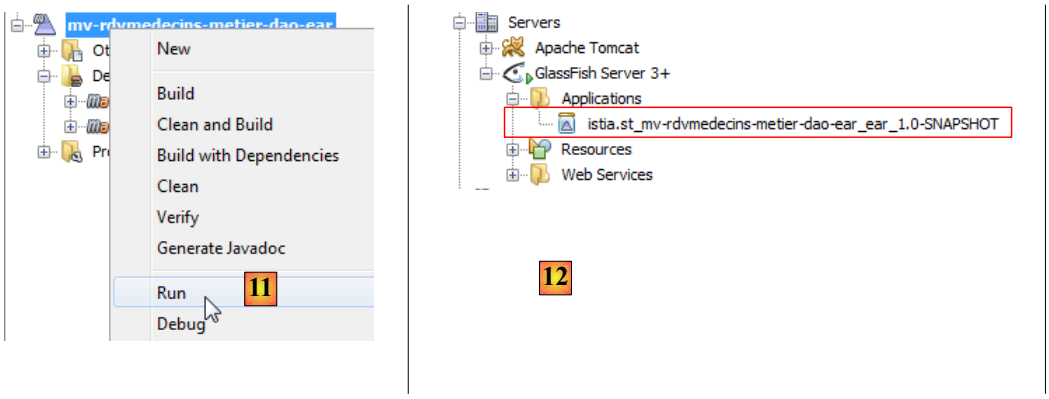
Maintenant que le projet d'entreprise est créé, nous pouvons lui définir ses modules.



- en [1], on crée une nouvelle dépendance,
- en [2], on choisit le projet de l'EJB [DAO],
- en [3], on déclare que c'est un EJB. Ne pas laisser le type vide car dans ce cas c'est le type *jar* qui sera utilisé et ici ce type ne convient pas,
- en [4], on utilise la portée [compile],
- en [5], le projet avec sa nouvelle dépendance,

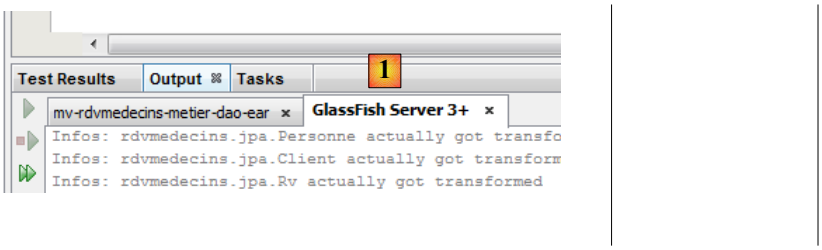


- en [6, 7, 8], on recommence pour ajouter l'EJB de la couche [métier],
- en [9], les deux dépendances,
- en [10], on construit le projet,



- en [11], on l'exécute,
- en [12], dans l'onglet [Services], on voit que le projet a été déployé sur le serveur Glassfish. Cela veut dire que les deux Ejb sont désormais présents sur le serveur.

Dans les logs du serveur Glassfish, on trouve des informations sur le déploiement des deux EJB :



- en [1], l'onglet des logs de Glassfish.

On y trouve les logs suivants :

```
1.  Infos: rdvmedecins.jpa.Creneau actually got transformed
2.  Infos: rdvmedecins.jpa.Medecin actually got transformed
3.  Infos: rdvmedecins.jpa.Personne actually got transformed
4.  Infos: rdvmedecins.jpa.Client actually got transformed
5.  Infos: rdvmedecins.jpa.Rv actually got transformed
6.  Infos: EclipseLink, version: Eclipse Persistence Services - 2.3.2.v20111125-r10461
7.  Infos: file:/D:/data/istia-1112/netbeans/dvp/jsf2-pf-pfm/maven/netbeans/rdvmedecins-jsf2-ejb/mv-
    rdvmedecins-metier-dao/mv-rdvmedecins-metier-dao-ear/target/gfdeploy/istia.st_mv-rdvmedecins-
    metier-dao-ear_ear_1.0-SNAPSHOT/mv-rdvmedecins-ejb-dao-jpa-1.0-SNAPSHOT_jar/_dbrdvmedecins2-PU
    login successful
8.  Infos: EJB5181:Portable JNDI names for EJB DaoJpa: [java:global/istia.st_mv-rdvmedecins-metier-
    dao-ear_ear_1.0-SNAPSHOT/mv-rdvmedecins-ejb-dao-jpa-1.0-SNAPSHOT/DaoJpa!
    rdvmedecins.dao.IDaoRemote, java:global/istia.st_mv-rdvmedecins-metier-dao-ear_ear_1.0-
    SNAPSHOT/mv-rdvmedecins-ejb-dao-jpa-1.0-SNAPSHOT/DaoJpa!rdvmedecins.dao.IDaoLocal]
9.  Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB DaoJpa:
    [rdvmedecins.dao#rdvmedecins.dao.IDaoRemote, rdvmedecins.dao]
10. Infos: EJB5181:Portable JNDI names for EJB Metier: [java:global/istia.st_mv-rdvmedecins-metier-
    dao-ear_ear_1.0-SNAPSHOT/mv-rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!
    rdvmedecins.metier.service.IMetierRemote, java:global/istia.st_mv-rdvmedecins-metier-dao-
    ear_ear_1.0-SNAPSHOT/mv-rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!
    rdvmedecins.metier.service.IMetierLocal]
11. Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB Metier:
    [rdvmedecins.metier.service.IMetierRemote#rdvmedecins.metier.service.IMetierRemote,
    rdvmedecins.metier.service.IMetierRemote]
```

- lignes 1-5 : les entités JPA ont été reconnues,
- ligne 7 : indique que la construction de l'unité de persistance [dbrdvmedecins2-PU] a réussi et que la connexion à la base de données associée a été faite,
- ligne 8 : les noms portables des interfaces distante et locale de l'EJB [DaoJpa]. *portable* veut dire reconnu par tous les serveurs d'application,
- ligne 9 : la même chose mais avec des noms propriétaires à Glassfish,
- lignes 10-11 : même chose pour l'EJB [Metier].

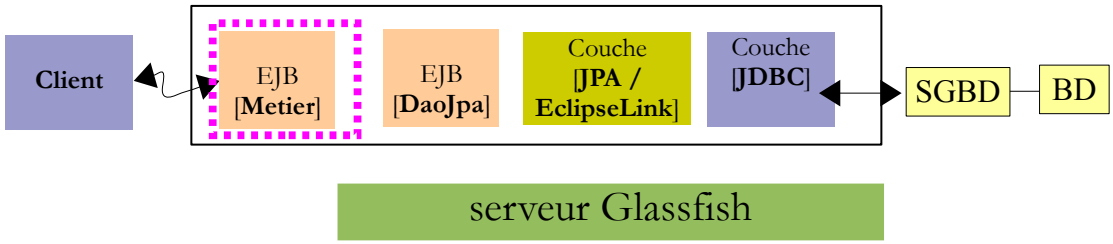
Nous retiendrons le nom portable de l'interface distante de l'EJB [Metier] :

```
java:global/istia.st_mv-rdvmedecins-metier-dao-ear_ear_1.0-SNAPSHOT/mv-rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!rdvmedecins.metier.service.IMetierRemote
```

Nous en aurons besoin lors des tests de la couche [métier].

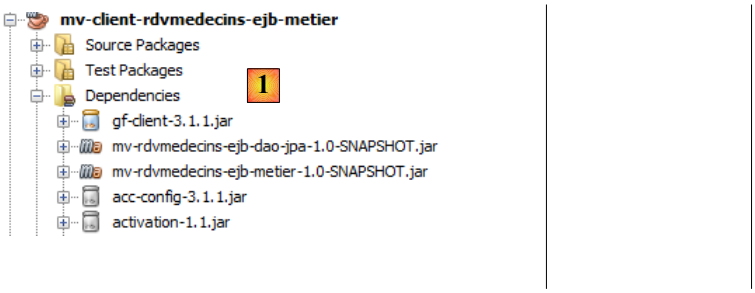
3.5.3 Test de la couche [métier]

Comme nous l'avons fait pour la couche [DAO], nous allons tester la couche [métier] dans le cadre d'une application client / serveur :



Le client va tester l'interface distante de l'EJB [Metier] déployé sur le serveur Glassfish.

Nous commençons par créer un nouveau projet Maven. Pour cela, nous suivons la démarche utilisée pour créer le projet de test de la couche [dao] (cf page 186), la création du test JUnit exclue. Le projet ainsi créé est le suivant



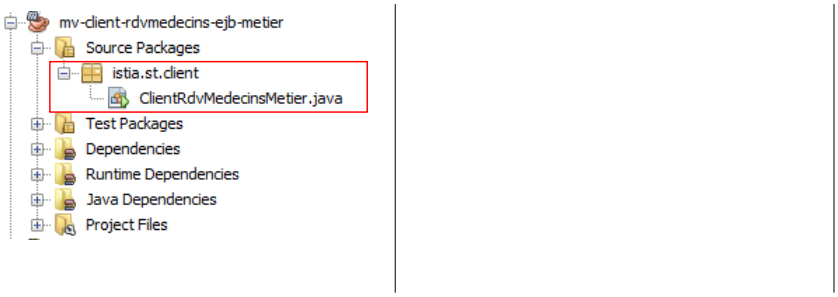
- en [1], le projet créé avec ses dépendances : vis à vis de l'EJB de la couche [dao], de celui de l'EJB de la couche [métier], de la bibliothèque [gf-client].

A ce point, le fichier [pom.xml] du projet 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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-client-rdvmedecins-ejb-metier</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>mv-client-rdvmedecins-ejb-metier</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <properties>
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.   <dependencies>
18.     <dependency>
19.       <groupId>org.glassfish.appclient</groupId>
```

```
20.       <artifactId>gf-client</artifactId>
21.       <version>3.1.1</version>
22.     </dependency>
23.     <dependency>
24.       <groupId>${project.groupId}</groupId>
25.       <artifactId>mv-rdvmedecins-ejb-dao-jpa</artifactId>
26.       <version>${project.version}</version>
27.     </dependency>
28.     <dependency>
29.       <groupId>${project.groupId}</groupId>
30.       <artifactId>mv-rdvmedecins-ejb-metier</artifactId>
31.       <version>${project.version}</version>
32.     </dependency>
33.   </dependencies>
34. </project>
```

On s'assurera de bien avoir les dépendances décrites lignes 17-33. Le test sera une simple classe console :



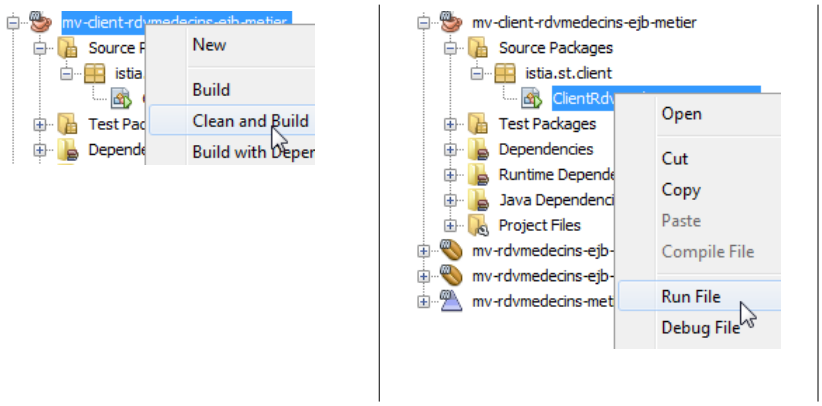
Le code de la classe [ClientRdvMedecinsMetier] est le suivant :

```
1.  package istia.st.client;
2.
3.  import java.util.Date;
4.  import java.util.List;
5.
6.  import javax.naming.InitialContext;
7.  import rdvmedecins.jpa.Client;
8.  import rdvmedecins.jpa.Creneau;
9.
10. import rdvmedecins.jpa.Medecin;
11. import rdvmedecins.jpa.Rv;
12. import rdvmedecins.metier.entites.AgendaMedecinJour;
13. import rdvmedecins.metier.service.IMetierRemote;
14.
15. public class ClientRdvMedecinsMetier {
16.
17.   // le nom de l'interface distante de l'EJB [Metier]
18.   private static String IDaoRemoteName = "java:global/istia.st_mv-rdvmedecins-metier-dao-ear_ear_1.0-
    SNAPSHOT/mv-rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!rdvmedecins.metier.service.IMetierRemote";
19.   // date du jour
20.   private static Date jour = new Date();
21.
22.   public static void main(String[] args) {
23.     try {
24.       // contexte JNDI du serveur Glassfish
25.       InitialContext initialContext = new InitialContext();
26.       // référence sur couche [metier] distante
27.       IMetierRemote metier = (IMetierRemote) initialContext.lookup(IDaoRemoteName);
28.       // affichage clients
29.       List<Client> clients = metier.getAllClients();
30.       display("Liste des clients :", clients);
31.       // affichage médecins
32.       List<Medecin> medecins = metier.getAllMedecins();
33.       display("Liste des médecins :", medecins);
34.       // affichage créneaux d'un médecin
35.       Medecin medecin = medecins.get(0);
36.       List<Creneau> creneaux = metier.getAllCreneaux(medecin);
37.       display(String.format("Liste des créneaux du médecin %s", medecin), creneaux);
38.       // liste des Rv d'un médecin, un jour donné
39.       display(String.format("Liste des rendez-vous du médecin %s, le [%s]", medecin, jour),
    metier.getRvMedecinJour(medecin, jour));
```

```
40.       // affichage agenda
41.       AgendaMedecinJour agenda = metier.getAgendaMedecinJour(medecin, jour);
42.       System.out.println(agenda);
43.       // ajouter un RV
44.       Rv rv = null;
45.       Creneau creneau = creneaux.get(2);
46.       Client client = clients.get(0);
47.       System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s", jour,
    creneau, client));
48.       rv = metier.ajouterRv(jour, creneau, client);
49.       System.out.println("Rv ajouté");
50.       display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    metier.getRvMedecinJour(medecin, jour));
51.       // affichage agenda
52.       agenda = metier.getAgendaMedecinJour(medecin, jour);
53.       System.out.println(agenda);
54.       // supprimer un RV
55.       System.out.println("Suppression du Rv ajouté");
56.       metier.supprimerRv(rv);
57.       System.out.println("Rv supprimé");
58.       display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    metier.getRvMedecinJour(medecin, jour));
59.       // affichage agenda
60.       agenda = metier.getAgendaMedecinJour(medecin, jour);
61.       System.out.println(agenda);
62.     } catch (Throwable ex) {
63.       System.out.println("Erreur...");
64.       while (ex != null) {
65.         System.out.println(String.format("%s : %s", ex.getClass().getName(), ex.getMessage()));
66.         ex = ex.getCause();
67.       }
68.     }
69.   }
70.
71.   // méthode utilitaire - affiche les éléments d'une collection
72.   private static void display(String message, List elements) {
73.     System.out.println(message);
74.     for (Object element : elements) {
75.       System.out.println(element);
76.     }
77.   }
78. }
```

- ligne 18 : le nom portable de l'interface distante de l'Ejb [Metier] a été pris dans les logs de Glassfish,
- lignes 24-27 : on obtient une référence sur l'interface distante de l'Ejb [Metier],
- lignes 29-30 : affichent les clients,
- lignes 32-33 : affichent les médecins,
- lignes 35-37 : affichent les créneaux d'un médecin,
- ligne 39 : affichent les rendez-vous d'un médecin un jour donné,
- lignes 41-42 : l'agenda de ce même médecin pour le même jour,
- lignes 44-49 : on ajoute un rendez-vous,
- ligne 50 : on affiche les rendez-vous du médecin. Il doit y en avoir un de plus,
- lignes 52-53 : on affiche l'agenda du médecin. On doit voir le rendez-vous ajouté,
- lignes 55-57 : on supprime le rendez-vous qu'on vient d'ajouter,
- ligne 58 : cela doit se refléter dans la liste des rendez-vous du médecin,
- lignes 60-61 : et dans son agenda.

On exécute le test :



Les affichages écran obtenus sont les suivants :

```
1.  Liste des clients :
2.  Client[1,Mr,Jules,MARTIN]
3.  Client[2,Mme,Christine,GERMAN]
4.  Client[3,Mr,Jules,JACQUARD]
5.  Client[4,Melle,Brigitte,BISTROU]
6.  Liste des médecins :
7.  Médecin[1,Mme,Marie,PELISSIER]
8.  Médecin[2,Mr,Jacques,BROMARD]
9.  Médecin[3,Mr,Philippe,JANDOT]
10. Médecin[4,Melle,Justine,JACQUEMOT]
11. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER]
12. Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]]
13. Creneau [2, 1, 8:20, 8:40,Médecin[1,Mme,Marie,PELISSIER]]
14. Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]]
15. Creneau [4, 1, 9:0, 9:20,Médecin[1,Mme,Marie,PELISSIER]]
16. Creneau [5, 1, 9:20, 9:40,Médecin[1,Mme,Marie,PELISSIER]]
17. Creneau [6, 1, 9:40, 10:0,Médecin[1,Mme,Marie,PELISSIER]]
18. Creneau [7, 1, 10:0, 10:20,Médecin[1,Mme,Marie,PELISSIER]]
19. Creneau [8, 1, 10:20, 10:40,Médecin[1,Mme,Marie,PELISSIER]]
20. Creneau [9, 1, 10:40, 11:0,Médecin[1,Mme,Marie,PELISSIER]]
21. Creneau [10, 1, 11:0, 11:20,Médecin[1,Mme,Marie,PELISSIER]]
22. Creneau [11, 1, 11:20, 11:40,Médecin[1,Mme,Marie,PELISSIER]]
23. Creneau [12, 1, 11:40, 12:0,Médecin[1,Mme,Marie,PELISSIER]]
24. Creneau [13, 1, 14:0, 14:20,Médecin[1,Mme,Marie,PELISSIER]]
25. Creneau [14, 1, 14:20, 14:40,Médecin[1,Mme,Marie,PELISSIER]]
26. Creneau [15, 1, 14:40, 15:0,Médecin[1,Mme,Marie,PELISSIER]]
27. Creneau [16, 1, 15:0, 15:20,Médecin[1,Mme,Marie,PELISSIER]]
28. Creneau [17, 1, 15:20, 15:40,Médecin[1,Mme,Marie,PELISSIER]]
29. Creneau [18, 1, 15:40, 16:0,Médecin[1,Mme,Marie,PELISSIER]]
30. Creneau [19, 1, 16:0, 16:20,Médecin[1,Mme,Marie,PELISSIER]]
31. Creneau [20, 1, 16:20, 16:40,Médecin[1,Mme,Marie,PELISSIER]]
32. Creneau [21, 1, 16:40, 17:0,Médecin[1,Mme,Marie,PELISSIER]]
33. Creneau [22, 1, 17:0, 17:20,Médecin[1,Mme,Marie,PELISSIER]]
34. Creneau [23, 1, 17:20, 17:40,Médecin[1,Mme,Marie,PELISSIER]]
35. Creneau [24, 1, 17:40, 18:0,Médecin[1,Mme,Marie,PELISSIER]]
36. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 16:25:26 CEST 2012]
37. Agenda[Médecin[1,Mme,Marie,PELISSIER],23/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
```

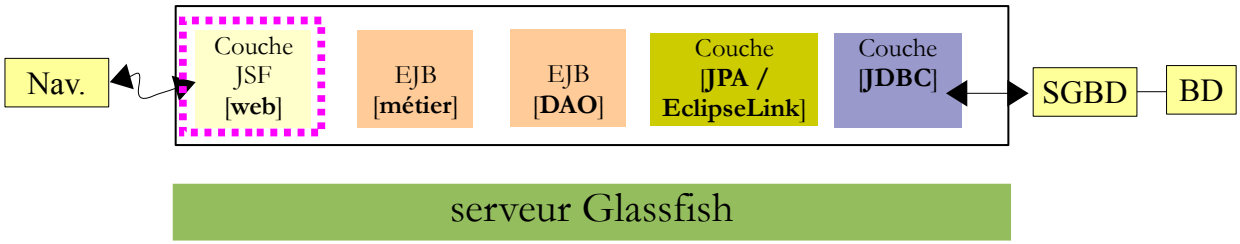
```
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
38. Ajout d'un Rv le [Wed May 23 16:25:26 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
39. Rv ajouté
40. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 16:25:26 CEST 2012]
41. Rv[252, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
42. Agenda[Médecin[1,Mme,Marie,PELISSIER],23/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] Rv[252, Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
43. Suppression du Rv ajouté
44. Rv supprimé
45. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Wed May 23 16:25:26 CEST 2012]
46. Agenda[Médecin[1,Mme,Marie,PELISSIER],23/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
```

- ligne 37 : l'agenda de Mme PELISSIER, le 23 mai 2012. Aucun créneau n'est réservé,
- ligne 39 : ajout d'un rendez-vous,
- ligne 42 : le nouvel agenda de Mme PELISSIER. Un créneau est désormais réservé à Mr MARTIN,
- ligne 44 : le rendez-vous a été supprimé,
- ligne 46 : l'agenda de Mme PELISSIER montre qu'aucun créneau n'est réservé.

Nous considérons désormais que les couches [DAO] et [métier] sont opérationnelles. Il nous reste à écrire la couche [web] avec le framework JSF. Pour cela, nous allons utiliser les connaissances acquises au début de ce document.

3.6 La couche [web]

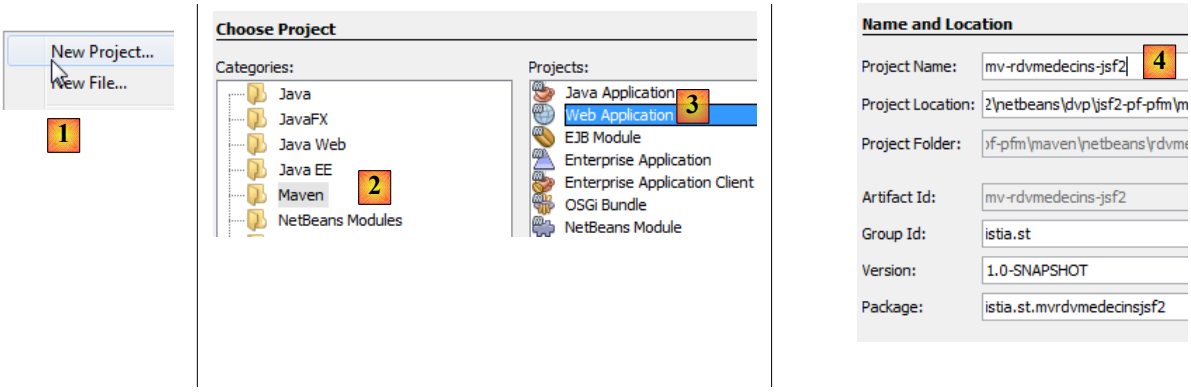
Revenons à l'architecture en cours de construction :



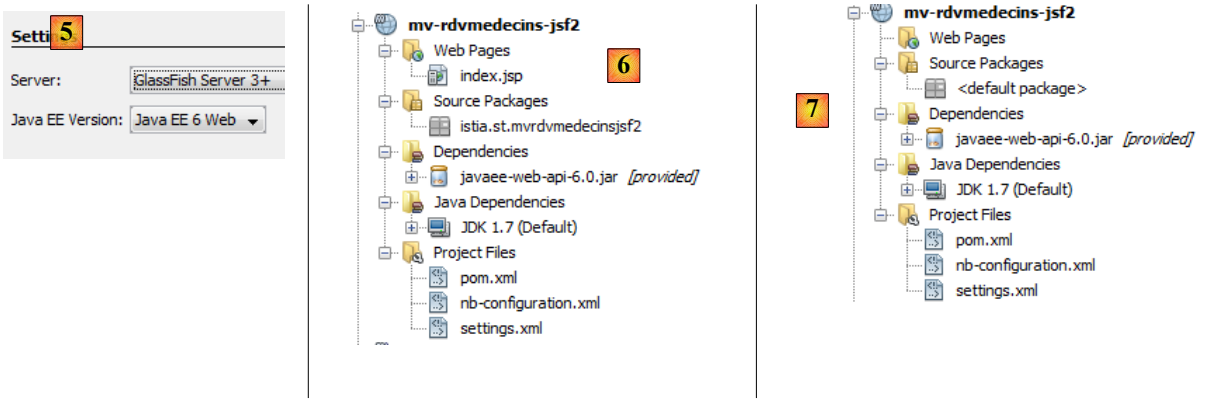
Nous allons construire la dernière couche, celle de la couche [web].

3.6.1 Le projet Netbeans

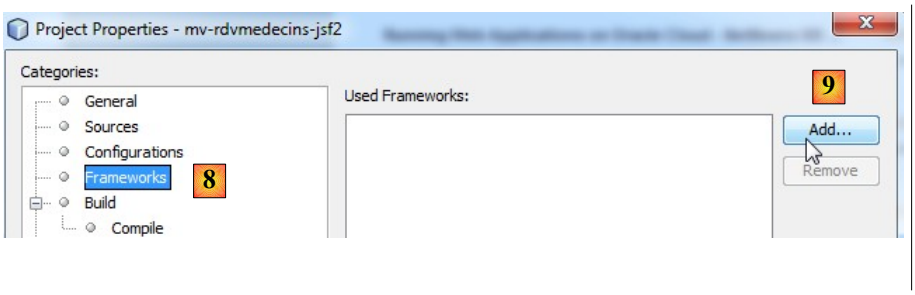
Nous construisons un projet Maven :

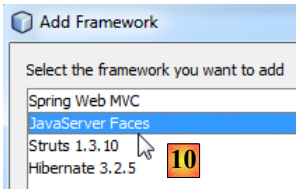


- en [1], on crée un nouveau projet,
- en [2, 3], un projet Maven de type [Web Application],
- en [4], on lui donne un nom,

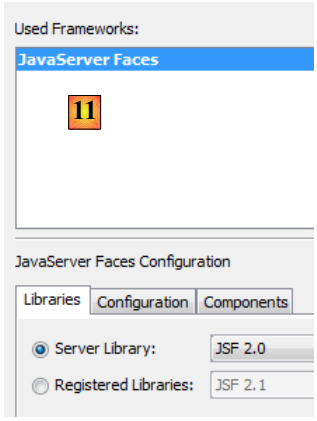


- en [5], on choisit le serveur Glassfish et Java EE 6 Web,
- en [6], le projet ainsi créé,
- en [7], le projet une fois éliminés la page [index.jsp] et le paquetage présent dans [Source Packages],





- en [8, 9], dans les propriétés du projet, on ajoute un framework,
- en [10], on choisit Java Server Faces,



- en [11], la configuration de Java Server Faces. On laisse les valeurs par défaut. On note que c'est JSF 2 qui est utilisé,
- en [12], le projet est alors modifié en deux points : un fichier [web.xml] est généré ainsi qu'une page [index.html].

Le fichier [web.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.      <context-param>
4.          <param-name>javax.faces.PROJECT_STAGE</param-name>
5.          <param-value>Development</param-value>
6.      </context-param>
7.      <servlet>
8.          <servlet-name>Faces Servlet</servlet-name>
9.          <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.         <load-on-startup>1</load-on-startup>
11.     </servlet>
12.     <servlet-mapping>
13.         <servlet-name>Faces Servlet</servlet-name>
14.         <url-pattern>/faces/*</url-pattern>
15.     </servlet-mapping>
16.     <session-config>
17.         <session-timeout>
18.             30
19.         </session-timeout>
20.     </session-config>
21.     <welcome-file-list>
22.         <welcome-file>faces/index.xhtml</welcome-file>
23.     </welcome-file-list>
24. </web-app>
```

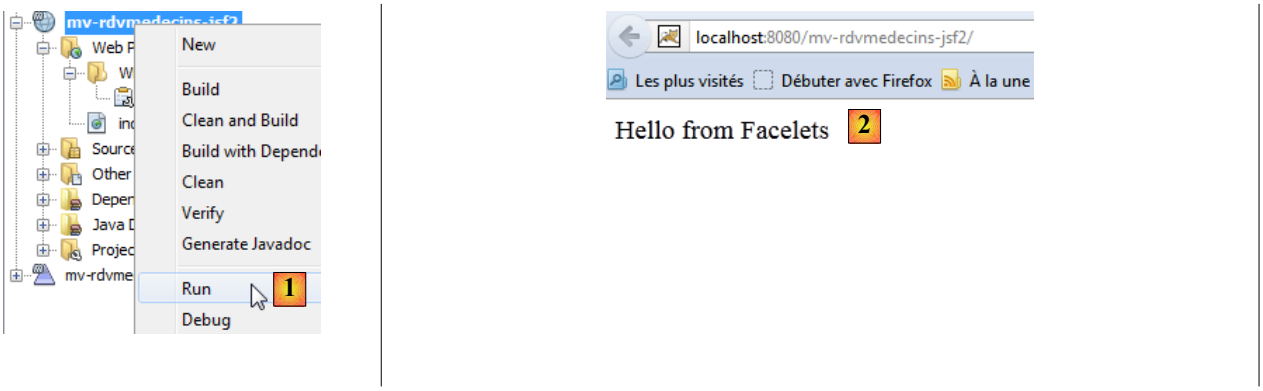
Nous avons déjà rencontré ce fichier.

- lignes 7-11 : définissent la servlet qui va traiter toutes les requêtes faites à l'application. C'est la servlet de JSF,
- lignes 12-15 : définissent les URL traitées par cette servlet. Ce sont les URL de la forme /faces/*,
- lignes 21-23 : définissent la page [index.xhtml] comme page d'accueil.

Cette page est la suivante :

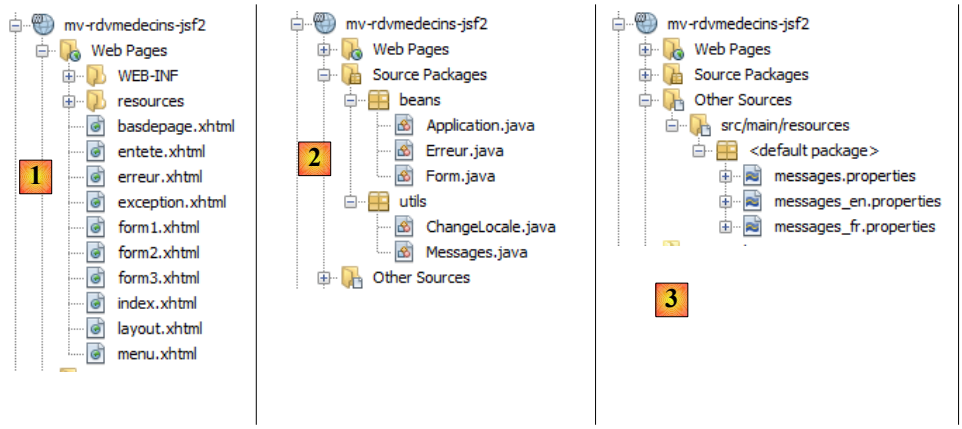
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <h:head>
6.      <title>Facelet Title</title>
7.    </h:head>
8.    <h:body>
9.      Hello from Facelets
10.   </h:body>
11. </html>
```

Nous l'avons déjà rencontrée. Nous pouvons exécuter ce projet :

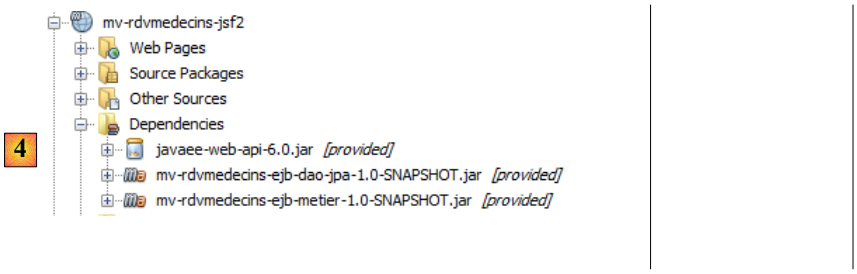


- en [1], nous exécutons le projet et nous obtenons le résultat [2] dans le navigateur.

Nous présentons maintenant le projet complet pour en détailler ensuite les différents éléments.



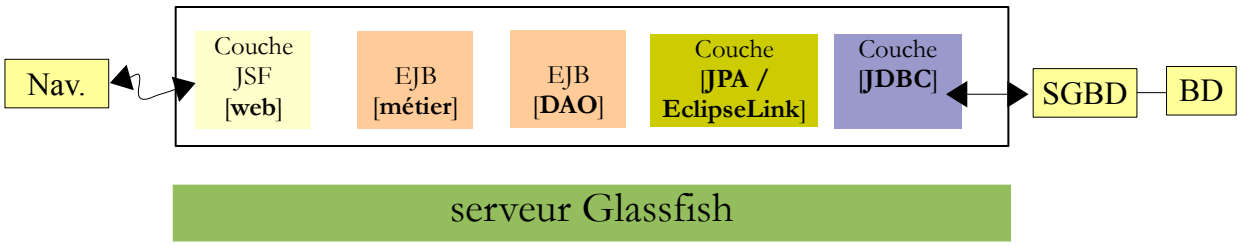
- en [1], les pages XHTML du projet,
- en [2], les codes Java,
- en [3], les fichiers de messages car l'application est internationalisée,



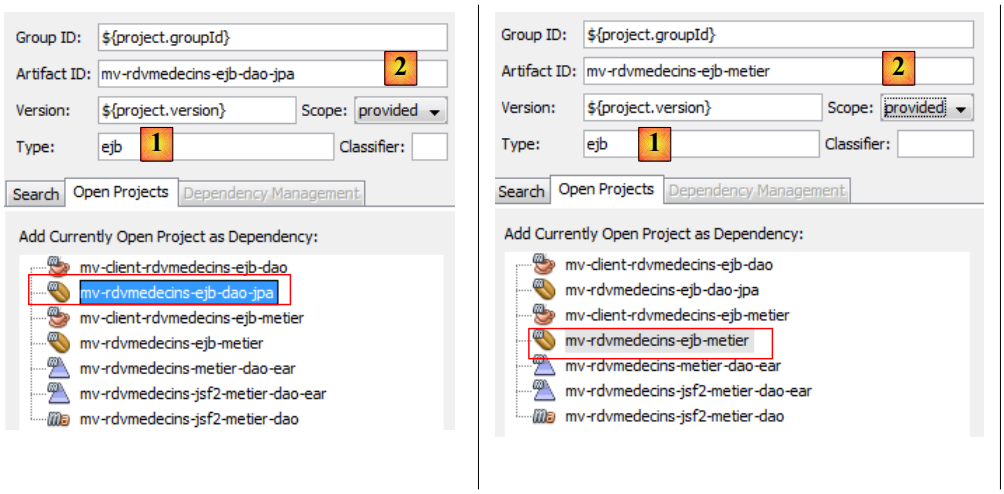
- en [4], les dépendances du projet.

3.6.2 Les dépendances du projet

Revenons à l'architecture du projet :



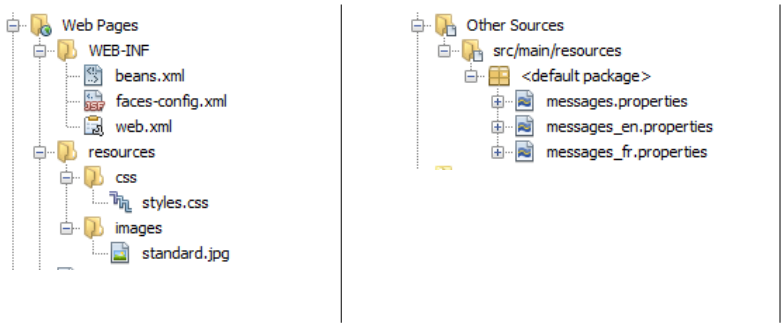
La couche JSF s'appuie sur les couches [métier], [DAO] et [JPA]. Ces trois couches sont encapsulées dans les deux projets Maven que nous avons construits, ce qui explique les dépendances du projet [4]. Simplement, montrons comment ces dépendances sont ajoutées :



- en [1], on mettra **ejb** pour indiquer que la dépendance est sur un projet EJB,
- en [2], on mettra [provided]. En effet, le projet web va être déployé en même temps que les deux projets EJB. Donc il n'a pas besoin d'embarquer les jars des EJB.

3.6.3 La configuration du projet

La configuration du projet est celle des projets JSF que nous avons étudiée au début de ce document. Nous listons les fichiers de configuration sans les réexpliquer.



[web.xml] : configure l'application web.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.    <context-param>
4.      <param-name>javax.faces.PROJECT_STAGE</param-name>
5.      <param-value>Production</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
9.      <param-value>true</param-value>
10.   </context-param>
11.   <servlet>
12.     <servlet-name>Faces Servlet</servlet-name>
13.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
14.     <load-on-startup>1</load-on-startup>
15.   </servlet>
16.   <servlet-mapping>
17.     <servlet-name>Faces Servlet</servlet-name>
18.     <url-pattern>/faces/*</url-pattern>
19.   </servlet-mapping>
20.   <session-config>
21.     <session-timeout>
22.       30
23.     </session-timeout>
24.   </session-config>
25.   <welcome-file-list>
26.     <welcome-file>faces/index.xhtml</welcome-file>
27.   </welcome-file-list>
28.   <error-page>
29.     <error-code>500</error-code>
30.     <location>/faces/exception.xhtml</location>
31.   </error-page>
32.   <error-page>
33.     <exception-type>Exception</exception-type>
34.     <location>/faces/exception.xhtml</location>
35.   </error-page>
36.
37. </web-app>
```

On notera, ligne 26 que la page [index.xhtml] est la page d'accueil de l'application.

[faces-config.xml] : configure l'application JSF

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ============ FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <resource-bundle>
```

```
      <base-name>
        messages
      </base-name>
      <var>msg</var>
    </resource-bundle>
    <message-bundle>messages</message-bundle>
  </application>
</faces-config>
```

[beans.xml] : vide mais nécessaire pour l'annotation *@Named*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

[styles.css] : la feuille de style de l'application

```
.reservationsHeaders {
   text-align: center;
   font-style: italic;
   color: Snow;
   background: Teal;
}

.creneau {
   height: 25px;
   text-align: center;
   background: MediumTurquoise;
}
.client {
   text-align: left;
   background: PowderBlue;
}

.action {
   width: 6em;
   text-align: left;
   color: Black;
   background: MediumTurquoise;
}
.erreursHeaders {
   background: Teal;
   background-color: #ff6633;
   color: Snow;
   font-style: italic;
   text-align: center

}

.erreurClasse {
   background: MediumTurquoise;
   background-color: #ffcc66;
   height: 25px;
   text-align: center
}

.erreurMessage {
   background: PowderBlue;
   background-color: #ffcc99;
   text-align: left
}
```

[messages_fr.properties] : le fichier des messages en français

```
# layout
layout.entete=Les M\u00e9decins Associ\u00e9s
layout.basdepage=ISTIA, universit\u00e9 d'Angers
layout.entete.langue1=Fran\u00e7ais
layout.entete.langue2=Anglais
# exception
exception.header=L'exception suivante s'est produite
```

```
exception.httpCode=Code HTTP de l'erreur
exception.message=Message de l'exception
exception.requestUri=Url demand\u00e9e lors de l'erreur
exception.servletName=Nom de la servlet demand\u00e9e lorsque l'erreur s'est produite
# formulaire 1
form1.titre=R\u00e9servations
form1.medecin=M\u00e9decin
form1.jour=Jour (jj/mm/aaaa)
form1.button.agenda=Agenda
form1.jour.required=date requise
form1.jour.erreur=date erron\u00e9e
# formulaire 2
form2.titre=Agenda de {0} {1} {2} le {3}
form2.titre_detail=Agenda de {0} {1} {2} le {3}
form2.creneauHoraire=Cr\u00e9neau horaire
form2.client=Client
form2.accueil=Accueil
form2.supprimer=Supprimer
form2.reserver=R\u00e9server
# formulaire 3
form3.titre=Prise de rendez-vous de {0} {1} {2}, le {3} dans le cr\u00e9neau {4,number,#00}:{5,number,#00}
- {6,number,#00}:{7,number,#00}
form3.titre_detail=Prise de rendez-vous de {0} {1} {2}, le {3} dans le cr\u00e9neau {4,number,#00}:
{5,number,#00} - {6,number,#00}:{7,number,#00}
form3.client=Client
form3.valider=Valider
form3.annuler=Annuler
# erreur
erreur.titre=Une erreur s'est produite.
erreur.message=Message d'erreur
erreur.accueil=Page d'accueil
erreur.classe=Cause
```

[messages_en.properties] : le fichier des messages en anglais

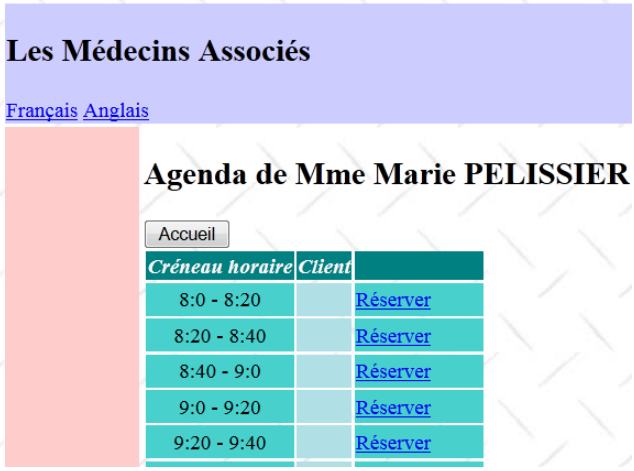
```
# layout
layout.entete=Associated Doctors
layout.basdepage=ISTIA, Angers university
layout.entete.langue1=French
layout.entete.langue2=English
# exception
exception.header=The following exceptions occurred
exception.httpCode=Error HTTP code
exception.message=Exception message
exception.requestUri=Url targeted when error occurred
exception.servletName=Servlet targeted's name when error occurred
# formulaire 1
form1.titre=Reservations
form1.medecin=Doctor
form1.jour=Date (dd/mm/yyyy)
form1.button.agenda=Diary
form1.jour.required=The date is required
form1.jour.erreur=The date is invalid
# formulaire 2
form2.titre={0} {1} {2}'' diary on {3}
form2.titre_detail={0} {1} {2}'' diary on {3}
form2.creneauHoraire=Time Period
form2.client=Client
form2.accueil=Welcome Page
form2.supprimer=Delete
form2.reserver=Reserve
# formulaire 3
form3.titre=Reservation for {0} {1} {2}, on {3} in the time period {4,number,#00}:{5,number,#00} -
{6,number,#00}:{7,number,#00}
form3.titre_detail=Reservation for {0} {1} {2}, on {3} in the time period {4,number,#00}:{5,number,#00} -
{6,number,#00}:{7,number,#00}
form3.client=Client
form3.valider=Submit
form3.annuler=Cancel
# erreur
erreur.titre=An error occurred
erreur.message=Error message
erreur.accueil=Welcome Page
erreur.classe=Cause
```

3.6.4 Les vues du projet

Rappelons le fonctionnement de l'application. La page d'accueil est la suivante :

A partir de cette première page, l'utilisateur (Secrétariat, Médecin) va engager un certain nombre d'actions. Nous les présentons ci-dessous. La vue de gauche présente la vue à partir de laquelle l'utilisateur fait une **demande**, la vue de droite la **réponse** envoyée par le serveur.

L'utilisateur a sélectionné un médecin et a saisi un jour de RV



On obtient la liste (vue partielle ici) des RV du médecin sélectionné pour le jour indiqué.

On réserve

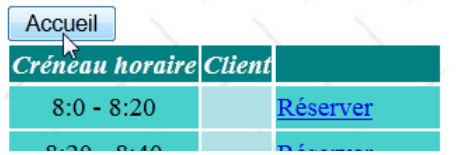
On obtient une page à renseigner

On la renseigne

Le nouveau RV apparaît dans la liste

L'utilisateur peut supprimer un RV

Le RV supprimé a disparu de la liste des RV



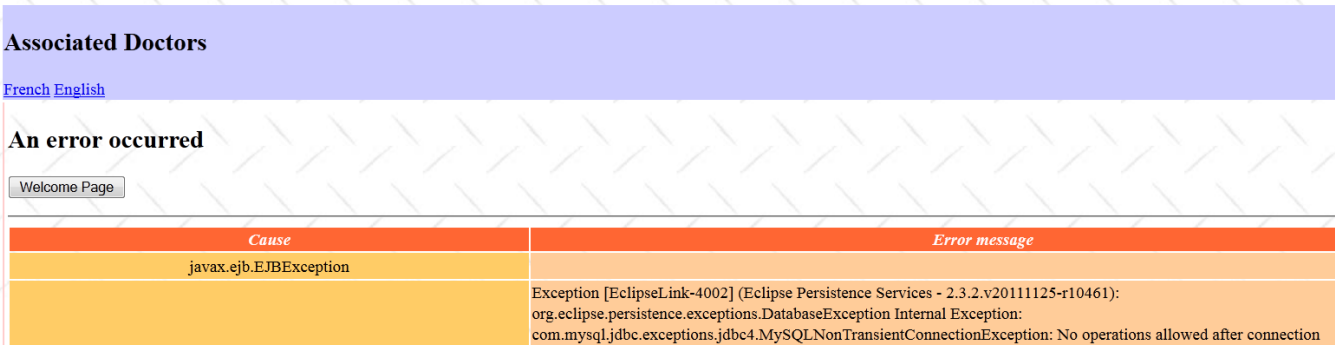
Une fois l'opération de prise de RV ou d'annulation de RV faite, l'utilisateur peut revenir à la page d'accueil

Il la retrouve dans son dernier état

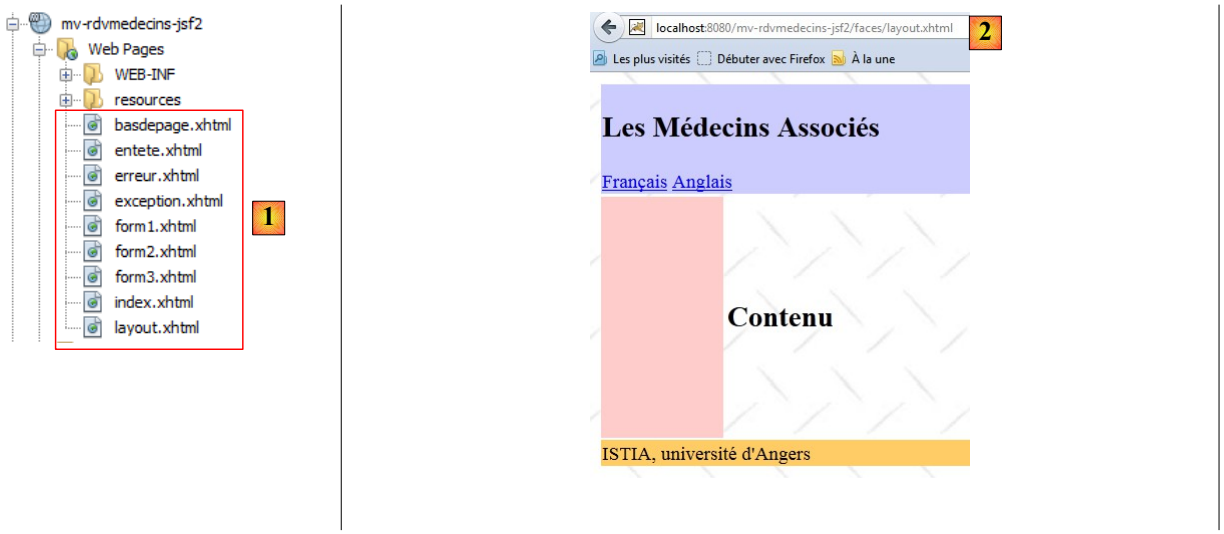
On peut changer la langue

La langue a été changée

Enfin, on peut également obtenir une page d'erreurs :



Ces différentes vues sont obtenues avec les pages suivantes du projet web :



- en [1], les pages [basdepage, entete, layout] assurent la mise en forme de toutes les vues,
- en [2], la vue produite par [layout.xhtml].

C'est la technologie des facelets qui a été utilisée ici. Celle-ci a été décrite au paragraphe 2.11, page 146. Nous nous contentons de donner le code des pages XHTML utilisées pour la mise en page :

[entete.xhtml]

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.    <body>
8.      <h2><h:outputText value="#{msg['layout.entete']}"/></h2>
9.      <div align="left">
10.       <h:commandLink value="#{msg['layout.entete.langue1']}" actionListener="#{changeLocale.setFrenchLocale}"/>
11.       <h:outputText value=" "/>
12.       <h:commandLink value="#{msg['layout.entete.langue2']}" actionListener="#{changeLocale.setEnglishLocale}"/>
13.   </div>
14.   </body>
15. </html>
```

On notera lignes 10-12, les deux liens pour changer la langue de l'application.

[basdepage.xhtml]

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html">
5.    <body>
6.      <h:outputText value="#{msg['layout.basdepage']}"/>
7.    </body>
8.  </html>
```

[layout.xhtml]

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <f:view locale="#{changeLocale.locale}">
```

```
9.      <h:head>
10.       <title>RdvMedecins</title>
11.       <h:outputStylesheet library="css" name="styles.css"/>
12.     </h:head>
13.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
14.       <h:form id="formulaire">
15.         <table style="width: 1200px">
16.           <tr>
17.             <td colspan="2" bgcolor="#ccccff">
18.               <ui:include src="entete.xhtml"/>
19.             </td>
20.           </tr>
21.           <tr>
22.             <td style="width: 100px; height: 200px" bgcolor="#ffcccc">
23.             </td>
24.             <td>
25.               <ui:insert name="contenu" >
26.                 <h2>Contenu</h2>
27.               </ui:insert>
28.             </td>
29.           </tr>
30.           <tr bgcolor="#ffcc66">
31.             <td colspan="2">
32.               <ui:include src="basdepage.xhtml"/>
33.             </td>
34.           </tr>
35.         </table>
36.       </h:form>
37.     </h:body>
38.   </f:view>
39. </html>
```

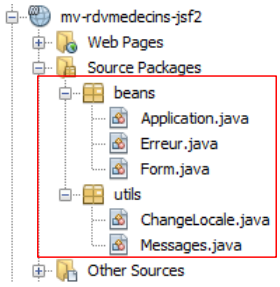
Cette page est le modèle (template) de la page [index.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.    <ui:composition template="layout.xhtml">
8.      <ui:define name="contenu">
9.        <h:panelGroup rendered="#{form.form1Rendered}">
10.         <ui:include src="form1.xhtml"/>
11.       </h:panelGroup>
12.       <h:panelGroup rendered="#{form.form2Rendered}">
13.         <ui:include src="form2.xhtml"/>
14.       </h:panelGroup>
15.       <h:panelGroup rendered="#{form.form3Rendered}">
16.         <ui:include src="form3.xhtml"/>
17.       </h:panelGroup>
18.       <h:panelGroup rendered="#{form.erreurRendered}">
19.         <ui:include src="erreur.xhtml"/>
20.       </h:panelGroup>
21.     </ui:define>
22.   </ui:composition>
23. </html>
```

Les lignes 8-21 définissent la zone appelée " contenu " (ligne 8) dans [layout.xhtml] (ligne 7). C'est la zone centrale des vues :

La page [index.xhtml] est l'unique page de l'application. Il n'y aura donc aucune navigation entre pages. Elle affiche l'une des quatre pages [form1.xhtml, form2.xhtml, form3.xhtml, erreur.xhtml]. Cet affichage est contrôlé par quatre booléens [form1Rendered, form2Rendered, form3Rendered, erreurRendered] du bean **form** que nous allons décrire prochainement.

3.6.5 Les beans du projet



Les classes du paquetage [utils] ont déjà été présentées :

- la classe [ChangeLocale] est la classe qui assure le changement de langue. Elle a déjà été étudiée (paragraphe 2.4.4, page 42).
- la classe [Messages] est une classe qui facilite l'internationalisation des messages d'une application. Elle a été étudiée au paragraphe 2.8.5.7, page 124.

3.6.5.1 Le bean Application

Le bean [Application] est le suivant :

```
1.  package beans;
2.
3.  import java.util.ArrayList;
4.  import java.util.HashMap;
5.  import java.util.List;
6.  import java.util.Map;
7.  import javax.annotation.PostConstruct;
8.  import javax.ejb.EJB;
9.  import javax.enterprise.context.ApplicationScoped;
10. import javax.inject.Named;
11. import rdvmedecins.jpa.Client;
12. import rdvmedecins.jpa.Medecin;
13. import rdvmedecins.metier.service.IMetierLocal;
```

```
14.
15. @Named(value = "application")
16. @ApplicationScoped
17. public class Application implements Serializable{
18.
19.   // couche métier
20.   @EJB
21.   private IMetierLocal metier;
22.   // cache
23.   private List<Medecin> medecins;
24.   private List<Client> clients;
25.   private Map<Long, Medecin> hMedecins = new HashMap<Long, Medecin>();
26.   private Map<Long, Client> hClients = new HashMap<Long, Client>();
27.   // erreurs
28.   private List<Erreur> erreurs = new ArrayList<Erreur>();
29.   private Boolean erreur = false;
30.
31.   public Application() {
32.   }
33.
34.   @PostConstruct
35.   public void init() {
36.     // on met les médecins et les clients en cache
37.     try {
38.       medecins = metier.getAllMedecins();
39.       clients = metier.getAllClients();
40.     } catch (Throwable th) {
41.       // on note l'erreur
42.       erreur = true;
43.       erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
44.       while (th.getCause() != null) {
45.         th = th.getCause();
46.         erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
47.       }
48.       return;
49.     }
50.     // vérification des listes
51.     if (medecins.size() == 0) {
52.       // on note l'erreur
53.       erreur = true;
54.       erreurs.add(new Erreur("", "La liste des médecins est vide"));
55.     }
56.     if (clients.size() == 0) {
57.       // on note l'erreur
58.       erreur = true;
59.       erreurs.add(new Erreur("", "La liste des clients est vide"));
60.     }
61.     // erreur ?
62.     if (erreur) {
63.       return;
64.     }
65.
66.     // les dictionnaires
67.     for (Medecin m : medecins) {
68.       hMedecins.put(m.getId(), m);
69.     }
70.     for (Client c : clients) {
71.       hClients.put(c.getId(), c);
72.     }
73.   }
74.
75.   // getters et setters
76.   ...
77. }
```

- lignes 15-16 : la classe [Application] est un bean de portée Application. Elle est créée une fois au début du cycle de vie de l'application et est accessible à toutes les requêtes de tous les utilisateurs. On utilise ici des annotations différentes de celles utilisées dans les exemples JSF étudiés au début de ce document. Ce sont des annotations CDI (Context Dependency Injection). Elles ne sont utilisables que dans un conteneur Java EE6. La correspondance entre les annotations JSF et CDI est la suivante :

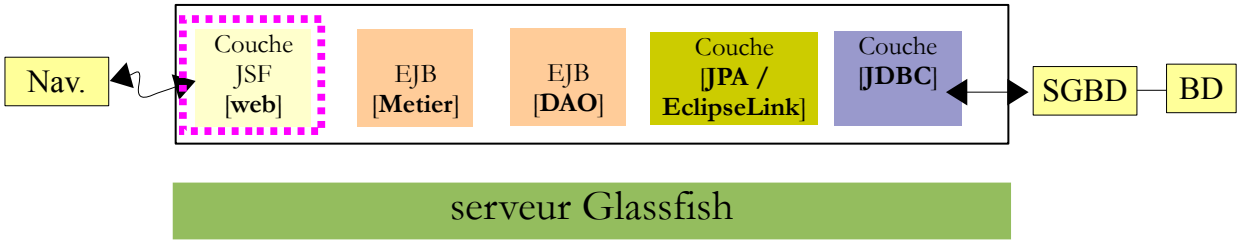
JSF	CDI
`@ManagedBean (javax.faces.bean.ManagedBean)`	`@Named (javax.inject.Named)`

`@RequestScoped (javax.faces.bean.RequestScoped)`	`@RequestScoped (javax.enterprise.context.RequestScoped)`
`@SessionScoped (javax.faces.bean.SessionScoped)`	`@SessionScoped (javax.enterprise.context.SessionScoped)`
`@ApplicationScoped (javax.faces.bean.ApplicationScoped)`	`@ApplicationScoped (javax.enterprise.context.ApplicationScoped)`
`@ManagedProperty (javax.faces.bean.ManagedProperty)`	`@Inject (javax.inject.Inject)`

Certaines annotations sont identiques dans les deux packages et prêtent à confusion. On ne mélangera pas les annotations JSF et CDI. On s'en tiendra à l'un des packages.

On place en général, dans un bean de portée Application, des données en lecture seule. Ici, nous y placerons la liste des médecins et celle des clients. Nous faisons donc l'hypothèse que celles-ci ne changent pas souvent. Les pages XHTML y ont accès via le nom **application**,

- lignes 20-21 : une référence sur l'interface locale de l'EJB [Metier] sera injectée par le conteneur EJB de Glassfish. Rappelons-nous l'architecture de l'application :



L'application JSF et l'EJB [Metier] vont s'exécuter dans la même JVM (Java Virtual Machine). Donc la couche [JSF] va utiliser l'interface locale de l'EJB. Ici, le bean **application** utilise l'EJB [Metier]. Même si ce n'était pas le cas, il serait normal d'y trouver une référence sur la couche [métier]. C'est en effet une information qui peut être partagée par toutes les requêtes de tous les utilisateurs donc une donnée de portée *Application*.

- lignes 34-35 : la méthode **init** est exécutée juste après l'instanciation de la classe [Application] (présence de l'annotation *@PostConstruct*),
- lignes 36-73, la méthode crée les éléments suivants : la liste des médecins de la ligne 23, celle des clients de la ligne 24, un dictionnaire des médecins indexé par leur **id** en ligne 25, et le même pour les clients ligne 26. Il peut se produire des erreurs. Celles-ci sont consignées dans la liste de la ligne 28.

La classe [Erreur] est la suivante :

```
1.  package beans;
2.
3.  public class Erreur {
4.
5.    public Erreur() {
6.    }
7.
8.    // champ
9.    private String classe;
10.   private String message;
11.
12.   // constructeur
13.   public Erreur(String classe, String message){
14.     this.setClasse(classe);
15.     this.message=message;
16.   }
17.
18.   // getters et setters
19. ...
20. }
```

- ligne 9, le nom d'une classe d'exception si une exception a été lancée,
- ligne 10 : un message d'erreur.

3.6.5.2 Le bean [Form]

Son code est le suivant :

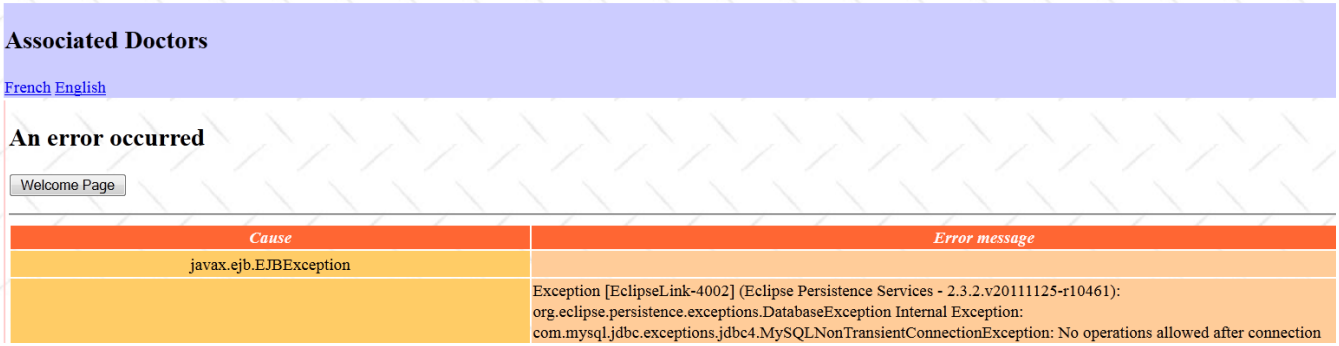
```
1.  package beans;
2.
3.  ...
4.
5.  @Named(value = "form")
6.  @SessionScoped
7.  public class Form implements Serializable {
8.
9.    public Form() {
10.   }
11.
12.   // bean Application
13.   @Inject
14.   private Application application;
15.
16.   // modèle
17.   private Long idMedecin;
18.   private Date jour = new Date();
19.   private Boolean form1Rendered = true;
20.   private Boolean form2Rendered = false;
21.   private Boolean form3Rendered = false;
22.   private Boolean erreurRendered = false;
23.   private String form2Titre;
24.   private String form3Titre;
25.   private AgendaMedecinJour agendaMedecinJour;
26.   private Long idCreneau;
27.   private Medecin medecin;
28.   private Client client;
29.   private Long idClient;
30.   private CreneauMedecinJour creneauChoisi;
31.   private List<Erreur> erreurs;
32.
33.   @PostConstruct
34.   private void init() {
35.     // l'initialisation s'est-elle bien passée ?
36.     if (application.getErreur()) {
37.       // on récupère la liste des erreurs
38.       erreurs = application.getErreurs();
39.       // la vue des erreurs est affichée
40.       setForms(false, false, false, true);
41.     }
42.   }
43.
44.   // affichage vue
45.   private void setForms(Boolean form1Rendered, Boolean form2Rendered, Boolean form3Rendered, Boolean
    erreurRendered) {
46.     this.form1Rendered = form1Rendered;
47.     this.form2Rendered = form2Rendered;
48.     this.form3Rendered = form3Rendered;
49.     this.erreurRendered = erreurRendered;
50.   }
51. ..............................................
52. }
```

- lignes 5-7 : la classe [Form] est un bean de nom **form** et de portée **session**. On rappelle qu'alors la classe doit être sérialisable.
- lignes 13-14 : le bean **form** a une référence sur le bean **application**. Celle-ci sera injectée par le conteneur de servlets dans lequel s'exécute l'application (présence de l'annotation **@Inject**).
- lignes 17-31 : le modèle des pages [form1.xhtml, form2.xhtml, form3.xhtml, erreur.xhtml]. L'affichage de ces pages est contrôlé par les booléens des lignes 19-22. On remarquera que par défaut, c'est la page [form1.xhtml] qui est rendue,
- lignes 33-34 : la méthode **init** est exécutée juste après l'instanciation de la classe (présence de l'annotation *@PostConstruct*),
- lignes 35-41 : la méthode **init** est utilisée pour savoir quelle page doit être affichée en premier : normalement la page [form1.xhtml] (ligne 19) sauf si l'initialisation de l'application s'est mal passée (ligne 36) auquel cas c'est la page [erreur.xhtml] qui sera affichée (ligne 40).

La page [erreur.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.
8.    <body>
9.      <h2><h:outputText value="#{msg['erreur.titre']}"/></h2>
10.     <p>
11.       <h:commandButton value="#{msg['erreur.accueil']}" actionListener="#{form.accueil()}"/>
12.     </p>
13.     <hr/>
14.     <h:dataTable value="#{form.erreurs}" var="erreur" headerClass="erreursHeaders"
    columnClasses="erreurClasse,erreurMessage">
15.       <h:column>
16.         <f:facet name="header">
17.           <h:outputText value="#{msg['erreur.classe']}"/>
18.         </f:facet>
19.         <h:outputText value="#{erreur.classe}"/>
20.       </h:column>
21.       <h:column>
22.         <f:facet name="header">
23.           <h:outputText value="#{msg['erreur.message']}"/>
24.         </f:facet>
25.         <h:outputText value="#{erreur.message}"/>
26.       </h:column>
27.     </h:dataTable>
28.   </body>
29. </html>
```

Elle utilise une balise <h:dataTable> (lignes 14-27) pour afficher la liste des erreurs. Cela donne une page analogue à la suivante :

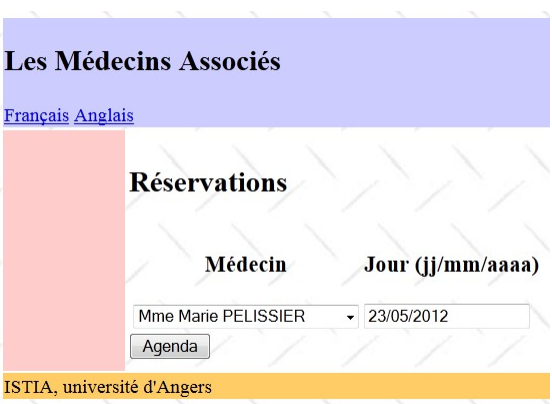


Nous allons maintenant définir les différentes phases de la vie de l'application.

3.6.6 Interactions entre pages et modèle

3.6.6.1 L'affichage de la page d'accueil

Si tout va bien, la première page affichée est [form1.xhtml]. Cela donne la vue suivante :



La page [form1.xhtml] est la suivante :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

  <body>
    <h2><h:outputText value="#{msg['form1.titre']}"/></h2>
    <h:panelGrid columns="3">
      <h:panelGroup>
      <div align="center"><h3><h:outputText value="#{msg['form1.medecin']}"/></h3></div>
      </h:panelGroup>
      <h:panelGroup>
      <div align="center"><h3><h:outputText value="#{msg['form1.jour']}"/></h3></div>
      </h:panelGroup>
      <h:panelGroup/>
      <h:selectOneMenu value="#{form.idMedecin}">
        <f:selectItems value="#{form.medecins}" var="medecin" itemLabel="#{medecin.titre} #{medecin.prenom} #{medecin.nom}" itemValue="#{medecin.id}"/>
      </h:selectOneMenu>
      <h:inputText id="jour" value="#{form.jour}"  required="true" requiredMessage="#{msg['form1.jour.required']}" converterMessage="#{msg['form1.jour.erreur']}">
        <f:convertDateTime pattern="dd/MM/yyyy"/>
      </h:inputText>
      <h:message for="jour" styleClass="error"/>
    </h:panelGrid>
    <h:commandButton value="#{msg['form1.button.agenda']}" actionListener="#{form.getAgenda}"/>
  </body>
</html>
```

Cette page est alimentée par le modèle suivant :

```
@Named(value = "form")
@SessionScoped
public class Form implements Serializable {

   // bean Application
   @Inject
   private Application application;
   // modèle
   private Long idMedecin;
   private Date jour = new Date();

// liste des médecins
   public List<Medecin> getMedecins() {
     return application.getMedecins();
   }
   // agenda
   public void getAgenda() {
     ...
}
```

- le champ de la ligne 9 alimente en lecture et écriture la valeur de la liste de la ligne 18 de la page. A l'affichage initial de la page, elle fixe la valeur sélectionnée dans le combo. A l'affichage initial, *idMedecin* est égal à *null*, donc c'est le premier médecin qui sera sélectionné,
- la méthode des lignes 13-15 génère les éléments du combo des médecins (ligne 19 de la page). Chaque option générée aura pour **label** (itemLabel) les **titre**, **nom**, **prénom** du médecin et pour valeur (itemValue), l'**id** du médecin,
- le champ de la ligne 10 alimente en lecture / écriture le champ de saisie de la ligne 21 de la page. A l'affichage initial, c'est donc la date du jour qui est affichée,
- lignes 17-19 : la méthode **getAgenda** gère le clic sur le bouton [Agenda] de la ligne 26 de la page. Comme il n'y a pas de navigation (c'est toujours la page [index.html] qui est demandée), on utilisera souvent l'attribut *actionListener* à la place de l'attribut *action*. Dans ce cas, la méthode appelée dans le modèle ne rend aucun résultat.

Lorsqu'a lieu le clic sur le bouton [Agenda],

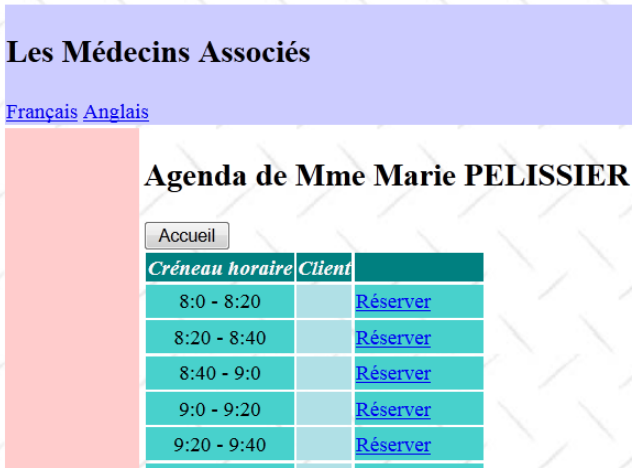
- des valeurs sont postées : la valeur sélectionnée dans le combo des médecins est enregistrée dans le champ **idMedecin** du modèle et le jour choisi dans le champ **jour**,
- la méthode **getAgenda** du modèle est appelée.

La méthode **getAgenda** est la suivante :

```
1.    // bean Application
2.    @Inject
3.    private Application application;
4.
5.    // modèle
6.    private Long idMedecin;
7.    private Date jour = new Date();
8.    private Boolean form1Rendered = true;
9.    private Boolean form2Rendered = false;
10.   private Boolean form3Rendered = false;
11.   private Boolean erreurRendered = false;
12.   private String form2Titre;
13.   private AgendaMedecinJour agendaMedecinJour;
14.   private Medecin medecin;
15.   private List<Erreur> erreurs;
16.
17.   // agenda
18.   public void getAgenda() {
19.     try {
20.       // on récupère le médecin
21.       medecin = application.gethMedecins().get(idMedecin);
22.       // titre formulaire 2
23.       form2Titre = Messages.getMessage(null, "form2.titre", new Object[]{medecin.getTitre(),
    medecin.getPrenom(), medecin.getNom(), new SimpleDateFormat("dd MMM yyyy").format(jour)}).getSummary();
24.       // l'agenda du médecin pour un jour donné
25.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
26.       // on affiche le formulaire 2
27.       setForms(false, true, false, false);
28.     } catch (Throwable th) {
29.       // vue des erreurs
30.       prepareVueErreur(th);
31.     }
32.   }
33.
34.   // préparation vueErreur
35.   private void prepareVueErreur(Throwable th) {
36.     // on crée la liste des erreurs
37.     erreurs = new ArrayList<Erreur>();
38.     erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
39.     while (th.getCause() != null) {
40.       th = th.getCause();
41.       erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
42.     }
43. // la vue des erreurs est affichée
44.     setForms(false, false, false, true);
45. }
```

Rappelons ce que doit afficher la méthode **getAgenda** :

L'utilisateur a sélectionné un médecin et a saisi un jour de RV

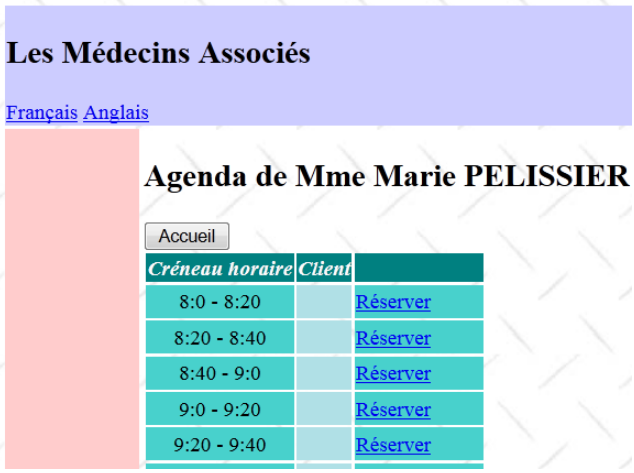


On obtient la liste (vue partielle ici) des RV du médecin sélectionné pour le jour indiqué.

- ligne 21 : on récupère le médecin sélectionné dans le dictionnaire des médecins qui a été stocké dans le bean *application*. On utilise pour cela son **id** qui a été posté dans **idMedecin**,
- ligne 23 : on prépare le titre de la page [form2.xhtml] qui va être affichée. Ce message est pris dans le fichier des messages afin qu'il soit internationalisé. Cette technique a été décrite au paragraphe 2.8.5.7, page 124.
- ligne 25 : on fait appel à la couche [métier] pour calculer l'agenda du médecin choisi pour le jour choisi,
- ligne 27 : on affiche [form2.xhtml],
- ligne 28 : si on a une exception, une liste d'erreurs est alors construite (lignes 37-42) et la page [erreur.xhtml] est affichée (ligne 44).

3.6.6.2 Afficher l'agenda d'un médecin

La page [form2.xhtml] correspond à la vue suivante :



Le code de la page [form2.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
```

```
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets"
7.        xmlns:c="http://java.sun.com/jsp/jstl/core">
8.
9.    <body>
10.     <h2><h:outputText value="#{form.form2Titre}"/></h2>
11.     <h:commandButton value="#{msg['form2.accueil']}" action="#{form.accueil}" />
12.     <h:dataTable value="#{form.agendaMedecinJour.creneauxMedecinJour}" var="creneauMedecinJour"
    headerClass="reservationsHeaders" columnClasses="creneau,client,action">
13.       <h:column>
14.         <f:facet name="header">
15.           <h:outputText value="#{msg['form2.creneauHoraire']}"/>
16.         </f:facet>
17.         <h:outputText value="#{creneauMedecinJour.creneau.hdebut}:#{creneauMedecinJour.creneau.mdebut} -
    #{creneauMedecinJour.creneau.hfin}:#{creneauMedecinJour.creneau.mfin}" />
18.       </h:column>
19.       <h:column>
20.         <f:facet name="header">
21.           <h:outputText value="#{msg['form2.client']}"/>
22.         </f:facet>
23.         <c:if test="#{creneauMedecinJour.rv==null}">
24.           <h:outputText value=""/>
25.           <c:otherwise>
26.             <h:outputText value="#{creneauMedecinJour.rv.client.titre}
    #{creneauMedecinJour.rv.client.prenom} #{creneauMedecinJour.rv.client.nom}"/>
27.           </c:otherwise>
28.         </c:if>
29.       </h:column>
30.       <h:column>
31.         <f:facet name="header"/>
32.         <h:commandLink action="#{form.action()}" value="#{creneauMedecinJour.rv==null ?
    msg['form2.reserver'] : msg['form2.supprimer']}">
33.           <f:setPropertyActionListener value="#{creneauMedecinJour.creneau.id}"
    target="#{form.idCreneau}"/>
34.         </h:commandLink>
35.       </h:column>
36.     </h:dataTable>
37.   </body>
38. </html>
```

On se rappelle que la méthode *getAgenda* a initialisé deux champs dans le modèle :

```
1.  // modèle
2.    private String form2Titre;
3.  private AgendaMedecinJour agendaMedecinJour;
```

Ces deux champs alimentent la page [form2.xhtml] :

- ligne 10, le titre de la page,
- ligne 12 : l'agenda du médecin est affiché par une balise <h:dataTable> à trois colonnes,
- lignes 13-18 : la première colonne affiche les créneaux horaires,
- lignes 19-30 : la deuxième colonne affiche le nom du client qui a éventuellement réservé le créneau horaire ou rien sinon. Pour faire ce choix, on utilise les balises de la bibliothèque **JSTL Core** référencée ligne 7,
- lignes 30-35 : la troisième colonne affiche le lien [Réserver] si le créneau est libre, le lien [Supprimer] si le créneau est occupé.

Les liens de la troisième colonne sont liés au modèle suivant :

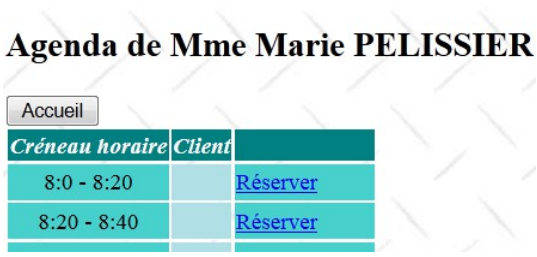
```
1.  // modèle
2.    private Long idCreneau;
3.
4.    // action sur RV
5.    public void action() {
6.      ...
7.  }
```

- la méthode *action* est appelée lorsque l'utilisateur clique sur le lien Réserver / Supprimer (ligne 32). On remarquera qu'on a utilisé ici l'attribut *action*. La méthode pointée par cet attribut devrait avoir la signature *String action*() parce que la méthode doit alors rendre une clé de navigation. Or ici, elle est *void action*(). Cela n'a pas provoqué d'erreur et on peut supposer que dans ce cas il n'y a pas de navigation. C'est ce qui était désiré. Mettre *actionListener* au lieu d'*action* provoquait un dysfonctionnement,
- le champ **idCreneau** de la ligne 2 va récupérer l'**id** du créneau horaire du lien qui a été cliqué (ligne 33 de la page).

3.6.6.3 Suppression d'un rendez-vous

Examinons le code qui gère la suppression d'un rendez-vous. Cela correspond à la séquence de vues suivante :

L'utilisateur peut supprimer un RV



Le RV supprimé a disparu de la liste des RV

Le code concerné par cette opération est le suivant :

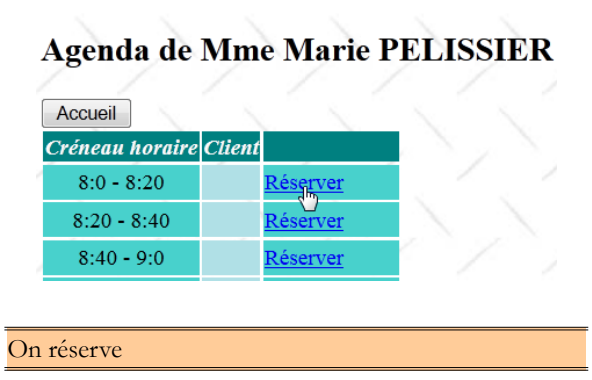
```
1.  // bean Application
2.    @Inject
3.    private Application application;
4.
5.    // modèle
6.    private Boolean form1Rendered = true;
7.    private Boolean form2Rendered = false;
8.    private Boolean form3Rendered = false;
9.    private Boolean erreurRendered = false;
10.   private AgendaMedecinJour agendaMedecinJour;
11.   private Long idCreneau;
12.   private CreneauMedecinJour creneauChoisi;
13.   private List<Erreur> erreurs;
14.
15.   // action sur RV
16.   public void action() {
17.     // on recherche le créneau dans l'agenda
18.     int i = 0;
19.     Boolean trouvé = false;
20.     while (!trouvé && i < agendaMedecinJour.getCreneauxMedecinJour().length) {
21.       if (agendaMedecinJour.getCreneauxMedecinJour()[i].getCreneau().getId() == idCreneau) {
22.         trouvé = true;
23.       } else {
24.         i++;
25.       }
26.     }
27.     // a-t-on trouvé ?
28.     if (!trouvé) {
29.       // c'est bizarre - on réaffiche form2
30.       setForms(false, true, false, false);
31.       return;
32.     }
33.     // on a trouvé
34.     creneauChoisi = agendaMedecinJour.getCreneauxMedecinJour()[i];
35.     // selon l'action désirée
36.     if (creneauChoisi.getRv() == null) {
37.       reserver();
38.     } else {
39.       supprimer();
```

```
40.     }
41.   }
42.   // réservation
43.
44.   public void reserver() {
45.     ...
46.   }
47.
48.   public void supprimer() {
49.     try {
50.       // suppression d'un Rdv
51.       application.getMetier().supprimerRv(creneauChoisi.getRv());
52.       // on remet à jour l'agenda
53.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
54.       // on affiche form2
55.       setForms(false, true, false, false);
56.     } catch (Throwable th) {
57.       // vue erreurs
58.       prepareVueErreur(th);
59.     }
60.   }
```

- ligne 16 : lorsque la méthode *action* démarre, l'id du créneau horaire sélectionné a été posté dans *idCreneau* (ligne 11),
- lignes 18-26 : on cherche à récupérer le créneau horaire à partir de son *id* (ligne 21). On le cherche dans l'agenda courant, *agendaMedecinJour* de la ligne 10. Normalement on doit le trouver. Si ce n'est pas le cas, on ne fait rien (lignes 28-32),
- ligne 34 : si on a touvé le créneau cherché, on en récupère une référence qu'on stocke en ligne 12,
- ligne 36 : on regarde si le créneau choisi avait un rendez-vous. Si oui, on le supprime (ligne 39), sinon on en réserve un (ligne 37),
- ligne 51 : le rendez-vous du créneau choisi est supprimé. C'est la couche [métier] qui fait ce travail,
- ligne 53 : on demande à la couche [métier] le nouvel agenda du médecin. On va bien sûr y voir un rendez-vous de moins. Mais comme l'application est multi-utilisateurs, on peut y voir des modifications apportées par d'autres utilisateurs,
- ligne 55 : on réaffiche la page [form2.xhtml],
- ligne 58 : comme la couche [métier] a été sollicitée, des exceptions peuvent surgir. Dans ce cas, on mémorise la pile des exceptions dans la liste d'erreurs de la ligne 13 et on les affiche à l'aide de la vue [erreur.xhtml].

3.6.6.4 Prise de rendez-vous

La prise de rendez-vous correspond à la séquence suivante :



On réserve

On obtient une page à renseigner

Le modèle impliqué dans cette action est le suivant :

```
1.  // modèle
2.    private Date jour = new Date();
3.    private Boolean form1Rendered = true;
4.    private Boolean form2Rendered = false;
5.    private Boolean form3Rendered = false;
6.    private Boolean erreurRendered = false;
7.    private String form3Titre;
8.    private AgendaMedecinJour agendaMedecinJour;
9.    private Medecin medecin;
10.   private CreneauMedecinJour creneauChoisi;
11.   private List<Erreur> erreurs;
12.
13.   // action sur RV
14.   public void action() {
15. ...
16.     // on a trouvé
17.     creneauChoisi = agendaMedecinJour.getCreneauxMedecinJour()[i];
18.     // selon l'action désirée
19.     if (creneauChoisi.getRv() == null) {
20.       reserver();
21.     } else {
22.       supprimer();
23.     }
24.   }
25.   // réservation
26.
27.     public void reserver() {
28.     try {
29.       // titre formulaire 3
30.       form3Titre = Messages.getMessage(null, "form3.titre", new Object[]{medecin.getTitre(),
    medecin.getPrenom(), medecin.getNom(), new SimpleDateFormat("dd MMM yyyy").format(jour),
31.                 creneauChoisi.getCreneau().getHdebut(), creneauChoisi.getCreneau().getMdebut(),
    creneauChoisi.getCreneau().getHfin(), creneauChoisi.getCreneau().getMfin()}).getSummary();
32.       // client sélectionné dans le combo
33.       idClient=null;
34.       // on affiche le formulaire 3
35.       setForms(false, false, true, false);
36.     } catch (Throwable th) {
37.       // vue erreurs
38.       prepareVueErreur(th);
39.     }
40.   }
```

- ligne 14 : si le créneau choisi n'a pas de rendez-vous alors c'est une réservation,
- ligne 30 : on prépare le titre de la page [form3.xhtml] avec la même technique que celle utilisée pour le titre de la page [form2.xhtml],
- ligne 34 : dans ce formulaire, il y a un combo dont la valeur est alimentée par *idClient*. On met la valeur de ce champ à *null* pour ne sélectionner personne,
- ligne 36 : on affiche la page [form3.xhtml],
- ligne 39 : ou la page d'erreurs s'il y a eu une exception.

La page [form3.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.
8.    <body>
9.      <h2><h:outputText value="#{form.form3Titre}"/></h2>
10.     <h:panelGrid columns="2">
11.       <h:outputText value="#{msg['form3.client']}"/>
12.       <h:selectOneMenu value="#{form.idClient}">
13.         <f:selectItems value="#{form.clients}" var="client" itemLabel="#{client.titre} #{client.prenom}
    #{client.nom}" itemValue="#{client.id}"/>
14.       </h:selectOneMenu>
15.       <h:panelGroup>
16.         <h:commandButton value="#{msg['form3.valider']}" actionListener="#{form.validerRv}" />
```

```
17.         <h:commandButton value="#{msg['form3.annuler']}" actionListener="#{form.annulerRv}"/>
18.       </h:panelGroup>
19.     </h:panelGrid>
20.   </body>
21. </html>
```

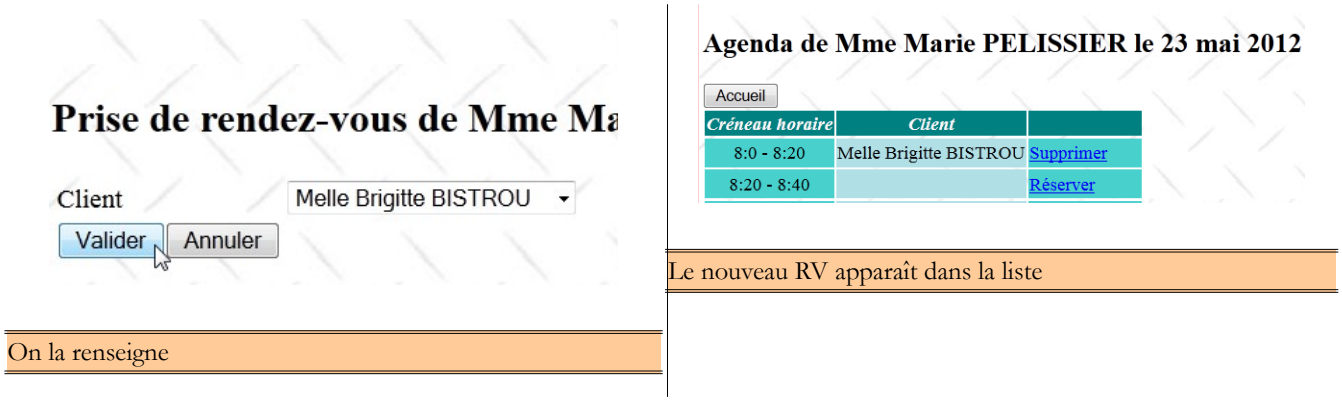
Cette page est alimentée par le modèle suivant :

```
1.  // bean Application
2.    @Inject
3.    private Application application;
4.
5.    // modèle
6.    private Long idClient;
7.
8.    // liste des clients
9.    public List<Client> getClients() {
10.     return application.getClients();
11.   }
```

- ligne 6 : le n° du client alimente l'attribut *value* du combo des clients de la ligne 12 de la page. Il fixe l'élément du combo sélectionné,
- lignes 9-11 : la méthode *getClients* alimente le contenu du combo (ligne 13). Le libellé (*itemLabel*) de chaque option est [Titre Prénom Nom] du client, et la valeur associée (*itemValue*) est l'**id** du client. C'est donc cette valeur qui sera postée.

3.6.6.5 Validation d'un rendez-vous

La validation d'un rendez-vous correspond à la séquence suivante :



et correspond au clic sur le bouton [Valider] :

```
<h:commandButton value="#{msg['form3.valider']}" actionListener="#{form.validerRv}" />
```

C'est donc la méthode [Form].validerRv qui va gérer cet évènement. Son code est le suivant :

```
1.    // bean Application
2.    @Inject
3.    private Application application;
4.
5.    // modèle
6.    private Date jour = new Date();
7.    private Boolean form1Rendered = true;
8.    private Boolean form2Rendered = false;
9.    private Boolean form3Rendered = false;
10.   private Boolean erreurRendered = false;
11.   private Long idCreneau;
12.   private Long idClient;
13.   private List<Erreur> erreurs;
14.
```

```
15.   // validation Rv
16.   public void validerRv() {
17.     try {
18.       // on récupère une instance du créneau horaire choisi
19.       Creneau creneau = application.getMetier().getCreneauById(idCreneau);
20.       // on ajoute le Rv
21.       application.getMetier().ajouterRv(jour, creneau, application.gethClients().get(idClient));
22.       // on remet à jour l'agenda
23.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
24.       // on affiche form2
25.       setForms(false, true, false, false);
26.     } catch (Throwable th) {
27.       // vue erreurs
28.       prepareVueErreur(th);
29.     }
30. }
```

- ligne 12 : avant que la méthode **validerRv** ne s'exécute, le champ **idClient** a reçu l'**id** du client sélectionné par l'utilisateur,
- ligne 19 : à partir de l'**id** du créneau horaire mémorisé dans une précédente étape (le bean est de portée **session**), on demande à la couche [métier] une référence sur le créneau horaire lui-même,
- ligne 21 : on demande à la couche [métier] d'ajouter un rendez-vous pour le jour choisi (jour), le créneau horaire choisi (creneau) et le client choisi (idClient),
- ligne 23 : on demande à la couche [métier] de rafraîchir l'agenda du médecin. On verra le rendez-vous ajouté plus toutes les modifications que d'autres utilisateurs de l'application ont pu faire,
- ligne 25 : on réaffiche l'agenda [form2.xhtml],
- ligne 28 : on affiche la page d'erreur si une erreur se produit.

3.6.6.6 *Annulation d'une prise de rendez-vous*

Cela correspond à la séquence suivante :

On annule la prise de rendez-vous

On retrouve l'agenda

Le bouton [Annuler] dans la page [form3.xhtml] est le suivant :

```
<h:commandButton value="#{msg['form3.annuler']}" actionListener="#{form.annulerRv}"/>
```

La méthode *[Form].annulerRv* est donc appelée :

```
1.    // annulation prise de Rdv
2.    public void annulerRv() {
3.      // on affiche form2
4.      setForms(false, true, false, false);
5.  }
```

3.6.6.7 Retour à la page d'accueil

Il reste une action à voir, celle de la séquence suivante :

Une fois l'opération de prise de RV ou d'annulation de RV faite, l'utilisateur peut revenir à la page d'accueil

Il la retrouve dans son dernier état

Le code du bouton [Accueil] dans la page [form2.xhtml] est le suivant :

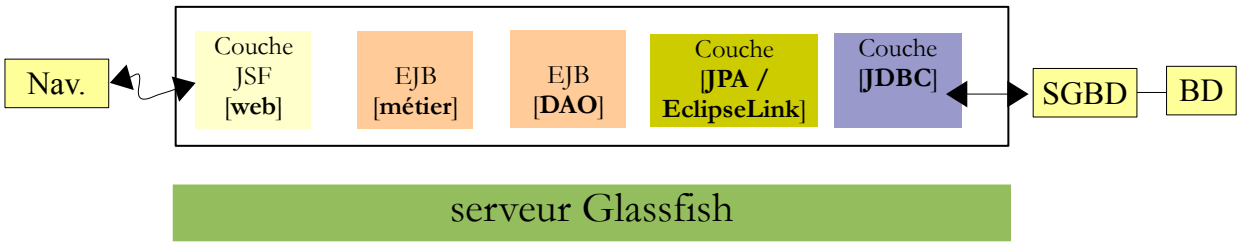
```
<h:commandButton value="#{msg['form2.accueil']}" action="#{form.accueil}" />
```

La méthode *[Form].accueil* est la suivante :

```
1.    public void accueil() {
2.      // on affiche la page d'accueil
3.      setForms(true, false, false, false);
4.  }
```

3.7 Conclusion

Nous avons construit l'application suivante :

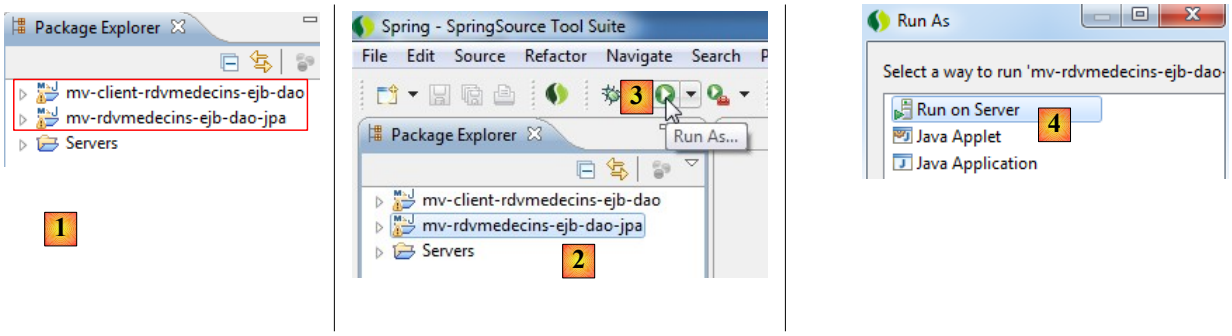


Nous nous sommes intéressés aux fonctionnalités de l'application plus qu'à son aspect pour l'utilisateur. Celui-ci sera amélioré avec l'utilisation de la bibliothèque de composants **PrimeFaces**. Nous avons construit une application basique mais néanmoins représentative d'une architecture Java EE en couches utilisant des EJB. L'application peut être améliorée de diverses façons :

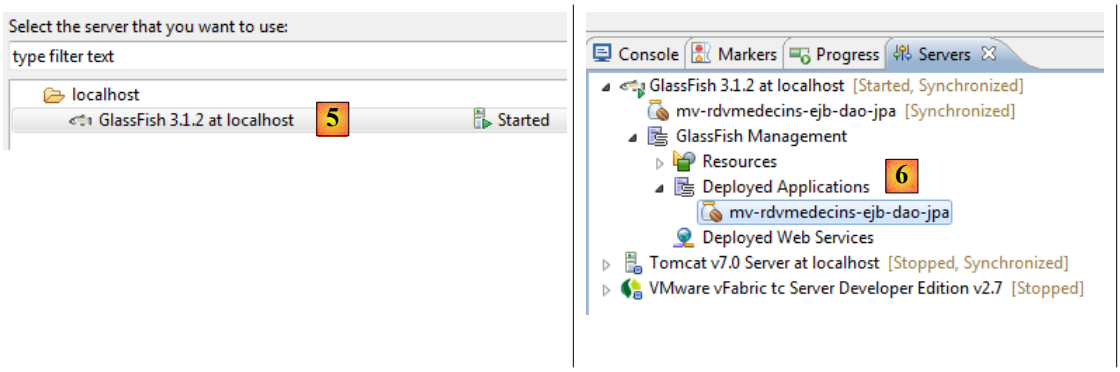
- une authentification est nécessaire. Tout le monde n'est pas autorisé à ajouter / supprimer des rendez-vous,
- on devrait pouvoir faire défiler l'agenda en avant et en arrière lorsqu'on cherche un jour avec des créneaux libres,
- on devrait pouvoir demander la liste des jours où il existe des créneaux libres pour un médecin. En effet, si celui-ci est ophtalmologue ses rendez-vous sont généralement pris six mois à l'avance,
- ...

3.8 Les tests avec Eclipse

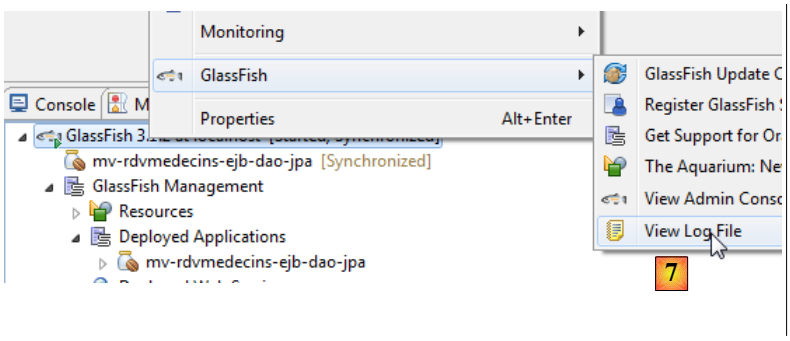
3.8.1 La couche [DAO]



- en [1], on importe le projet EJB de la couche [DAO] et son client,
- en [2], on sélectionne le projet EJB de la couche [DAO] et on l'exécute [3],
- en [4], on l'exécute sur un serveur,



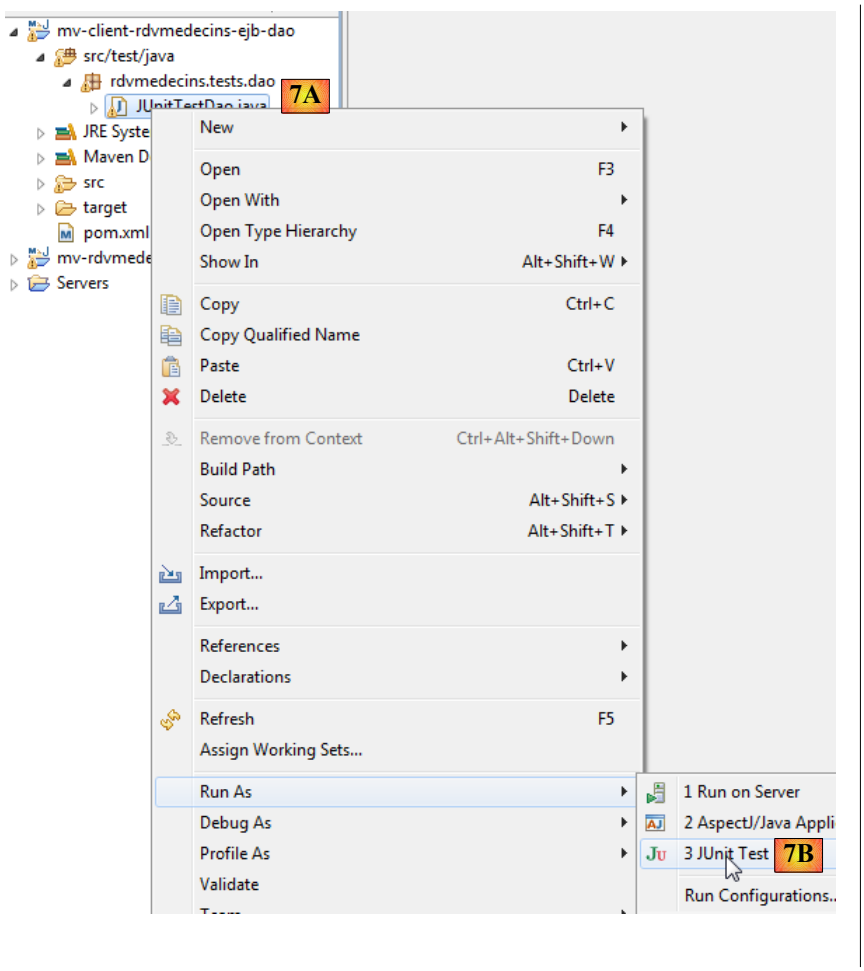
- en [5], seul le serveur Glassfish est proposé car c'est le seul ayant un conteneur EJB,
- en [6], le module EJB a été déployé,



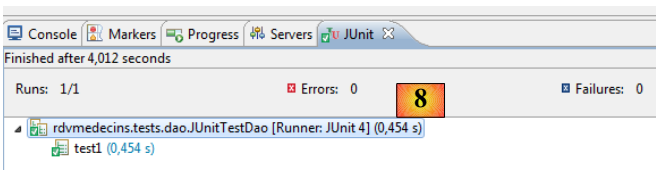
- en [7], on affiche les logs :

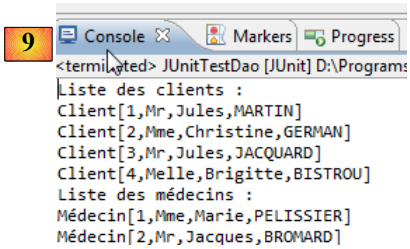
```
1. Infos: Hibernate Validator 4.2.0.Final
2.
3. Infos: Created EjbThreadPoolExecutor with thread-core-pool-size 16 thread-max-pool-size 32 thread-keep-
   alive-seconds 60 thread-queue-capacity 2147483647 allow-core-thread-timeout false
4. ...
5.
6. Infos: EJB5181:Portable JNDI names for EJB DaoJpa: [java:global/mv-rdvmedecins-ejb-dao-jpa/DaoJpa!
   rdvmedecins.dao.IDaoRemote, java:global/mv-rdvmedecins-ejb-dao-jpa/DaoJpa!rdvmedecins.dao.IDaoLocal]
7. Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB DaoJpa:
   [rdvmedecins.dao#rdvmedecins.dao.IDaoRemote, rdvmedecins.dao]
8. Infos: mv-rdvmedecins-ejb-dao-jpa a été déployé en 5 523 ms.
```

Ce sont ceux qu'on avait avec Netbeans.

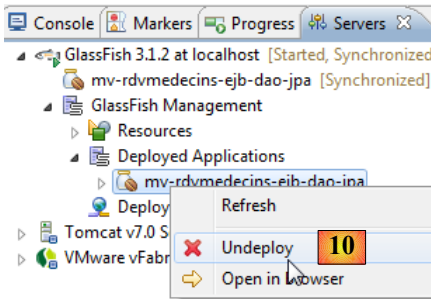


- en [7A] [7B] on exécute le test JUnit du client,



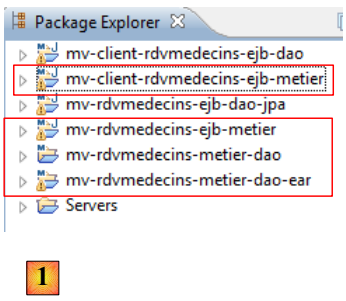


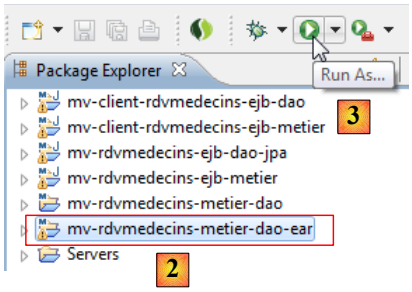
- en [8], le test est réussi,
- en [9], les logs de la console.

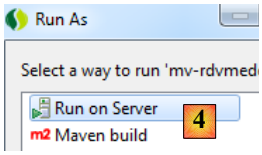


En [10], on décharge l'application EJB.

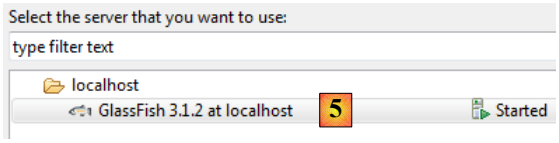
3.8.2 La couche [métier]

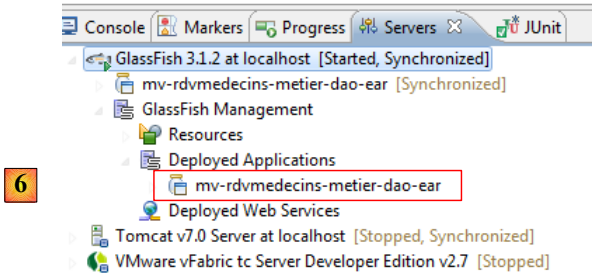




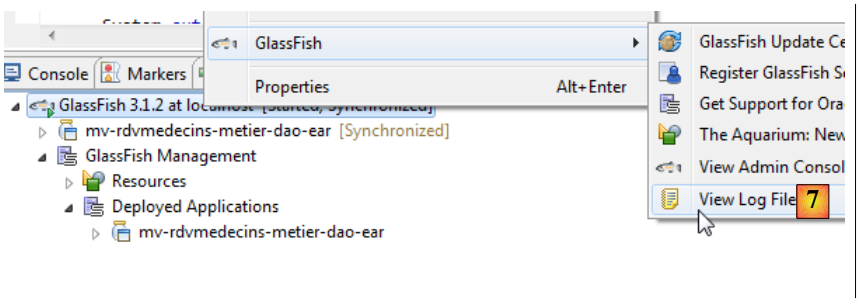


- en [1], on importe les quatre projets Maven de la couche [métier],
- en [2], on sélectionne le projet d'entreprise et on l'exécute en [3], sur un serveur Glassfish [4] [5],





- en [6], le projet d'entreprise a été déployé sur Glassfish,

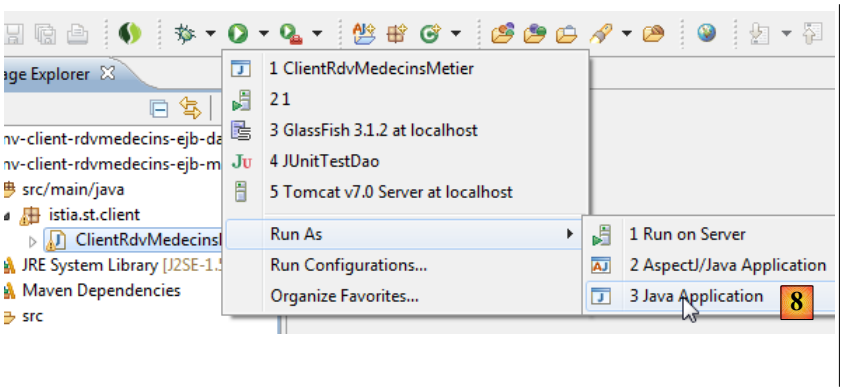


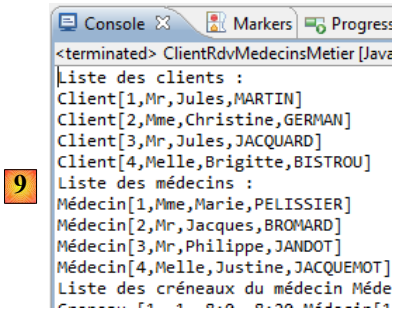
- en [7], on regarde les logs de Glassfish,

```
1. Infos: EJB5181:Portable JNDI names for EJB DaoJpa: [java:global/mv-rdvmedecins-metier-dao-ear/mv-
   rdvmedecins-ejb-dao-jpa-1.0-SNAPSHOT/DaoJpa!rdvmedecins.dao.IDaoLocal, java:global/mv-rdvmedecins-metier-
   dao-ear/mv-rdvmedecins-ejb-dao-jpa-1.0-SNAPSHOT/DaoJpa!rdvmedecins.dao.IDaoRemote]
2. Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB DaoJpa:
   [rdvmedecins.dao#rdvmedecins.dao.IDaoRemote, rdvmedecins.dao]
3. Infos: EJB5181:Portable JNDI names for EJB Metier: [java:global/mv-rdvmedecins-metier-dao-ear/mv-
   rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!rdvmedecins.metier.service.IMetierLocal, java:global/mv-
   rdvmedecins-metier-dao-ear/mv-rdvmedecins-ejb-metier-1.0-SNAPSHOT/Metier!
   rdvmedecins.metier.service.IMetierRemote]
4. Infos: EJB5182:Glassfish-specific (Non-portable) JNDI names for EJB Metier:
   [rdvmedecins.metier.service.IMetierRemote#rdvmedecins.metier.service.IMetierRemote,
   rdvmedecins.metier.service.IMetierRemote]
```

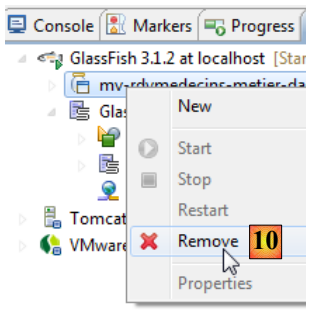
Ligne 3, nous notons le nom portable de l'EJB [Metier] et nous le collons dans le client console de cet EJB :

```
1. public class ClientRdvMedecinsMetier {
2.
3.   // le nom de l'interface distante de l'EJB [Metier]
4.   private static String IDaoRemoteName = "java:global/mv-rdvmedecins-metier-dao-ear/mv-rdvmedecins-ejb-
   metier-1.0-SNAPSHOT/Metier!rdvmedecins.metier.service.IMetierRemote";
5.   // date du jour
6. private static Date jour = new Date();
```



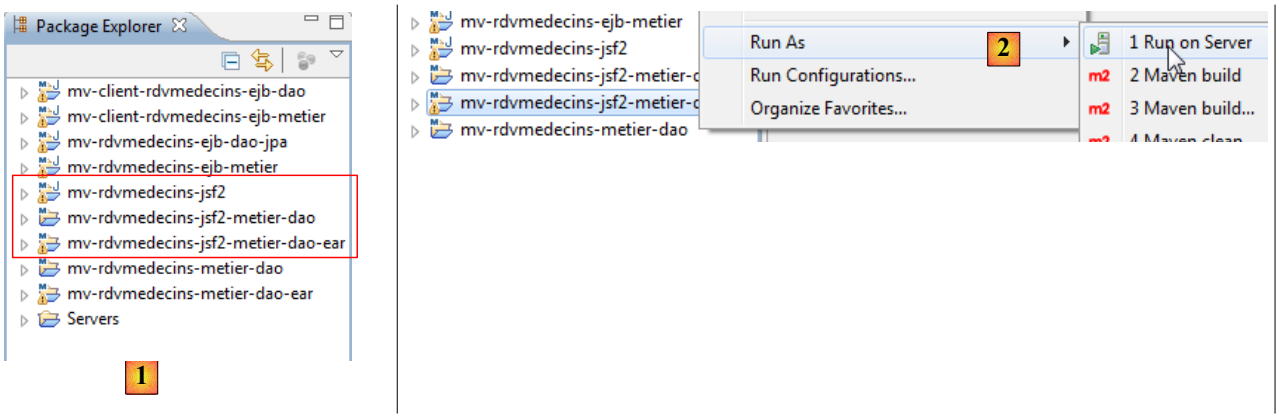


- en [8], on exécute le client console,
- en [9], ses logs.

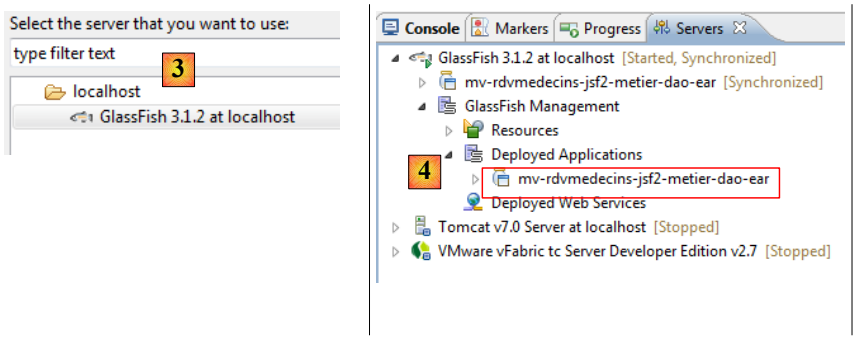


- en [10], on décharge l'application d'entreprise ;

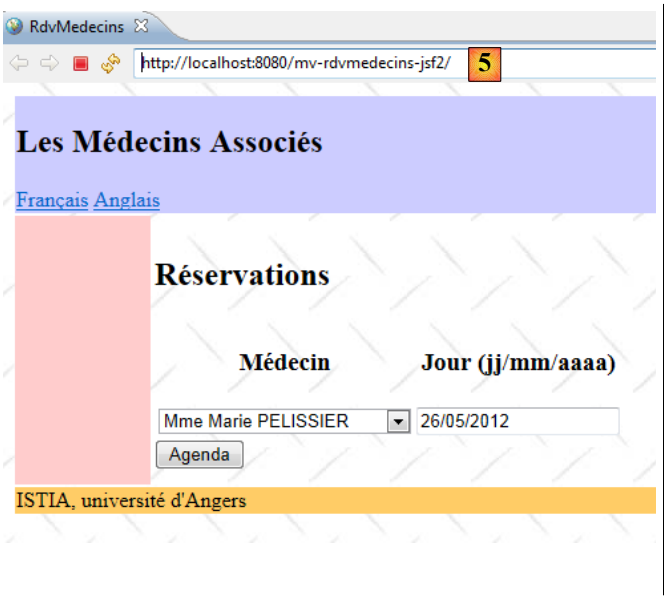
3.8.3 La couche [web]



- en [1], on importe les trois projets Maven de la couche [web]. Celui suffixé par **ear** est le projet d'entreprise qu'il faut déployer sur Glassfish,
- en [2], on l'exécute,



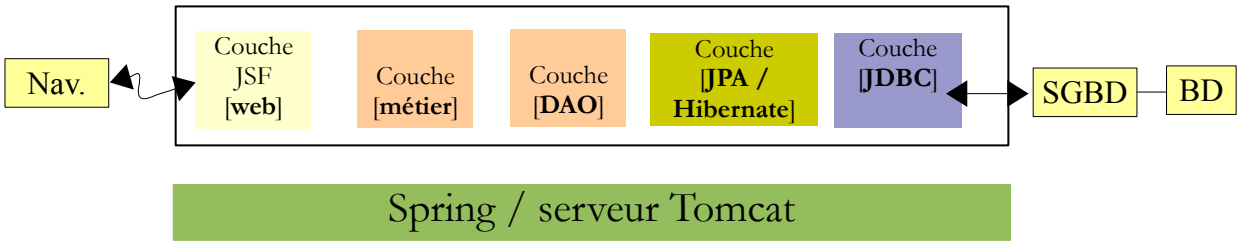
- sur le serveur Glassfish [3],
- en [4], l'application d'entreprise a bien été déployée,



- en [5], on demande l'URL de l'application dans le navigateur interne d'Eclipse.

4 Application exemple – 02 : rdvmedecins-jsf2-spring

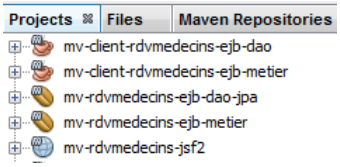
Nous nous proposons maintenant de porter l'application précédente dans un environnement Spring / Tomcat :



Il s'agit réellement d'un portage. Nous allons partir de l'application précédente et l'adapter au nouvel environnement. Nous ne commenterons que les modifications. Elles sont de trois ordres :

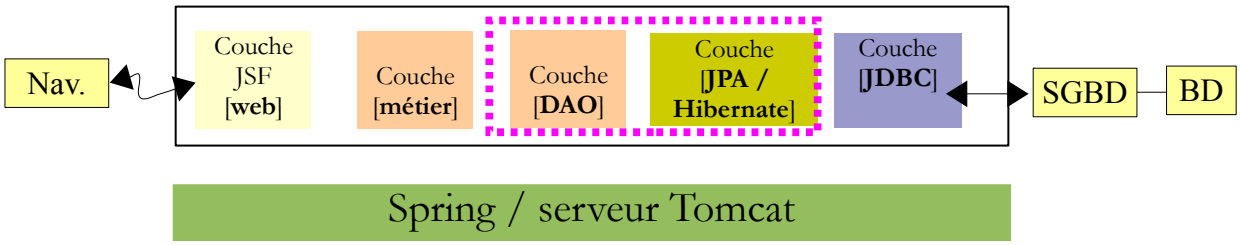
- le serveur n'est plus Glassfish mais Tomcat, un serveur léger qui n'a pas de conteneur EJB,
- pour remplacer les EJB, on utilisera Spring, le concurrent principal des EJB [http://www.springsource.com/],
- l'implémentation JPA utilisée sera Hibernate à la place d'EclipseLink.

Parce que nous allons beaucoup procéder par copier / coller entre l'ancien projet et le nouveau, nous gardons les précédents projets ouverts dans Netbeans :



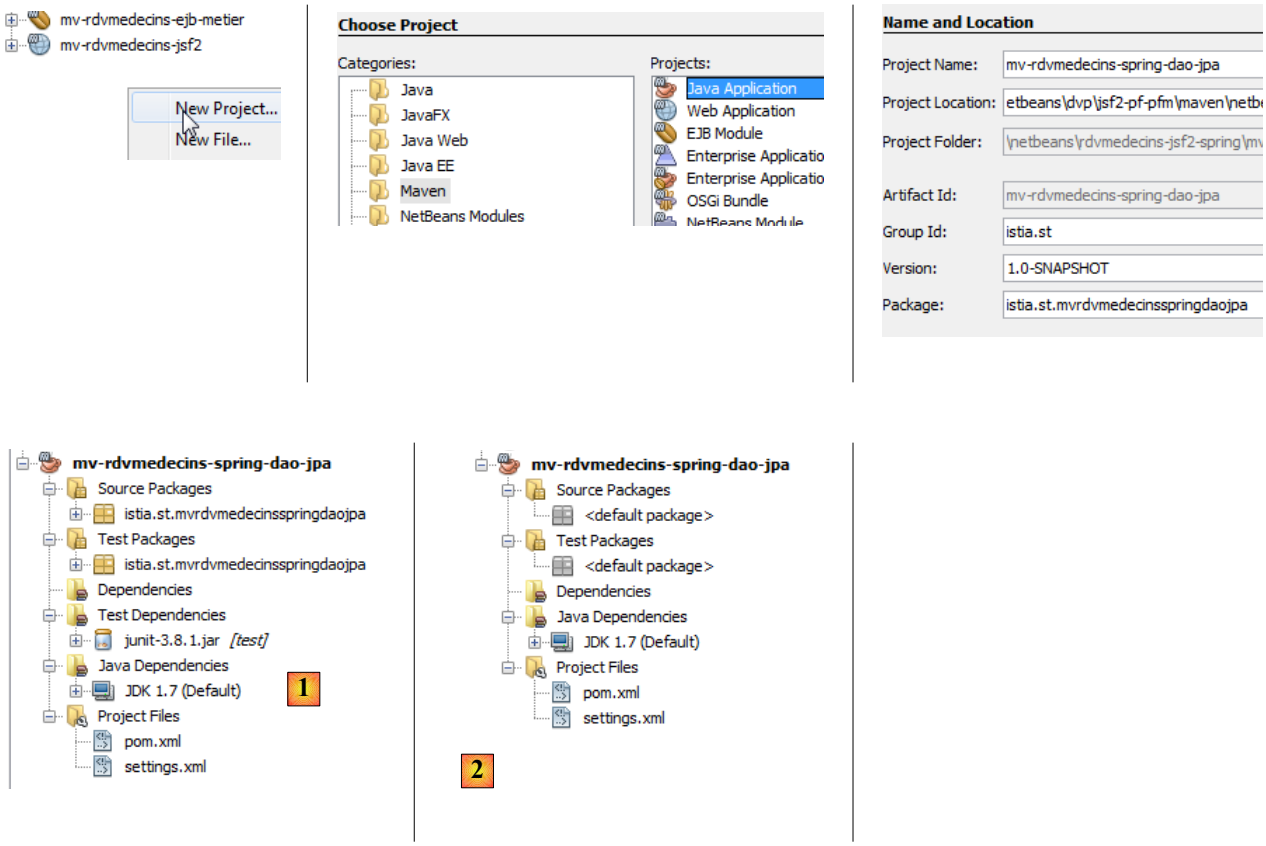
L'utilisation du framework Spring nécessite certaines connaissances qu'on trouvera dans [**ref7**] (cf page 154).

4.1 Les couches [DAO] et [JPA]



4.1.1 Le projet Netbeans

Nous construisons un projet Maven de type [Java Application] :



- en [1], le projet créé,
- en [2], le même débarrassé des packages de [Source Packages] et [Test Packages] et de la dépendance [junit-3.8.1].

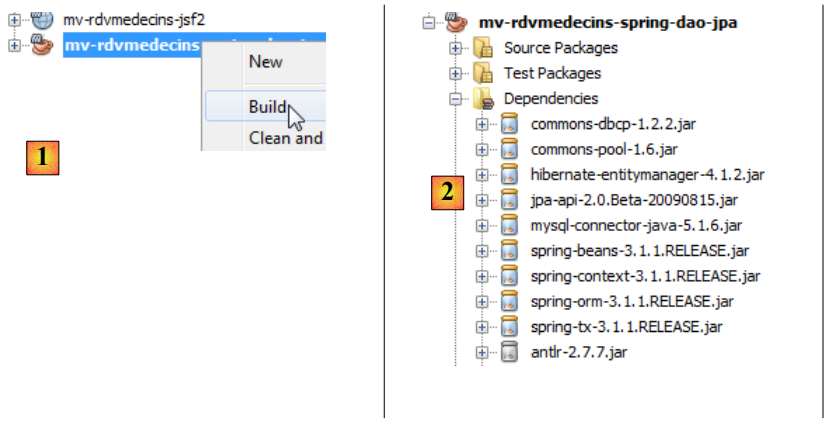
Le plus difficile dans les projets Maven est de trouver les bonnes dépendances. Pour ce projet Spring / JPA / Hibernate, ce sont les suivantes :

```
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.
5.    <groupId>istia.st</groupId>
6.    <artifactId>mv-rdvmedecins-spring-dao-jpa</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>mv-rdvmedecins-spring-dao-jpa</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <properties>
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.     <dependencies>
18.     <dependency>
19.       <groupId>org.hibernate</groupId>
20.       <artifactId>hibernate-entitymanager</artifactId>
21.       <version>4.1.2</version>
22.       <type>jar</type>
23.     </dependency>
24.     <dependency>
25.       <groupId>org.hibernate.java-persistence</groupId>
26.       <artifactId>jpa-api</artifactId>
27.       <version>2.0.Beta-20090815</version>
28.       <type>jar</type>
29.     </dependency>
30.     <dependency>
31.       <groupId>mysql</groupId>
32.       <artifactId>mysql-connector-java</artifactId>
```

```
33.       <version>5.1.6</version>
34.     </dependency>
35.     <dependency>
36.       <groupId>junit</groupId>
37.       <artifactId>junit</artifactId>
38.       <version>4.10</version>
39.       <scope>test</scope>
40.       <type>jar</type>
41.     </dependency>
42.     <dependency>
43.       <groupId>commons-dbcp</groupId>
44.       <artifactId>commons-dbcp</artifactId>
45.       <version>1.2.2</version>
46.     </dependency>
47.     <dependency>
48.       <groupId>commons-pool</groupId>
49.       <artifactId>commons-pool</artifactId>
50.       <version>1.6</version>
51.     </dependency>
52.     <dependency>
53.       <groupId>org.springframework</groupId>
54.       <artifactId>spring-tx</artifactId>
55.       <version>3.1.1.RELEASE</version>
56.       <type>jar</type>
57.     </dependency>
58.     <dependency>
59.       <groupId>org.springframework</groupId>
60.       <artifactId>spring-beans</artifactId>
61.       <version>3.1.1.RELEASE</version>
62.       <type>jar</type>
63.     </dependency>
64.     <dependency>
65.       <groupId>org.springframework</groupId>
66.       <artifactId>spring-context</artifactId>
67.       <version>3.1.1.RELEASE</version>
68.       <type>jar</type>
69.     </dependency>
70.     <dependency>
71.       <groupId>org.springframework</groupId>
72.       <artifactId>spring-orm</artifactId>
73.       <version>3.1.1.RELEASE</version>
74.       <type>jar</type>
75.     </dependency>
76.   </dependencies>
77.
78. </project>
```

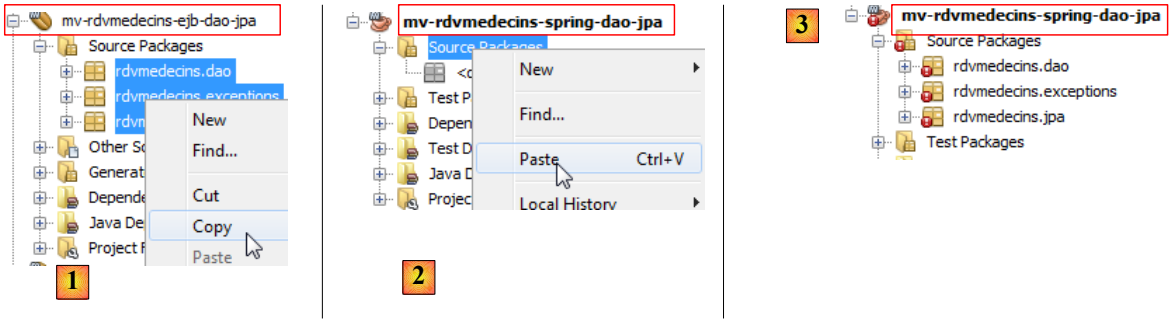
- lignes 18-29 : pour Hibernate,
- lignes 30-34 : pour le pilote JDBC de MySQL,
- lignes 35-41 : pour le test JUnit,
- lignes 42-51 : pour le pool de connexions Apache Commons DBCP. Un pool de connexions est un pool de connexions ouvertes. Lorsque l'application a besoin d'une connexion, il la demande au pool. Lorsqu'il n'en a plus besoin il la rend. Les connexions sont ouvertes au démarrage de l'application et le restent durant de la vie de l'application. Cela évite le coût d'ouvertures / fermetures répétées des connexions. Ce type de pool existait dans Glassfish mais son utilisation a été transparente pour nous. Ce sera également le cas ici mais il nous faut l'installer et le configurer,
- lignes 52-75 : pour Spring.

Ajoutons ces dépendances et construisons le projet :



- en [1], on construit le projet ce qui va forcer Maven à télécharger les dépendances,
- en [2], celles-ci apparaissent alors dans la branche [Dependencies]. Elles sont très nombreuses, les frameworks Hibernate et Spring ayant eux-mêmes de très nombreuses dépendances. Là encore, grâce à Maven, nous n'avons pas à nous préoccuper de ces dernières. Elles sont téléchargées automatiquement.

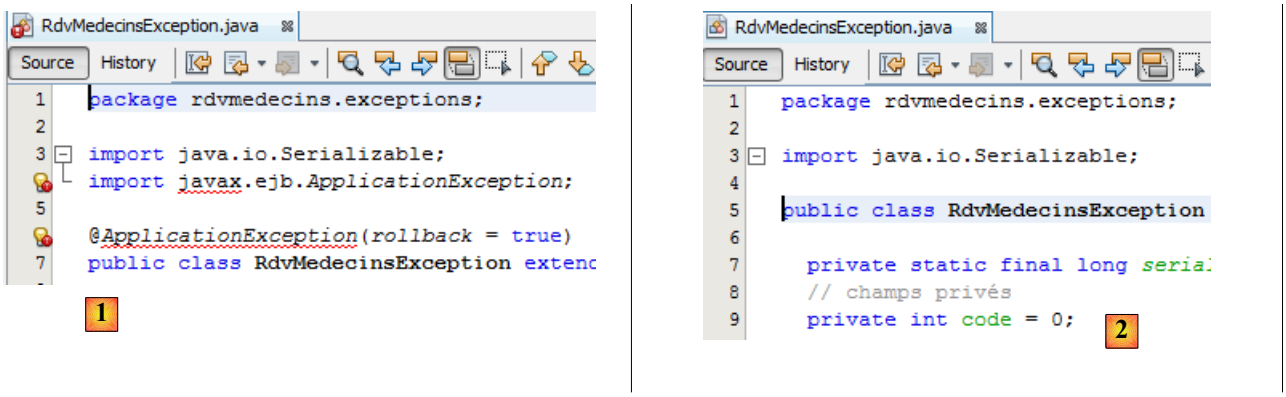
Maintenant que nous avons les dépendances, nous collons le code du projet EJB de la couche [dao] dans le projet Spring de la couche [dao] :



- en [1], on copie dans le projet source,
- en [2], on colle dans le projet destination,
- en [3], le résultat.

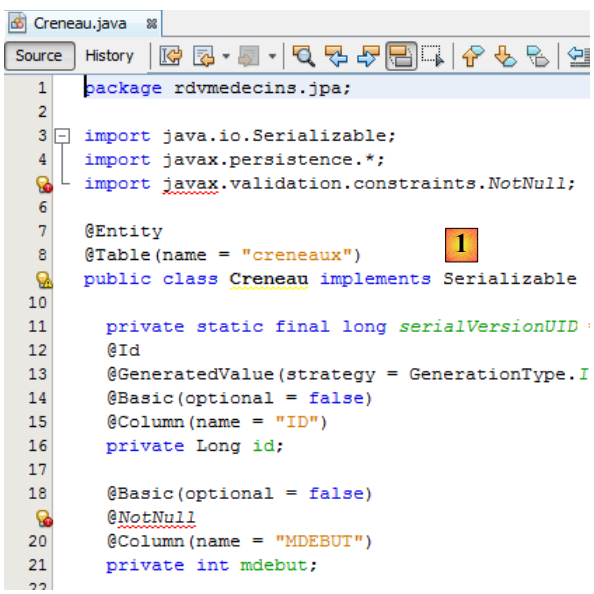
Une fois la copie effectuée, il faut corriger les erreurs.

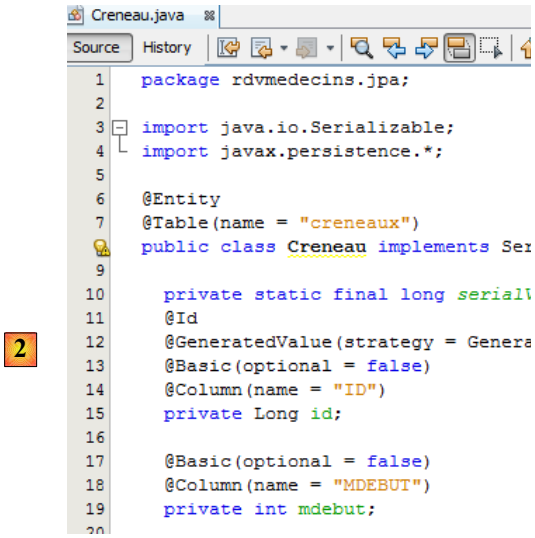
4.1.2 Le paquetage [exceptions]



La classe [RdvMedecinsExceptions] [1] a des erreurs à cause du paquetage [javax] ligne 4 qui n'existe plus. C'est un paquetage propre aux EJB. L'erreur de la ligne 6 dérive de celle de la ligne 4. On supprime ces deux lignes. Cela supprime les erreurs [2].

4.1.3 Le paquetage [jpa]

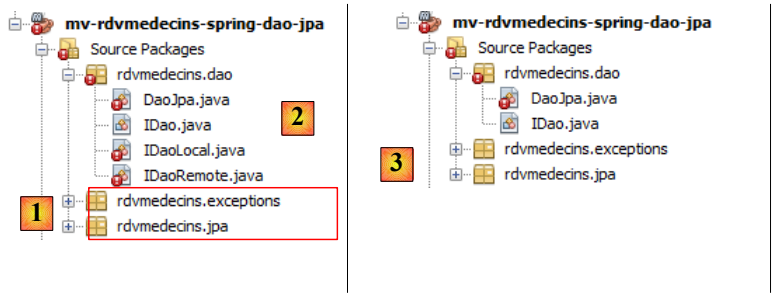




- en [1], la classe [Creneau] est erronée à cause de l'absence du paquetage de validation de la ligne [5]. On aurait pu ajouter ce paquetage dans les dépendances du projet. Mais aux tests, Hibernate lance une exception à cause de lui. Comme il n'est pas indispensable à notre application, nous l'avons enlevé. Pour corriger la classe, il suffit de supprimer toutes les lignes erronées [2]. On fait cela pour toutes les classes erronées.

4.1.4 Le paquetage [dao]

Nous en sommes au point suivant :



- en [1], les deux paquetages corrigés,
- en [2], le paquetage [dao]. Comme il n'y a plus d'EJB, il n'y a plus non plus la notion d'interface distante et locale de l'EJB. Nous les supprimons [3].

DaoJpa.java

Source | History

```
1  package rdvmedecins.dao;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import java.util.List;
6  import javax.ejb.Singleton;
7  import javax.ejb.TransactionAttribute;
8  import javax.ejb.TransactionAttributeType;
9  import javax.persistence.EntityManager;
10 import javax.persistence.PersistenceContext;
11 import rdvmedecins.exceptions.RdvMedecinsException;
12 import rdvmedecins.jpa.Client;
13 import rdvmedecins.jpa.Creneau;
14 import rdvmedecins.jpa.Medecin;
15 import rdvmedecins.jpa.Rv;
16
17 @Singleton(mappedName = "rdvmedecins.dao")
18 @TransactionAttribute(TransactionAttributeType.REQUIRED)
19 public class DaoJpa implements IDaoLocal, IDaoRemote, Serializable {
20
```

1

- en [1], les erreurs de la classe [DaoJpa] ont deux origines :
  - l'importation d'un paquetage lié aux EJB (lignes 6-8) ;
  - l'utilisation des interfaces locale et distante que nous venons de supprimer.

Nous supprimons les lignes erronées et utilisons l'interface [IDao] à la place des interfaces locale et distante [2].

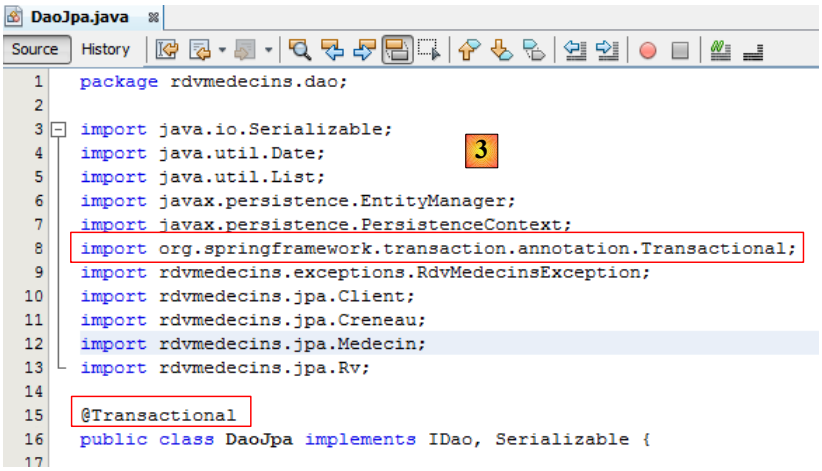
DaoJpa.java

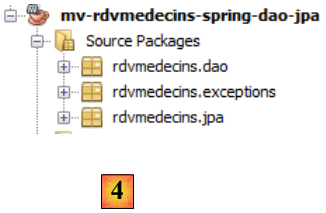
Source | History

```
1  package rdvmedecins.dao;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import java.util.List;
6  import javax.persistence.EntityManager;
7  import javax.persistence.PersistenceContext;
8  import rdvmedecins.exceptions.RdvMedecinsException;
9  import rdvmedecins.jpa.Client;
10 import rdvmedecins.jpa.Creneau;
11 import rdvmedecins.jpa.Medecin;
12 import rdvmedecins.jpa.Rv;
13
14 public class DaoJpa implements IDao, Serializable {
15
16   private static final long serialVersionUID = 1L;
17   @PersistenceContext
18   private EntityManager em;
```

2

Dans le projet EJB, la classe [DaoJpa] était un singleton et ses méthodes s'exécutaient au sein d'une transaction. Nous verrons que la classe [DaoJpa] va être un bean géré par Spring. Par défaut tout bean Spring est un singleton. Voilà pour la première propriété. La seconde est obtenue avec l'annotation **@Transactional** de Spring [3] :



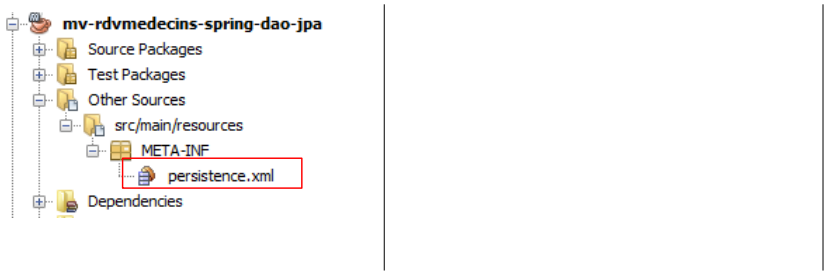


Ceci fait, le projet ne présente plus d'erreurs [4].

4.1.5 Configuration de la couche [JPA]

Dans le projet EJB, nous avions configuré la couche [JPA] avec le fichier [persistence.xml]. Nous avons ici une couche [JPA] et donc nous devons créer ce fichier. Dans le projet EJB, nous l'avions généré avec Glassfish. Ici, nous le construisons à la main. La raison principale en est qu'une partie de la configuration du fichier [persistence.xml] migre dans le fichier de configuration de Spring lui-même.

Nous créons le fichier [persistence.xml] :



avec le contenu suivant :

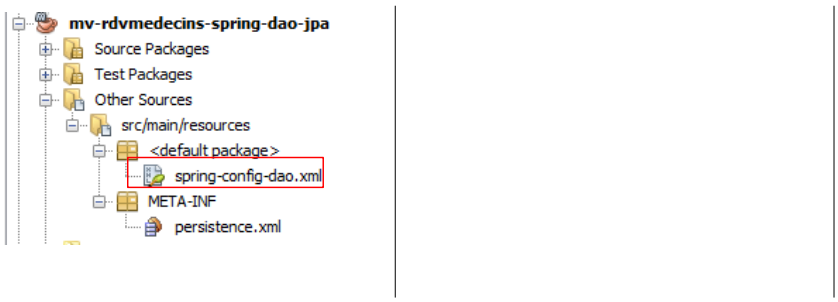
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
3.    <persistence-unit name="spring-dao-jpa-hibernate-mysqlPU" transaction-type="RESOURCE_LOCAL">
4.      <class>rdvmedecins.jpa.Client</class>
5.      <class>rdvmedecins.jpa.Creneau</class>
6.      <class>rdvmedecins.jpa.Medecin</class>
7.      <class>rdvmedecins.jpa.Rv</class>
8.    </persistence-unit>
9.  </persistence>
```

- ligne 3 : on donne un nom à l'unité de persistance,
- ligne 3 : le type des transactions est RESOURCE_LOCAL. Dans le projet EJB, c'était JTA pour indiquer que les transactions étaient gérées par le conteneur EJB. La valeur RESOURCE_LOCAL indique que l'application gère elle-même ses transactions. Ce sera le cas ici au travers de Spring,
- lignes 4-7 : les noms complets des quatre entités JPA. C'est facultatif car Hibernate les cherche automatiquement dans le ClassPath du projet.

C'est tout. Le nom du provider JPA, ses propriétés, les caractéristiques JDBC de la source de données sont désormais dans le fichier de configuration de Spring.

4.1.6 Le fichier de configuration de Spring

Nous avons dit que la classe [DaoJpa] était un bean géré par Spring. Cela se fait au moyen d'un fichier de configuration. Ce fichier va comporter également la configuration de l'accès à la base de données ainsi que la gestion des transactions. Il doit être dans le *ClassPath* du projet. Nous le mettons dans la branche [Other sources] :



Le fichier [spring-config-dao.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
3.         xmlns:tx="http://www.springframework.org/schema/tx"
4.         xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
5.
6.    <!-- couches applicatives -->
7.    <bean id="dao" class="" />
8.
9.    <!-- EntityManagerFactory -->
10.   <bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
11.     <property name="dataSource" ref="dataSource" />
12.     <property name="jpaVendorAdapter">
13.       <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
14.         <property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15.         <!--
16.         <property name="showSql" value="true" />
17.         <property name="generateDdl" value="true" />
18.         -->
19.       </bean>
20.     </property>
21.   </bean>
22.
23.   <!-- la source de donnéees DBCP -->
24.   <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
25.     <property name="driverClassName" value="com.mysql.jdbc.Driver" />
26.     <property name="url" value="jdbc:mysql://localhost:3306/dbrdvmedecins2" />
27.     <property name="username" value="root" />
28.     <property name="password" value="" />
29.   </bean>
30.
31.   <!-- le gestionnaire de transactions -->
32.   <tx:annotation-driven transaction-manager="txManager" />
33.   <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
34.     <property name="entityManagerFactory" ref="entityManagerFactory" />
35.   </bean>
36.
37.   <!-- traduction des exceptions -->
38.   <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
39.
40.   <!-- persistence -->
41.   <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
42.
43. </beans>
```

C'est un fichier compatible Spring 2.x. Nous n'avons pas cherché à utiliser les nouvelles caractéristiques des versions 3.x.

- lignes 2-4 : la balise racine <beans> du fichier de configuration. Nous ne commentons pas les divers attributs de cette balise. On prendra soin de faire un copier / coller parce que se tromper dans l'un de ces attributs provoque des erreurs parfois difficiles à comprendre,
- ligne 7 : le bean "dao" est une référence sur une instance de la classe [`rdvmedecins.dao.DaoJpa`]. Une instance unique sera créée (singleton) et implémentera la couche [dao] de l'application,
- lignes 24-29 : une source de données est définie. Elle fournit le service de "pool de connexions" dont nous avons parlé. C'est [DBCP] du projet *Apache commons DBCP* [http://jakarta.apache.org/commons/dbcp/] qui est ici utilisé,
- lignes 25-28 : pour créer des connexions avec la base de données cible, la source de données a besoin de connaître le pilote JDBC utilisé (ligne 25), l'URL de la base de données (ligne 26), l'utilisateur de la connexion et son mot de passe (lignes 27-28),
- lignes 10-21 : configurent la couche JPA,
- ligne 10 : définit un bean de type [EntityManagerFactory] capable de créer des objets de type [EntityManager] pour gérer les contextes de persistance. La classe instanciée [LocalContainerEntityManagerFactoryBean] est fournie par Spring. Elle a besoin d'un certain nombre de paramètres pour s'instancier, définis lignes 11-20,
- ligne 11 : la source de données à utiliser pour obtenir des connexions au SGBD. C'est la source [DBCP] définie aux lignes 24-29,
- lignes 12-20 : l'implémentation JPA à utiliser,
- ligne 13 : définit Hibernate comme implémentation JPA à utiliser,
- ligne 14 : le dialecte SQL qu'Hibernate doit utiliser avec le SGBD cible, ici MySQL5,
- ligne 16 (en commentaires) : demande que les ordres SQL exécutés par Hibernate soient logués sur la console,
- ligne 17 (en commentaires) : demande qu'au démarrage de l'application, la base de données soit générée (drop et create),
- ligne 32 : indique que les transactions sont gérées avec des annotations Java (elles auraient pu être également déclarées dans *spring-config.xml*). C'est en particulier l'annotation **@Transactional** rencontrée dans la classe [DaoJpa],
- lignes 33-35 : définissent le gestionnaire de transactions à utiliser,
- ligne 33 : le gestionnaire de transactions est une classe fournie par Spring,
- ligne 34 : le gestionnaire de transactions de Spring a besoin de connaître l'*EntityManagerFactory* qui gère la couche JPA. C'est celui défini aux lignes 10-21,
- ligne 41 : définit la classe qui gère les annotations de persistance Spring,
- ligne 38 : définissent la classe Spring qui gère notamment l'annotation **@Repository** qui rend une classe ainsi annotée, éligible pour la traduction des exceptions natives du pilote JDBC du SGBD en exceptions génériques Spring de type [DataAccessException]. Cette traduction encapsule l'exception JDBC native dans un type [DataAccessException] ayant diverses sous-classes :

Direct Known Subclasses:
CleanupFailureDataAccessException, ConcurrencyFailureException, DataAccessResourceFailureException, DataIntegrityViolationException, DataRetrievalFailureException, DataSourceLookupFailureException, InvalidDataAccessApiUsageException, InvalidDataAccessResourceUsageException, PermissionDeniedDataAccessException, UncategorizedDataAccessException

Cette traduction permet au programme client de gérer les exceptions de façon générique quelque soit le SGBD cible. Nous n'avons pas utilisé l'annotation **@Repository** dans notre code Java. Aussi la ligne 38 est-elle inutile. Nous l'avons laissée par simple souci d'information.

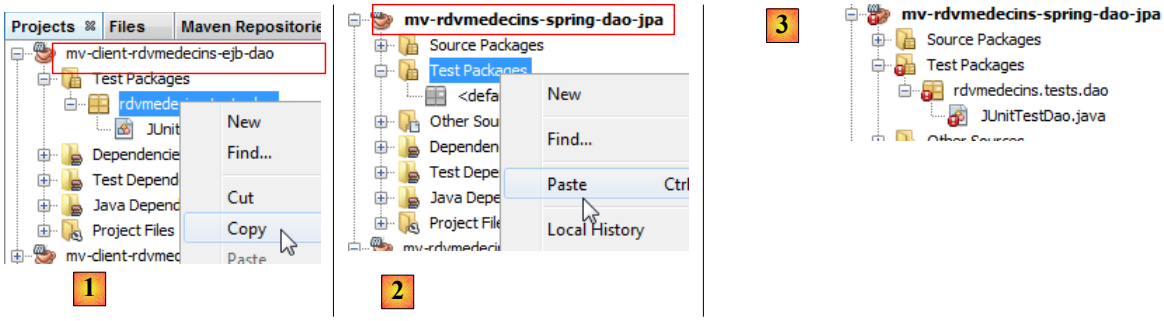
Nous en avons fini avec le fichier de configuration de Spring. Il a été tiré de la documentation Spring. Son adaptation à diverses situations se résume souvent à deux modifications :

- celle de la base de données cible : lignes 24-29,
- celle de l'implémentation JPA : lignes 12-20.

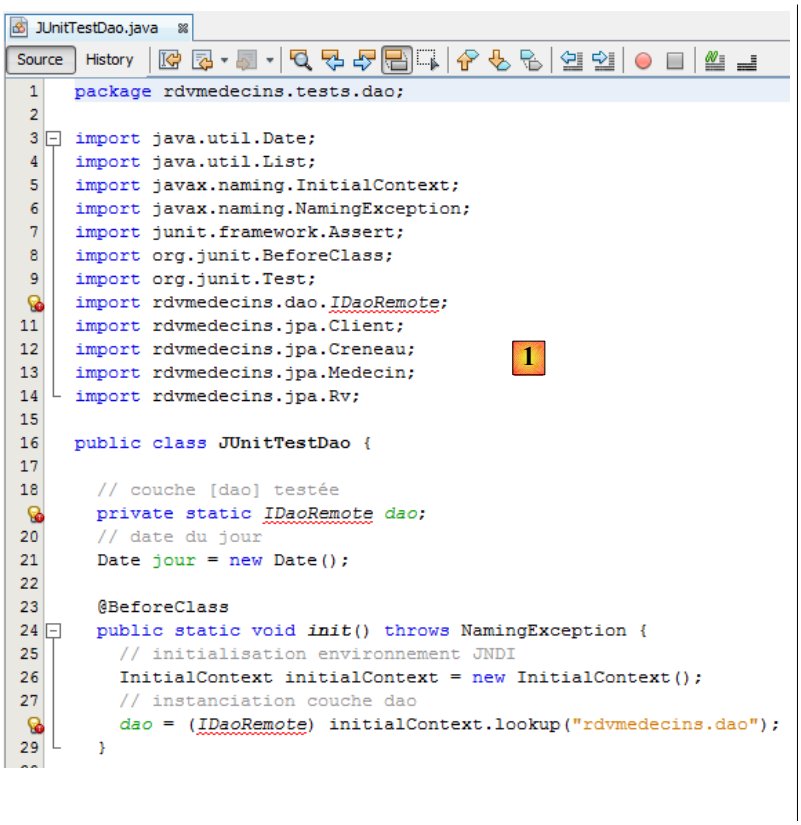
A l'exécution du code, tous les beans du fichier de configuration seront instanciés. Nous verrons comment.

4.1.7 La classe de test JUnit

Nous avions testé la couche [DAO] du projet EJB avec un test JUnit. Nous faisons de même pour la couche [DAO] du projet Spring :



- en [1] et [2], le copier / coller du test JUnit entre les deux projets,
- en [3], le test importé présente des erreurs dans son nouvel environnement.



L'erreur signalée [1] est celle de l'interface distante de l'EJB qui n'existe plus. Par ailleurs, le code d'initialisation du champ [dao] de la ligne 19 était un appel JNDI propre aux EJB (lignes 25-28). Pour instancier le champ [dao] de la ligne 19, il nous faut exploiter le fichier de configuration de Spring. Cela se fait de la façon suivante :

JUnitTestDao.java

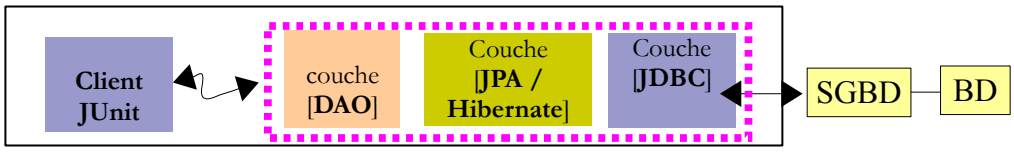
```
4  import java.util.List;
5  import javax.naming.InitialContext;
6  import javax.naming.NamingException;
7  import junit.framework.Assert;
8  import org.junit.BeforeClass;
9  import org.junit.Test;
10 import org.springframework.context.ApplicationContext;
11 import org.springframework.context.support.ClassPathXmlApplicationContext;
12 import rdvmedecins.dao.IDao;
13 import rdvmedecins.jpa.Client;
14 import rdvmedecins.jpa.Creneau;
15 import rdvmedecins.jpa.Medecin;
16 import rdvmedecins.jpa.Rv;
17
18 public class JUnitTestDao {
19
20   // couche [dao] testée
21   private static IDao dao;
22   // date du jour
23   Date jour = new Date();
24
25   @BeforeClass
26   public static void init() throws NamingException {
27     // instanciation couche [dao]
28     ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-dao.xml");
29     dao = (IDao) ctx.getBean("dao");
30   }
```

- ligne 21 : le type de l'interface est devenu [IDao],
- ligne 28 : instancie tous les beans déclarés dans le fichier [spring-config-dao.xml], notamment celui-ci :

```
<bean id="dao" class="rdvmedecins.dao.DaoJpa" />
```

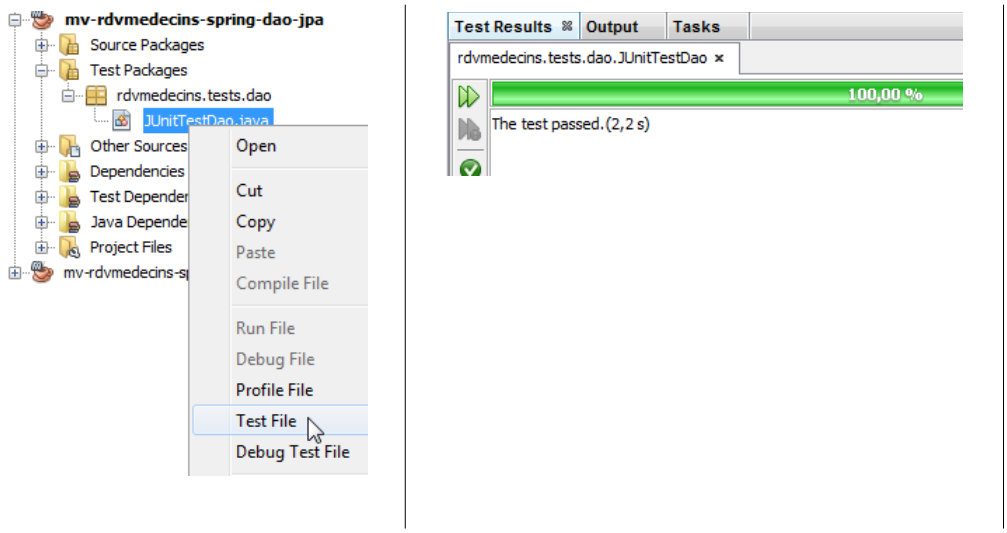
- la ligne 29 demande au contexte Spring de la ligne 28, une référence sur le bean qui a `id="dao"`. On obtient alors une référence sur le singleton [DaoJpa] (*class* ci-dessus) que Spring a instancié.

Les lignes 28-29 construisent les blocs suivants (pointillés roses) :



Lorsque les tests du client JUnit s'exécutent, la couche [DAO] a été instanciée. On peut donc tester ses méthodes. Notons qu'il n'y a pas besoin de serveur pour faire ce test au contraire du test de l'EJB [DAO] qui avait nécessité le serveur Glassfish. Ici, tout s'exécute dans la même JVM.

On peut maintenant exécuter le test JUnit. Il faut que le serveur MySQL soit lancé. Les résultats sont les suivants :



Le test JUnit a réussi. Examinons les logs du test comme il avait été fait lors du test de l'EJB :

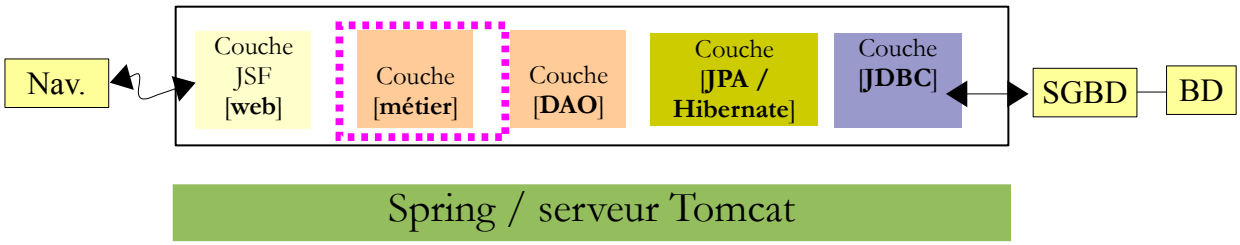
```
1.  mai 24, 2012 5:10:29 PM org.springframework.context.support.AbstractApplicationContext
    prepareRefresh
2.  Infos: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@67291453:
    startup date [Thu May 24 17:10:29 CEST 2012]; root of context hierarchy
3.  mai 24, 2012 5:10:29 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
4.  ...
5.  mai 24, 2012 5:10:30 PM org.hibernate.annotations.common.Version <clinit>
6.  INFO: HCANN000001: Hibernate Commons Annotations {4.0.1.Final}
7.  mai 24, 2012 5:10:30 PM org.hibernate.Version logVersion
8.  INFO: HHH000412: Hibernate Core {4.1.2}
9.  mai 24, 2012 5:10:30 PM org.hibernate.cfg.Environment <clinit>
10. ...
11. Infos: Pre-instantiating singletons in
    org.springframework.beans.factory.support.DefaultListableBeanFactory@6affe94b: defining beans
    [dao,entityManagerFactory,dataSource,org.springframework.aop.config.internalAutoProxyCreator,org.s
    pringframework.transaction.annotation.AnnotationTransactionAttributeSource#0,org.springframework.t
    ransaction.interceptor.TransactionInterceptor#0,org.springframework.transaction.config.internalTra
    nsactionAdvisor,txManager,org.springframework.dao.annotation.PersistenceExceptionTranslationPostPr
    ocessor#0,org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor#0]; root of
    factory hierarchy
12. Liste des clients :
13. Client[1,Mr,Jules,MARTIN]
14. Client[2,Mme,Christine,GERMAN]
15. Client[3,Mr,Jules,JACQUARD]
16. Client[4,Melle,Brigitte,BISTROU]
17. Liste des médecins :
18. Médecin[1,Mme,Marie,PELISSIER]
19. Médecin[2,Mr,Jacques,BROMARD]
20. Médecin[3,Mr,Philippe,JANDOT]
21. Médecin[4,Melle,Justine,JACQUEMOT]
22. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER]
23. Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]]
24. Creneau [2, 1, 8:20, 8:40,Médecin[1,Mme,Marie,PELISSIER]]
25. Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]]
26. Creneau [4, 1, 9:0, 9:20,Médecin[1,Mme,Marie,PELISSIER]]
27. Creneau [5, 1, 9:20, 9:40,Médecin[1,Mme,Marie,PELISSIER]]
28. Creneau [6, 1, 9:40, 10:0,Médecin[1,Mme,Marie,PELISSIER]]
29. Creneau [7, 1, 10:0, 10:20,Médecin[1,Mme,Marie,PELISSIER]]
30. Creneau [8, 1, 10:20, 10:40,Médecin[1,Mme,Marie,PELISSIER]]
31. Creneau [9, 1, 10:40, 11:0,Médecin[1,Mme,Marie,PELISSIER]]
32. Creneau [10, 1, 11:0, 11:20,Médecin[1,Mme,Marie,PELISSIER]]
33. Creneau [11, 1, 11:20, 11:40,Médecin[1,Mme,Marie,PELISSIER]]
34. Creneau [12, 1, 11:40, 12:0,Médecin[1,Mme,Marie,PELISSIER]]
35. Creneau [13, 1, 14:0, 14:20,Médecin[1,Mme,Marie,PELISSIER]]
36. Creneau [14, 1, 14:20, 14:40,Médecin[1,Mme,Marie,PELISSIER]]
37. Creneau [15, 1, 14:40, 15:0,Médecin[1,Mme,Marie,PELISSIER]]
38. Creneau [16, 1, 15:0, 15:20,Médecin[1,Mme,Marie,PELISSIER]]
39. Creneau [17, 1, 15:20, 15:40,Médecin[1,Mme,Marie,PELISSIER]]
40. Creneau [18, 1, 15:40, 16:0,Médecin[1,Mme,Marie,PELISSIER]]
41. Creneau [19, 1, 16:0, 16:20,Médecin[1,Mme,Marie,PELISSIER]]
42. Creneau [20, 1, 16:20, 16:40,Médecin[1,Mme,Marie,PELISSIER]]
```

```
43. Creneau [21, 1, 16:40, 17:0,Médecin[1,Mme,Marie,PELISSIER]]
44. Creneau [22, 1, 17:0, 17:20,Médecin[1,Mme,Marie,PELISSIER]]
45. Creneau [23, 1, 17:20, 17:40,Médecin[1,Mme,Marie,PELISSIER]]
46. Creneau [24, 1, 17:40, 18:0,Médecin[1,Mme,Marie,PELISSIER]]
47. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER], le [Thu May 24 17:10:30 CEST 2012]
48. Rv[211, Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]],
    Client[4,Melle,Brigitte,BISTROU]]
49. Ajout d'un Rv le [Thu May 24 17:10:30 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
50. avant persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
51. après persist : Rv[216, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
52. Rv ajouté
53. mai 24, 2012 5:10:31 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
54. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Thu May 24 17:10:30 CEST 2012]
55. WARN: SQL Error: 1062, SQLState: 23000
56. Rv[211, Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]],
    Client[4,Melle,Brigitte,BISTROU]]
57. Rv[216, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
58. mai 24, 2012 5:10:31 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper logExceptions
59. Ajout d'un Rv le [Thu May 24 17:10:30 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
60. ERROR: Duplicate entry '2012-05-24-3' for key 'UNQ1_RV'
61. avant persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
62. javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException:
    Duplicate entry '2012-05-24-3' for key 'UNQ1_RV'
63. javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException:
    Duplicate entry '2012-05-24-3' for key 'UNQ1_RV'
64. javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException:
    Duplicate entry '2012-05-24-3' for key 'UNQ1_RV'
65. javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException:
    Duplicate entry '2012-05-24-3' for key 'UNQ1_RV'
66. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Thu May 24 17:10:30 CEST 2012]
67. Rv[211, Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]],
    Client[4,Melle,Brigitte,BISTROU]]
68. Rv[216, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
69. Suppression du Rv ajouté
70. Rv supprimé
71. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Thu May 24 17:10:30 CEST 2012]
72. Rv[211, Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]],
    Client[4,Melle,Brigitte,BISTROU]]
```

- lignes 1-4 : des logs de Spring,
- lignes 5-10 : des logs d'Hibernate,
- ligne 11 : Spring signale tous les beans qu'il a instanciés. On y retrouve en premier, le bean [dao],
- lignes 12 et suivantes : les logs du test JUnit,
- lignes 60-65 : on voit clairement l'exception provoquée par l'ajout d'un rendez-vous déjà présent dans la base. On rappelle qu'avec l'EJB, on n'avait pas eu cette exception à cause d'un problème de sérialisation.

La couche [dao] est opérationnelle. Nous construisons maintenant la couche [métier].

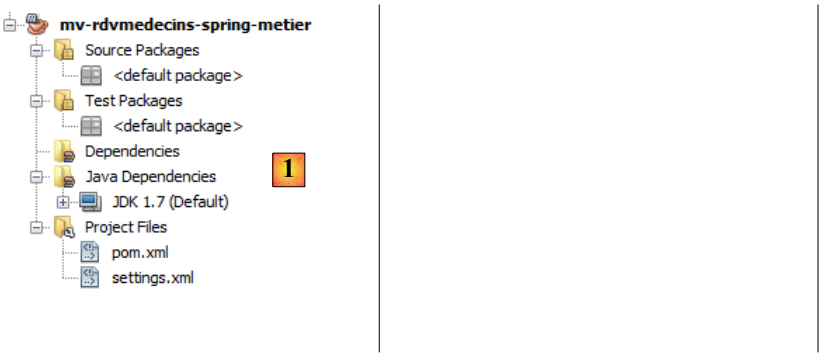
4.2 La couche [métier]



Nous procédons de la même façon que pour la couche [DAO], par copier / coller du projet EJB vers le projet Spring.

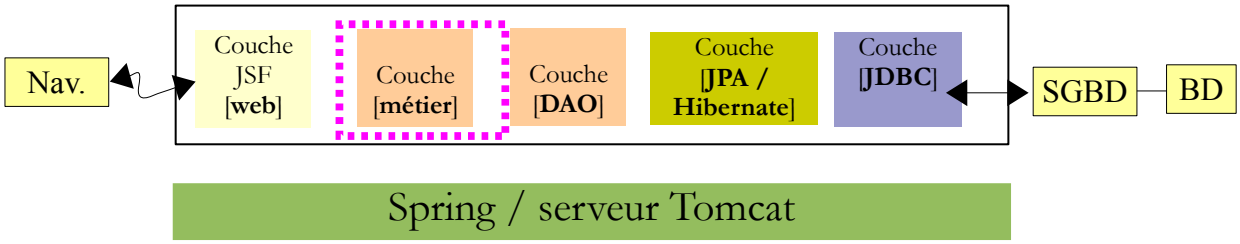
4.2.1 Le projet Netbeans

Nous construisons un nouveau projet Maven de type [Java Application] nettoyé de tout ce qu'on ne veut pas garder [1] :

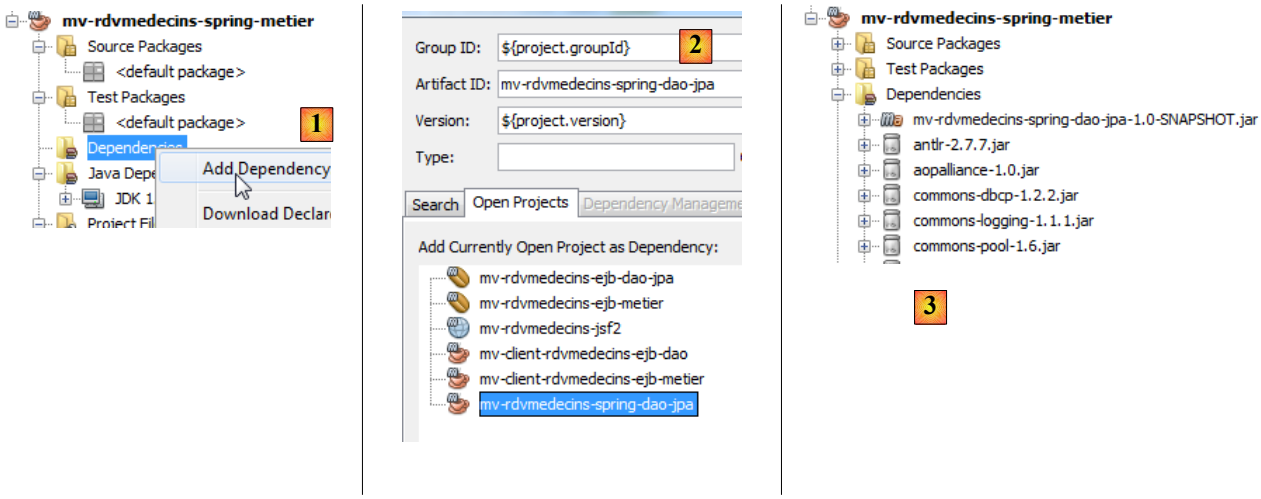


4.2.2 Les dépendances du projet

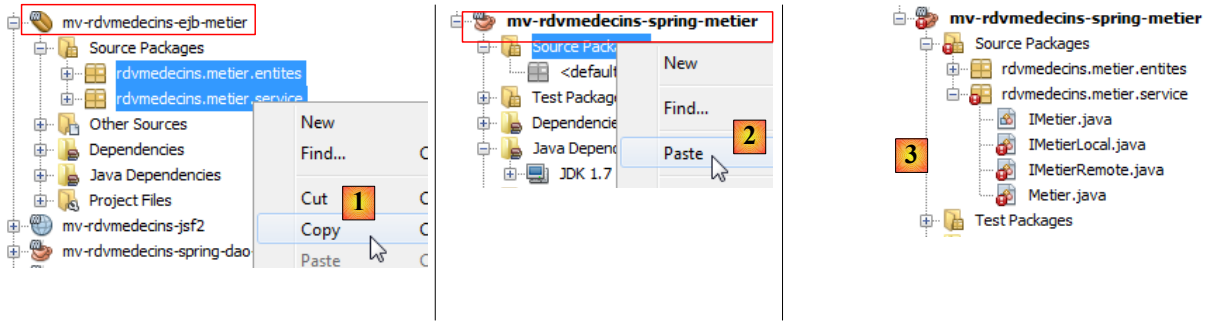
Dans l'architecture :



la couche [métier] s'appuie sur la couche [dao]. Nous ajoutons donc une dépendance sur le projet précédent :

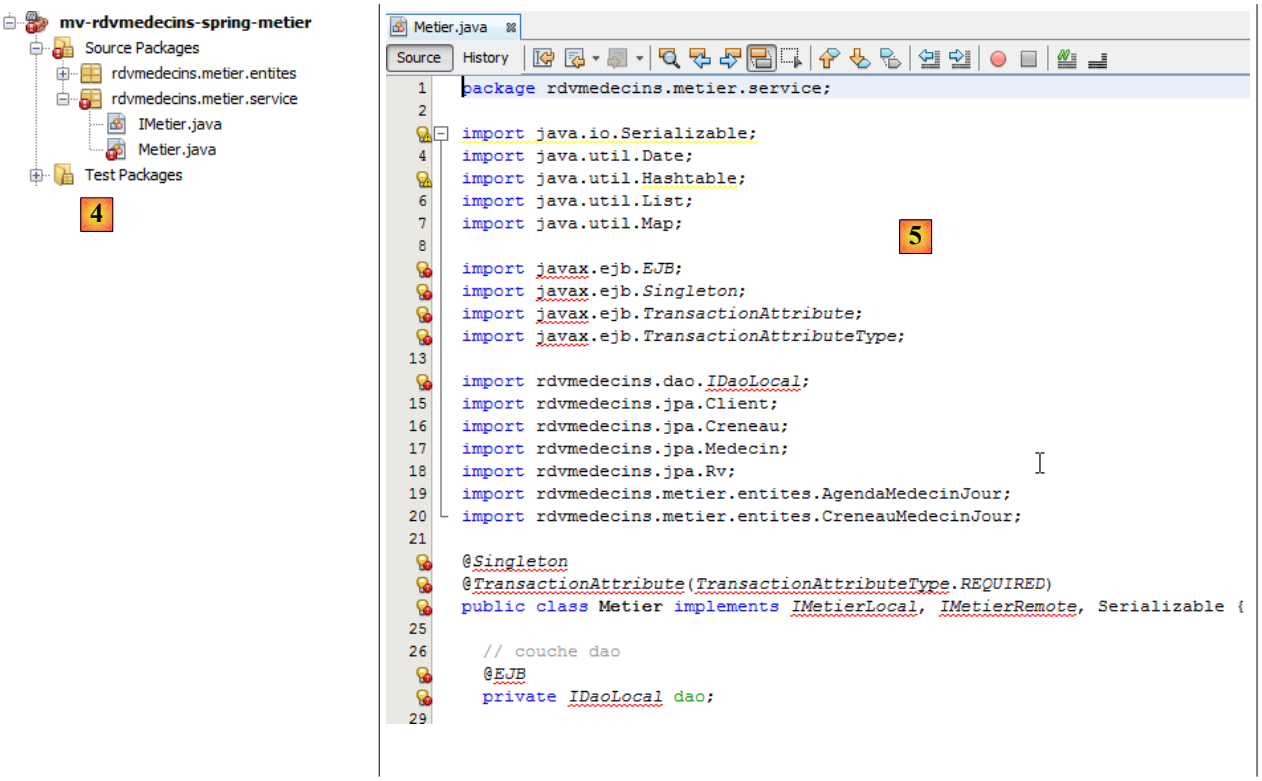


- en [1] et [2], on ajoute une dépendance sur le projet de la couche [dao],
- en [3], cette dépendance a amené d'autres dépendances, celles du projet de la couche [dao].



- en [1] et [2], on copie les sources Java du projet EJB vers le projet Spring,
- en [3], les sources importés présentent des erreurs dans leur nouvel environnement.

On commence par supprimer les interfaces distante et locale de la couche [métier] qui n'existent plus [4] :



- en [5], les erreurs de la classe [Metier] ont plusieurs causes :
  - l'utilisation du paquetage [javax.ejb] qui n'existe plus ;
  - l'utilisation de l'interface [IDaoLocal] qui n'existe plus ;
  - l'utilisation des interfaces [IMetierRemote] et [IMetierLocal] qui n'existent plus.

Nous

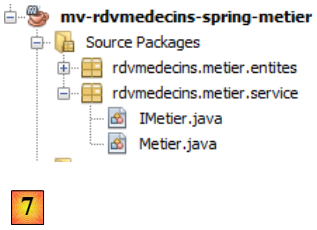
- supprimons toutes les lignes erronées liées au paquetage [javax.ejb],
- remplaçons l'interface [IDaoLocal] par l'interface [IDao],
- remplaçons les interfaces [IMetierRemote] et [IMetierLocal] par l'interface [IMetier].

Metier.java

Source | History

```
1  package rdvmedecins.metier.service;
2
4  import java.io.Serializable;
5  import java.util.Date;
6  import java.util.Hashtable;
7  import java.util.List;
8  import java.util.Map;
9
10 import rdvmedecins.dao.IDao;
11 import rdvmedecins.jpa.Client;
12 import rdvmedecins.jpa.Creneau;
13 import rdvmedecins.jpa.Medecin;
14 import rdvmedecins.jpa.Rv;
15 import rdvmedecins.metier.entites.AgendaMedecinJour;
16 import rdvmedecins.metier.entites.CreneauMedecinJour;
17
18 public class Metier implements IMetier, Serializable {
19
20   // couche dao
21   private IDao dao;
```

6



7

- en [6], la classe ainsi corrigée,
- en [7], il n'y a plus d'erreurs.

Nous avons supprimé les références aux EJB mais il nous faut maintenant retrouver leurs propriétés :

Metier.java

Source | History

```
20 import rdvmedecins.metier.entites.CreneauMedecinJour;
21
22 @Singleton
23 @TransactionAttribute(TransactionAttributeType.REQUIRED)
24 public class Metier implements IMetierLocal, IMetierRemote, Serializable {
25
26   // couche dao
27   @EJB
28   private IDaoLocal dao;
```

- ligne 22 : on avait un singleton. Cette caractéristique sera obtenue en faisant de la classe un bean géré par Spring,
- ligne 23 : chaque méthode se déroulait dans une transaction. Ce sera obtenu avec l'annotation Spring **@Transactional**,
- lignes 27-28 : la référence sur la couche [DAO] était obtenue par injection du conteneur EJB. Nous utiliserons une injection Spring.

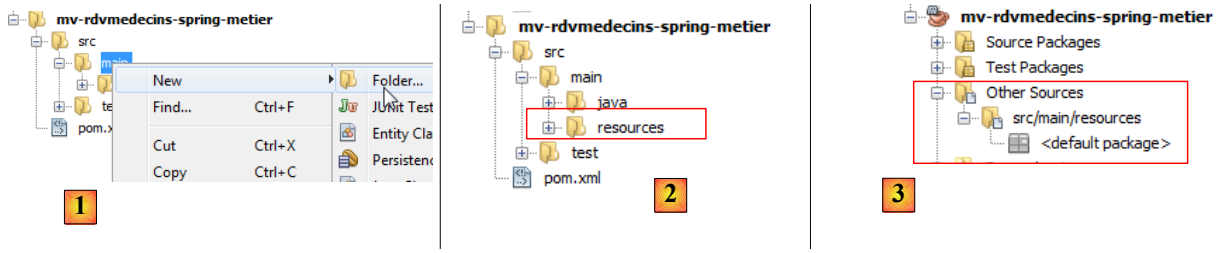
Le code de la classe [Metier] du projet Spring évolue donc comme suit :

```
Metier.java
Source | History
 2
 3  import java.io.Serializable;
 4  import java.util.Date;
 5  import java.util.Hashtable;
 6  import java.util.List;
 7  import java.util.Map;
 8  import org.springframework.transaction.annotation.Transactional;
 9
10  import rdvmedecins.dao.IDao;
11  import rdvmedecins.jpa.Client;
12  import rdvmedecins.jpa.Creneau;
13  import rdvmedecins.jpa.Medecin;
14  import rdvmedecins.jpa.Rv;
15  import rdvmedecins.metier.entites.AgendaMedecinJour;
16  import rdvmedecins.metier.entites.CreneauMedecinJour;
17
18  @Transactional
19  public class Metier implements IMetier, Serializable {
20
21    // couche dao
22    private IDao dao;
```

C'est tout pour le code Java. Le reste se passe dans le fichier de configuration de Spring.

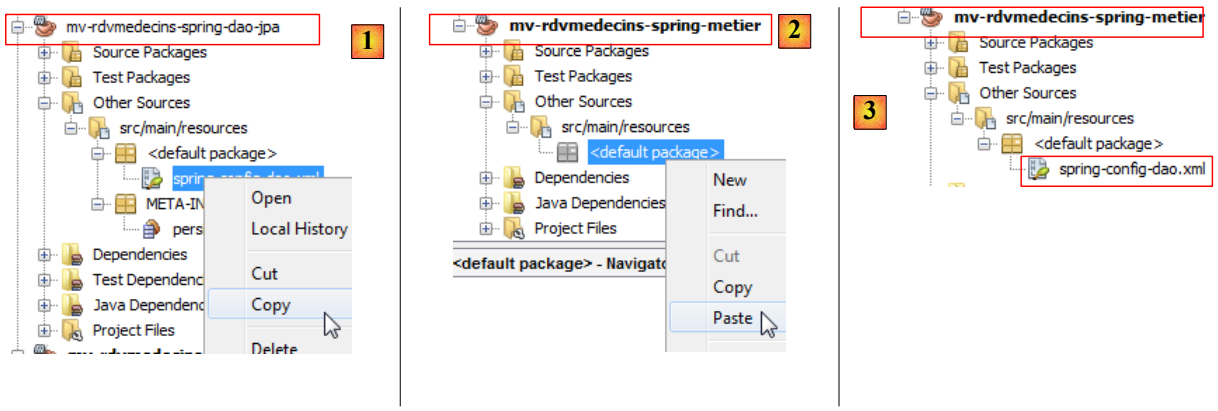
4.2.3 Le fichier de configuration de Spring

Nous copions le fichier de configuration de Spring du projet de la couche [DAO] dans le projet de la couche [métier]. Nous commençons par créer la branche [Other Resources] dans le projet de la couche [métier] si elle n'existe pas :



- en [1], dans l'onglet [Files], on crée un sous-dossier au dossier [main],
- en [2], il doit s'appeler [resources],
- en [3], dans l'onglet [Projects], la branche [Other Sources] a été créée.

Nous pouvons passer au copier / coller du fichier de configuration de Spring :



- en [1] on copie le fichier du projet [DAO] dans le projet [métier] [2],
- en [3], le fichier copié.

Le fichier de configuration qui a été copié configure la couche [DAO]. Nous lui ajoutons un bean pour configurer la couche [métier] :

```
1.    <!-- couches applicatives -->
2.    <bean id="dao" class="rdvmedecins.dao.DaoJpa" />
3.    <bean id="metier" class="rdvmedecins.metier.service.Metier">
4.      <property name="dao" ref="dao"/>
5.  </bean>
```

- ligne 2 : le bean de la couche [DAO],
- lignes 3-5 : le bean de la couche [métier],
- ligne 3 : le bean s'appelle *métier* (attribut *id*) et est une instance de la classe [rdvmedecins.metier.service.Metier] (attribut *class*). Ce bean sera instancié comme les autres au démarrage de l'application.

Rappelons le code du bean [rdvmedecins.metier.service.Metier] :

```
1.  package rdvmedecins.metier.service;
2.
3.  ...
4.
5.  public class Metier implements IMetier, Serializable {
6.
7.    // couche dao
8.    private IDao dao;
9.
10.   public Metier() {
11. }
```

- ligne 8 : le champ [dao] va être instancié par Spring en même temps que le bean *métier*. Revenons à la définition de ce bean dans le fichier de configuration de Spring :

```
1.    <!-- couches applicatives -->
2.    <bean id="dao" class="rdvmedecins.dao.DaoJpa" />
3.    <bean id="metier" class="rdvmedecins.metier.service.Metier">
4.      <property name="dao" ref="dao"/>
5.  </bean>
```

- ligne 4 : la balise <property> sert à initialiser des champs du bean instancié. Le nom du champ est donné par l'attribut *name*. C'est donc le champ **dao** de la classe [rdvmedecins.metier.service.Metier] qui va être instancié. Il le sera via une méthode **setDao** qui doit exister. La valeur qui lui sera affectée est celle de l'attribut **ref**. Cette valeur est ici, la référence du bean **dao** de la ligne 2.

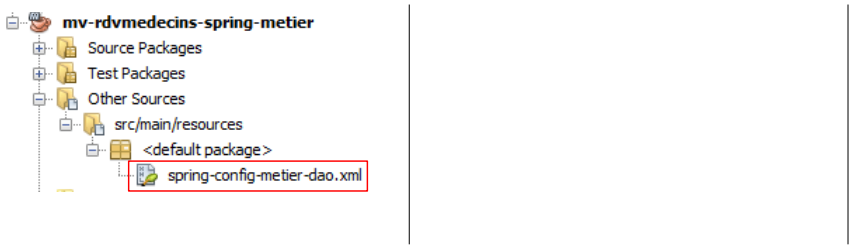
Dit plus simplement, dans le code :

```
12. package rdvmedecins.metier.service;
13.
14. ...
15.
16. public class Metier implements IMetier, Serializable {
17.
18.   // couche dao
19.   private IDao dao;
20.
21.   public Metier() {
22. }
```

Le champ **dao** de la ligne 19 va être initialisé par Spring avec une référence sur la couche [dao]. C'est ce que nous voulions. Le champ **dao** sera initialisé par Spring via un *setter* que nous devons rajouter :

```
1.    // setter
2.
3.    public void setDao(IDao dao) {
4.      this.dao = dao;
5.  }
```

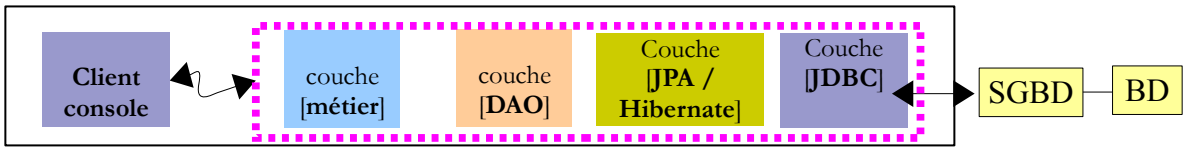
Nous renommons le fichier de configuration de Spring pour tenir compte des changements :



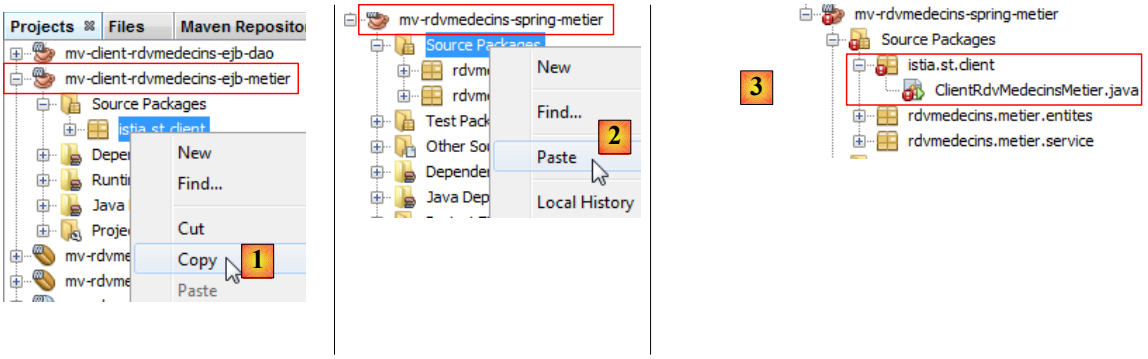
Nous sommes désormais prêts pour un test. Nous reprenons le test console utilisé pour tester l'EJB [Metier].

4.2.4 Test de la couche [métier]

Le test se déroulera avec l'architecture suivante :



Nous copions le test console du projet EJB dans le projet Spring :



- en [1] et [2], le copier / coller entre les deux projets,
- en [3], le code importé présente des erreurs.

ClientRdvMedecinsMetier.java

```
1  package istia.st.client;
2
3  import java.util.Date;
4  import java.util.List;
5
6  import javax.naming.InitialContext;
7  import rdvmedecins.jpa.Client;
8  import rdvmedecins.jpa.Creneau;
9
10 import rdvmedecins.jpa.Medecin;
11 import rdvmedecins.jpa.Rv;
12 import rdvmedecins.metier.entites.AgendaMedecinJour;
13 import rdvmedecins.metier.service.IMetierRemote;
14
15 public class ClientRdvMedecinsMetier {
16
17   // le nom de l'interface distante de l'EJB [Metier]
18   private static String IDaoRemoteName = "java:global/istia.st_mv-rdvmedecins-metier
19   // date du jour
20   private static Date jour = new Date();
21
22   public static void main(String[] args) {
23     try {
24       // contexte JNDI du serveur Glassfish
25       InitialContext initialContext = new InitialContext();
26       // référence sur couche [metier] distante
27       IMetierRemote metier = (IMetierRemote) initialContext.lookup(IDaoRemoteName);
28       // affichage clients
```

Le code importé présente deux types d'erreur :

- ligne 13 : l'interface [IMetierRemote] a été remplacée par l'interface [IMetier],
- lignes 24-27 : l'instanciation de la couche [métier] ne se fait plus avec un appel JNDI mais par instanciation des beans du fichier de configuration de Spring.

Nous corrigeons ces deux points :

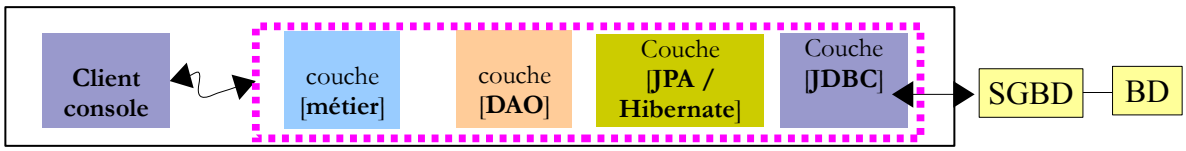
ClientRdvMedecinsMetier.java

```
1  package istia.st.client;
2
3  import java.util.Date;
4  import java.util.List;
5  import org.springframework.context.ApplicationContext;
6  import org.springframework.context.support.ClassPathXmlApplicationContext;
7  import rdvmedecins.jpa.Client;
8  import rdvmedecins.jpa.Creneau;
9  import rdvmedecins.jpa.Medecin;
10 import rdvmedecins.jpa.Rv;
11 import rdvmedecins.metier.entites.AgendaMedecinJour;
12 import rdvmedecins.metier.service.IMetier;
13
14 public class ClientRdvMedecinsMetier {
15
16   // date du jour
17   private static Date jour = new Date();
18
19   public static void main(String[] args) {
20     try {
21       // instanciation couche [métier]
22       ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-metier-dao.xml");
23       IMetier metier = (IMetier) ctx.getBean("metier");
24       // affichage clients
```

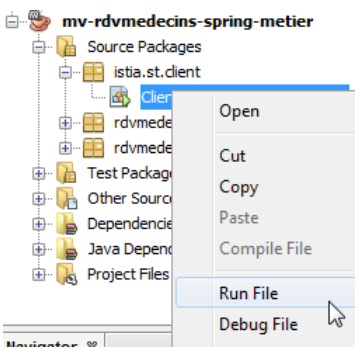
- ligne 22 : le fichier [spring-config-metier-dao.xml] est exploité. Tous les beans de ce fichier sont alors instanciés. Parmi eux, il y a ceux-ci :

```
1.    <!-- couches applicatives -->
2.    <bean id="dao" class="rdvmedecins.dao.DaoJpa" />
3.    <bean id="metier" class="rdvmedecins.metier.service.Metier">
4.      <property name="dao" ref="dao"/>
5.  </bean>
```

Ces deux beans représentent les couches [DAO] et [métier] de l'architecture du test :



Ceci fait, le test peut se dérouler :



Les logs du test sont alors les suivants :

```
1.  mai 25, 2012 9:45:07 AM org.springframework.context.support.AbstractApplicationContext
    prepareRefresh
2.  Infos: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@22a92801:
    startup date [Fri May 25 09:45:07 CEST 2012]; root of context hierarchy
3.  mai 25, 2012 9:45:07 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader
    loadBeanDefinitions
4.  ....
5.  Infos: Pre-instantiating singletons in
    org.springframework.beans.factory.support.DefaultListableBeanFactory@38a0a058: defining beans
    [dao,metier,entityManagerFactory,dataSource,org.springframework.aop.config.internalAutoProxyCreato
    r,org.springframework.transaction.annotation.AnnotationTransactionAttributeSource#0,org.springfram
    ework.transaction.interceptor.TransactionInterceptor#0,org.springframework.transaction.config.inte
    rnalTransactionAdvisor,txManager,org.springframework.dao.annotation.PersistenceExceptionTranslatio
    nPostProcessor#0,org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor#0];
    root of factory hierarchy
6.  Liste des clients :
7.  Client[1,Mr,Jules,MARTIN]
8.  Client[2,Mme,Christine,GERMAN]
9.  Client[3,Mr,Jules,JACQUARD]
10. Client[4,Melle,Brigitte,BISTROU]
11. Liste des médecins :
12. Médecin[1,Mme,Marie,PELISSIER]
13. Médecin[2,Mr,Jacques,BROMARD]
14. Médecin[3,Mr,Philippe,JANDOT]
15. Médecin[4,Melle,Justine,JACQUEMOT]
16. Liste des créneaux du médecin Médecin[1,Mme,Marie,PELISSIER]
17. Creneau [1, 1, 8:0, 8:20,Médecin[1,Mme,Marie,PELISSIER]]
18. Creneau [2, 1, 8:20, 8:40,Médecin[1,Mme,Marie,PELISSIER]]
19. Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]]
20. Creneau [4, 1, 9:0, 9:20,Médecin[1,Mme,Marie,PELISSIER]]
21. Creneau [5, 1, 9:20, 9:40,Médecin[1,Mme,Marie,PELISSIER]]
```

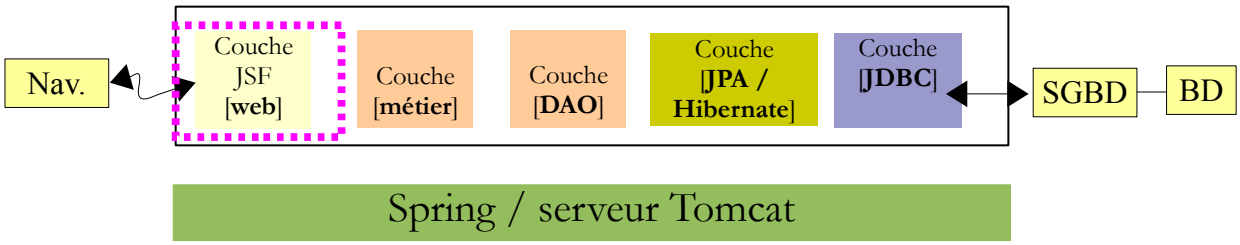
```
22. Creneau [6, 1, 9:40, 10:0,Médecin[1,Mme,Marie,PELISSIER]]
23. Creneau [7, 1, 10:0, 10:20,Médecin[1,Mme,Marie,PELISSIER]]
24. Creneau [8, 1, 10:20, 10:40,Médecin[1,Mme,Marie,PELISSIER]]
25. Creneau [9, 1, 10:40, 11:0,Médecin[1,Mme,Marie,PELISSIER]]
26. Creneau [10, 1, 11:0, 11:20,Médecin[1,Mme,Marie,PELISSIER]]
27. Creneau [11, 1, 11:20, 11:40,Médecin[1,Mme,Marie,PELISSIER]]
28. Creneau [12, 1, 11:40, 12:0,Médecin[1,Mme,Marie,PELISSIER]]
29. Creneau [13, 1, 14:0, 14:20,Médecin[1,Mme,Marie,PELISSIER]]
30. Creneau [14, 1, 14:20, 14:40,Médecin[1,Mme,Marie,PELISSIER]]
31. Creneau [15, 1, 14:40, 15:0,Médecin[1,Mme,Marie,PELISSIER]]
32. Creneau [16, 1, 15:0, 15:20,Médecin[1,Mme,Marie,PELISSIER]]
33. Creneau [17, 1, 15:20, 15:40,Médecin[1,Mme,Marie,PELISSIER]]
34. Creneau [18, 1, 15:40, 16:0,Médecin[1,Mme,Marie,PELISSIER]]
35. Creneau [19, 1, 16:0, 16:20,Médecin[1,Mme,Marie,PELISSIER]]
36. Creneau [20, 1, 16:20, 16:40,Médecin[1,Mme,Marie,PELISSIER]]
37. Creneau [21, 1, 16:40, 17:0,Médecin[1,Mme,Marie,PELISSIER]]
38. Creneau [22, 1, 17:0, 17:20,Médecin[1,Mme,Marie,PELISSIER]]
39. Creneau [23, 1, 17:20, 17:40,Médecin[1,Mme,Marie,PELISSIER]]
40. Creneau [24, 1, 17:40, 18:0,Médecin[1,Mme,Marie,PELISSIER]]
41. Liste des rendez-vous du médecin Médecin[1,Mme,Marie,PELISSIER], le [Fri May 25 09:45:07 CEST
    2012]
42. Agenda[Médecin[1,Mme,Marie,PELISSIER],25/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
43. Ajout d'un Rv le [Fri May 25 09:45:07 CEST 2012] dans le créneau Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] pour le client Client[1,Mr,Jules,MARTIN]
44. avant persist : Rv[null, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
45. après persist : Rv[220, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]],
    Client[1,Mr,Jules,MARTIN]]
46. Rv ajouté
47. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Fri May 25 09:45:07 CEST 2012]
48. Rv[220, Creneau [3, 1, 8:40, 9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]
49. Agenda[Médecin[1,Mme,Marie,PELISSIER],25/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] Rv[220, Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]], Client[1,Mr,Jules,MARTIN]]] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
```

```
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
50. Suppression du Rv ajouté
51. Rv supprimé
52. Liste des Rv du médecin Médecin[1,Mme,Marie,PELISSIER], le [Fri May 25 09:45:07 CEST 2012]
53. Agenda[Médecin[1,Mme,Marie,PELISSIER],25/05/2012, [Creneau [1, 1, 8:0,
    8:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [2, 1, 8:20,
    8:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [3, 1, 8:40,
    9:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [4, 1, 9:0,
    9:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [5, 1, 9:20,
    9:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [6, 1, 9:40,
    10:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [7, 1, 10:0,
    10:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [8, 1, 10:20,
    10:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [9, 1, 10:40,
    11:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [10, 1, 11:0,
    11:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [11, 1, 11:20,
    11:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [12, 1, 11:40,
    12:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [13, 1, 14:0,
    14:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [14, 1, 14:20,
    14:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [15, 1, 14:40,
    15:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [16, 1, 15:0,
    15:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [17, 1, 15:20,
    15:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [18, 1, 15:40,
    16:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [19, 1, 16:0,
    16:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [20, 1, 16:20,
    16:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [21, 1, 16:40,
    17:0,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [22, 1, 17:0,
    17:20,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [23, 1, 17:20,
    17:40,Médecin[1,Mme,Marie,PELISSIER]] null] [Creneau [24, 1, 17:40,
    18:0,Médecin[1,Mme,Marie,PELISSIER]] null]]
```

- lignes 1-4 : les logs de Spring et Hibernate,
- ligne 5 : les beans instanciés par Spring. On notera les beans **dao** et **metier**,
- lignes 6-53 : les logs du test. Ils sont conformes à ce qui avait été obtenu avec le test du projet EJB. Nous renvoyons le lecteur aux commentaires de ce test (paragraphe 3.5.3, page 206).

Nous avons construit la couche [métier]. Nous passons à la dernière couche, la couche [web].

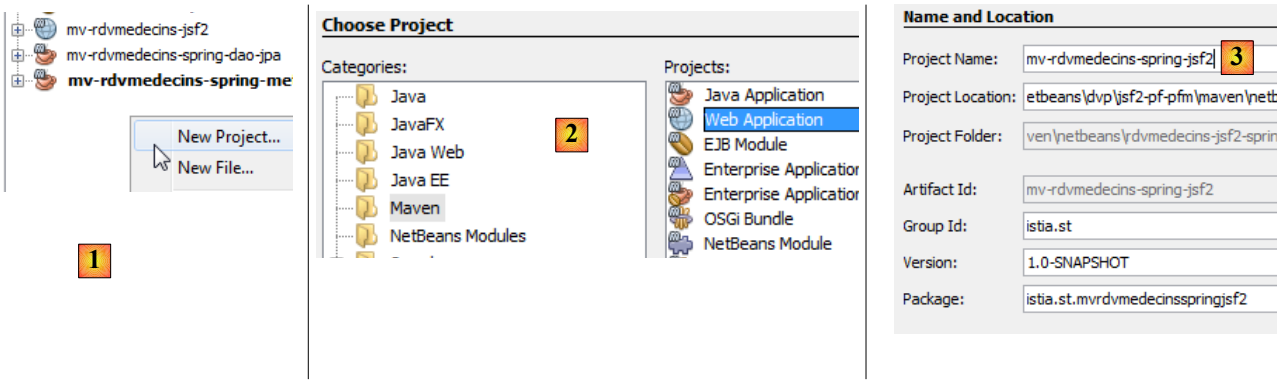
4.3 La couche [web]



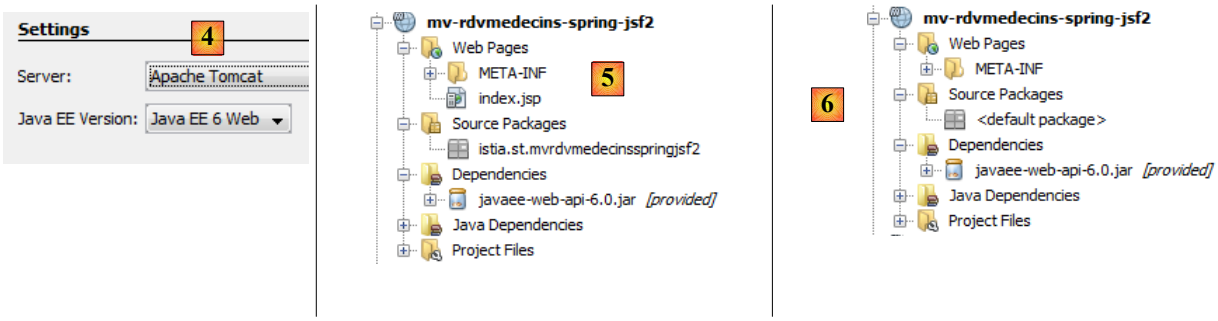
Pour construire la couche [web], nous allons procéder de la même façon que pour les deux autres couches, par copier / coller à partir de la couche [web] du projet EJB.

4.3.1 Le projet Netbeans

Nous construisons d'abord un projet web :



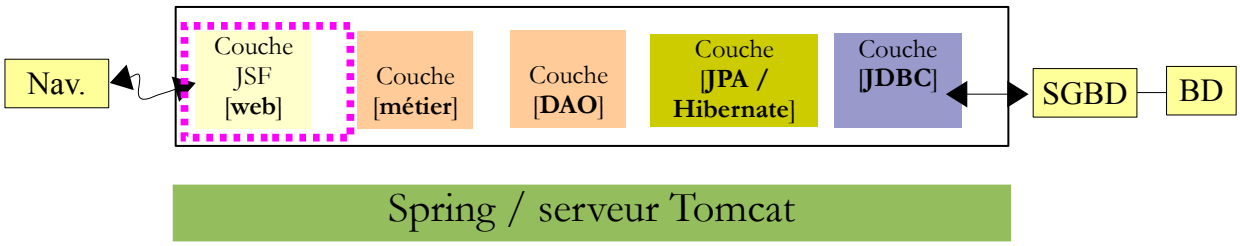
- en [1], on crée un nouveau projet,
- en [2], un projet Maven de type [Web Application],
- en [3], on lui donne un nom,



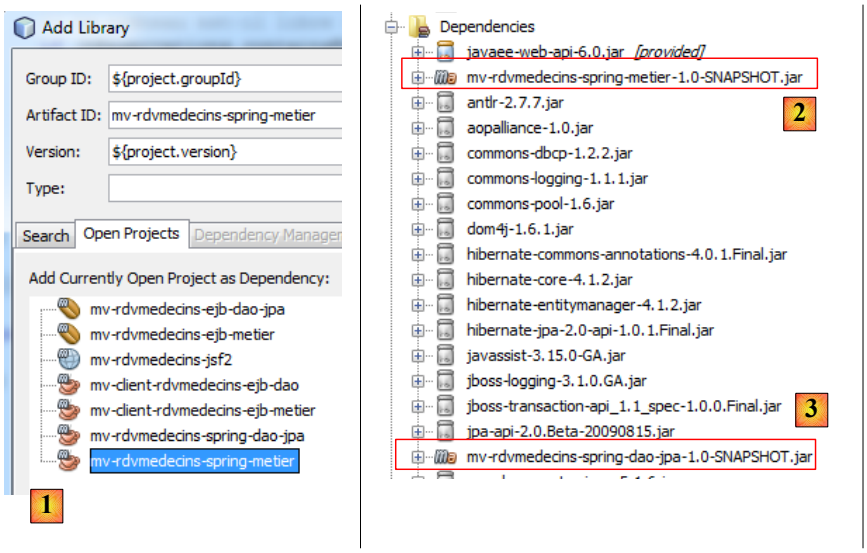
- en [4], on choisit cette fois, le serveur Tomcat et non pas Glassfish qui a servi au projet EJB,
- en [5], le projet obtenu,
- en [6], le projet après la suppression de [index.jsp] et du paquetage de [Source Packages].

4.3.2 Les dépendances du projet

Regardons l'architecture du projet :

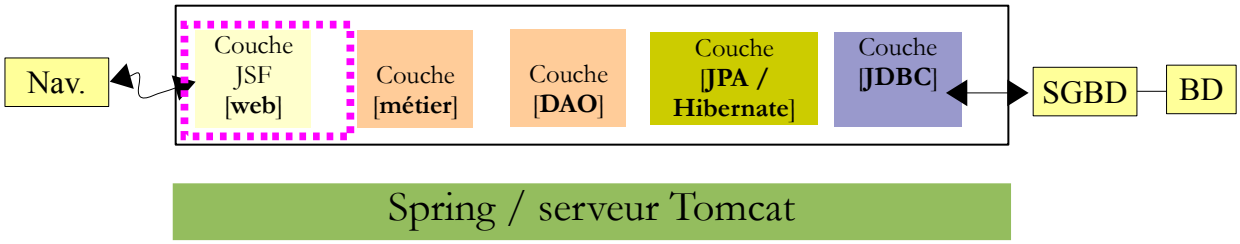


La couche [web] a besoin des couches [métier] [DAO] [JPA]. Celles-ci font partie des deux projets que nous venons de construire. D'où une dépendance vers chacun de ces projets :



- en [1], nous ajoutons la dépendance sur le projet Spring / métier,
- en [2], le projet Spring / métier a été ajouté. Comme il avait lui-même une dépendance sur le projet Spring / DAO / JPA, celui a été automatiquement ajouté dans les dépendances [3].

Revenons à la structure de notre application :



La couche web est une couche JSF. Il nous faut donc les bibliothèques de Java Server Faces. Le serveur Tomcat ne les a pas. La dépendance ne sera donc pas de portée (scope) [provided] comme elle l'avait été avec le serveur Glassfish mais de portée [compile] qui est la portée par défaut lorsqu'on ne précise pas de portée.

Nous ajoutons ces dépendances directement dans le code de [pom.xml] :

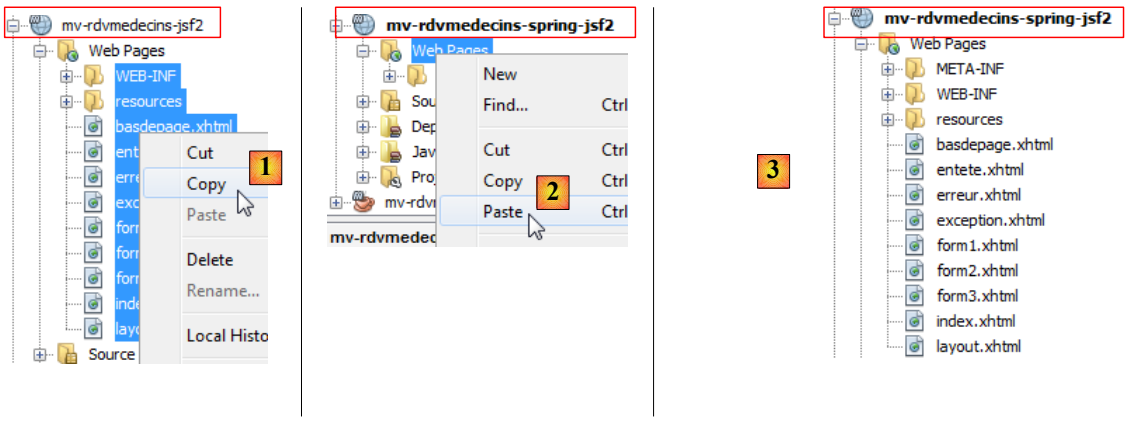
```
1.  <dependencies>
2.      <dependency>
3.        <groupId>${project.groupId}</groupId>
4.        <artifactId>mv-rdvmedecins-spring-metier</artifactId>
5.        <version>${project.version}</version>
6.      </dependency>
7.      <dependency>
8.        <groupId>com.sun.faces</groupId>
9.        <artifactId>jsf-api</artifactId>
10.       <version>2.1.7</version>
11.     </dependency>
12.     <dependency>
13.       <groupId>com.sun.faces</groupId>
14.       <artifactId>jsf-impl</artifactId>
15.       <version>2.1.7</version>
16.     </dependency>
17.     <dependency>
18.       <groupId>javax</groupId>
19.       <artifactId>javaee-web-api</artifactId>
20.       <version>6.0</version>
21.       <scope>provided</scope>
22.     </dependency>
23.   </dependencies>
```

- les lignes 7-16 ont été ajoutées au fichier [pom.xml]. Ce sont les dépendances vis à vis de JSF. Ce sont celles utilisées dans le projet EJB / Glassfish. On notera qu'elles n'ont pas la balise <scope>. Elles ont donc par défaut la portée [compile]. La bibliothèque JSF sera donc embarquée dans l'archive [war] du projet web.

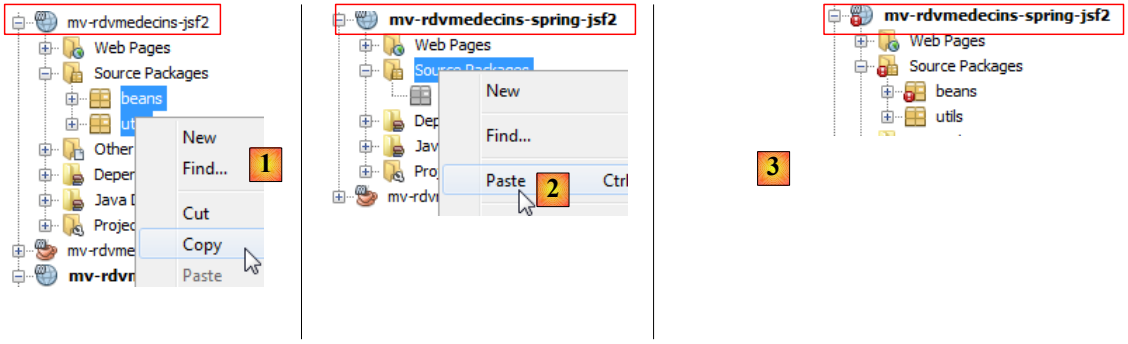
Après avoir ajouté ces dépendances au fichier [pom.xml], nous compilons le projet pour qu'elles soient téléchargées.

4.3.3 Portage du projet JSF / Glassfish vers le projet JSF / Tomcat

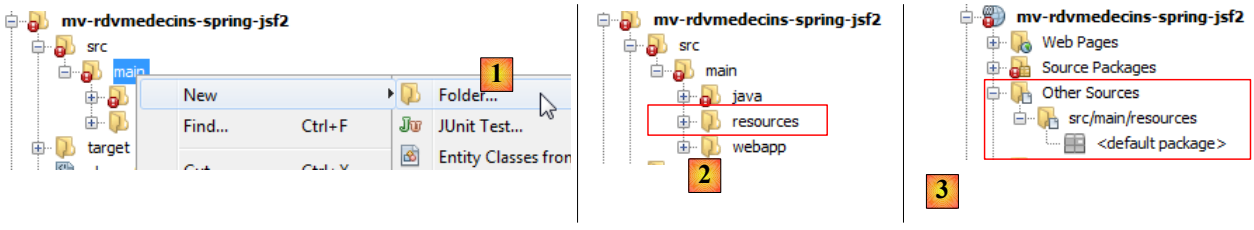
Nous copions l'intégralité des codes du projet JSF / Glassfish vers le projet JSF / Tomcat :



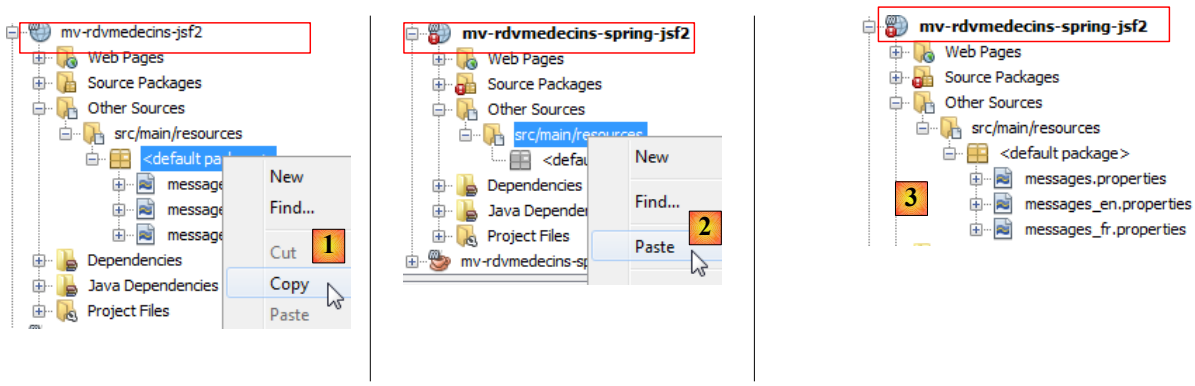
- [1, 2, 3] : copie des pages web de l'ancien projet vers le nouveau,



- [1, 2, 3] : copie des codes Java de l'ancien projet vers le nouveau. Il y a des erreurs. C'est normal. Nous les corrigerons,



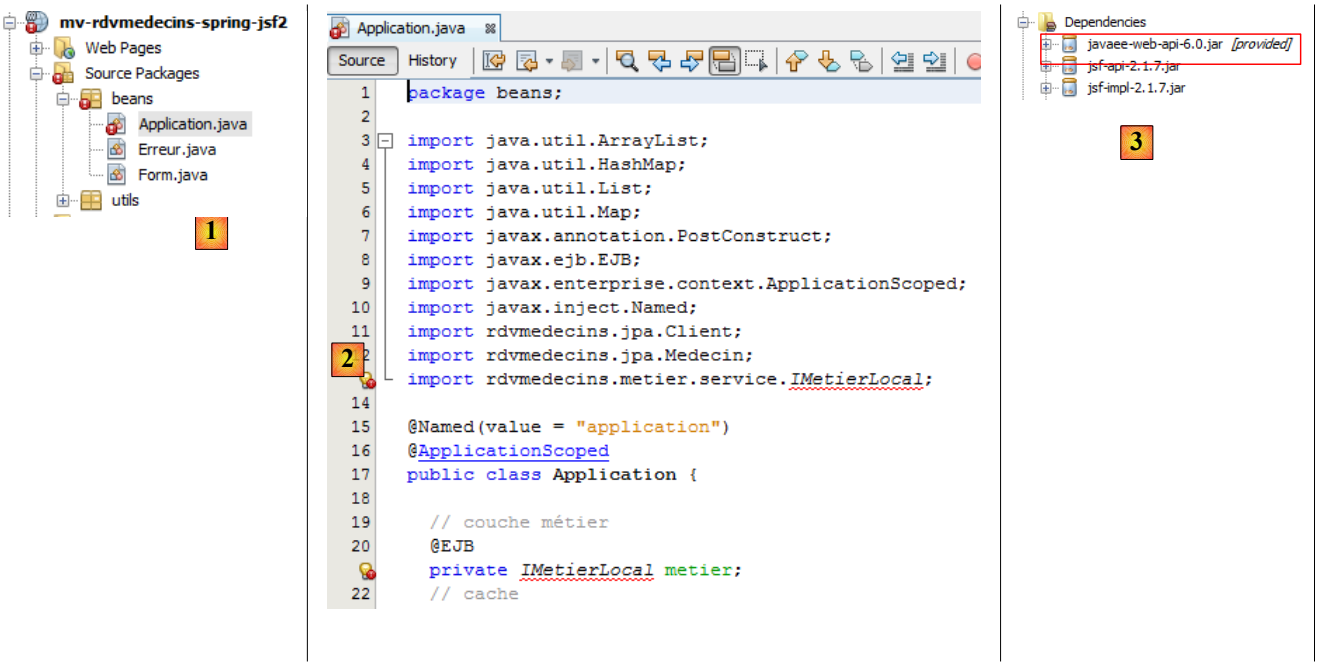
- en [1], dans l'onglet [Files] de Netbeans, on crée un sous-dossier [resources] au dossier [main],
- cela crée dans l'onglet [Projects], la branche [Other Sources] [3],



- [1, 2, 3] : on copie les fichiers de messages de l'ancien projet vers le nouveau projet.

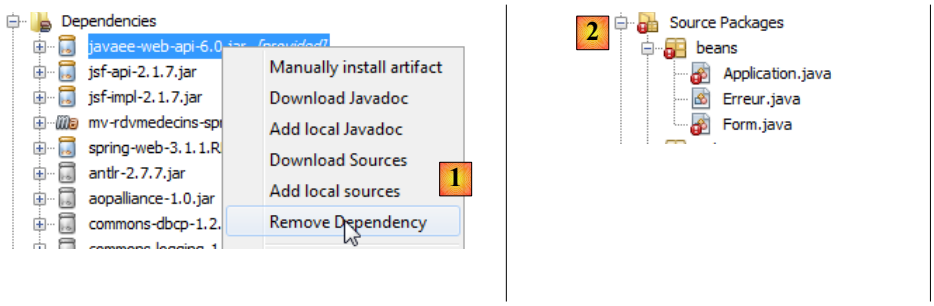
4.3.4 Modifications du projet importé

Nous avons signalé que le code Java importé comportait des erreurs. Examinons les :

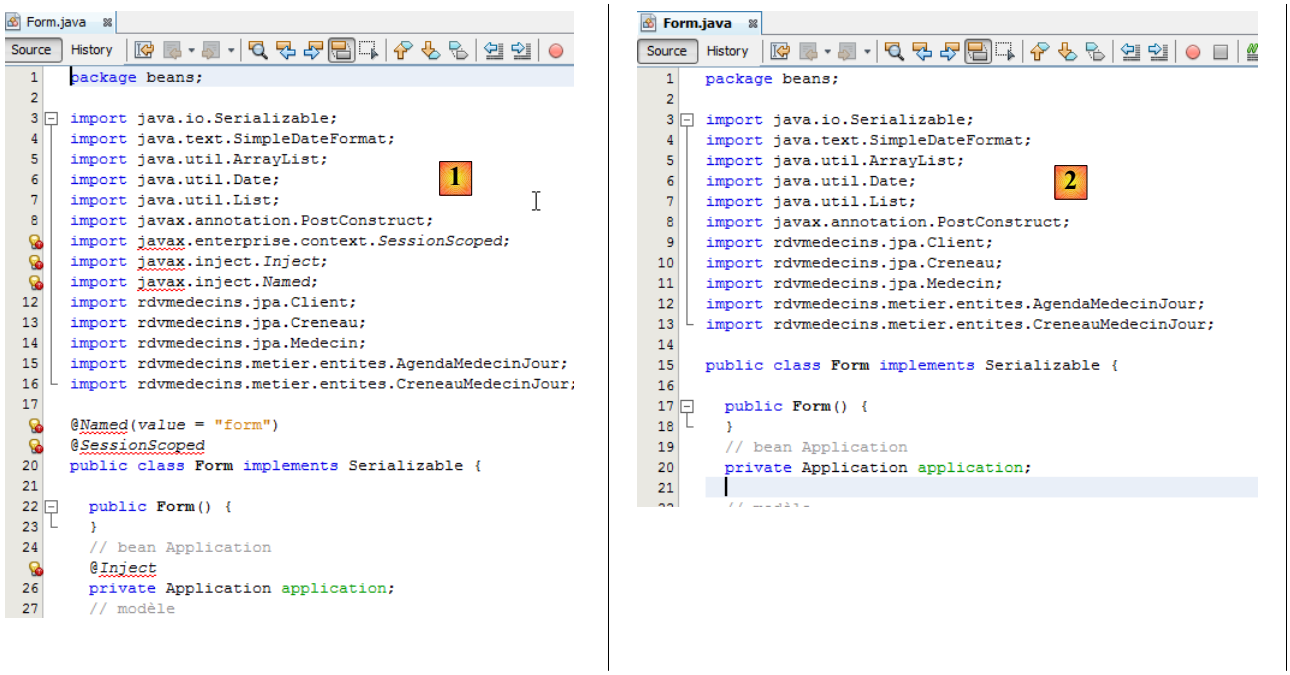


- en [1], seul le bean [Application] est erroné,
- en [2], l'erreur est due uniquement à l'interface [IMetierLocal] qui n'existe plus. Ici, on peut être étonnés que la ligne 20 ne soit pas mise en erreur. L'annotation @EJB fait explicitement référence aux EJB et est ici reconnue. Cela est du à la présence de la dépendance [javaee-web-api-6.0] [3]. Java EE 6 a amené avec lui une architecture qui permet de déployer une application web s'appuyant sur des EJB sans interface distante, sur des serveurs n'ayant pas de conteneur EJB. Il suffit que le serveur fournisse la dépendance [javaee-web-api-6.0]. On voit en effet que celle-ci a la portée [provided] [3].

Ici nous n'allons pas utiliser la dépendance [javaee-web-api-6.0]. Nous la supprimons [1] :

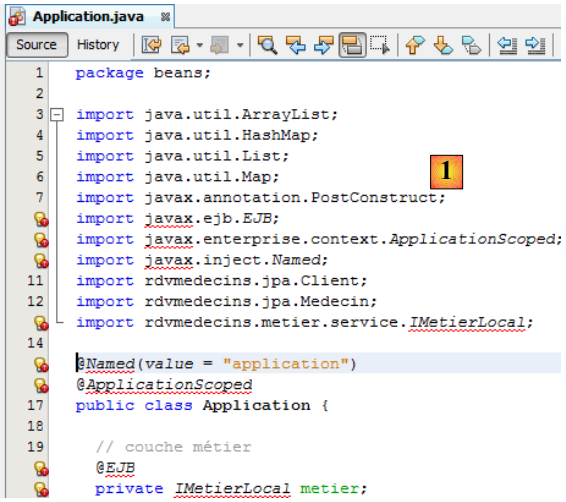


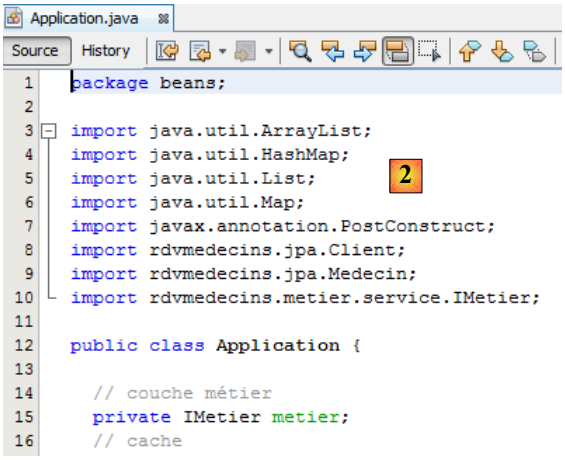
Cela amène de nouvelles erreurs [2]. Nous allons commencer par celles du bean [Form] :



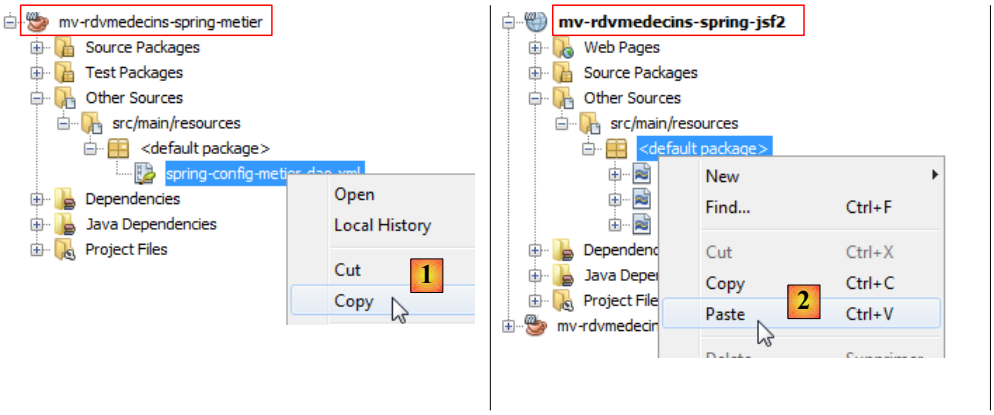
- en [1], les lignes erronées sont liée à la perte du paquetage [javax]. Nous les supprimons toutes [2]. Les lignes erronées faisaient de la classe [Form] un bean de portée session (lignes 18-20 de [1]). Par ailleurs, le bean [Application] était injecté ligne 25. Ces informations vont migrer vers le fichier de configuration de JSF [faces-config.xml].

Passons au bean [Application] :

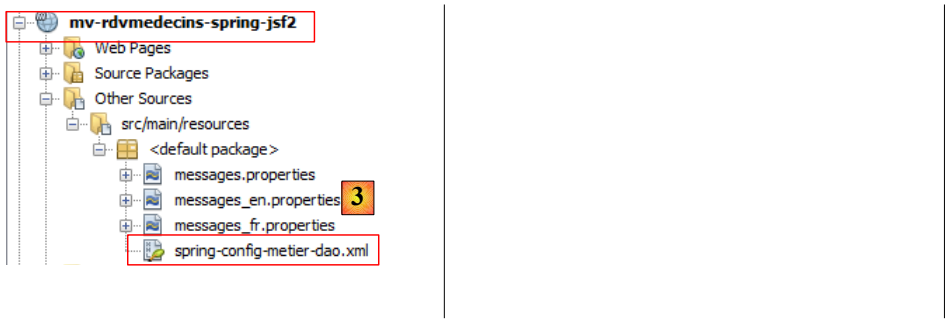




On supprime toutes les lignes erronées de [1] et on change l'interface [IMetierLocal] des lignes 13 et 21 en [IMetier]. En [2], il n'y a plus d'erreurs. En [1], nous avons supprimé les lignes 15-16 qui faisaient de la classe [Application] un bean de portée *application*. Cette information va migrer vers le fichier de configuration de JSF [faces-config.xml]. Nous avons également supprimé la ligne 20 qui injectait une référence de la couche [métier] dans le bean. Maintenant, celle-ci va être initialisée par Spring. Nous avons déjà le fichier de configuration nécessaire, c'est celui du projet Spring / Métier. Nous le copions :



- en [1, 2], nous copions le fichier de configuration de Spring du projet Spring / Métier vers le projet Spring / JSF,



En [3], le résultat.

Dans le bean [Application], il faut exploiter ce fichier de configuration pour avoir une référence sur la couche [métier]. Cela se fait dans sa méthode [init] :

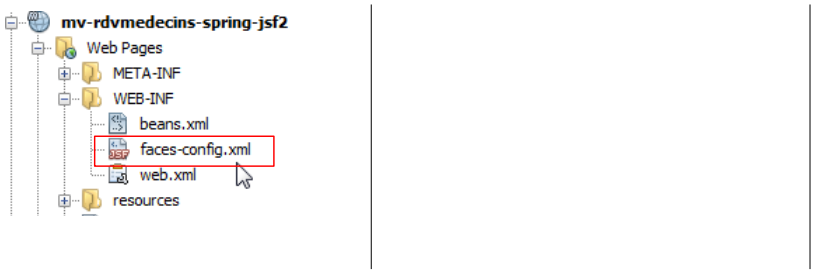
```
1. package beans;
```

```
2.
3.  ...
4.  import org.springframework.context.ApplicationContext;
5.  import org.springframework.context.support.ClassPathXmlApplicationContext;
6.
7.  public class Application {
8.
9.    // couche métier
10.   private IMetier metier;
11. ...
12.
13.   public Application() {
14.   }
15.
16.   @PostConstruct
17.   public void init() {
18.     try {
19.       // instanciation couche [métier]
20.       ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-metier-dao.xml");
21.       metier = (IMetier) ctx.getBean("metier");
22.       // on met les médecins et les clients en cache
23. ...
24.     } catch (Throwable th) {
25. ...
26.     }
27. ...
28.   }
```

- ligne 20 : les beans du fichier de configuration de Spring sont instanciés,
- ligne 21 : on demande une référence sur le bean **metier** donc sur la couche [métier].

De façon générale, l'instanciation des beans de Spring est à faire dans la méthode **init** du bean de portée **application**. Il existe une autre méthode où l'instanciation des beans est faite par une servlet de Spring. Cela implique de changer le fichier [web.xml] et d'ajouter une dépendance sur l'artifact [spring-web]. Nous ne l'avons pas fait ici pour rester en ligne avec ce qui avait été utilisé dans les codes précédents.

Nous avons supprimé les annotations dans les classes [Application] et [Form] qui en faisaient des beans JSF. Ces classes doivent rester des beans JSF. Au lieu des annotations, on utilise alors le fichier de configuration de JSF [WEB-INF / faces.config.xml] pour déclarer les beans.



Ce fichier est désormais le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <!-- le fichier des messages -->
12.     <resource-bundle>
13.       <base-name>
14.         messages
15.       </base-name>
```

```
16.       <var>msg</var>
17.     </resource-bundle>
18.     <message-bundle>messages</message-bundle>
19.   </application>
20.     <!-- le bean applicationBean -->
21.     <managed-bean>
22.       <managed-bean-name>applicationBean</managed-bean-name>
23.       <managed-bean-class>beans.Application</managed-bean-class>
24.       <managed-bean-scope>application</managed-bean-scope>
25.     </managed-bean>
26.     <!-- le bean form -->
27.     <managed-bean>
28.       <managed-bean-name>form</managed-bean-name>
29.       <managed-bean-class>beans.Form</managed-bean-class>
30.       <managed-bean-scope>session</managed-bean-scope>
31.       <managed-property>
32.         <property-name>application</property-name>
33.         <value>#{applicationBean}</value>
34.       </managed-property>
35.     </managed-bean>
36. </faces-config>
```

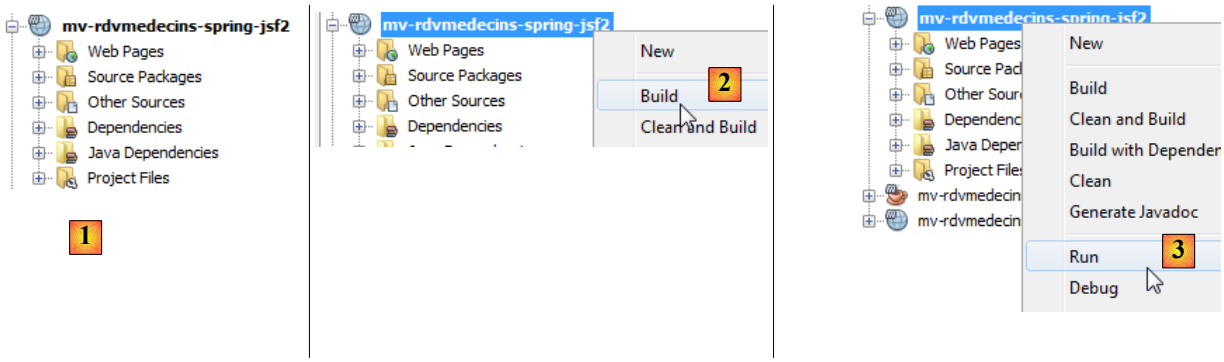
- les lignes 10-19 configurent le fichier des messages. C'était la seule configuration qu'on avait dans le projet JSF / EJB,
- les lignes 21-35 déclarent les beans de l'application JSF. C'était la méthode standard avec JSF 1.x. JSF 2 a introduit les annotations mais la méthode de JSF 1.x est toujours supportée,
- lignes 21-25 : déclarent le bean **applicationBean**,
- ligne 22 : le nom du bean. On pourrait être tenté par le nom **application**. A éviter car c'est le nom d'un bean prédéfini de JSF,
- ligne 23 : le nom complet de la classe du bean,
- ligne 24 : sa portée,
- lignes 27-35 : définissent le bean **form**,
- ligne 28 : le nom du bean,
- ligne 29 : le nom complet de la classe du bean,
- ligne 30 : sa portée,
- lignes 31-34 : définissent une propriété de la classe [beans.Form],
- ligne 32 : nom de la propriété. La classe [beans.Form] doit avoir un champ avec ce nom et le **setter qui va avec**,
- ligne 33 : la valeur du champ. Ici c'est la référence au bean **applicationBean** défini ligne 21. On opère donc là, **l'injection** du bean de portée *application* dans le bean de portée *session* afin que celui-ci ait accès aux données de portée *application*.

Nous avons dit plus haut que le champ [application] du bean [beans.Form] allait être initialisé via un *setter*. Il faut alors le rajouter à la classe [beans.Form] s'il n'existe pas déjà :

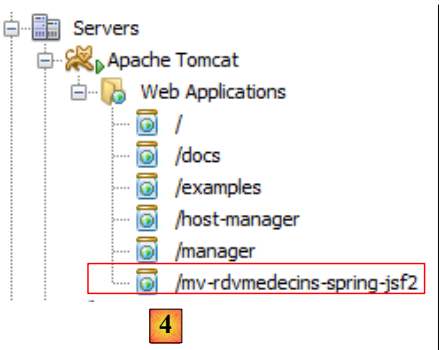
```
1.  public void setApplication(Application application) {
2.      this.application = application;
3.    }
```

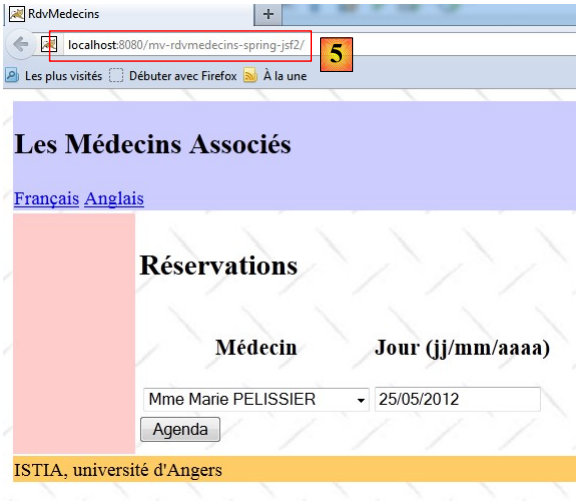
4.3.5 Test de l'application

Notre application est désormais sans erreur et prête pour les tests :

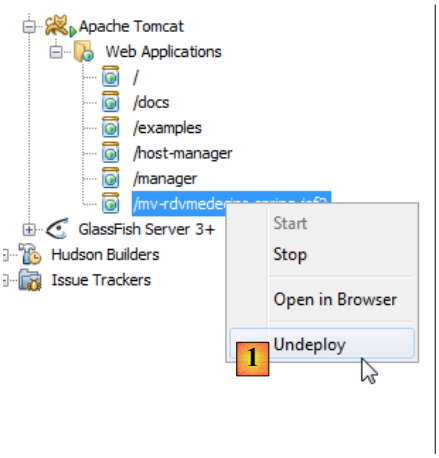


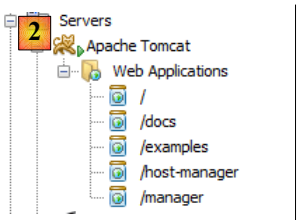
- en [1], le projet corrigé,
- en [2], on le construit,
- en [3], on l'exécute. Il faut que le SGBD MySQL soit lancé. Le serveur Tomcat va alors être lancé [4] s'il ne l'était pas, puis la page d'accueil de l'application va être affichée [5] :





A partir de là, on retrouve l'application étudiée. Nous laissons le lecteur vérifier qu'elle fonctionne. Maintenant arrêtons l'application :





- en [1], on décharge l'application,
- en [2], elle n'est plus là.

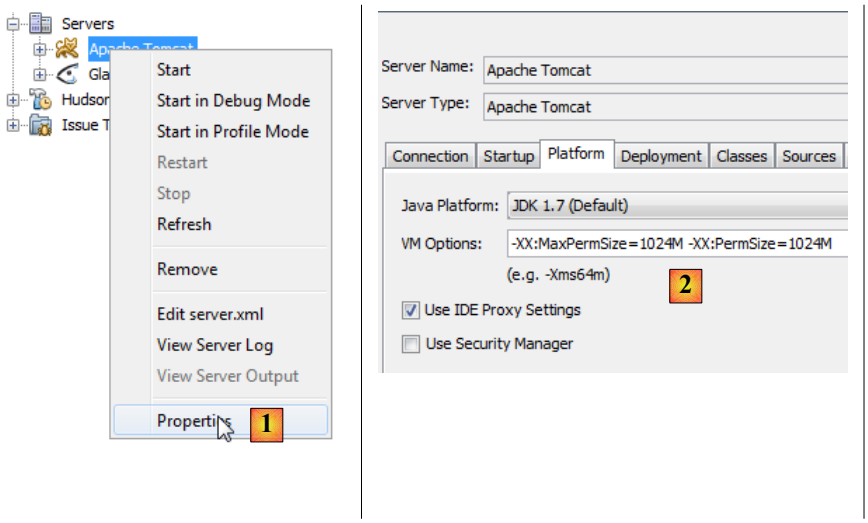
Regardons maintenant les logs de Tomcat :

```
1.  mai 25, 2012 2:15:57 PM org.apache.catalina.loader.WebappClassLoader clearReferencesJdbc
2.  Grave: The web application [/mv-rdvmedecins-spring-jsf2] registered the JDBC driver
    [com.mysql.jdbc.Driver] but failed to unregister it when the web application was stopped. To
    prevent a memory leak, the JDBC Driver has been forcibly unregistered.
3.  mai 25, 2012 2:15:57 PM org.apache.catalina.loader.WebappClassLoader clearReferencesThreads
4.  Grave: The web application [/mv-rdvmedecins-spring-jsf2] appears to have started a thread named
    [MySQL Statement Cancellation Timer] but has failed to stop it. This is very likely to create a
    memory leak.
5.  mai 25, 2012 2:15:59 PM org.apache.catalina.startup.HostConfig checkResources
6.  Infos: Repli (undeploy) de l'application web ayant pour chemin de contexte /mv-rdvmedecins-spring-
    jsf2
```

Les lignes 2 et 4 signalent un dysfonctionnement à l'arrêt de l'application. La ligne 4 indique qu'il y a un risque probable de fuite mémoire. Effectivement, celui-ci a lieu et au bout d'un moment Netbeans n'est plus utilisable. Ce problème est particulièrement irritant car il faut alors relancer Netbeans à chaque nouvelle exécution du projet. Ce problème a été déjà rencontré dans le document " Introduction à Struts 2 par l'exemple " [http://tahe.developpez.com/java/struts2].

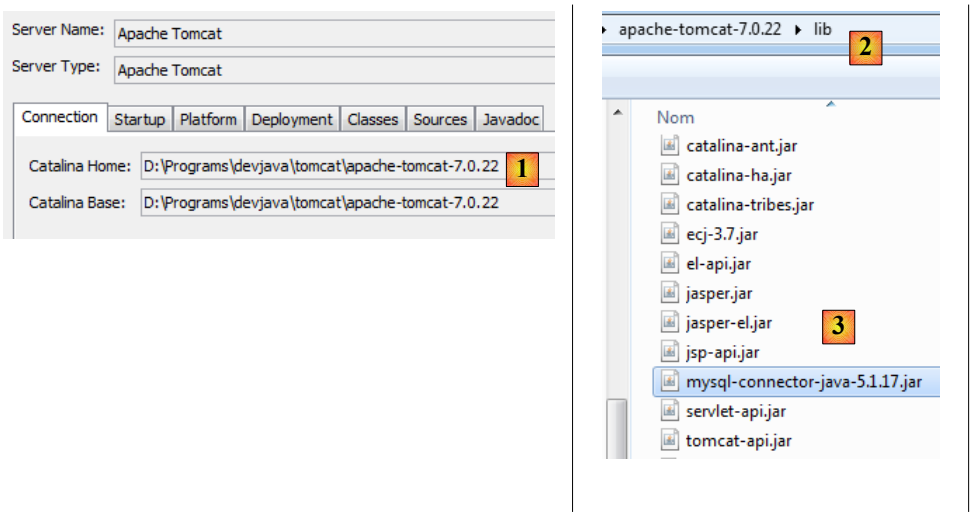
On trouve sur internet beaucoup d'informations sur cette erreur. Elle apparaît lorsqu'on charge / décharge de Tomcat une application de façon répétée. On obtient au bout d'un moment l'erreur *java.lang.OutOfMemoryError: PermGen space*. Il apparaît qu'il n'y a pas de solution pour éviter cette erreur lorsque celle-ci provient d'archives tierces (jar) comme c'est le cas ici. Il faut alors relancer Tomcat pour la faire disparaître.

On peut cependant retarder la survenue de cette erreur. Premièrement, on augmente l'espace de la mémoire qui a débordé.



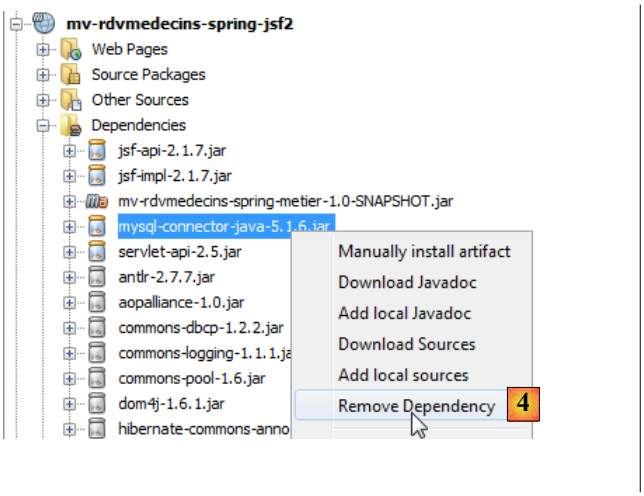
- en [1], on va dans les propriétés du serveur Tomcat,
- en [2], dans l'onglet [Platform], on fixe la valeur de la mémoire qui déborde. Ici, on a mis 1 Go parce qu'on avait une mémoire totale de 8 Go. On pourra mettre 512M (512 mégaoctets) avec une mémoire plus petite.

Ensuite, on met le pilote JDBC de MySQL dans *<tomcat>/lib* où *<tomcat>* est le répertoire d'installation de Tomcat.



- en [1], dans les propriétés de Tomcat, on note son répertoire d'installation <tomcat>,
- dans <tomcat>/lib [2], on place un pilote JDBC récent de MySQL [3].

Ensuite, on enlève la dépendance qu'avait le projet sur le pilote JDBC de MySQL [4].

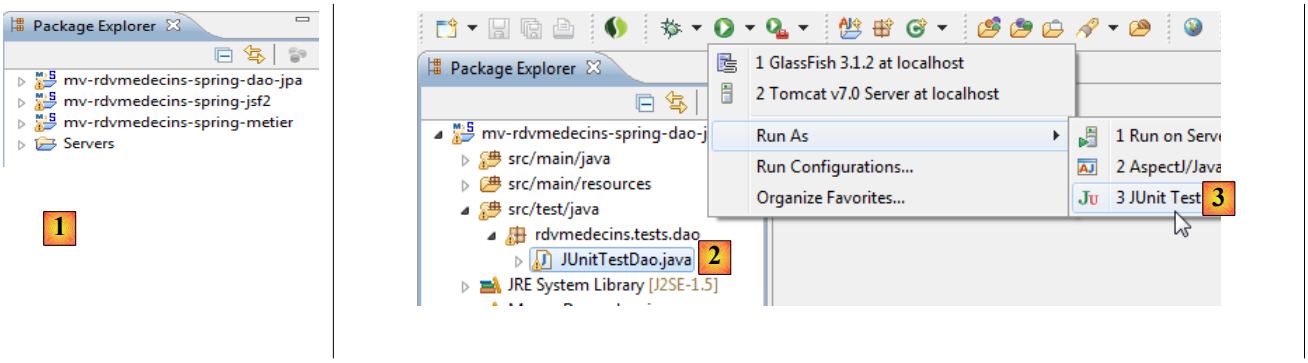


Ceci fait, on teste l'application. On constate qu'on peut faire des chargements / déchargements répétés de l'application. Les problèmes de fuite mémoire ne sont cependant pas résolus. Ils interviennent simplement plus tard.

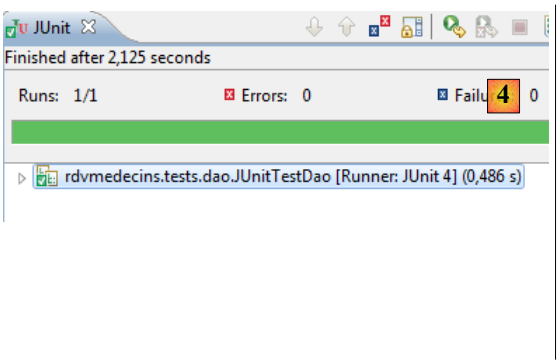
4.4 Conclusion

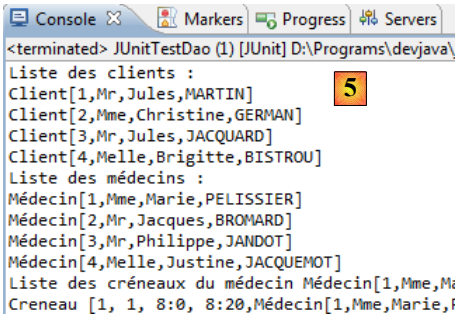
Nous avons porté l'application JSF / EJB / Glassfish vers un environnement JSF / Spring / Tomcat. Cela s'est fait essentiellement par du copier / coller entre les deux projets. Cela a été possible parce que les technologies Spring et EJB3 présentent de fortes similarités. EJB3 a été en effet créé après que Spring se soit montré plus performant que les EJB2. EJB3 a alors repris les bonnes idées de Spring.

4.5 Les tests avec Eclipse

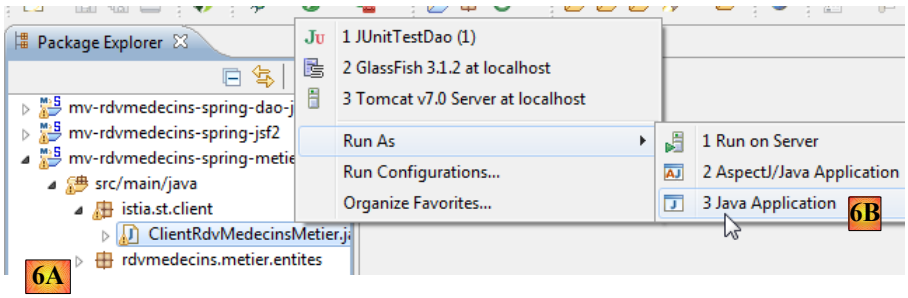


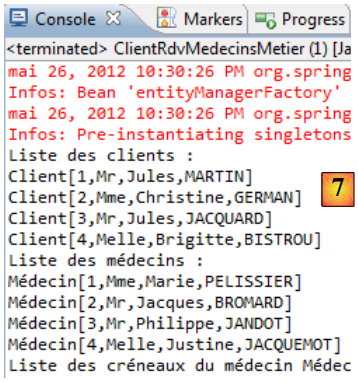
- en [1], on importe les trois projets Spring,
- en [2], on sélectionne le test JUnit de la couche [DAO] et on l'exécute en [3],



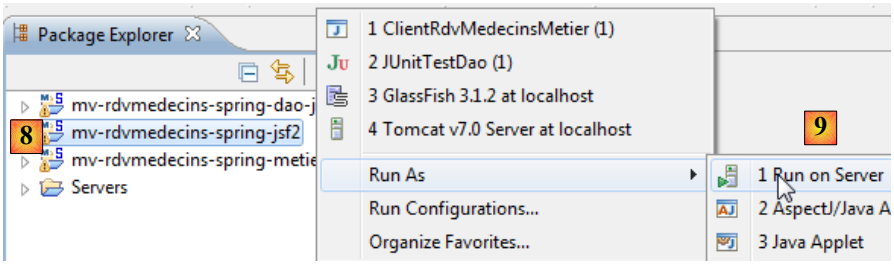


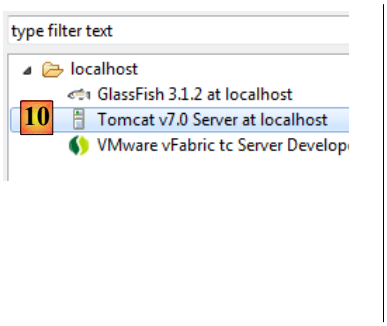
- en [4], le test est réussi,
- en [5], les logs de la console.





- en [6A] [6B], on exécute le client console de la couche [métier],
- en [7], l'affichage console obtenu,





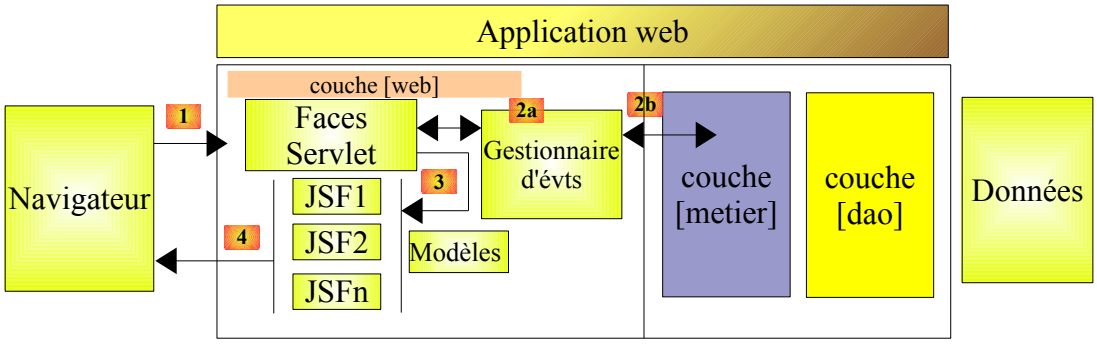
- en [8] [9], on exécute le projet web sur un serveur Tomcat 7 [10],

- en [11], la page d'accueil de l'application est affichée dans le navigateur interne d'Eclipse.

5 Introduction à la bibliothèque de composants PrimeFaces

5.1 La place de Primefaces dans une application JSF

Revenons à l'architecture d'une application JSF telle que nous l'avons étudiée au début de ce document :



Les pages JSF étaient construites avec trois bibliothèques de balises :

```
1.  <html xmlns="http://www.w3.org/1999/xhtml"
2.        xmlns:h="http://java.sun.com/jsf/html"
3.        xmlns:f="http://java.sun.com/jsf/core"
4.        xmlns:ui="http://java.sun.com/jsf/facelets">
```

- ligne 2 : les balises **<h:x>** de l'espace de noms [http://java.sun.com/jsf/html] qui correspondent aux balises HTML,
- ligne 3 : les balises **<f:y>** de l'espace de noms [http://java.sun.com/jsf/core] qui correspondent aux balises JSF,
- ligne 4 : les balises **<ui:z>** de l'espace de noms [http://java.sun.com/jsf/facelets] qui correspondent aux balises des facelets.

Pour construire les pages JSF, nous allons ajouter une quatrième bibliothèque de balises, celles des composants **Primefaces**.

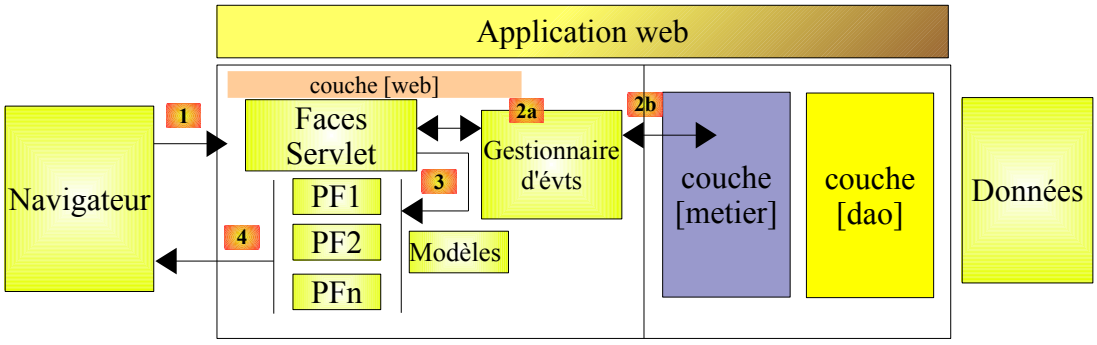
```
1.  <html xmlns="http://www.w3.org/1999/xhtml"
2.        ...
3.  xmlns:p="http://primefaces.org/ui">
```

- ligne 3 : les balises **<p:z>** de l'espace de noms [http://primefaces.org/ui] correspondent aux composants Primefaces.

C'est la **seule modification** qui va apparaître. Elle apparaît donc dans les vues. Les gestionnaires d'événements et les modèles restent ce qu'ils étaient avec JSF. C'est un point important à comprendre.

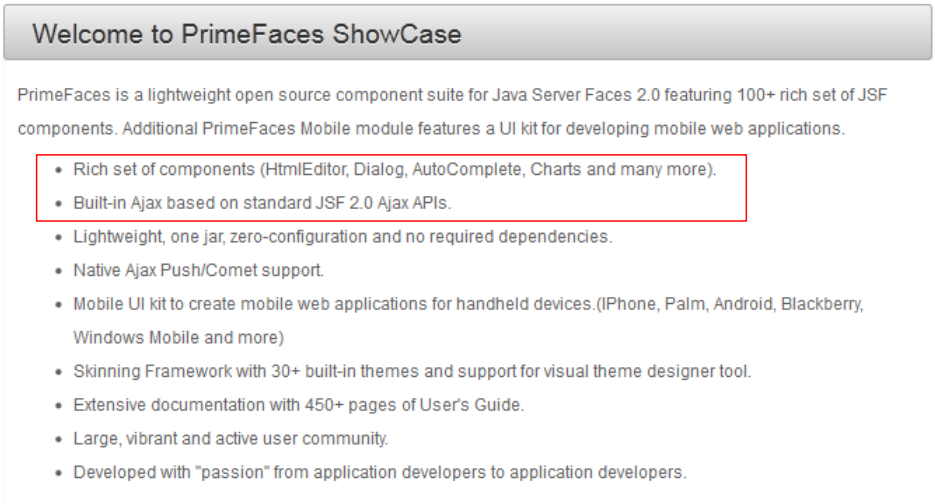
L'utilisation des composants Primefaces permet de créer des interfaces web plus **conviviales** grâce au nombreux composants de cette bibliothèque et plus **fluides** grâce à la technologie **AJAX** qu'elle utilise nativement. On parle alors d'**interfaces riches** ou **RIA** (Rich Internet Application).

L'architecture JSF précédente deviendra l'architecture PF (Primefaces) suivante :



5.2 Les apports de Primefaces

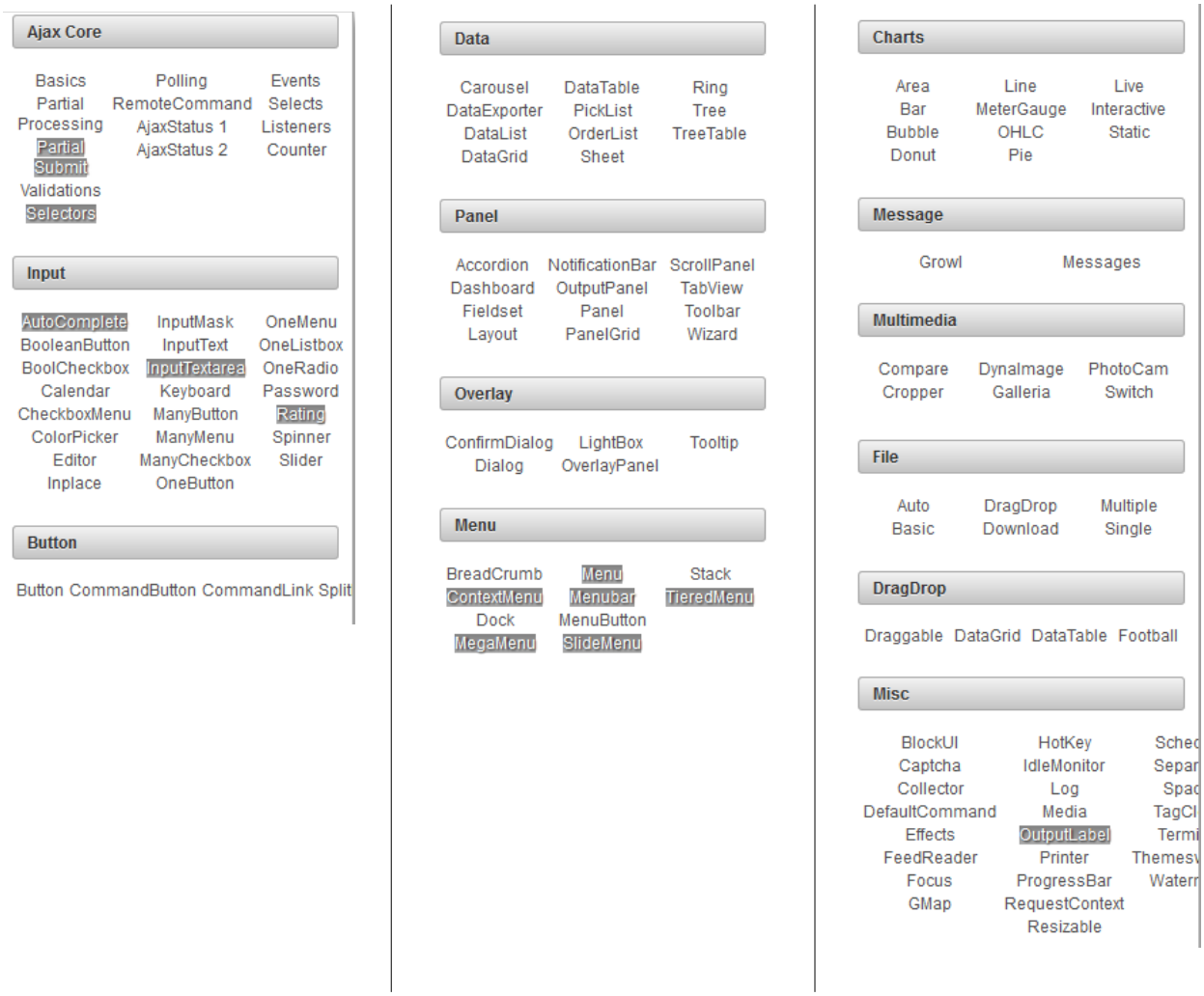
Le site de Primefaces [http://www.primefaces.org/showcase/ui/home.jsf] donne la liste des composants utilisables dans une page PF :



Dans les exemples à venir, nous utiliserons les deux premières caractéristiques de Primefaces :

- certains de la centaine de composants offerts,
- le comportement AJAX natif de ceux-ci.

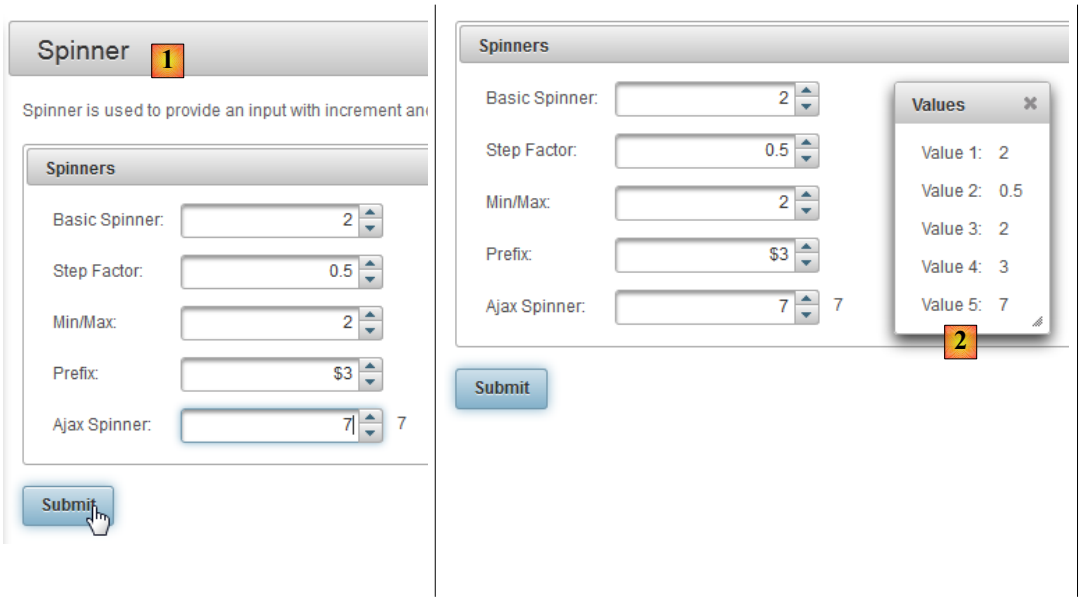
Parmi les composants offerts :



Nous n'utiliserons qu'une quinzaine d'entre-eux dans nos exemples, mais ce sera suffisant pour comprendre les principes de construction d'une page Primefaces.

5.3 Apprentissage de Primefaces

Primefaces offre des exemples d'utilisation de chacun de ses composants. Il suffit de cliquer sur son lien. Regardons un exemple :



- en [1], l'exemple pour le composant [Spinner],
- en [2], la boîte de dialogue affichée après un clic sur le bouton [Submit].

Il y a là pour nous, trois nouveautés :

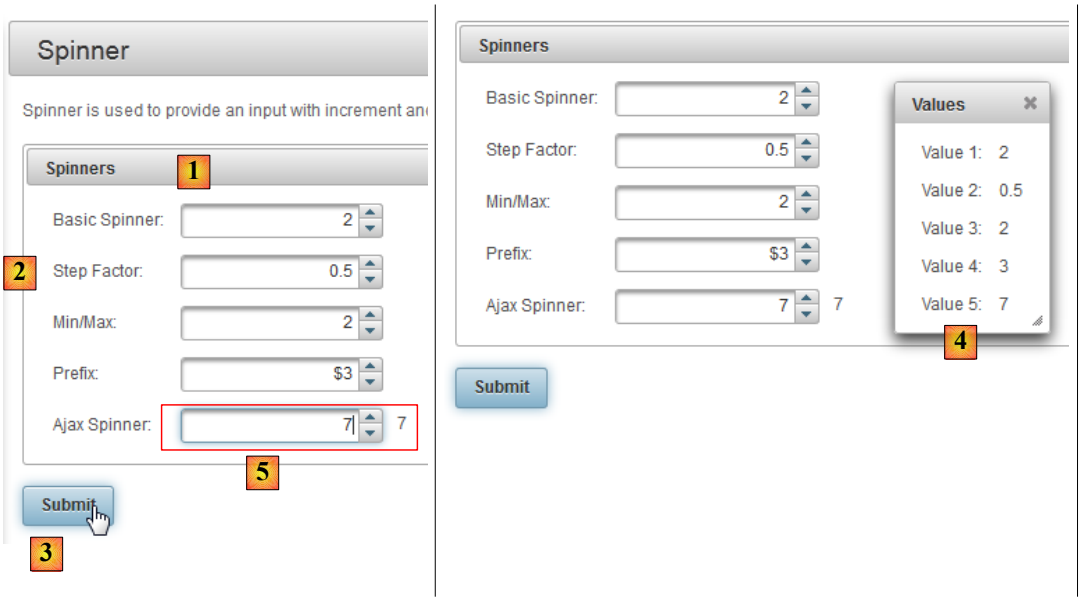
- le composant [Spinner] qui n'existe pas de base en JSF,
- idem pour la boîte de dialogue,
- enfin, le POST provoqué par le [Submit] est **ajaxifié**. Si on observe bien le navigateur lors du POST, on ne voit pas le sablier. La page n'est pas rechargée. Elle est simplement modifiée : un nouveau composant, ici la boît de dialogue apparaît dans la page.

Voyons comment tout cela est produit. Le code XHTML de l'exemple est le suivant :

```
1.  <h:form>
2.      <p:panel header="Spinners">
3.          <h:panelGrid id="grid" columns="2" cellpadding="5">
4.              <h:outputLabel for="spinnerBasic" value="Basic Spinner: " />
5.              <p:spinner id="spinnerBasic" value="#{spinnerController.number1}"/>
6.              <h:outputLabel for="spinnerStep" value="Step Factor: " />
7.              <p:spinner id="spinnerStep" value="#{spinnerController.number2}" stepFactor="0.25"/>
8.              <h:outputLabel for="minmax" value="Min/Max: " />
9.              <p:spinner id="minmax" value="#{spinnerController.number3}" min="0" max="100"/>
10.             <h:outputLabel for="prefix" value="Prefix: " />
11.             <p:spinner id="prefix" value="0" prefix="$" min="0" value="#{spinnerController.number4}"/>
12.             <h:outputLabel for="ajaxspinner" value="Ajax Spinner: " />
13.             <p:outputPanel>
14.                 <p:spinner id="ajaxspinner" value="#{spinnerController.number5}">
15.                     <p:ajax update="ajaxspinnervalue" process="@this" />
16.                 </p:spinner>
17.                 <h:outputText id="ajaxspinnervalue" value="#{spinnerController.number5}"/>
18.             </p:outputPanel>
19.         </h:panelGrid>
20.     </p:panel>
21.     <p:commandButton value="Submit" update="display" oncomplete="dialog.show()" />
22.
23.     <p:dialog header="Values" widgetVar="dialog" showEffect="fold" hideEffect="fold">
24.         ...
25.     </p:dialog>
26. </h:form>
```

Tout d'abord, constatons qu'on retrouve des balises JSF classiques : **<h:form>** ligne 1, **<h:panelGrid>** ligne 3, **<h:outputLabel>** ligne 4. Certaines balises JSF sont reprises par PF et enrichies : <**p:commandButton**> ligne 21. Ensuite, on trouve des balises PF de mise en forme : **<p:panel>** ligne 2, **<p:outputPanel>** ligne 13, **<p:dialog>** ligne 23. Enfin, on a des balises de saisie : <**p:spinner**> ligne 5.

Analysons ce code en correspondance avec la vue :



- en [1], le composant obtenu avec la balise <p:panel> de la ligne 2,
- en [2], le champ de saisie obtenu par la combinaison d'une balise <p:outputLabel> et <p:spinner>, lignes 6 et 7,
- en [3], le bouton du POST obtenu avec la balise <p:commandButton> de la ligne 21,
- en [4], la boîte de dialogue des lignes 23-25,
- en [5], un conteneur invisible pour deux composants. Il est créé par la balise <p:outputPanel> de la ligne 13.

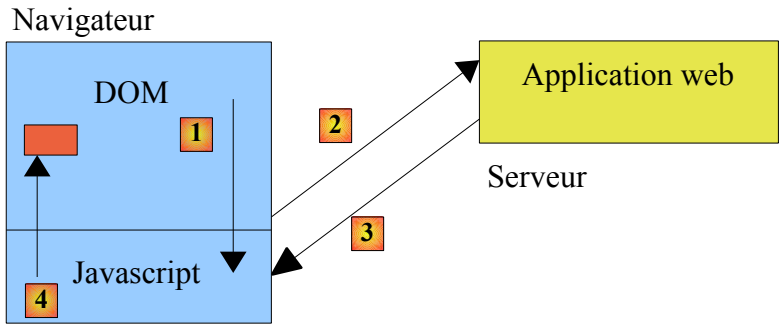
Analysons le code suivant qui met en oeuvre une action AJAX :

```
1.              <h:outputLabel for="ajaxspinner" value="Ajax Spinner: " />
2.              <p:outputPanel>
3.                  <p:spinner id="ajaxspinner" value="#{spinnerController.number5}">
4.                      <p:ajax update="ajaxspinnervalue" process="@this" />
5.                  </p:spinner>
6.                  <h:outputText id="ajaxspinnervalue" value="#{spinnerController.number5}"/>
7.  </p:outputPanel>
```

Ce code génère la vue suivante :

- ligne 1 : affiche le texte [1]. Est en même temps un libellé pour le composant d'*id=ajaxspinner* (attribut **for**). Ce composant est celui de la ligne 3(attribut **id**),,
- lignes 3-5 : affichent le composant [2]. Ce composant est un composant de saisie / affichage associé au modèle `#{spinnerController.number5}` (attribut **value**),
- ligne 6 : affiche le composant [3]. Ce composant est un composant d'affichage lié au modèle `#{spinnerController.number5}` (attribut **value**),
- ligne 4 : la balise <**p:ajax**> ajoute un comportement AJAX au *spinner*. A chaque fois que celui-ci change de valeur, un POST de cette valeur (attribut **process="@this"**) est fait au modèle `#{spinnerController.number5}`. Ceci fait, une mise à jour de la page est faite (attribut **update**). Cet attribut a pour valeur l'**id** d'un composant de la page, ici celui de la ligne 6. Le composant cible de l'attribut **update** est alors mis à jour avec le modèle. Celui-ci est de nouveau `#{spinnerController.number5}`, donc la valeur du *spinner*. Ainsi la zone [3] suit les saisies de la zone [2].

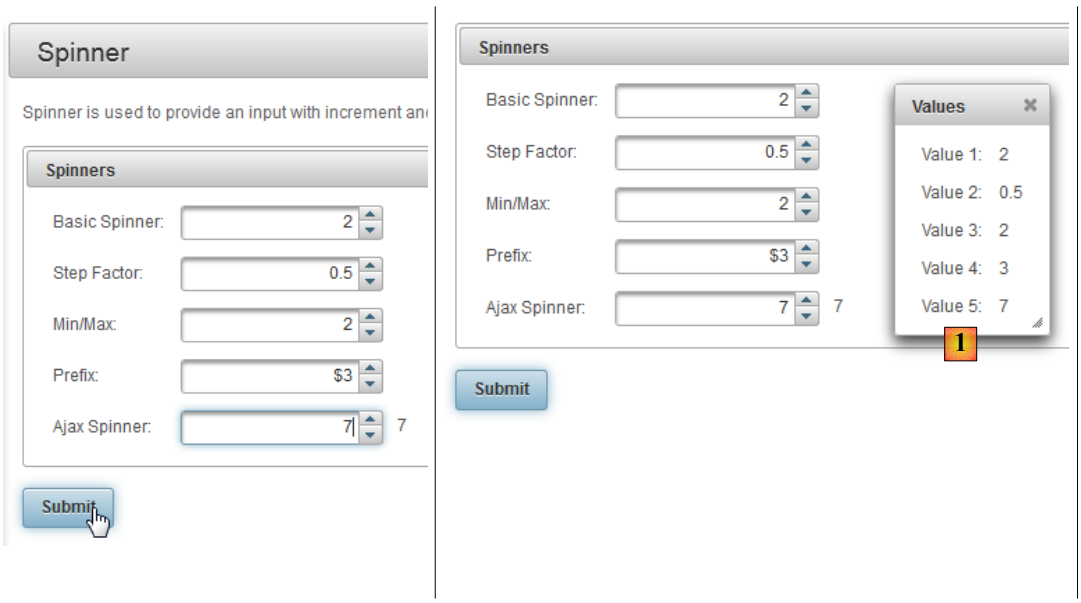
On a là un comportement AJAX, acronyme qui signifie **A**synchronous **J**avascript **A**nd **X**ML. De façon générale, un comportement AJAX est le suivant :



- le navigateur affiche une page HTML qui contient du code Javascript (J de AJAX). Les éléments de la page forment un objet Javascript qu'on appelle le DOM (Document Object Model),
- le serveur loge l'application web qui a produit cette page,
- en [1], un événement se produit dans la page. Par exemple l'incrémentation du *spinner*. Cet événement est géré par du Javascript,
- en [2], le Javascript fait un POST à l'application web. Il le fait de façon asynchrone (le **A** de **A**JAX). L'utilisateur peut continuer à travailler avec la page. Elle n'est pas gelée mais on peut si nécessaire la geler. Le POST met à jour le modèle de la page à partir des valeurs postées, ici le modèle `#{spinnerController.number5}`,
- en [3], l'application web renvoie au Javascript, une réponse XML (le X de AJAX) ou JSON (JavaScript Object Notation),
- en [4], le Javascript utilise cette réponse pour mettre à jour une zone précise du DOM, ici la zone d'*id=ajaxspinnervalue*.

Lorsqu'on utilise JSF et Primefaces, le Javascript est généré par Primefaces. Cette bibliothèque s'appuie sur la bibliothèque Javascript **JQuery**. De même les composants Primefaces s'appuient sur ceux de la bibliothèque de composants **JQuery UI** (User Interface). Donc **JQuery** est à la base de Primefaces.

Revenons à notre exemple et présentons maintenant le POST du bouton [Submit] :



Le code associé au POST est le suivant :

```
1.  <p:commandButton value="Submit" update="display" oncomplete="dialog.show()" />
2.
3.     <p:dialog header="Values" widgetVar="dialog" showEffect="fold" hideEffect="fold">
4.        <h:panelGrid id="display" columns="2" cellpadding="5">
5.           <h:outputText value="Value 1: " />
6.           <h:outputText value="#{spinnerController.number1}" />
7.
8.           <h:outputText value="Value 2: " />
9.           <h:outputText value="#{spinnerController.number2}" />
10.
```

```
11.           <h:outputText value="Value 3: " />
12.           <h:outputText value="#{spinnerController.number3}" />
13.
14.           <h:outputText value="Value 4: " />
15.           <h:outputText value="#{spinnerController.number4}" />
16.
17.           <h:outputText value="Value 5: " />
18.           <h:outputText value="#{spinnerController.number5}" />
19.       </h:panelGrid>
20.     </p:dialog>
21.
22. </h:form>
```

- ligne 1 : le POST est provoqué par le bouton de la ligne 1. Dans Primefaces, les balises qui provoquent un POST le font par défaut sous la forme d'un appel AJAX. C'est pourquoi ces balises ont un attribut **update** pour indiquer la zone à mettre à jour, une fois la réponse du serveur reçue. Ici, la zone mise à jour est le **panelGrid** de la ligne 4. Donc au retour du POST, cette zone va être mise à jour par les valeurs postées au modèle. Cependant, elles sont à l'intérieur d'une boîte de dialogue non visible par défaut. C'est l'attribut **oncomplete** de la ligne 1 qui l'affiche. Cet événement se produit à la fin du traitement du POST. La valeur de cet attribut est du code Javascript. Ici, on affiche la boîte de dialogue qui a l'**id=dialog**, donc celle de la ligne 3 (attribut **widgetVar**),
- ligne 3 : on voit divers attributs de la boîte de dialogue. Il faut expérimenter pour voir ce qu'ils font.

Nous avons parlé du modèle mais ne l'avons pas encore présenté. C'est celui-ci :

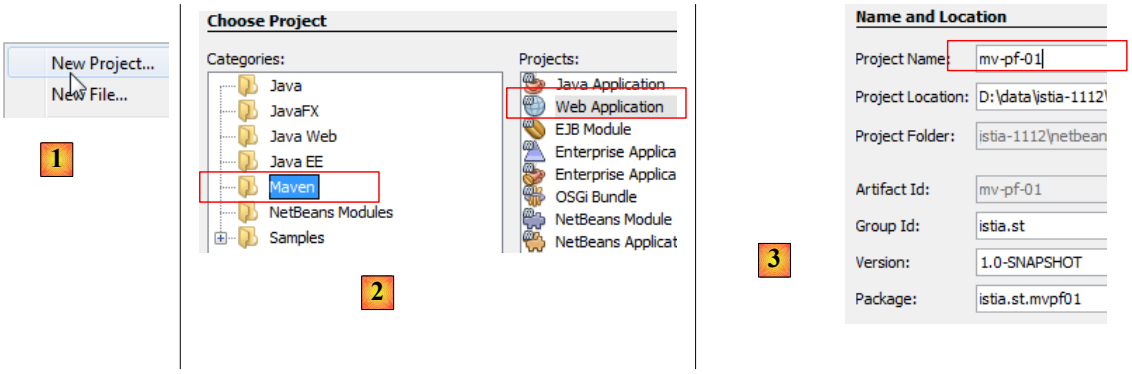
```
1. public class SpinnerController {
2.
3.     private int number1;
4.     private double number2;
5.     private int number3;
6.     private int number4;
7.     private int number5;
8.
9.     // getters et setters
10. ...
11. }
```

De façon générale, on peut procéder comme suit :

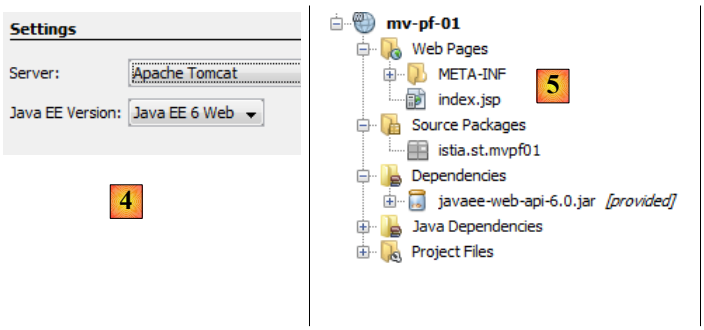
- repérer le composant Primefaces qu'on veut utiliser,
- étudier son exemple. Les exemples de Primefaces sont bien faits et facilement compréhensibles.

5.4 Un premier projet Primefaces : mv-pf-01

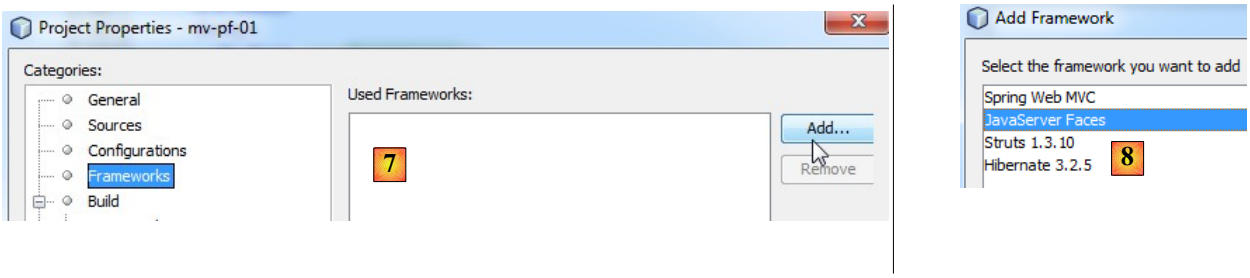
Construisons un projet web Maven avec Netbeans :



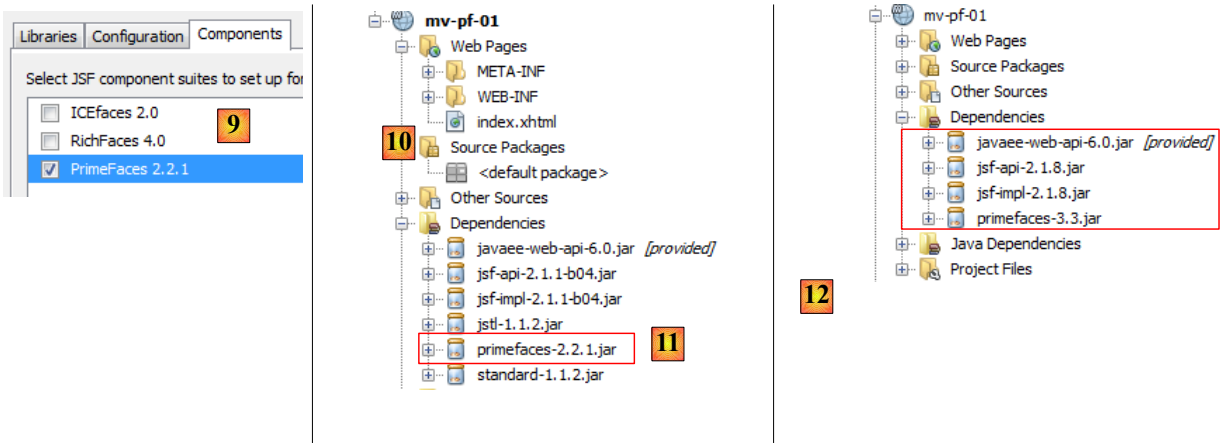
- [1, 2, 3] : on construit un projet Maven de type [Web Application],



- [4] : le serveur sera Tomcat,
- en [5], le projet généré,
- en [6], on le nettoie du fichier [index.jsp] et du paquetage Java,



- en [7, 8] : dans les propriétés du projet, on ajoute un support pour Java server Faces,



- en [9], dans l'onglet [Components] on choisit la bibliothèque de composants Primefaces. Netbeans offre un support pour d'autres bibliothèques de composants : ICEFaces et RichFaces.
- en[10], le projet généré. En [11], on notera la dépendance sur Primefaces.

En clair, un projet Primefaces est un projet JSF classique auquel on a ajouté une dépendance sur Primefaces. Rien de plus.

Ayant compris cela, nous modifions le fichier [pom.xml] pour travailler avec les dernières versions des bibliothèques :

```
1.      <dependency>
2.        <groupId>com.sun.faces</groupId>
3.        <artifactId>jsf-impl</artifactId>
4.        <version>2.1.8</version>
5.        <scope>compile</scope>
6.      </dependency>
7.      <dependency>
```

```
        <groupId>org.primefaces</groupId>
        <artifactId>primefaces</artifactId>
        <version>3.3</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>6.0</version>
        <scope>provided</scope>
      </dependency>
    </dependencies>
    <repositories>
      <repository>
        <id>jsf20</id>
        <name>Repository for library Library[jsf20]</name>
        <url>http://download.java.net/maven/2/</url>
      </repository>
      <repository>
        <id>primefaces</id>
        <name>Repository for library Library[primefaces]</name>
        <url>http://repository.primefaces.org/</url>
      </repository>
  </repositories>
```

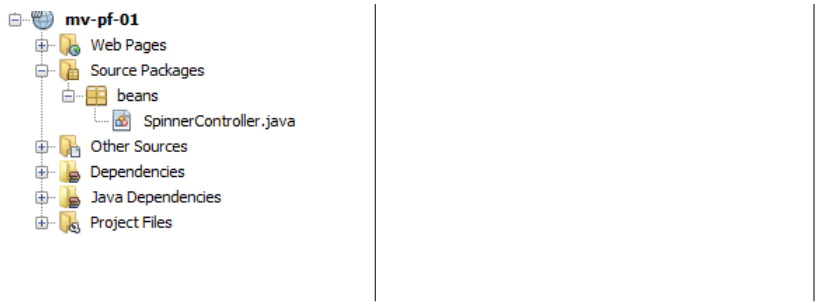
Lignes 26-30, on notera le dépôt Maven pour Primefaces. Ces modifications faites, on construit le projet pour lancer le téléchargement des dépendances. On obtient alors le projet [12].

Maintenant, essayons de reproduire l'exemple que nous avons étudié. La page [index.html] devient la suivante :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>Spinner</title>
  </h:head>
  <h:body>
    <!-- formulaire -->
    <h:form>
      <p:panel header="Spinners">
        <h:panelGrid id="grid" columns="2" cellpadding="5">
          <h:outputLabel for="spinnerBasic" value="Basic Spinner: " />
          <p:spinner id="spinnerBasic" value="#{spinnerController.number1}"/>
          <h:outputLabel for="spinnerStep" value="Step Factor: " />
          <p:spinner id="spinnerStep" value="#{spinnerController.number2}" stepFactor="0.25"/>
          <h:outputLabel for="minmax" value="Min/Max: " />
          <p:spinner id="minmax" value="#{spinnerController.number3}" min="0" max="100"/>
          <h:outputLabel for="prefix" value="Prefix: " />
          <p:spinner id="prefix" prefix="$" min="0" value="#{spinnerController.number4}"/>
          <h:outputLabel for="ajaxspinner" value="Ajax Spinner: " />
          <p:outputPanel>
            <p:spinner id="ajaxspinner" value="#{spinnerController.number5}">
              <p:ajax update="ajaxspinnervalue" process="@this" />
            </p:spinner>
            <h:outputText id="ajaxspinnervalue" value="#{spinnerController.number5}"/>
          </p:outputPanel>
        </h:panelGrid>
      </p:panel>
      <p:commandButton value="Submit" update="display" oncomplete="dialog.show()" />
      <!-- boîte de dialogue -->
      <p:dialog header="Values" widgetVar="dialog" showEffect="fold" hideEffect="fold">
        <h:panelGrid id="display" columns="2" cellpadding="5">
          <h:outputText value="Value 1: " />
          <h:outputText value="#{spinnerController.number1}" />
          <h:outputText value="Value 2: " />
          <h:outputText value="#{spinnerController.number2}" />
          <h:outputText value="Value 3: " />
          <h:outputText value="#{spinnerController.number3}" />
          <h:outputText value="Value 4: " />
          <h:outputText value="#{spinnerController.number4}" />
```

```
45.           <h:outputText value="Value 5: " />
46.           <h:outputText value="#{spinnerController.number5}" />
47.         </h:panelGrid>
48.       </p:dialog>
49.     </h:form>
50.   </h:body>
51. </html>
```

On n'oubliera pas la ligne 5 qui déclare l'espace de noms de la bibliothèque de balises Primefaces. On rajoute au projet, le bean qui sert de modèle à la page :

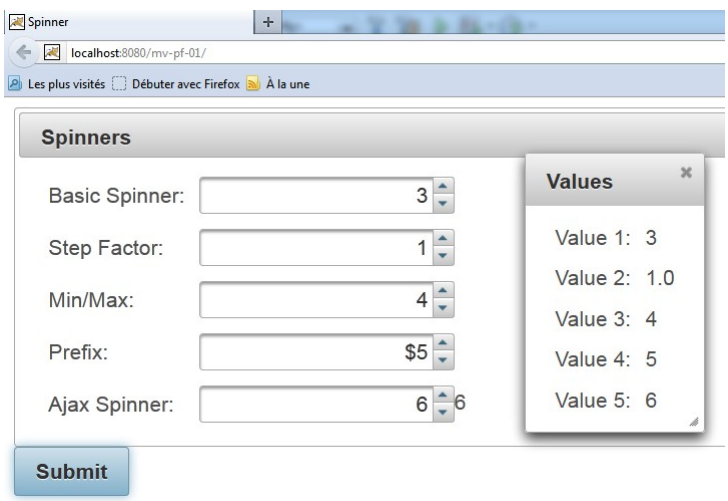


Le bean est le suivant :

```
1.  package beans;
2.
3.  import javax.faces.bean.RequestScoped;
4.  import javax.faces.bean.ManagedBean;
5.
6.  @ManagedBean
7.  @RequestScoped
8.  public class SpinnerController {
9.
10.   // modèle
11.   private int number1;
12.   private double number2;
13.   private int number3;
14.   private int number4;
15.   private int number5;
16.
17.   // getters et setters
18.   ...
19. }
```

La classe est un bean (ligne 6) de portée requête (ligne 7). Comme on n'a pas indiqué de nom, le bean porte le nom de la classe avec le premier caractère en minuscule : **spinnerController**.

Lorsqu'on exécute le projet, on obtient la chose suivante :



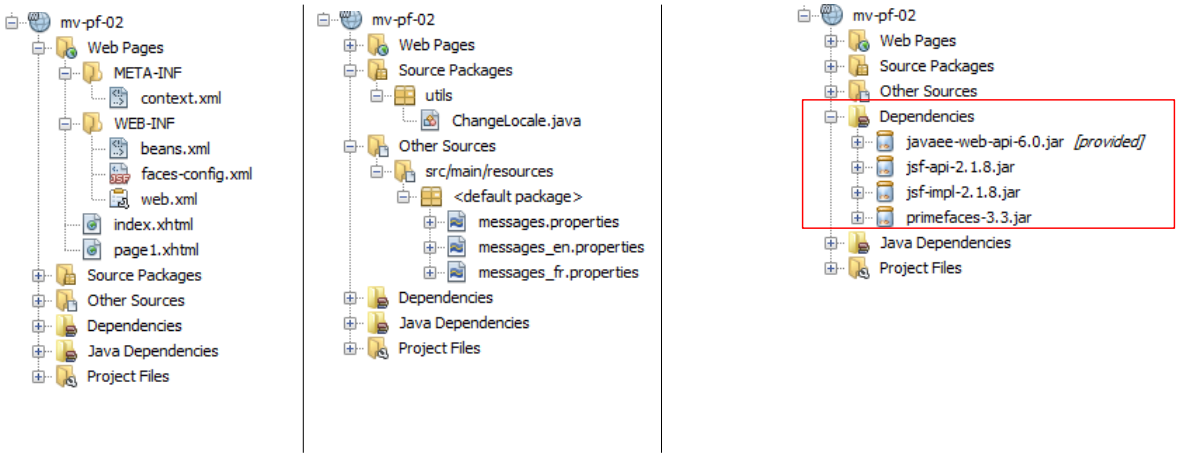
Nous venons ainsi de montrer comment tester un exemple pris sur le site de Primefaces. Tous les exemples peuvent être testés de cette façon.

Dans la suite, nous allons nous intéresser qu'à certains composants de Primefaces. Nous allons tout d'abord reprendre les exemples étudiés avec JSF et remplacer certaines balises JSF par des balises Primefaces. L'aspect des pages sera un peu modifié, elles auront un comportement AJAX mais les beans associés n'auront pas à être changés. Dans chacun des exemples à venir, nous nous contentons de présenter le code XHTML des pages et les copies d'écran associées. Le lecteur est invité à tester les exemples afin de déceler les différences entre les pages JSF et les pages PF.

5.5 Exemple mv-pf-02 : gestionnaire d'événement – internationalisation - navigation entre pages

Ce projet est le portage du projet JSF [mv-jsf2-02] (paragraphe 2.4, page 36) :

Le projet Netbeans est le suivant :



La page [index.html] est la suivante :

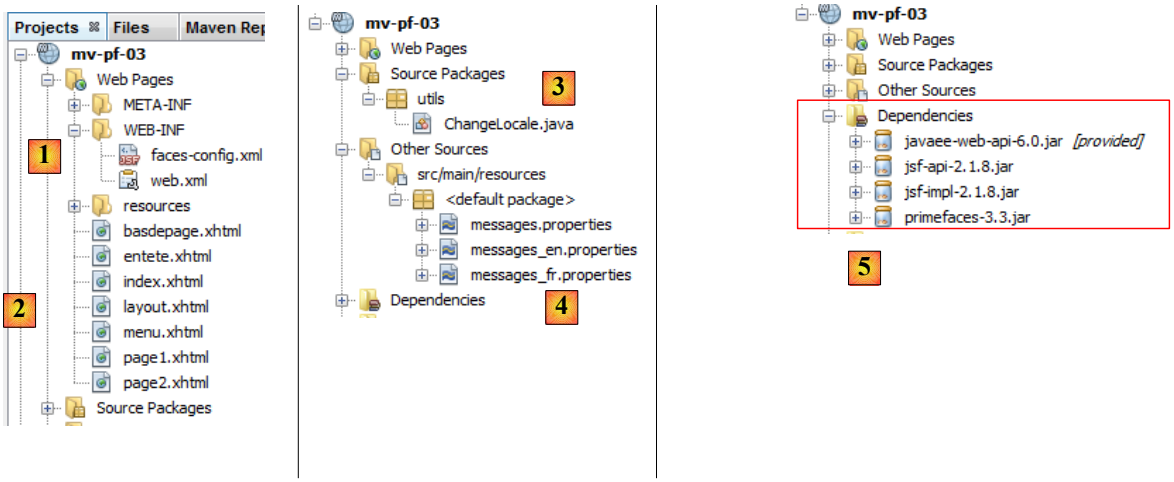
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <f:view locale="#{changeLocale.locale}">
9.      <h:head>
10.       <title><h:outputText value="#{msg['welcome.titre']}" /></title>
11.     </h:head>
12.     <body>
13.       <h:form id="formulaire">
14.         <h:panelGrid columns="2">
15.           <p:commandLink value="#{msg['welcome.langue1']}"
    action="#{changeLocale.setFrenchLocale}" ajax="false"/>
16.           <p:commandLink value="#{msg['welcome.langue2']}"
    action="#{changeLocale.setEnglishLocale}" ajax="false"/>
17.         </h:panelGrid>
18.         <h1><h:outputText value="#{msg['welcome.titre']}" /></h1>
19.          <p:commandLink value="#{msg['welcome.page1']}" action="page1" ajax="false"/>
20.       </h:form>
21.     </body>
22.   </f:view>
23. </html>
```

Aux lignes 15, 16 et 19 les balises **<h:commandLink>** ont été remplacées par des balises **<p:commandLink>**. Cette balise a un comportement AJAX par défaut qu'on peut inhiber en mettant l'attribut **ajax="false"**. Donc ici, les balises *<p:commandLink>* se comportent comme des balises *<h:commandLink>* : il y aura un rechargement de la page lors d'un clic sur ces liens.

5.6 Exemple mv-pf-03 : mise en page à l'aide des facelets

Ce projet présente la création de pages XHTML à l'aide des modèles facelets de l'exemple [mv-jsf2-09] (paragraphe 2.11, page 146) :

Le projet Netbeans est le suivant :



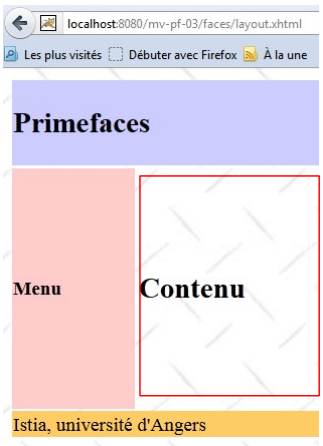
- en [1], les fichiers de configuration du projet JSF,
- en [2], les pages XHTML,
- en [3], le bean support pour le changement de langues,
- en [4], les fichiers de messages,
- en [5], les dépendances.

Les pages du projet ont pour modèle la page [layout.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:p="http://primefaces.org/ui"
7.        xmlns:f="http://java.sun.com/jsf/core"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <f:view locale="#{changeLocale.locale}">
10.     <h:head>
11.       <title>JSF</title>
12.       <h:outputStylesheet library="css" name="styles.css"/>
13.     </h:head>
14.     <h:body style="background-image: url('${request.contextPath}/resources/images/standard.jpg');">
```

```
15.       <h:form id="formulaire">
16.         <table style="width: 600px">
17.           <tr>
18.             <td colspan="2" bgcolor="#ccccff">
19.               <ui:include src="entete.xhtml"/>
20.             </td>
21.           </tr>
22.           <tr>
23.             <td style="width: 100px; height: 200px" bgcolor="#ffcccc">
24.               <ui:include src="menu.xhtml"/>
25.             </td>
26.             <td>
27.               <p:outputPanel id="contenu">
28.                 <ui:insert name="contenu" >
29.                   <h2>Contenu</h2>
30.                 </ui:insert>
31.               </p:outputPanel>
32.             </td>
33.           </tr>
34.           <tr bgcolor="#ffcc66">
35.             <td colspan="2">
36.               <ui:include src="basdepage.xhtml"/>
37.             </td>
38.           </tr>
39.         </table>
40.       </h:form>
41.     </h:body>
42.   </f:view>
43. </html>
```

- ligne 9 : une balise **<f:view>** encadre toute la page afin de profiter de l'internationalisation qu'elle permet,
- ligne 15 : un formulaire d'**id formulaire.** Ce formulaire forme le corps de la page. Dans ce corps, il n'y a qu'une partie dynamique, celle des lignes 28-30. C'est là que viendra s'insérer la partie variable de la page :



- la zone encadrée ci-dessus sera mise à jour par des appels AJAX. Afin de l'identifier, nous l'avons incluse dans un conteneur Primefaces généré par la balise **<p:outputPanel>** (ligne 27). Et ce conteneur a été nommé **contenu** (attribut **id**). Comme il se trouve dans un formulaire qui est lui-même un conteneur de nom **formulaire**, le nom complet de la zone dynamique est **:formulaire:contenu**. Le premier **:** indique qu'on part de la racine du document, puis on passe dans le conteneur de nom **formulaire**, puis dans le conteneur de nom **contenu**. Une difficulté avec AJAX est de nommer correctement les zones à mettre à jour par un appel AJAX. Le plus simple est de regarder le code source de la page HTML reçue :

```
1.             <td><span id="formulaire:contenu">
2.                   <h2>Contenu</h2></span>
3.  </td>
```

Ci-dessus, on voit que la balise **<h:outputPanel>** a généré une balise HTML . Dans cet exemple, le nom relatif **formulaire:contenu** (sans le **:** de départ) et le nom complet **:formulaire:contenu** (avec le **:** de départ) désignent le même objet.

On retiendra que les appels AJAX (<p:commandButton>, <p:commandLink>) mettant à jour la zone dynamique auront l'attribut **update=":formulaire:contenu"**.

La page [index.xhtml] est l'unique page affichée par le projet :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <ui:composition template="layout.xhtml">
9.      <ui:define name="contenu">
10.       <ui:fragment rendered="#{requestScope.page1 || requestScope.page2==null}">
11.         <ui:include src="page1.xhtml"/>
12.       </ui:fragment>
13.       <ui:fragment rendered="#{requestScope.page2}">
14.         <ui:include src="page2.xhtml"/>
15.       </ui:fragment>
16.     </ui:define>
17.   </ui:composition>
18. </html>
```

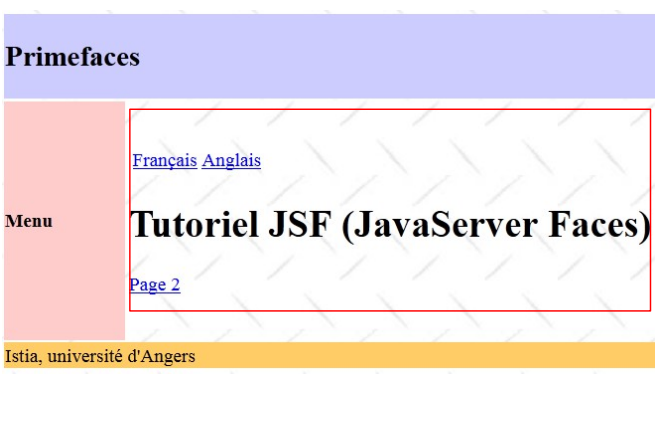
- ligne 8, le modèle de [index.xhtml] est la page [layout.xhtml] qu'on vient de présenter,
- ligne 9, c'est la zone d'id **contenu** qui est mise à jour par [index.html]. Dans cette zone, il y a deux fragments :
    - le fragment [page1.xhtml] ligne 11 ;
    - le fragment [page2.xhtml] ligne 14.

  Ces deux fragments sont exclusifs l'un de l'autre.
- ligne 10 : le fragment [page1.xhtml] est affiché si la requête a l'attribut **page1** à **true** ou si l'attribut **page2** n'existe pas. C'est le cas de la toute première requête où aucun de ces attributs ne sera présent dans la requête. Dans ce cas, on affichera le fragment [page1.xhtml],
- ligne 11, le fragment [page2.xhtml] est affiché si la requête a l'attribut **page2** à **true**

Le fragment [page1.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.
9.    <body>
10.     <h:panelGrid columns="2">
11.       <p:commandLink value="#{msg['page1.langue1']}" actionListener="#{changeLocale.setFrenchLocale}"
    ajax="true" update=":formulaire:contenu"/>
12.       <p:commandLink value="#{msg['page1.langue2']}" actionListener="#{changeLocale.setEnglishLocale}"
    ajax="true" update=":formulaire:contenu"/>
13.     </h:panelGrid>
14.     <h1><h:outputText value="#{msg['page1.titre']}" /></h1>
15.      <p:commandLink value="#{msg['page1.lien']}" update=":formulaire:contenu">
16.       <f:setPropertyActionListener value="#{true}" target="#{requestScope.page2}" />
17.     </p:commandLink>
18.   </body>
19. </html>
```

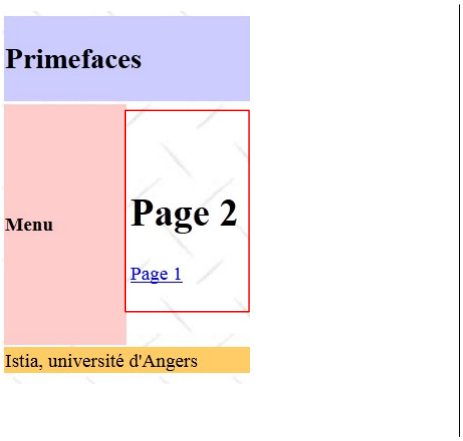
et affiche le contenu suivant :



- lignes 11 et 12, les deux liens pour changer la langue. Ces deux liens provoquent des appels AJAX (*ajax=true*). C'est la valeur par défaut. On peut donc ne pas mettre l'attribut *ajax=true*. Nous ne le ferons plus par la suite. On notera que ces deux liens mettent à jour la zone **:formulaire:contenu** (attribut *update*), celle qui est encadrée ci-dessus,
- ligne 15 : un lien AJAX de navigation qui là encore met à jour la zone **:formulaire:contenu**,
- ligne 16 : on utilise la balise <*h:setPropertyActionListener*> pour mettre l'attribut **page2** dans la requête avec la valeur **true**. Cela va avoir pour effet d'afficher le fragment [page2.xhtml] (ligne 6 ci-dessous) dans la page [index.xhtml] :

```
1.    <ui:composition template="layout.xhtml">
2.      <ui:define name="contenu">
3.        <ui:fragment rendered="#{requestScope.page1 || requestScope.page2==null}">
4.          <ui:include src="page1.xhtml"/>
5.        </ui:fragment>
6.        <ui:fragment rendered="#{requestScope.page2}">
7.          <ui:include src="page2.xhtml"/>
8.        </ui:fragment>
9.      </ui:define>
10. </ui:composition>
```

Le fragment [page2.xhtml] est analogue :



Le code de [page2.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.
9.    <body>
10.     <h1><h:outputText value="#{msg['page2.entete']}"/></h1>
11.     <p:commandLink value="#{msg['page2.lien']}" update=":formulaire:contenu">
```

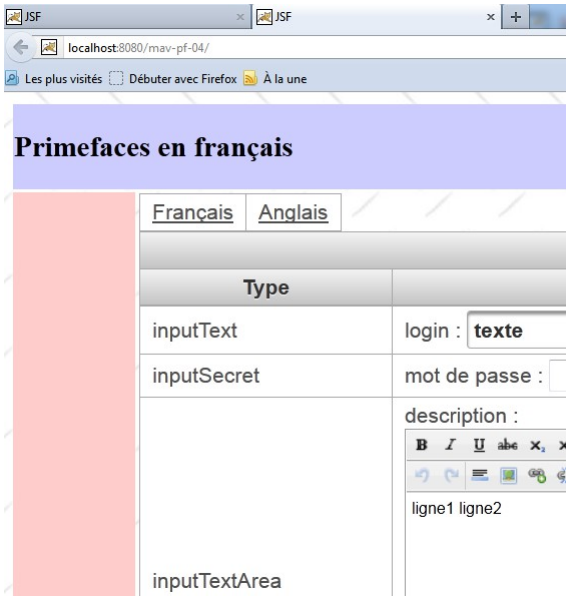
```
12.       <f:setPropertyActionListener value="#{true}" target="#{requestScope.page1}" />
13.     </p:commandLink>
14.   </body>
15. </html>
```

De cet exemple, nous retiendrons les points suivants pour la suite :

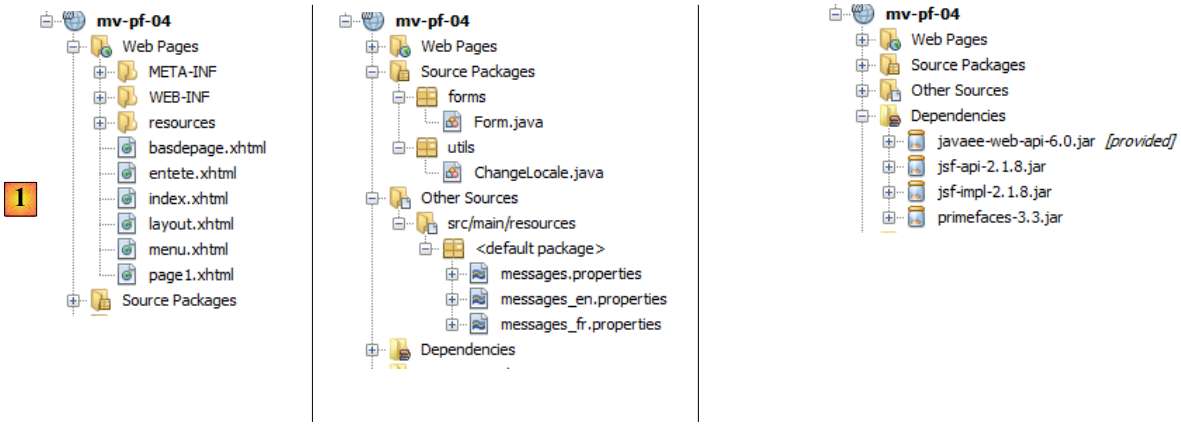
- nous utiliserons le modèle [layout.xhtml] comme modèle des pages,
- la zone dynamique sera identifiée par l'id **:formulaire:contenu** et sera mise à jour par des appels AJAX.

5.7 Exemple mv-pf-04 : formulaire de saisie

Ce projet est le portage du projet JSF2 [mv-jsf2-03] (cf paragraphe 2.5, page 57) :



Le projet Netbeans est le suivant :



Ci-dessus, en [1], les pages XHTML du projet. La mise en page est assurée par le modèle [layout.xhtml] étudié précédemment. La page [index.xhtml] est l'unique page du projet. Elle s'affiche dans la zone **:formulaire:contenu**. Son code est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
```

```
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <ui:composition template="layout.xhtml">
9.      <ui:define name="contenu">
10.       <ui:include src="page1.xhtml"/>
11.     </ui:define>
12.   </ui:composition>
13. </html>
```

Elle se contente d'afficher le frament [page1.xhtml]. Celui-ci est l'équivalent du formulaire étudié dans l'exemple [mv-jsf2-03]. Rappelons que celui-ci avait pour but de présenter les balises JSF de saisie. Ces balises ont été ici remplacées par des balises Primefaces.

PanelGrid

Pour mettre en forme les éléments de [page1.xhtml], nous utilisons la balise <**p:panelGrid**>. Par exemple, pour les deux liens des langues :

```
1.  <!-- langues -->
2.      <p:panelGrid columns="2">
3.        <p:commandLink value="#{msg['form.langue1']}" actionListener="#{changeLocale.setFrenchLocale}"
    update=":formulaire:contenu"/>
4.        <p:commandLink value="#{msg['form.langue2']}" actionListener="#{changeLocale.setEnglishLocale}"
    update=":formulaire:contenu"/>
5.      </p:panelGrid>
```

Cela donne le rendu suivant :

Une autre forme de la balise <**p:panelGrid**> est la suivante :

```
1.  <p:panelGrid>
2.
3.        <f:facet name="header">
4.          <p:row>
5.            <p:column colspan="3"><h:outputText value="#{msg['form.titre']}"/></p:column>
6.          </p:row>
7.          <p:row>
8.            <p:column><h:outputText value="#{msg['form.headerCol1']}"/></p:column>
9.            <p:column><h:outputText value="#{msg['form.headerCol2']}"/></p:column>
10.           <p:column><h:outputText value="#{msg['form.headerCol3']}"/></p:column>
11.         </p:row>
12.       </f:facet>
13.
14.       <p:row>
15.         <p:column>
16.           <h:outputText value="inputText"/>
17.         </p:column>
18.         <p:column>
19.           <h:outputLabel for="inputText" value="#{msg['form.loginPrompt']}" />
20.           <p:inputText id="inputText" value="#{form.inputText}"/>
21.         </p:column>
22.         <p:column>
23.           <h:outputText id="inputTextValue" value="#{form.inputText}"/>
24.         </p:column>
25.       </p:row>
26. ...
27.     <f:facet name="footer">
28.         <p:row>
29.           <p:column colspan="3">
30.             <div align="center">
31.               <p:commandButton value="#{msg['form.submitText']}" update=":formulaire:contenu"/>
32.             </div>
33.           </p:column>
34.         </p:row>
35.       </f:facet>
36. </p:panelGrid>
```

Les lignes et les colonnes du tableau sont identifiées par les balises <**p:row**> et <**p:column**>.

Les lignes 3-12 définissent l'entête du tableau :

Java Server Faces - les tags		
Type	**Champs de saisie**	**Valeurs du modèle de la page**

Les lignes 14-25 définissent une ligne du tableau :

inputText	login : texte	texte

Les lignes 27-35 définissent le pied de page du tableau :

Valider

inputText

```
1.        <p:row>
2.          <p:column>
3.            <h:outputText value="inputText"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="inputText" value="#{msg['form.loginPrompt']}" />
7.             <p:inputText id="inputText" value="#{form.inputText}"/>
8.          </p:column>
9.          <p:column>
10.           <h:outputText id="inputTextValue" value="#{form.inputText}"/>
11.         </p:column>
12. </p:row>
```

inputText	login : texte	texte

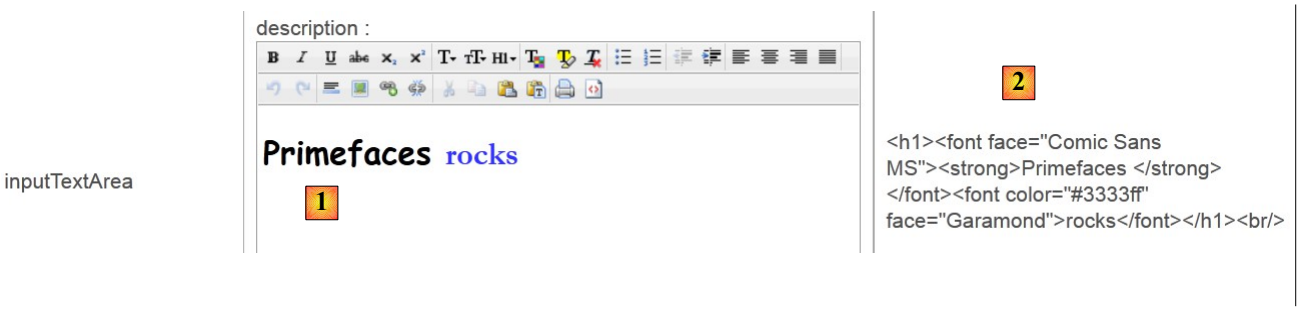
password

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="inputSecret"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="inputSecret" value="#{msg['form.passwdPrompt']}"/>
7.             <p:password id="inputSecret" value="#{form.inputSecret}" feedback="true"
8.                  promptLabel="#{msg['form.promptLabel']}" weakLabel="#{msg['form.weakLabel']}"
9.                  goodLabel="#{msg['form.goodLabel']}" strongLabel="#{msg['form.strongLabel']}"
    />
10.         </p:column>
11.         <p:column>
12.           <h:outputText id="inputSecretValue" value="#{form.inputSecret}"/>
13.         </p:column>
14.       </p:row>
```

Ligne 7, l'attribut **feedback=true** permet d'avoir un retour sur la qualité [1] du mot de passe.

inputTextArea

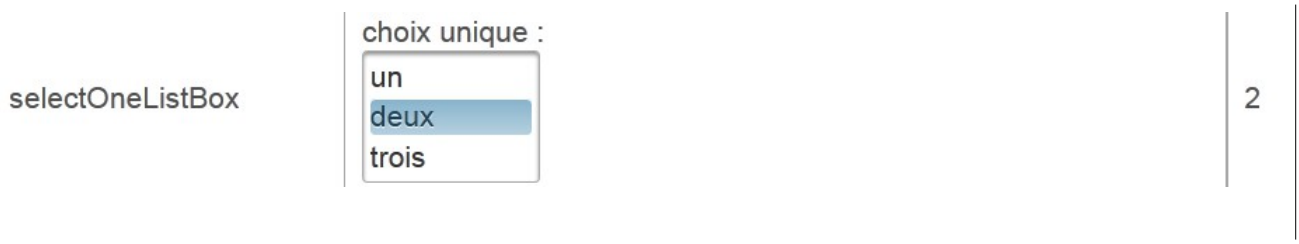
```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="inputTextArea"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="inputTextArea" value="#{msg['form.descPrompt']}"/>
7.             <p:editor id="inputTextArea" value="#{form.inputTextArea}" rows="4"/>
8.          </p:column>
9.          <p:column>
10.           <h:outputText id="inputTextAreaValue" value="#{form.inputTextArea}"/>
11.         </p:column>
12.       </p:row>
```



Ligne 7, la balise <**p:editor**> affiche un éditeur riche qui permet de mettre en forme le texte (police, taille, couleur, alignement, ...). Ce qui est posté au serveur est le code HTML du texte saisi [2].

selectOneListBox

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectOneListBox"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectOneListBox1" value="#{msg['form.selectOneListBox1Prompt']}"/>
7.            <p:selectOneListbox id="selectOneListBox1" value="#{form.selectOneListBox1}">
8.              <f:selectItem itemValue="1" itemLabel="un"/>
9.              <f:selectItem itemValue="2" itemLabel="deux"/>
10.             <f:selectItem itemValue="3" itemLabel="trois"/>
11.           </p:selectOneListbox>
12.         </p:column>
13.         <p:column>
14.           <h:outputText id="selectOneListBox1Value" value="#{form.selectOneListBox1}"/>
15.         </p:column>
16.       </p:row>
```

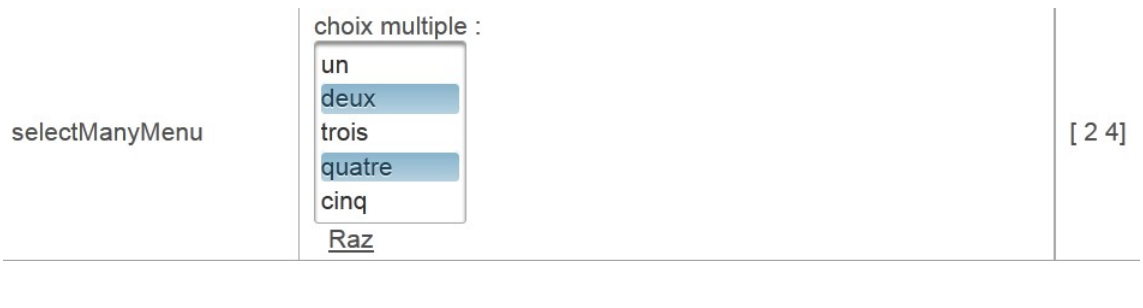


selectOneMenu

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectOneMenu"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectOneMenu" value="#{msg['form.selectOneMenuPrompt']}"/>
7.             <p:selectOneMenu id="selectOneMenu" value="#{form.selectOneMenu}">
8.               <f:selectItem itemValue="1" itemLabel="un"/>
9.               <f:selectItem itemValue="2" itemLabel="deux"/>
10.              <f:selectItem itemValue="3" itemLabel="trois"/>
11.              <f:selectItem itemValue="4" itemLabel="quatre"/>
12.              <f:selectItem itemValue="5" itemLabel="cinq"/>
13.            </p:selectOneMenu>
14.         </p:column>
15.         <p:column>
16.           <h:outputText id="selectOneMenuValue" value="#{form.selectOneMenu}"/>
17.         </p:column>
18.       </p:row>
```

selectManyMenu

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectManyMenu"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectManyMenu" value="#{msg['form.selectManyMenuPrompt']}"/>
7.             <p:selectManyMenu id="selectManyMenu" value="#{form.selectManyMenu}" >
8.               <f:selectItem itemValue="1" itemLabel="un"/>
9.               <f:selectItem itemValue="2" itemLabel="deux"/>
10.              <f:selectItem itemValue="3" itemLabel="trois"/>
11.              <f:selectItem itemValue="4" itemLabel="quatre"/>
12.              <f:selectItem itemValue="5" itemLabel="cinq"/>
13.            </p:selectManyMenu>
14.           <p:commandLink value="#{msg['form.buttonRazText']}"
    actionListener="#{form.clearSelectManyMenu()}" update=":formulaire:selectManyMenu" style="margin-
    left: 10px"/>
15.         </p:column>
16.         <p:column>
17.           <h:outputText id="selectManyMenuValue" value="#{form.selectManyMenuValue}"/>
18.         </p:column>
19.       </p:row>
```



Ligne 14, on notera que le lien [Raz] fait une mise à jour AJAX de la zone **:formulaire:selectManyMenu** qui correspond au composant de la ligne 6. Néanmoins, il faut savoir que lors du POST AJAX, toutes les valeurs du formulaire sont postées. Donc c'est la totalité du modèle qui est mis à jour. Avec ce modèle, on ne met cependant à jour que la zone **:formulaire:selectManyMenu.**

selectBooleanCheckbox

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectBooleanCheckbox"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectBooleanCheckbox"
    value="#{msg['form.selectBooleanCheckboxPrompt']}"/>
7.             <p:selectBooleanCheckbox id="selectBooleanCheckbox"
     value="#{form.selectBooleanCheckbox}"/>
8.          </p:column>
9.          <p:column>
10.           <h:outputText id="selectBooleanCheckboxValue" value="#{form.selectBooleanCheckbox}"/>
11.         </p:column>
12.       </p:row>
```

selectBooleanCheckbox	marié(e) : ☑	true

selectManyCheckbox

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectManyCheckbox"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectManyCheckbox" value="#{msg['form.selectManyCheckboxPrompt']}"/>
7.             <p:selectManyCheckbox id="selectManyCheckbox" value="#{form.selectManyCheckbox}">
8.               <f:selectItem itemValue="1" itemLabel="rouge"/>
9.               <f:selectItem itemValue="2" itemLabel="bleu"/>
10.              <f:selectItem itemValue="3" itemLabel="blanc"/>
11.              <f:selectItem itemValue="4" itemLabel="noir"/>
12.            </p:selectManyCheckbox>
13.         </p:column>
14.         <p:column>
15.           <h:outputText id="selectManyCheckboxValue" value="#{form.selectManyCheckboxValue}"/>
16.         </p:column>
17.       </p:row>
```

selectManyCheckbox	couleurs préférées : ☑ rouge ☐ bleu ☑ blanc ☐ noir	[1 3]

selectOneRadio

```
1.  <p:row>
2.          <p:column>
3.            <h:outputText value="selectOneRadio"/>
4.          </p:column>
5.          <p:column>
6.            <h:outputLabel for="selectOneRadio" value="#{msg['form.selectOneRadioPrompt']}"/>
7.             <p:selectOneRadio id="selectOneRadio" value="#{form.selectOneRadio}" >
8.               <f:selectItem itemValue="1" itemLabel="voiture"/>
9.               <f:selectItem itemValue="2" itemLabel="vélo"/>
10.              <f:selectItem itemValue="3" itemLabel="scooter"/>
11.              <f:selectItem itemValue="4" itemLabel="marche"/>
12.            </p:selectOneRadio>
13.         </p:column>
14.         <p:column>
15.           <h:outputText id="selectOneRadioValue" value="#{form.selectOneRadio}"/>
```

```
16.          </p:column>
17.        </p:row>
```

selectOneRadio	moyen de transport préféré : ○ voiture ◉ vélo ○ scooter ○ marche	2

5.8 Exemple : mv-pf-05 : listes dynamiques

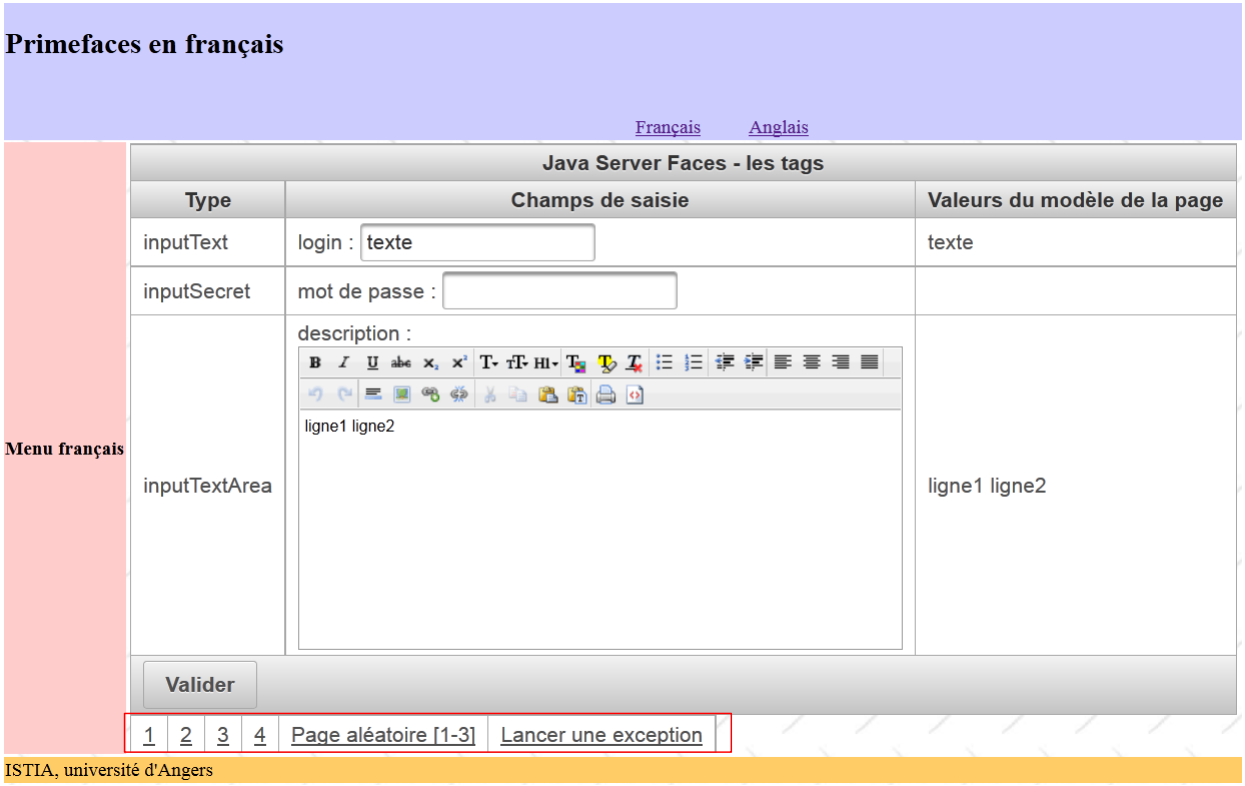
Ce projet est le portage du projet JSF2 [mv-jsf2-04] (cf paragraphe 2.6, page 92) :

selectOneListBox	choix unique : A0 A1 A2
selectOneMenu	choix unique : D1
selectManyMenu	choix multiple : E0 E1 E2 E3 Raz

Ce projet n'amène pas de nouvelles balises Primefaces vis à vis du projet précédent. Aussi ne le commenterons-nous pas. Il fait partie de la liste des exemples mis à disposition du lecteur sur le site du document.

5.9 Exemple : mv-pf-06 : navigation – session – gestion des exceptions

Ce projet est le portage du projet JSF2 [mv-jsf2-05] (cf paragraphe 2.7, page 97) :



De nouveau, cet exemple n'amène pas de balises Primefaces nouvelles. Nous ne commenterons que le tableau des liens encadré ci-dessus :

```
1.  <p:panelGrid columns="6">
2.    <p:commandLink value="1" action="form1?faces-redirect=true" ajax="false"/>
3.    <p:commandLink value="2" action="#{form.doAction2}" ajax="false"/>
4.    <p:commandLink value="3" action="form3?faces-redirect=true" ajax="false"/>
5.    <p:commandLink value="4" action="#{form.doAction4}" ajax="false"/>
6.    <p:commandLink value="#{msg['form.pagealeatoireLink']}" action="#{form.doAlea}" ajax="false"/>
7.    <p:commandLink value="#{msg['form.exceptionLink']}" action="#{form.throwException}" ajax="false"/>
8.  </p:panelGrid>
```

- les liens ont tous l'attribut **ajax=false**. Il y a donc un chargement de page normal,
- on notera lignes 2 et 4, la façon de faire une redirection.

5.10 Exemple : mv-pf-07 : validation et conversion des saisies

Ce projet est le portage du projet JSF2 [mv-jsf2-06] (cf paragraphe 2.8, page 108) :

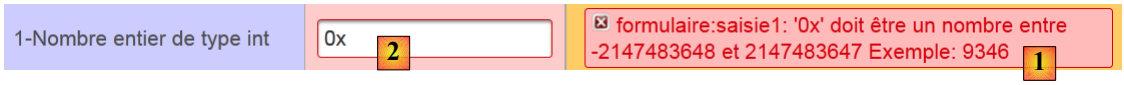
Type de la saisie	Champ de saisie	Erreur de saisie	Valeurs du modèle du formulaire
1-Nombre entier de type int	0x	formulaire:saisie1: '0x' doit être un nombre entre -2147483648 et 2147483647 Exemple: 9346	0
2-Nombre entier de type int	0x	**formulaire:saisie2: '0x' doit être un nombre constitué d'un ou plusieurs chiffres**	0
3-Nombre entier de type int	0x	Vous devez entrer un nombre entier	0
4-Nombre entier de type int dans l'intervalle [1,10]	0	4-Vous devez entrer un nombre entier dans l'intervalle [1,10]	0

L'application introduit deux nouvelles balises, la balise <**p:messages**> :

```
<p:messages globalOnly="true"/>
```

et la balise **<p:message>** :

```
1.  <p:inputText id="saisie1" value="#{form.saisie1}" styleClass="saisie"/>
2.  <p:message for="saisie1" styleClass="error"/>
```



Vis à vis de la balise <**h:message**> de JSF, la balise <**p:message**> de PF amène les changements suivants :

- l'apparence du message d'erreur est différente [1],
- la zone de saisie erronée est entourée d'un cadre rouge [2].

5.11 Exemple : mv-pf-08 : événements liés au changement d'état de composants

Ce projet est le portage du projet JSF2 [mv-jsf2-07] (cf paragraphe 2.9, page 131) :

Le projet JSF introduisait la notion de *listeners*. La gestion du *listener* avec Primefaces a été faite différemment.

Avec JSF :

```
1.          <!-- ligne 1 -->
2.          <h:outputText value="#{msg['combo1.prompt']}"/>
3.          <h:selectOneMenu id="combo1" value="#{form.combo1}" immediate="true"
    onchange="submit();" valueChangeListener="#{form.combo1ChangeListener}"
    styleClass="combo">
4.            <f:selectItems value="#{form.combo1Items}"/>
5.          </h:selectOneMenu>
6.          <h:panelGroup></h:panelGroup>
7.  <h:outputText value="#{form.combo1}"/>
```

Avec Primefaces :

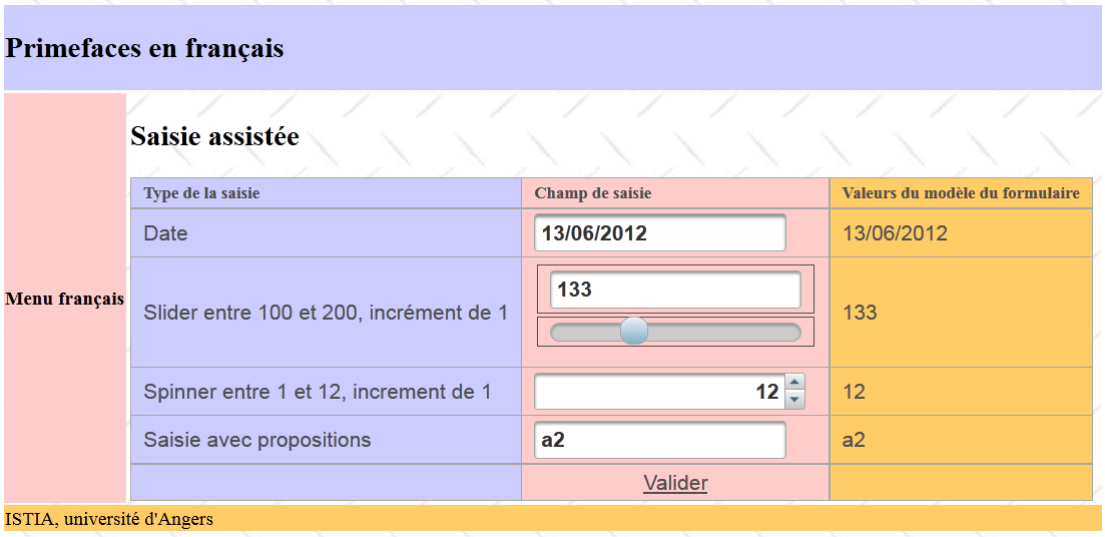
```
1.  <h:outputText value="#{msg['combo1.prompt']}"/>
2.          <p:selectOneMenu id="combo1" value="#{form.combo1}" styleClass="combo">
3.            <f:selectItems value="#{form.combo1Items}"/>
4.            <p:ajax update=":formulaire:combo2"/>
5.          </p:selectOneMenu>
6.          <h:panelGroup></h:panelGroup>
```

```
7.          <h:outputText value="#{form.combo1}"/>
8.
9.          <h:outputText value="#{msg['combo2.prompt']}"/>
10.         <p:selectOneMenu id="combo2" value="#{form.combo2}" styleClass="combo">
11.           <f:selectItems value="#{form.combo2Items}"/>
12.         </p:selectOneMenu>
```

- ligne 2 : la balise **<h:selectOneMenu>** sans attribut **valueChangeListener**,
- ligne 4 : la balise **<p:ajax>** ajoute un comportement AJAX à sa balise parent **<h:selectOneMenu>**. Par défaut, elle réagit à l'événement " changement de valeur " de la liste *combo1*. Sur cet événement, les valeurs du formulaire auquel elle appartient vont être postées au serveur par un appel AJAX. Le modèle est donc mis à jour. On utilise ce nouveau modèle pour mettre à jour la liste déroulante identifiée par *combo2* (ligne 10). On notera, ligne 4, que l'appel AJAX n'exécute pas de méthode du modèle. C'est inutile ici. On veut simplement changer le modèle par le POST des valeurs saisies.

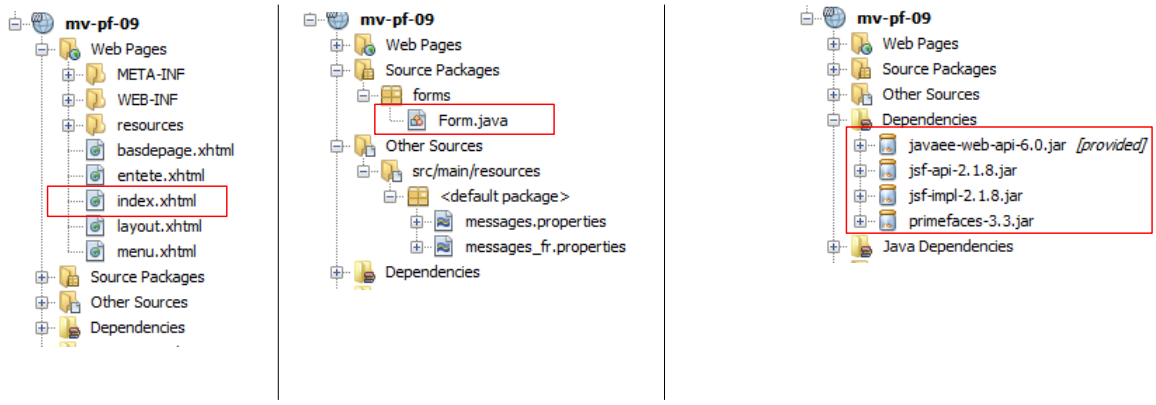
5.12 Exemple : mv-pf-09 : saisie assistée

Ce projet présente des balises de saisie spécifiques à Primefaces qui facilitent la saisie de certains types de données :



5.12.1 Le projet Netbeans

Le projet Netbeans est le suivant :



L'intérêt du projet réside dans :

- l'unique page [index.html] affichée par celui-ci,

- le modèle [Form.java] de cette dernière.

5.12.2 Le modèle

Le formulaire présente quatre champs de saisie associés au modèle suivant :

```
1.  package forms;
2.
3.  import java.io.Serializable;
4.  import java.util.ArrayList;
5.  import java.util.Date;
6.  import java.util.List;
7.  import javax.faces.bean.RequestScoped;
8.  import javax.faces.bean.ManagedBean;
9.
10. @ManagedBean
11. @SessionScoped
12. public class Form implements Serializable {
13.
14.   private Date calendrier;
15.   private Integer slider = 100;
16.   private Integer spinner = 1;
17.   private String autocompleteValue;
18.
19.   public Form() {
20.   }
21.
22.   public List<String> autocomplete(String query) {
23.     ...
24.   }
25.   // getters et setters
26. ...
27. }
```

Les quatre saisies sont associées aux champs des lignes 14-17.

5.12.3 Le formulaire

Le formulaire est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <html xmlns="http://www.w3.org/1999/xhtml"
3.        xmlns:h="http://java.sun.com/jsf/html"
4.        xmlns:p="http://primefaces.org/ui"
5.        xmlns:f="http://java.sun.com/jsf/core"
6.        xmlns:ui="http://java.sun.com/jsf/facelets">
7.
8.    <ui:composition template="layout.xhtml">
9.      <ui:define name="contenu">
10.       <h2><h:outputText value="#{msg['app.titre']}"/></h2>
11.       <p:growl id="messages" autoUpdate="true"/>
12.       <p:panelGrid columns="3" columnClasses="col1,col2,col3,col4">
13.         <h:outputText value="#{msg['saisie.type']}" styleClass="entete"/>
14.         <h:outputText value="#{msg['saisie.champ']}" styleClass="entete"/>
15.         <h:outputText value="#{msg['bean.valeur']}" styleClass="entete"/>
16.
17.         <!-- calendrier -->
18.               ...
19.
20.         <!-- slider -->
21.               ...
22.
23.         <!-- spinner -->
24.               ...
25.
26.         <!-- autocomplete -->
27.               ...
28.
29.       </p:panelGrid>
30.     </ui:define>
31.   </ui:composition>
```

```
32. </html>
```

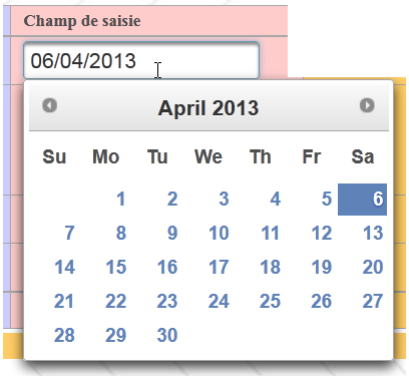
Examinons les quatre champs de saisie.

5.12.4 Le calendrier

La balise <**p:calendar**> permet de sélectionner une date à partir d'un calendrier. Cette balise admet différents attributs.

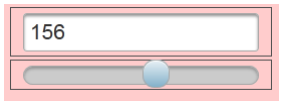
```
1.  <h:outputText value="#{msg['calendar.prompt']}"/>
2.          <p:calendar id="calendrier" value="#{form.calendrier}" pattern="dd/MM/yyyy"
    timeZone="Europe/Paris"/>
3.          <h:outputText id="calendrierValue" value="#{form.calendrier}">
4.            <f:convertDateTime pattern="dd/MM/yyyy" type="date" timeZone="Europe/Paris"/>
5.          </h:outputText>
```

Ligne 2, on indique que la date doit être affichée au format " jj/mm/aaaa " et que la zone horaire est celle de Paris. Lorsqu'on place le curseur dans la zone de saisie, un calendrier est affiché :



5.12.5 Le slider

La balise <**p:slider**> permet de saisir un entier en faisant glisser un curseur le long d'une barre :



Le code de la balise est le suivant :

```
1.          <h:outputText value="#{msg['slider.prompt']}"/>
2.          <h:panelGrid columns="1" style="margin-bottom:10px">
3.            <p:inputText id="slider" value="#{form.slider}" required="true"
    requiredMessage="#{msg['slider.required']}" validatorMessage="#{msg['slider.invalide']}">
4.              <f:validateLongRange minimum="100" maximum="200"/>
5.            </p:inputText>
6.            <p:slider for="slider" minValue="100" maxValue="200"/>
7.          </h:panelGrid>
8.  <h:outputText id="sliderValue" value="#{form.slider}"/>
```

- ligne 3 : on a là une balise **<p:inputText>** classique qui permet de saisir le nombre entier. Celui-ci peut être également saisi grâce au slider,

- ligne 4 : la balise **<p:slider>** est associée à la balise de saisie <**p:inputText**> (attribut **for**). On lui fixe une valeur minimale et une valeur maximale.

5.12.6 Le spinner

On a eu déjà l'occasion de présenter ce composant :

```
1.          <h:outputText value="#{msg['spinner.prompt']}"/>
2.          <p:spinner id="spinner" min="1" max="12" value="#{form.spinner}" required="true"
    requiredMessage="#{msg['spinner.required']}" validatorMessage="#{msg['spinner.invalide']}">
3.            <f:validateLongRange minimum="1" maximum="12"/>
4.          </p:spinner>
5.  <h:outputText id="spinnerValue" value="#{form.spinner}"/>
```

Ligne 3, le spinner permet une saisie d'un nombre entier entre 1 et 12. On peut saisir le nombre directement dans la zone de saisie du spinner ou bien utiliser les flèches pour augmenter / diminuer le nombre saisi.

5.12.7 La saisie assistée

La saisie assistée consiste à taper les premiers caractères de la saisie. Des propositions apparaissent alors dans une liste déroulante. On peut sélectionner l'une d'elles. On utilise ce composant à la place des listes déroulantes lorsque celles-ci ont un contenu trop important. Supposons que l'on veuille proposer une liste déroulante des villes de France. Cela fait plusieurs milliers de villes. Si on laisse l'utilisateur taper les trois premiers caractères de la ville, on peut alors lui proposer une liste réduite des villes commençant par ceux-ci.

Le code de ce composant est le suivant :

```
1.          <h:outputText value="#{msg['autocomplete.prompt']}"/>
2.          <p:autoComplete value="#{form.autocompleteValue}"
    completeMethod="#{form.autocomplete}" required="true"
    requiredMessage="#{msg['autocomplete.required']}"/>
3.          <h:outputText id="autocompleteValue" value="#{form.autocompleteValue}"/>
4.          <h:panelGroup/>
5.          <h:panelGroup>
6.          <center><p:commandLink value="#{msg['valider']}" update="formulaire:contenu"/></center>
7.          </h:panelGroup>
8.  <h:panelGroup/>
```

La balise <**p:autoComplete**> de la ligne 2 est celle qui permet la saisie assistée. Le paramètre qui nous intéresse ici est l'attribut **completeMethod** dont la valeur est le nom d'une méthode du modèle, responsable de faire des propositions correspondant aux caractères tapés par l'utilisateur. Cette méthode est ici la suivante :

```
1.    public List<String> autocomplete(String query) {
2.      List<String> results = new ArrayList<String>();
3.
4.      for (int i = 0; i < 10; i++) {
5.        results.add(query + i);
```

```
6.      }
7.
8.      return results;
9.  }
```

- ligne 1 : la méthode reçoit comme paramètre la chaîne des caractères tapés par l'utilisateur dans la zone de saisie. Elle rend une liste de propositions,
- lignes 4-6 : on construit une liste de 10 propositions qui reprend les caractères reçus en paramètres et leur ajoute un chiffre de 0 à 9.

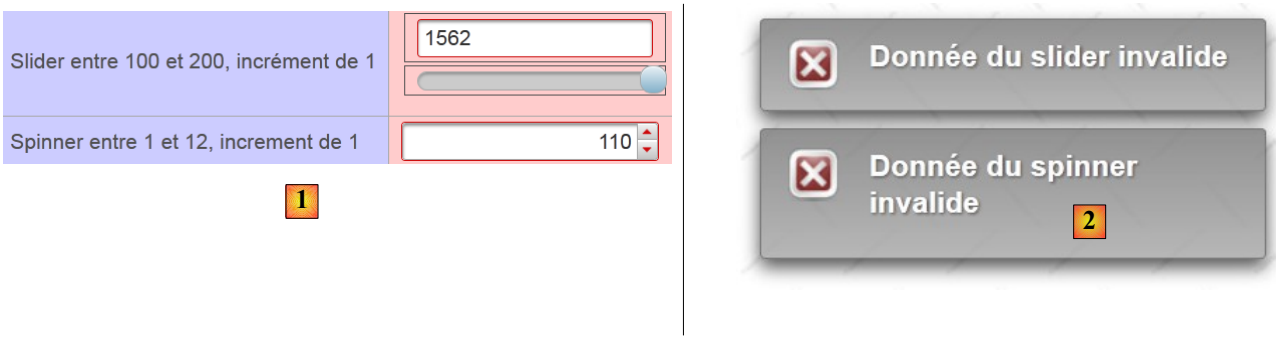
5.12.8 La balise <p:growl>

La balise <**p:growl**> est un remplacement possible de la balise <**p:messages**> qui affiche les messages d'erreur du formulaire.

```
<p:growl id="messages" autoUpdate="true"/>
```

Ci-dessus, l'attribut **id** n'est pas utilisé. L'attribut **autoUpdate=true** indique que la liste des messages d'erreurs doit être réactualisée à chaque POST du formulaire.

Supposons que l'on valide le formulaire suivant [1] :



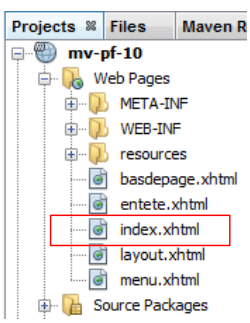
- en [2], la balise <**p:growl**> affiche alors les messages d'erreurs associés aux saisies erronées.

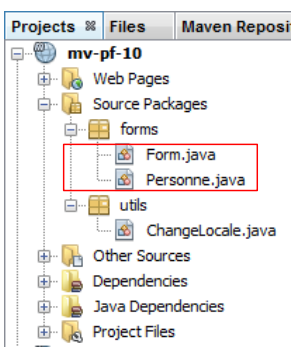
5.13 Exemple : mv-pf-10 : dataTable - 1

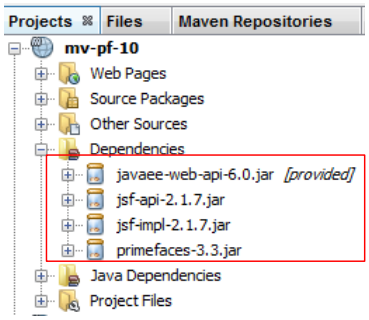
Ce projet présente la balise <**p:dataTable**> qui sert à afficher des listes de données

5.13.1 Le projet Netbeans

Le projet Netbeans est le suivant :







L'intérêt du projet réside dans :

- l'unique page [index.html] affichée par celui-ci,
- le modèle [Form.java] de cette dernière et le bean [Personne].

5.13.2 Le fichier des messages

Le fichier [messages_fr.properties] est le suivant :

```
app.titre=intro-08
app.titre2=DataTable - 1
submit=Valider
personnes.headers.id=Id
personnes.headers.nom=Nom
personnes.headers.prenom=Pr\u00e9nom
layout.hautdepage=Primefaces en fran\u00e7ais
layout.menu=Menu fran\u00e7ais
layout.basdepage=ISTIA, universit\u00e9 d'Angers
form.langue1=Fran\u00e7ais
form.langue2=Anglais
form.noData=La liste des personnes est vide
form.listePersonnes=Liste de personnes
form.action=Action
```

5.13.3 Le modèle

Le bean [Personne] représente une personne :

```
package forms;

import java.io.Serializable;

public class Personne implements Serializable{
  // data
  private int id;
  private String nom;
  private String prénom;

  // constructeurs
  public Personne(){

  }

  public Personne(int id, String nom, String prénom){
    this.id=id;
```

```
18.     this.nom=nom;
19.     this.prénom=prénom;
20.   }
21.
22.   // toString
23.   public String toString(){
24.     return String.format("Personne[%d,%s,%s]", id,nom,prénom);
25.   }
26.
27.   // getter et setters
28. ...
29. }
```

Le modèle de la page [index.xhtml] est la classe [Form] suivante :

```
1.  package forms;
2.
3.  import java.io.Serializable;
4.  import java.util.ArrayList;
5.  import java.util.List;
6.  import javax.faces.bean.ManagedBean;
7.  import javax.faces.bean.SessionScoped;
8.
9.  @ManagedBean
10. @SessionScoped
11. public class Form implements Serializable{
12.
13.   // modèle
14.   private List<Personne> personnes;
15.   private int personneId;
16.
17.   // constructeur
18.   public Form() {
19.     // initialisation de la liste des personnes
20.     personnes = new ArrayList<Personne>();
21.     personnes.add(new Personne(1, "dupont", "jacques"));
22.     personnes.add(new Personne(2, "durand", "élise"));
23.     personnes.add(new Personne(3, "martin", "jacqueline"));
24.   }
25.
26.   public void retirerPersonne() {
27. ...
28.   }
29.
30.   // getters et setters
31. ...
32. }
```

- lignes 9-10 : le bean est de portée **session**,
- lignes 18-24 : le constructeur crée une liste de trois personnes, liste qui va donc vivre au fil des requêtes,
- ligne 15 : le n° d'une personne à supprimer de la liste,
- lignes 26-28 : la méthode de suppression.

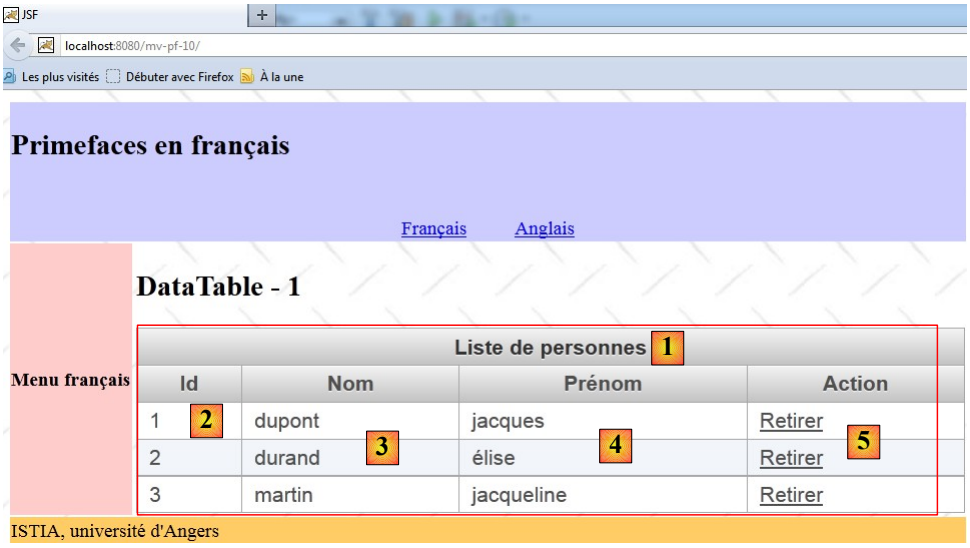
5.13.4 Le formulaire

Le formulaire est le suivant [index.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:p="http://primefaces.org/ui"
7.        xmlns:f="http://java.sun.com/jsf/core"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <ui:composition template="layout.xhtml">
10.     <ui:define name="contenu">
11.       <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
12.       <p:dataTable value="#{form.personnes}" var="personne" emptyMessage="#{msg['form.noData']}">
```

```
13.         <f:facet name="header">
14.           #{msg['form.listePersonnes']}
15.         </f:facet>
16.         <p:column>
17.           <f:facet name="header">
18.             #{msg['personnes.headers.id']}
19.           </f:facet>
20.           #{personne.id}
21.         </p:column>
22.         <p:column>
23.           <f:facet name="header">
24.             #{msg['personnes.headers.nom']}
25.           </f:facet>
26.           #{personne.nom}
27.         </p:column>
28.         <p:column>
29.           <f:facet name="header">
30.             #{msg['personnes.headers.prenom']}
31.           </f:facet>
32.           #{personne.prénom}
33.         </p:column>
34.         <p:column>
35.           <f:facet name="header">
36.             #{msg['form.action']}
37.           </f:facet>
38.           <p:commandLink value="Retirer" action="#{form.retirerPersonne}" update=":formulaire:contenu">
39.             <f:setPropertyActionListener target="#{form.personneId}" value="#{personne.id}"/>
40.           </p:commandLink>
41.         </p:column>
42.       </p:dataTable>
43.     </ui:define>
44.   </ui:composition>
45. </html>
```

Cela produit la vue suivante (encadré ci-dessous) :



- ligne 12 : génère le tableau encadré ci-dessus. L'attribut **value** désigne la collection affichée par le tableau, ici la liste des personnes du modèle. L'attribut **emptyMessage** et facultatif. Il désigne le message à afficher lorsque la liste est vide. Par défaut c'est '*no records found*'. Ici, ce sera :

- lignes 13-15 : génèrent l'entête [1],
- lignes 16-21 : génèrent la colonne [2],
- lignes 22-27 : génèrent la colonne [3],
- lignes 28-33 : génèrent la colonne [4],
- lignes 34-41 : génèrent la colonne [5].

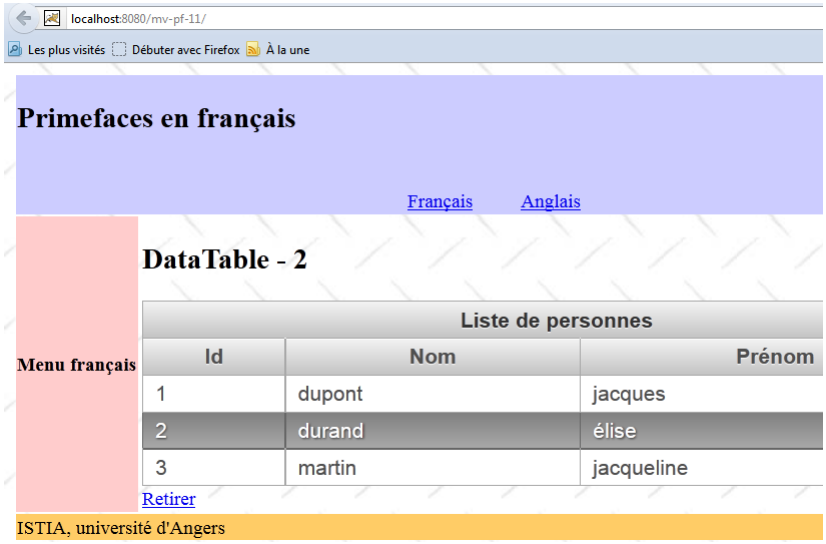
Le lien [Retirer] permet de retirer une personne de la liste. Ligne [38], c'est la méthode *[Form].retirerPersonne* qui fait ce travail. Elle a besoin de connaître le n° de la personne à retirer. Celui-ci lui est donné ligne 39. Ligne 38, on a utilisé l'attribut **action**. A d'autres occasions, on a utilisé l'attribut **actionListener**. Je ne suis pas sûr de bien comprendre la différence fonctionnelle entre ces deux attributs. A l'usage, on remarque cependant que les attributs positionnés par les balises <setPropertyActionListener> le sont **avant** exécution de la méthode désignée par l'attribut **action**, et que ce n'est pas le cas pour l'attribut **actionListener**. En clair, dès qu'on a des paramètres à envoyer à l'action appelée, il faut utiliser l'attribut **action**.

La méthode pour retirer une personne est la suivante :

```
1.  ...
2.  @ManagedBean
3.  @SessionScoped
4.  public class Form implements Serializable{
5.
6.    // modèle
7.    private List<Personne> personnes;
8.    private int personneId;
9.
10.   public void retirerPersonne() {
11.     // on recherche la personne sélectionnée
12.     int i = 0;
13.     for (Personne personne : personnes) {
14.       // personne courante = personne sélectionnée ?
15.       if (personne.getId() == personneId) {
16.         // on supprime la personne courante de la liste
17.         personnes.remove(i);
18.         // on a fini
19.         break;
20.       } else {
21.         // personne suivante
22.         i++;
23.       }
24.     }
25.   }
26. ...
27. }
```

5.14 Exemple : mv-pf-11 : dataTable - 2

Ce projet présente un tableau affichant une liste de données où une ligne peut être sélectionnée :



Le fait de sélectionner une ligne du tableau envoie, au moment du POST, des informations au modèle sur la ligne sélectionnée. Du coup, on n'a plus besoin d'un lien [Retirer] par personne. Un seul pour l'ensemble du tableau est suffisant.

Le projet Netbeans est identique au précédent à quelques détails près : le formulaire et son modèle. Le formulaire [index.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:p="http://primefaces.org/ui"
7.        xmlns:f="http://java.sun.com/jsf/core"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <ui:composition template="layout.xhtml">
10.     <ui:define name="contenu">
11.       <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
12.       <p:dataTable value="#{form.personnes}" var="personne" emptyMessage="#{msg['form.noData']}"
13.                    rowKey="#{personne.id}"  selection="#{form.personneChoisie}" selectionMode="single">
14.         ...
15.       </p:dataTable>
16.       <p:commandLink value="Retirer" action="#{form.retirerPersonne}" update=":formulaire:contenu"/>
17.     </ui:define>
18.   </ui:composition>
19. </html>
```

- ligne 13 : l'attribut **selectionMode** permet de choisir un mode de sélection *single* ou *multiple*. Ici, nous avons choisi de ne sélectionner qu'une seule ligne,
- ligne 13 : l'attribut **rowkey** désigne un attribut des éléments affichés qui permet de les sélectionner de façon unique. Ici, nous avons choisi l'**id** de la personne sélectionnée,
- ligne 13 : l'attribut **selection** désigne l'attribut du modèle qui recevra une référence de la personne sélectionnée. Grâce à l'attribut **rowkey** précédent, côté serveur une référence de la personne sélectionnée va pouvoir être calculée. On n'a pas les détails de la méthode utilisée. On peut imaginer que la collection est parcourue séquentiellement à la recherche de l'élément correspondant au **rowkey** sélectionné. Cela veut dire que si la méthode qui associe **rowkey** à **selection** est plus complexe, alors cette méthode n'est pas utilisable,

Ceci expliqué, la méthode *[Form].retirerPersonne* évolue comme suit :

```
1.  ...
2.
3.  @ManagedBean
4.  @SessionScoped
5.  public class Form implements Serializable {
6.
7.    // modèle
8.    private List<Personne> personnes;
9.    private Personne personneChoisie;
```

```
10.
11.   // constructeur
12.   public Form() {
13.   ...
14.   }
15.
16.   public void retirerPersonne() {
17.     // on enlève la personne choisie
18.     personnes.remove(personneChoisie);
19.   }
20.
21.   // getters et setters
22. ...
23. }
```

- ligne 9 : à chaque POST, la référence de la ligne 9 est initialisée avec la référence, dans la liste de la ligne 8, de la personne sélectionnée,
- en 18 : la suppression de la personne s'en trouve simplifiée. La recherche que nous avions faite dans l'exemple précédent a été faite par la balise <*dataTable*>.

5.15 Exemple : mv-pf-12 : dataTable - 3

Ce projet est analogue au précédent. La vue est notamment identique :

Le projet Netbeans est identique au précédent à quelques détails près que nous allons passer en revue. Le formulaire [index.xhtml] évolue comme suit :

```
1.  ...
2.    <ui:composition template="Layout.xhtml">
3.      <ui:define name="contenu">
4.        <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
5.        <p:dataTable value="#{form.personnes}" var="personne" emptyMessage="#{msg['form.noData']}"
6.                     selectionMode="single" selection="#{form.personneChoisie}">
7.    ...
8.        </p:dataTable>
9.        <p:commandLink value="Retirer" action="#{form.retirerPersonne}" update=":formulaire:contenu"/>
10.     </ui:define>
11.   </ui:composition>
12. </html>
```

- ligne 6, l'attribut **rowkey** a disparu, l'attribut **selection** reste. Le lien entre les attributs **rowkey** et **selection** se fait désormais au travers d'une classe. L'attribut **value** de la ligne 5 a désormais pour valeur une instance de l'interface Primefaces **SelectableDataModel<T>**. La méthode *[Form].getPersonnes* du modèle évolue comme suit :

```
1.    public DataTableModel getPersonnes() {
2.      return new DataTableModel(personnes);
3.  }
```

Un nouveau bean est ainsi ajouté au projet :

Ce bean est le suivant :

```
1.  package forms;
2.
3.  import java.util.List;
4.  import javax.faces.model.ListDataModel;
5.  import org.primefaces.model.SelectableDataModel;
6.
7.  public class DataTableModel extends ListDataModel<Personne> implements SelectableDataModel<Personne> {
8.
9.    // constructeurs
10.   public DataTableModel() {
11.   }
12.
13.   public DataTableModel(List<Personne> personnes) {
14.     super(personnes);
15.   }
16.
17.   @Override
18.   public Object getRowKey(Personne personne) {
19.     return personne.getId();
20.   }
21.
22.   @Override
23.   public Personne getRowData(String rowKey) {
24.     // liste des personnes
25.     List<Personne> personnes = (List<Personne>) getWrappedData();
26.     // la clé est un entier
27.     int key = Integer.parseInt(rowKey);
28.     // on recherche la personne sélectionnée
29.     for (Personne personne : personnes) {
30.       if (personne.getId() == key) {
31.         return personne;
32.       }
33.     }
34.     // on n'a rien trouvé
35.     return null;
36.   }
37. }
```

- ligne 7 : la classe est une instance de l'interface *SelectableDataModel*. Deux classes au moins implémentent cette interface : **ListDataModel** dont le constructeur admet une liste pour paramètre, et **ArrayDataModel** dont le constructeur admet un tableau pour paramètre. Ici, notre bean étend la classe **ListDataModel**,
- lignes 13-15 : le constructeur admet pour paramètre la liste des personnes que nous gérons. Ce paramètre est passé à la classe parent,
- ligne 18 : la méthode **getRowKey** joue le rôle de l'attribut **rowkey** qui a été enlevé. Elle doit rendre l'objet qui permet d'identifier une personne de façon unique, ici l'**id** de la personne,
- ligne 23 : la méthode **getRowData** doit rendre l'objet sélectionné, à partir de son **rowkey**. Donc ici, rendre une personne à partir de son **id**. La référence ainsi obtenue sera affectée à l'objet cible de l'attribut **selection** dans la balise **dataTable**, ici l'attribut `selection="#{form.personneChoisie}"`. Le paramètre de la méthode est le **rowkey** de l'objet sélectionné par l'utilisateur, sous forme d'une chaîne de caractères,

- lignes 24-35 : délivrent la référence sur la personne dont on a reçu l'**id**. Cette référence sera affectée au modèle *[Form].personneChoisie*. La méthode [retirerPersonne] reste donc inchangée :

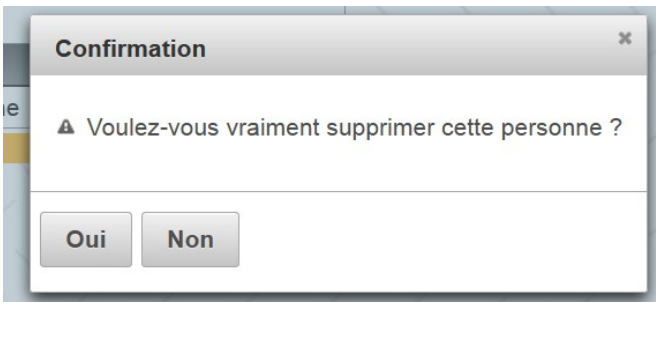
```
1.    public void retirerPersonne() {
2.      // on enlève la personne choisie
3.      personnes.remove(personneChoisie);
4.    }
```

C'est la technique à utiliser lorsque le lien entre les attributs **rowkey** et **selection** n'est pas un simple lien de propriété (**rowkey**) à objet (**selection**).

5.16 Exemple : mv-pf-13 : dataTable - 4

Ce projet est analogue au précédent si ce n'est que le mode de sélection de la personne à retirer change :

Ci-dessus, nous voyons que l'objet est sélectionné grâce à un menu contextuel (clic droit). Une confirmation de la suppression est demandée :

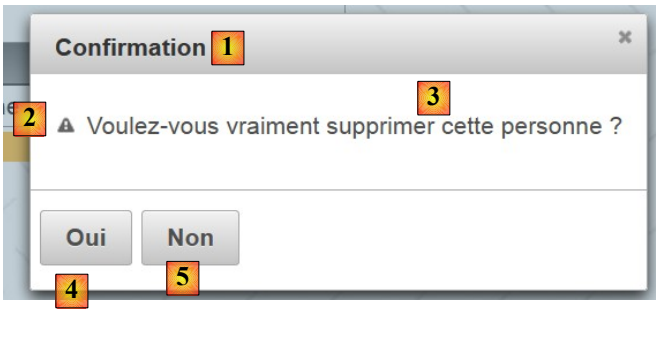


Le formulaire [index.xhtml] évolue comme suit :

```
1.  ...
2.    <ui:composition template="layout.xhtml">
3.      <ui:define name="contenu">
4.
5.        <!-- titre -->
6.        <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
7.
8.        <!-- menu contextuel -->
```

```
9.        <p:contextMenu for="personnes">
10.         <p:menuitem value="#{msg['form.supprimer']}" onclick="confirmation.show()"/>
11.       </p:contextMenu>
12.
13.       <!-- boîte de dialogue -->
14.       <p:confirmDialog widgetVar="confirmation" message="#{msg['form.suppression.confirmation']}"
15.                        header="#{msg['form.suppression.message']}" severity="alert" >
16.         <p:commandButton value="#{msg['form.supprimer.oui']}" update=":formulaire:contenu"
    action="#{form.retirerPersonne}" oncomplete="confirmation.hide()"/>
17.         <p:commandButton value="#{msg['form.supprimer.non']}" onclick="confirmation.hide()" type="button"
    />
18.       </p:confirmDialog>
19.
20.       <!-- dataTable-->
21.       <p:dataTable id="personnes" value="#{form.personnes}" var="personne"
    emptyMessage="#{msg['form.noData']}"
22.                    selection="#{form.personneChoisie}" selectionMode="single">
23.         ...
24.       </p:dataTable>
25.     </ui:define>
26.   </ui:composition>
27. </html>
```

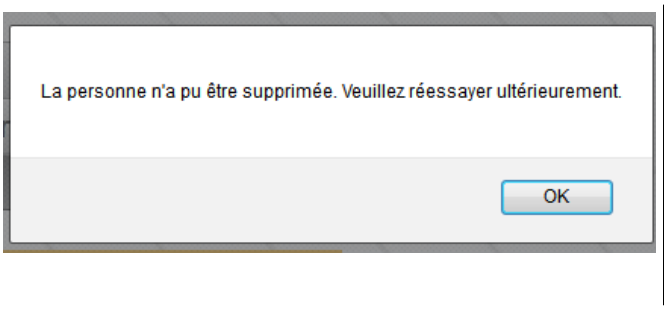
- lignes 9-11 : définissent un menu contextuel pour (attribut **for**) le *dataTable* de la ligne 21 (attribut **id**). C'est donc sur un clic droit sur le tableau des personnes que ce menu contextuel apparaît,
- ligne 10 : notre menu n'a qu'une option (balise **menuItem**). Lorsque cette option est cliquée, le code Javascript de l'attribut **onclick** est exécuté. Le code Javascript [`confirmation.show()`] fait afficher la boîte de dialogue de la ligne 14 (attribut **widgetVar**). Celle-ci est la suivante :



- ligne 14 : l'attribut **message** fait afficher [3], l'attribut **header** fait afficher [1], l'attribut **severity** fait afficher l'icône [2],
- ligne 16 : fait afficher [4]. Sur un clic, la personne est supprimée (attribut **action**), puis la boîte de dialogue est fermée (attribut **oncomplete**). L'attribut **oncomplete** est du code Javascript qui est exécuté une fois que l'action côté serveur a été exécutée,
- ligne 17 : fait afficher [5]. Sur un clic, la boîte de dialogue est fermée et la personne n'est pas supprimée.

5.17 Exemple : mv-pf-14 : dataTable - 5

Ce projet montre qu'il est possible d'avoir un retour du serveur après exécution d'un appel AJAX. On utilise pour cela l'attribut **oncomplete** de l'appel AJAX :



Le formulaire [index.xhtml] évolue comme suit :

```
1.  ...
2.    <ui:composition template="layout.xhtml">
3.      <ui:define name="contenu">
4.  ...
5.  <!-- boîte de dialogue 1 -->
6.        <p:confirmDialog widgetVar="confirmation" ... >
7.          <p:commandButton value="#{msg['form.supprimer.oui']}" update=":formulaire:contenu"
    action="#{form.retirerPersonne}" oncomplete="handleRequest(xhr, status, args);confirmation.hide()"/>
8.          <p:commandButton ... />
9.        </p:confirmDialog>
10.
11.       <!-- Javascript -->
12.       <script type="text/javascript">
13.         function handleRequest(xhr, status, args) {
14.           // erreur ?
15.           if(args.msgErreur) {
16.             alert(args.msgErreur);
17.           }
18.         }
19.       </script>
20. ...
21.       </p:dataTable>
22.     </ui:define>
23.   </ui:composition>
24. </html>
```

- ligne 7 : l'attribut **oncomplete** appelle la fonction Javascript des lignes 13-18,
- ligne 13 : la signature de la méthode doit être celle-ci. **args** est un dictionnaire que le modèle côté serveur peut enrichir,
- ligne 15 : on regarde si le dictionnaire **args** a un attribut nommé '**msgErreur**'. Si oui, il est affiché (ligne 16).

Dans le modèle, la méthode [retirerPersonne] évolue comme suit :

```
1.  public void retirerPersonne() {
2.      // suppression aléatoire
3.      int i = (int) (Math.random() * 2);
4.      if (i == 0) {
5.        // on enlève la personne choisie
6.        personnes.remove(personneChoisie);
7.      } else {
8.        // on renvoie une erreur
9.        String msgErreur = Messages.getMessage(null, "form.msgErreur", null).getSummary();
10.       RequestContext.getCurrentInstance().addCallbackParam("msgErreur", msgErreur);
11.     }
12.   }
```

- ligne 3 : on tire un nombre aléatoire 0 ou 1,
- lignes 4-6 : si c'est 0, la personne sélectionnée par l'utilisateur est supprimée de la liste des personnes,
- lignes 9 : sinon, on construit un message d'erreur internationalisé :

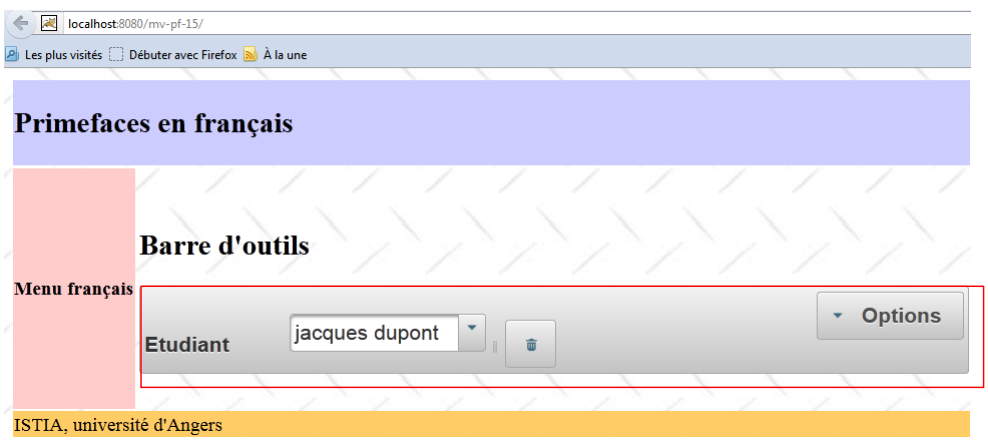
```
1.  form.msgErreur=La personne n'a pu \u00eatre supprim\u00e9e. Veuillez r\u00e9essayer ult\u00e9rieurement.
2.  form.msgErreur_detail=La personne n'a pu \u00eatre supprim\u00e9e. Veuillez r\u00e9essayer
    ult\u00e9rieurement.
```

- ligne 10 : une instruction compliquée qui a pour but d'ajouter dans le dictionnaire **args** dont nous avons parlé, l'attribut nommé **'msgErreur'** avec la valeur **msgErreur** construite ligne 9. Cet attribut est récupéré ensuite par la méthode Javascript de [index.xhtml] :

```
1.        <!-- Javascript -->
2.        <script type="text/javascript">
3.          function handleRequest(xhr, status, args) {
4.            // erreur ?
5.            if(args.msgErreur) {
6.              alert(args.msgErreur);
7.            }
8.          }
9.  </script>
```

5.18 Exemple : mv-pf-15 : la barre d'outils

Dans ce projet nous construisons une barre d'outils :



La barre d'outils est le composant encadré ci-dessus. Elle est obtenue avec le code XHTML suivant [index.xhtml] :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:p="http://primefaces.org/ui"
7.        xmlns:f="http://java.sun.com/jsf/core"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <ui:composition template="layout.xhtml">
10.     <ui:define name="contenu">
11.       <!-- titre -->
12.       <h2><h:outputText value="#{msg['app.titre2']}"/></h2>
13.
14.       <!-- barre d'outils-->
15.       <p:toolbar>
16.         <p:toolbarGroup align="left">
17.           ...
18.         </p:toolbarGroup>
19.         <p:toolbarGroup align="right">
20.           ...
21.         </p:toolbarGroup>
22.       </p:toolbar>
23.     </ui:define>
24.   </ui:composition>
25. </html>
```

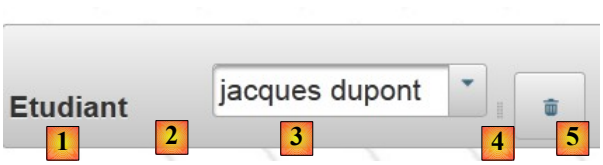
- lignes 15-22 : la barre d'outils,

- lignes 16-18 : définissent le groupe de composants à gauche de la barre,
- lignes 19-21 : idem pour les composants à droite.

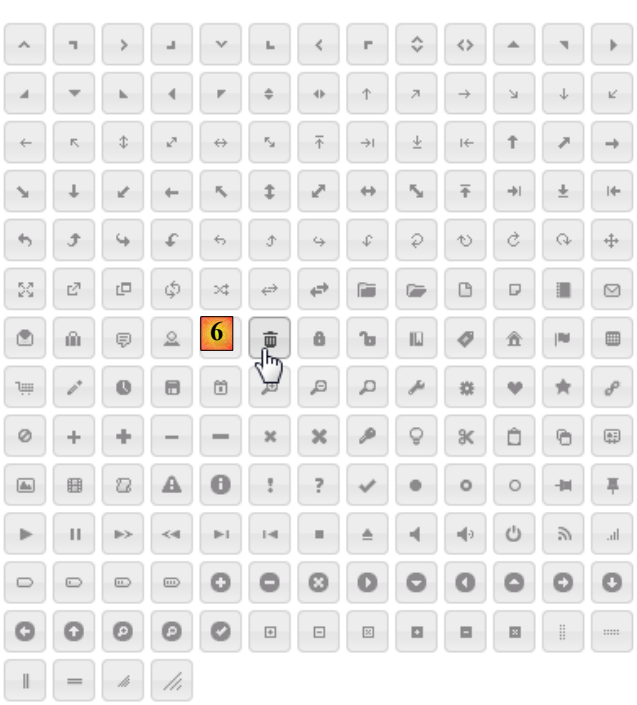
Les composants à gauche de la barre d'outils sont les suivants :

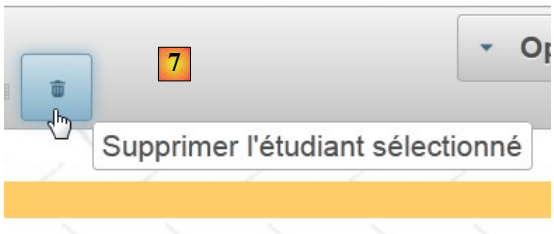
```
1.          <p:toolbarGroup align="left">
2.            <h:outputText value="#{msg['form.etudiant']}"/>
3.            <p:spacer width="50px"/>
4.            <p:selectOneMenu value="#{form.personneId}" effect="fade">
5.              <f:selectItems value="#{form.personnes}" var="personne" itemLabel="#{personne.prénom}
    #{personne.nom}" itemValue="#{personne.id}"/>
6.            </p:selectOneMenu>
7.            <p:separator/>
8.            <p:commandButton id="delete-personne" icon="ui-icon-trash"
    action="#{form.supprimerPersonne}" update=":formulaire:contenu"/>
9.            <p:tooltip for="delete-personne" value="#{msg['form.delete.personne']}"/>
10. </p:toolbarGroup>
```

Ils affichent la vue ci-dessous :



- ligne 2 : affiche [1],
- ligne 3 : affiche un espace de 30 pixels [2],
- lignes 4-6 : affichent une liste déroulante avec une liste de personnes [3],
- ligne 7 : affiche un séparateur [4],
- ligne 8 : affiche un bouton [5] chargé de supprimer la personne sélectionnée dans la liste déroulante. Le bouton a une icône. Ces icônes sont celles de JQuery UI. On trouve leur liste à l'URL [http://jqueryui.com/themeroller/] [6] :





- pour connaître le nom d'une icône, il suffit de placer la souris dessus. Ensuite ce nom est utilisé dans l'attribut **icon** du composant <*commandButton*>, par exemple `icon="ui-icon-trash"`. On notera que ci-dessus, le nom donné sera **.ui-icon-trash** et qu'on enlève le point initial de ce nom dans l'attribut *icon,*
- ligne 9 : crée une bulle d'aide pour le bouton (attribut **for**). Lorsqu'on laisse le curseur sur le bouton, le message d'aide s'affiche [7].

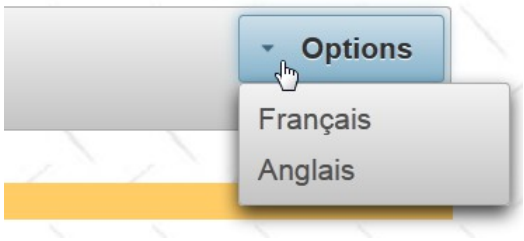
Le modèle associé à ces composants est le suivant :

```
1.  package forms;
2.
3.  import java.io.Serializable;
4.  import java.util.ArrayList;
5.  import java.util.List;
6.  import javax.faces.bean.ManagedBean;
7.  import javax.faces.bean.SessionScoped;
8.
9.  @ManagedBean
10. @SessionScoped
11. public class Form implements Serializable {
12.
13.   // modèle
14.   private List<Personne> personnes;
15.   private int personneId;
16.
17.   // constructeur
18.   public Form() {
19.     // initialisation de la liste des personnes
20.     personnes = new ArrayList<Personne>();
21.     personnes.add(new Personne(1, "dupont", "jacques"));
22.     personnes.add(new Personne(2, "durand", "élise"));
23.     personnes.add(new Personne(3, "martin", "jacqueline"));
24.   }
25.
26.   public void supprimerPersonne() {
27.     // on recherche la personne sélectionnée
28.     int i = 0;
29.     for (Personne personne : personnes) {
30.       // personne courante = personne sélectionnée ?
31.       if (personne.getId() == personneId) {
32.         // on supprime la personne courante de la liste
33.         personnes.remove(i);
34.         // on a fini
35.         break;
36.       } else {
37.         // personne suivante
38.         i++;
39.       }
40.     }
41.   }
42.
43.   // getters et setters
44.   ...
45. }
```

Les composants à droite de la barre d'outils sont les suivants :

```
1.  <p:toolbar>
2.          <p:toolbarGroup align="left">
3.            ...
4.          </p:toolbarGroup>
5.          <p:toolbarGroup align="right">
6.            <p:menuButton value="#{msg['form.options']}">
7.              <p:menuitem id="menuitem-francais" value="#{msg['form.francais']}"
    actionListener="#{changeLocale.setFrenchLocale}" update=":formulaire"/>
8.              <p:menuitem id="menuitem-anglais" value="#{msg['form.anglais']}"
    actionListener="#{changeLocale.setEnglishLocale}" update=":formulaire"/>
9.            </p:menuButton>
10.         </p:toolbarGroup>
11.       </p:toolbar>
```

Ils affichent la vue ci-dessous :



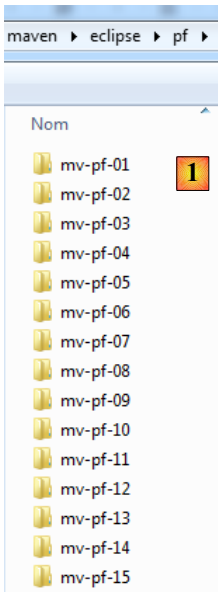
- lignes 6-9 : un bouton menu. Il contient des options de menu,
- ligne 7 : l'option pour passer le formulaire en français,
- ligne 8 : celle pour le passer en anglais.

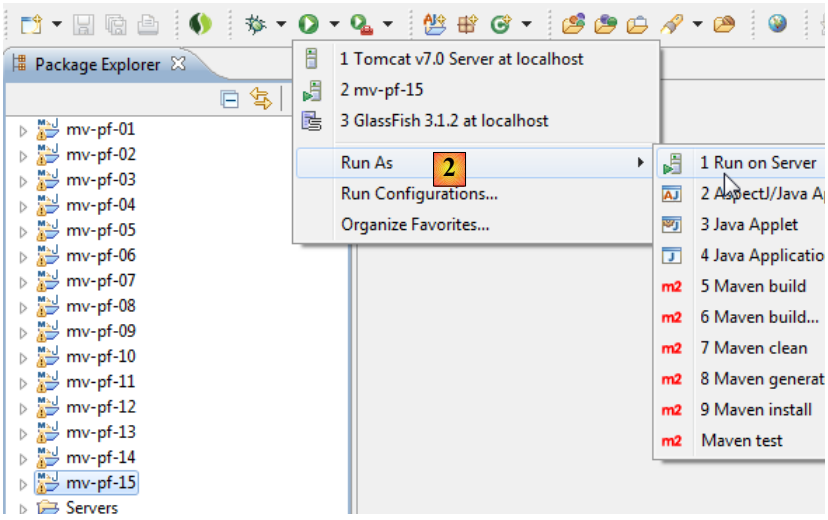
5.19 Conclusion

Nous en savons assez pour porter notre application exemple sur Primefaces. Nous n'avons vu qu'une quinzaine de composants alors que la bibliothèque en possède plus de 100. Le lecteur est invité à chercher le composant qui lui manque, directement sur le site de Primefaces.

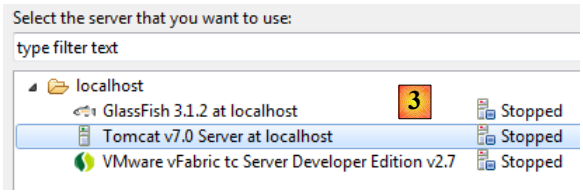
5.20 Les tests avec Eclipse

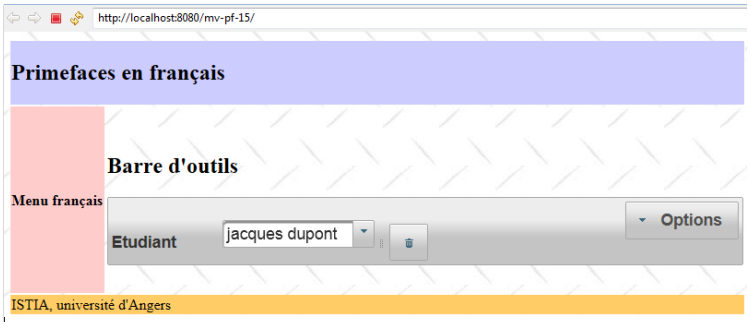
Les projets Maven sont disponibles sur le site des exemples [1] :





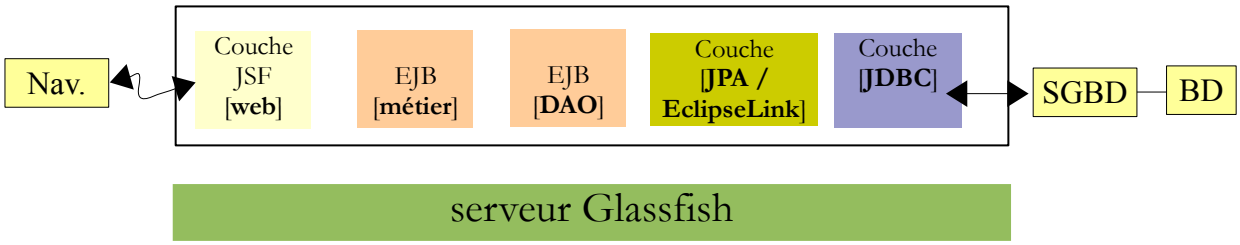
Une fois importés dans Eclipse, on peut les exécuter [2]. On sélectionne Tomcat en [3]. Ils s'affichent alors dans le navigateur interne d'Eclipse [3].


Select the server that you want to use:
type filter text
localhost
GlassFish 3.1.2 at localhost
Tomcat v7.0 Server at localhost
VMware vFabric tc Server Developer Edition v2.7
3
Stopped
Stopped
Stopped

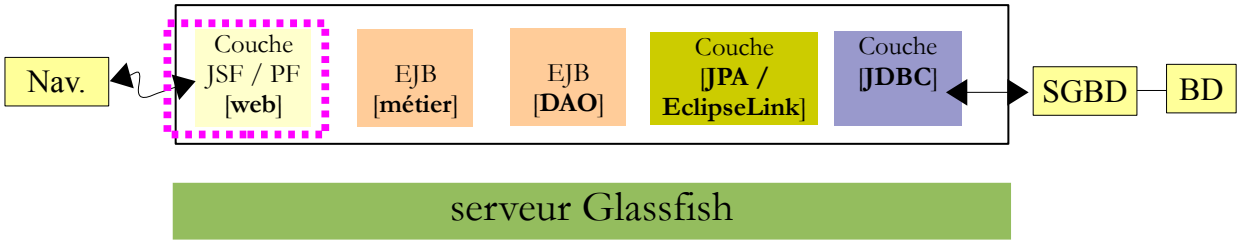

http://localhost:8080/mv-pf-15/
Primefaces en français
Menu français
Barre d'outils
Etudiant
jacques dupont
Options
ISTIA, université d'Angers

6 Application exemple-03 : rdvmedecins-pf-ejb

Rappelons la structure de l'application exemple développée pour le serveur Glassfish :

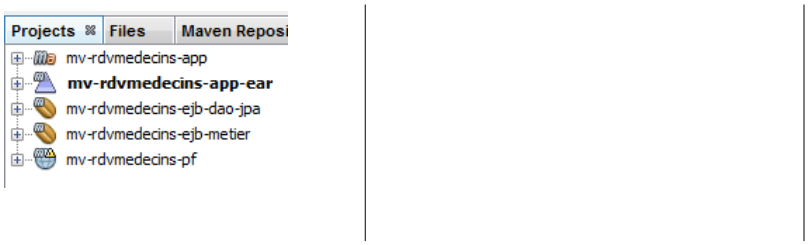


Nous ne changeons rien à cette architecture si ce n'est la couche web qui sera ici réalisée à l'aide de JSF et Primefaces.



6.1 Le projet Netbeans

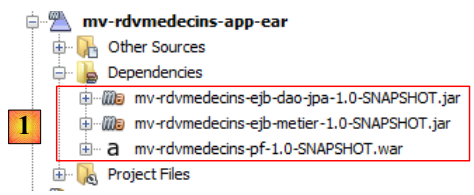
Ci-dessus, les couches [métier] et [DAO] sont celles de l'exemple 01 JSF / EJB / Glassfish. Nous les réutilisons.



- [mv-rdvmedecins-ejb-dao-jpa] : projet EJB des couches [DAO] et [JPA] de l'exemple 01,
- [mv-rdvmedecins-ejb-metier] : projet EJB de la couche [métier] de l'exemple 01,
- [mv-rdvmedecins-pf] : projet de la couche [web] / Primefaces – nouveau,
- [mv-rdvmedecins-app-ear] : projet d'entreprise pour déployer l'application sur le serveur Glassfish – nouveau.

6.2 Le projet d'entreprise

Le projet d'entreprise ne sert qu'au déploiement des trois modules [mv-rdvmedecins-ejb-dao-jpa], [mv-rdvmedecins-ejb-metier], [mv-rdvmedecins-pf] sur le serveur Glassfish. Le projet Netbeans est le suivant :



Le projet n'existe que pour ces trois dépendances [1] définies dans le fichier [pom.xml] de la façon suivante :

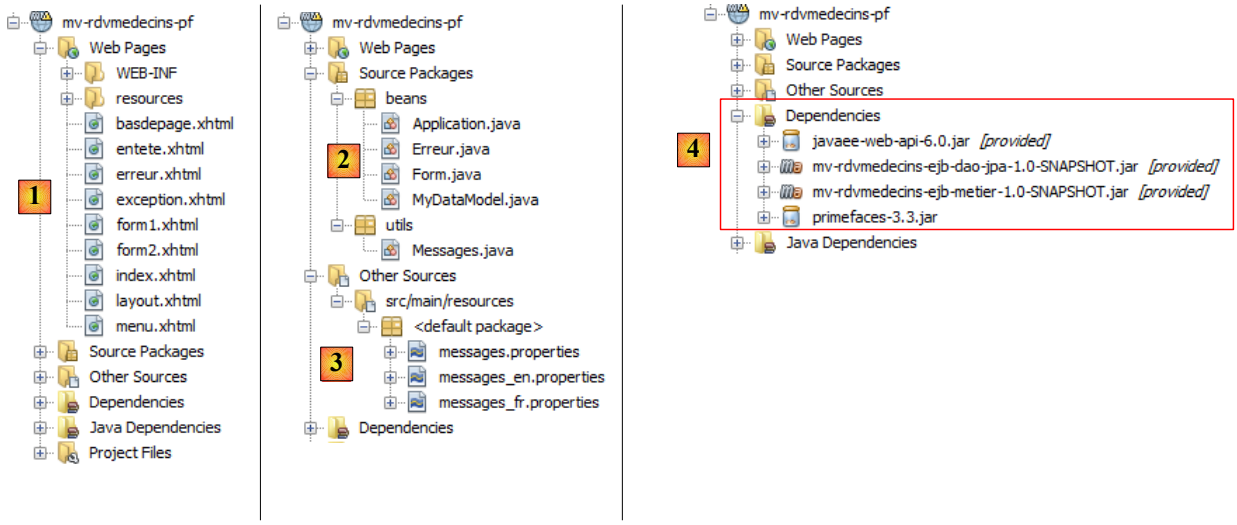
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.    <modelVersion>4.0.0</modelVersion>
4.    <parent>
5.      <artifactId>mv-rdvmedecins-app</artifactId>
6.      <groupId>istia.st</groupId>
7.      <version>1.0-SNAPSHOT</version>
8.    </parent>
9.
10.   <groupId>istia.st</groupId>
11.   <artifactId>mv-rdvmedecins-app-ear</artifactId>
12.   <version>1.0-SNAPSHOT</version>
13.   <packaging>ear</packaging>
14.
15.   <name>mv-rdvmedecins-app-ear</name>
16.
17.   ...
18.     <dependencies>
19.         <dependency>
20.             <groupId>${project.groupId}</groupId>
21.             <artifactId>mv-rdvmedecins-ejb-dao-jpa</artifactId>
22.             <version>${project.version}</version>
23.             <type>ejb</type>
24.         </dependency>
25.         <dependency>
26.             <groupId>${project.groupId}</groupId>
27.             <artifactId>mv-rdvmedecins-ejb-metier</artifactId>
28.             <version>${project.version}</version>
29.             <type>ejb</type>
30.         </dependency>
31.         <dependency>
32.             <groupId>${project.groupId}</groupId>
33.             <artifactId>mv-rdvmedecins-pf</artifactId>
34.             <version>${project.version}</version>
35.             <type>war</type>
36.         </dependency>
37.     </dependencies>
38. </project>
```

- lignes 10-13 : l'artifact Maven du projet d'entreprise,
- lignes 18-37 : les trois dépendances du projet. On notera bien le type de celles-ci (lignes 23, 29, 35).

Pour exécuter l'application web, il faudra exécuter ce projet d'entreprise.

6.3 Le projet web Primefaces

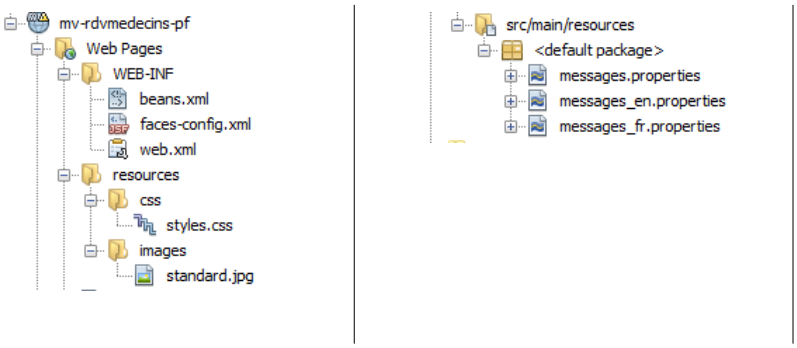
Le projet web Primefaces est le suivant :



- en [1], les pages du projet. La page [index.xhtml] est l'unique page du projet. Elle comporte trois fragments [form1.xhtml], [form2.xhtml] et [erreur.xhtml]. Les autres pages ne sont là que pour la mise en forme.
- en [2], les beans Java. Le bean [Application] de portée *application*, le bean [Form] de portée *session*. La classe [Erreur] encapsule une erreur. La classe [MyDataModel] sert de modèle à une balise <*dataTable*> de Primefaces,
- en [3], les fichiers de messages pour l'internationalisation,
- en [4], les dépendances. Le projet web a des dépendances sur le projet EJB de la couche [DAO], le projet EJB de la couche [métier] et Primefaces pour la couche [web].

6.4 La configuration du projet

La configuration du projet est celle des projets Primefaces ou JSF que nous avons étudiés. Nous listons les fichiers de configuration sans les réexpliquer.



[web.xml] : configure l'application web.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.    <context-param>
4.      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
5.      <param-value>client</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.PROJECT_STAGE</param-name>
9.      <param-value>Production</param-value>
10.   </context-param>
11.   <context-param>
12.     <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
13.     <param-value>true</param-value>
14.   </context-param>
```

```
15.   <servlet>
16.     <servlet-name>Faces Servlet</servlet-name>
17.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
18.     <load-on-startup>1</load-on-startup>
19.   </servlet>
20.   <servlet-mapping>
21.     <servlet-name>Faces Servlet</servlet-name>
22.     <url-pattern>/faces/*</url-pattern>
23.   </servlet-mapping>
24.   <session-config>
25.     <session-timeout>
26.       30
27.     </session-timeout>
28.   </session-config>
29.   <welcome-file-list>
30.     <welcome-file>faces/index.xhtml</welcome-file>
31.   </welcome-file-list>
32.   <error-page>
33.     <error-code>500</error-code>
34.     <location>/faces/exception.xhtml</location>
35.   </error-page>
36.   <error-page>
37.     <exception-type>Exception</exception-type>
38.     <location>/faces/exception.xhtml</location>
39.   </error-page>
40.
41. </web-app>
```

On notera, ligne 30 que la page [index.xhtml] est la page d'accueil de l'application.

[faces-config.xml] : configure l'application JSF

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <resource-bundle>
12.       <base-name>
13.         messages
14.       </base-name>
15.       <var>msg</var>
16.     </resource-bundle>
17.     <message-bundle>messages</message-bundle>
18.   </application>
19. </faces-config>
```

[beans.xml] : vide mais nécessaire pour l'annotation *@Named*

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://java.sun.com/xml/ns/javaee"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
5.  </beans>
```

[styles.css] : la feuille de style de l'application

```
1.  .col1{
2.     background-color: #ccccff
3.  }
4.
5.  .col2{
6.     background-color: #ffcccc
7.  }
```

La bibliothèque Primefaces vient avec ses propres feuilles de style. La feuille de style ci-dessus n'est utilisée que pour la page à afficher en cas d'exception, une page non gérée par l'application. C'est la page [exception.xhtml] qui est alors affichée.

[messages_fr.properties] : le fichier des messages en français

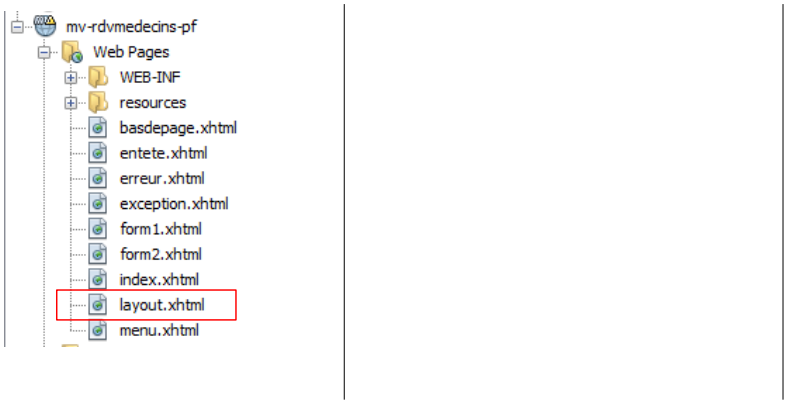
```
# layout
layout.entete=Les M\u00e9decins Associ\u00e9s
layout.basdepage=ISTIA, universit\u00e9 d'Angers - application propuls\u00e9e par PrimeFaces et JQuery

# exception
exception.header=L'exception suivante s'est produite
exception.httpCode=Code HTTP de l'erreur
exception.message=Message de l'exception
exception.requestUri=Url demand\u00e9e lors de l'erreur
exception.servletName=Nom de la servlet demand\u00e9e lorsque l'erreur s'est produite

# formulaire 1
form1.titre=R\u00e9servations
form1.medecin=M\u00e9decin
form1.jour=Jour
form1.options=Options
form1.francais=Fran\u00e7ais
form1.anglais=Anglais
form1.rafraichir=Rafra\u00eechir
form1.precedent=Jour pr\u00e9c\u00e9dent
form1.suivant=Jour suivant
form1.agenda=Affiche l'agenda du m\u00e9decin choisi pour le jour choisi
form1.today=Aujourd'hui

# formulaire 2
form2.titre=Agenda de {0} {1} {2} le {3}
form2.titre_detail=Agenda de {0} {1} {2} le {3}
form2.creneauHoraire=Cr\u00e9neau horaire
form2.client=Client
form2.accueil=Accueil
form2.supprimer=Supprimer
form2.reserver=R\u00e9server
form2.valider=Valider
form2.annuler=Annuler
form2.erreur=Erreur
form2.emtyMessage=Pas de cr\u00e9neaux entr\u00e9s dans la base
form2.suppression.confirmation=Etes-vous s\u00fbr(e) ?
form2.suppression.message=Suppression d'un rendez-vous
form2.supprimer.oui=Oui
form2.supprimer.non=Non
form2.erreurClient=Client [{0}] inconnu
form2.erreurClient_detail=Client {0} inconnu
form2.erreurAction=Action non autoris\u00e9e
form2.erreurAction_detail=Action non autoris\u00e9e

# erreur
erreur.titre=Une erreur s'est produite.
erreur.exceptions=Cha\u00eene des exceptions
erreur.type=Type de l'exception
erreur.message=Message associ\u00e9
erreur.accueil=Accueil
```

[messages_en.properties] : le fichier des messages en anglais

```
# layout
layout.entete=Associated Doctors
layout.basdepage=ISTIA, Angers university - Application powered by PrimeFaces and JQuery

# exception
exception.header=The following exceptions occurred
exception.httpCode=Error HTTP code
exception.message=Exception message
exception.requestUri=Url targeted when error occurred
exception.servletName=Servlet targeted's name when error occurred

# formulaire 1
form1.titre=Reservations
form1.medecin=Doctor
form1.jour=Date
```

```
form1.options=Options
form1.francais=French
form1.anglais=English
form1.rafraichir=Refresh
form1.precedent=Previous Day
form1.suivant=Next day
form1.agenda=Show the doctor's diary for the chosen doctor and the chosen day
form1.today=Today

# formulaire 2
form2.titre={0} {1} {2}'' diary on {3}
form2.titre_detail={0} {1} {2}'' diary on {3}
form2.creneauHoraire=Time Period
form2.client=Client
form2.accueil=Welcome Page
form2.supprimer=Delete
form2.reserver=Reserve
form2.valider=Submit
form2.annuler=Cancel
form2.erreur=Error
form2.emtyMessage=No Time periods in the database
form2.suppression.confirmation=Are-you sure ?
form2.suppression.message=Booking deletion
form2.supprimer.oui=Yes
form2.supprimer.non=No
form2.erreurClient=Unknown Client {0}
form2.erreurClient_detail=Unknown Client [{0}]
form2.erreurAction=Unauthorized action
form2.erreurAction_detail=Action non autoris\u00e9e

# erreur
erreur.titre=The following exceptions occurred
erreur.exceptions=Exceptions' chain
erreur.type=Exception type
erreur.message=Associated Message
erreur.accueil=Welcome
```

6.5 Le modèle des pages [layout.xhtml]



Le modèle [layout.xhtml] est le suivant :

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
  <f:view locale="#{form.locale}">
    <h:head>
      <title>JSF</title>
```

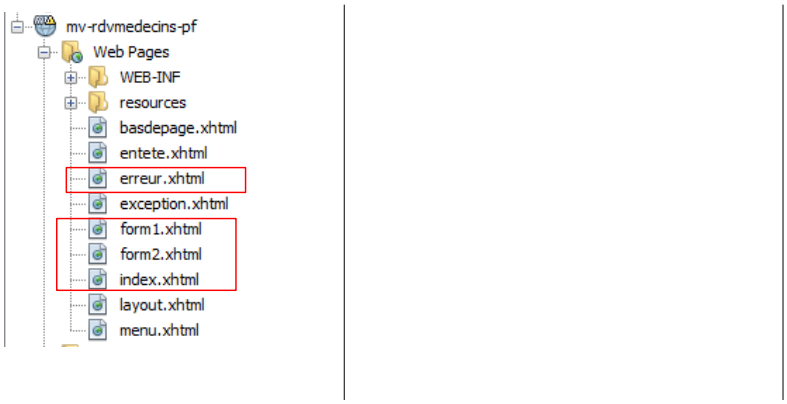
```
12.       <h:outputStylesheet library="css" name="styles.css"/>
13.     </h:head>
14.     <h:body style="background-image: url('#{request.contextPath}/resources/images/standard.jpg');">
15.       <h:form id="formulaire">
16.         <table style="width: 1200px">
17.           <tr>
18.             <td colspan="2" bgcolor="#ccccff">
19.               <ui:include src="entete.xhtml"/>
20.             </td>
21.           </tr>
22.           <tr>
23.             <td style="width: 10px;" bgcolor="#ffcccc">
24.               <ui:include src="menu.xhtml"/>
25.             </td>
26.             <td>
27.               <p:outputPanel id="contenu">
28.                 <ui:insert name="contenu">
29.                   <h2>Contenu</h2>
30.                 </ui:insert>
31.               </p:outputPanel>
32.             </td>
33.           </tr>
34.           <tr bgcolor="#ffcc66">
35.             <td colspan="2">
36.               <ui:include src="basdepage.xhtml"/>
37.             </td>
38.           </tr>
39.         </table>
40.       </h:form>
41.     </h:body>
42.   </f:view>
43. </html>
```

L'unique partie variable de ce modèle est la zone des lignes 28-30. Cette zone se trouve dans la zone d'id **:formulaire:contenu** (ligne 27). On s'en souviendra. Les appels AJAX qui mettent à jour cette zone auront l'attribut **update=":formulaire:contenu"**. Par ailleurs, le formulaire commence à la ligne 15. Donc le fragment inséré en lignes 28-30 s'insère dans ce formulaire.

L'aspect donné par ce modèle est le suivant :

La partie dynamique de la page viendra s'insérer dans la zone encadrée ci-dessus.

6.6 La page [index.xhtml]



Le projet affiche toujours la même page, la page [index.xhtml] suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <ui:composition template="layout.xhtml">
9.      <ui:define name="contenu">
10.       <ui:fragment rendered="#{form.form1Rendered}">
11.         <ui:include src="form1.xhtml"/>
12.       </ui:fragment>
13.       <ui:fragment rendered="#{form.form2Rendered}">
14.         <ui:include src="form2.xhtml"/>
15.       </ui:fragment>
16.       <ui:fragment rendered="#{form.erreurRendered}">
17.         <ui:include src="erreur.xhtml"/>
18.       </ui:fragment>
19.     </ui:define>
20.   </ui:composition>
21. </html>
```

- lignes 8-9 : ce fragment XHTML viendra s'insérer dans la zone dynamique du modèle [layout.xhtml],
- la page comprend trois sous-fragments :
    - [form1.xhtml], lignes 10-12 ;
    - [form2.xhtml], lignes 13-15 ;
    - [erreur.xhtml], lignes 16-18.

La présence de ces fragments dans [index.xhtml] est contrôlée par des booléens du modèle [Form.java] associé à la page. Donc en jouant sur ceux-ci, la page rendue diffère.

Le fragment [form1.xhtml] a le rendu suivant :

Le fragment [form2.xhtml] a le rendu suivant :

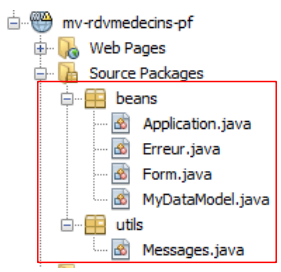
Créneau horaire	Client
08:00 - 08:20	Mr Jules MARTIN
08:20 - 08:40	
08:40 - 09:00	

Le fragment [erreur.xhtml] a le rendu suivant :

Une erreur s'est produite.

Chaîne des exceptions	
Type de l'exception	**Message associé**
javax.ejb.EJBException	Unable to complete container-managed transaction.
javax.transaction.SystemException	
javax.transaction.xa.XAException	com.sun.appserv.connectors.internal.api.PoolingException: javax.resource.spi.LocalTransactionException: Connection.close() has already been called. Invalid operation in this state.

6.7 Les beans du projet



La classe du paquetage [utils] a déjà été présentée : la classe [Messages] est une classe qui facilite l'internationalisation des messages d'une application. Elle a été étudiée au paragraphe 2.8.5.7, page 124.

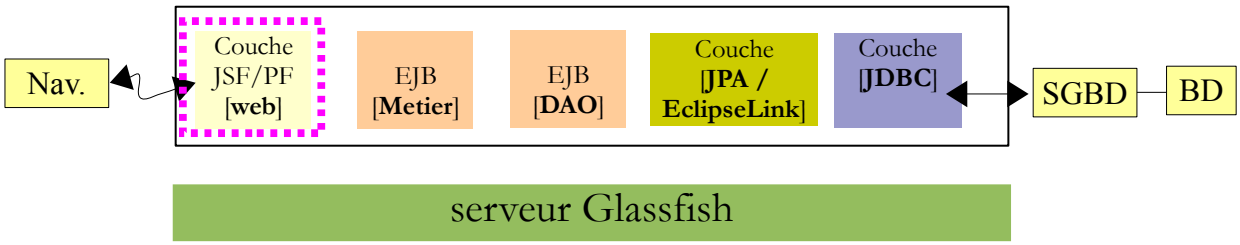
6.7.1 Le bean Application

Le bean [Application.java] est un bean de portée *application*. On se rappelle que ce type de bean sert à mémoriser des données en lecture seule et disponibles pour tous les utilisateurs de l'application. Ce bean est le suivant :

```
1.  package beans;
2.
3.  import javax.ejb.EJB;
4.  import javax.enterprise.context.ApplicationScoped;
5.  import javax.inject.Named;
6.  import rdvmedecins.metier.service.IMetierLocal;
7.
8.  @Named(value = "application")
9.  @ApplicationScoped
10. public class Application {
11.
12.   // couche métier
13.   @EJB
```

```
14.   private IMetierLocal metier;
15.
16.   public Application() {
17.   }
18.
19.   // getters
20.
21.   public IMetierLocal getMetier() {
22.     return metier;
23.   }
24.
25. }
```

- ligne 8 : on donne au bean le nom **application**,
- ligne 9 : il est de portée **application**,
- lignes 13-14 : une référence sur l'interface locale de la couche [métier] lui sera injectée par le conteneur EJB du serveur d'application. Rappelons-nous l'architecture de l'application :



L'application JSF et l'EJB [Metier] vont s'exécuter dans la même JVM (Java Virtual Machine). Donc la couche [JSF] va utiliser l'interface locale de l'EJB. C'est tout. Le bean [Application] ne contient rien d'autre. Pour avoir accès à la couche [métier], les autres beans iront la chercher dans ce bean.

6.7.2 Le bean [Erreur]

La classe [Erreur] est la suivante :

```
1.  package beans;
2.
3.  public class Erreur {
4.
5.    public Erreur() {
6.    }
7.
8.    // champ
9.    private String classe;
10.   private String message;
11.
12.   // constructeur
13.   public Erreur(String classe, String message){
14.     this.setClasse(classe);
15.     this.message=message;
16.   }
17.
18.   // getters et setters
19. ...
20. }
```

- ligne 9, le nom d'une classe d'exception si une exception a été lancée,
- ligne 10 : un message d'erreur.

6.7.3 Le bean [Form]

Son code est le suivant :

```java
package beans;

import java.io.IOException;
...

@Named(value = "form")
@SessionScoped
public class Form implements Serializable {

  public Form() {
  }

// bean Application
  @Inject
  private Application application;

  // cache de la session
  private List<Medecin> medecins;
  private List<Client> clients;
  private Map<Long, Medecin> hMedecins = new HashMap<Long, Medecin>();
  private Map<Long, Client> hClients = new HashMap<Long, Client>();
  private Map<String, Client> hIdentitesClients = new HashMap<String, Client>();

  // modèle
  private Long idMedecin;
  private Date jour = new Date();
  private Boolean form1Rendered = true;
  private Boolean form2Rendered = false;
  private Boolean erreurRendered = false;
  private String form2Titre;
  private AgendaMedecinJour agendaMedecinJour;
  private Long idCreneauChoisi;
  private Medecin medecin;
  private Long idClient;
  private CreneauMedecinJour creneauChoisi;
  private List<Erreur> erreurs;
  private Boolean erreur = false;
  private String identiteClient;
  private String action;
  private String msgErreurClient;
  private Boolean erreurClient;
  private String msgErreurAction;
  private Boolean erreurAction;
  private String locale = "fr";

  @PostConstruct
  private void init() {
    // on met les médecins et les clients en cache
    try {
      medecins = application.getMetier().getAllMedecins();
      clients = application.getMetier().getAllClients();
    } catch (Throwable th) {
      // on note l'erreur
      prepareVueErreur(th);
      return;
    }

    // les dictionnaires
    for (Medecin m : medecins) {
      hMedecins.put(m.getId(), m);
    }
    for (Client c : clients) {
      hClients.put(c.getId(), c);
      hIdentitesClients.put(identite(c), c);
    }
  }

  ...

  // affichage vue
  private void setForms(Boolean form1Rendered, Boolean form2Rendered, Boolean erreurRendered) {
    this.form1Rendered = form1Rendered;
    this.form2Rendered = form2Rendered;
    this.erreurRendered = erreurRendered;
  }

```

```
77.   // préparation vueErreur
78.   private void prepareVueErreur(Throwable th) {
79.     // on crée la liste des erreurs
80.     erreurs = new ArrayList<Erreur>();
81.     erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
82.     while (th.getCause() != null) {
83.       th = th.getCause();
84.       erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
85.     }
86. // la vue des erreurs est affichée
87.     setForms(true, false, true);
88.   }
89.
90.   // getters et setters
91.   ...
92. }
```

- lignes 6-8 : la classe [Form] est un bean de nom **form** et de portée **session**. On rappelle qu'alors la classe doit être sérialisable,
- lignes 14-15 : le bean **form** a une référence sur le bean **application**. Celle-ci sera injectée par le conteneur de servlets dans lequel s'exécute l'application (présence de l'annotation @**Inject**).
- lignes 17-44 : le modèle des pages [form1.xhtml, form2.xhtml, erreur.xhtml]. L'affichage de ces pages est contrôlé par les booléens des lignes 27-29. On remarquera que par défaut, c'est la page [form1.xhtml] qui est rendue (ligne 27),
- lignes 46-47 : la méthode **init** est exécutée juste après l'instanciation de la classe (présence de l'annotation @*PostConstruct*),
- lignes 50-51 : on demande à la couche [métier], la liste des médecins et des clients,
- lignes 59-65 : si tout s'est bien passé, les dictionnaires des médecins et des clients sont construits. Ils sont indexés par leur numéro. Ensuite, la page [form1.xhtml] sera affichée (ligne 27),
- ligne 54 : en cas d'erreur, le modèle de la page [erreur.xhtml] est construit. Ce modèle est la liste d'erreurs de la ligne 36,
- lignes 78-88 : la méthode [prepareVueErreur] construit la liste d'erreurs à afficher. La page [index.xhtml] affiche alors les fragments [form1.xhtml] et [erreur.xhtml] (ligne 87).

La page [erreur.xhtml] est la suivante :

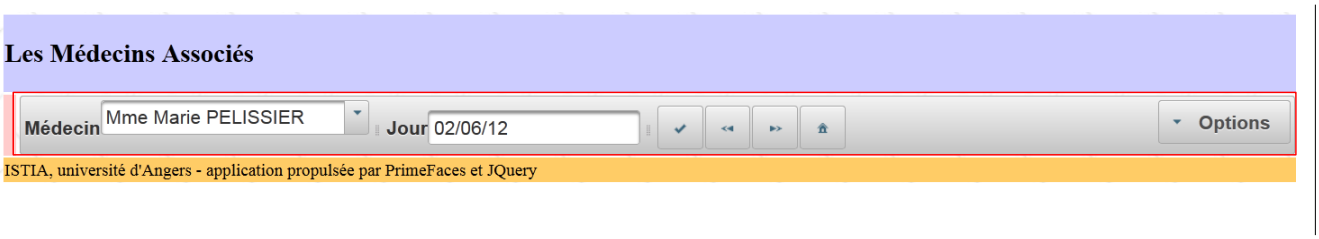
```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.
9.    <body>
10.     <p:panel header="#{msg['erreur.titre']}" closable="true" >
11.       <hr/>
12.       <p:dataTable value="#{form.erreurs}" var="erreur">
13.         <f:facet name="header">
14.           <h:outputText value="#{msg['erreur.exceptions']}"/>
15.         </f:facet>
16.         <p:column>
17.           <f:facet name="header">
18.             <h:outputText value="#{msg['erreur.type']}"/>
19.           </f:facet>
20.           <h:outputText value="#{erreur.classe}"/>
21.         </p:column>
22.         <p:column>
23.           <f:facet name="header">
24.             <h:outputText value="#{msg['erreur.message']}"/>
25.           </f:facet>
26.           <h:outputText value="#{erreur.message}"/>
27.         </p:column>
28.       </p:dataTable>
29.     </p:panel>
30.   </body>
31. </html>
```

Elle utilise une balise <**p:dataTable**> (lignes 12-28) pour afficher la liste des erreurs. Cela donne une page d'erreur analogue à la suivante :

Nous allons maintenant définir les différentes phases de la vie de l'application. Pour chaque action de l'utilisateur, nous étudierons les vues concernées et les gestionnaires des événements.

6.8 L'affichage de la page d'accueil

Si tout va bien, la première page affichée est [form1.xhtml]. Cela donne la vue suivante :



La page [form1.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets">
8.    <p:toolbar>
9.      <p:toolbarGroup align="left">
10.       ...
11.     </p:toolbarGroup>
12.     <p:toolbarGroup align="right">
13.  ...
14.     </p:toolbarGroup>
15.   </p:toolbar>
16. </html>
```

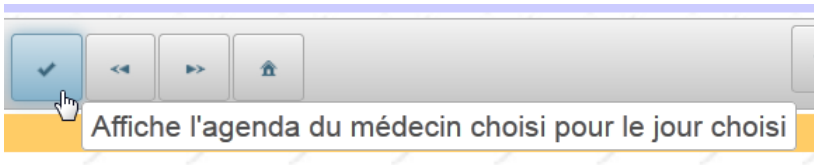
La barre d'outils encadrée dans la copie d'écran est le composant Primefaces *Toolbar*. Celui-ci est défini aux lignes 8-14. Il contient deux groupes de composants chacun étant défini par une balise <**toolbarGroup**>, lignes 9-11 et 12-14. L'un des groupes est aligné à gauche de la barre d'outils (ligne 9), l'autre à droite (ligne 12).

Examinons certains composants du groupe de gauche :

```
1.  <p:toolbar>
2.      <p:toolbarGroup align="left">
3.        <h:outputText value="#{msg['form1.medecin']}"/>
4.        <p:selectOneMenu value="#{form.idMedecin}" effect="fade">
5.          <f:selectItems value="#{form.medecins}" var="medecin" itemLabel="#{medecin.titre} #{medecin.prenom}
    #{medecin.nom}" itemValue="#{medecin.id}"/>
6.          <p:ajax update=":formulaire:contenu" listener="#{form.hideAgenda}" />
```

```
7.        </p:selectOneMenu>
8.        <p:separator/>
9.        <h:outputText value="#{msg['form1.jour']}"/>
10.       <p:calendar id="calendrier" value="#{form.jour}" readOnlyInputText="true">
11.         <p:ajax event="dateSelect" listener="#{form.hideAgenda}" update=":formulaire:contenu"/>
12.       </p:calendar>
13.       <p:separator/>
14.       <p:commandButton id="resa-agenda" icon="ui-icon-check" actionListener="#{form.getAgenda}"
    update=":formulaire:contenu"/>
15.       <p:tooltip for="resa-agenda" value="#{msg['form1.agenda']}"/>
16.       ...
17.     </p:toolbarGroup>
18. ...
```

- lignes 4-7 : le combo des médecins auquel on a ajouté un effet (`effect="fade"`),
- ligne 6 : un comportement AJAX. Lorsqu'il y aura un changement dans le combo, la méthode **[Form].hideAgenda** (`listener="#{form.hideAgenda}"`) sera exécutée et la zone dynamique **:formulaire:contenu** (`update=":formulaire:contenu"`) sera mise à jour,
- ligne 8 : inclut un séparateur dans la barre d'outils,
- lignes 10-12 : la zone de saisie de la date. On utilise ici le calendrier de Primefaces. La zone de saisie est en lecture seule (`readOnlyInputText="true"`),
- ligne 11 : un comportement AJAX. Lorsqu'il y aura un changement de date, la méthode **[Form].hideAgenda** sera exécutée et la zone dynamique **:formulaire:contenu** mise à jour,
- ligne 14 : un bouton. Un clic dessus fait exécuter un appel AJAX vers la méthode **[Form].getAgenda ()**, le modèle sera alors modifié et la réponse du serveur sera utilisée pour mettre à jour la zone dynamique **:formulaire:contenu**,
- ligne 15 : la balise <**tooltip**> permet d'associer une bulle d'aide à un composant. L'**id** de ce dernier est désigné par l'attribut **for** du tooltip. Ici (`for="resa-agenda"`) désigne le bouton de la ligne 14 :



Cette page est alimentée par le modèle suivant :

```
1.  @Named(value = "form")
2.  @SessionScoped
3.  public class Form implements Serializable {
4.
5.    public Form() {
6.    }
7.
8.    // cache de la session
9.    private List<Medecin> medecins;
10.   private List<Client> clients;
11.   // modèle
12.   private Long idMedecin;
13.   private Date jour = new Date();
14.
15.   // liste des médecins
16.   public List<Medecin> getMedecins() {
17.     return medecins;
18.   }
19.
20.   // liste des clients
21.   public List<Client> getClients() {
22.     return clients;
23.   }
24.
25.   // agenda
26.   public void getAgenda() {
27.     ...
28.   }
```

- le champ de la ligne 12 alimente en lecture et écriture la valeur de la liste de la ligne 4 de la page. A l'affichage initial de la page, elle fixe la valeur sélectionnée dans le combo. A l'affichage initial, *idMedecin* est égal à *null*, donc c'est le premier médecin qui sera sélectionné,
- la méthode des lignes 16-18 génère les éléments du combo des médecins (ligne 5 de la page). Chaque option générée aura pour **label** (itemLabel) les **titre**, **nom**, **prénom** du médecin et pour valeur (itemValue), l'**id** du médecin,
- le champ de la ligne 13 alimente en lecture / écriture le champ de saisie de la ligne 10 de la page. A l'affichage initial, c'est donc la date du jour qui est affichée,
- lignes 26-28 : la méthode **getAgenda** gère le clic sur le bouton [Agenda] de la ligne 14 de la page. Elle est quasi identique à ce qu'elle était dans la version JSF :

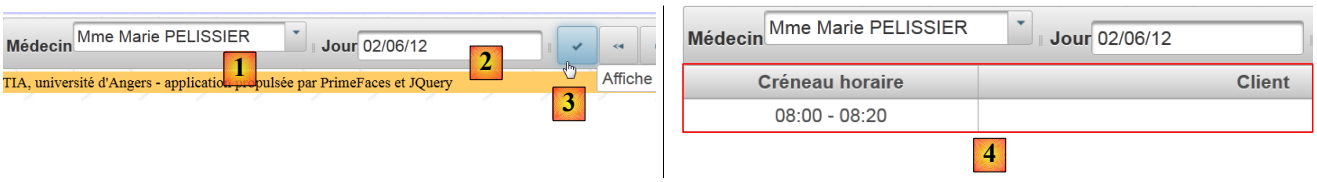
```
1.    // bean Application
2.    @Inject
3.    private Application application;
4.    // cache de la session
5.    private List<Medecin> medecins;
6.    private Map<Long, Medecin> hMedecins = new HashMap<Long, Medecin>();
7.    // modèle
8.    private Long idMedecin;
9.    private Date jour = new Date();
10.   private Boolean form1Rendered = true;
11.   private Boolean form2Rendered = false;
12.   private Boolean erreurRendered = false;
13.   private AgendaMedecinJour agendaMedecinJour;
14.   private Long idCreneauChoisi;
15.   private CreneauMedecinJour creneauChoisi;
16.   private List<Erreur> erreurs;
17.   private Boolean erreur = false;
18.
19.   public void getAgenda() {
20.     try {
21.       // on récupère le médecin
22.       medecin = hMedecins.get(idMedecin);
23.       // l'agenda du médecin pour un jour donné
24.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
25.       // on affiche le formulaire 2
26.       setForms(true, true, false);
27.     } catch (Throwable th) {
28.       // vue des erreurs
29.       prepareVueErreur(th);
30.     }
31.     // aucun créneau choisi pour l'instant
32.     creneauChoisi = null;
33. }
```

Nous ne commenterons pas ce code. Cela a déjà été fait.

6.9 Afficher l'agenda d'un médecin

6.9.1 Vue d'ensemble de l'agenda

C'est le cas d'utilisation suivant :

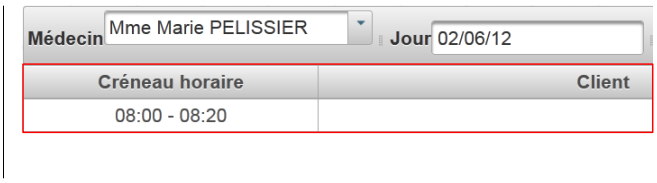


- en [1], on sélectionne un médecin [1] et un jour [2] puis on demande [3] l'agenda du médecin pour le jour choisi,
- en [4], celui-ci apparaît sous la barre d'outils.

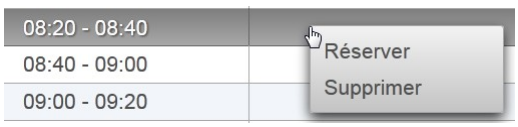
Le code de la page [form2.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:ui="http://java.sun.com/jsf/facelets"
8.        xmlns:c="http://java.sun.com/jsp/jstl/core">
9.
10.   <body>
11.     <!-- menu contextuel -->
12.     <p:contextMenu for="agenda">
13.       ...
14.     </p:contextMenu>
15.     <!-- agenda -->
16.     <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
17.                  selectionMode="single" selection="#{form.creneauChoisi}"
    emptyMessage="#{msg['form2.emtyMessage']}">
18.       <!-- colonne des horaires -->
19.       <p:column style="width: 100px">
20.         ...
21.       </p:column>
22.       <!-- colonne des clients -->
23.       <p:column style="width: 300px">
24.         ...
25.       </p:column>
26.     </p:dataTable>
27.
28.     <!-- confirmation suppression RV -->
29.     <p:confirmDialog id="confirmDialog" message="#{msg['form2.suppression.confirmation']}"
30.                      header="#{msg['form2.suppression.message']}" severity="alert"
    widgetVar="confirmation">
31.       ...
32.     </p:confirmDialog>
33.
34.     <!-- message d'erreur -->
35.     <p:dialog header="#{msg['form2.erreur']}" widgetVar="dlgErreur" height="100" >
36.       ...
37.     </p:dialog>
38.
39.     <!-- gestion du retour serveur -->
40.     <script type="text/javascript">
41.       ...
42.       }
43.     </script>
44.   </body>
45. </html>
```

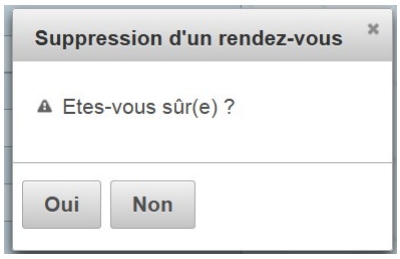
- lignes 16-26 : l'élément principal de la page est la table <**dataTable**> qui affiche l'agenda du médecin,



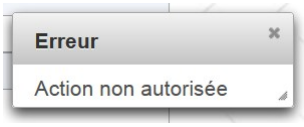
- lignes 12-14 : nous utiliserons un menu contextuel pour ajouter / supprimer un rendez-vous :



- lignes 29-32 : une boîte de confirmation sera affichée lorsque l'utilisateur voudra supprimer un rendez-vous :



- lignes 35-37 : une boîte de dialogue sera utilisée pour signaler une erreur :



- lignes 40-43 : nous aurons besoin d'introduire un peu de Javascript.

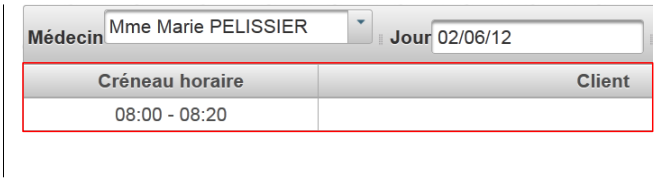
6.9.2 Le tableau des rendez-vous

Nous abordons ici le modèle d'un tableau de données tel qu'étudié au paragraphe 5.15, page 311.

Examinons l'élément principal de la page, le tableau qui affiche l'agenda :

```
1.  <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
2.                   selectionMode="single" selection="#{form.creneauChoisi}"
    emptyMessage="#{msg['form2.emtyMessage']}">
3.        <!-- colonne des horaires -->
4.        <p:column style="width: 100px">
5.          ...
6.        </p:column>
7.        <!-- colonne des clients -->
8.        <p:column style="width: 300px">
9.          ...
10.       </p:column>
11.     </p:dataTable>
```

Le rendu est le suivant :



C'est un tableau à deux colonnes (lignes 4-6 et 8-10) alimenté par la source **[Form].getMyDataModel()** (`value="#{form.myDataModel}"`). Une seule ligne peut être sélectionnée à la fois (`selectionMode="single"`). A chaque POST, une référence de l'élément sélectionné est affectée à **[Form].creneauChoisi** (`selection="#{form.creneauChoisi}"`).

On se rappelle que la méthode *getAgenda* a initialisé le champ suivant dans le modèle :

```
1.  // modèle
2.  private AgendaMedecinJour agendaMedecinJour;
```

Le modèle du tableau est obtenu par appel de la méthode *[Form].getMyDataModel* (attribut **value** de la balise <dataTable>) suivante :

```
1.    // le modèle du dataTable
2.    public MyDataModel getMyDataModel() {
3.      return new MyDataModel(agendaMedecinJour.getCreneauxMedecinJour());
4.  }
```

Examinons la classe [MyDataModel] qui sert de modèle à la balise <p:dataTable> :

```
1.  package beans;
2.
3.  import javax.faces.model.ArrayDataModel;
4.  import org.primefaces.model.SelectableDataModel;
5.  import rdvmedecins.metier.entites.CreneauMedecinJour;
6.
7.  public class MyDataModel extends ArrayDataModel<CreneauMedecinJour> implements
    SelectableDataModel<CreneauMedecinJour> {
8.
9.    // constructeurs
10.   public MyDataModel() {
11.   }
12.
13.   public MyDataModel(CreneauMedecinJour[] creneauxMedecinJour) {
14.     super(creneauxMedecinJour);
15.   }
16.
17.   @Override
18.   public Object getRowKey(CreneauMedecinJour creneauMedecinJour) {
19.     return creneauMedecinJour.getCreneau().getId();
20.   }
21.
22.   @Override
23.   public CreneauMedecinJour getRowData(String rowKey) {
24.     // liste des créneaux
25.     CreneauMedecinJour[] creneauxMedecinJour = (CreneauMedecinJour[]) getWrappedData();
26.     // la clé est un entier long
27.     long key = Long.parseLong(rowKey);
28.     // on recherche le créneau sélectionné
29.     for (CreneauMedecinJour creneauMedecinJour : creneauxMedecinJour) {
30.       if (creneauMedecinJour.getCreneau().getId().longValue() == key) {
31.         return creneauMedecinJour;
32.       }
33.     }
34.     // rien
35.     return null;
36.   }
37. }
```

- ligne 7 : la classe [MyDataModel] est le modèle de la balise <p**:dataTable**>. Cette classe a pour but de faire le lien entre l'élément **rowkey** qui est posté, avec l'élément associé à cette ligne,
- ligne 7 : la classe implémente l'interface [SelectableDataModel] au travers de la classe [ArrayDataModel]. Cela signifie que le paramètre du constructeur est un tableau. C'est ce tableau qui alimente la balise <**dataTable**>. Ici chaque ligne du tableau sera associée à un élément de type [CreneauMedecinJour],
- lignes 13-15 : le constructeur passe son paramètre à sa classe parent,
- lignes 18-20 : chaque ligne du tableau correspond à un créneau horaire et sera identifiée par l'**id** du créneau horaire (ligne 19). C'est cet **id** qui sera posté au serveur,
- ligne 23 : le code qui sera exécuté côté serveur lorsque l'**id** d'un créneau horaire sera posté. Le but de cette méthode est de rendre la référence de l'objet [CreneauMedecinJour] associé à cet **id**. Cette référence sera affectée à la cible de l'attribut **selection** de la balise <**dataTable**> :

```
1.  <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
2.  selectionMode="single" selection="#{form.creneauChoisi}" emptyMessage="#{msg['form2.emtyMessage']}">
```

Le champ *[Form].creneauChoisi* contiendra donc la référence de l'objet [CreneauMedecinJour] qu'on veut ajouter ou supprimer.

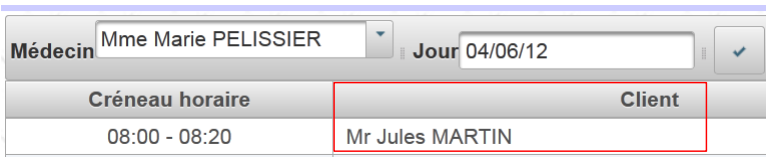
6.9.3 La colonne des créneaux horaires

La colonne des créneaux horaires est obtenue avec le code suivant :

```
1.  <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
2.                   selectionMode="single" selection="#{form.creneauChoisi}"
    emptyMessage="#{msg['form2.emtyMessage']}">
3.        <!-- colonne des horaires -->
4.        <p:column style="width: 100px">
5.          <f:facet name="header">
6.            <h:outputText value="#{msg['form2.creneauHoraire']}"/>
7.          </f:facet>
8.          <div align="center">
9.            <h:outputFormat value="{0,number,#00}:{1,number,#00} - {2,number,#00}:{3,number,#00}">
10.             <f:param value="#{creneauMedecinJour.creneau.hdebut}" />
11.             <f:param value="#{creneauMedecinJour.creneau.mdebut}" />
12.             <f:param value="#{creneauMedecinJour.creneau.hfin}" />
13.             <f:param value="#{creneauMedecinJour.creneau.mfin}" />
14.           </h:outputFormat>
15.         </div>
16.       </p:column>
17.
18.       <!-- colonne des clients -->
19.       <p:column style="width: 300px">
20.         ...
21.       </p:column>
22.     </p:dataTable>
```

- lignes 5-7 : l'entête de la colonne,
- lignes 8-15 : l'élément courant de la colonne. On notera ligne 9, l'utilisation de la balise <**h:outputFormat**> qui permet de formater des éléments à afficher. Le paramètre **value** indique la chaîne de caractères à afficher. La notation **{i,type,format}** désigne le paramètre n° i, le type de ce paramètre et son format. Il y a ici 4 paramètres numérotés de 0 à 3, le type de ceux-ci est numérique et il seront affichés avec deux chiffres,
- lignes 10-13 : les quatre paramètres attendus par la balise <h:outputFormat>.

6.9.4 La colonne des clients



La colonne des clients est obtenue avec le code suivant :

```
1.  <!-- agenda -->
2.      <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
3.                   selectionMode="single" selection="#{form.creneauChoisi}"
    emptyMessage="#{msg['form2.emtyMessage']}">
4.        <!-- colonne des horaires -->
5.        ...
6.        <!-- colonne des clients -->
7.        <p:column style="width: 300px">
8.          <f:facet name="header">
9.            <h:outputText value="#{msg['form2.client']}"/>
10.         </f:facet>
11.         <ui:fragment rendered="#{creneauMedecinJour.rv!=null}">
```

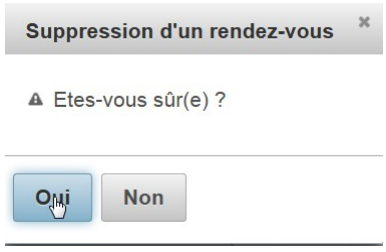
```
12.           <h:outputText value="#{creneauMedecinJour.rv.client.titre}
    #{creneauMedecinJour.rv.client.prenom} #{creneauMedecinJour.rv.client.nom}" />
13.         </ui:fragment>
14.         <ui:fragment rendered="#{creneauMedecinJour.rv==null and form.creneauChoisi!=null and
    form.creneauChoisi.creneau.id==creneauMedecinJour.creneau.id}">
15.           ...
16.         </ui:fragment>
17.       </p:column>
18.     </p:dataTable>
```

- lignes 8-10 : l'entête de la colonne,
- lignes 11-13 : l'élément courant lorsqu'il y a un rendez-vous pour le créneau horaire. Dans ce cas, on affiche les titre, prénom et nom du client pour qui ce rendez-vous a été pris,
- lignes 14-16 : un autre fragment sur lequel nous reviendrons.

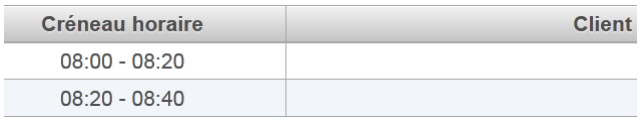
6.10 Suppression d'un rendez-vous

La suppression d'un rendez-vous correspond à la séquence suivante :

L'utilisateur peut supprimer un RV



Une confirmation lui est demandée



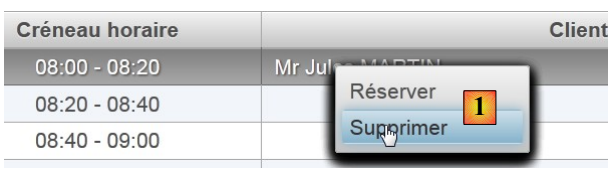
Le rendez-vous a été supprimé

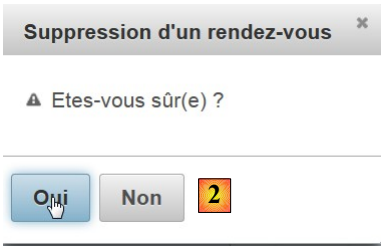
La vue concernée par cette action est la suivante :

```
1.  <!-- menu contextuel -->
2.      <p:contextMenu for="agenda">
3.  ...
4.        <p:menuitem value="#{msg['form2.supprimer']}" onclick="confirmation.show()"/>
5.      </p:contextMenu>
6.      <!-- agenda -->
7.      <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
8.                   selectionMode="single" selection="#{form.creneauChoisi}"
    emptyMessage="#{msg['form2.emtyMessage']}">
9.        ...
10.     </p:dataTable>
11.
12.     <!-- confirmation suppression RV -->
13.     <p:confirmDialog id="confirmDialog" message="#{msg['form2.suppression.confirmation']}"
14.        header="#{msg['form2.suppression.message']}" severity="alert" widgetVar="confirmation">
15.       <p:commandButton value="#{msg['form2.supprimer.oui']}" update=":formulaire:contenu"
    action="#{form.action}"
```

```
16.                         oncomplete="handleRequest(xhr, status, args); confirmation.hide()">
17.           <f:setPropertyActionListener value="supprimer" target="#{form.action}"/>
18.         </p:commandButton>
19.         <p:commandButton value="#{msg['form2.supprimer.non']}" onclick="confirmation.hide()"
    type="button" />
20.       </p:confirmDialog>
```

- lignes 2-5 : un menu contextuel lié au tableau de données (attribut **for**). Il a deux options [1] :





- ligne 4 : l'option [Supprimer] déclenche l'affichage de la boîte de dialogue [2] des lignes 13-20,
- ligne 15 : le clic sur le [Oui] déclenche l'exécution de [Form.action] qui va supprimer le rendez-vous. Normalement, le menu contextuel ne devrait pas offrir l'option [Supprimer] si l'élément sélectionné n'a pas de rendez-vous, et pas l'option [Réserver] si l'élément sélectionné a un rendez-vous. Nous n'avons pas réussi à ce que le menu contextuel soit aussi subtil. On y arrive pour le premier élément sélectionné mais ensuite on constate que le menu contextuel garde la configuration acquise pour cette première sélection. Il devient alors incorrect. Donc on a gardé les deux options et on a décidé de fournir un retour à l'utilisateur s'il supprimait un élément sans rendez-vous,
- ligne 16 : l'attribut *oncomplete* qui permet de définir du code Javascript à exécuter après exécution de l'appel AJAX. Ce code sera ici le suivant :

```
1.  <!-- message d'erreur -->
2.      <p:dialog header="#{msg['form2.erreur']}" widgetVar="dlgErreur" height="100" >
3.        <h:outputText value="#{form.msgErreur}" />
4.      </p:dialog>
5.
6.      <!-- gestion du retour serveur -->
7.      <script type="text/javascript">
8.        function handleRequest(xhr, status, args) {
9.          // erreur ?
10.         if(args.erreur) {
11.           dlgErreur.show();
12.         }
13.       }
14.     </script>
```

- ligne 10 : le code Javascript regarde si le dictionnaire *args* a l'attribut *erreur*. Si oui, il fait afficher la boîte de dialogue de la ligne 2 (attribut *widgetVar*). Cette boîte affiche le modèle *[Form].msgErreur*.

Regardons le code exécuté pour gérer la suppression d'un rendez-vous :

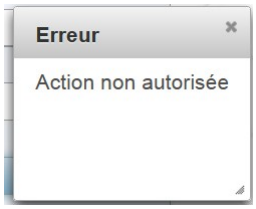
```
1.      <p:confirmDialog ...>
2.        <p:commandButton value="#{msg['form2.supprimer.oui']}" update=":formulaire:contenu"
    action="#{form.action}"
3.                         ...>
4.          <f:setPropertyActionListener value="supprimer" target="#{form.action}"/>
5.        </p:commandButton>
6.        ...
7.  </p:confirmDialog>
```

- ligne 2 : la méthode [Form].action va être exécutée,
- ligne 4 : avant son exécution, le champ *action* aura reçu la valeur '*supprimer*'.

La méthode [action] est la suivante :

```
1.  // action sur RV
2.    public void action() {
3.      // selon l'action désirée
4.      if (action.equals("supprimer")) {
5.        supprimer();
6.      }
7.      ...
8.    }
9.
10.   public void supprimer() {
11.     // faut-il faire qq chose ?
12.     Rv rv = creneauChoisi.getRv();
13.     if (rv == null) {
14.       signalerActionIncorrecte();
15.       return;
16.     }
17.     try {
18.       // suppression d'un Rdv
19.       application.getMetier().supprimerRv(rv);
20.       // on remet à jour l'agenda
21.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
22.       // on affiche form2
23.       setForms(true, true, false);
24.     } catch (Throwable th) {
25.       // vue erreurs
26.       prepareVueErreur(th);
27.     }
28.     // raz du créneau choisi
29.     creneauChoisi = null;
30. }
```

- ligne 4 : si l'action est 'supprimer', on exécute la méthode [supprimer],
- ligne 12 : on récupère le rendez-vous du créneau sélectionné. On rappelle que [creneauChoisi] a été initialisé par la référence de l'élément [CreneauMedecinJour] sélectionné,
- si ce rendez-vous existe, il est supprimé (ligne 19), l'agenda est régénéré (ligne 21) puis réaffiché (ligne 23),
- si la suppression a échoué, on affiche la page d'erreurs (ligne 26),
- si l'élément sélectionné n'a pas de rendez-vous (ligne 13) alors on est dans la situation où l'utilisateur a cliqué [Supprimer] sur un créneau qui n'a pas de rendez-vous. On signale cette erreur :



La méthode [signalerActionIncorrecte] est la suivante :

```
1.  // signaler une action incorrecte
2.    private void signalerActionIncorrecte() {
3.      // raz créneau choisi
4.      creneauChoisi = null;
5.      // erreur
6.      msgErreur = Messages.getMessage(null, "form2.erreurAction", null).getSummary();
7.      RequestContext.getCurrentInstance().addCallbackParam("erreur", true);
8.    }
```

- ligne 4 : on enlève la sélection,
- ligne 6 : on génère un message d'erreur internationalisé,
- ligne 7 : on ajoute dans le dictionnaire **args** de l'appel AJAX l'attribut ('erreur', true).

Revenons au code XHTML du bouton [Oui] :

```
1.  <p:commandButton value="#{msg['form2.supprimer.oui']}" update=":formulaire:contenu" action="#{form.action}"
2.  oncomplete="handleRequest(xhr, status, args); confirmation.hide()">
```

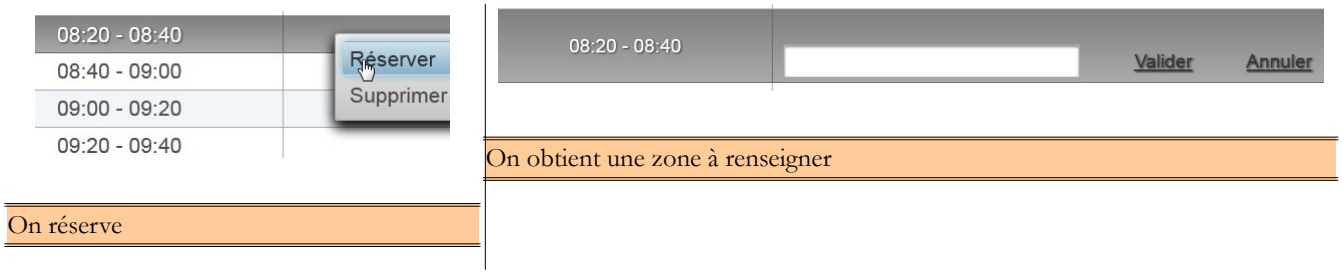
- ligne 2 : après exécution de la méthode [Form].action, la méthode Javascript *handleRequest* est exécutée :

```
1.       <!-- message d'erreur -->
2.      <p:dialog header="#{msg['form2.erreur']}" widgetVar="dlgErreur" height="100" >
3.        <h:outputText value="#{form.msgErreur}" />
4.      </p:dialog>
5.
6.      <!-- gestion du retour serveur -->
7.      <script type="text/javascript">
8.        function handleRequest(xhr, status, args) {
9.          // erreur ?
10.         if(args.erreur) {
11.           dlgErreur.show();
12.         }
13.       }
14. </script>
```

- ligne 10 : on regarde si le dictionnaire **args** a l'attribut nommé '*erreur*'. Si oui, la boîte de dialogue de la ligne 2 est affichée,
- ligne 3 : elle fait afficher le message d'erreur construit par le modèle.

6.11 Prise de rendez-vous

La prise de rendez-vous correspond à la séquence suivante :

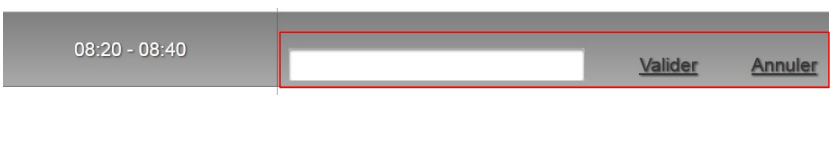


La vue impliquée par cette action est la suivante :

```
1.  <!-- menu contextuel -->
2.      <p:contextMenu for="agenda">
3.        <p:menuitem value="#{msg['form2.reserver']}" update=":formulaire:contenu" action="#{form.action}"
    oncomplete="handleRequest(xhr, status, args)">
4.          <f:setPropertyActionListener value="reserver" target="#{form.action}"/>
5.        </p:menuitem>
6.        ...
7.      </p:contextMenu>
8.      <!-- agenda -->
9.      <p:dataTable id="agenda" value="#{form.myDataModel}" var="creneauMedecinJour" style="width: 800px"
10.    selectionMode="single" selection="#{form.creneauChoisi}" emptyMessage="#{msg['form2.emtyMessage']}">
11.       <!-- colonne des horaires -->
12.       <p:column style="width: 100px">
13. ...
14.       </p:column>
15.       <!-- colonne des clients -->
16.       <p:column style="width: 300px">
17.         <f:facet name="header">
18.           <h:outputText value="#{msg['form2.client']}"/>
19.         </f:facet>
20. ...
21.         <ui:fragment rendered="#{creneauMedecinJour.rv==null and form.creneauChoisi!=null and
    form.creneauChoisi.creneau.id==creneauMedecinJour.creneau.id}">
22.           <p:autoComplete completeMethod="#{form.completeClients}" value="#{form.identiteClient}"
    size="30"/>
23.           <p:spacer width="50px"/>
```

```
24.           <p:commandLink action="#{form.action()}" value="#{msg['form2.valider']}"
    update=":formulaire:contenu" oncomplete="handleRequest(xhr, status, args)">
25.             <f:setPropertyActionListener value="valider" target="#{form.action}"/>
26.           </p:commandLink>
27.           <p:spacer width="50px"/>
28.           <p:commandLink action="#{form.action()}" value="#{msg['form2.annuler']}"
    update=":formulaire:contenu">
29.             <f:setPropertyActionListener value="annuler" target="#{form.action}"/>
30.           </p:commandLink>
31.         </ui:fragment>
32.       </p:column>
33.     </p:dataTable>
34. ...
```

- lignes 21-31 : affichent la chose suivante :



- ligne 21 : l'affichage a lieu s'il n'y a pas de rendez-vous et qu'il y a bien eu sélection et que l'id du créneau choisi correspond à celui de l'élément courant du tableau. Si on ne met pas cette condition, le fragment est affiché pour tous les créneaux,
- ligne 22 : la zone de saisie sera une zone de saisie assistée. On suppose ici qu'il peut y avoir beaucoup de clients,
- lignes 24-26 : le lien [Valider],
- lignes 28-30 : le lien [Annuler].

La zone de saisie assistée est générée par le code suivant :

```
<p:autoComplete completeMethod="#{form.completeClients}" value="#{form.identiteClient}" size="30"/>
```

La méthode *[Form].completeClients* est chargée de faire des propositions à l'utilisateur à partir des caractères tapés dans la zone de saisie :

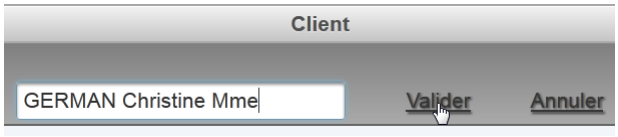
Les propositions sont de la forme [Nom prénom titre]. Le code de la méthode *[Form].completeClients* est le suivant :

```
1.    // la méthode du texte autocomplete
2.    public List<String> completeClients(String query) {
3.      List<String> identites = new ArrayList<String>();
4.      // on recherche les clients qui correspondent
5.      for (Client c : clients) {
6.        String identite = identite(c);
7.        if (identite.toLowerCase().startsWith(query.toLowerCase())) {
8.          identites.add(identite);
9.        }
10.     }
11.     return identites;
12.   }
13.
14.   private String identite(Client c) {
15.     return c.getNom() + " " + c.getPrenom() + " " + c.getTitre();
16. }
```

- ligne 2 : *query* est la chaîne de caractères tapée par l'utilisateur,
- ligne 3 : la liste des propositions. Au départ une liste vide,
- lignes 5-10 : on construit les identités [Nom prénom titre] des clients. Si une identité commence par *query* (ligne 7), elle est retenue dans la liste des propositions (ligne 8).

6.12 Validation d'un rendez-vous

La validation d'un rendez-vous correspond à la séquence suivante :



Créneau horaire	Client
08:00 - 08:20	Mme Christine GERMAN

Le code du lien [Valider] est le suivant :

```
1.            <p:commandLink action="#{form.action()}" value="#{msg['form2.valider']}"
    update=":formulaire:contenu" oncomplete="handleRequest(xhr, status, args)">
2.              <f:setPropertyActionListener value="valider" target="#{form.action}"/>
3.  </p:commandLink>
```

C'est donc la méthode *[Form].action()* qui va gérer cet évènement. Entre-temps, le modèle *[Form].action* aura reçu la chaîne 'valider'. Le code est le suivant :

```
1.    // bean Application
2.    @Inject
3.    private Application application;
4.    // cache de la session
5.  ...
6.    private Map<String, Client> hIdentitesClients = new HashMap<String, Client>();
7.    // modèle
8.    private Date jour = new Date();
9.    private Boolean form1Rendered = true;
10.   private Boolean form2Rendered = false;
11.   private Boolean erreurRendered = false;
12.   private AgendaMedecinJour agendaMedecinJour;
13.   private CreneauMedecinJour creneauChoisi;
14.   private List<Erreur> erreurs;
15.   private Boolean erreur = false;
16.   private String identiteClient;
17.   private String action;
18.   private String msgErreur;
19.
20.   @PostConstruct
21.   private void init() {
22.     ...
23.     for (Client c : clients) {
24.       hClients.put(c.getId(), c);
25.       hIdentitesClients.put(identite(c), c);
26.     }
27.   }
28.
29.   // action sur RV
30.   public void action() {
31.     // selon l'action désirée
32. ...
33.     if (action.equals("valider")) {
34.       validerResa();
35.     }
36. }
37.
38.   // validation Rv
39.   public void validerResa() {
40.     // validation de la réservation
41.     try {
42.       // le client existe-t-il ?
43.       Boolean erreur = !hIdentitesClients.containsKey(identiteClient);
44.       if (erreur) {
```

```
45.         msgErreur = Messages.getMessage(null, "form2.erreurClient", new Object[]
    {identiteClient}).getSummary();
46.         RequestContext.getCurrentInstance().addCallbackParam("erreur", true);
47.         return;
48.       }
49.       // on ajoute le Rv
50.       application.getMetier().ajouterRv(jour, creneauChoisi.getCreneau(),
    hIdentitesClients.get(identiteClient));
51.       // on remet à jour l'agenda
52.       agendaMedecinJour = application.getMetier().getAgendaMedecinJour(medecin, jour);
53.       // on affiche form2
54.       setForms(true, true, false);
55.     } catch (Throwable th) {
56.       // vue erreurs
57.       prepareVueErreur(th);
58.     }
59.     // raz du créneau choisi
60.     creneauChoisi = null;
61.     // raz client
62.     identiteClient = null;
63. }
```

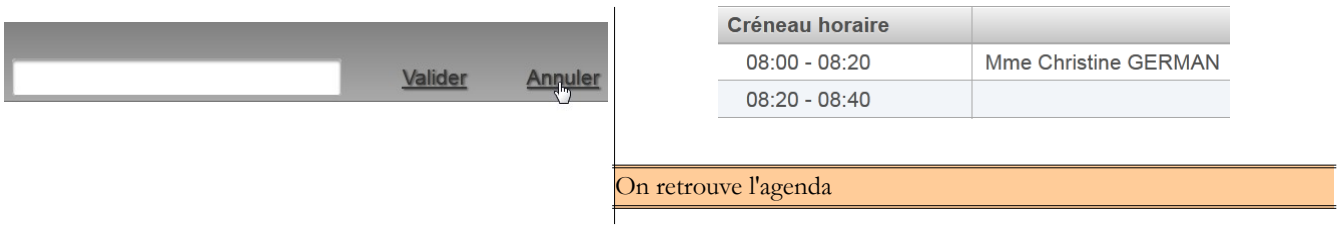
- lignes 33-35 : à cause du valeur du champ *action*, la méthode [validerResa] va être exécutée,
- ligne 43 : on vérifie d'abord que le client existe. En effet, dans la zone de saisie assistée, l'utilisateur a pu faire une saisie en dur sans s'aider des propositions qui lui ont été faites. La saisie assistée est associée au modèle *[Form].identiteClient*. On regarde donc si cette identité existe dans le dictionnaire *identitesClients* construit à l'instanciation du modèle (ligne 20). Celui-ci associe à une identité client de type [Nom prénom titre], le client lui-même (ligne 25),
- ligne 44 : si le client n'existe pas, on renvoie une erreur au navigateur,
- ligne 45 : un message d'erreur internationalisé,
- ligne 46 : on ajoute l'attribut ('erreur',true) au dictionnaire **args** de l'appel AJAX. L'appel AJAX a été défini de la façon suivante :

```
1.  <p:commandLink action="#{form.action()}" value="#{msg['form2.valider']}" update=":formulaire:contenu"
    oncomplete="handleRequest(xhr, status, args)">
2.              <f:setPropertyActionListener value="valider" target="#{form.action}"/>
3.  </p:commandLink>
```

  Ligne 3 ci-dessus, on voit que le lien [Valider] a un attribut **oncomplete**. C'est cet attribut qui va faire afficher le message d'erreur selon une technique déjà rencontrée.
- ligne 50 : on demande à la couche [métier] d'ajouter un rendez-vous pour le jour choisi (*jour*), le créneau horaire choisi (*creneauChoisi.getCreneau()*) et le client choisi (*hIdentitesClients.get(identiteClient)*),
- ligne 52 : on demande à la couche [métier] de rafraîchir l'agenda du médecin. On verra le rendez-vous ajouté plus toutes les modifications que d'autres utilisateurs de l'application ont pu faire,
- ligne 54 : on réaffiche l'agenda [form2.xhtml],
- ligne 57 : on affiche la page d'erreur si une erreur se produit.

6.13 Annulation d'une prise de rendez-vous

Cela correspond à la séquence suivante :



Le bouton [Annuler] dans la page [form2.xhtml] est le suivant :

```
1.  <p:commandLink action="#{form.action()}" value="#{msg['form2.annuler']}" update=":formulaire:contenu">
```

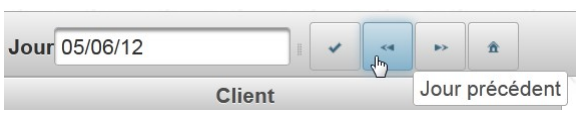
```
2.             <f:setPropertyActionListener value="annuler" target="#{form.action}"/>
3.           </p:commandLink>
```

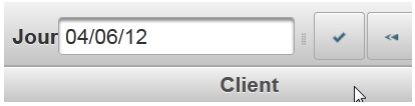
La méthode *[Form].action* est donc appelée :

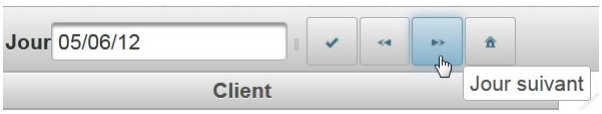
```
1.  // action sur RV
2.    public void action() {
3.      // selon l'action désirée
4.  ...
5.      if (action.equals("annuler")) {
6.        annulerRv();
7.      }
8.    }
9.
10. // annulation prise de Rdv
11.   public void annulerRv() {
12.     // on affiche form2
13.     setForms(true, true, false);
14.     // raz du créneau choisi
15.     creneauChoisi = null;
16.     // raz client
17.     identiteClient = null;
18.   }
```

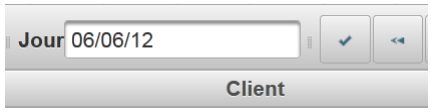
6.14 Navigation dans le calendrier

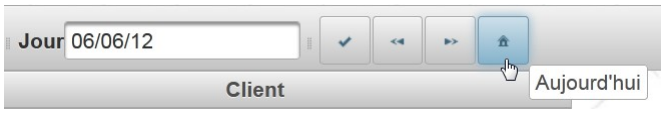
La barre d'outils permet de naviguer dans le calendrier :

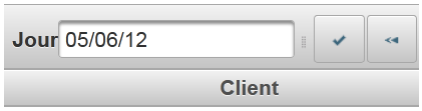












Non montré sur les copies d'écran ci-dessus, l'agenda est mis à jour avec les rendez-vous du nouveau jour choisi.

Les balises des trois boutons concernés sont les suivantes dans [form1.xhtml] :

```
1.    <p:toolbar>
2.      <p:toolbarGroup align="left">
3.  ...
4.        <h:outputText value="#{msg['form1.jour']}"/>
5.        <p:calendar id="calendrier" value="#{form.jour}" readOnlyInputText="true">
6.          <p:ajax event="dateSelect" listener="#{form.hideAgenda}" update=":formulaire:contenu"/>
7.        </p:calendar>
8.        <p:separator/>
9.        <p:commandButton id="resa-agenda" icon="ui-icon-check" actionListener="#{form.getAgenda}"
    update=":formulaire:contenu"/>
10.       <p:tooltip for="resa-agenda" value="#{msg['form1.agenda']}"/>
```

```
11.       <p:commandButton id="resa-precedent" icon="ui-icon-seek-prev"
    actionListener="#{form.getPreviousAgenda}" update=":formulaire:contenu"/>
12.       <p:tooltip for="resa-precedent" value="#{msg['form1.precedent']}"/>
13.       <p:commandButton id="resa-suivant" icon="ui-icon-seek-next"
    actionListener="#{form.getNextAgenda}" update=":formulaire:contenu"/>
14.       <p:tooltip for="resa-suivant" value="#{msg['form1.suivant']}"/>
15.       <p:commandButton id="resa-today" icon="ui-icon-home" actionListener="#{form.today}"
    update=":formulaire:contenu"/>
16.       <p:tooltip for="resa-today" value="#{msg['form1.today']}"/>
17.     </p:toolbarGroup>
18.     <p:toolbarGroup align="right">
19.       ...
20.     </p:toolbarGroup>
21. </p:toolbar>
```

Les méthode *[Form].getPreviousAgenda*, *[Form].getNextAgenda*, *[Form].today* sont les suivantes :

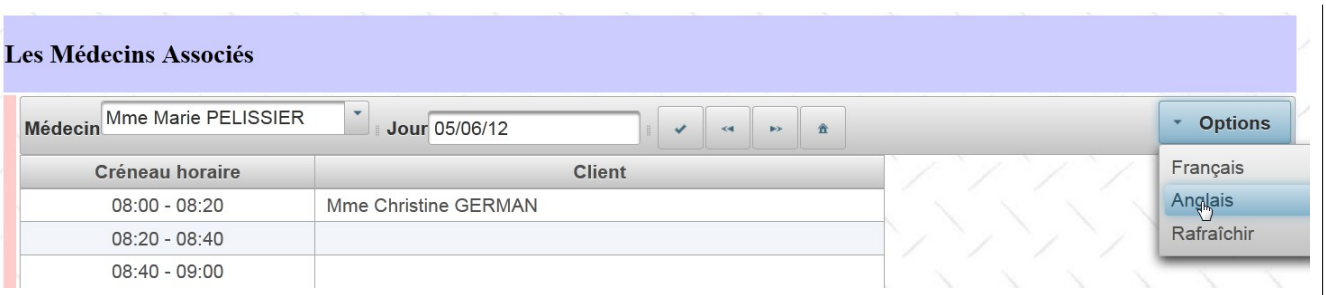
```
1.  private Date jour = new Date();
2.
3.  public void getPreviousAgenda() {
4.      // on passe au jour précédent
5.      Calendar cal = Calendar.getInstance();
6.      cal.setTime(jour);
7.      cal.add(Calendar.DAY_OF_YEAR, -1);
8.      jour = cal.getTime();
9.      // agenda
10.     if (form2Rendered) {
11.       getAgenda();
12.     }
13.   }
14.
15.   public void getNextAgenda() {
16.     // on passe au jour suivant
17.     Calendar cal = Calendar.getInstance();
18.     cal.setTime(jour);
19.     cal.add(Calendar.DAY_OF_YEAR, 1);
20.     jour = cal.getTime();
21.     // agenda
22.     if (form2Rendered) {
23.       getAgenda();
24.     }
25.   }
26.
27.   // agenda aujourd'hui
28.   public void today() {
29.     jour = new Date();
30.     // agenda
31.     if (form2Rendered) {
32.       getAgenda();
33.     }
34. }
```

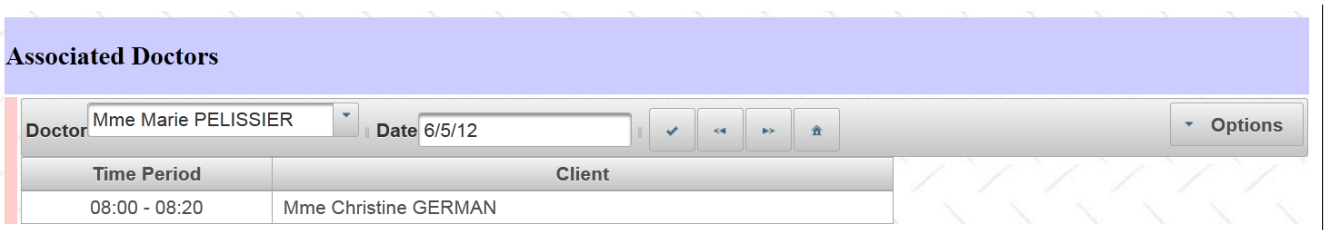
- ligne 1 : le jour d'affichage de l'agenda,
- ligne 5 : on utilise un calendrier,
- ligne 6 : qu'on initialise au jour actuel de l'agenda,
- ligne 7 : on retire un jour au calendrier,
- ligne 8 : et on rénitialise avec, le jour d'affichage de l'agenda,
- ligne 11 : on réaffiche l'agenda si celui-ci est affiché. En effet, l'utilisateur peut utiliser la barre d'outils sans que l'agenda soit affiché.

Les autres méthodes sont analogues.

6.15 Changement de langue d'affichage

Le changement de langue est géré par le bouton menu de la barre d'outils :





Les balises du bouton menu sont les suivantes :

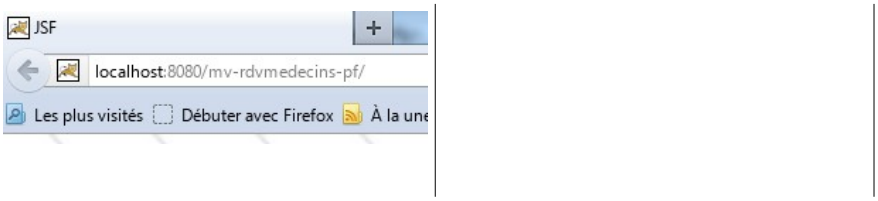
```
<p:toolbar>
    <p:toolbarGroup align="left">
...
    </p:toolbarGroup>
    <p:toolbarGroup align="right">
      <p:menuButton value="#{msg['form1.options']}">
        <p:menuitem id="menuitem-francais" value="#{msg['form1.francais']}" actionListener="#{form.setFrenchLocale}" update=":formulaire"/>
        <p:menuitem id="menuitem-anglais" value="#{msg['form1.anglais']}" actionListener="#{form.setEnglishLocale}" update=":formulaire"/>
        <p:menuitem id="menuitem-rafraichir" value="#{msg['form1.rafraichir']}" actionListener="#{form.refresh}" update=":formulaire:contenu"/>
      </p:menuButton>
    </p:toolbarGroup>
  </p:toolbar>
```

Les méthodes exécutées dans le modèle sont les suivantes :

```
private String locale = "fr";

  public void setFrenchLocale() {
    locale = "fr";
    // on recharge la page
    redirect();
  }

  public void setEnglishLocale() {
    locale = "en";
    // on recharge la page
    redirect();
  }

  private void redirect() {
    // on redirige le client vers la servlet
    ExternalContext ctx = FacesContext.getCurrentInstance().getExternalContext();
    try {
      ctx.redirect(ctx.getRequestContextPath());
    } catch (IOException ex) {
      Logger.getLogger(Form.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Les méthodes des lignes 3 et 9 se contentent d'initialiser le champ *locale* de la ligne 1 puis redirigent le navigateur client vers la même page. Une redirection est une réponse dans laquelle le serveur demande au navigateur de charger une autre page. Le navigateur fait alors un GET vers cette nouvelle page.

- ligne 17 : [ExternalContext] est une classe JSF qui permet d'accéder à la servlet en cours d'exécution,
- ligne 19 : on fait la redirection. Le paramètre de la méthode *redirect* est l'URL de la page vers laquelle le navigateur client doit se rediriger. Ici nous voulons nous rediriger vers [/mv-rdvmedecins-pf] qui est le nom de notre application :



la méthode [getRequestContextPath] permet d'avoir ce nom. Il va donc y avoir chargement de la page d'accueil [index.xhtml] de notre application. Cette page est associée au modèle [Form] de portée **session**. Ce modèle gère trois booléens qui commandent l'apparence de la page [index.xhtml] :

```
1.    private Boolean form1Rendered = true;
2.  private Boolean form2Rendered = false;
3.  private Boolean erreurRendered = false;
```

Comme le modèle est de portée **session**, ces trois booléens ont conservé leurs valeurs. La page [index.xhtml] va donc apparaître dans l'état où elle était avant la redirection. Cette page est mise en forme avec le template facelet [layout.xhtml] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.
4.  <html xmlns="http://www.w3.org/1999/xhtml"
5.        xmlns:h="http://java.sun.com/jsf/html"
6.        xmlns:p="http://primefaces.org/ui"
7.        xmlns:f="http://java.sun.com/jsf/core"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <f:view locale="#{form.locale}">
10.     ....
11.   </f:view>
12. </html>
```

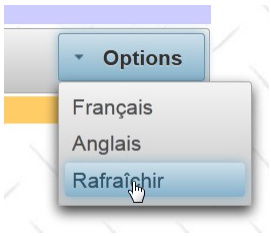
La balise de la ligne 9 fixe la langue d'affichage de la page avec son attribut **locale**. La page va donc passer en français ou en anglais selon les cas. Maintenant pourquoi une redirection ? Revenons aux balises des options de changement de langues :

```
1.  <p:toolbar>
2.      <p:toolbarGroup align="left">
3.  ...
4.      </p:toolbarGroup>
5.      <p:toolbarGroup align="right">
6.        <p:menuButton value="#{msg['form1.options']}">
7.          <p:menuitem id="menuitem-francais" value="#{msg['form1.francais']}"
    actionListener="#{form.setFrenchLocale}" update=":formulaire"/>
8.          <p:menuitem id="menuitem-anglais" value="#{msg['form1.anglais']}"
    actionListener="#{form.setEnglishLocale}" update=":formulaire"/>
9.          <p:menuitem id="menuitem-rafraichir" value="#{msg['form1.rafraichir']}"
    actionListener="#{form.refresh}" update=":formulaire:contenu"/>
10.       </p:menuButton>
11.     </p:toolbarGroup>
12. </p:toolbar>
```

Elles avaient été écrites initialement pour mettre à jour par un appel AJAX la zone d'id **formulaire** (attribut **update** des lignes 7 et 8). Mais aux tests, le changement de langue ne fonctionnait pas à tout coup. D'où la redirection pour résoudre ce problème. On aurait peut-être pu également mettre l'attribut **ajax='false'** aux balises pour provoquer un rechargement de page. Cela aurait alors évité la redirection.

6.16 Rafraîchissement des listes

Cela correspond à l'action suivante :



La balise associée à l'option [Rafraîchir] est la suivante :

```
<p:menuitem id="menuitem-rafraichir" value="#{msg['form1.rafraichir']}" actionListener="#{form.refresh}"
update=":formulaire:contenu"/>
```

La méthode *[Form].refresh* est la suivante :

```
1.    public void refresh() {
2.      // on rafraîchit les listes
3.      init();
4.  }
```

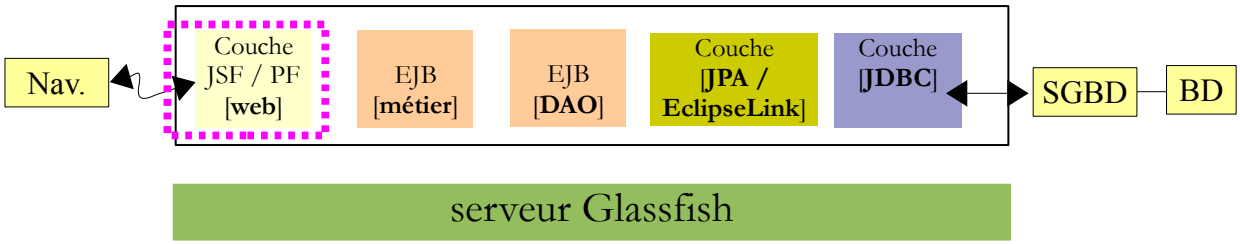
La méthode *init* est la méthode exécutée juste après la construction du bean [Form]. Elle a pour objet de mettre en cache des données de la base de données dans le modèle :

```
1.  // bean Application
2.    @Inject
3.    private Application application;
4.    // cache de la session
5.    private List<Medecin> medecins;
6.    private List<Client> clients;
7.    private Map<Long, Medecin> hMedecins = new HashMap<Long, Medecin>();
8.    private Map<Long, Client> hClients = new HashMap<Long, Client>();
9.    private Map<String, Client> hIdentitesClients = new HashMap<String, Client>();
10.   ...
11.
12.   @PostConstruct
13.   private void init() {
14.     // on met les médecins et les clients en cache
15.     try {
16.       medecins = application.getMetier().getAllMedecins();
17.       clients = application.getMetier().getAllClients();
18.     } catch (Throwable th) {
19.       ...
20.     }
21.     ...
22.     // les dictionnaires
23.     for (Medecin m : medecins) {
24.       hMedecins.put(m.getId(), m);
25.     }
26.     for (Client c : clients) {
27.       hClients.put(c.getId(), c);
28.       hIdentitesClients.put(identite(c), c);
29.     }
30.   }
```

La méthode *init* construit les listes et dictionnaires des lignes 5-9. L'inconvénient de cette technique est que ces éléments ne prennent plus en compte les changements en base de données (ajout d'un client, d'un médecin, ...). La méthode *refresh* force la reconstruction de ces listes et dictionnaires. Aussi l'utilisera-t-on à chaque fois qu'un changement en base est fait, l'ajout d'un nouveau client par exemple.

6.17 Conclusion

Rappelons l'architecture de l'application que nous venons de construire :



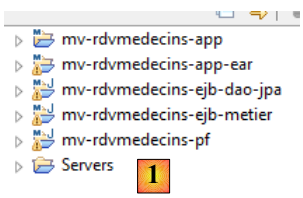
Nous nous sommes largement appuyés sur la version JSF2 déjà construite :

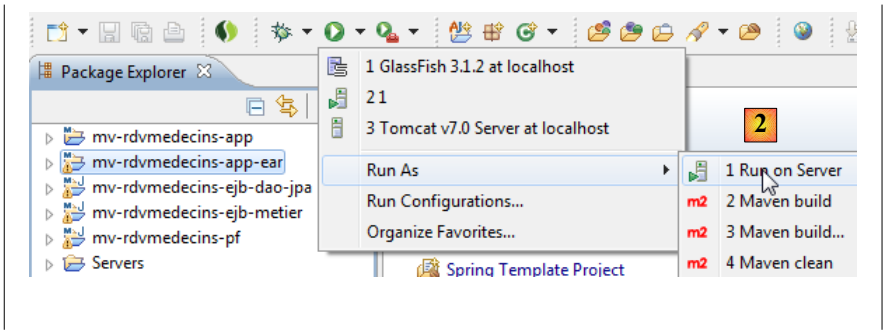
- les couches [métier], [DAO], [JPA] ont été conservées,
- les beans [Application] et [Form] de la couche web ont été conservés mais de nouvelles fonctionnalités leur ont été ajoutées à cause de l'enrichissement de l'interface utilisateur,
- l'interface utilisateur a été profondément modifiée. Elle est notamment plus riche en fonctionnalités et plus conviviale.

Le passage de JSF à Primefaces pour construire l'interface web nécessite une certaine expérience car au départ on est un peu noyés devant le grand nombre de composants utilisables et au final on ne sait trop lesquels utiliser. Il faut alors se pencher sur l'ergonomie souhaitée pour l'interface.

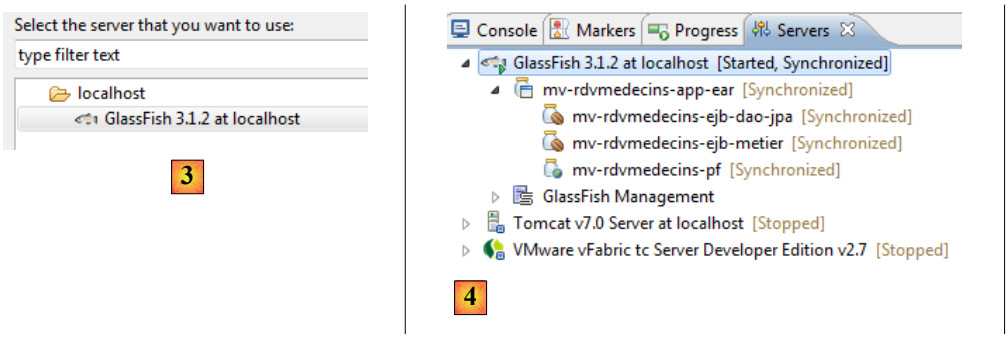
6.18 Tests Eclipse

Comme nous l'avons fait pour les précédentes versions de l'application exemple, nous montrons comment tester cette version 03 avec Eclipse. Tout d'abord, nous importons dans Eclipse les projets Maven de l'exemple 03 [1] :

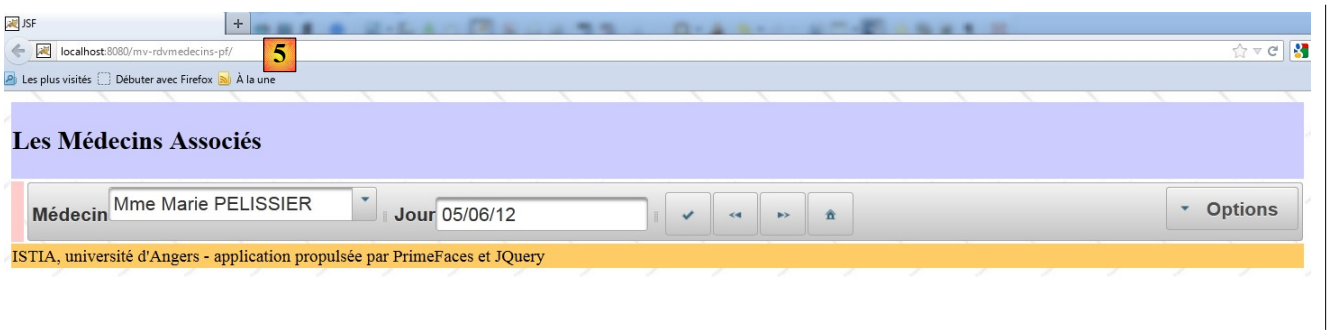




- [mv-rdvmedecins-ejb-dao-jpa] : les couches [DAO] et [JPA],
- [mv-rdvmedecins-ejb-metier] : la couche [métier],
- [mv-rdvmedecins-pf] : la couche [web] implémentée avec JSF et Primefaces,
- [mv-rdvmedecins-app] : le parent du projet d'entreprise [mv-rdvmedecins-app-ear]. Lorsqu'on importe le projet parent, le projet fils est automatiquement importé,
- en [2], on exécute le projet d'entreprise [mv-rdvmedecins-app-ear],



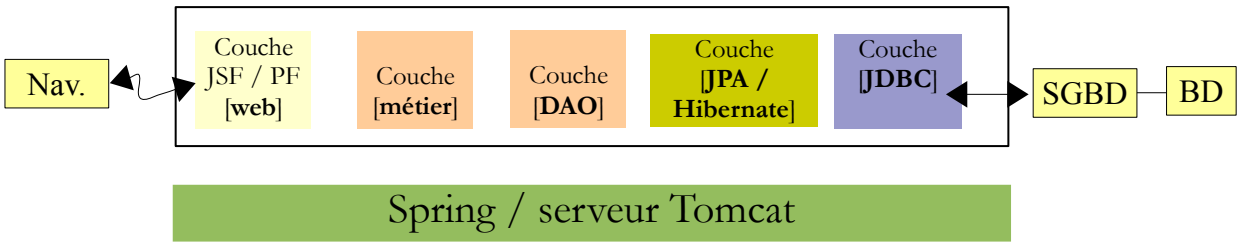
- en [3], on choisit le serveur Glassfish,
- en [4], dans l'onglet [Servers], l'application a été déployée. Elle ne s'exécute pas d'elle-même. Il faut demander son URL [http://localhost:8080/mv-rdvmedecins-pf/] dans un navigateur [5] :



7 Application exemple-04 : rdvmedecins-pf-spring

7.1 Le portage

Nous portons maintenant l'application précédente dans un environnement Spring / Tomcat :

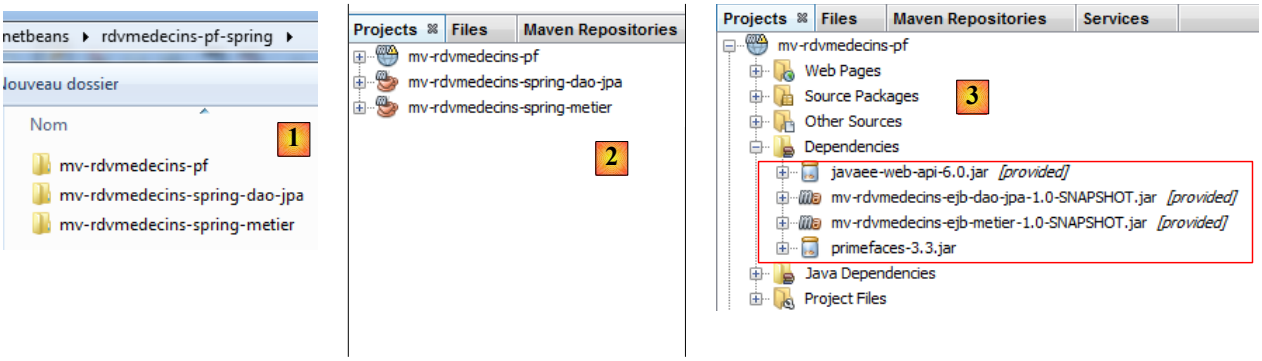


Nous allons nous appuyer sur deux applications déjà écrites. Nous allons utiliser :

- les couches [DAO] et [JPA] de la version 02 JSF / Spring,
- la couche [web] / Primefaces de la version 03 PF / EJB,
- les fichiers de configuration de Spring de la version 02

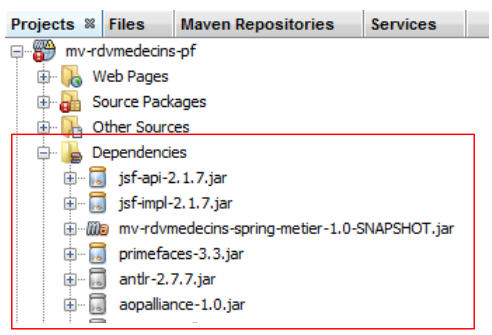
Nous refaisons là un travail analogue à celui qui avait été fait pour porter l'application JSF2 / EJB / Glassfish dans un environnement JSF2 / Spring Tomcat. Aussi donnerons-nous moins d'explications. Le lecteur peut, s'il en est besoin, se reporter à ce portage.

Nous mettons tous les projets nécessaires au portage dans un nouveau dossier [rdvmedecins-pf-spring] [1] :



- [mv-rdvmedecins-spring-dao-jpa] : les couches [DAO] et [JPA] de la version 02 JSF / Spring,
- [mv-rdvmedecins-spring-metier] : la couche [métier] de la version 02 JSF / Spring,
- [mv-rdvmedecins-pf] : la couche [web] de la version 03 Primefaces / EJB,
- en [2], nous les chargeons dans Netbeans,
- en [3], les dépendances du projet web ne sont plus correctes :
    - les dépendances sur les couches [DAO], [JPA], [métier] doivent être changées pour cibler désormais les projets Spring ;
    - le serveur Glassfish fournissait les bibliothèques de JSF. Ce n'est plus le cas avec le serveur Tomcat. Il faut donc les ajouter aux dépendances.

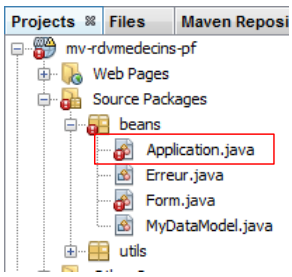
Le projet [web] évolue comme suit :

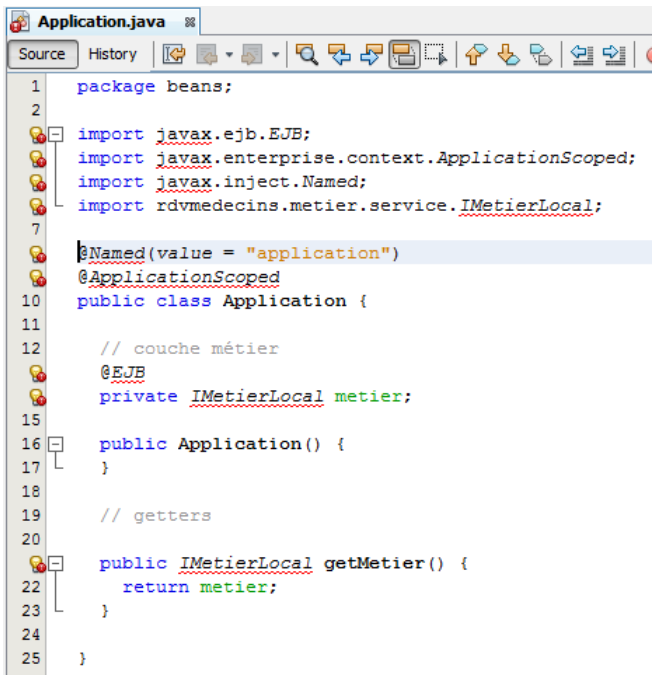


Le fichier [pom.xml] de la couche [web] a désormais les dépendances suivantes :

```
<dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>mv-rdvmedecins-spring-metier</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>2.1.7</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>2.1.7</version>
    </dependency>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>primefaces</artifactId>
      <version>3.3</version>
    </dependency>
  </dependencies>
```

Des erreurs apparaissent. Elles sont dues aux références EJB de la couche [web]. Etudions d'abord le bean [Application] :

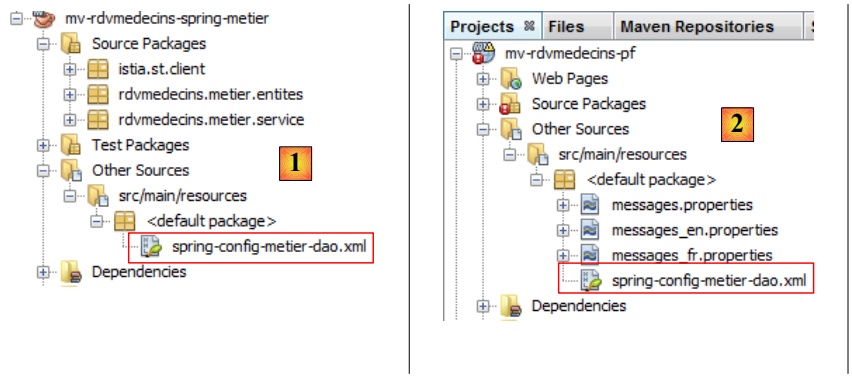




Nous enlevons toutes les lignes erronées à cause de paquetages manquants, nous renommons [IMetier] (c'est son nom dans la couche [métier] Spring) l'interface [IMetierLocal] et nous utilisons Spring pour l'instancier :

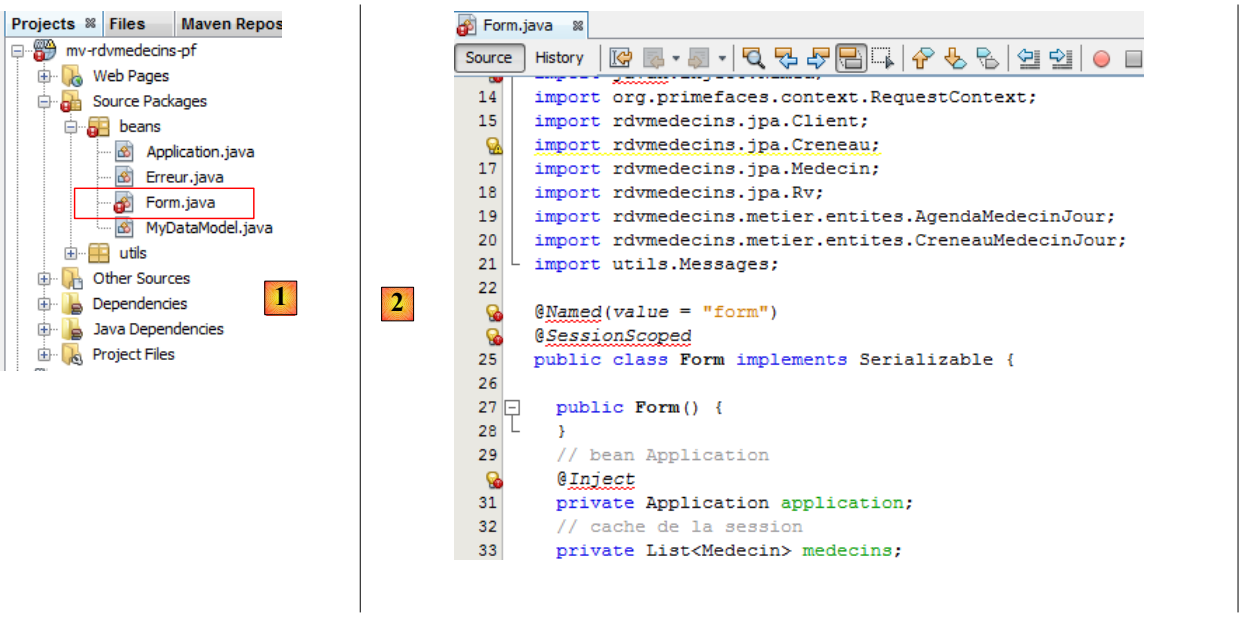
```
1.  package beans;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import org.springframework.context.ApplicationContext;
6.  import org.springframework.context.support.ClassPathXmlApplicationContext;
7.  import rdvmedecins.metier.service.IMetier;
8.
9.  public class Application {
10.
11.   // couche métier
12.   private IMetier metier;
13.   // erreurs
14.   private List<Erreur> erreurs = new ArrayList<Erreur>();
15.   private Boolean erreur = false;
16.
17.   public Application() {
18.     try {
19.       // instanciation couche [métier]
20.       ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-metier-dao.xml");
21.       metier = (IMetier) ctx.getBean("metier");
22.     } catch (Throwable th) {
23.       // on note l'erreur
24.       erreur = true;
25.       erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
26.       while (th.getCause() != null) {
27.         th = th.getCause();
28.         erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
29.       }
30.       return;
31.     }
32.   }
33.
34.   // getters
35.   public Boolean getErreur() {
36.     return erreur;
37.   }
38.
39.   public List<Erreur> getErreurs() {
40.     return erreurs;
41.   }
42.
43.   public IMetier getMetier() {
44.     return metier;
45.   }
46. }
```

- lignes 20-21 : instanciation de la couche [métier] à partir du fichier de configuration de Spring. Ce fichier est celui utilisé par la couche [métier] [1]. Nous le copions dans le projet web [2] :

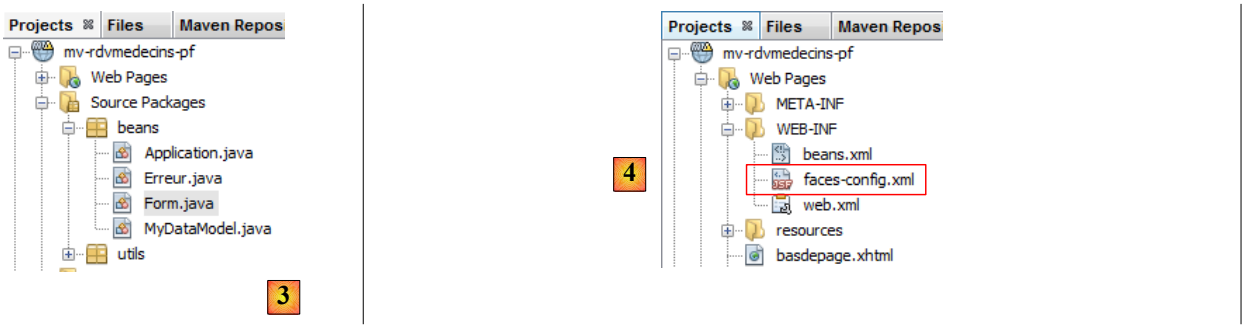


- lignes 22-31 : nous gérons une éventuelle exception et mémorisons sa pile.

Ceci fait, le bean [Application] n'a plus d'erreurs. Regardons maintenant le bean [Form] [1], [2] :



Nous supprimons toutes les lignes erronées (import et annotations) à cause de paquetages manquants. Cela suffit pour éliminer toutes les erreurs [3].



Par ailleurs, il faut ajouter dans le code du bean [Form], le getter et le setter du champ

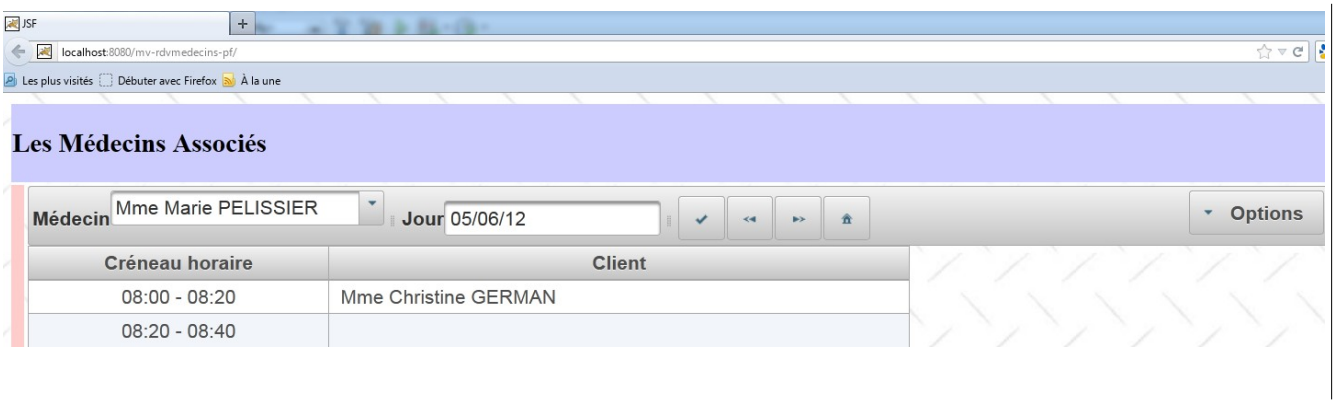
```
1.    // bean Application
2.  private Application application;
```

Certaines des annotations supprimées dans les beans [Application] et [Form] déclaraient les classes comme étant des beans avec une certaine portée. Désormais, cette configuration est faite dans le fichier [faces-config.xml] [4] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <!-- le fichier des messages -->
12.     <resource-bundle>
13.       <base-name>
14.         messages
15.       </base-name>
16.       <var>msg</var>
17.     </resource-bundle>
18.     <message-bundle>messages</message-bundle>
19.   </application>
```

```
20.     <!-- le bean applicationBean -->
21.   <managed-bean>
22.     <managed-bean-name>applicationBean</managed-bean-name>
23.     <managed-bean-class>beans.Application</managed-bean-class>
24.     <managed-bean-scope>application</managed-bean-scope>
25.   </managed-bean>
26.     <!-- le bean form -->
27.   <managed-bean>
28.     <managed-bean-name>form</managed-bean-name>
29.     <managed-bean-class>beans.Form</managed-bean-class>
30.     <managed-bean-scope>session</managed-bean-scope>
31.     <managed-property>
32.       <property-name>application</property-name>
33.       <value>#{applicationBean}</value>
34.     </managed-property>
35.   </managed-bean>
36. </faces-config>
```

Normalement, le portage est terminé. Il nous restera cependant quelques détails à régler. Nous pouvons tenter d'exécuter l'application web.



Nous laissons le lecteur tester cette nouvelle application. Nous pouvons l'améliorer légèrement pour gérer le cas où l'initialisation du bean [Application] a échoué. On sait que dans ce cas, les champs suivants ont été initialisés :

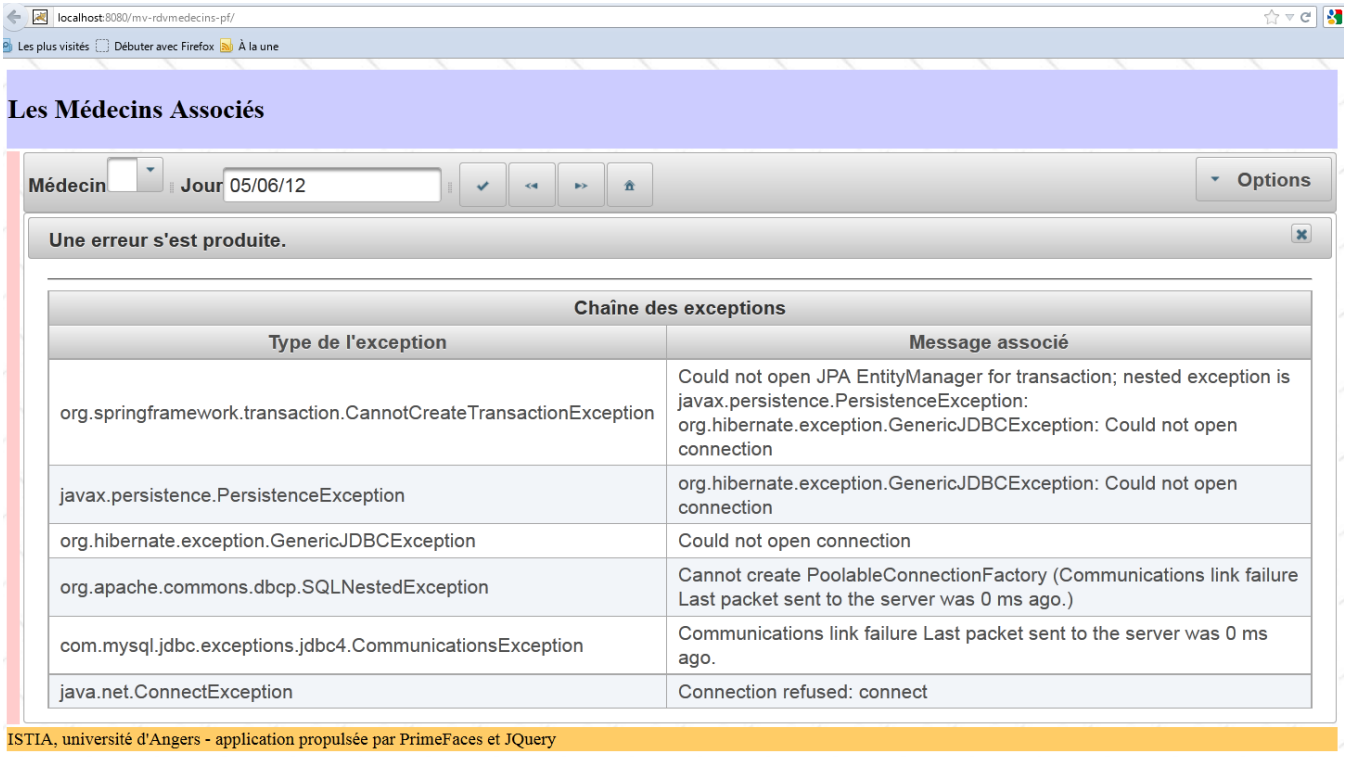
```
1.    // erreurs
2.    private List<Erreur> erreurs = new ArrayList<Erreur>();
3.  private Boolean erreur = false;
```

Ce cas peut être prévu dans la méthode *init* du bean [Form] :

```
1.  @PostConstruct
2.    private void init() {
3.
4.      // l'initialisation s'est-elle bien passée ?
5.      if (application.getErreur()) {
6.        // on récupère la liste des erreurs
7.        erreurs = application.getErreurs();
8.        // la vue des erreurs est affichée
9.        setForms(false, false, true);
10.     }
11.
12.     // on met les médecins et les clients en cache
13.     ...
14.   }
```

- ligne 5 : si le bean [Application] s'est mal initialisé,
- ligne 7 : on récupère la liste des erreurs,
- ligne 9 : et on affiche la page d'erreur.

Ainsi, si on arrête le SGBD MySQL et qu'on relance l'application, on reçoit maintenant la page suivante :

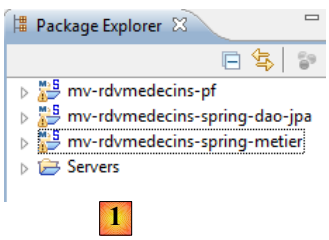


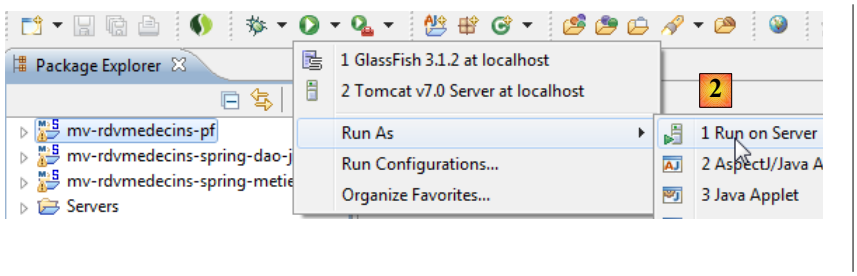
7.2 Conclusion

Le portage de l'application Primefaces / EJB / Glassfish vers un environnement Primefaces / Spring / Tomcat s'est révélé simple. Le problème de la fuite mémoire signalée dans l'étude de l'application JSF / Spring / Tomcat (paragraphe 4.3.5, page 271) demeure. On le résoudra de la même façon.

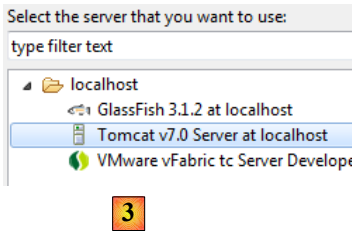
7.3 Tests avec Eclipse

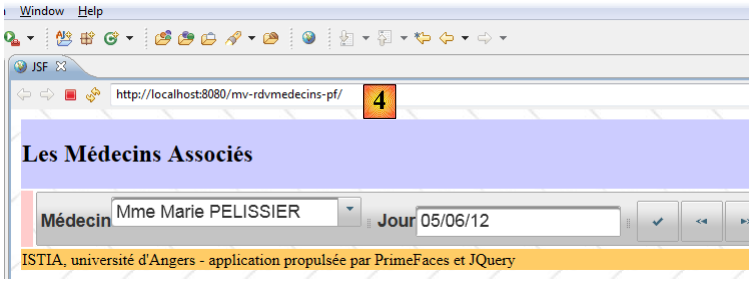
Nous importons les projets Maven dans Eclipse [1] :





Nous exécutons le projet web [2].



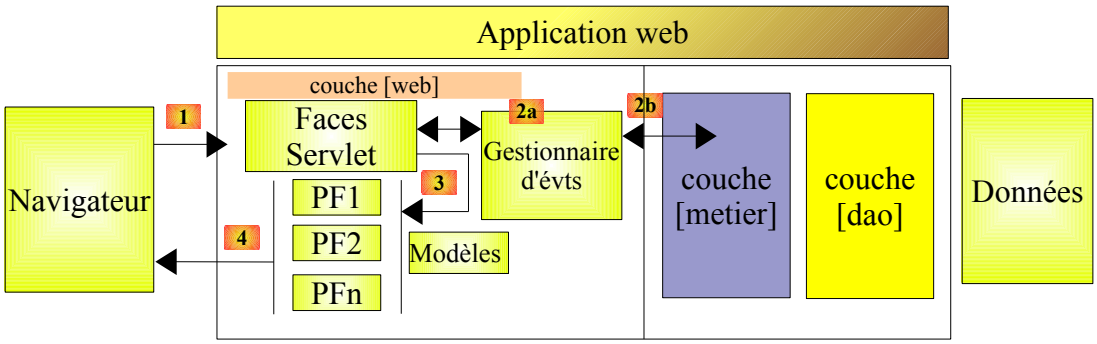


Nous choisissons le serveur Tomcat [3]. La page d'accueil de l'application est alors affichée dans le navigateur interne d'Eclipse [4].

8 Introduction à Primefaces mobile

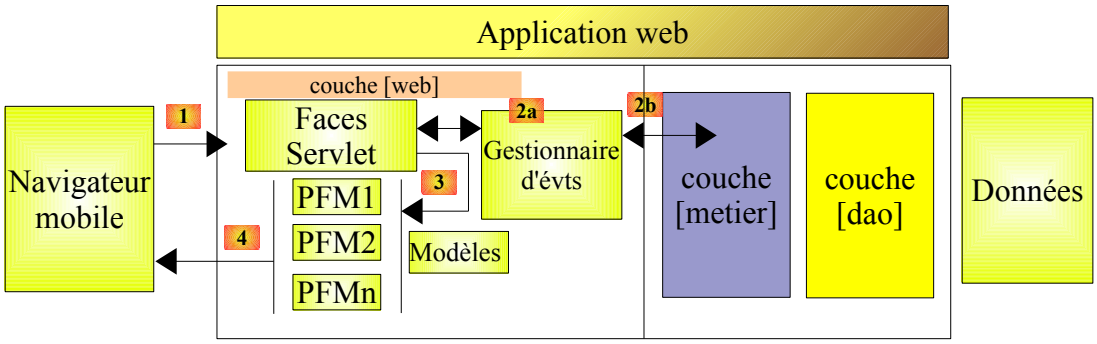
8.1 La place de Primefaces mobile dans une application JSF

Revenons à l'architecture d'une application Primefaces telle que nous l'avons étudiée au début de ce document :



Lorsque la cible est le navigateur d'un téléphone mobile, il nous faut changer la présentation. En effet, la taille de l'écran d'un smartphone doit être prise en compte et l'ergonomie des pages repensée. Il existe des bibliothèques spécialisées dans la construction de pages web pour mobiles. C'est le cas de la bibliothèque **Primefaces mobile** qui s'appuie elle-même sur la bibliothèque **JQuery mobile**.

L'architecture d'une application web pour mobiles sera identique à la précédente si ce n'est qu'en plus de JSF et Primefaces, nous utiliserons la bibliothèque Primefaces mobile (PFM).



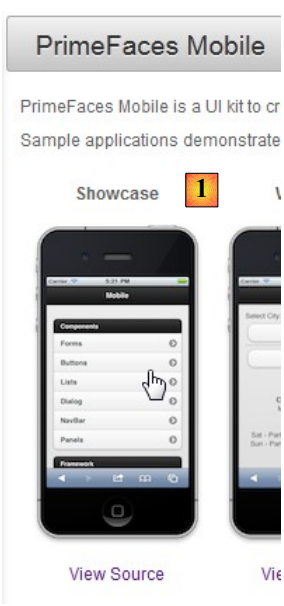
Nous allons retrouver tous les concepts étudié avec JSF et Primefaces. Seule la couche [présentation] (les pages principalement) va changer.

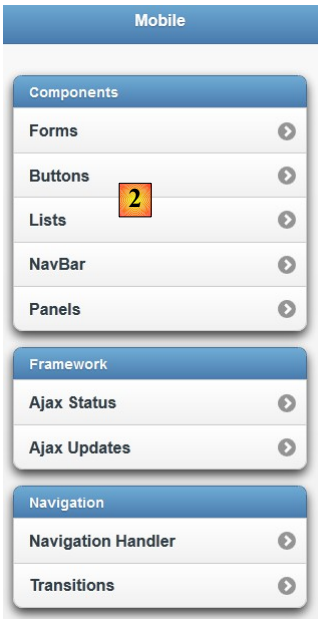
8.2 Les apports de Primefaces mobile

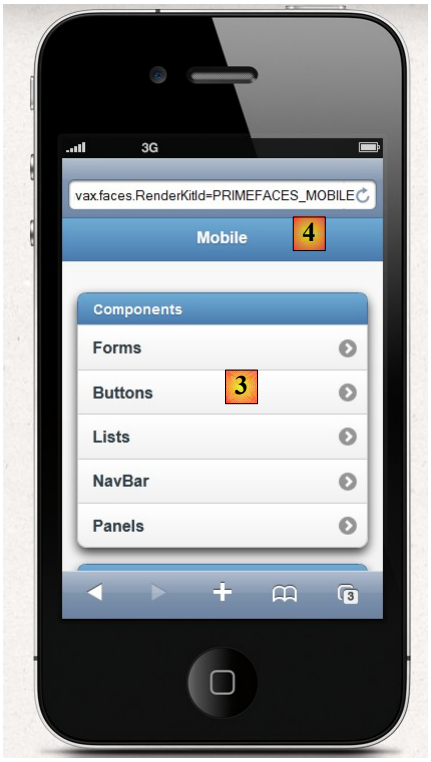
Le site de Primefaces mobile

```
[http://www.primefaces.org/showcase-labs/mobile/showcase.jsf?javax.faces.RenderKitId=PRIMEFACES_MOBILE]
```

donne la liste des composants utilisables dans une page PFM [1, 2] :







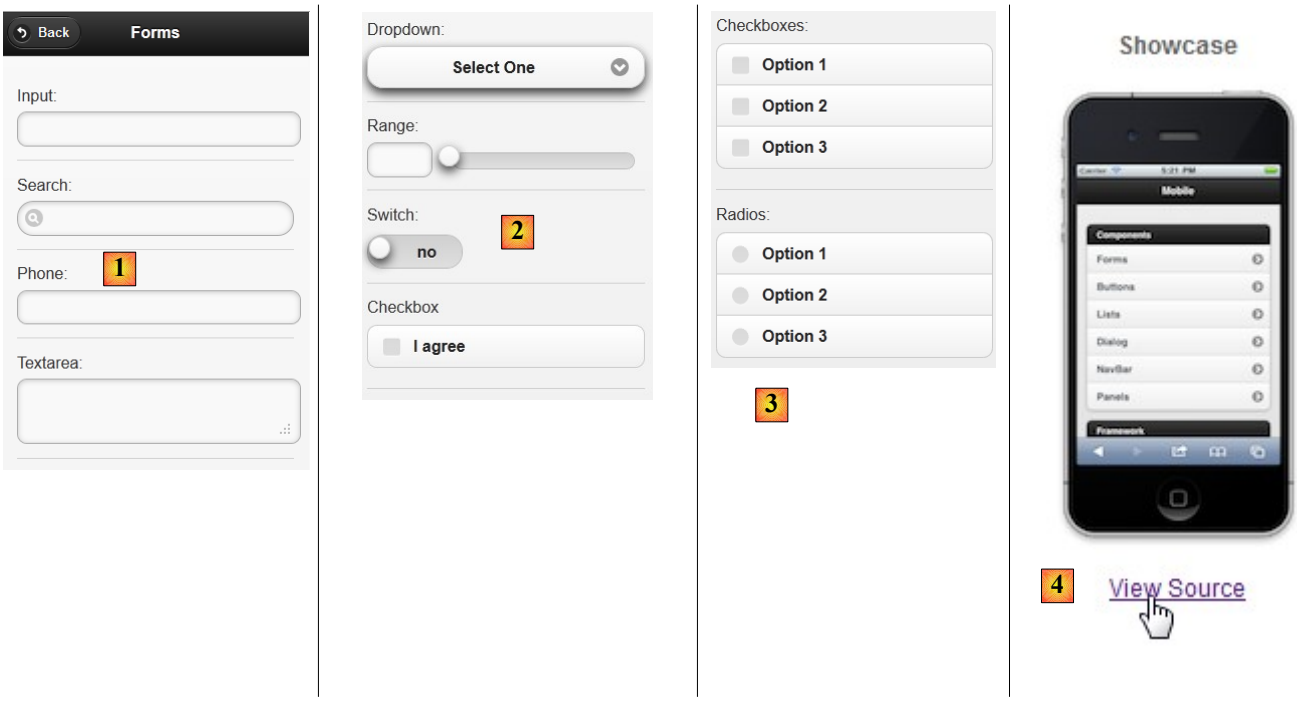
Il est possible d'utiliser des simulateurs de smartphone pour visualiser l'application de démo ci-dessus. L'un d'eux est disponible pour l'iphone 4 à l'URL [http://iphone4simulator.com/] [3]. On entre l'adresse de l'application à tester en [4] et celle-ci est affichée aux dimensions de l'iphone. De tels simulateurs sont utiles en phase de développement mais au final, l'application doit être testée en réel sur différents mobiles.

8.3 Apprentissage de Primefaces mobile

Primefaces mobile offre beaucoup moins de composants que Primefaces. Le site de Primefaces mobile

```
[http://www.primefaces.org/showcase-labs/mobile/showcase.jsf?javax.faces.RenderKitId=PRIMEFACES_MOBILE]
```

en donne la liste. Voici par exemple les composants utilisables dans un formulaire [1], [2], [3] :



Il est possible de télécharger le code source des exemples [4]. Ainsi nous découvrons que la page PFM qui affiche les composants [1], [2], [3] ci-dessus est la suivante :

```
1.  <?xml version="1.0" encoding="windows-1250"?>
2.  <f:view xmlns="http://www.w3.org/1999/xhtml"
3.          xmlns:f="http://java.sun.com/jsf/core"
4.          xmlns:h="http://java.sun.com/jsf/html"
5.      xmlns:ui="http://java.sun.com/jsf/facelets"
6.          xmlns:p="http://primefaces.org/ui"
7.          xmlns:pm="http://primefaces.org/mobile"
8.          contentType="text/html">
9.
10.     <pm:page title="Components">
11.         <!-- Forms -->
12.         <pm:view id="forms">
13.             <pm:header title="Forms">
14.                 <f:facet name="left"><p:button value="Back" icon="back" href="#main?
    reverse=true"/></f:facet>
15.             </pm:header>
16.             <pm:content>
17.                 <p:inputText label="Input:" />
18.                 <p:inputText label="Search:" type="search"/>
19.                 <p:inputText label="Phone:" type="tel"/>
20.                 <p:inputTextarea id="inputTextarea" label="Textarea:"/>
21.                 <pm:field>
22.                     <h:outputLabel for="selectOneMenu" value="Dropdown: "/>
23.                     <h:selectOneMenu id="selectOneMenu">
24.                         <f:selectItem itemLabel="Select One" itemValue="" />
25.                         <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
26.                         <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
27.                         <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
28.                     </h:selectOneMenu>
29.                 </pm:field>
30.                 <pm:inputRange id="range" minValue="0" maxValue="100" label="Range:" />
31.                 <pm:switch id="switch" onLabel="yes" offLabel="no" label="Switch:" />
32.                 <p:selectBooleanCheckbox id="booleanCheckbox" value="false" itemLabel="I agree"
    label="Checkbox"/>
33.                 <p:selectManyCheckbox id="checkbox" label="Checkboxes: ">
34.                     <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
35.                     <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
36.                     <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
37.                 </p:selectManyCheckbox>
38.                 <p:selectOneRadio id="radio" label="Radios: ">
39.                     <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
40.                     <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
```

```
41.                     <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
42.                 </p:selectOneRadio>
43.             </pm:content>
44.         </pm:view>
45.     </pm:page>
46. </f:view>
```

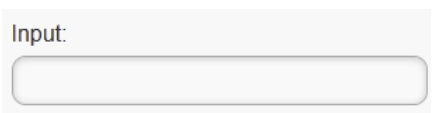
- ligne 7 : un nouvel espace de noms apparaît, celui des composants Primefaces mobile de préfixe **pm**,
- ligne 10 : définit une page. Une page va être composée de plusieurs vues (ligne 12). Une seule vue est visible à un moment donné. On retrouve là un concept que nous avons abondamment utilisé dans nos exemples Primefaces, une unique page avec plusieurs fragments qu'on affiche ou qu'on cache. Ici, nous n'aurons pas à faire ce jeu. Nous désignerons simplement la vue à afficher, les autres étant forcément cachées,
- ligne 12 : une vue,
- lignes 13-15 : définissent l'en-tête de la vue :

```
1. <pm:header title="Forms">
2.     <f:facet name="left"><p:button value="Back" icon="back" href="#main?reverse=true"/></f:facet>
3. </pm:header>
```

On notera la balise pour générer un bouton. L'attribut **href** permet de naviguer. Sa valeur ici est le nom d'une vue. Un clic sur le bouton va ramener (**reverse=true**) vers la vue nommée **main** (non représentée ici).

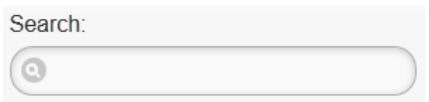
- ligne 16 : introduit le contenu de la vue,
- ligne 17 : génère une zone de saisie :

```
<p:inputText label="Input:" />
```



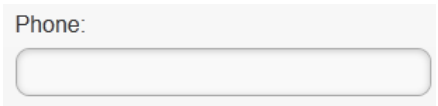
- ligne 18 : une zone de saisie avec une icône particulière :

```
<p:inputText label="Search:" type="search"/>
```



- ligne 19 : une zone de saisie pour un n° de téléphone :

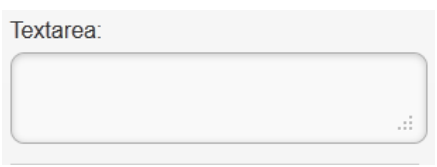
```
<p:inputText label="Phone:" type="tel"/>
```



L'attribut **type** d'une balise <**inputText**> a une signification : elle induit le type du clavier proposé à l'utilisateur pour faire sa saisie. Ainsi pour le **type='tel'** sera proposé un clavier numérique,

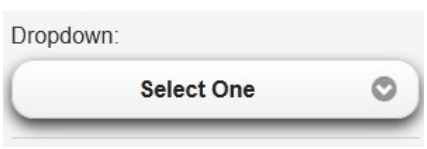
- ligne 20 : une zone de saisie multi-lignes et extensible :

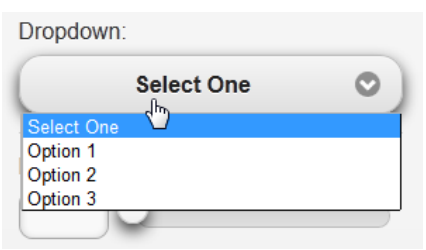
```
<p:inputTextarea id="inputTextarea" label="Textarea:"/>
```



- lignes 21-29 : un champ rassemblant plusieurs composants :

```
1.                  <pm:field>
2.                      <h:outputLabel for="selectOneMenu" value="Dropdown: "/>
3.                      <h:selectOneMenu id="selectOneMenu">
4.                          <f:selectItem itemLabel="Select One" itemValue="" />
5.                          <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
6.                          <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
7.                          <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
8.                      </h:selectOneMenu>
9.  </pm:field>
```





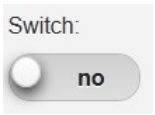
  - ligne 2 : le libellé pour le composant de la ligne 3 ;
  - ligne 3 : une liste déroulante ;
  - lignes 4-7 : son contenu,

- ligne 30 : un slider :

```
<pm:inputRange id="range" minValue="0" maxValue="100" label="Range:" />
```

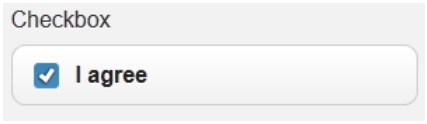
- ligne 31 : un bouton à deux valeurs :

```
<pm:switch id="switch" onLabel="yes" offLabel="no" label="Switch:" />
```



- ligne 32 : une case à cocher :

```
<p:selectBooleanCheckbox id="booleanCheckbox" value="false" itemLabel="I agree" label="Checkbox"/>
```



- lignes 33-37 : un groupe de cases à cocher :

```
<p:selectManyCheckbox id="checkbox" label="Checkboxes: ">
  <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
  <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
  <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
</p:selectManyCheckbox>
```

- lignes 38-42 : un groupe de boutons radio

```
<p:selectOneRadio id="radio" label="Radios: ">
  <f:selectItem itemLabel="Option 1" itemValue="Option 1" />
  <f:selectItem itemLabel="Option 2" itemValue="Option 2" />
  <f:selectItem itemLabel="Option 3" itemValue="Option 3" />
</p:selectOneRadio>
```

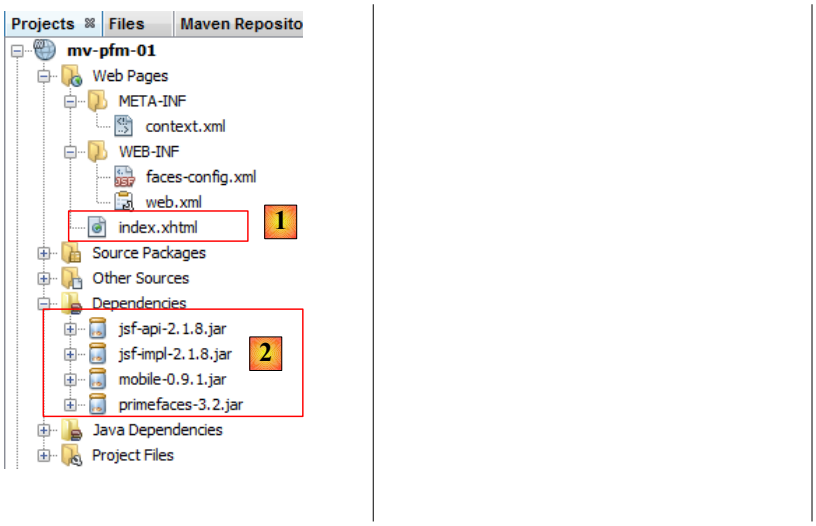
On notera que la page utilise des balises des trois bibliothèques JSF, PF et PFM. Nous en savons assez pour construire notre premier projet Primefaces mobile.

8.4 Un premier projet Primefaces mobile : mv-pfm-01

Nous allons reprendre l'exemple étudié précédemment pour l'intégrer dans un projet Maven. Nous allons ainsi découvrir les dépendances nécessaires à un projet Primefaces mobile.

8.4.1 Le projet Netbeans

Le projet Netbeans est le suivant :



Le projet Netbeans est au départ un projet web Maven de base auquel on a ajouté les dépendances [2]. Cela se traduit dans le fichier [pom.xml] par le code suivant :

```
1.  <dependency>
2.        <groupId>com.sun.faces</groupId>
3.        <artifactId>jsf-impl</artifactId>
4.        <version>2.1.8</version>
5.      </dependency>
6.      <dependency>
7.        <groupId>org.primefaces</groupId>
8.        <artifactId>primefaces</artifactId>
9.        <version>3.2</version>
10.     </dependency>
11.     <dependency>
12.       <groupId>org.primefaces</groupId>
13.       <artifactId>mobile</artifactId>
14.       <version>0.9.1</version>
15.       <type>jar</type>
16.     </dependency>
17.   </dependencies>
18.   <repositories>
19.     <repository>
20.       <url>http://download.java.net/maven/2/</url>
21.       <id>jsf20</id>
22.       <layout>default</layout>
23.       <name>Repository for library Library[jsf20]</name>
24.     </repository>
25.     <repository>
26.       <url>http://repository.primefaces.org/</url>
27.       <id>primefaces</id>
28.       <layout>default</layout>
29.       <name>Repository for library Library[primefaces]</name>
30.     </repository>
31.   </repositories>
```

En lignes 11-16, la dépendance vers Primefaces mobile. Nous avons pris ici la version 0.9.1 (ligne 14). Par ailleurs, nous avons pris la version 3.2 de Primefaces (ligne 9). Ces **numéros de version sont importants**. En effet, j'ai rencontré pas mal de problèmes avec PFM qui est une bibliothèque non finalisée. Il a ainsi existé pendant un certain temps une version 0.9.2 qui était boguée. Elle a

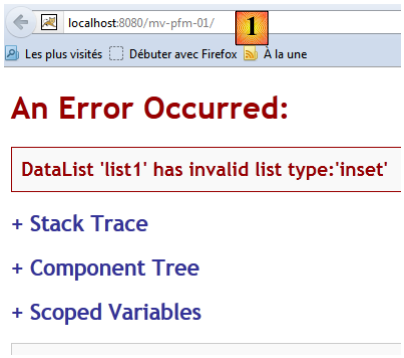
depuis été retirée. On est revenu à la version 0.9.1. Le projet PFM a donc régressé. Les tests ont également échoué avec la version 1.0-SNAPSHOT de la version 1.0 en cours de construction (début juin 2012).

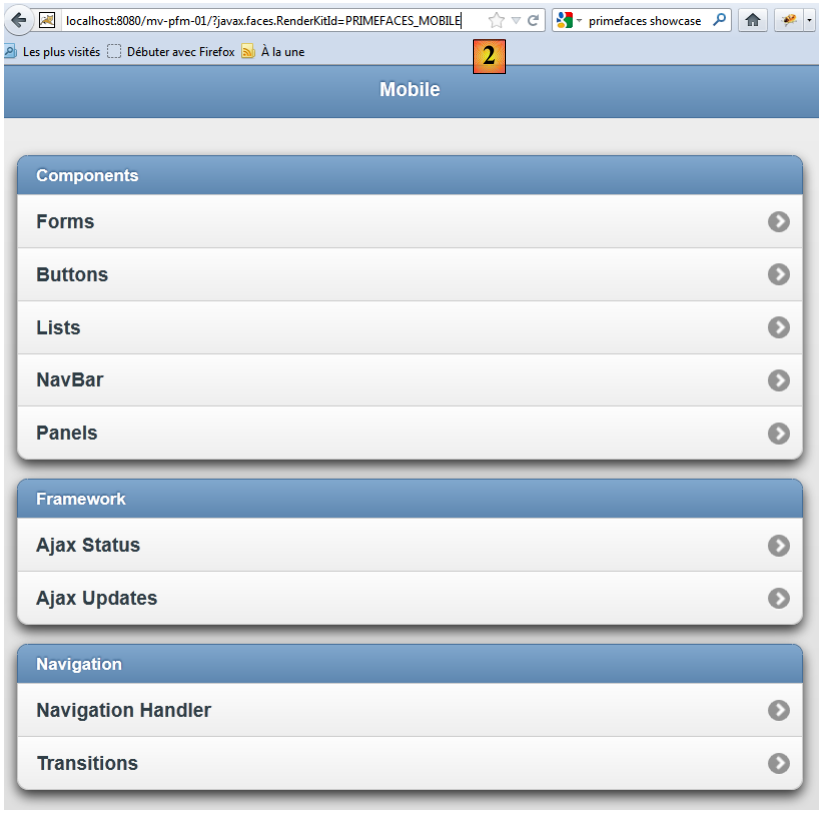
La page d'accueil [index.xhtml] [1] est la page de démonstration des composants Primefaces mobile :

```
1.  <?xml version="1.0" encoding="windows-1250"?>
2.  <f:view xmlns="http://www.w3.org/1999/xhtml"
3.          xmlns:f="http://java.sun.com/jsf/core"
4.          xmlns:h="http://java.sun.com/jsf/html"
5.          xmlns:ui="http://java.sun.com/jsf/facelets"
6.          xmlns:p="http://primefaces.org/ui"
7.          xmlns:pm="http://primefaces.org/mobile"
8.          contentType="text/html">
9.
10.   <pm:page title="Components">
11.
12.     <!-- Main View -->
13.     <pm:view id="main">
14.       ...
15.     </pm:view>
16.
17.     <!-- Forms -->
18.     <pm:view id="forms">
19.       ...
20.     </pm:view>
21.
22.
23.     <!-- Buttons -->
24.     <pm:view id="buttons">
25.       ...
26.     </pm:view>
27.
28.     ...
29.   </pm:page>
30. </f:view>
```

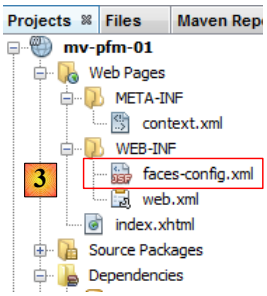
Il y a une page unique (ligne 10) composée de douze vues. Nous ne détaillons pas celles-ci. Le but ici est de simplement montrer comment construire un projet Primefaces mobile.

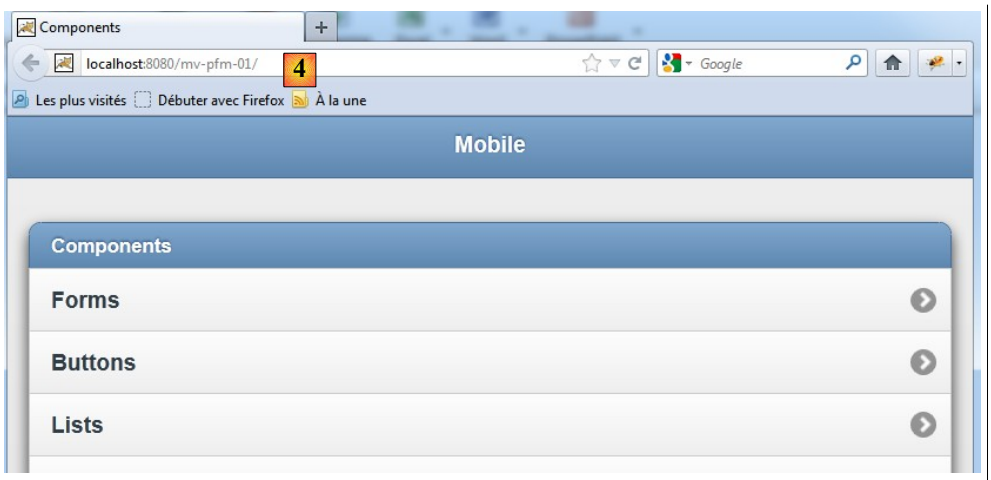
Exécutons ce projet. On obtient une page d'erreur [1] :





La raison en est qu'une application Primefaces mobile doit être appelée avec le paramètre [**?javax.faces.RenderKitId=PRIMEFACES_MOBILE**] comme en [2]. Il est possible de se passer de ce paramètre en modifiant le fichier [faces-config.xml] (ou en le créant s'il n'existe pas comme ici) [3] :





Son contenu est le suivant :

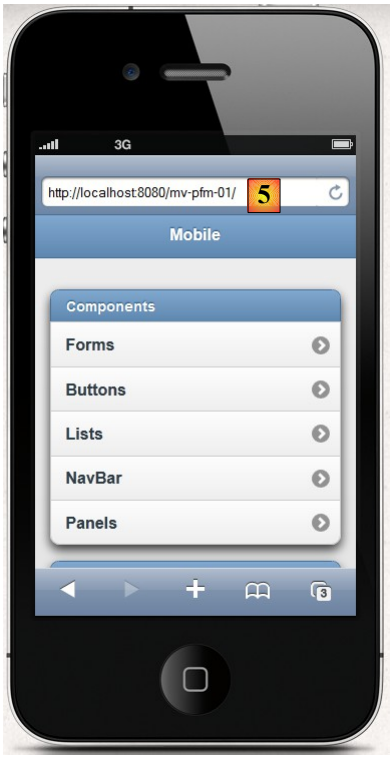
```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
```

```
9.
10.    <application>
11.      <default-render-kit-id>PRIMEFACES_MOBILE</default-render-kit-id>
12.    </application>
13. </faces-config>
```

- ligne 11 : fixe la valeur du paramètre [**javax.faces.RenderKitId**].

A l'exécution, on obtient la même page que précédemment mais sans avoir à mettre de paramètre à l'URL [4]. Nous laissons au lecteur le soin de tester cette application. On y découvre des bogues : la balise <*p:selectManyCheckbox*> ne s'affiche pas, idem pour la balise <*p:selectOneRadio*>, la balise <*pm:range*> ne fonctionne pas. Ces dysfonctionnements, qui n'existent pas sur le site de la démo, laissent à penser que celle-ci ne fonctionne pas avec la combinaisons utilisée ici de Primefaces mobile 0.9.1 avec Primefaces 3.2. Mais c'est néanmoins un point de départ pour découvrir les balises de cette bibliothèque en cours de construction.

Pour avoir un rendu plus réaliste, on pourra tester l'application sur un simulateur comme celui de l'iphone 4 [http://iphone4simulator.com/]. On obtient la chose suivante :



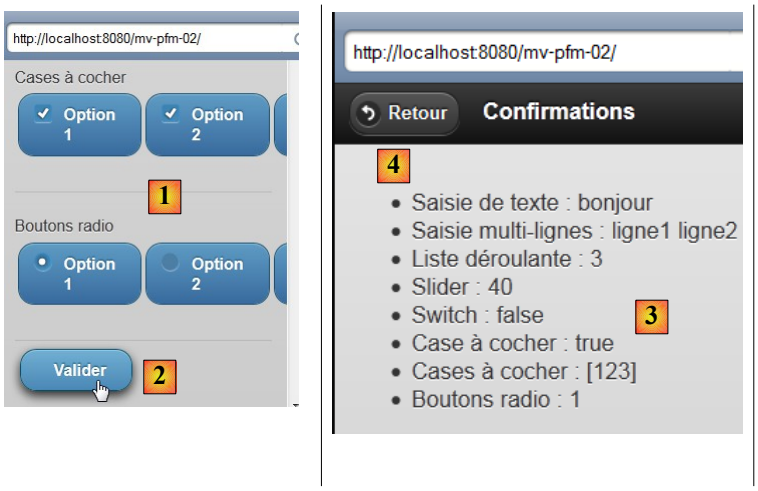
On rentre en [5], la même URL utilisée par le navigateur précédent en [4].

8.5 Exemple mv-pfm-02 : formulaire et modèle

Dans ce projet, nous montrons les interactions entre une page Primefaces mobile et son modèle. Celles-ci suivent les règles du framework JSF sur lequel repose au final Primefaces mobile.

8.5.1 Les vues

Il y a deux vues dans le projet



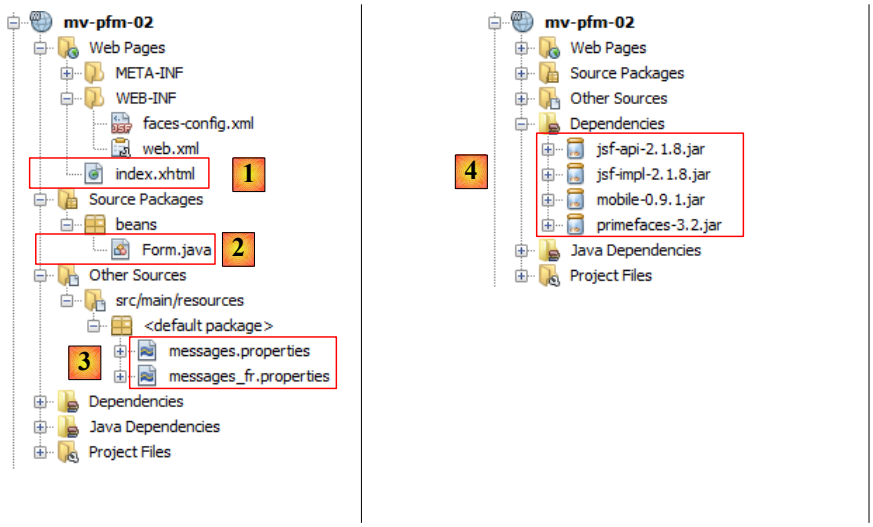
- la vue 1 [1] présente un formulaire qu'on peut valider [2],
- la vue 2 [3] confirme les saisies et offre la posibilité de revenir au formulaire [4].

Le projet montre deux choses :

- la relation entre une page et son modèle,
- la navigation entre vues de la page.

8.5.2 Le projet Netbeans

Le projet Netbeans est le suivant :



- en [1], la page Primefaces mobile,
- en [2], son modèle
- en [3], les fichiers de messages car l'application est internationalisée,
- en [4], les dépendances du projet.

8.5.3 Configuration du projet

Nous donnons les fichiers de configuration sans les expliquer. Ils sont désormais classiques.

[web.xml] : configure l'application web :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

[faces-config.xml] : configure l'application JSF :

```xml
<?xml version='1.0' encoding='UTF-8'?>

<!-- ========== FULL CONFIGURATION FILE ================================== -->

<faces-config version="2.0"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
facesconfig_2_0.xsd">

  <application>
    <resource-bundle>
      <base-name>
        messages
      </base-name>
      <var>msg</var>
    </resource-bundle>
    <message-bundle>messages</message-bundle>
    <default-render-kit-id>PRIMEFACES_MOBILE</default-render-kit-id>
  </application>
</faces-config>
```

[messages_fr.properties] : le fichier des messages français :

```
forms=Formulaire
input=Saisie de texte
textarea=Saisie multi-lignes
dropdown=Liste d\u00e9roulante
range=Slider
switch=Switch
checkbox=Case \u00e0 cocher
checkboxes=Cases \u00e0 cocher
radios=Boutons radio
valider=Valider
confirmations=Confirmations
back=Retour
```

8.5.4 La page Primefaces mobile

La page [index.xhtml] est la suivante :

```xml
<?xml version="1.0" encoding='UTF-8'?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:p="http://primefaces.org/ui"
        xmlns:pm="http://primefaces.org/mobile"
        contentType="text/html">

  <pm:page title="mv-pfm-02">

    <!-- vue Forms -->
    <pm:view id="forms" swatch="b">
      <pm:header title="#{msg['forms']}"/>
      <pm:content>
        <h:form id="formulaire">
          <p:inputText id="inputText" label="#{msg['input']}" value="#{form.inputText}"/>
          <p:inputTextarea id="inputTextArea" label="#{msg['textarea']}" value="#{form.inputTextArea}"/>
          <pm:field>
            <h:outputLabel for="selectOneMenu" value="#{msg['dropdown']}"/>
            <h:selectOneMenu id="selectOneMenu" value="#{form.selectOneMenu}">
              <f:selectItem itemLabel="Option 1" itemValue="1" />
              <f:selectItem itemLabel="Option 2" itemValue="2" />
              <f:selectItem itemLabel="Option 3" itemValue="3" />
            </h:selectOneMenu>
          </pm:field>
          <pm:inputRange id="range" minValue="0" maxValue="100" label="#{msg['range']}"
value="#{form.range}"/>
          <pm:switch id="switch" onLabel="yes" offLabel="no" label="#{msg['switch']}"
value="#{form.bswitch}"/>
          <p:selectBooleanCheckbox id="booleanCheckbox" itemLabel="I agree" label="#{msg['checkbox']}"
value="#{form.booleanCheckbox}"/>
          <pm:field>
            <h:outputLabel for="manyCheckbox" value="#{msg['checkboxes']}"/>
            <h:selectManyCheckbox id="manyCheckbox" value="#{form.manyCheckbox}">
              <f:selectItem itemLabel="Option 1" itemValue="1" />
              <f:selectItem itemLabel="Option 2" itemValue="2" />
              <f:selectItem itemLabel="Option 3" itemValue="3" />
            </h:selectManyCheckbox>
          </pm:field>
          <pm:field>
            <h:outputLabel for="radio" value="#{msg['radios']}"/>
            <h:selectOneRadio id="radio" value="#{form.radio}">
              <f:selectItem itemLabel="Option 1" itemValue="1" />
              <f:selectItem itemLabel="Option 2" itemValue="2" />
              <f:selectItem itemLabel="Option 3" itemValue="3" />
            </h:selectOneRadio>
          </pm:field>
          <p:commandButton inline="true" value="#{msg['valider']}" update=":infos"
action="#{form.valider}"/>
        </h:form>
      </pm:content>
    </pm:view>

    <!-- vue infos -->
    <pm:view id="infos" swatch="b">
      <pm:header title="#{msg['confirmations']}">
        <f:facet name="left"><p:button value="#{msg['back']}" icon="back" href="#forms?
reverse=true"/></f:facet>
      </pm:header>
      <pm:content>
        <ul>
          <li>#{msg['input']} : ${form.inputText}</li>
          <li>#{msg['textarea']} : ${form.inputTextArea}</li>
          <li>#{msg['dropdown']} : ${form.selectOneMenu}</li>
          <li>#{msg['range']} : ${form.range}</li>
          <li>#{msg['switch']} : ${form.bswitch}</li>
          <li>#{msg['checkbox']} : ${form.booleanCheckbox}</li>
          <li>#{msg['checkboxes']} : ${form.manyCheckboxValue}</li>
          <li>#{msg['radios']} : ${form.radio}</li>
        </ul>
      </pm:content>
    </pm:view>
  </pm:page>
</f:view>
```

Nous avons repris l'exemple de démonstration du site de Primefaces mobile en l'aménageant pour qu'il fonctionne :

- la balise <*p:selectManyCheckbox*> n'était pas affichée. Nous l'avons remplacée par une balise <*h:selectManyCheckbox*> (ligne 32),
- la balise <*p:selectOneRadio*> n'était pas affichée. Nous l'avons remplacée par une balise <*p:selectOneRadio*> (ligne 40),
- ligne 27 : nous avons gardé la balise <*pm:inputRange*> qui affiche un slider. Celui-ci est bogué : on ne peut pas déplacer le curseur du slider. Mais on peut saisir un nombre qui le fait alors bouger,
- on notera qu'il y a peu de balises propres à PFM (lignes 13, 14, 15, 19, 27, 28 dans la première vue). Les autres balises sont des balises JSF et PF classiques. Primefaces mobile repose sur ces deux frameworks.

Notons la structure de la page :

```
1.  <?xml version="1.0" encoding='UTF-8'?>
2.  <f:view xmlns="http://www.w3.org/1999/xhtml"
3.          xmlns:f="http://java.sun.com/jsf/core"
4.          xmlns:h="http://java.sun.com/jsf/html"
5.          xmlns:ui="http://java.sun.com/jsf/facelets"
6.          xmlns:p="http://primefaces.org/ui"
7.          xmlns:pm="http://primefaces.org/mobile"
8.          contentType="text/html">
9.
10.   <pm:page title="mv-pfm-02">
11.
12.     <!-- vue Forms -->
13.     <pm:view id="forms" swatch="b">
14.       <pm:header ...>
15.       <pm:content>
16.         <h:form id="formulaire">
17.           ...
18.           <p:commandButton inline="true" value="#{msg['valider']}" update=":infos"
    action="#{form.valider}"/>
19.         </h:form>
20.       </pm:content>
21.     </pm:view>
22.
23.     <!-- vue infos -->
24.     <pm:view id="infos" swatch="b">
25.       <pm:header title="#{msg['confirmations']}">
26.         <f:facet name="left"><p:button value="#{msg['back']}" icon="back" href="#forms?
    reverse=true"/></f:facet>
27.       </pm:header>
28.       <pm:content>
29.         ...
30.       </pm:content>
31.     </pm:view>
32.   </pm:page>
33. </f:view>
```

- ligne 10 : définit une page PFM. L'attribut **title** fixe le titre affiché par le navigateur,
- une page est composée d'une ou plusieurs vues. Au premier affichage de la page, c'est la première vue qui est affichée. Ensuite on passe d'une vue à l'autre par navigation. Ici il y a deux vues **forms** ligne 13 et **infos** ligne 24,
- ligne 14 : la balise **header** définit l'entête d'une vue,
- ligne 15 : la balise **content** définit le contenu d'une vue,
- ligne 16 : la vue **forms** contient un formulaire JSF,
- ligne 18 : ce formulaire sera validé par un bouton Primefaces classique. Le formulaire sera posté au modèle [Form] et sera traité par la méthode *[Form].valider*. Cette méthode fera son travail. Nous verrons qu'elle demande l'affichage de la vue **infos**. Auparavant cette vue aura été mise à jour par l'appel AJAX (attribut **update**),
- ligne 26 : on pourra passer de la vue **infos** à la vue **forms** avec un bouton. On notera la forme de l'attribut **href** [`href="#forms?reverse=true"`]. La notation **#forms** désigne la vue **forms**. Le paramètre *reverse=true* demande un retour en arrière. J'ai repris cet attribut de la démo. Lorsqu'on l'enlève, on ne voit aucune différence sur un simulateur... Peut-être y-en-a-t-il une sur de vrais smartphones,
- on notera enfin l'utilisation de la bibliothèque Primefaces mobile ligne 7.

Même si elle apporte des nouveautés, cette page PFM est tout à fait compréhensible. Il en est de même pour son modèle [Form.java].

8.5.5 Le modèle [Form.java]

Ce modèle est le suivant :

```
1.  package beans;
2.
3.  import java.io.Serializable;
4.  import javax.faces.bean.ManagedBean;
5.  import javax.faces.bean.SessionScoped;
6.
7.  @ManagedBean
8.  @SessionScoped
9.  public class Form implements Serializable {
10.   // modèle
11.
12.   private String inputText = "bonjour";
13.   private String inputTextArea = "ligne1\nligne2";
14.   private String selectOneMenu = "2";
15.   private int range = 4;
16.   private boolean bswitch = true;
17.   private boolean booleanCheckbox = false;
18.   private String[] manyCheckbox = {"1", "3"};
19.   private String radio = "3";
20.
21.   public String valider() {
22.     // on passe à la vue infos
23.     return "pm:infos";
24.   }
25.
26.   public String getManyCheckboxValue() {
27.     StringBuffer str = new StringBuffer("[");
28.     for (String checkbox : manyCheckbox) {
29.       str.append(checkbox);
30.     }
31.     str.append("]");
32.     return str.toString();
33.   }
34.
35.   // getters et setters
36.   ...
37. }
```

Il n'y a là rien de neuf. Tout a été déjà vu. On notera quand même la forme de la méthode **valider** appelée par le formulaire. Ligne 21 : elle rend une chaîne de caractères qui a la forme '**pm:nomdelavueaafficher**'. Ici, la vue **infos** sera donc affichée. La méthode ne fait rien d'autre que cette navigation. Auparavant les champs des lignes 12-19 ont reçu les valeurs postées. Celles-ci vont servir à mettre à jour la vue **infos** (attribut **update** ci-dessous) :

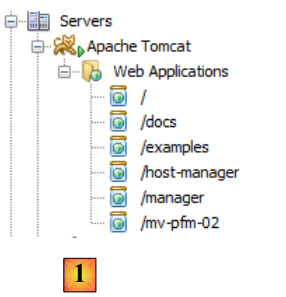
```
<p:commandButton inline="true" value="#{msg['valider']}" update=":infos" action="#{form.valider}"/>
```

La vue **infos** va alors afficher les valeurs postées :

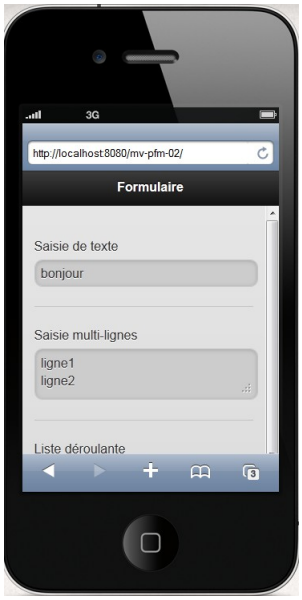
```
1.  <pm:view id="infos" swatch="b">
2.        <pm:header title="#{msg['confirmations']}">
3.          <f:facet name="left"><p:button value="#{msg['back']}" icon="back" href="#forms"/></f:facet>
4.        </pm:header>
5.        <pm:content>
6.          <ul>
7.            <li>#{msg['input']} : ${form.inputText}</li>
8.            <li>#{msg['textarea']} : ${form.inputTextArea}</li>
9.            <li>#{msg['dropdown']} : ${form.selectOneMenu}</li>
10.           <li>#{msg['range']} : ${form.range}</li>
11.           <li>#{msg['switch']} : ${form.bswitch}</li>
12.           <li>#{msg['checkbox']} : ${form.booleanCheckbox}</li>
13.           <li>#{msg['checkboxes']} : ${form.manyCheckboxValue}</li>
14.           <li>#{msg['radios']} : ${form.radio}</li>
15.         </ul>
16.       </pm:content>
17.     </pm:view>
```

8.5.6 Les tests

On exécute le projet Netbeans. Une fois le projet déployé sur Tomcat [1], on peut le tester sur un simulateur [http://iphone4simulator.com/] [2] :

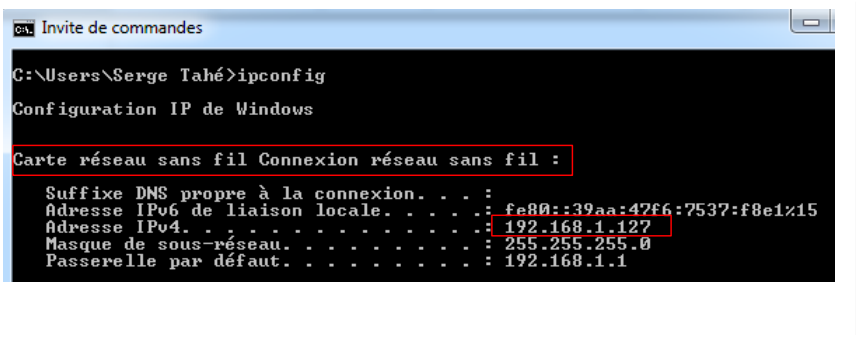


2

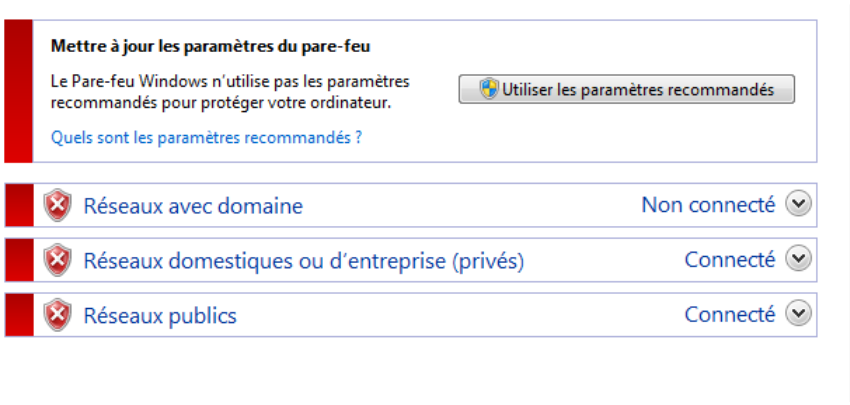


Nous allons maintenant montrer comment le tester sur un vrai smartphone. Nous allons connecter le serveur qui héberge l'application et le smartphone sur un même réseau WIFI :

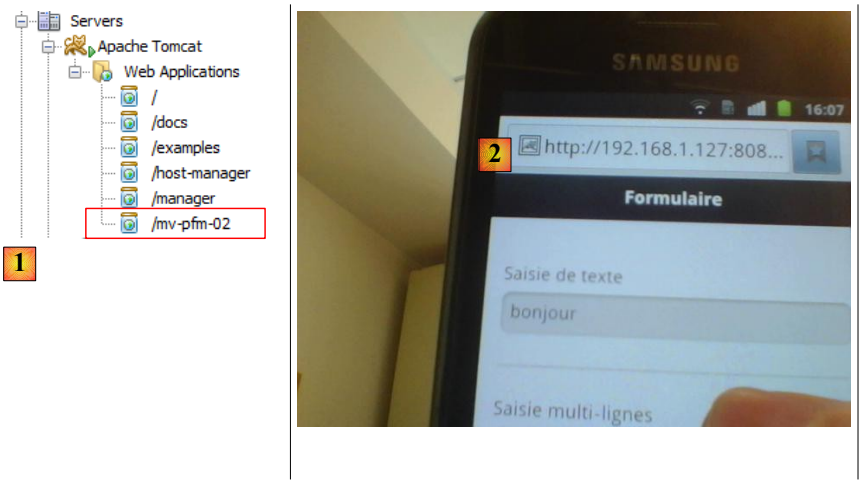
On connecte le serveur au réseau wifi. On peut ensuite vérifier son adresse IP :



On notera ci-dessus l'adresse IP du serveur : 192.168.1.127. On inhibe les éventuels pare-feu et antivirus du serveur :



Sur le serveur, on lance l'application [mv-pfm-02] dans Netbeans [1] :



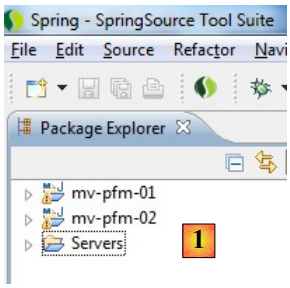
On connecte le smartphone sur le même réseau wifi que le serveur pour que les deux appareils se voient. Puis on ouvre un navigateur et on demande l'URL [http://192.168.1.127:8080/mv-pfm-02/] [2]. Ensuite on peut tester l'application. On découvrira ainsi avec bonheur que le slider qui ne marche pas dans le simulateur marche dans le smartphone. Il y a donc un décalage entre simulateurs et machines.

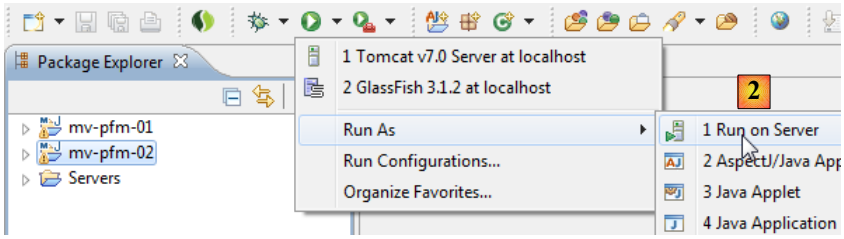
8.6 Conclusion

Nous n'avons présenté que les balises de formulaire de la bibliothèque Primefaces mobile mais c'est suffisant pour notre application exemple.

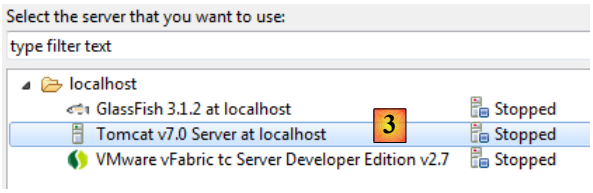
8.7 Les tests avec Eclipse

On importe dans Eclipse les projets Maven [1] et on les exécute [2],





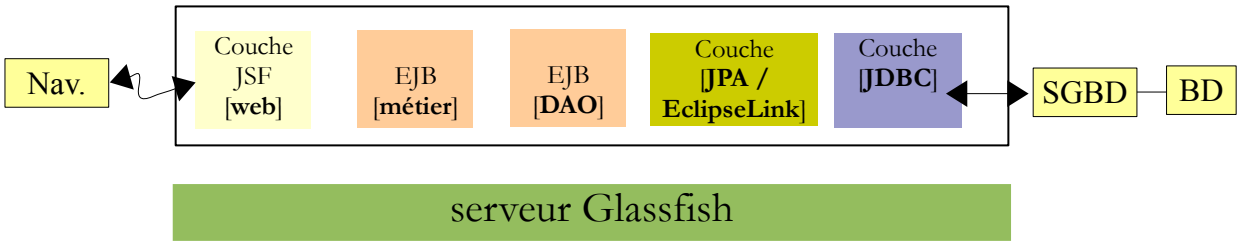
sur le serveur Tomcat [3]. L'application exécutée apparaît alors dans le navigateur interne à Eclipse [4].



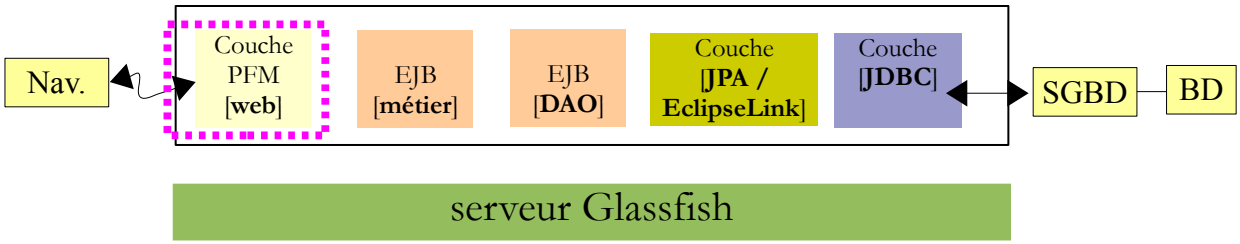
On notera que l'application [mv-pfm-02] a présenté des dysfonctionnements dans le navigateur d'Eclipse. Si on la charge dans les navigateurs Firefox ou IE, ceux-ci ont disparu.

9 Application exemple 05 : rdvmedecins-pfm-ejb

Rappelons la structure de l'application exemple 01 JSF / EJB développée pour le serveur Glassfish :



Nous ne changeons rien à cette architecture si ce n'est la couche web qui sera ici réalisée à l'aide de JSF, Primefaces et Primefaces mobile. Le navigateur cible sera celui d'un mobile.



Nous avons développé deux applications avec Glassfish :

- l'application 01 utilisait JSF / EJB et avait une interface assez rustique,
- l'application 03 utilisait PF / EJB et avait une interface riche.

Du fait du faible nombre de composants disponibles dans Primefaces mobile, nous allons revenir à l'interface rustique de l'application 01. Par ailleurs, les vues devront pouvoir s'adapter à la taille réduite des écrans de mobiles.

9.1 Les vues

Pour fixer les idées, nous donnons quelques vues de l'application exécutée dans le simulateur d'un iphone 4 :

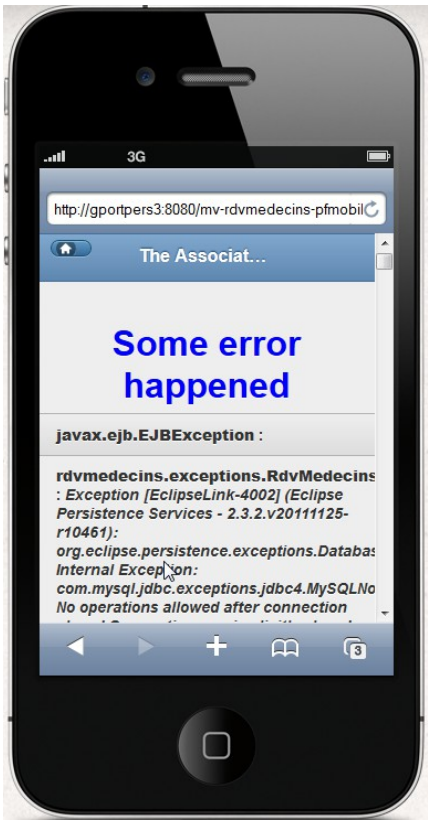
- en [1], la page d'accueil. On notera qu'il a fallu cette fois donner le nom de la machine (on peut aussi donner son adresse IP), parce qu'avec la machine *localhost*, ça n'affichait rien,
- en [2], le combo des médecins,
- en [3], la date désirée pour le rendez-vous,
- en [4], le bouton pour demander l'agenda du jour,
- en [5], la nouvelle vue affiche les créneaux horaires du médecin,
- en [6], une série de boutons pour naviguer dans le calendrier,
- en [7], un message pour rappeler le médecin et le jour,
- en [8], un créneau horaire pour réserver. On le fait,

- en [9], la vue de choix du client,
- en [10], un message rappelant le médecin, le jour et le créneau horaire concernés par le rendez-vous,
- en [11], le combo des clients,
- en [12], le bouton de validation,
- en [13], la validation nous fait revenir sur l'agenda,
- en [14], le créneau horaire occupé est désormais réservé. On va maintenant supprimer la réservation,

- en [15], on reste sur la même vue,
- mais en [16], le rendez-vous a été supprimé,

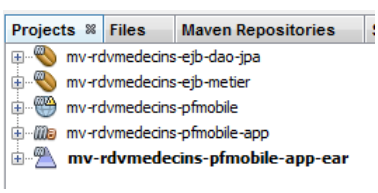
Il est possible dans la page d'accueil de changer de langue [17], [18], [19] :

Enfin, on a prévu une vue d'erreurs :



9.2 Le projet Netbeans

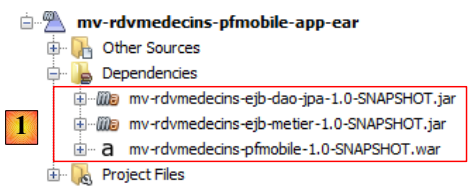
Le projet Netbeans est le suivant :



- [mv-rdvmedecins-ejb-dao-jpa] : projet EJB des couches [DAO] et [JPA] de l'exemple 01,
- [mv-rdvmedecins-ejb-metier] : projet EJB de la couche [métier] de l'exemple 01,
- [mv-rdvmedecins-pfmobile] : projet de la couche [web] / Primefaces mobile – nouveau,
- [mv-rdvmedecins-pfmobile-app-ear] : projet d'entreprise pour déployer l'application sur le serveur Glassfish – nouveau.

9.3 Le projet d'entreprise

Le projet d'entreprise ne sert qu'au déploiement des trois modules [mv-rdvmedecins-ejb-dao-jpa], [mv-rdvmedecins-ejb-metier], [mv-rdvmedecins-pfmobile] sur le serveur Glassfish. Le projet Netbeans est le suivant :



Le projet n'existe que pour ces trois dépendances [1] définies dans 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"
    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.  ...
5.
6.    <groupId>istia.st</groupId>
7.    <artifactId>mv-rdvmedecins-pfmobile-app-ear</artifactId>
8.    <version>1.0-SNAPSHOT</version>
9.    <packaging>ear</packaging>
10.
11.   <name>mv-rdvmedecins-pfmobile-app-ear</name>
12.
13. ...
14.     <dependencies>
15.         <dependency>
16.             <groupId>${project.groupId}</groupId>
17.             <artifactId>mv-rdvmedecins-ejb-dao-jpa</artifactId>
18.             <version>${project.version}</version>
19.             <type>ejb</type>
20.         </dependency>
21.         <dependency>
22.             <groupId>${project.groupId}</groupId>
23.             <artifactId>mv-rdvmedecins-ejb-metier</artifactId>
24.             <version>${project.version}</version>
25.             <type>ejb</type>
26.         </dependency>
27.         <dependency>
```

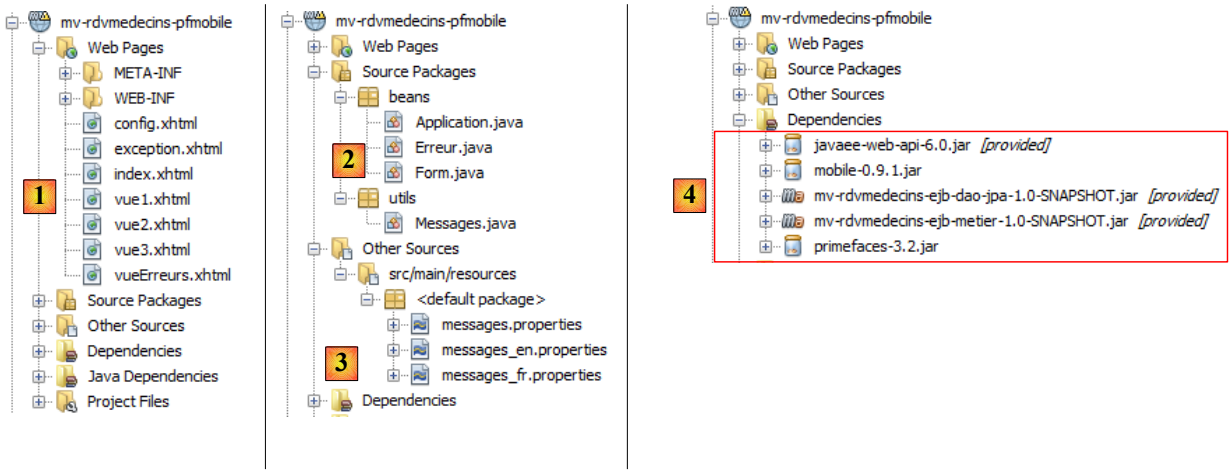
```
28.             <groupId>${project.groupId}</groupId>
29.             <artifactId>mv-rdvmedecins-pfmobile</artifactId>
30.             <version>${project.version}</version>
31.             <type>war</type>
32.         </dependency>
33.     </dependencies>
34. </project>
```

- lignes 6-9 : l'artifact Maven du projet d'entreprise,
- lignes 14-33 : les trois dépendances du projet. On notera bien le type de celles-ci (lignes 19, 25, 31).

Pour exécuter l'application web, il faudra exécuter ce projet d'entreprise.

9.4 Le projet web Primefaces mobile

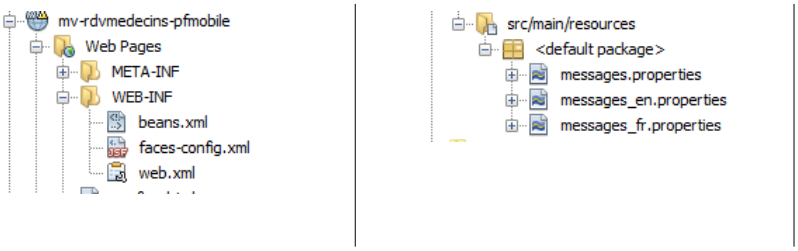
Le projet web Primefaces mobile est le suivant :



- en [1], les pages du projet. La page [index.xhtml] est l'unique page du projet. Elle comporte cinq vues [vue1.xhtml], [vue2.xhtml], [vue3.xhtml], [vueErreurs.xhtml] et [config.xhtml],
- en [2], les beans Java. Le bean [Application] de portée *application*, le bean [Form] de portée *session*. La classe [Erreur] encapsule une erreur,
- en [3], les fichiers de messages pour l'internationalisation,
- en [4], les dépendances. Le projet web a des dépendances sur le projet EJB de la couche [DAO], le projet EJB de la couche [métier] et Primefaces mobile pour la couche [web].

9.5 La configuration du projet

La configuration du projet est celle des projets Primefaces ou JSF que nous avons étudiés. Nous listons les fichiers de configuration sans les réexpliquer.



[web.xml] : configure l'application web.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
3.    <context-param>
4.      <param-name>javax.faces.PROJECT_STAGE</param-name>
5.      <param-value>Development</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
9.      <param-value>true</param-value>
10.   </context-param>
11.   <servlet>
12.     <servlet-name>Faces Servlet</servlet-name>
13.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
14.     <load-on-startup>1</load-on-startup>
15.   </servlet>
16.   <servlet-mapping>
17.     <servlet-name>Faces Servlet</servlet-name>
18.     <url-pattern>/faces/*</url-pattern>
19.   </servlet-mapping>
20.   <session-config>
21.     <session-timeout>
22.       30
23.     </session-timeout>
24.   </session-config>
25.   <welcome-file-list>
26.     <welcome-file>faces/index.xhtml</welcome-file>
27.   </welcome-file-list>
28.   <error-page>
29.     <error-code>500</error-code>
30.     <location>/faces/exception.xhtml</location>
31.   </error-page>
32.   <error-page>
33.     <exception-type>Exception</exception-type>
34.     <location>/faces/exception.xhtml</location>
35.   </error-page>
36.
37. </web-app>
```

On notera, ligne 26 que la page [index.xhtml] est la page d'accueil de l'application.

[faces-config.xml] : configure l'application JSF

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <resource-bundle>
12.       <base-name>
13.         messages
14.       </base-name>
15.       <var>msg</var>
16.     </resource-bundle>
17.       <message-bundle>messages</message-bundle>
18.     <default-render-kit-id>PRIMEFACES_MOBILE</default-render-kit-id>
19.   </application>
20. </faces-config>
```

[beans.xml] : vide mais nécessaire pour l'annotation *@Named*

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://java.sun.com/xml/ns/javaee"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
```

```
</beans>
```

[messages_fr.properties] : le fichier des messages en français

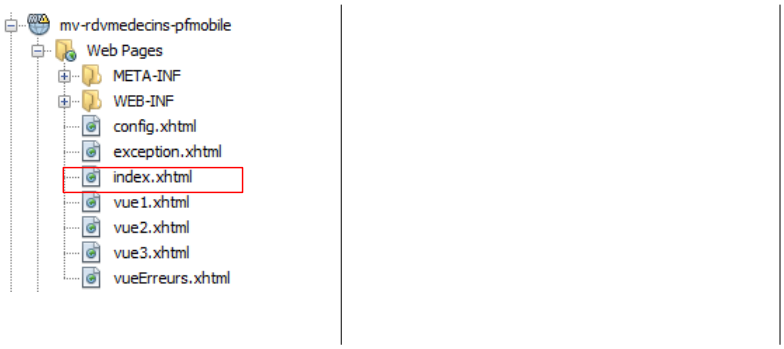
```
# page
page.titre=Les M\u00e9decins Associ\u00e9s
format.date=dd/MM/yyyy
format.date_detail=dd/MM/yyyy

# vue1
vue1.header=Les M\u00e9decins Associ\u00e9s - R\u00e9servations

# formulaire 1
form1.titre=R\u00e9servations
form1.medecin=M\u00e9decin
form1.jour=Jour (jj/mm/aaaa)
form1.date.requise=La date est n\u00e9cessaire
form1.date.invalide=La date est invalide
form1.date.invalide_detail=La date est invalide
form1.agenda=Agenda
form1.options=Options

# formulaire 2
form2.titre={0} {1} {2}<br/>{3}
form2.titre_detail={0} {1} {2}<br/>{3}
form2.retour=Retour
form2.supprimer=Supprimer
form2.reserver=R\u00e9server
form2.precedent=Jour pr\u00e9c\u00e9dent
form2.suivant=Jour suivant
form2.today=Aujourd'hui

# formulaire 3
form3.client=Client
form3.valider=Valider
form3.titre={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
form3.titre_detail={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
form3.retour=Retour

# erreur
erreur.titre=Une erreur s'est produite.

# config
config.retour=Retour
config.titre=Configuration
config.langue=Langue
config.langue.francais=Fran\u00e7ais
config.langue.anglais=Anglais
config.valider=Valider

#exception
exception.titre=Application indisponible. Veuillez recommencer ult\u00e9rieurement.
```

[messages_en.properties] : le fichier des messages en anglais

```
# page
page.titre=The Associated Doctors
format.date=dd/MM/yyyy
format.date_detail=dd/MM/yyyy

# vue1
vue1.header=The Associated Doctors - Reservations

# formulaire 1
form1.titre=Reservations
form1.medecin=Doctor
form1.jour=day (dd/mm/yyyy)
form1.date.requise=The date is necessary
form1.date.invalide=Invalid date
form1.date.invalide_detail=invalid date
form1.agenda=Diary
form1.options=Options

# formulaire 2
```

```
form2.titre={0} {1} {2}<br/>{3}
form2.titre_detail={0} {1} {2}<br/>{3}
form2.retour=Back
form2.supprimer=Delete
form2.reserver=Reserve
form2.precedent=Previous Day
form2.suivant=Next Day
form2.today=Today

# formulaire 3
form3.client=Patient
form3.valider=Validate
form3.titre={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
form3.titre_detail={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
form3.retour=Back

# erreur
erreur.titre=Some error happened

# config
config.retour=Back
config.titre=Configuration
config.langue=Language
config.langue.francais=French
config.langue.anglais=English
config.valider=Validate

#exception
exception.titre=Application not available. Please try again later.
```

9.6 La page [index.xhtml]



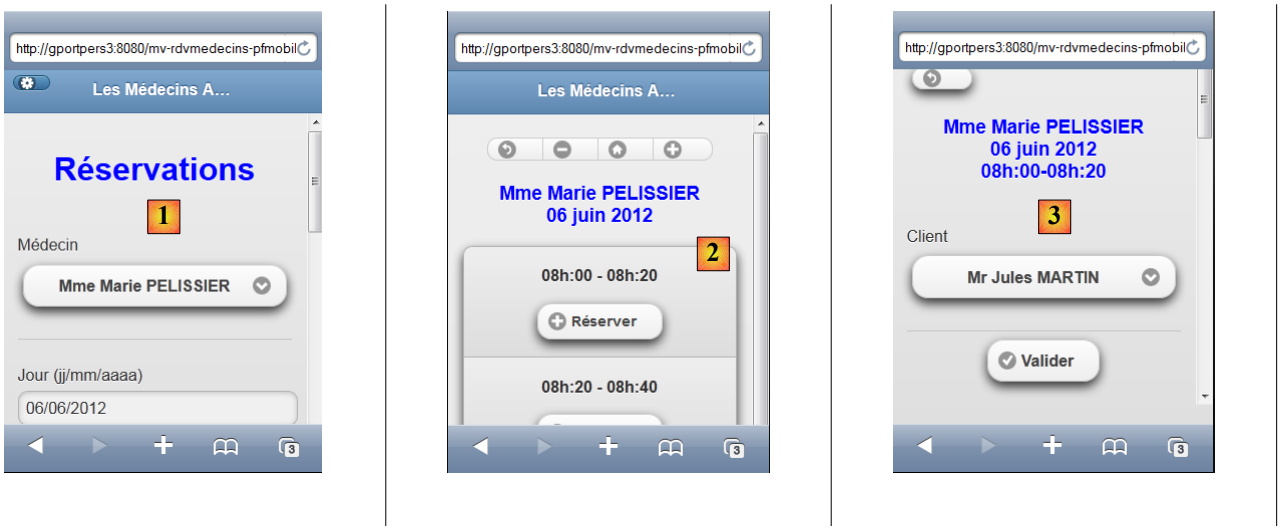
Le projet affiche toujours la même page, la page [index.xhtml] suivante :

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:p="http://primefaces.org/ui"
        xmlns:pm="http://primefaces.org/mobile"
        contentType="text/html"
        locale="#{form.locale}">

  <pm:page title="#{msg['page.titre']}">
    <pm:view id="vue1">
      <ui:fragment rendered="#{form.form1Rendered}">
        <ui:include src="vue1.xhtml"/>
      </ui:fragment>
      <ui:fragment rendered="#{form.erreurInit}">
        <ui:include src="vueErreurs.xhtml"/>
      </ui:fragment>
    </pm:view>
    <pm:view id="vue2">
      <ui:fragment rendered="#{form.form2Rendered}">
        <ui:include src="vue2.xhtml"/>
      </ui:fragment>
```

```
23.       </pm:view>
24.       <pm:view id="vue3">
25.         <ui:fragment rendered="#{form.form3Rendered}">
26.           <ui:include src="vue3.xhtml"/>
27.         </ui:fragment>
28.       </pm:view>
29.       <pm:view id="vueErreurs">
30.         <ui:fragment rendered="#{form.erreurRendered}">
31.           <ui:include src="vueErreurs.xhtml"/>
32.         </ui:fragment>
33.       </pm:view>
34.       <pm:view id="config">
35.         <ui:include src="config.xhtml"/>
36.       </pm:view>
37.     </pm:page>
38.   </f:view>
```

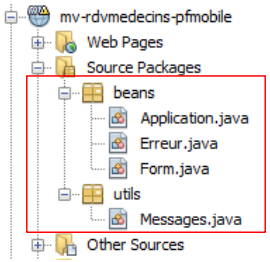
- ligne 8 : la page est internationalisée (attribut **locale**),
- ligne 10 : la page contient cinq vues : **vue1** ligne 11, **vue2** ligne 19, **vue3** ligne 24, **vueErreurs** ligne 29, **config** ligne 34. A un moment donné seule l'une de ces vues est visible. Au démarrage de l'application, c'est la vue **vue1** qui est affichée. Ici, on a rencontré la difficulté suivante : si l'initialisation de l'application s'est bien passée, on doit afficher [vue1.xhtml], sinon c'est [vueErreurs.xhtml] qui doit être affichée. On a réglé le problème en faisant en sorte que le contenu de la vue **vue1** soit géré par le modèle qui donne des valeurs aux booléens *[Form].form1rendered* (ligne 12) et *[Form].erreurInit* (ligne 15) pour fixer le contenu de **vue1** (ligne 11),

Dans un simulateur, la vue [vue1.xhtml] a le rendu [1], la vue [vue2.xhtml] le rendu [2], la vue [vue3.xhtml] le rendu [3] :



la vue [vueErreurs.xhtml] le rendu [4], la vue [config.xhtml] le rendu [5] :

9.7 Les beans du projet



La classe du paquetage [utils] a déjà été présentée : la classe [Messages] est une classe qui facilite l'internationalisation des messages d'une application. Elle a été étudiée au paragraphe 2.8.5.7, page 124.

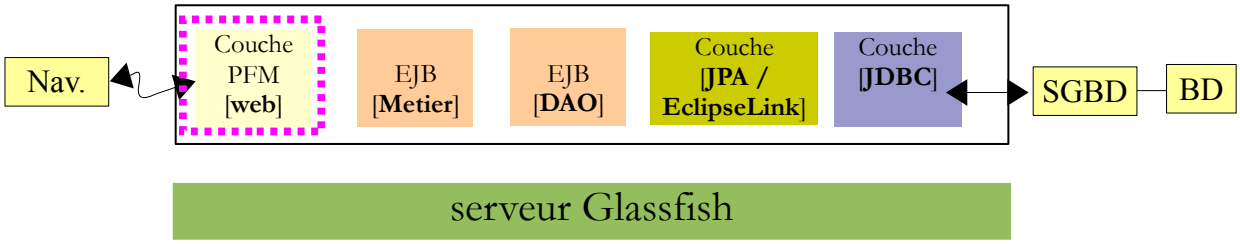
9.7.1 Le bean Application

Le bean [Application.java] est un bean de portée *application*. On se rappelle que ce type de bean sert à mémoriser des données en lecture seule et disponibles pour tous les utilisateurs de l'application. Ce bean est le suivant :

```
1.  package beans;
2.
3.  import javax.ejb.EJB;
4.  import javax.enterprise.context.ApplicationScoped;
5.  import javax.inject.Named;
6.  import rdvmedecins.metier.service.IMetierLocal;
7.
8.  @Named(value = "application")
9.  @ApplicationScoped
10. public class Application {
11.
12.   // couche métier
13.   @EJB
14.   private IMetierLocal metier;
15.
16.   public Application() {
17.   }
18.
19.   // getters
20.
```

```
21.   public IMetierLocal getMetier() {
22.     return metier;
23.   }
24.
25. }
```

- ligne 8 : on donne au bean le nom **application**,
- ligne 9 : il est de portée **application**,
- lignes 13-14 : une référence sur l'interface locale de la couche [métier] lui sera injectée par le conteneur EJB du serveur d'application. Rappelons-nous l'architecture de l'application :



L'application PFM et l'EJB [Metier] vont s'exécuter dans la même JVM (Java Virtual Machine). Donc la couche [PFM] va utiliser l'interface locale de l'EJB. C'est tout. Le bean [Application] ne contient rien d'autre. Pour avoir accès à la couche [métier], les autres beans iront la chercher dans ce bean.

9.7.2 Le bean [Erreur]

La classe [Erreur] est la suivante :

```
1.  package beans;
2.
3.  public class Erreur {
4.
5.    public Erreur() {
6.    }
7.
8.    // champ
9.    private String classe;
10.   private String message;
11.
12.   // constructeur
13.   public Erreur(String classe, String message){
14.     this.setClasse(classe);
15.     this.message=message;
16.   }
17.
18.   // getters et setters
19. ...
20. }
```

- ligne 9, le nom d'une classe d'exception si une exception a été lancée,
- ligne 10 : un message d'erreur.

9.7.3 Le bean [Form]

Son code est le suivant :

```
1.  package beans;
2.
3.  ...
4.
5.  @Named(value = "form")
6.  @SessionScoped
7.  public class Form implements Serializable {
```

```
8.
9.    public Form() {
10.   }
11.   // bean Application
12.   @Inject
13.   private Application application;
14.   private IMetierLocal metier;
15.   // cache de la session
16.   private List<Medecin> medecins;
17.   private List<Client> clients;
18.   private Map<Long, Medecin> hMedecins = new HashMap<Long, Medecin>();
19.   private Map<Long, Client> hClients = new HashMap<Long, Client>();
20.   // modèle
21.   private Long idMedecin;
22.   private Date jour = new Date();
23.   private String strJour;
24.   private Boolean form1Rendered;
25.   private Boolean form2Rendered;
26.   private Boolean form3Rendered;
27.   private Boolean erreurRendered;
28.   private String form2Titre;
29.   private String form3Titre;
30.   private AgendaMedecinJour agendaMedecinJour;
31.   private Long idCreneauChoisi;
32.   private Medecin medecin;
33.   private Long idClient;
34.   private CreneauMedecinJour creneauChoisi;
35.   private List<Erreur> erreurs;
36.   private Boolean erreurInit = false;
37.   private String action;
38.   private String locale = "fr";
39.   private String msgErreurDate = "";
40.   private SimpleDateFormat dateFormatter;
41.   private Boolean erreurDate;
42.
43.   @PostConstruct
44.   private void init() {
45.     System.out.println("init");
46.     // au départ pas d'erreur
47.     erreurInit = false;
48.     // le formatage des dates
49.     dateFormatter = new SimpleDateFormat(Messages.getMessage(null, "format.date", null).getSummary());
50.     dateFormatter.setLenient(false);
51.     // le jour courant
52.     strJour = dateFormatter.format(jour);
53.     // on récupère la couche métier
54.     metier = application.getMetier();
55.     // on met les médecins et les clients en cache
56.     try {
57.       medecins = metier.getAllMedecins();
58.       clients = metier.getAllClients();
59.     } catch (Throwable th) {
60.       // on note l'erreur
61.       erreurInit = true;
62.       prepareVueErreur(th);
63.       return;
64.     }
65.     // vérification des listes
66.     if (medecins.size() == 0) {
67.       // on note l'erreur
68.       erreurInit = true;
69.       erreurs = new ArrayList<Erreur>();
70.       erreurs.add(new Erreur("", "La liste des médecins est vide"));
71.     }
72.     if (clients.size() == 0) {
73.       // on note l'erreur
74.       erreurInit = true;
75.       erreurs = new ArrayList<Erreur>();
76.       erreurs.add(new Erreur("", "La liste des clients est vide"));
77.     }
78.     // erreur ?
79.     if (erreurInit) {
80.       // la vue des erreurs est affichée
81.       setForms(false, false, false, true);
82.       return;
83.     }
```

```
84.
85.     // les dictionnaires
86.     for (Medecin m : medecins) {
87.       hMedecins.put(m.getId(), m);
88.     }
89.     for (Client c : clients) {
90.       hClients.put(c.getId(), c);
91.     }
92.     // la vue 1
93.     setForms(true, false, false, false);
94.   }
95.
96.   // affichage vue
97.   private void setForms(Boolean form1Rendered, Boolean form2Rendered, Boolean form3Rendered, Boolean
    erreurRendered) {
98.     this.form1Rendered = form1Rendered;
99.     this.form2Rendered = form2Rendered;
100.    this.form3Rendered = form3Rendered;
101.    this.erreurRendered = erreurRendered;
102.  }
103.
104.  // préparation vueErreur
105.  private void prepareVueErreur(Throwable th) {
106.    // on crée la liste des erreurs
107.    erreurs = new ArrayList<Erreur>();
108.    erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
109.    while (th.getCause() != null) {
110.      th = th.getCause();
111.      erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
112.    }
113.// la vue des erreurs est affichée
114.    setForms(false, false, false, true);
115.  }
116.
117.  // getters et setters
118.  ..
119.}
```

- lignes 5-7 : la classe [Form] est un bean de nom **form** et de portée **session**. On rappelle qu'alors la classe doit être sérialisable,
- lignes 12-13 : le bean **form** a une référence sur le bean **application**. Celle-ci sera injectée par le conteneur de servlets dans lequel s'exécute l'application (présence de l'annotation @**Inject**).
- lignes 24-27 : contrôlent l'affichage des vues **vue1** (ligne 24), **vue2** (ligne 25), **vue3** (ligne 26), **vueErreurs** (ligne 27),
- lignes 43-44 : la méthode **init** est exécutée juste après l'instanciation de la classe (présence de l'annotation @*PostConstruct*),
- lignes 49-50 : gèrent le format des dates. Primefaces mobile n'offre pas de calendrier. Par ailleurs, les validateurs JSF ne sont pas utilisables dans une page PFM. Aussi devrons-nous gérer à la main la saisie de la date de l'agenda,
- ligne 49 : fixe le format des dates. Ce format est pris dans les fichiers internationalisés :

```
format.date=dd/MM/yyyy
format.date_detail=dd/MM/yyyy
```

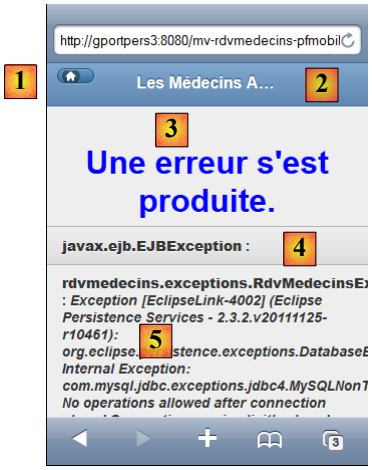
- ligne 50 : on indique qu'on ne doit pas être " *laxiste* " vis à vis des dates. Si on ne le fait pas, une saisie comme 32/12/2011 qui est une saisie incorrecte est considérée comme la date valide 01/01/2012,
- ligne 54 : on récupère une référence sur la couche [métier] auprès du bean [Application],
- lignes 57-58 : on demande à la couche [métier], la liste des médecins et des clients,
- lignes 66-91 : si tout s'est bien passé, les dictionnaires des médecins et des clients sont construits. Ils sont indexés par leur numéro. Ensuite, la vue [vue1.xhtml] sera affichée (ligne 93),
- ligne 59 : en cas d'erreur, le modèle de la page [vueErreurs.xhtml] est construit. Ce modèle est la liste d'erreurs de la ligne 35,
- lignes 105-115 : la méthode [prepareVueErreur] construit la liste d'erreurs à afficher. La page [index.xhtml] affiche alors la vue [vueErreurs.xhtml] (ligne 114).

La page [erreur.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
```

```
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- Vue Erreurs -->
11.   <pm:header title="#{msg['page.titre']}" swatch="b">
12.     <f:facet name="left">
13.       <p:button icon="home" value=" " href="#vue1?reverse=true" />
14.     </f:facet>
15.   </pm:header>
16.   <pm:content>
17.     <div align="center">
18.       <h1><h:outputText value="#{msg['erreur.titre']}" style="color: blue"/></h1>
19.     </div>
20.
21.     <p:dataList value="#{form.erreurs}" var="erreur">
22.       <b>#{erreur.classe}</b> : <i>#{erreur.message}</i>
23.     </p:dataList>
24.   </pm:content>
25. </html>
```

Elle utilise une balise <**p:dataList**> (lignes 21-23) pour afficher la liste des erreurs. Le bouton de la ligne 13 permet de revenir sur la vue **vue1**.



- le bouton [1] est produit par la ligne 13. L'attribut **icon** fixe l'icône du bouton. Le bouton ramène à la vue **vue1** (attribut **href**),
- l'entête [2] est produit par les lignes 11-15,
- le titre [3] est produit par la ligne 18,
- le texte [4] est produit par le modèle **#{erreur.classe}** de la ligne 22,
- le texte [5] est produit par le modèle **#{erreur.message}** de la ligne 22.

Nous allons maintenant définir les différentes phases de la vie de l'application. Pour chaque action de l'utilisateur, nous étudierons les vues concernées et les gestionnaires des événements qui s'y produisent.

9.8 L'affichage de la page d'accueil

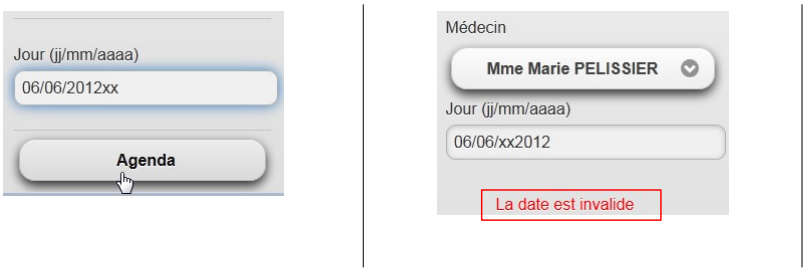
Si tout va bien, la première vue affichée est [vue1.xhtml]. Cela donne la vue suivante :

Le code de la vue [vue1.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- Vue 1 -->
11.   <pm:header title="#{msg['page.titre']}" swatch="b">
12.     <f:facet name="left">
13.       <p:button icon="gear" value=" "  href="#config" />
14.     </f:facet>
15.   </pm:header>
16.   <pm:content>
17.     <h:form id="form1">
18.       <div align="center">
19.         <h1><h:outputText value="#{msg['form1.titre']}" style="color: blue"/></h1>
20.       </div>
21.       <pm:field>
22.         <h:outputLabel value="#{msg['form1.medecin']}" for="choixMedecin"/>
23.         <h:selectOneMenu id="choixMedecin" value="#{form.idMedecin}">
24.           <f:selectItems value="#{form.medecins}" var="medecin" itemLabel="#{medecin.titre}
    #{medecin.prenom} #{medecin.nom}" itemValue="#{medecin.id}"/>
25.         </h:selectOneMenu>
26.       </pm:field>
27.       <pm:field>
28.         <h:outputLabel value="#{msg['form1.jour']}" for="jour"/>
29.         <p:inputText id="jour" value="#{form.strJour}"/>
30.         <ui:fragment rendered="#{form.erreurDate}">
31.           <p:spacer width="50px"/>
32.           <h:outputText id="msgErreurDate" value="#{form.msgErreurDate}" style="color: red"/>
33.         </ui:fragment>
34.       </pm:field>
35.       <p:commandButton value="#{msg['form1.agenda']}" update=":form1, :vue2, :vueErreurs"
    action="#{form.getAgenda}" />
36.     </h:form>
37.   </pm:content>
38. </html>
```

- lignes 11-15 : produisent l'entête [1],
- ligne 13 : produit le bouton [2]. Un clic sur celui-ci affiche la vue **config** (attribut **href**),
- ligne 19 : produit le titre [3],
- lignes 21-26 : produisent le combo des médecins [4],

- lignes 27-34 : produisent la saisie de la date [5]. Cette saisie est faite avec une balise <*p:inputText*> sans validateur. La validation de la date sera faite côté serveur. Si la date est incorrecte, celui-ci positionnera un message d'erreur affiché par les lignes 30-34,



- ligne 35 : le bouton qui valide le formulaire. Il met à jour trois zones : **form1** (le formulaire dans vue1), **vue2** et **vueErreurs**. En effet, si la date est invalide c'est **form1** qui doit être mise à jour. Si la date est correcte c'est **vue2** qui doit être mise à jour. Enfin s'il se produit une exception (connexion à la base cassée par exemple), c'est **vueErreurs** qui doit être affichée. On peut être tenté de mettre **vue1** à la place de **form1** (on met à jour toute la vue). Dans ce cas, l'application bogue.

Cette vue est adossée au modèle suivant :

```
1.  @Named(value = "form")
2.  @SessionScoped
3.  public class Form implements Serializable {
4.
5.    public Form() {
6.    }
7.
8.    // bean Application
9.    @Inject
10.   private Application application;
11.   private IMetierLocal metier;
12.   // cache de la session
13.   private List<Medecin> medecins;
14.   private List<Client> clients;
15.   // modèle
16.   private Long idMedecin;
17.   private Date jour = new Date();
18.   private String strJour;
19.   private Boolean form1Rendered;
20.   private Boolean form2Rendered;
21.   private Boolean form3Rendered;
22.   private Boolean erreurRendered;
23.   private String msgErreurDate = "";
24.   private Boolean erreurDate;
25.
26.     // liste des médecins
27.   public List<Medecin> getMedecins() {
28.     return medecins;
29.   }
30.
31.   // agenda
32.   public String getAgenda() {
33. ...
34.     }
35.   }
```

- le champ de la ligne 16 alimente en lecture et écriture la valeur de la liste de la ligne 23 de la page. A l'affichage initial de la page, elle fixe la valeur sélectionnée dans le combo,
- la méthode des lignes 27-29 génère les éléments du combo des médecins (ligne 24 de la vue). Chaque option générée aura pour **label** (itemLabel) les **titre**, **nom**, **prénom** du médecin et pour valeur (itemValue), l'**id** du médecin,
- le champ de la ligne 18 alimente en lecture / écriture le champ de saisie de la ligne 29 de la page,
- lignes 32-34 : la méthode **getAgenda** gère le clic sur le bouton [Agenda] de la ligne 35 de la page. Son code est le suivant :

```
1.  // agenda
```

```
2.    public String getAgenda() {
3.      try {
4.        // on vérifie le jour
5.        jour = dateFormatter.parse(strJour);
6.        // pas d'erreur
7.        erreurDate=false;
8.        msgErreurDate = "";
9.        // on crée l'agenda
10.       return getAgenda(jour);
11.     } catch (ParseException ex) {
12.       // msg d'erreur
13.       erreurDate=true;
14.       msgErreurDate = Messages.getMessage(null, "form1.date.invalide", null).getSummary();
15.       // vue1
16.       setForms(true, false, false, false);
17.       return "pm:vue1";
18.     }
19.   }
```

- la méthode commence par vérifier la validité de la date saisie par l'utilisateur,
- ligne 5 : la date est *parsée* selon le format de date initialisé par la méthode *init* à l'instanciation du modèle,
- ligne 11 : s'il se produit une exception, une erreur est positionnée (ligne 13), un message d'erreur internationalisé est construit (ligne 14), la vue **vue1** est préparée (ligne 16) et on affiche la vue **vue1** (ligne 17),
- ligne 10 : si la date est correcte, on exécute la méthode suivante :

```
1.    // agenda
2.    public String getAgenda(Date jour) {
3.      // aucun créneau choisi pour l'instant
4.      creneauChoisi = null;
5.      try {
6.        // on récupère le médecin choisi
7.        medecin = hMedecins.get(idMedecin);
8.        // titre formulaire 2
9.        form2Titre = Messages.getMessage(null, "form2.titre", new Object[]{medecin.getTitre(),
    medecin.getPrenom(), medecin.getNom(), new SimpleDateFormat("dd MMM yyyy").format(jour)}).getSummary();
10.       // l'agenda du médecin pour un jour donné
11.       agendaMedecinJour = metier.getAgendaMedecinJour(medecin, jour);
12.       // on affiche la vue 2
13.       setForms(false, true, false, false);
14.       return "pm:vue2";
15.     } catch (Throwable th) {
16.       System.out.println(th);
17.       // vue des erreurs
18.       prepareVueErreur(th);
19.       return "pm:vueErreurs";
20.     }
21. }
```

On retrouve là un code déjà rencontré à diverses reprises. Ligne 9, un message internationalisé est construit pour la vue **vue2** :

```
1.  form2.titre={0} {1} {2}<br/>{3}
2.  form2.titre_detail={0} {1} {2}<br/>{3}
```

On notera que nous avons mis du XHTML dans le message. Il sera affiché de la façon suivante :

9.9 Afficher l'agenda d'un médecin

L'agenda du médecin est affiché par la vue [vue2.xhtml] :

Le code de la vue [vue2.xhtml] est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- Vue 2 -->
11.   <pm:header title="#{msg['page.titre']}" swatch="b"/>
12.   <pm:content>
13.     <h:form id="form2">
14.       <div align="center">
15.         <pm:buttonGroup orientation="horizontal">
16.           <p:commandButton inline="true" icon="back" value=" " action="#{form.showVue1}" update=":vue1"/>
17.           <p:commandButton inline="true" icon="minus" value=" " action="#{form.getPreviousAgenda}"
    update=":form2"/>
```

```
18.           <p:commandButton inline="true" icon="home" value=" " action="#{form.getTodayAgenda}"
    update=":form2"/>
19.           <p:commandButton inline="true" icon="plus" value=" " action="#{form.getNextAgenda}"
    update=":form2"/>
20.         </pm:buttonGroup>
21.         <h3><h:outputText value="#{form.form2Titre}" style="color: blue" escape="false"/></h3>
22.       </div>
23.
24.       <p:dataList id="creneaux" type="inset" value="#{form.agendaMedecinJour.creneauxMedecinJour}"
    var="creneauMedecinJour">
25.         <p:column>
26.           <div align="center">
27.             <h2>
28.               <h:outputFormat value="{0,number,#00}h:{1,number,#00} - {2,number,#00}h:{3,number,#00}">
29.                 <f:param value="#{creneauMedecinJour.creneau.hdebut}" />
30.                 <f:param value="#{creneauMedecinJour.creneau.mdebut}" />
31.                 <f:param value="#{creneauMedecinJour.creneau.hfin}" />
32.                 <f:param value="#{creneauMedecinJour.creneau.mfin}" />
33.               </h:outputFormat>
34.               <ui:fragment rendered="#{creneauMedecinJour.rv!=null}">
35.                 <br/>
36.                 <h:outputText value="#{creneauMedecinJour.rv.client.titre}
    #{creneauMedecinJour.rv.client.prenom} #{creneauMedecinJour.rv.client.nom}" style="color: blue"/>
37.               </ui:fragment>
38.             </h2>
39.           </div>
40.           <div align="center">
41.             <ui:fragment rendered="#{creneauMedecinJour.rv!=null}">
42.               <p:commandButton inline="true" action="#{form.action}" value="#{msg['form2.supprimer']}"
    icon="minus" update=":form2, :vue3, :vueErreurs">
43.                 <f:setPropertyActionListener value="#{creneauMedecinJour.creneau.id}"
    target="#{form.idCreneauChoisi}"/>
44.               </p:commandButton>
45.             </ui:fragment>
46.             <ui:fragment rendered="#{creneauMedecinJour.rv==null}">
47.               <p:commandButton inline="true" action="#{form.action}" value="#{msg['form2.reserver']}"
    icon="plus" update=":form2, :vue3, :vueErreurs">
48.                 <f:setPropertyActionListener value="#{creneauMedecinJour.creneau.id}"
    target="#{form.idCreneauChoisi}"/>
49.               </p:commandButton>
50.             </ui:fragment>
51.           </div>
52.         </p:column>
53.       </p:dataList>
54.     </h:form>
55.   </pm:content>
56. </html>
```

- lignes 11 : produit l'entête [1],
- ligne 15-20 : produit le groupe de boutons [2],
- ligne 21 : produit le titre [3]. On notera la valeur de l'attribut **escape**. C'est ce qui permet d'interpréter le code XHTML que nous avons mis dans **form2Titre**,
- ligne 24 : l'affichage des créneaux horaires est faite à l'aide d'un **dataList**,
- lignes 28-33 : produisent l'intitulé du créneau horaire [4],
- lignes 34-37 : affichent un fragment s'il y a un rendez-vous dans le créneau horaire,
- ligne 36 : affiche l'identité du client ayant pris le rendez-vous,
- lignes 41-45 : affichent le bouton [Supprimer] s'il y a un rendez-vous,
- lignes 46-50 : affichent le bouton [Réserver] s'il n'y a pas de rendez-vous.

Cette vue est principalement alimentée par le modèle suivant :

```
private AgendaMedecinJour agendaMedecinJour;
```

qui alimente le **dataList** de la ligne 24. Ce champ a été construit par la méthode **getAgenda**, lorsqu'on est passé de la vue **vue1** à la vue **vue2**.

9.10 Suppression d'un rendez-vous

La suppression d'un rendez-vous correspond à la séquence suivante :

La vue concernée par cette action est la suivante :

```
1.  <ui:fragment rendered="#{creneauMedecinJour.rv!=null}">
2.                <p:commandButton inline="true" action="#{form.action}" value="#{msg['form2.supprimer']}"
    icon="minus" update=":form2, :vueErreurs">
3.                  <f:setPropertyActionListener value="#{creneauMedecinJour.creneau.id}"
    target="#{form.idCreneauChoisi}"/>
4.                </p:commandButton>
5.              </ui:fragment>
```

- ligne 2 : le bouton [Supprimer] est associé à la méthode *[Form].action* (attribut **action**),
- ligne 3 : l'**id** du créneau sur lequel on est positionné sera envoyé au modèle *[Form].idCreneauChoisi*,
- ligne 2 : l'appel AJAX mettra à jour les zones **form2** (formulaire de **vue2**) et la vue **vueErreurs**. On a en effet deux cas : sii tout se passe bien la vue **vue2** sera réaffichée sinon c'est la vue **vueErreurs** qui le sera.

La méthode [action] est la suivante :

```
1.    // action sur RV
2.    public String action() {
3.      // on recherche le créneau dans l'agenda
4.      int i = 0;
5.      Boolean trouvé = false;
6.      while (!trouvé && i < agendaMedecinJour.getCreneauxMedecinJour().length) {
7.        if (agendaMedecinJour.getCreneauxMedecinJour()[i].getCreneau().getId() == idCreneauChoisi) {
8.          trouvé = true;
9.        } else {
10.         i++;
11.       }
12.     }
13.     // a-t-on trouvé ?
14.     if (!trouvé) {
15.       // c'est bizarre - on réaffiche form2
16.       setForms(false, true, false, false);
17.       return "pm:vue2";
18.     } else {
19.       creneauChoisi = agendaMedecinJour.getCreneauxMedecinJour()[i];
20.     }
21.     // on a trouvé
22.     // selon l'action désirée
23.     if (creneauChoisi.getRv() == null) {
24.       return reserver();
25.     } else {
26.       return supprimer();
27.     }
28.   }
29.
```

```
30.   // réservation
31.   public String reserver() {
32.  ...
33.   }
34.
35.   public String supprimer() {
36.     try {
37.       // suppression d'un Rdv
38.       metier.supprimerRv(creneauChoisi.getRv());
39.       // on remet à jour l'agenda
40.       agendaMedecinJour = metier.getAgendaMedecinJour(medecin, jour);
41.       // on affiche form2
42.       setForms(false, true, false, false);
43.       return "pm:vue2";
44.     } catch (Throwable th) {
45.       // vue erreurs
46.       prepareVueErreur(th);
47.       return "pm:vueErreurs";
48.     }
49. }
```

- lignes 3-12 : on recherche le créneau horaire dont on a reçu l'**id** (ligne 7),
- si on le trouve pas, ce qui est anormal, on réaffiche **vue2** (lignes 16-17),
- ligne 19 : si on le trouve, on mémorise l'objet [CreneauMedecinJour] correspondant. C'est lui qui nous donne accès au rendez-vous à supprimer,
- ligne 26 : on le supprime,
- lignes 35-48 : la méthode **supprimer** renvoie la vue **vue2** si la suppression s'est bien passée (lignes 42-43) ou la vue **vueErreurs** s'il y a eu un problème (lignes 46-47).

9.11 Prise de rendez-vous

La prise de rendez-vous correspond à la séquence suivante :

On passe de la vue **vue2** à la vue **vue3**. Le code impliquée par cette action est le suivant :

```
1.  <ui:fragment rendered="#{creneauMedecinJour.rv==null}">
2.                <p:commandButton inline="true" action="#{form.action}"
    value="#{msg['form2.reserver']}" icon="plus" update=":vue3, :vueErreurs">
3.                  <f:setPropertyActionListener value="#{creneauMedecinJour.creneau.id}"
    target="#{form.idCreneauChoisi}"/>
4.                </p:commandButton>
5.              </ui:fragment>
```

- ligne 2 : le bouton [Réserver] est associé à la méthode *[Form].action* (attribut *action*), donc la même que pour le bouton [Supprimer]. L'appel AJAX met à jour les vues **vue3** et **vueErreurs** selon qu'il y a ou non des erreurs lors du traitement de l'appel.
- ligne 3 : comme pour le bouton [Supprimer], l'**id** du créneau horaire est passé au modèle.

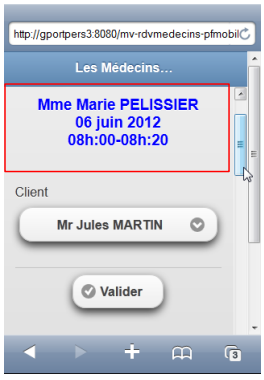
Le modèle qui traite cette action est le suivant :

```
1.  // action sur RV
2.    public String action() {
3.      ...
4.      // selon l'action désirée
5.      if (creneauChoisi.getRv() == null) {
6.        return reserver();
7.      } else {
8.        return supprimer();
9.      }
10.   }
11.
12.   // réservation
13.   public String reserver() {
14.     try {
15.       // titre formulaire 3
16.       form3Titre = Messages.getMessage(null, "form3.titre", new Object[]{medecin.getTitre(),
    medecin.getPrenom(), medecin.getNom(), new SimpleDateFormat("dd MMM yyyy").format(jour),
17.                 creneauChoisi.getCreneau().getHdebut(), creneauChoisi.getCreneau().getMdebut(),
    creneauChoisi.getCreneau().getHfin(), creneauChoisi.getCreneau().getMfin()}).getSummary();
18.       // on affiche le formulaire 3
19.       setForms(false, false, true, false);
20.       return "pm:vue3";
21.     } catch (Throwable th) {
22.       // vue erreurs
23.       prepareVueErreur(th);
24.       return "pm:vueErreurs";
25.     }
26.   }
```

- lignes 2-10 : la méthode *action* va déterminer la référence *creneauChoisi* de l'objet [CreneauMedecinJour] qui fait l'objet d'une réservation puis appeler la méthode *reserver*,
- ligne 16 : un message internationalisé est construit. C'est le suivant :

```
form3.titre={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
form3.titre_detail={0} {1} {2}<br/>{3}<br/>{4,number,#00}h:{5,number,#00}-{6,number,#00}h:{7,number,#00}
```

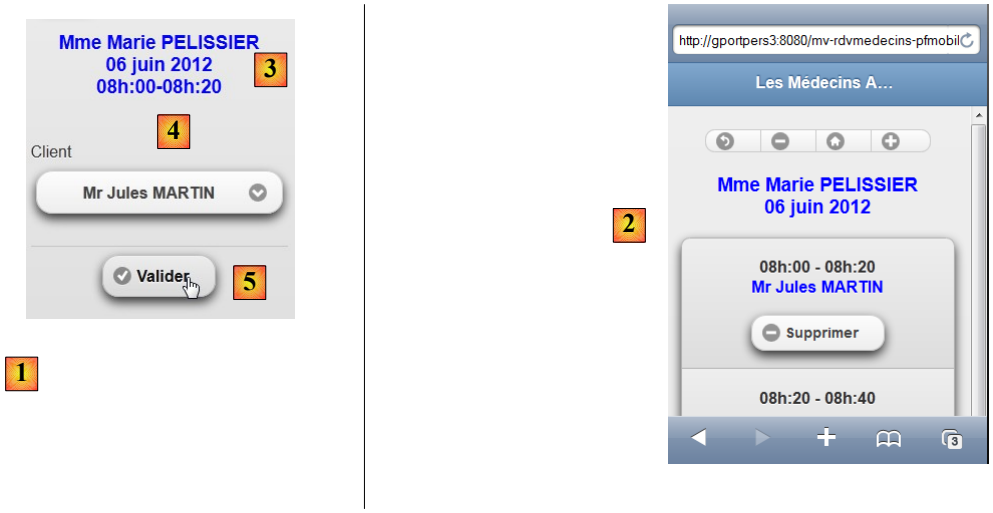
Ce sera le titre de la vue **vue3**. Comme pour la vue **vue2**, ce titre comporte du XML. Il comporte également des paramètres formatés pour afficher les horaires du créneau,



- lignes 19-20 : le vue **vue3** est affichée,
- lignes 23-24 : la vue **vueErreurs** est affichée si on a rencontré des problèmes.

9.12 Validation d'un rendez-vous

La validation d'un rendez-vous correspond à la séquence suivante :



En [1], la vue **vue3** et en [2], la vue **vue2** après ajout d'un rendez-vous.

Le code [vue3.xhtml] de **vue3** est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- Vue 3 -->
11.   <pm:header title="#{msg['page.titre']}" swatch="b"/>
12.   <pm:content>
13.     <h:form id="form3">
14.       <p:commandButton inline="true" value=" " icon="back" action="#{form.showVue2}" update=":vue2"/>
15.       <div align="center">
16.         <h3><h:outputText value="#{form.form3Titre}" style="color: blue" escape="false"/></h3>
17.       </div>
18.       <pm:field>
19.         <h:outputLabel value="#{msg['form3.client']}" for="choixClient"/>
20.         <h:selectOneMenu id="choixClient" value="#{form.idClient}">
21.           <f:selectItems value="#{form.clients}" var="client" itemLabel="#{client.titre} #{client.prenom}
    #{client.nom}" itemValue="#{client.id}"/>
22.         </h:selectOneMenu>
23.       </pm:field>
24.       <div align="center">
25.         <p:commandButton inline="true" value="#{msg['form3.valider']}" action="#{form.validerRv}"
    update=":vue2, :vueErreurs" icon="check"/>
26.       </div>
27.     </h:form>
28.   </pm:content>
29. </html>
```

- ligne 16 : produit le titre de la vue [3]. On notera la valeur de l'attribut **escape** qui permet l'interprétation des caractères XHTML dans le titre,
- lignes 18-23 : produisent le combo des clients [4],
- ligne 25 : produit le bouton [Valider] [5]. La méthode *[Form].validerRv* est associée à ce bouton :

```
1.  // validation Rv
```

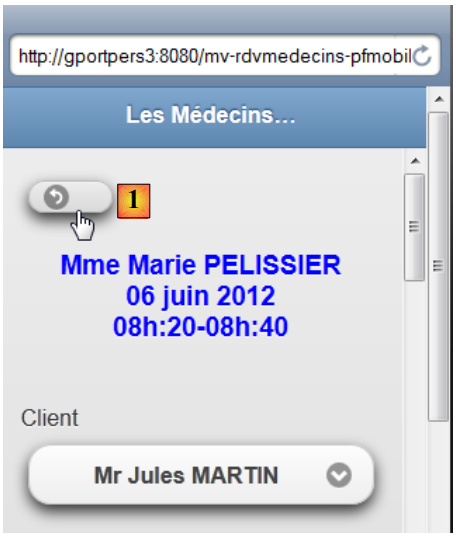
```
2.    public String validerRv() {
3.      try {
4.        // on récupère une instance du créneau choisi
5.        Creneau creneau = metier.getCreneauById(idCreneauChoisi);
6.        // on ajoute le Rv
7.        metier.ajouterRv(jour, creneau, hClients.get(idClient));
8.        // on remet à jour l'agenda
9.        agendaMedecinJour = metier.getAgendaMedecinJour(medecin, jour);
10.       // on affiche form2
11.       setForms(false, true, false, false);
12.       return "pm:vue2";
13.     } catch (Throwable th) {
14.       // vue erreurs
15.       prepareVueErreur(th);
16.       return "pm:vueErreurs";
17.     }
18.   }
```

Ce code a déjà été rencontré dans la version 01. On notera simplement l'affichage des vues :

- la vue **vue2** (lignes 11-12) si tout s'est bien passé,
- la vue **vueErreurs** (lignes 15-16) sinon.

9.13 Annulation d'une prise de rendez-vous

Cela correspond à la séquence suivante :



On retrouve l'agenda

Le bouton [1] dans la vue [vue3.xhtml] est le suivant :

```
<p:commandButton inline="true" value=" " icon="back" action="#{form.showVue2}" update=":vue2"/>
```

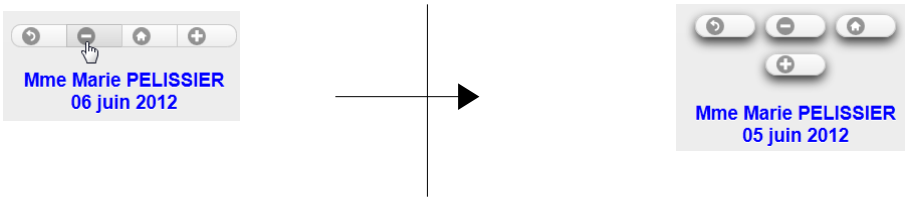
La méthode *[Form].showVue2* est donc appelée. Elle se contente d'afficher **vue2** :

```
1.    public String showVue2() {
2.      // vue2
3.      setForms(false, true, false, false);
4.      return "pm:vue2?reverse=true";
5.  }
```

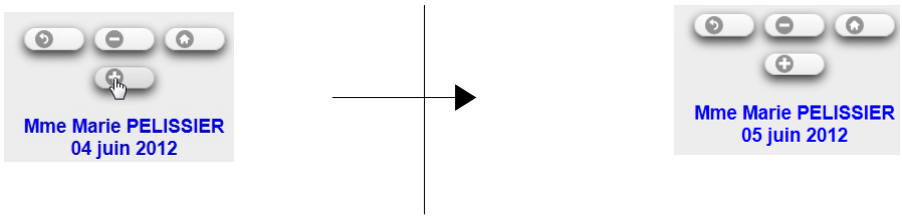
9.14 Navigation dans le calendrier

Dans **vue2**, des boutons permettent de naviguer dans le calendrier :

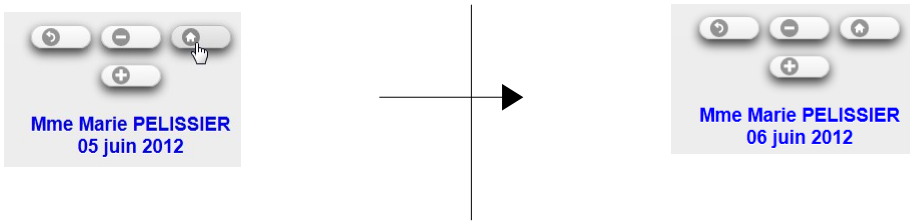
Jour précédent :



Jour suivant :



Aujourd'hui :



Non montré sur les copies d'écran ci-dessus, l'agenda est mis à jour et affiche les rendez-vous du nouveau jour choisi.

Les balises des trois boutons concernés sont les suivantes dans [vue2.xhtml] :

```
1.  <pm:buttonGroup orientation="horizontal">
2.            <p:commandButton inline="true" icon="back" value=" " action="#{form.showVue1}" update=":vue1"/>
3.            <p:commandButton inline="true" icon="minus" value=" " action="#{form.getPreviousAgenda}"
    update=":form2"/>
4.            <p:commandButton inline="true" icon="home" value=" " action="#{form.getTodayAgenda}"
    update=":form2"/>
5.            <p:commandButton inline="true" icon="plus" value=" " action="#{form.getNextAgenda}"
    update=":form2"/>
6.          </pm:buttonGroup>
```

Les méthode *[Form].getPreviousAgenda*, *[Form].getNextAgenda*, *[Form].today* ont été étudiées dans l'exemple 03.

9.15 Changement de langue d'affichage

Le changement de langue est géré par un bouton de la page d'accueil :

Le code du bouton est le suivant :

```
<p:button icon="gear" value=" "  href="#config" />
```

Il fait donc passer à la vue **config** [2]. La vue [config.xhtml] est la suivante :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- Vue 1 -->
11.   <pm:header title="#{msg['page.titre']}" swatch="b">
12.     <f:facet name="left">
13.       <p:button icon="back" value=" " href="#vue1?reverse=true" />
14.     </f:facet>
15.   </pm:header>
16.   <pm:content>
17.     <h:form id="frmConfig">
18.       <div align="center">
19.         <h3><h:outputText value="#{msg['config.titre']}" style="color: blue"/></h3>
20.       </div>
21.       <pm:field>
22.         <h:outputLabel value="#{msg['config.langue']}" for="langue"/>
23.         <h:selectOneRadio id="langue" value="#{form.locale}">
24.           <f:selectItem itemLabel="#{msg['config.langue.francais']}" itemValue="fr"/>
25.           <f:selectItem itemLabel="#{msg['config.langue.anglais']}" itemValue="en" />
26.         </h:selectOneRadio>
27.       </pm:field>
28.       <p:commandButton value="#{msg['config.valider']}" action="#{form.configurer}" update=":vue1"/>
29.     </h:form>
30.   </pm:content>
31. </html>
```

- ligne 11 : fait afficher [3],
- ligne 13 : fait afficher le bouton [4]. Ce bouton permet de revenir à la vue **vue1**,
- ligne 17 : le formulaire de la vue,
- ligne 19 : fait afficher le titre de la vue [5],
- lignes 21-27 : font afficher les boutons radio. La valeur (**itemValue**) du bouton radio sélectionné sera postée au modèle *[Form].locale* (attribut **value** de la ligne 23),
- ligne 28 : fait afficher le bouton [Valider]. L'appel AJAX met à jour la vue **vue1** (attribut **update**). La méthode appelée est *[Form].configurer* :

```
1.  public String configurer(){
```

```
2.      // après configuration - on réaffiche vue1
3.      redirect();
4.      return null;
5.    }
6.
7.    private void redirect() {
8.      // on redirige le client vers la servlet
9.      ExternalContext ctx = FacesContext.getCurrentInstance().getExternalContext();
10.     try {
11.       ctx.redirect(ctx.getRequestContextPath());
12.     } catch (IOException ex) {
13.       Logger.getLogger(Form.class.getName()).log(Level.SEVERE, null, ex);
14.     }
15.   }
```

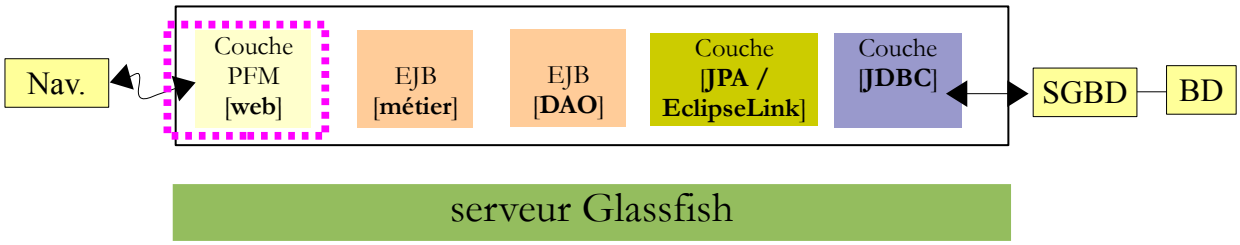
La méthode **configurer** (ligne 1) se contente de rediriger le navigateur du mobile vers l'URL de l'application. C'est donc la page [index.xhtml] qui va être chargée :

```
1.  <f:view xmlns="http://www.w3.org/1999/xhtml"
2.          xmlns:f="http://java.sun.com/jsf/core"
3.          xmlns:h="http://java.sun.com/jsf/html"
4.          xmlns:ui="http://java.sun.com/jsf/facelets"
5.          xmlns:p="http://primefaces.org/ui"
6.          xmlns:pm="http://primefaces.org/mobile"
7.          contentType="text/html"
8.          locale="#{form.locale}">
9.
10.   <pm:page title="#{msg['page.titre']}">
11.     <pm:view id="vue1">
12.       ...
13.     </pm:view>
14.     ...
15.   </pm:page>
16. </f:view>
```

- ligne 8 : la vue va utiliser la langue qui vient d'être changée (attribut **locale**) et va afficher **vue1** (ligne 11).

9.16 Conclusion

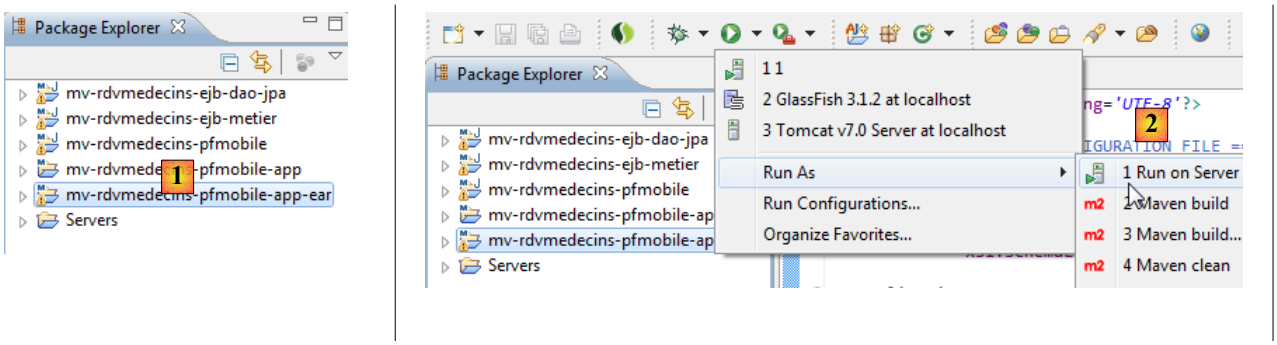
Rappelons l'architecture de l'application que nous venons de construire :



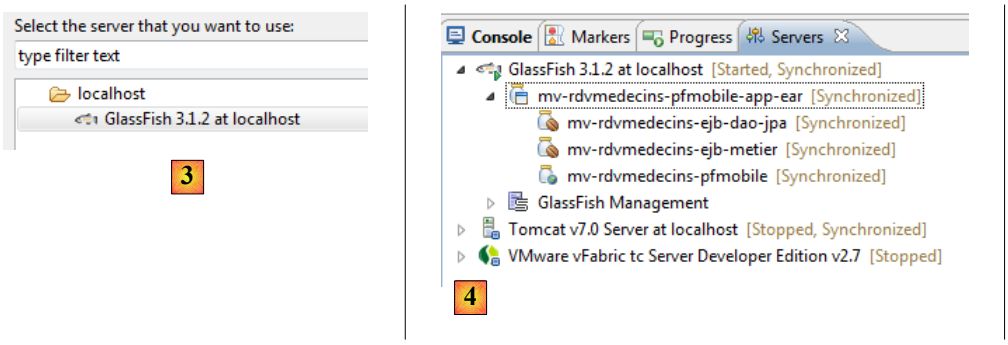
Le passage à une interface pour mobiles a nécessité la réécriture des pages XHTML. Le modèle en revanche a lui peu bougé. Les couches basses [métier], [DAO], [JPA] n'ont, elles, pas bougé du tout.

9.17 Tests Eclipse

Comme nous l'avons fait pour les précédentes versions de l'application exemple, nous montrons comment tester cette version 05 avec Eclipse. Tout d'abord, nous importons dans Eclipse les projets Maven de l'exemple 05 [1] :



- [mv-rdvmedecins-ejb-dao-jpa] : les couches [DAO] et [JPA],
- [mv-rdvmedecins-ejb-metier] : la couche [métier],
- [mv-rdvmedecins-pfmobile] : la couche [web] implémentée par Primefaces mobile,
- [mv-rdvmedecins-pfmobile-app] : le parent du projet d'entreprise [mv-rdvmedecins-pfmobile-app-ear]. Lorsqu'on importe le projet parent, le projet fils est automatiquement importé,
- en [2], on exécute le projet d'entreprise [mv-rdvmedecins-pfmobile-app-ear],



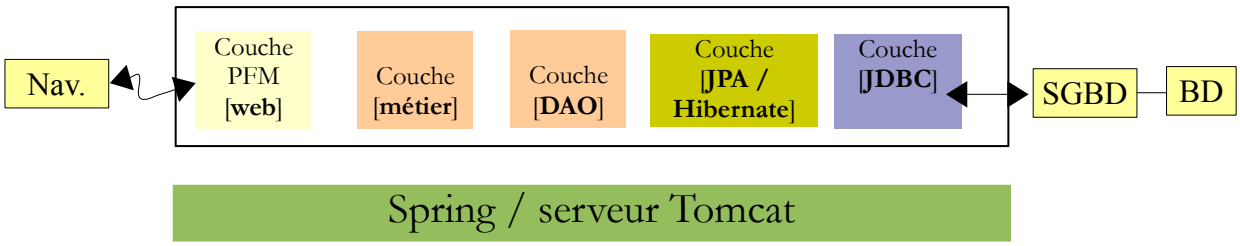
- en [3], on choisit le serveur Glassfish,
- en [4], dans l'onglet [Servers], l'application a été déployée. Elle ne s'exécute pas d'elle-même. Il faut demander son URL [http://localhost:8080/mv-rdvmedecins-pfmobile/] dans un navigateur ou un simulateur de mobile [5] :


3G
http://localhost:8080/mv-rdvmedecins-pfmobile/
Les Médecins A...
Réservations
5
Médecin
Mme Marie PELISSIER
Jour (jj/mm/aaaa)
06/06/2012

10 Application exemple-06 : rdvmedecins-pfm-spring

10.1 Le portage

Nous portons maintenant l'application précédente dans un environnement Spring / Tomcat :

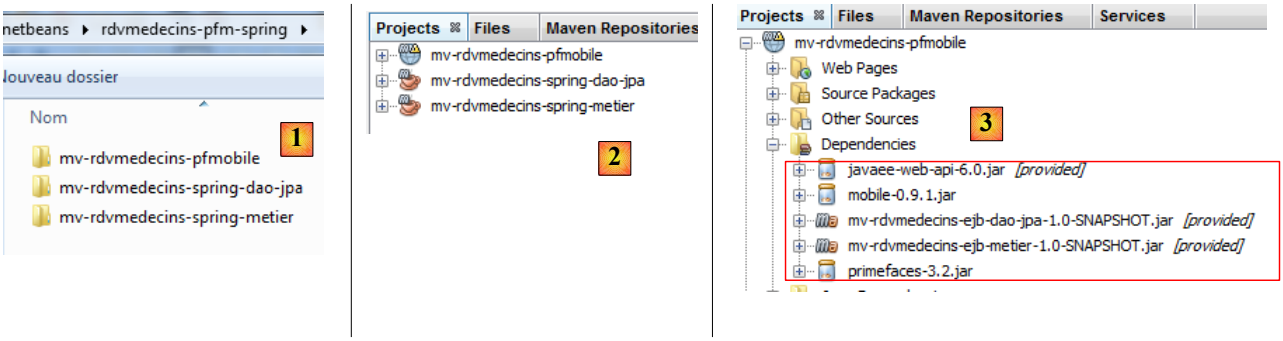


Nous allons nous appuyer sur deux applications déjà écrites. Nous allons utiliser :

- les couches [DAO] et [JPA] de la version 02 JSF / Spring,
- la couche [web] / Primefaces mobile de la version 05 PFM / EJB,
- les fichiers de configuration de Spring de la version 02

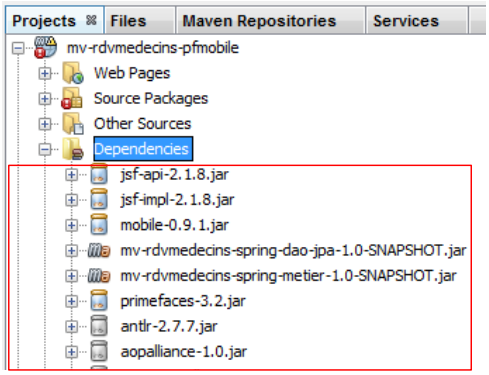
Nous refaisons là un travail analogue à celui qui avait été fait pour porter l'application JSF2 / EJB / Glassfish dans un environnement JSF2 / Spring Tomcat. Aussi donnerons-nous moins d'explications. Le lecteur peut, s'il en est besoin, se reporter à ce portage.

Nous mettons tous les projets nécessaires au portage dans un nouveau dossier [rdvmedecins-pfm-spring] [1] :



- [mv-rdvmedecins-spring-dao-jpa] : les couches [DAO] et [JPA] de la version 02 JSF / Spring,
- [mv-rdvmedecins-spring-metier] : la couche [métier] de la version 02 JSF / Spring,
- [mv-rdvmedecins-pfmobile] : la couche [web] de la version 05 Primefaces mobile / EJB,
- en [2], nous les chargeons dans Netbeans,
- en [3], les dépendances du projet web ne sont plus correctes :
  - les dépendances sur les couches [DAO], [JPA], [métier] doivent être changées pour cibler désormais les projets Spring ;
  - le serveur Glassfish fournissait les bibliothèques de JSF. Ce n'est plus le cas avec le serveur Tomcat. Il faut donc les ajouter aux dépendances.

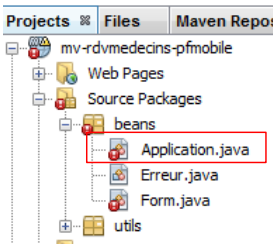
Le projet [web] évolue comme suit :

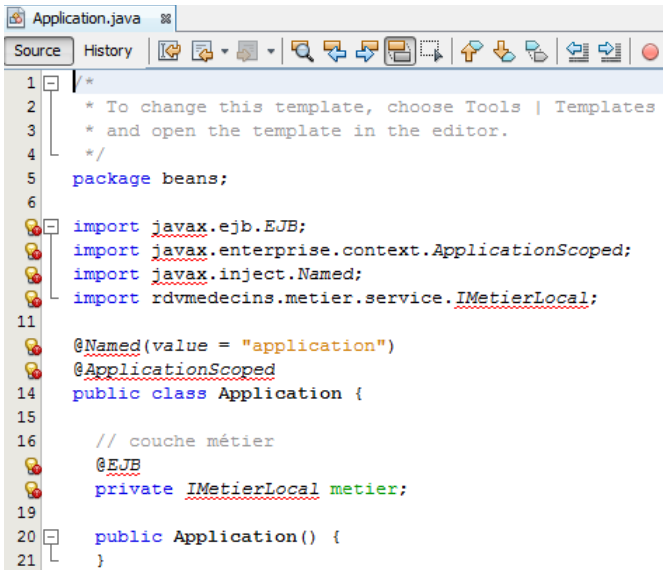


Le fichier [pom.xml] de la couche [web] a désormais les dépendances suivantes :

```
<dependencies>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>primefaces</artifactId>
      <version>3.2</version>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>mobile</artifactId>
      <version>0.9.1</version>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>2.1.8</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>2.1.8</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>mv-rdvmedecins-spring-dao-jpa</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>mv-rdvmedecins-spring-metier</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
```

Des erreurs apparaissent. Elles sont dues aux références EJB de la couche [web]. Etudions d'abord le bean [Application] :



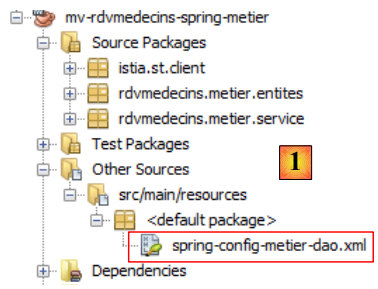


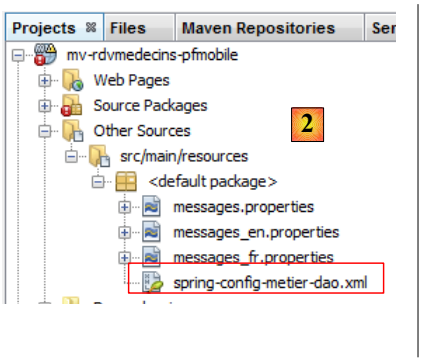
Nous enlevons toutes les lignes erronées à cause de paquetages manquants, nous renommons [IMetier] (c'est son nom dans la couche [métier] Spring) l'interface [IMetierLocal] et nous utilisons Spring pour l'instancier :

```
1.  package beans;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.  import org.springframework.context.ApplicationContext;
6.  import org.springframework.context.support.ClassPathXmlApplicationContext;
7.  import rdvmedecins.metier.service.IMetier;
8.
9.  public class Application {
10.
11.   // couche métier
12.   private IMetier metier;
13.   // erreurs
14.   private List<Erreur> erreurs = new ArrayList<Erreur>();
15.   private Boolean erreur = false;
16.
17.   public Application() {
18.     try {
19.       // instanciation couche [métier]
20.       ApplicationContext ctx = new ClassPathXmlApplicationContext("spring-config-metier-dao.xml");
21.       metier = (IMetier) ctx.getBean("metier");
22.     } catch (Throwable th) {
23.       // on note l'erreur
24.       erreur = true;
25.       erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
26.       while (th.getCause() != null) {
27.         th = th.getCause();
28.         erreurs.add(new Erreur(th.getClass().getName(), th.getMessage()));
29.       }
30.       return;
31.     }
32.   }
33.
34.   // getters
35.   public Boolean getErreur() {
36.     return erreur;
37.   }
38.
39.   public List<Erreur> getErreurs() {
40.     return erreurs;
41.   }
42.
43.   public IMetier getMetier() {
44.     return metier;
45.   }
```

```
46. }
```

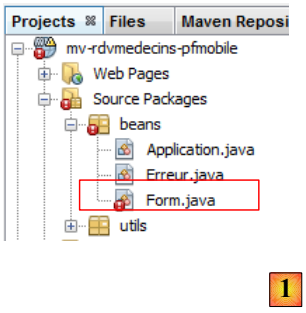
- lignes 20-21 : instanciation de la couche [métier] à partir du fichier de configuration de Spring. Ce fichier est celui utilisé par la couche [métier] [1]. Nous le copions dans le projet web [2] :

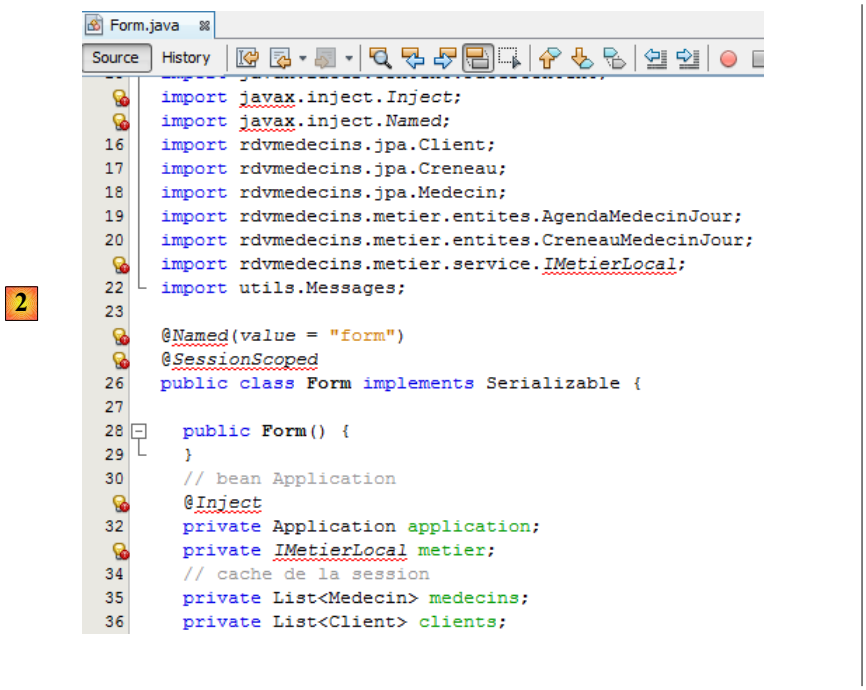




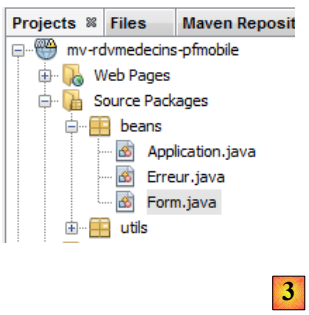
- lignes 22-31 : nous gérons une éventuelle exception et mémorisons sa pile dans le champ de la ligne 14.

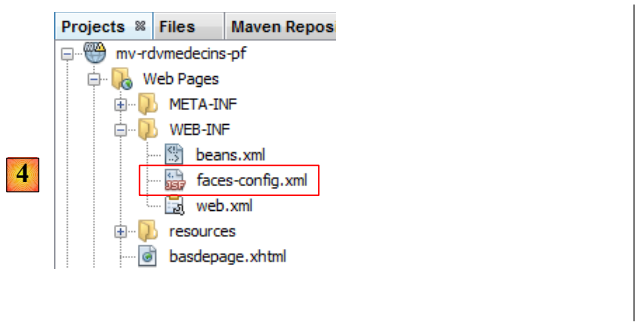
Ceci fait, le bean [Application] n'a plus d'erreurs. Regardons maintenant le bean [Form] [1], [2] :





Nous supprimons toutes les lignes erronées (import et annotations) à cause de paquetages manquants et renommons [IMetier] l'interface [ImetierLocal]. Cela suffit pour éliminer toutes les erreurs [3].





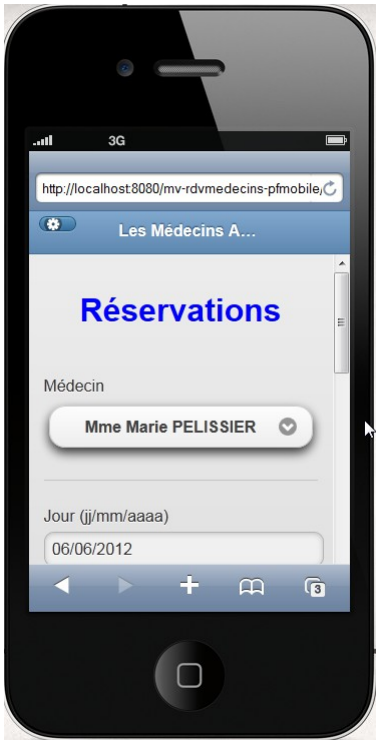
Par ailleurs, il faut ajouter dans le code du bean [Form], le getter et le setter du champ

```
1.    // bean Application
2.  private Application application;
```

Certaines des annotations supprimées dans les beans [Application] et [Form] déclaraient les classes comme étant des beans avec une certaine portée. Désormais, cette configuration est faite dans le fichier [faces-config.xml] [4] suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
    facesconfig_2_0.xsd">
9.
10.   <application>
11.     <!-- le fichier des messages -->
12.     <resource-bundle>
13.       <base-name>
14.         messages
15.       </base-name>
16.       <var>msg</var>
17.     </resource-bundle>
18.     <message-bundle>messages</message-bundle>
19.     <default-render-kit-id>PRIMEFACES_MOBILE</default-render-kit-id>
20.   </application>
21.     <!-- le bean applicationBean -->
22.   <managed-bean>
23.     <managed-bean-name>applicationBean</managed-bean-name>
24.     <managed-bean-class>beans.Application</managed-bean-class>
25.     <managed-bean-scope>application</managed-bean-scope>
26.   </managed-bean>
27.     <!-- le bean form -->
28.   <managed-bean>
29.     <managed-bean-name>form</managed-bean-name>
30.     <managed-bean-class>beans.Form</managed-bean-class>
31.     <managed-bean-scope>session</managed-bean-scope>
32.     <managed-property>
33.       <property-name>application</property-name>
34.       <value>#{applicationBean}</value>
35.     </managed-property>
36.   </managed-bean>
37. </faces-config>
```

Le portage est terminé. Nous pouvons tenter d'exécuter l'application web.



Nous laissons le lecteur tester cette nouvelle application. Nous pouvons l'améliorer légèrement pour gérer le cas où l'initialisation du bean [Application] a échoué. On sait que dans ce cas, les champs suivants ont été initialisés :

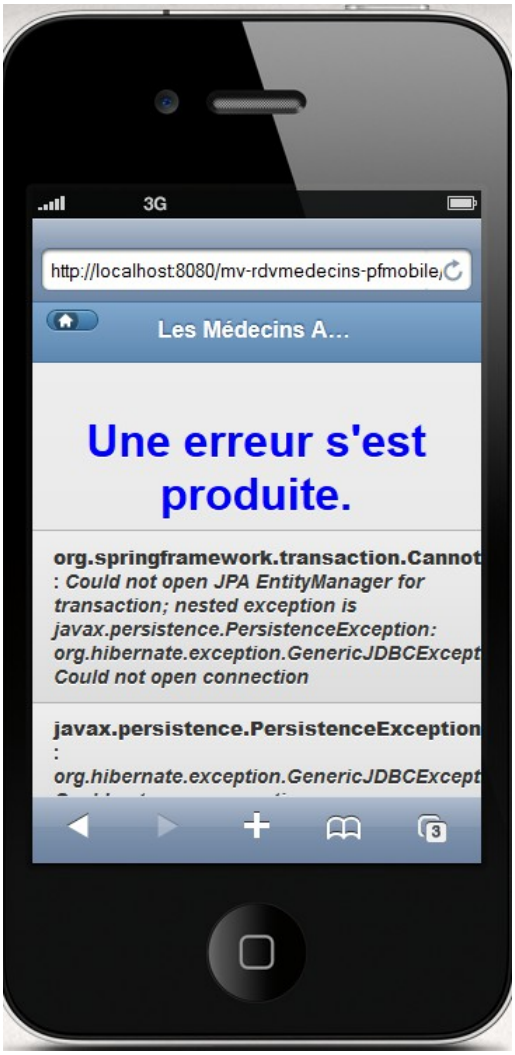
```
1.    // erreurs
2.    private List<Erreur> erreurs = new ArrayList<Erreur>();
3.  private Boolean erreur = false;
```

Ce cas peut être prévu dans la méthode *init* du bean [Form] :

```
@PostConstruct
  private void init() {

    // l'initialisation s'est-elle bien passée ?
    if (application.getErreur()) {
      // on récupère la liste des erreurs
      erreurs = application.getErreurs();
      // la vue des erreurs est affichée
      setForms(false, false, true);
    }

    // on met les médecins et les clients en cache
    ...
  }
```

- ligne 5 : si le bean [Application] s'est mal initialisé,
- ligne 7 : on récupère la liste des erreurs,
- ligne 9 : et on affiche la page d'erreur.

Ainsi, si on arrête le SGBD MySQL et qu'on relance l'application, on reçoit maintenant la page suivante :

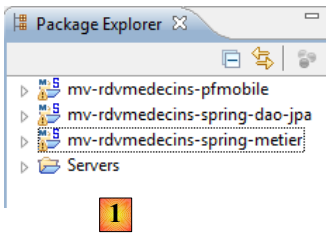


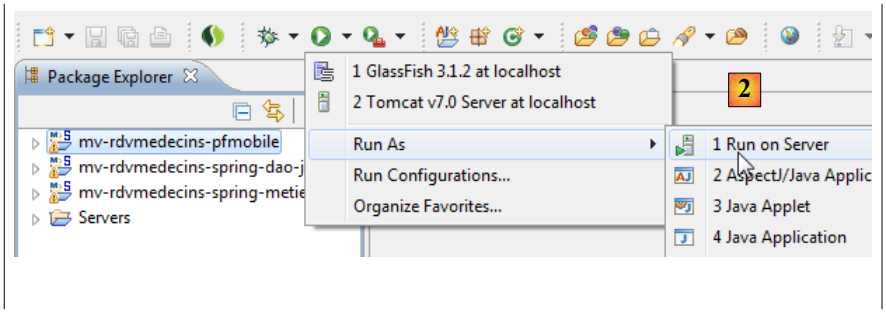
10.2 Conclusion

Le portage de l'application Primefaces mobile/ EJB / Glassfish vers un environnement Primefaces mobile / Spring / Tomcat s'est révélé simple. Le problème de la fuite mémoire signalée dans l'étude de l'application JSF / Spring / Tomcat (paragraphe 4.3.5, page 271) demeure. On le résoudra de la même façon.

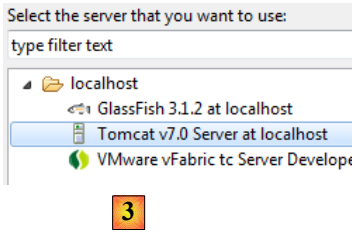
10.3 Tests avec Eclipse

Nous importons les projets Maven dans Eclipse [1] :





Nous exécutons le projet web [2].



Nous choisissons le serveur Tomcat [3]. La page d'accueil de l'application est alors affichée dans le navigateur interne d'Eclipse [4].

10.4 Tests avec mobile

Pour tester l'application sur un mobile, on procèdera comme il a été indiqué au paragraphe 8.5.6, page 376. Voici des photos de l'application :


SAMSUNG
21:28
Les Médecins A...
Mme Marie PELISSIER
12 juin 2012
08h:20-08h:40
Client
Mr Jules MARTIN
Valider

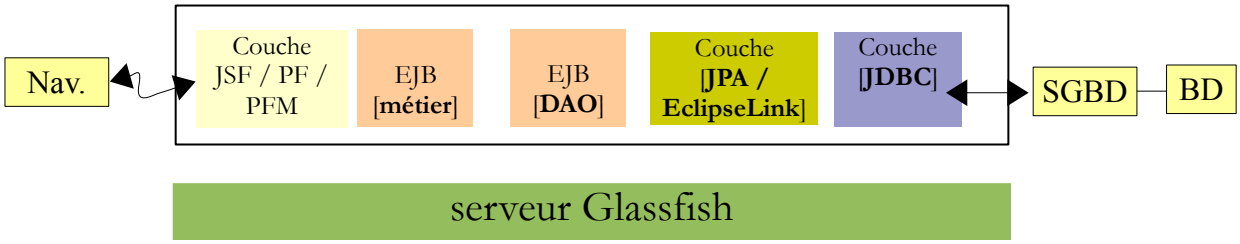

SAMSUNG
21:29
Les Médecins A...
Configuration
Langue
Français
Anglais
Valider

11 Conclusion

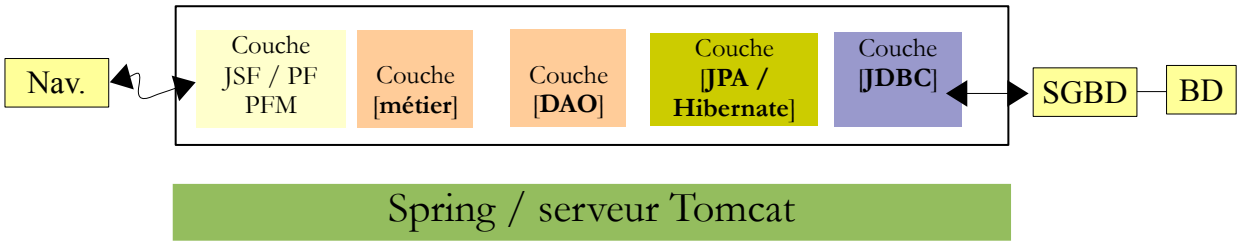
Récapitulons le travail fait dans ce document. Nous avons présenté trois frameworks web : Java Server Faces, Primefaces et Primefaces mobile et nous les avons mis en oeuvre dans six applications Java EE :

- 01 : JSF / EJB3 / Glassfish,
- 02 : JSF / Spring / Tomcat,
- 03 : PF / EJB3 / Glassfish,
- 04 : PF / Spring / Tomcat,
- 05 : PFM / EJB3 / Glassfish,
- 06 : PFM / Spring / Tomcat.

L'architecture de l'application exemple était la suivante dans un environnement EJB / Glassfish :



ou bien celle-ci dans un environnement Spring / Tomcat :



Nous avons utilisé une architecture en couches dès la première version. Celle-ci a été portée dans toutes les versions suivantes. Enfin la couche [web] a été implémentée successivement avec les frameworks Java Server Faces, Primefaces et Primefaces mobile. Elle respecte le modèle MVC (Modèle Vue Contrôleur).

Ce fut un long travail qui peut être amélioré. A la relecture j'ai découvert que certains choix faits n'étaient pas forcément les meilleurs. Je laisse au lecteur le soin de s'approprier ce document pour le dépasser ensuite.

Table des matières