Introduction à STRUTS
par la pratique
- déc 2003, rév août 2004 -

serge.tahe@istia.univ-angers.fr

Mots clés : méthodologie STRUTS, programmation WEB, servlets, pages JSP, serveur web TOMCAT

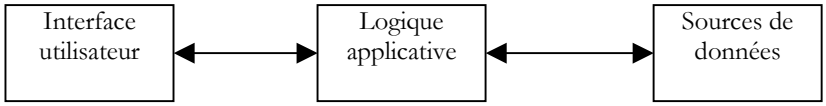
I - Généralités

I.1 Objectifs

On se propose ici de découvrir une méthode de développement appelée **STRUTS**. Jakarta Struts est un projet de l'Apache Software Foundation (www.apache.org) qui a pour but de fournir un cadre standard de développement d'applications web en Java respectant l'architecture dite MVC (Modèle-Vue-Contrôleur).

I.2 Le modèle MVC

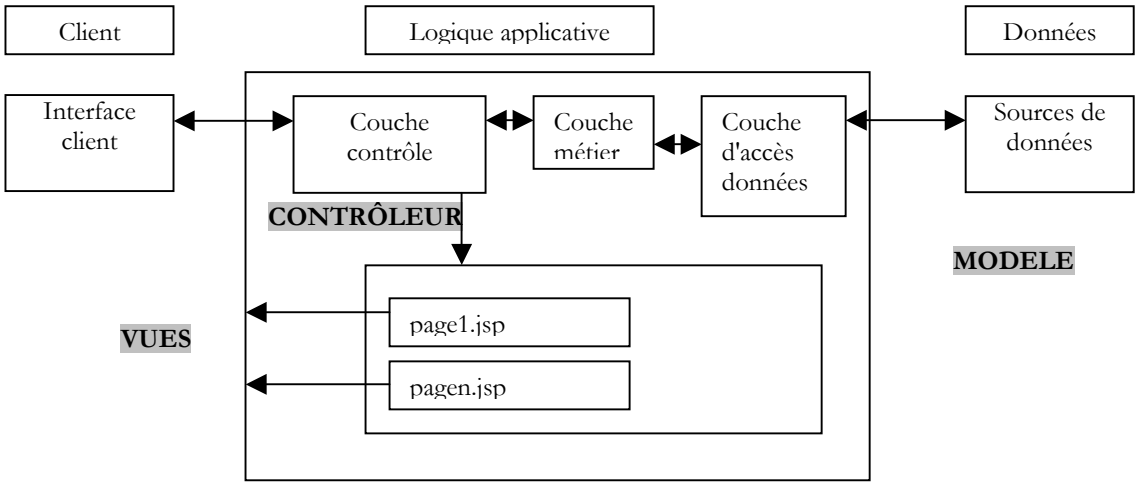
Le modèle MVC cherche à séparer les couches présentation, traitement et accès aux données. Une application web respectant ce modèle pourrait être architecturée de la façon suivante :



Une telle architecture est appelée 3-tiers ou à 3 niveaux :

- l'interface utilisateur est le V (la vue)
- la logique applicative est le C (le contrôleur)
- les sources de données sont le M (Modèle)

L'**interface utilisateur** est souvent un navigateur web mais cela peut être également une application autonome qui via le réseau envoie des requêtes HTTP au service web et met en forme les résultats que celui-ci lui renvoie. La **logique applicative** est constituée des scripts traitant les demandes de l'utilisateur. La **source de données** est souvent une base de données mais cela peut être aussi de simples fichiers plats, un annuaire LDAP, un service web distant,... Le développeur a intérêt à maintenir une grande indépendance entre ces trois entités afin que si l'une d'elles change, les deux autres n'aient pas à changer ou peu. Nous verrons que ce modèle de base donne naissance à des architectures présentant des couches plus fines que celles définies ci-dessus. Lorsqu'on cherche à appliquer ce modèle avec des servlets et pages JSP, on obtient l'architecture suivante :



Dans le bloc [Logique Applicative], on distingue

- le bloc [Couche de contrôle] est composé d'une ou plusieurs servlets et de classes annexes. Elle assure l'interface avec le client :
  - le client fait une demande
  - la couche contrôle la fait exécuter par la couche métier
  - selon le résultat de cette exécution, la couche contrôle envoie telle ou telle vue en réponse au client.

  Le rôle de la couche de contrôle s'arrête à cela. On ne doit pas y trouver des classes qui devraient relever de la couche métier.
- le bloc [Couche métier] regroupe les classes Java nécessaires à la logique métier. Cette couche est indépendante de toute forme d'interface avec l'utilisateur. Ainsi elle doit être utilisable aussi bien avec une interface console, une interface web, une interface de client riche. Elle doit ainsi pouvoir être testée en-dehors de l'interface web et notamment avec une interface console.

- le bloc [Couche d'accès aux données] qui regroupe les classes Java capables de fournir les données nécessaires à la couche métier, souvent des données persistantes (BD, fichiers,...). Ce bloc doit être indépendant de la couche métier et doit pouvoir être testé indépendamment de celle-ci par exemple avec une application console indépendante.
- le bloc [Couche de présentation] formé des pages JSP qui constituent les vues de l'application.

Les quatre entités précédentes définissent assez bien la structure de nombreuses applications web en quatre couches.

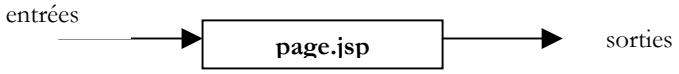
I.3 Une démarche de développement MVC à l'aide de servlets et pages JSP

Nous définissons maintenant une démarche possible pour le développement d'applications web java respectant le modèle MVC précédent. Elle ne doit être prise que comme un exemple de ce qui pourrait être fait.

1. On commencera par définir toutes les vues de l'application. Celles-ci sont les pages Web présentées à l'utilisateur. On se placera donc du point de vue de celui-ci pour dessiner les vues. On distingue trois types de vues :
   - le formulaire de saisie qui vise à obtenir des informations de l'utilisateur. Celui-ci dispose en général d'un bouton pour envoyer les informations saisies au serveur.
   - la page de réponse qui ne sert qu'à donner de l'information à l'utilisateur. Celle-ci dispose souvent d'un lien permettant à l'utilisateur de poursuivre l'application avec une autre page.
   - la page mixte : la servlet a envoyé au client une page contenant des informations qu'elle a générées. Cette même page va servir au client pour fournir à la servlet d'autres informations.

2. Chaque vue donnera naissance à une page JSP. Pour chacune de celles-ci :
   - on dessinera l'aspect de la page
   - on déterminera quelles sont les parties dynamiques de celle-ci :
     - les informations à destination de l'utilisateur qui devront être fournies par la servlet en paramètres à la vue JSP
     - les données de saisie qui devront être transmises à la servlet pour traitement. Celles-ci devront faire partie d'un formulaire HTML.

3. On pourra schématiser les E/S de chaque vue



   - les entrées sont les données que devra fournir la servlet à la page JSP souvent dans la requête (Request), la session (Session) ou le contexte de l'application (ServletContext).
   - les sorties sont les données que devra fournir la page JSP à la servlet. Elles font partie d'un formulaire HTML et la servlet les récupèrera par une opération du type **request.getparameter(...)**.

4. On écrira le code Java/JSP de chaque vue. Il aura le plus souvent la forme suivante :

```
<%@ page ... %> // importations de classes le plus souvent
<%!
  // variables d'instance de la page JSP (=globales)
  // nécessaire que si la page JSP a des méthodes partageant des variables (rare)
  ...
%>
<%
  // récupération des données envoyées par la servlet
  // soit dans la requête (request) soit dans la session (session)
  ...
%>

<html>
...
   // on cherchera ici à minimiser le code java
</html>
```

5. On peut alors passer aux premiers tests. La méthode de déploiement expliquée ci-dessous est propre au serveur Tomcat :

   - le contexte de l'application doit être créé dans le fichier **server.xml** de Tomcat. On pourra commencer par tester ce contexte. Soit **C** ce contexte et **DC** le dossier associé à celui-ci. On construira un fichier statique **test.html** qu'on placera dans le dossier **DC**. Après avoir lancé Tomcat, on demandera avec un navigateur l'URL **http://localhost:8080/DC/test.html**.
   - chaque page JSP peut être testée. Si une page JSP s'appelle **formulaire.jsp**, on demandera avec un navigateur l'URL **http://localhost:8080/DC/formulaire.jsp**. La page JSP attend des valeurs de la servlet qui l'appelle. Ici on l'appelle directement donc elle ne recevra pas les paramètres attendus. Afin que les tests soient néanmoins possibles on

initialisera soi-même dans la page JSP les paramètres attendus avec des constantes. Ces premiers tests permettent de vérifier que les pages JSP sont syntaxiquement correctes.

6. On écrit ensuite le code de la servlet. Celle-ci a deux méthodes bien distinctes :
   - la méthode **init** qui sert à :
     - récupérer les paramètres de configuration de l'application dans le fichier **web.xml** de celle-ci
     - éventuellement créer des instances de classes métier qu'elle sera amenée à utiliser ensuite
     - gérer une éventuelle liste d'erreurs d'initialisation qui sera renvoyée aux futurs utilisateurs de l'application. Cette gestion d'erreurs peut aller jusqu'à l'envoi d'un mél à l'administrateur de l'application afin de le prévenir d'un dysfonctionnement
   - la méthode **doGet** ou **doPost** selon la façon dont la servlet reçoit ses paramètres de ses clients. Si la servlet gère plusieurs formulaires, il est bon que chacun d'entre-eux envoie une information qui l'identifie de façon unique. Cela peut se faire au moyen d'un attribut caché dans le formulaire du type **<input type="hidden" name="action" value="...">**. La servlet peut commencer par lire la valeur de ce paramètre puis déléguer le traitement de la requête à une méthode privée interne chargée de traiter ce type de requête.
   - On évitera au maximum de mettre du code métier dans la servlet. Elle n'est pas faite pour cela. La servlet est une sorte de chef d'équipe (contrôleur) qui reçoit des demandes de ses clients (clients web) et qui les fait exécuter par les personnes les plus appropriées (les classes métier). Lors de l'écriture de la servlet, on déterminera l'interface des classes métier à écrire (constructeurs, méthodes). Cela si ces classes métier sont à construire. Si elles existent déjà, alors la servlet doit s'adapter à l'interface existante.
   - le code de la servlet sera compilé.

7. On écrira le squelette des classes métier nécessaires à la servlet. Par exemple, si la servlet utilise un objet de type **proxyArticles** et que cette classe doit avoir une méthode **getCodes** rendant une liste (ArrayList) de chaînes de caractères, on peut se contenter dans un premier temps d'écrire :

```
public ArrayList getCodes(){
  String[] codes= {"code1","code2","code3"};
  ArrayList aCodes=new ArrayList();
  for(int i=0;i<codes.length;i++){
    aCodes.add(codes[i]);
  }
    return aCodes;
}
```

8. On peut alors passer aux tests de la servlet.

   - le fichier de configuration **web.xml** de l'application doit être créé. Il doit contenir toutes les informations attendues par la méthode **init** de la servlet (<init-param>). Par ailleurs, on fixe l'URL au travers de laquelle sera atteinte la servlet principale (<servlet-mapping>).
   - toutes les classes nécessaires (servlet, classes métier) sont placées dans **WEB-INF/classes**.
   - toutes les bibliothèques de classes (.jar) nécesaires sont placées dans **WEB-INF/lib.** Ces bibliothèques peuvent contenir des classes métier, des pilotes JDBC, ...
   - les vues JSP sont placées sous la racine de l'application ou un dossier propre. On fait de même pour les autres ressources (html, images, son, vidéos, ...)
   - ceci fait, l'application est testée et les premières erreurs corrigées. A la fin de cette phase, l'architecture de l'application est opérationnelle. Cette phase de test peut être délicate sachant qu'on n'a pas d'outil de débogage avec Tomcat. il faudrait pour cela que Tomcat soit lui-même intégré dans un outil de développement (Jbuilder Developer, Sun One Studio, ...). On pourra s'aider d'instructions **System.out.println(".....")** qui écrivent dans la fenêtre de Tomcat. La première chose à vérifier est que la méthode **init** récupère bien toutes les données provenant du fichier **web.xml**. On pourra pour cela écrire la valeur de celles-ci dans la fenêtre de Tomcat. On vérifiera de la même façon que les méthodes **doGet** et **doPost** récupèrent correctement les paramètres des différents formulaires HTML de l'application.

9. On écrit les classes métier dont a besoin la servlet. On a là en général le développement classique d'une classe Java le plus souvent indépendante de toute application web. Elle sera tout d'abord testée en dehors de cet environnement avec une application console par exemple. Lorsqu'une classe métier a été écrite, on peut l'intégrer dans l'architecture de déploiement de l'application web et tester sa correcte intégration dans celui-ci. On procèdera ainsi pour chaque classe métier.

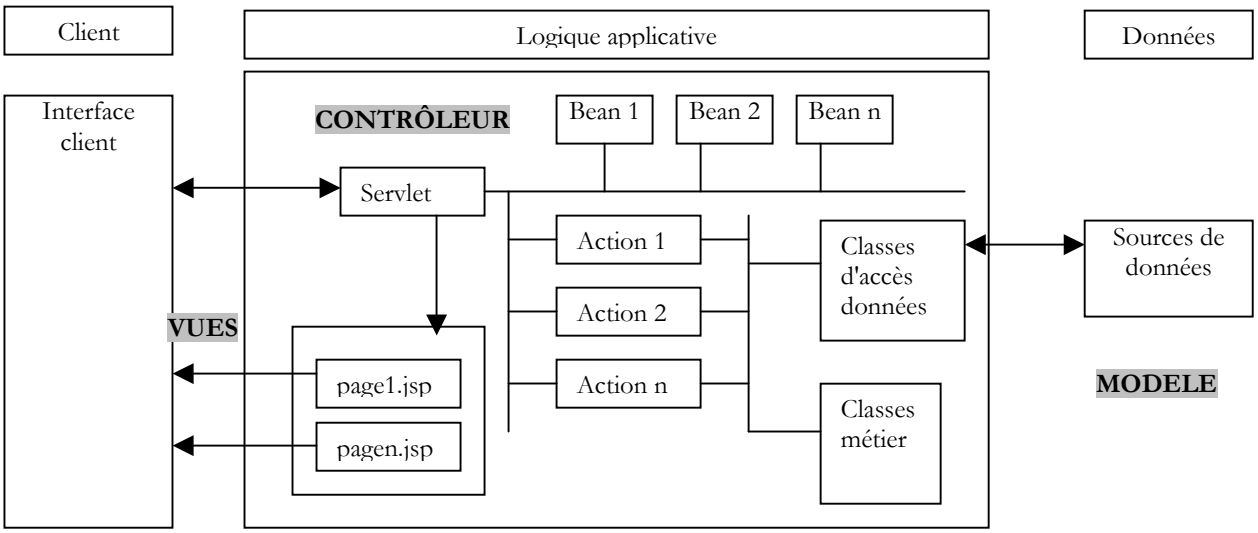
I.4 La démarche de développement STRUTS

Les créateurs de la méthodologie STRUTS ont cherché à définir une méthode de développement standard qui respecte l'architecture MVC pour les applications web écrites en java. Il y a deux aspects au projet STRUTS :

- la méthode de développement. Nous verrons qu'elle est, dans son esprit, proche de celle décrite ci-dessus pour les servlets et pages JSP

- les outils qui nous permettent d'appliquer cette méthode de développement. Ce sont des bibliothèques de classes Java que l'on trouve sur le site de la fondation Apache (www.apache.org).

L'architecture MVC utilisée par STRUTS est la suivante :



- le contrôleur est le coeur de l'application. Toutes les demandes du client transitent par lui. C'est une servlet générique fournie par STRUTS. On peut dans certains cas être amené à la dériver. Pour les cas simples, ce n'est pas nécessaire. Cette servlet générique prend les informations dont elle a besoin dans un fichier le plus souvent appelé **struts-config.xml**.
- si la requête du client contient des paramètres de formulaire, ceux-ci sont mis par le contrôleur dans un objet **Bean**. Une classe est dite de type **bean** si elle observe des règles de construction que nous verrons un peu plus loin. Les objets **bean** ainsi créés au fil du temps sont stockés dans la session ou la requête du client. Ce point est configurable. Ils n'ont pas à être recréés s'ils l'ont déjà été.
- dans le fichier de configuration **struts-config.html**, à chaque URL devant être traitée par programme (ne correspondant donc pas à une vue JSP qu'on pourrait demander directement) on associe certaines informations :
  - le **nom de la classe** de type **Action** chargée de traiter la requête. Là encore, l'objet **Action** instancié peut être conservé dans la session ou la requête.
  - si l'URL demandée est paramétrée (cas de l'envoi d'un formulaire au contrôleur), le nom du **bean** chargé de mémoriser les informations du formulaire est indiqué.
- muni de ces informations fournies par son fichier de configuration, à la réception d'une demande d'URL par un client, le contrôleur est capable de déterminer s'il y a un bean à créer et lequel. Une fois instancié, le bean peut vérifier que les données qu'il a stockées et qui proviennent du formulaire, sont valides ou non. Une méthode du bean appelée **validate** est appelée automatiquement par le contrôleur. Le bean est construit par le développeur. Celui-ci met donc dans la méthode **validate** le code vérifiant la validité des données du formulaire. Si les données se révèlent invalides, le contrôleur n'ira pas plus loin. Il passera la main à une vue dont il trouvera le nom dans son fichier de configuration. L'échange est alors terminé. On notera que le développeur peut demander à ce que la validité du formulaire ne soit pas vérifiée. Il fait cela également dans le fichier **struts-config.html**. Dans ce cas, le contrôleur n'appelle pas la méthode **validate** du bean.
- si les données du bean sont correctes, ou s'il n'y a pas de vérification ou s'il n'y a pas de bean, le contrôleur passe la main à l'objet de type **Action** associé à l'URL. Il le fait en demandant l'exécution de la méthode **execute** de cet objet à laquelle il transmet la référence du **bean** qu'il a éventuellement construit. C'est ici que le développeur fait ce qu'il a à faire : il devra éventuellement faire appel à des classes métier ou à des classes d'accès aux données. A la fin du traitement, l'objet **Action** rend au contrôleur le nom de la vue qu'il doit envoyer en réponse au client.
- le contrôleur envoie cette réponse. L'échange avec le client est terminé.

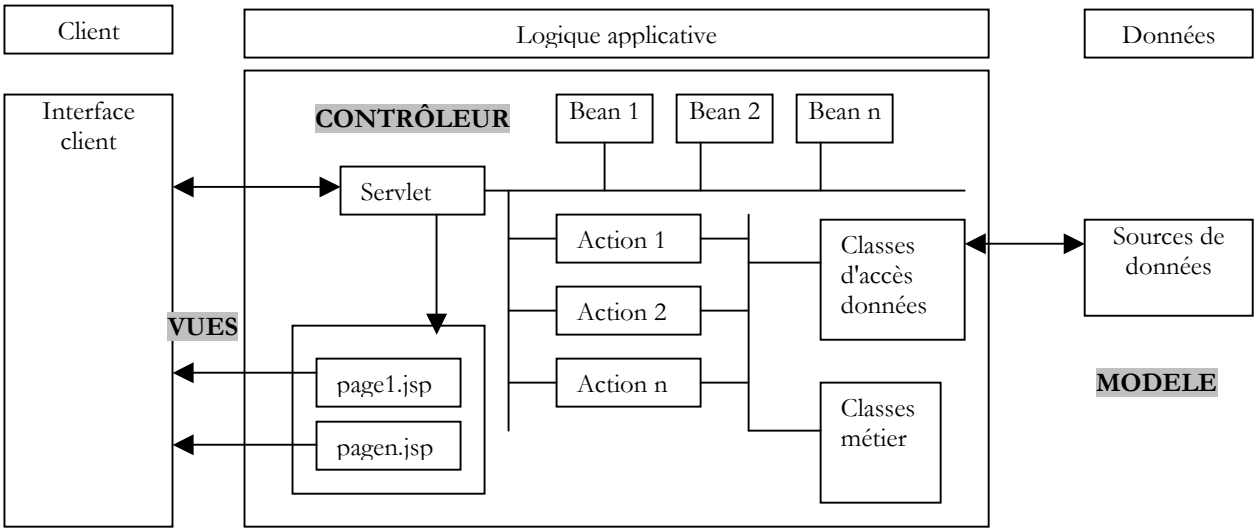
Commence à se dessiner la méthodologie de développement STRUTS :

- la définition des vues. On distingue les vues qui sont des formulaires et les autres.
  - chaque vue formulaire donne naissance à une définition dans le fichier **struts-config.xml**. On y définit les informations suivantes :
    - le nom de la classe **Bean** qui contiendra les données du formulaire ainsi que l'indication sur le fait que les données doivent être vérifiées ou non. Si elles doivent être vérifiées et qu'elles s'avèrent invalides, on doit indiquer la vue à envoyer en réponse au client dans ce cas.
    - le nom de la classe **Action** chargée de traiter le formulaire.
    - le nom de toutes les vues susceptibles d'être envoyées en réponse au client une fois que la requête a été traitée. La classe **Action** choisira l'une d'elles selon le résultat du traitement.
  - chaque vue fait l'objet d'une page JSP. Nous verrons que dans les vues notamment les vues formulaires, on utilise parfois une bibliothèque de balises propres à Struts.
- l'écriture des classes Javabean correspondant aux vues formulaires

- l'écriture des classes **Action** chargées de traiter les formulaires
- l'écriture des éventuelles classes métier ou d'accès aux données

II Les outils du développement web avec STRUTS

Reprenons le schéma type d'une application web/struts :



- l'application comporte un certain nombre de classes (servlets, beans, actions, classes métier, classes d'accès aux données) qu'il nous faut construire et compiler. Il existe d'excellents IDE Java sur le marché et nombre d'entre-eux sont gratuits (Jbuilder Foundation, Eclipse, ...).
- une application Struts s'appuie sur des bibliothèques de classes qu'il nous faudra télécharger.
- l'accès aux données est souvent un accès à une base de données. Là encore, il existe beaucoup de SGBD disponibles gratuitement (MySQL, Postgres, Firebird, Hypersonic, SQL Server, ...)
- les servlets sont exécutées dans un conteneur de servlets (Tomcat, Resin, Jetty, ...) eux-aussi disponibles gratuitement.
- l'affichage des vues nécessite un navigateur (Internet Explorer, Netscape, Mozilla, Opera, ...)

Tous ces outils sont disponibles pour les plate-formes Linux et Windows. En août 2004, ils peuvent être trouvés sur les sites suivants :

Jbuilder Foundation	http://www.borland.com/jbuilder/foundation/index.html
Eclipse	http://www.eclipse.org/
Bibliothèques Struts	http://struts.apache.org/
MySQL	http://www.mysql.com/
Postgres	http://www.postgresql.org/
Firebird	http://firebird.sourceforge.net/
Hypersonic	http://hsqldb.sourceforge.net/
SQL Server / MSDE	http://www.microsoft.com/sql/msde/downloads/download.asp
Tomcat	http://jakarta.apache.org/tomcat/
Resin	http://www.caucho.com/
Jetty	http://jetty.mortbay.org/jetty/
Netscape	http://www.netscape.com/
Mozilla	http://www.mozilla.org

Les application de ce document ont été construites et testées sur une plate-forme windows. Nous présentons, pour cette plate-forme, l'installation de quelques-uns des produits ci-dessus : Struts et Tomcat.

II.1 La bibliothèque Struts

Le projet STRUTS est l'un des projets de l'Apache Software Foundation. Plusieurs de ces projets sont réunis sous la dénomination **Jakarta** et sont disponibles à l'URL **http://jakarta.apache.org** :

http://jakarta.apache.org/ Search

	Frameworks and Engines
Cactus	Cactus is a simple test framework for unit testing server-side Java code (servlets, EJBs, tag libraries, filters, ...).
Gump	Gump is a tool for Continuous Integration. It builds all of Jakarta and other products in Jakarta Related Projects against the HEAD in CVS, nightly.
Lucene	Jakarta Lucene is a high-performance, full-featured text search engine written entirely in Java.
Struts	A model-view-controller framework for constructing web applications with servlets and JavaServer Pages
Tapestry	Tapestry is a web application framework based on highly reusable components within a very pure Model-View-Controller pattern.

Il est conseillé de lire cette page. De nombreux projets présentent de l'intérêt pour les développeurs java. Si nous suivons ci-dessus le lien **Struts**, nous arrivons à la page d'accueil du projet :

http://jakarta.apache.org/struts/index.html Search

The Jakarta Site - The Jakarta Project -- Ja... | The Apache Struts Web Application Frame...

Download

Binaries
Source Code

Development

home page is at http://jakarta.apache.org/struts.

Struts is a volunteer project and all support for the framework is provided by unpaid volunteers. This documentation bundle and the mailing lists are the primary ways to learn how to use Struts. The next few pages are devoted to helping you understand what resources are available to you. Since Struts is a volunteer project, and our resources are limited, it is important that we first help you help yourself.

Là aussi, il est conseillé de lire la page d'accueil. Pour télécharger les bibliothèques java de Struts, nous suivons le lien **Binaries** ci-dessus :

http://jakarta.apache.org/site/binindex.cgi

-- Ja... | The Jakarta Site - Binary Downloads

Struts KEYS

- 1.1 zip PGP MD5
- 1.1 tar.gz PGP MD5

Pour le monde Windows, on utilisera le lien 1.1.zip et 1.1.tar.gz pour le monde Unix (nov 2003). Une fois le fichier 1.1.zip décompressé, nous obtenons l'arborescence suivante :

Adresse E:\struts11

struts11

Sélectionnez un élément pour obtenir une description.

Nom
- contrib
- lib
- webapps
- INSTALL
- LICENSE
- README

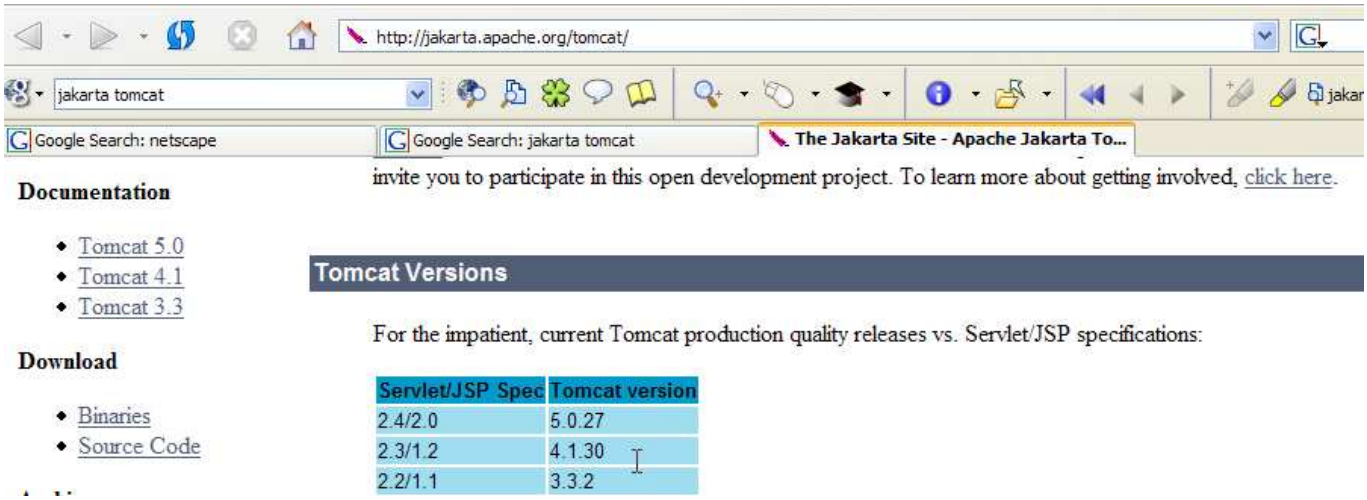
On trouvera dans cette arborescence les bibliothèques des classes java nécessaires au développement STRUTS. Celles-ci sont dans des fichiers .jar, fichiers analogues aux fichiers .zip. On peut les ouvrir avec les mêmes utilitaires. La plupart des bibliothèques nécessaires sont dans le dossier **lib** ci-dessus :

Outre les bibliothèques de classes .jar, on trouve des fichiers .dtd (Document Type Definition) qui contiennent des règles de validité de fichiers XML. Un fichier XML peut dans son contenu référencer un tel fichier DTD. Le programme (appelé **parseur**) qui analyse le contenu du fichier XML utilisera les règles de validité trouvées dans le fichier DTD référencé, pour déterminer si le fichier XML est syntaxiquement correct. Ainsi par exemple, le fichier **struts-config_1_1.dtd** fixe les règles de construction du fichier de configuration **struts-config.xml** pour la version 1.1 de Struts.

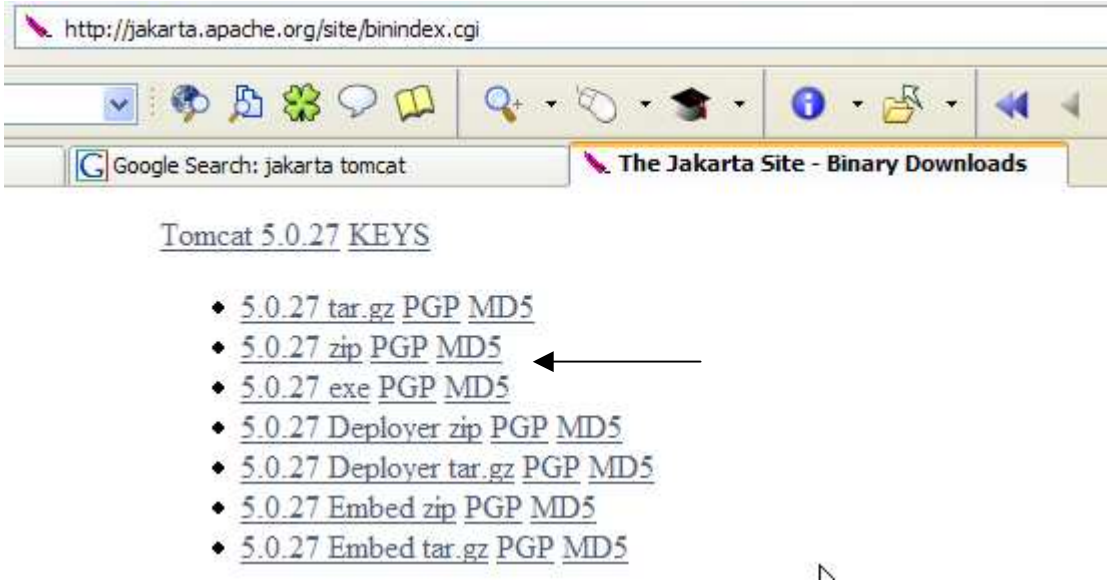
Dans la suite, le dossier d'installation des outils Struts sera désigné par **<struts>**.

II.2 Le conteneur de servlets Tomcat 5

Pour exécuter des servlets, il nous faut un conteneur de servlets. Nous utiliserons Tomcat 5 disponible à l'url **http://jakarta.apache.org/tomcat/** :

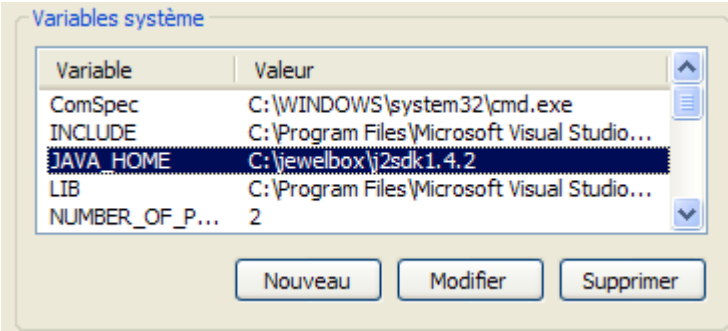


Pour télécharger le produit, on suivra le lien [Binaries] ci-dessus. On arrive alors à une page rassemblant tous les binaires des différents sous-projets du projet Jakarta de l'Apache Software Foundation. Nous choisissons celui de Tomcat 5 :

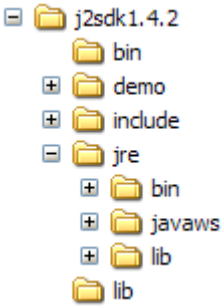


On pourra prendre le .zip ou le .exe destiné à la plate-forme windows. Le .exe est conseillé car il vient avec un assistant d'installation. Tomcat est une application Java et a donc besoin d'une machine virtuelle java (JVM) sur la plate-forme d'installation. Comme beaucoup d'applications Java, Tomcat utilise la variable d'environnement **JAVA_HOME** si elle est présente. Celle-ci doit désigner le dossier d'installation d'un JDK (Java Development Kit) ou d'un JRE (Java Runtime Environment). Sur une machine Windows XP une variable d'environnement peut être créée de la façon suivante :

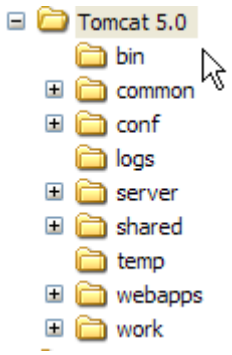
Menu Démarrer -> Panneau de configuration -> Système -> Onglet [Avancé] -> Bouton [Variables d'environnement] ->



Ici JAVA_HOME désigne un JDK de Sun dont l'arborescence est la suivante :



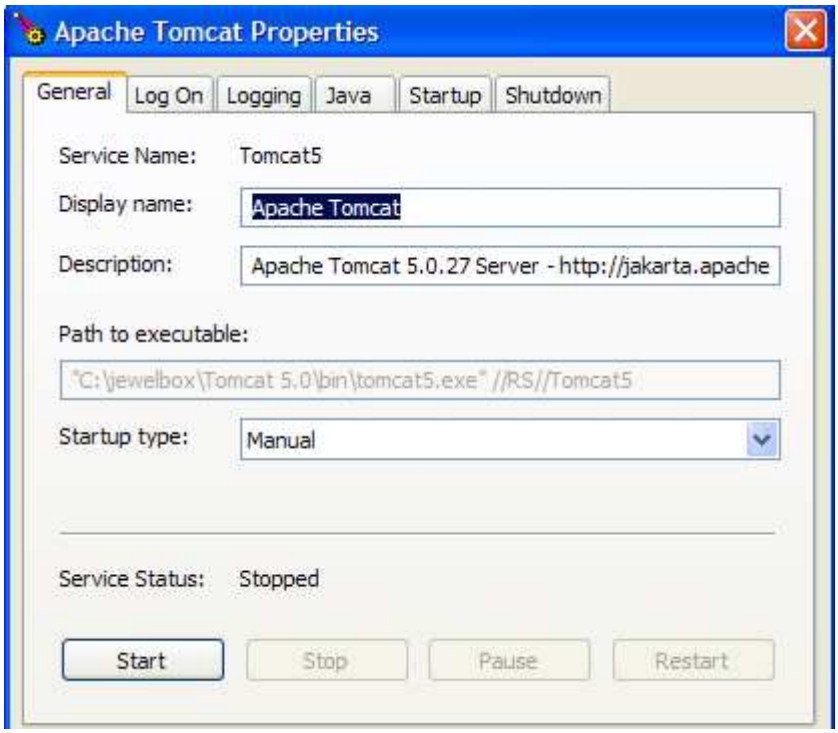
Tout JDK récent de Sun fera l'affaire. La variable d'environnement peut nécessiter un redémarrage du système pour être prise en compte. Une fois la variable d'environnement JAVA_HOME définie, l'installation de Tomcat peut être faite. Au cours de celle-ci, l'installateur va demander le dossier contenant la JVM. Il présente dans une liste déroulante les dossiers susceptibles de contenir une JVM. Ces informations sont obtenues soit par la variable JAVA_HOME soit par des clés de registres. La JVM et correspondant à la variable JAVA_HOME devrait être dans la liste des JVM présentées. Ce sera la seule s'il n'y a pas d'autres JVM sur la machine. On sélectionnera la JVM de son choix. La gestion de Tomcat via le web nécessite de s'authentifier via une page de login/mot de passe. Deux utilisateurs spéciaux sont appelés **admin** et **manager**. Au cours de l'installation de Tomcat, il nous est demandé de fixer le mot de passe de l'utilisateur **admin**. Dans La suite, nous supposerons que ce mot de passe est **admin**. L'installation de Tocat se fait dans un dossier choisi par l'utilisateur, que nous appellerons désormais **<tomcat>**. L'arborescence de ce dossier pour la version Tomcat 5.0.27 ci-dessus est la suivante :



L'installation de Tomcat 5.0.27 a amené un certain nombre de raccourcis dans le menu [Démarrer]. Nous utilison le lien [Monitor] ci-dessous pour lancer l'outil d'arrêt/démarrage de Tomcat :

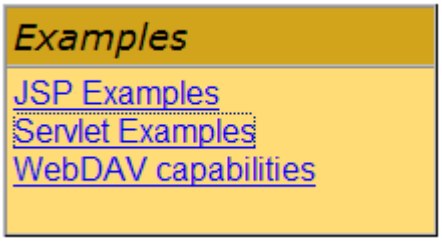
Une icône est insérée dans la barre des tâches en bas à droite de l'écran :

Le moniteur de Tomcat est activé par un double-clic sur cette icône :

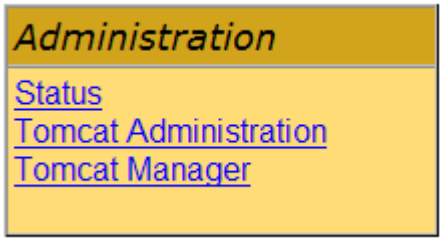


Les boutons Start - Stop - Pause - Restart nous permettent de lancer - arrêter - relancer le serveur. Nous lançons le serveur par [Start] puis avec un navigateur nous demandons l'url http://localhost:8080. Nous devons obtenir une page analogue à la suivante :

On pourra suivre les liens ci-dessous pour vérifier la correcte installation de Tomcat :



Tous les liens de la page [http://localhost:8080] présentent un intérêt et le lecteur est invité à les explorer. Nous aurons l'occasion de revenir sur les liens permettant de gérer les applications web déployées au sein du serveur :



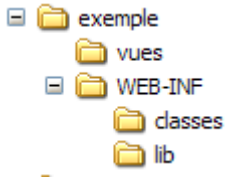
II.3 Déploiement d'une application web au sein du serveur Tomcat

Une application web doit suivre certaines règles pour être déployée au sein d'un conteneur de servlets. Soit <webapp> le dossier d'une application web. Une application web est composée de :

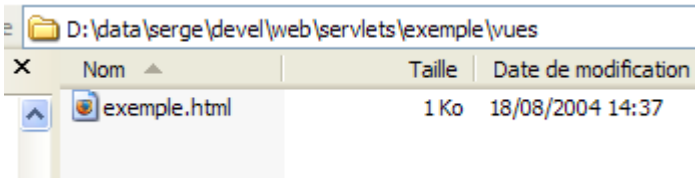
`classes`	elles seront placées dans le dossier <webapp>\WEB-INF\classes
`archives java`	elles seront placées dans le dossier <webapp>\WEB-INF\lib
`vues, ressources (.jsp, .html, ...)`	elles seront placées dans le dossier <webapp> ou des sous-dossiers mais en général en-dehors du sous-dossier WEB-INF

L'application web est configurée par un fichier XML : **<webapp>\WEB-INF\web.xml**.

Considérons l'application web dont l'arborescence serait la suivante :



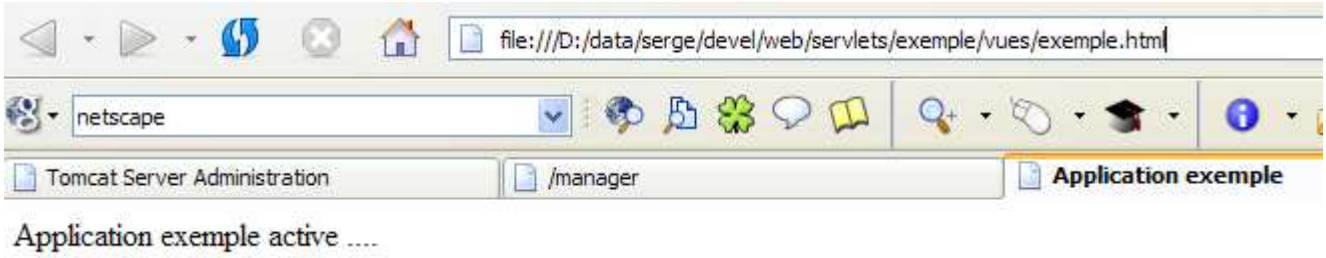
Les dossiers [classes] et [lib] sont ici vides. L'application n'ayant pas de classes, le fichier WEB-INF\web.xml est inutile et n'est donc pas présent. Le dossier [vues] contient un fichier html statique :



dont le contenu est le suivant :

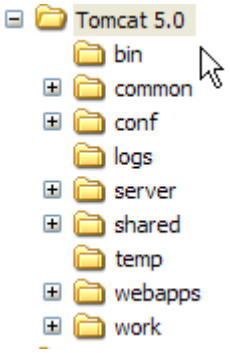
```
<html>
  <head>
    <title>Application exemple</title>
  </head>
  <body>
    Application exemple active ....
  </body>
</html>
```

Si on charge ce fichier dans un navigateur, on obtient la page suivante :

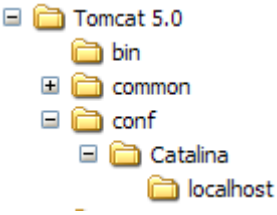


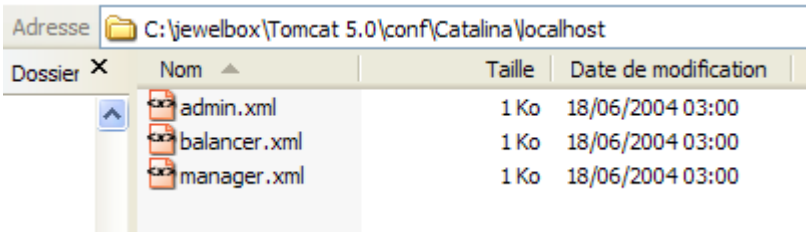
L'URL affichée par le navigateur montre que la page n'a pas été délivrée par un serveur web mais directement chargée par le navigateur. Nous voulons maintenant qu'elle soit disponible via le serveur web Tomcat.

Revenons sur l'arborescence du répertoire <tomcat> :

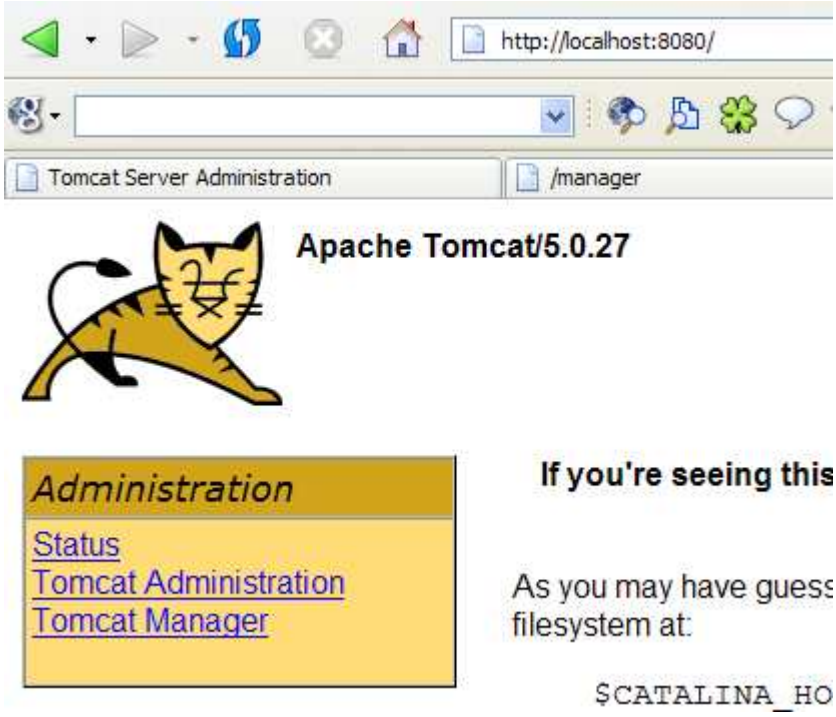


La configuration des applications web déployées au sein du serveur Tomcat se fait à l'aide de fichiers XML placés dans le dossier <tomcat>\conf\Catalina\localhost :





Ces fichiers XML peuvent être créés à la main car leur structure est très simple. Plutôt que d'adopter cette démarche, nous allons utiliser les outils web que nous offre Tomcat. Sur sa page d'entrée http://localhost:8080, le serveur nous offre des liens pour l'administrer :



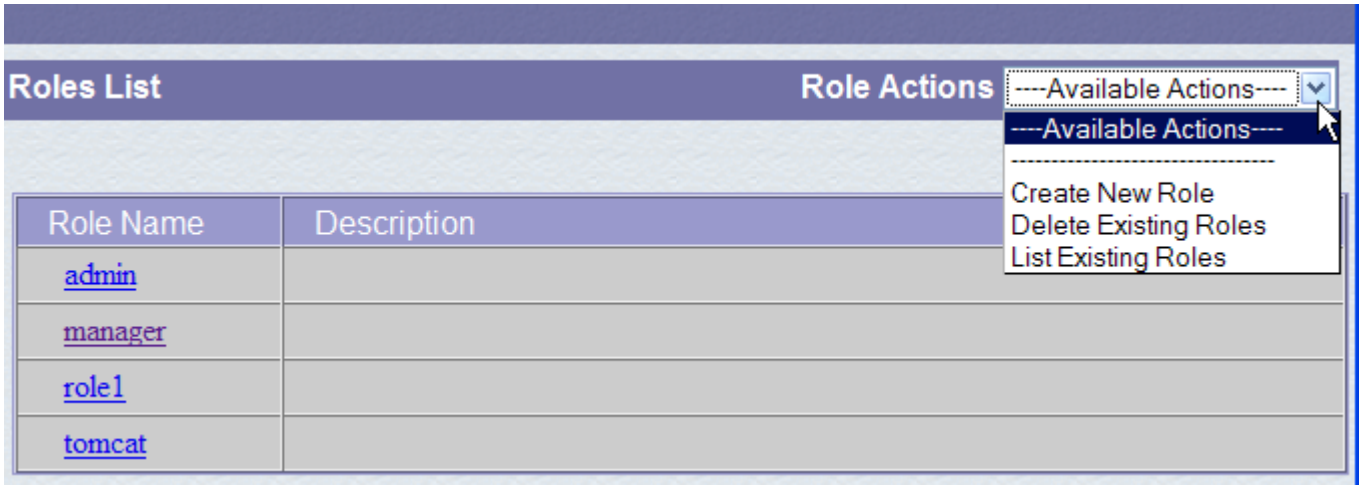
Le lien [Tomcat Administration] nous offre la possibilité de configurer les ressources mises par Tomcat à la disposition des applications web déployée en son sein. Un exemple classique est un pool de connexions à une base de données. Suivons le lien. Nous obtenons une page d'identification :

Ici, il faut redonner les informations que nous avons donnée au cours de l'installation de Tomcat. Dans notre cas, nous donnons le couple admin/admin. Le bouton [Login] nous amène à la page suivante :

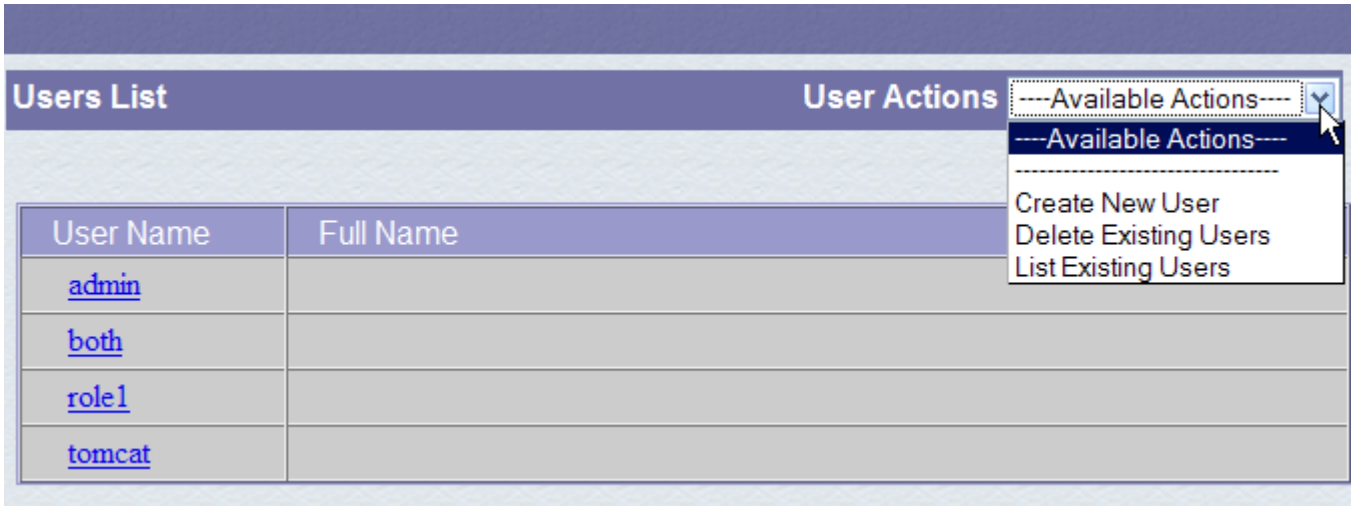
Cette page permet à l'administrateur de Tomcat de définir

- des sources de données (Data Sources),
- les informations nécessaires à l'envoi de courrier (Mail Sessions),
- des données d'environnement accessibles à toutes les applications (Environment Entries),
- de gérer les utilisateurs gérant Tomcat (Users),
- de gérer des groupes d'utilisateurs (Groups),
- de définir des rôles (= ce que peut faire ou non un utilisateur),
- de définir les caractéristiques des applications web déployées par le serveur (Service Catalina)

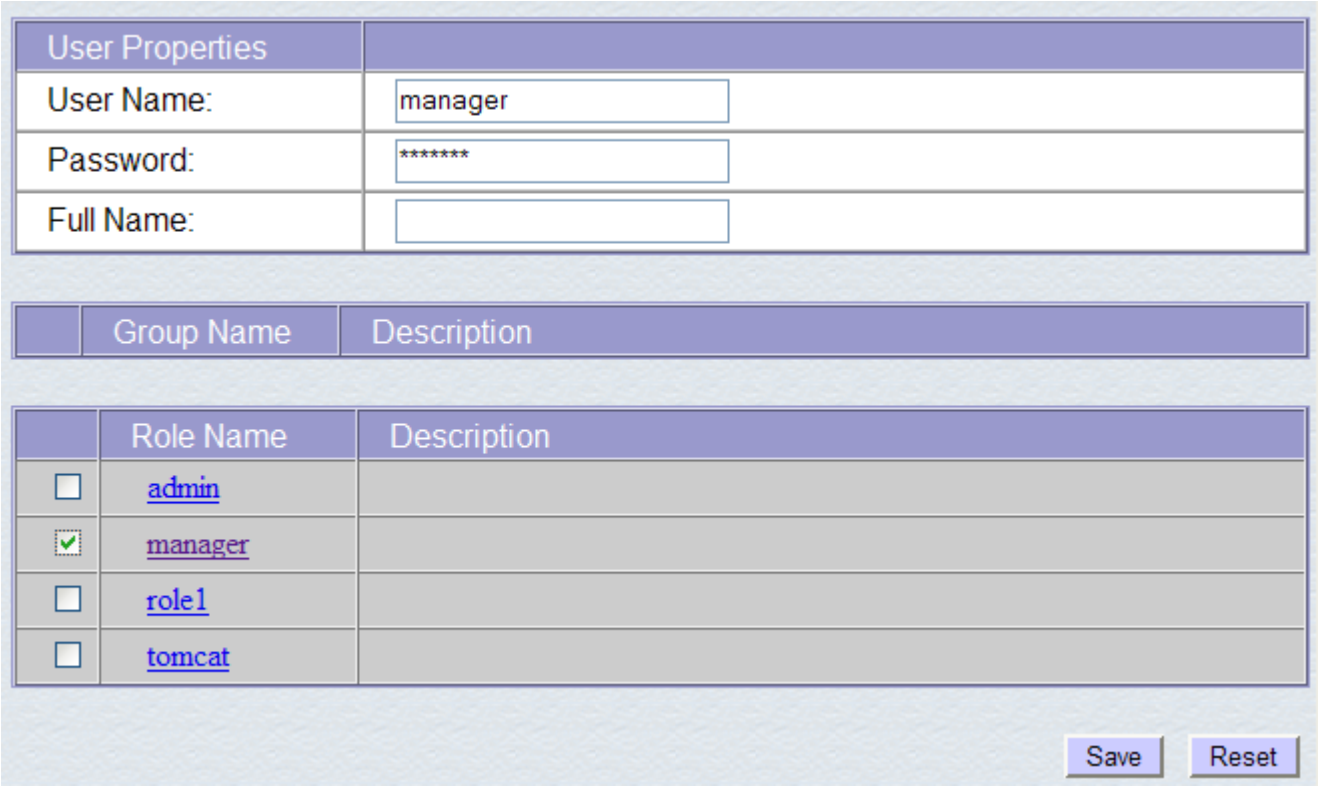
Suivons le lien [Roles] ci-dessus :



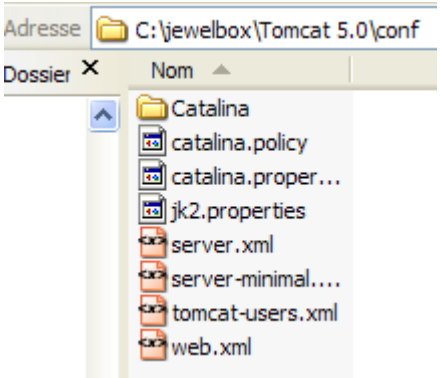
Un rôle permet de définir ce que peut faire ou ne pas faire un utilisateur ou un groupe d'utilisateurs. On associe un rôle à certains droits. Chaque utilisateur est associé à un ou plusieurs rôles et dispose des droits de ceux-ci. Le rôle [manager] ci-dessous donne le droit de gérer les applications web déployées dans Tomcat (déploiement, démarrage, arrêt, déchargement). Nous allons créer un utilisateur [manager] qu'on associera au rôle [manager] afin de lui permettre de gérer les applications de Tomcat. Pour cela, nous suivons le lien [Users] de la page d'administration :



Nous voyons qu'un certain nombre d'utilisateurs existent déjà. Nous utilisons l'option [Create New User] pour créer un nouvel utilisateur :



Nous donnons à l'utilisateur **manager** le mot de passe **manager** et nous lui attribuons le rôle **manager**. Nous utilisons le bouton [Save] pour valider cet ajout. Ce nouvel utlisateur va être ajouté dans le fichier <tomcat>\conf\tomcat-users.xml :



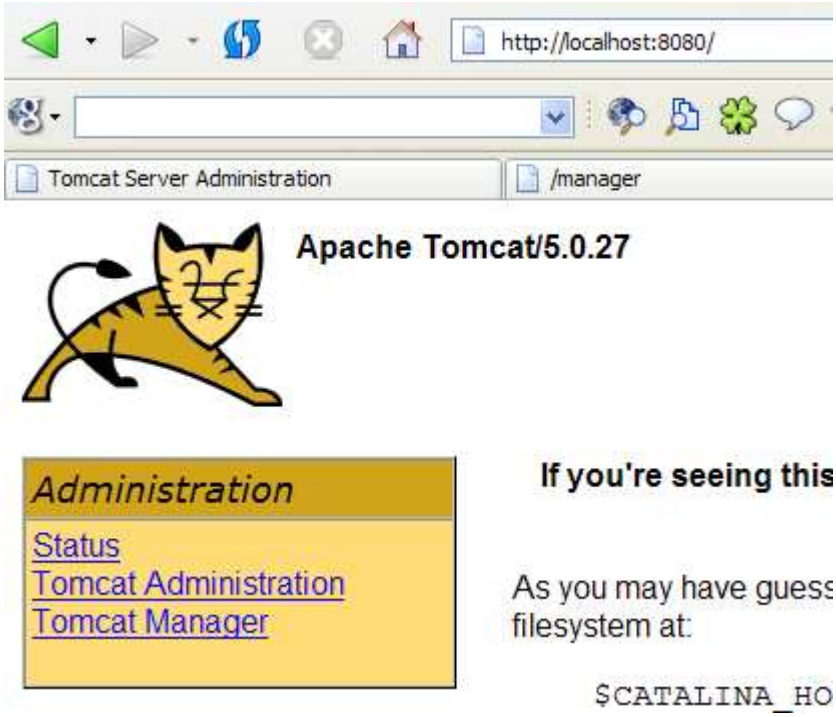
dont le contenu est le suivant :

```
<?xml version='1.0' encoding='utf-8'?>
```

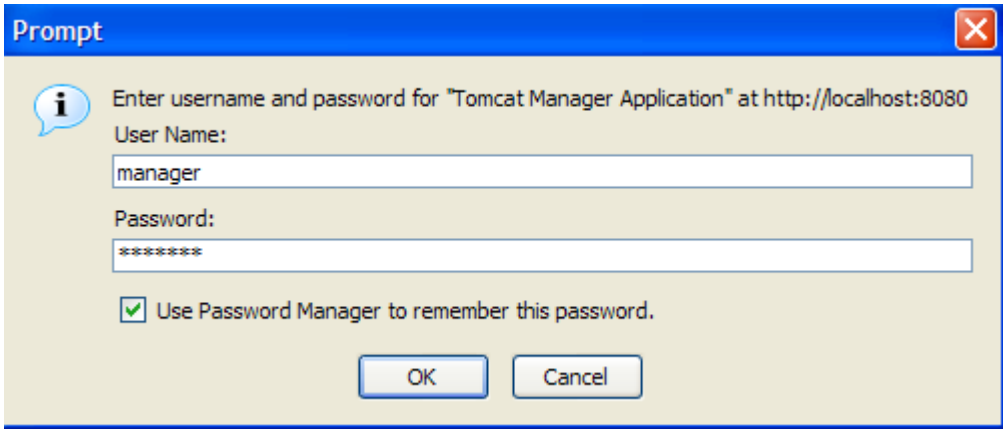
```
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="manager" password="manager" fullName="" roles="manager"/>
  <user username="admin" password="admin" roles="admin,manager"/>
</tomcat-users>
```

Une autre façon d'ajouter des utilisateurs est donc de modifier directement ce fichier. C'est notamment ainsi qu'il faut procéder si d'aventure on a oublié le mot de passe de l'administrateur **admin**.

Revenons maintenant à la page d'entrée [http://localhost:8080] et suivons le lien [Tomcat Manager] :



Nous obtenons alors une page d'authentification. Nous nous identifions comme manager/manager, c.a.d. l'utilisateur de rôle [manager] que nous venons de créer. En effet, seul un utilisateur ayant ce rôle peut utiliser ce lien :



Nous obtenons une page listant les applications actuellement déployées dans Tomcat :

Gestionnaire d'applications WEB Tomcat

Message:	OK

Manager			
List Applications	HTML Manager Help	Manager Help	Etat du serveur

Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arréter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arréter Recharger Undeploy
/balancer		true	0	Démarrer Arréter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arréter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arréter Recharger Undeploy
/servlets-examples	Servlet 2.4 Examples	true	0	Démarrer Arréter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer Arréter Recharger Undeploy
/webdav	Webdav Content Management	true	0	Démarrer Arréter Recharger Undeploy

Nous pouvons ajouter une nouvelle application grâce à des liens trouvés en bas de la page :

Deploy

Deploy directory or WAR file located on server

Context Path (optional):

XML Configuration file URL:

WAR or Directory URL:

Deploy

WAR file to deploy

Select WAR file to upload Browse...

Deploy

Ici, nous voulons déployer au sein de Tomcat, l'application **exemple** que nous avons construite précédemment. Nous le faisons de la façon suivante :

Deploy

Deploy directory or WAR file located on server

Context Path (optional): /exemple

XML Configuration file URL:

WAR or Directory URL: file://D:\data\serge\devel\web\servlets\exer

Deploy

`Context Path`	/exemple	le nom utilisé pour désigner l'application web à déployer
`Directory URL`	file:// D:\data\serge\devel\web\servlets\exemple	le dossier de l'application web

Pour obtenir le fichier D:\data\serge\devel\web\servlets\exemple\vues\exemple.html, nous demanderons à Tomcat l'URL http://localhost:8080/exemple/vues/exemple.html. Le contexte sert donc à donner un nom à la racine de l'arborescence de l'application web déployée. Nous utilisons le bouton [Deploy] pour effectuer le déploiement de l'application. Si tout se passe bien, nous obtenons la page réponse suivante :

Gestionnaire d'applications WEB Tomcat

Message:	OK - Application déployée pour le chemin de contexte /exemple

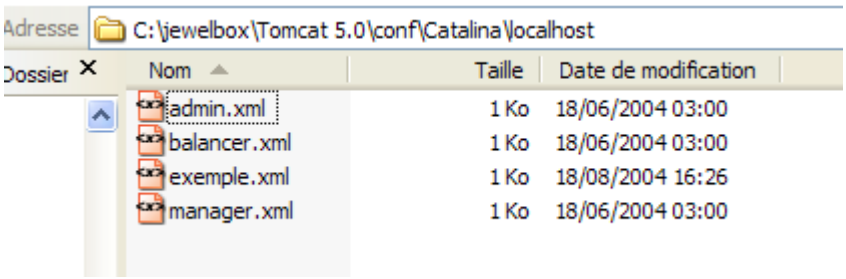
et la nouvelle application apparaît dans la liste des applications déployées :

Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arrêter Recharger Undeploy
/balancer		true	0	Démarrer Arrêter Recharger Undeploy
/exemple		true	0	Démarrer Arrêter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arrêter Recharger Undeploy

Commentons la ligne du contexte **/exemple** ci-dessus :

`/exemple`	lien sur http://localhost:8080/exemple
`Démarrer`	permet de démarrer l'application
`Arrêter`	permet d'arêter l'application
`Recharger`	permet de recharger l'application. C'est nécessaire par exemple lorsqu'on ajouté, modifié ou supprimé certaines classes de l'application.
`Undeploy`	suppression du contexte /exemple. L'application disparaît de la liste des applications disponibles.

Le déploiement a eu pour effet d'ajouter un nouveau descripteur XML dans le dossier <tomcat>\conf\Catalina\localhost :



Le fichier **exemple.xml** décrit l'application web ajoutée :

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="D:/data/serge/devel/web/servlets/exemple" path="/exemple">
</Context>
```

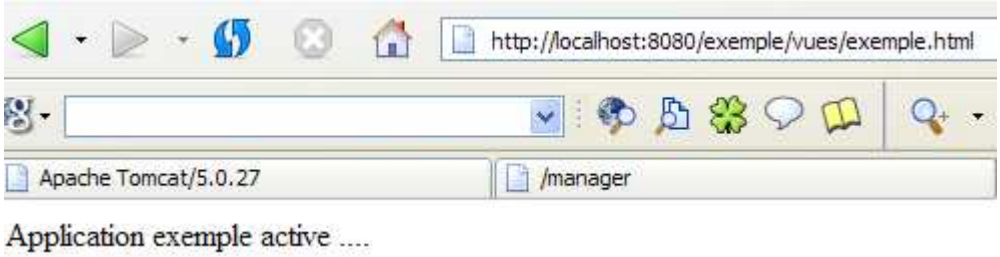
On retrouve tout simplement ce que nous avons saisi dans la page web de déploiement. Une autre façon de procéder est donc la suivante :

1. écrire à la main le descripteur XML de l'application à déployer
2. mettre ce descripteur dans le dossier <tomcat>\conf\Catalina\localhost
3. arrêter et relancer Tomcat

Maintenant que notre application **/exemple** est déployée, nous pouvons faire quelques tests.

1. Tout d'abord, nous demandons l'url racine du contexte http://localhost:8080/exemple/ :

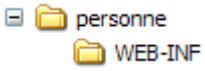
Nous obtenons une liste du contenu du dossier associé au contexte /exemple. Cependant le dossier WEB-INF d'une application web n'est lui jamais présenté. Nous pouvons demander la page exemple.html via l'url http://localhost:8080/exemple/vues/exemple.html :



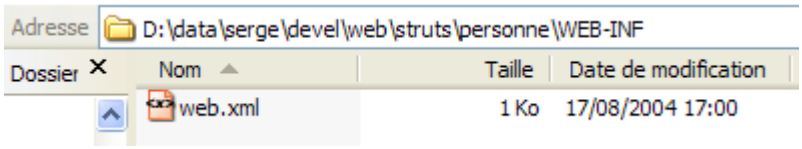
II.4 Déploiement d'une application Struts au sein du serveur Tomcat

Une application Struts est une application web comme une autre. Elle suit donc les règles de déploiement que nous venons de décrire. Ici une application, que nous appellerons **/strutspersonne**, sera exécutée par un serveur Tomcat version 5.x. Nous suivons donc les règles de déploiement des applications web au sein de ce serveur :

- nous créons le dossier physique qui sera associé au contexte **strutspersonne** ainsi que le sous-dossier **WEB-INF**.



- dans le dossier WEB-INF de l'application , nous définissons le fichier de configuration **web.xml** de l'application :

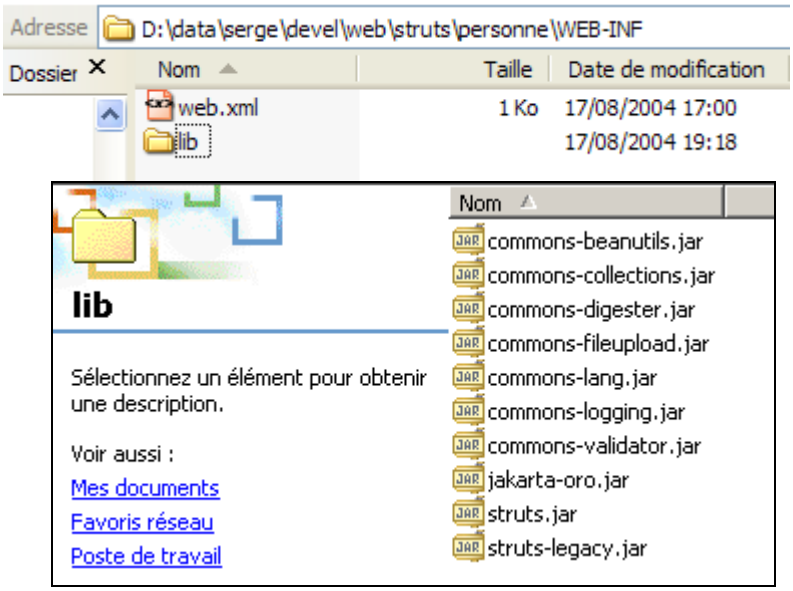


- le contenu du fichier **web.xml** est le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
 <servlet>
   <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
     <param-name>config</param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
  </servlet>
```

```
  <servlet-mapping>
   <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

</web-app>
```

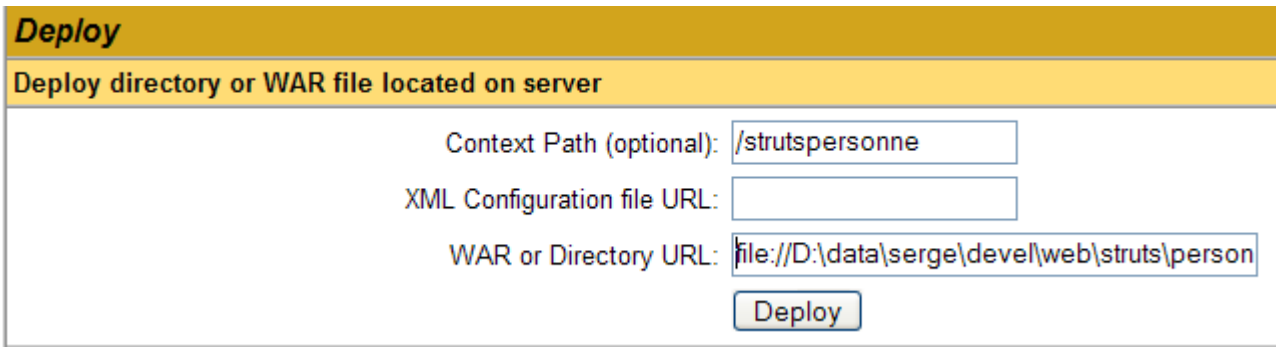
- la classe du contrôleur (servlet) de l'application est une classe prédéfinie de Stuts, appelé **ActionServlet**. Elle se trouve dans le fichier **struts.jar**. Afin que Tomcat puisse trouver cette classe, on placera **struts.jar** dans le dossier **/WEB-INF/lib** de l'application web, dossier à créer. En fait, nous placerons dans ce dossier tous les fichiers .jar trouvés dans le dossier **<struts>\lib** où **<struts>** est le dossier racine de l'arborescence de Struts.



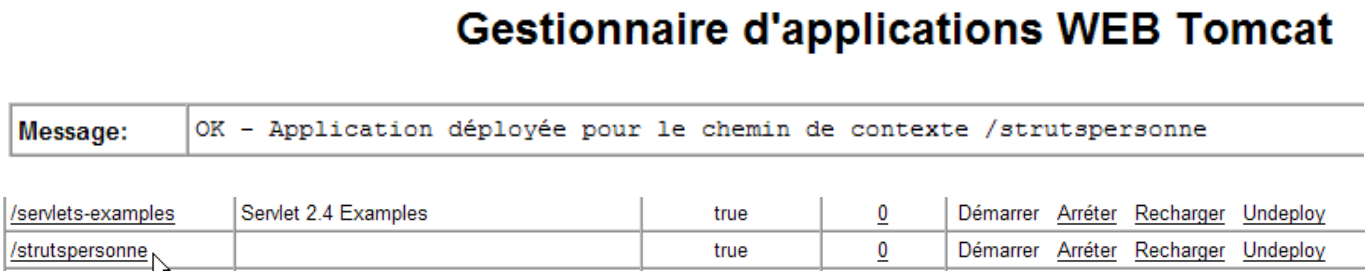
  Beaucoup de ces archives seront inutiles dans nos exemples. On a simplement voulu montrer où les trouver.
- nous avons indiqué que le contrôleur Struts avait besoin d'un certain nombre d'informations qu'il trouvait normalement dans un fichier appelé **struts-config.xml** placé dans le même dossier que **web.xml**. En fait, le nom de ce fichier est paramétrable. C'est le paramètre **config** du fichier **web.xml**, ci-dessus, qui fixe ce nom.
- la balise **<servlet-mapping>** indique que le contrôleur sera atteint via toutes les URL se terminant par le suffixe **.do**. Cette association est habituelle dans les applications Struts mais pas obligatoire. D'autres associations peuvent être choisies. Ces URL seront ensuite filtrées par le contrôleur qui n'acceptera que les URL déclarées dans son fichier de configuration **struts-config.xml**

Pour l'instant, notre fichier **web.xml** est suffisant.

1. Nous allons déployer l'application dans Tomcat. Le lecteur est invité à relire la démarche présentée pour l'application /exemple. Nous allons donc sur la page [Tomcat Manager] pour y définir les caractéristiques de l'application à déployer : son contexte (= son nom) et le dossier physique où elle se trouve :



Une fois déployée, l'application est dans la liste des applications déployées :



2. Nous allons demander l'URL **/main.do** à l'application **strutspersonne**. D'après le fichier **web.xml** précédent, cette URL sera donc transmise à la servlet **org.apache.struts.action.** La classe **ActionServlet** va être instanciée et sa méthode **init** appelée. Celle-ci essaie de lire le fichier de configuration défini par le paramètre **config**. Celui-ci doit donc exister. Nous créons le fichier **WEB-INF\struts-config.xml** suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <action-mappings>
    <action
      path="/main"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>
</struts-config>
```

Remarquons que le fichier DTD de **struts-config.xml** n'est pas le même que celui du fichier **web.xml**, indiquant par là qu'ils n'ont pas la même structure. Pour chaque URL que le contrôleur doit gérer, il nous faut définir une balise **<action>**. Celle-ci sert à indiquer au contrôleur ce qu'il doit faire lorsqu'on lui demande cette URL. Ici, nous indiquons les éléments suivants :

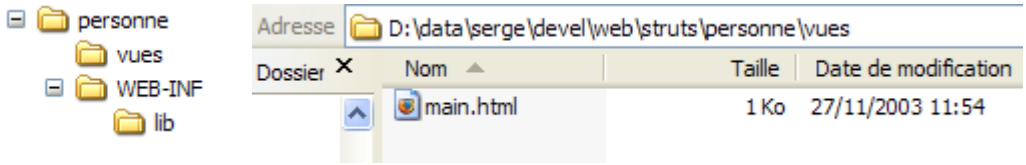
1. **path="/main"** : définit le nom de l'URL configurée par la balise **<action>**. Le suffixe **.do** est implicite.
2. **type="org.apache.struts.actions.ForwardAction"** : définit le nom de la classe **Action** qui doit traiter la demande. Ici, on utilise une classe **Action** prédéfinie dans Struts. Elle ne fait rien par elle-même et relaie la demande du client à l'URL indiquée dans l'attribut **parameter**.
3. **parameter="/vues/main.html"** : le nom de l'URL à qui doit être relayée la demande. Ici c'est un fichier HTML statique.

En résumé, lorsque l'utilisateur demandera l'URL **/main.do,** il obtiendra l'URL **/vues/main.html**.

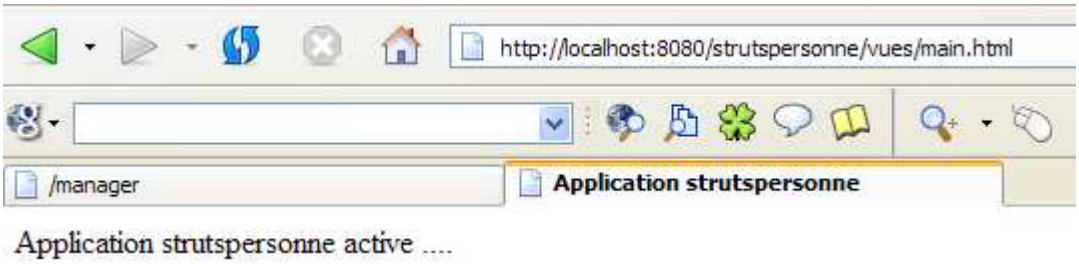
5. Le fichier **main.html** sera le suivant :

```
<html>
  <head>
    <title>Application strutspersonne</title>
  </head>
  <body>
    Application strutspersonne active ....
  </body>
</html>
```

Ce fichier est placé dans le dossier de l'application **strutspersonne/vues** :



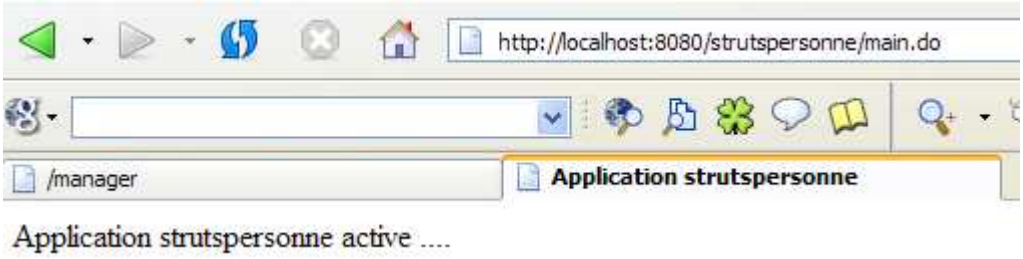
On peut le demander directement avec l'URL **http://localhost:8080/strutspersonne/vues/main.html** :



Ici, le contrôleur Struts de l'application **n'est pas intervenu** puisqu'il n'intervient que si on demande l'URL de type ***.do**. Or ici, on a demandé l'URL **/vues/main.html**.

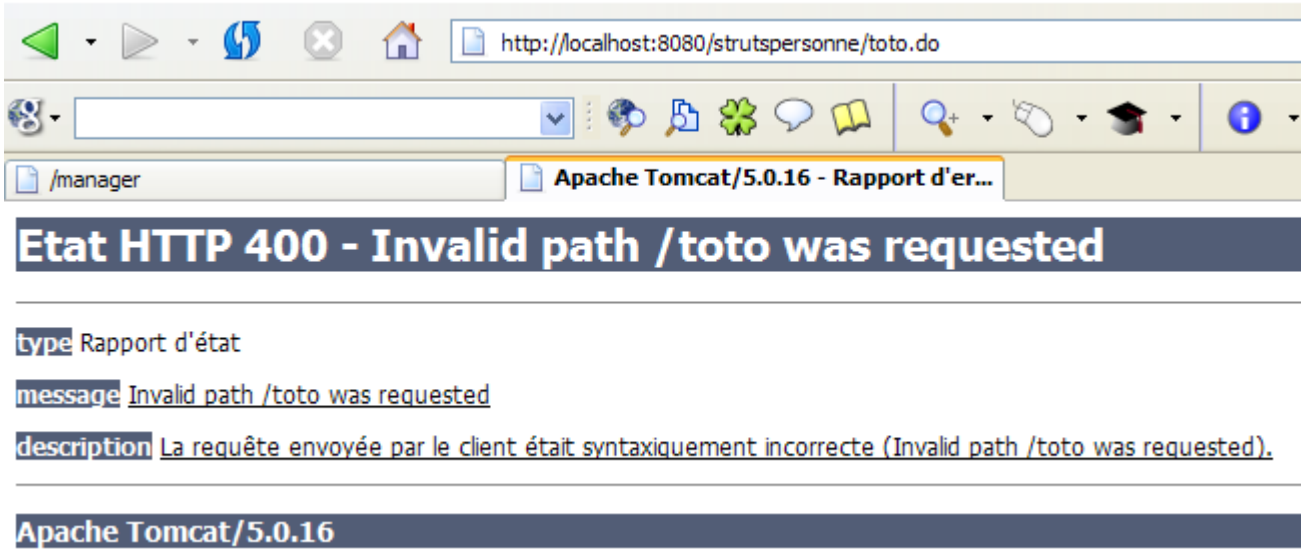
6. Le fichier **struts-config.xml** construit précédemment doit être placé dans le même dossier **WEB-INF** que le fichier **web.xml** :

7. Nous allons maintenant vérifier le bon fonctionnement du contrôleur de l'application **strutspersonne** en demandant l'URL **/main.do** après avoir, si nécessaire, relancé Tomcat.



Ici le contrôleur Struts est intervenu puisque nous avons demandé une URL de type ***.do**. Nous avons bien obtenu la page attendue (**main.html**). Nous avons donc les éléments de base du fonctionnement de notre application : le contexte **/strutspersonne**, les fichiers de configuration **web.xml** et **struts-config.xml**, les bibliothèques Struts.

Que se serait-il passé si nous avions demandé une URL de type **/toto.do** ? D'après le fichier **web.xml** de l'application strutspersonne, le contrôleur Struts est alors appelé pour la traiter. Celui-ci regarde alors son fichier de configuration **struts-config.html** et ne trouve aucune configuration pour l'URL **/toto**. Que fait-il alors ? Essayons :



Nous obtenons une page d'erreurs, ce qui semble normal.

II.5 Les fichiers de logs de Tomcat

Parfois une application web déployée au sein de Tomcat échoue à démarrer. La cause en est souvent un fichier web.xml incorrect. Pour exemple, introduisons une erreur dans le fichier web.xml de l'application précédente :

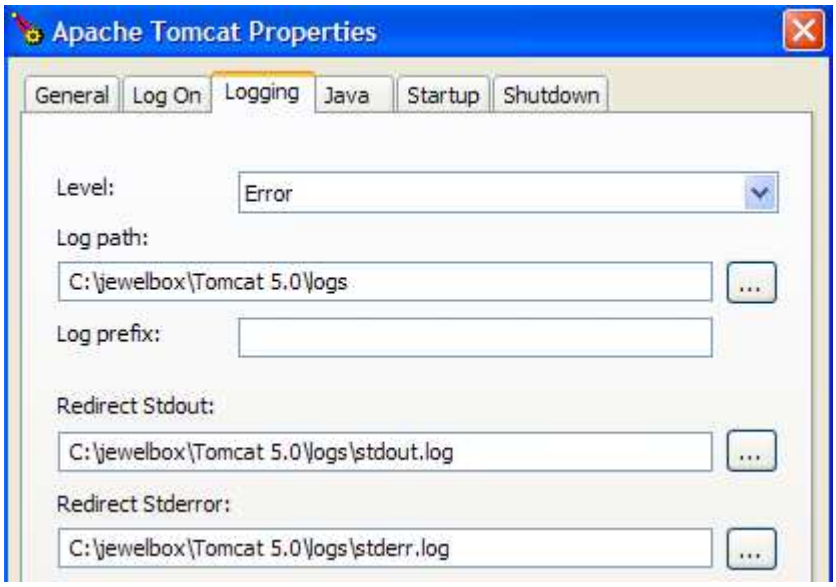
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config<param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

</web-app>
```

L'erreur introduite est dans la région encadrée. On a oublié la barre / dans la balise de fermeture </param-name>. Arrêtons/redémarrons l'application /strutspersonne avec l'aide de [Tomcat-manager]. On obtient alors un message d'erreur :

Gestionnaire d'applications WEB Tomcat

Message:	ECHEC - Lapplication pour le chemin de contexte {0} n'a pas puêtredémarrée

Malheureusement, cette page ne donne aucune information sur la cause de l'erreur. Toute l'activité de Tomcat est enregistrée dans des fichiers de logs mais où trouver ceux-ci ? L'outil [Configure] de Tomcat nous le dit :

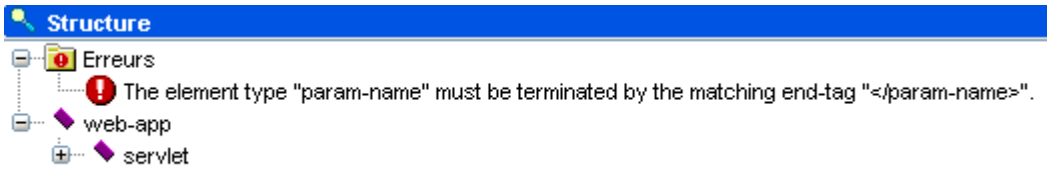


Examinons le fichier <tomcat>\logs\stdout.log et notamment la fin de ce fichier qui contient les logs des activités les plus récentes :

```
....
19 août 2004 15:23:12 org.apache.catalina.core.StandardHostDeployer start
INFO: standardHost.start /strutspersonne
19 août 2004 15:23:12 org.apache.commons.digester.Digester fatalError
GRAVE: Parse Fatal Error at line 14 column 7: The element type "param-name" must be terminated by the
matching end-tag "</param-name>".
```

```
org.xml.sax.SAXParseException: The element type "param-name" must be terminated by the matching end-tag
"</param-name>".
  at org.apache.xerces.util.ErrorHandlerWrapper.createSAXParseException(Unknown Source)
  ...
19 août 2004 15:23:12 org.apache.catalina.startup.ContextConfig applicationConfig
GRAVE: S'est produite à la ligne 14 colonne 7
19 août 2004 15:23:12 org.apache.catalina.startup.ContextConfig start
GRAVE: Cette application est marquée comme non disponible suite aux erreurs précédentes
19 août 2004 15:23:12 org.apache.catalina.core.StandardContext start
GRAVE: Error getConfigured
19 août 2004 15:23:12 org.apache.catalina.core.StandardContext start
GRAVE: Erreur de démarrage du contexte suite aux erreurs précédentes
```

L'erreur est clairement indiquée (encadré). Il faut donc songer à examiner ce fichier de logs lorsqu'une application échoue à démarrer. Il est bon également d'utiliser un IDE capable de vérifier la bonne syntaxe d'un fichier XML. Par exemple l'IDE Jbuilder Foundation indique l'erreur précédente :



Il est hautement souhaitable de travailler avec des IDE capables de vérifier la syntaxe de fichiers XML (Jbuilder, Eclipse, ...) afin d'éviter des erreurs pas toujours faciles à localiser.

III L'application /strutspersonne

Nous nous proposons d'introduire Struts avec une application simple qui nous permettra de découvrir les principaux mécanismes mis en oeuvre par une application Struts.

III.1 Le fonctionnement de l'application

Nous complétons maintenant l'application **/strutspersonne** dont nous avons initié l'écriture dans le chapitre précédent. Elle sera composée des éléments suivants :

- d'une servlet **main** qui assurera la logique de l'application.
- de trois pages JSP : **formulaire.personne.jsp**, **reponse.personne.jsp**, **erreurs.personne.jsp**

Le fonctionnement de l'application est le suivant. Elle est accessible à l'adresse **/strutspersonne/formulaire.do**. A cette url, on obtient un formulaire fourni par la page **formulaire.personne.jsp** :

L'utilisateur remplit le formulaire et appuie sur le bouton [Envoyer] de type **submit**. Le bouton [Rétablir] est de type **reset**, c.a.d. qu'il remet le document dans l'état où il a été reçu. Le bouton [Effacer] est de type **button**. L'utilisateur doit fournir un nom et un âge valides. Si ce n'est pas le cas, une page d'erreurs lui est envoyée au moyen de la page JSP **erreurs.personne.jsp**. Voici des exemples :

Echange n° 1

demande	réponse
Personne - formulaire Nom xx Age yy Envoyer Rétablir Effacer	**Les erreurs suivantes se sont produites** • L'âge [yy] est incorrect Retour au formulaire

Si on suit le lien [Retour au formulaire] on retrouve celui-ci dans l'état où on l'a laissé :

Echange n° 2

demande	réponse
Les erreurs suivantes se sont produites • Le champ [age] est erroné Retour au formulaire	**Personne - formulaire** Nom xx Age yy Envoyer Rétablir Effacer

Si l'utilisateur envoie des données correctes, l'application lui envoie une réponse au moyen de la page JSP **reponse.personne.jsp**.

Echange n° 1

demande	réponse
Personne - formulaire Nom pauline Age 18 Envoyer Rétablir Effacer	**Personne - réponse** Nom pauline Age 18 Retour au formulaire

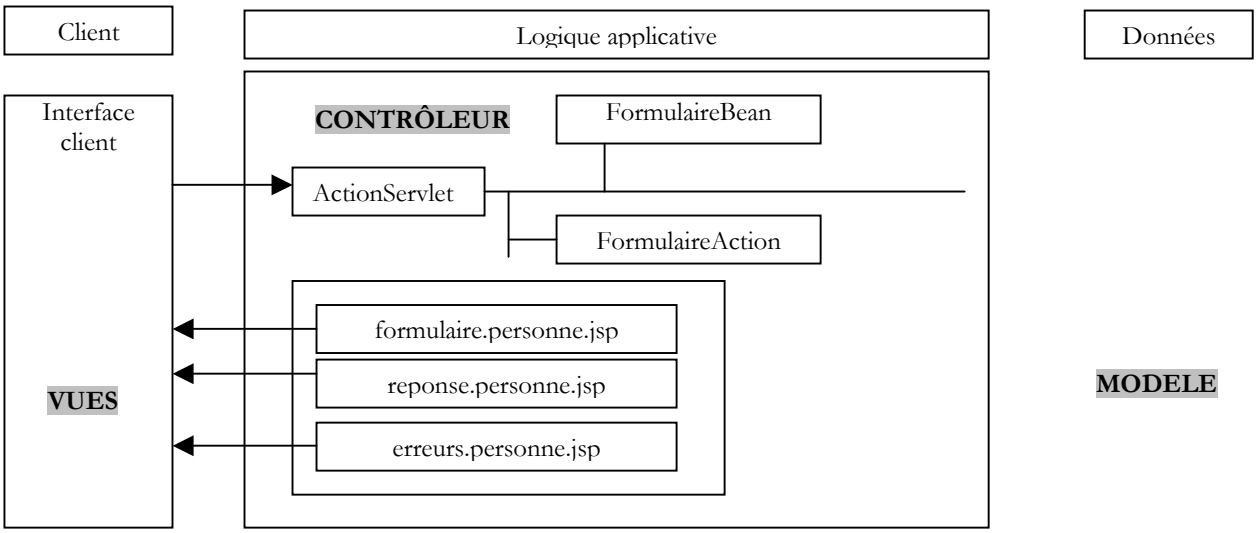
Si on suit le lien [Retour au formulaire] on retrouve celui-ci dans l'état où on l'a laissé :

Echange n° 2

demande	réponse
Personne - réponse Nom pauline Age 18 Retour au formulaire	**Personne - formulaire** Nom pauline Age 18 Envoyer Rétablir Effacer

III.2L'architecture Struts de l'application

Nous adopterons l'architecture Struts suivante :



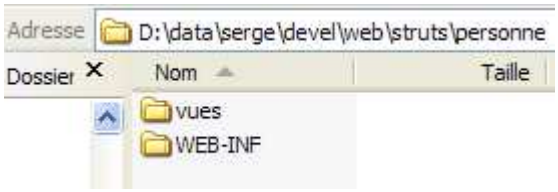
- il y aura trois vues
- le contrôleur sera celui fourni par Struts
- **FormulaireBean** est la classe chargée de mémoriser les valeurs du formulaire présenté par la vue **formulaire.personne.jsp**
- **FormulaireAction** est la classe chargée de traiter les valeurs de **FormulaireBean** et d'indiquer la page réponse à envoyer :
  - la vue **erreurs.personne.jsp** si les données du formulaire sont erronées
  - la vue **reponse.personne.jsp** sinon

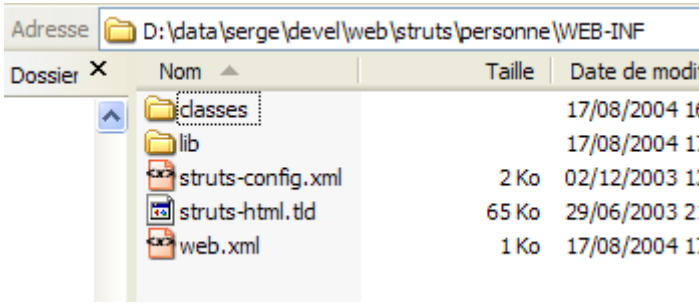
Pour le développeur, le travaille consiste à écrire le code :

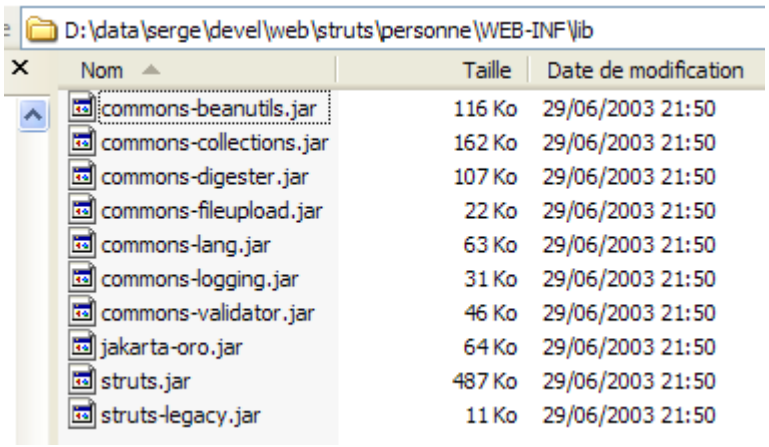
- des trois vues
- du bean **FormulaireBean** associé au formulaire
- de la classe **FormulaireAction** chargée de traiter le formulaire

III.3Configuration de l'application

L'arborescence du dossier de l'application **/strutspersonne** est la suivante :







- <strutspersonne>\vues contiendra les vues de l'application
- <strutspersonne>\WEB-INF contiendra le fichier de configuration web.xml de l'application ainsi que le fichier struts-config.xml de configuration de Struts. Le fichier web.xml est celui déjà rencontré et expliqué :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
 <servlet>
   <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
     <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
   <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

</web-app>
```

Le fichier struts-config.xml de départ sera celui utilisé dans la version précédente :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
 <action-mappings>
   <action
     path="/main"
     parameter="/vues/main.html"
     type="org.apache.struts.actions.ForwardAction"
   />
 </action-mappings>
</struts-config>
```

- <strutspersonne>\WEB-INF\classes contiendra les classes de l'application ainsi que le fichier des messages
- <strutspersonne>\WEB-INF\lib contiendra les bibliothèques .jar nécessaires à l'application. Ces .jar sont pris dans <struts>\lib. Ce sera ainsi dans tous les exemples à venir. Le fichier <strutspersonne>\WEB-INF\web.xml est le suivant :

Nous découvrirons les autres fichiers de l'arborescence au fur et à mesure de l'écriture de l'application.

III.4Les vues de l'application /strutspersonne

Les trois vues de l'application sont les suivantes :

- **formulaire.personne.jsp** : affiche le formulaire de saisie des nom et âge d'une personne
- **reponse.personne.jsp** : affiche les valeurs saisies si elles sont valides
- **erreurs.personne.jsp** : affiche les erreurs s'il y en a

III.4.1 La vue erreurs.personne.jsp

Cette vue qui affiche une liste d'erreurs sera définie comme suit :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <html:errors/>
     <html:link page="/formulaire.do">
      Retour au formulaire
    </html:link>
  </body>
</html>
```

Il y a deux nouveautés dans ce code :

1. la présence de balisee **<html:XX/>** qui ne sont pas des balise HTML. On peut en effet créer des bibiothèques de balises JSP qui lors de la transformation de la page JSP en servlet sont transformées en code Java.

2. la page JSP doit déclarer les bibliothèques de balises qu'elle utilise. Elle le fait ici avec la ligne

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
```

Cette ligne donne deux informations :

- **uri** : l'emplacement du fichier régissant les règles d'utilisation de la bibliothèque. Le fichier **struts-html.tld** est fourni dans la distribution de Struts. On le placera, dans l'exemple ci-dessus, dans le dossier WEB-INF.
- **prefix** : l'identificateur utilisé dans le code pour préfixer les balises de la bibliothèque. Cela permet d'éviter les conflits de noms qui pourraient survenir lors de l'utilisation simultanée de plusieurs bibliothèques de balises. Il serait possible de trouver deux balises de même nom dans deux bibliothèques différentes. En donnant un préfixe différent à chaque bibliothèque, on supprime toute ambiguïté.

- la balise <**html:errors**> affiche la liste d'erreurs que le contrôleur Struts lui tansmet.
- la balise <**html:link**> génère un lien pointant sur **/C/page** où
  - **C** est le contexte de l'application
  - **page** est l'URL indiquée dans l'attribut page de la balise

Le balise

```
    <html:link page="/formulaire.do">
    Retour au formulaire
  </html:link>
```

génèrera le code HTML suivant :

```
<a href="/C/formulaire.do">Retour au formulaire</a>
```

où **C** est le contexte de l'application

III.4.2 Test de la vue erreurs.personne.jsp

- le fichier **erreurs.personne.jsp** est placé dans le dossier **vues** de l'application **strutspersonne** :

- le fichier **struts-html.tld** est pris dans la distribution struts (**<struts>/lib**) et placé dans **WEB-INF** :

- le fichier **struts-config.xml** est modifié de la façon suivante :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <action-mappings>
    <action
      path="/main"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>
</struts-config>
```

Nous créons dans le fichier de configuration une nouvelle URL **/erreurs** destinée à être traitée par le contrôleur struts. L'URL **/erreurs.do** sera redirigée vers la vue **/vues/erreurs.personne.jsp**. Le fichier **struts-config.xml** est placé dans le dossier WEB-INF.

- nous réinitialisons l'application pour que le nouveau fichier **struts-config.xml** soit pris en compte. Pour cela, on s'aidera de la page web de gestion de Tomcat [Tomcat Manager] déjà présentée et qui liste les applications déployées dans Tomcat :

Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arréter Recharger Undeploy
/admin	Tomcat Administration Application	true	0	Démarrer Arréter Recharger Undeploy
/balancer		true	0	Démarrer Arréter Recharger Undeploy
/exemple		true	0	Démarrer Arréter Recharger Undeploy
/jsp-examples	JSP 2.0 Examples	true	0	Démarrer Arréter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arréter Recharger Undeploy
/servlets-examples	Servlet 2.4 Examples	true	0	Démarrer Arréter Recharger Undeploy
/strutspersonne		true	0	Démarrer Arréter Recharger Undeploy

Pour réinitialiser l'application [/strutspersonne], nous utilisons le lien [Recharger] de la ligne définissant l'application. Dans certains cas, comme une modification du fichier web.xml, un rechargement ne suffit pas. Il faut alors utiliser les liens [Arrêter] et [Redémarrer] pour réinitialiser l'application.

- puis nous demandons l'url **http://localhost:8080/strutspersonne/erreurs.do** :

La balise **<html:errors/>** n'a rien produit. C'est normal, l'action **ForwardAction** n'a pas généré la liste d'erreurs attendue par la balise. Néanmoins, la réponse ci-dessus montre que notre vue JSP est au moins syntaxiquement correcte sinon nous aurions obtenu une page d'erreurs. Vérifions le code HTML reçu par le navigateur (View/Source) :

```
<html>
  <head>
    <title>Personne - erreurs</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
     <a href="/strutspersonne/formulaire.do">Retour au formulaire</a>
  </body>
</html>
```

On notera le lien produit par la balise <**html:link**>. Le contexte **/strutspersonne** a été automatiquement inclus dans le lien. Cela permet de déplacer l'application d'un contexte à un autre (changement de machine par exemple) sans avoir à changer les liens générés par <**html:link**>.

III.4.3 La vue reponse.personne.jsp

Cette vue qui confirme les valeurs saisies dans le formulaire est la suivante :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
```

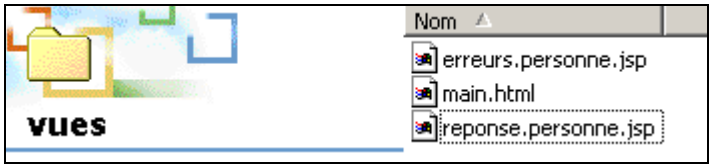
```
<%
  // on récupère les données nom, age
  //String nom=(String)request.getAttribute("nom");
```

```
  String nom="jean";

  //String age=(String)request.getAttribute("age");
  String age="24";
%>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Personne - réponse</h2>
     <hr>
     <table>
       <tr>
         <td>Nom</td>
          <td><%= nom %>
        </tr>
       <tr>
         <td>Age</td>
          <td><%= age %>
        </tr>
     </table>
     <br>
     <html:link page="/formulaire.do">
       Retour au formulaire
     </html:link>
  </body>
</html>
```

La page affiche deux informations [nom] et [age] qui lui seront passées par le contrôleur dans l'objet prédéfini **request**. Ici, nous faisons un test où le contrôleur n'aura pas l'occasion de fixer les valeurs de [nom] et [age]. Aussi initialisons-nous ces deux informations avec des valeurs arbitraires. Par ailleurs, ici également le lien de retour vers le formulaire est généré par une balise <**html:link**>.

III.4.4 Test de la vue reponse.personne.jsp

- le fichier **reponse.personne.jsp** est placé dans le dossier **vues** de l'application **strutspersonne** :



- le fichier **struts-config.xml** est modifié de la façon suivante :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <action-mappings>
    <action
      path="/main"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/reponse"
      parameter="/vues/reponse.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>
</struts-config>
```

Nous créons dans le fichier de configuration une nouvelle URL **/reponse** destinée à être traitée par le contrôleur struts. L'URL **/reponse.do** sera redirigée vers la vue **/vues/reponse.personne.jsp**. Le fichier **struts-config.xml** est placé dans le dossier WEB-INF.

- nous réinitialisons l'application /strutspersonne pour que le nouveau fichier **struts-config.xml** soit pris en compte, puis nous demandons l'url **http://localhost:8080/strutspersonne/reponse.do** :

Nous obtenons bien ce qui était attendu.

III.4.5 La vue formulaire.personne.jsp

Cette vue présente le formulaire de saisie des nom et âge de l'utilisateur. Son code JSP est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <meta http-equiv="pragma" content="no-cache">
  <head>
    <title>Personne - formulaire</title>
     <script language="javascript">
      function effacer(){
        with(document.frmPersonne){
          nom.value="";
           age.value="";
         }
       }
     </script>
  </head>

  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
       <html:form action="/main" name="frmPersonne" type="istia.st.struts.personne.FormulaireBean">
        <table>
          <tr>
            <td>Nom</td>
              <td><html:text property="nom" size="20"/></td>
           </tr>
           <tr>
            <td>Age</td>
              <td><html:text property="age" size="3"/></td>
           </tr>
          <tr>
         </table>
         <table>
          <tr>
            <td><html:submit value="Envoyer"/></td>
              <td><html:reset value="Rétablir"/></td>
              <td><html:button property="btnEffacer" value="Effacer" onclick="effacer()"/></td>
           </tr>
         </table>
       </html:form>
     </center>
  </body>
</html>
```

Nous retrouvons la bibliothèque de balises **struts-html.tld** utilisée dans la vue **erreurs**. De nouvelles balises apparaissent :

`html:form` sert à la fois à générer la balise HTML <**form**> et à donner des informations au contrôleur qui aura à traiter ce formulaire :

- `action` url où seront envoyées les valeurs du formulaire
- `name` nom du formulaire HTML. C'est aussi le nom du bean qui mémorisera les valeurs de celui-ci
- `type` nom de la classe qui devra être instanciée pour obtenir le bean de stockage du formulaire

On remarquera que la méthode d'envoi des paramètres du formulaire (GET/POST) au contrôleur n'est pas précisée. On pourrait le faire avec l'attribut **method**. En l'absence de celui-ci, c'est la méthode **POST** qui est utilisée par défaut.

`html:text` sert à générer la balise **<input type="text" value="...">** :

- `property` nom du champ du bean du formulaire qui sera associé à la zone de saisie. A l'envoi du formulaire au serveur (client -> serveur), le champ du bean prendra la valeur du champ de saisie. A l'affichage du formulaire (serveur -> client), la valeur contenue dans le champ du bean est affichée dans la zone de saisie.

`html:submit` sert à générer la balise HTML <**input type="submit"**...>

`html:reset` sert à générer la balise HTML <**input type="reset"**...>

`html:button` sert à générer la balise HTML <**input type="button"**...>

III.4.6 Le bean associé au formulaire formulaire.personne.jsp

Avec Struts, tout formulaire doit être associé à un bean chargé de mémoriser les valeurs du formulaire et de conserver celles-ci dans la session courante. Un bean est une classe Java qui doit respecter une syntaxe particulière. Le bean associé à un formulaire doit dériver de la classe **ActionForm** définie dans les bibliothèques de Struts :

```
public class FormulaireBean extends ActionForm {
```

- les noms des attributs du bean doivent correspondre aux champs du formulaire (attributs **property** des balises **html:text** du formulaire). D'après le code du formulaire précédent, le bean doit donc avoir deux champs appelés **nom** et **age**.
- pour chaque champ **unChamp** du formulaire, le bean doit définir deux méthodes :
  - public void **setUnChamp**(Type valeur) : pour affecter une valeur à l'attribut **unChamp**
  - Type **getUnChamp**() : pour obtenir la valeur du champ **unChamp**

  On notera la syntaxe particulières des méthodes **get** et **set**. La première lettre du nom de l'attribut **unChamp** est une minuscule. Les méthodes **get** et **set** transforment cette première lettre en majuscule.

Le bean associé au formulaire précédent pourrait être le suivant :

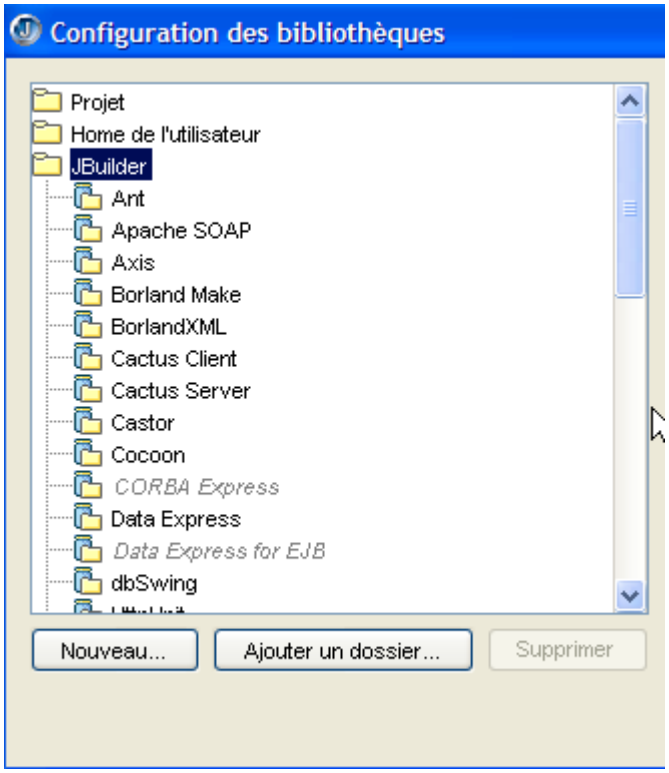
```
package istia.st.struts.personne;

import org.apache.struts.action.ActionForm;

public class FormulaireBean extends ActionForm {
  // nom
  private String nom = null;
  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }

  // age
  private String age = null;
  public String getAge() {
    return age;
  }
  public void setAge(String age) {
    this.age = age;
  }
}
```

On compilera cette classe.

III.4.7 Compiler les classes nécessaires à l'application Struts

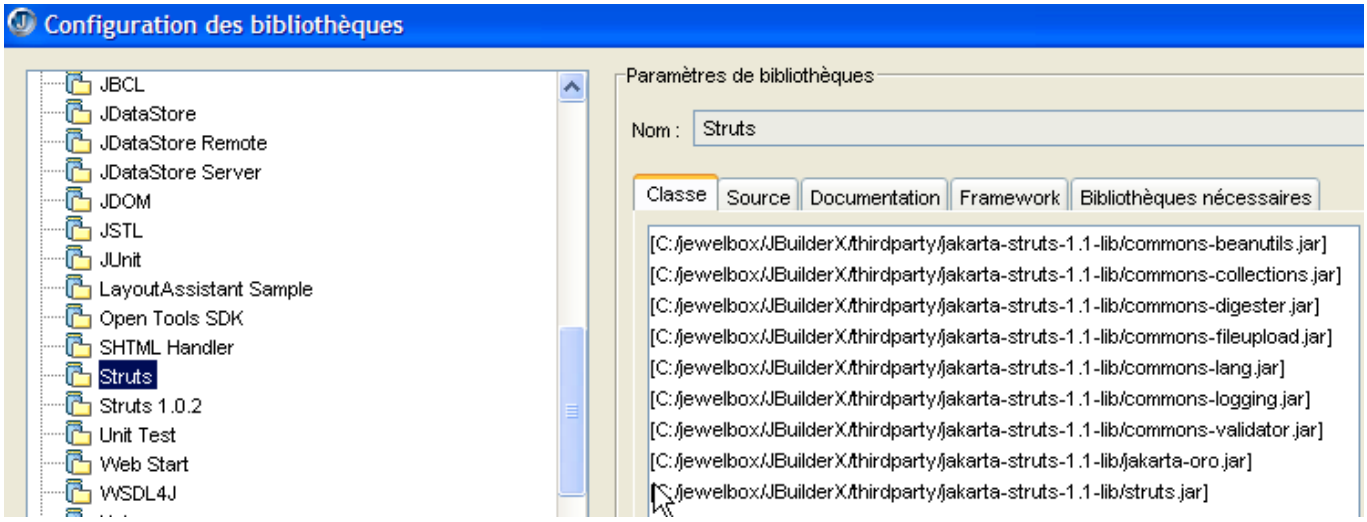
III.4.7.1 JBuilder

Pour compiler les classes nécessaires à notre application, nous utiliserons dans les exemples qui suivent, **Jbuilder Personal Edition,** un IDE renommé récemment **Jbuilder Foundation** et disponible gratuitement à l'URL http://www.borland.com/jbuilder. N'importe quel IDE convient et le lecteur est invité à utiliser son IDE habituel. L'IDE travaille avec un JDK dans lequel ne se trouvent normalement pas les classes nécessaires aux applications Struts. C'est le cas de JBuilder. On pourra ajouter les archives .jar de Struts à la liste des .jar explorés par le JDK lors de la compilation de la façon suivante. Dans le menu de JBuilder X :

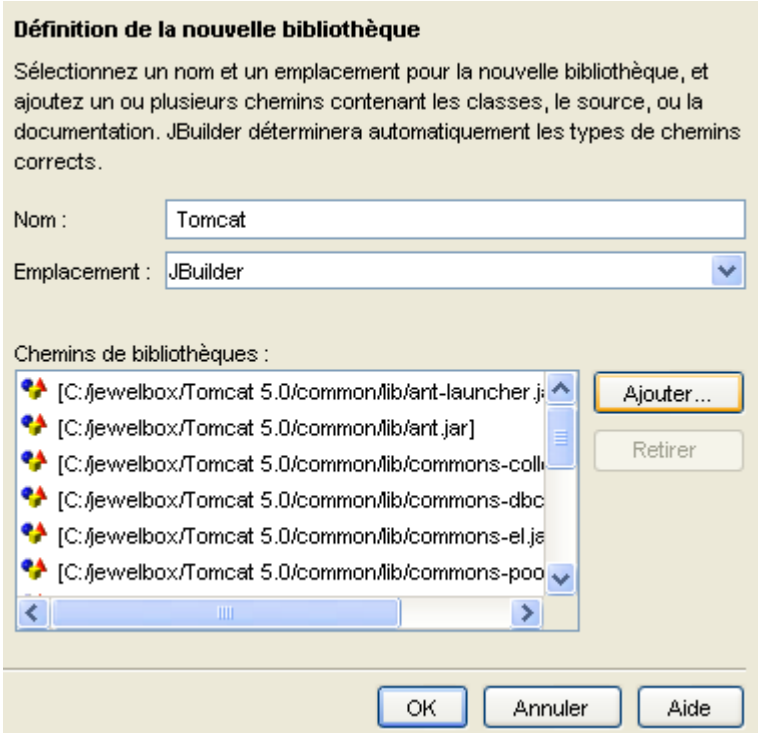


JBuilderX vient avec un certain nombre de bibliothèques, chacune d'elles comportant une ou plusieurs archives .jar. Ainsi ci-dessus, la bibliothèque [Ant] comporte les archives suivantes :

On peut ainsi découvrir que JbuilderX est fourni avec la bibliothèque Struts 1.1 :

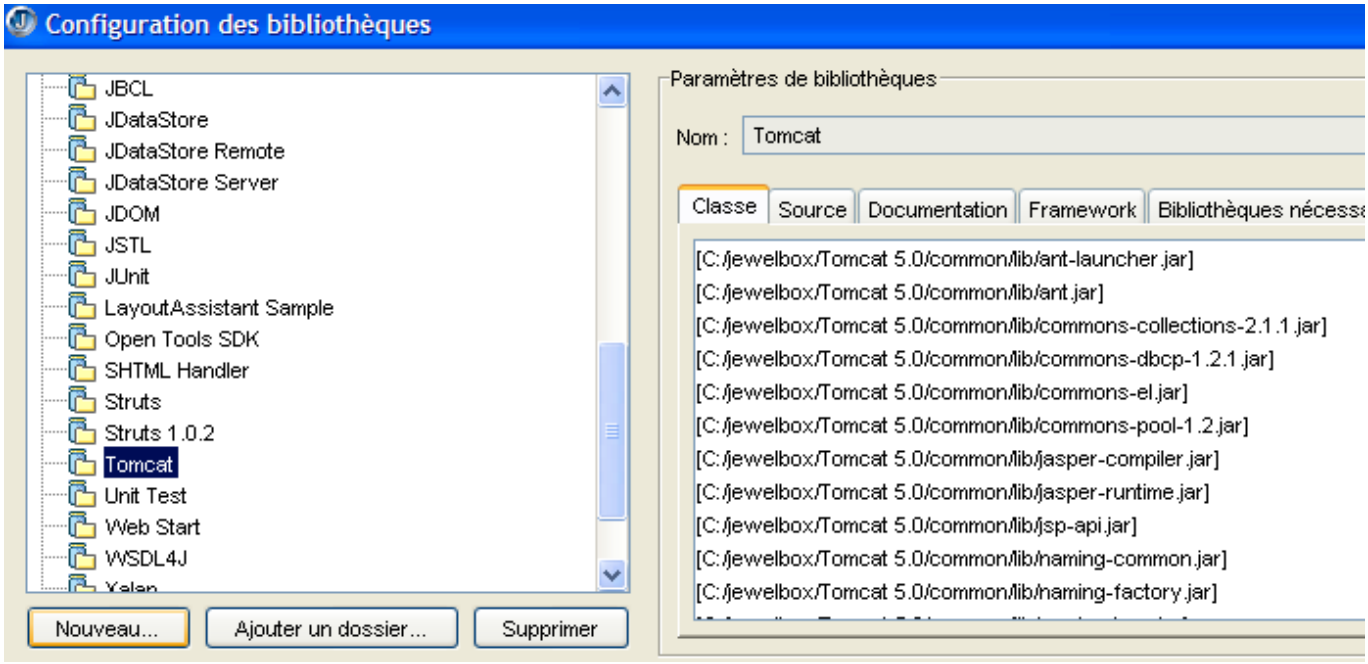


JBuilder Foundation ne dispose pas des archives permettant la compilation des servlets. Aussi allons-nous créer une nouvelle bibliothèque que nous appellerons Tomcat. Pour cela, nous utilisons le bouton [Nouveau] permettant de créer une nouvelle bibliothèque d'archives :



- nous appelons cette nouvelle bibliothèque Tomcat puisqu'elle utilise les archives amenées par cette application
- la visibilité de la bibliothèque sera [Jbuilder]. Cela signifie que cette bibliothèque sera disponible à toutes les applications construites avec JBuilder
- nous utilisons le bouton [Ajouter] pour aller désigner le dossier <tomcat>\common\lib qui contient les .jar de Tomcat et notamment servlet-api.jar qui permet la compilation des servlets. Il suffit de désigner le dossier. Tous les .jar qui y sont seront automatiquement ajoutés à la bibliothèque en cours de création.

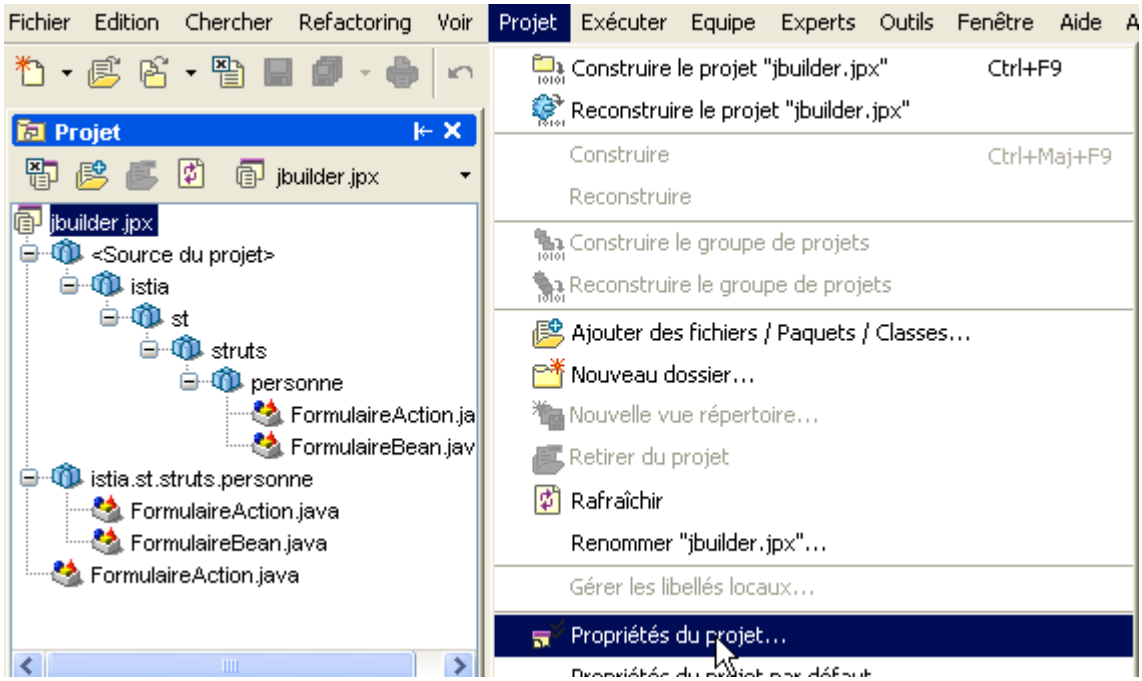
Une fois l'opération validée par [OK], nous disposons d'une nouvelle bibliothèque :



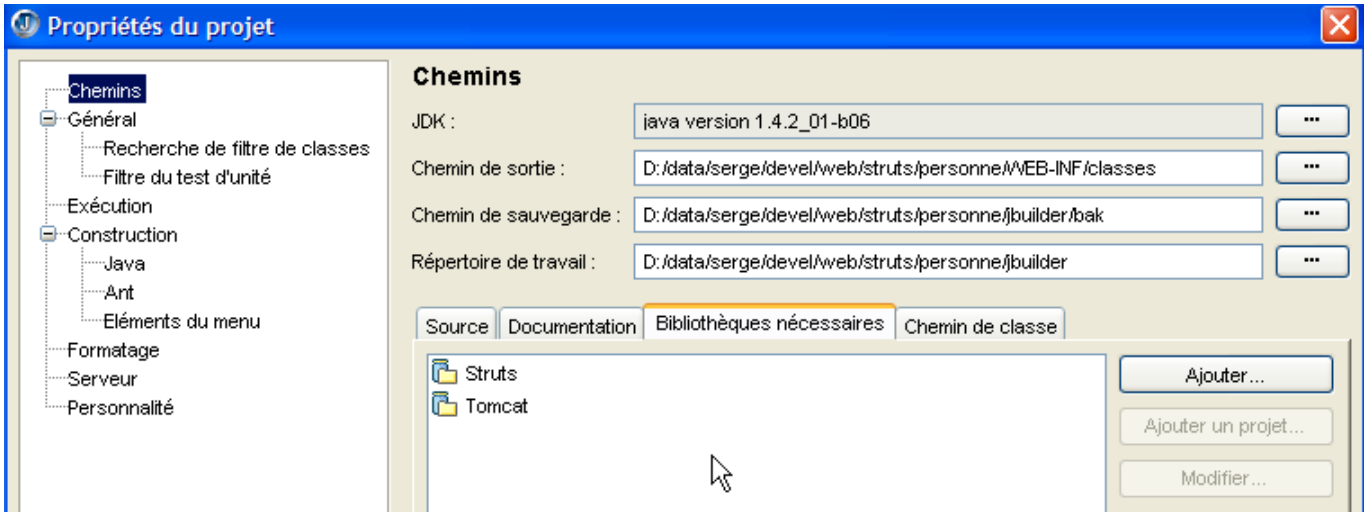
Si la bibliothèque Struts n'avait pas été présente, on l'aurait ajouter de façon analogue en incluant les archives du dossier **<struts>\lib**.

Pour chaque projet Jbuilder devant utiliser les bibliothèques Tomcat et Struts, nous procéderons de la façon suivante :

- modifier les propriétés du projet :



- ajouter les bibliothèques [Struts] et [Tomcat] au projet :

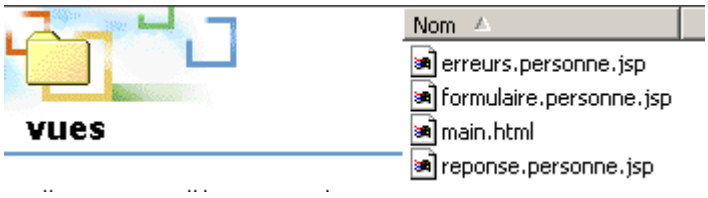


Les classes produites par la compilation doivent être placées dans le dossier WEB-INF\classes de l'application. Nous pouvons indiquer à JBuilderX où placer les .class produits par la compilation. C'est le champ [Chemin de sortie] de l'écran ci-dessus. Ici, nous avons indiqué de placer les .class dans le dossier WEB-INF\classes de l'application.

Ceci fait, nous pouvons compiler la classe **FormulaireBean** puis procéder au test de la vue **formulaire.personne.jsp**.

III.4.8 Test de la vue formulaire.personne.jsp

Le fichier **formulaire.personne.jsp** est placé dans le dossier **vues** de l'application **strutspersonne** :

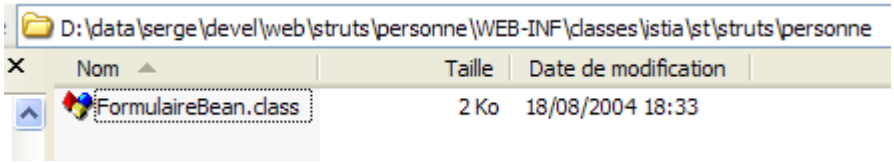


- le fichier **struts-config.xml** est modifié de la façon suivante :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <action-mappings>
    <action
      path="/main"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/reponse"
      parameter="/vues/reponse.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/formulaire"
      parameter="/vues/formulaire.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>
</struts-config>
```

Nous créons dans le fichier de configuration une nouvelle URL **/formulaire** destinée à être traitée par le contrôleur struts. L'URL **/formulaire.do** sera redirigée vers la vue **/vues/formulaire.personne.jsp**. Le fichier **struts-config.xml** est placé dans le dossier WEB-INF.

- nous plaçons la classe **FormulaireBean** dans **WEB-INF/classes/istia.st.struts.personne** :



- nous réinitialisons l'application [Arrêter-Démarrer] pour que le nouveau fichier **struts-config.xml** soit pris en compte, puis nous demandons l'url **http://localhost:8080/strutspersonne/formulaire.do** :

Nous obtenons bien le formulaire. Nous pouvons avoir la curiosité de regarder comment ont été "traduites" les balises **<html:XX>** qui parsemaient le code JSP du formulaire :

```
<html>
  <meta http-equiv="pragma" content="no-cache">
  <head>
    <title>Personne - formulaire</title>
     <script language="javascript">
      function effacer(){
        with(document.frmPersonne){
          nom.value="";
           age.value="";
         }
       }
     </script>
  </head>

  <body>
    <center>

      <h2>Personne - formulaire</h2>
       <hr>
       <form name="frmPersonne" method="post" action="/strutspersonne/main.do">
        <table>
          <tr>
            <td>Nom</td>
              <td><input type="text" name="nom" size="20" value=""></td>
           </tr>

           <tr>
            <td>Age</td>
              <td><input type="text" name="age" size="3" value=""></td>
           </tr>
          <tr>
         </table>
         <table>
          <tr>

            <td><input type="submit" value="Envoyer"></td>
              <td><input type="reset" value="Rétablir"></td>
              <td><input type="button" name="btnEffacer" value="Effacer" onclick="effacer()"></td>
```

```
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

Dans le formulaire obtenu, les boutons |Rétablir] et [Effacer] fonctionnent. Le bouton [Envoyer] de type **submit** renvoie à l'URL **/strutspersonne/main.do**. D'après le fichier **web.xml** de l'application, le contrôleur Struts va la prendre en charge. D'après le fichier **struts-config.html**, le contrôleur doit rediriger la requête vers la vue **/vues/main.html**. Essayons :

Tout se passe comme attendu. Il nous reste à traiter véritablement les valeurs du formulaire, c.a.d. écrire la classe de type **Action** qui va recevoir les données du formulaire dans un objet **FormulaireBean** que le contrôleur va lui passer.

III.5Traitement du formulaire par le contrôleur

Nous nous intéressons maintenant au traitement des valeurs du formulaire par le contrôleur lorsque l'utilisateur va appuyer sur le bouton [Envoyer] du formulaire.

III.5.1 Le fichier struts-config.xml

Le nouveau fichier de configuration **struts-config.xml** du contrôleur Struts devient le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean
      name="frmPersonne"
      type="istia.st.struts.personne.FormulaireBean"
    />
  </form-beans>

  <action-mappings>
    <action
      path="/main"
      name="frmPersonne"
      scope="session"
      validate="true"
      input="/erreurs.do"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/reponse"
      parameter="/vues/reponse.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/formulaire"
      parameter="/vues/formulaire.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>
```

```
  <message-resources parameter="ressources.personneressources"/>
</struts-config>
```

Nous avons mis en relief les changements :

- une section **<form-beans>** apparaît. Elle sert à définir les classes associées à chacun des formulaire de l'application. Il doit y avoir autant de balises **<form-bean>** que de formulaires différents dans l'application. Ici, nous n'avons qu'un formulaire donc une seule section **<form-bean>**. Pour chaque formulaire, nous devons définir :
    - son nom (attribut **name**)
    - le nom de la classe dérivée de **ActionForm** chargée de stocker les valeurs du formulaire (attribut **type**)

  Ces deux attributs ne peuvent être quelconques. Ils doivent être identiques à ceux utilisés dans la balise **<html:form>** du code HTML du formulaire. Rappelons celle-ci pour le formulaire (nom, age) :

```
    <html:form action="/main" name="frmPersonne" type="istia.st.struts.personne.FormulaireBean">
```

  Le formulaire doit être déclaré de la même façon dans le fichier **struts-config.html**. C'est ce qui est fait ici :

```
  <form-bean
    name="frmPersonne"
    type="istia.st.struts.personne.FormulaireBean"
  />
```

- la configuration de l'action **/main** a changé. Celle-ci est chargée de traiter les valeurs du formulaire. Aussi doit-on donner les informations dont elle a besoin sur celui-ci :

```
  <action
    path="/main"
    name="frmPersonne"
    scope="session"
    validate="true"
    input="/erreurs.do"
    parameter="/vues/main.html"
    type="org.apache.struts.actions.ForwardAction"
  />
```

  La servlet **/main** va traiter un formulaire dont il faut lui donner le nom. C'est l'attribut **name** qui fait ce travail. Ce nom doit référencer l'attribut **name** de l'une des sections **<form-bean>**, ici **frmPersonne**.

  L'attribut **scope="session"** indique que les valeurs du formulaire doivent être stockées dans la session. Ce n'est pas toujours nécessaire. Ici, ça l'est. En effet, nous trouvons dans les vues **/reponse.do** et **/erreurs.do** des liens ramenant au formulaire. Dans les deux cas, nous voulons afficher le formulaire avec les valeurs saisies par l'utilisateur lors d'un précédent échange client-serveur. D'où la nécessité de stocker le formulaire dans la session.

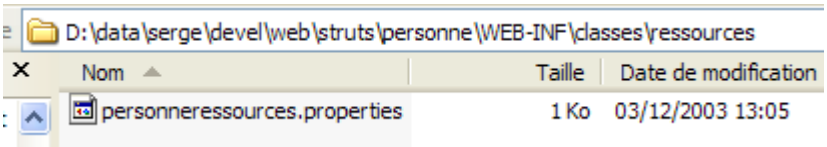
  L'attribut **validate** indique si la méthode **validate** de l'objet **frmPersonne** doit être appelée ou non. Cette méthode sert à vérifier la validité des données du formulaire. Ici, nous indiquons que les données doivent être vérifiées ce qui implique que nous aurons à écrire une méthode **validate** dans la classe **FormulaireBean**. La méthode **validate** du formulaire est appelée par le contrôleur Struts avant que la servlet **/main** soit appelée. Elle rend comme résultat un objet de type **ActionErrors** qui est analogue à une liste d'erreurs. Si cette liste existe et est non vide, le contrôleur Struts s'arrêtera là et enverra comme réponse, la vue indiquée par l'attribut **input**. La vue recevra dans la requête la liste **ActionErrors** qu'elle pourra afficher avec la balise **<html:errors>**. Ci-dessus, nous disons qu'en cas d'erreurs, la servlet **/main** doit envoyer la vue **/erreurs.do**. Rappelons que cette vue est associée à l'URL **/vues/erreurs.reponse.jsp** suivante :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <html:errors/>
     <html:link page="/formulaire.do">
      Retour au formulaire
    </html:link>
  </body>
</html>
```

  La vue utilise bien la balise **<html:errors>** qui permettra d'afficher la liste des erreurs. Dans cette liste d'erreurs, on trouve non pas des messages d'erreurs mais des identifiants de messages présents dans le fichier référencé par la balise **<message-resources>** (attention : **resources** avec un seul **s**) du fichier **struts-config.xml** :

```
  <message-resources parameter="ressources.personneressources"/>
```

La balise ci-dessous indique que le fichier contenant les messages utilisés par l'application se trouve dans le fichier **WEB-INF/classes/ressources/personneressources.properties** :



Que trouve-t-on dans ce fichier ? C'est un fichier de propriétés correspondant à la classe **Properties** de Java, c'est à dire un ensemble de lignes **clé=valeur** :

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge [{0}] est incorrect</li>
errors.header=<ul>
errors.footer=</ul>
```

Ce fichier de messages a au moins deux fonctions :

- il permet de changer les messages de l'application sans avoir à recompiler celle-ci
- il permet l'internationalisation des applications Struts. On peut en effet créer plusieurs fichiers de ressources, un par langue. Struts utilisera automatiquement le bon fichier de messages à condition de respecter certaines normes dans le nommage de ces fichiers.

- Si la méthode **validate** du formulaire rend une liste d'erreurs vide, alors le constrôleur Struts appelle la méthode **execute** de la servlet **ForwardAction**. Il est important de comprendre ici que lorsque la méthode **execute** de la servlet s'exécute c'est que les données du formulaire ont été considérées comme valides (si bien sûr elles ont été vérifiées par validate="true"). C'est dans la méthode **execute** de la servlet associée à l'action, que le développeur traite réellement le formulaire. C'est là que va se trouver le coeur du traitement (logique applicative, utilisation de classes métier et de classes d'accès aux données). Au final, la méthode rend un résultat de type **ActionForward** qui indique au contrôleur quelle vue doit être envoyée en réponse au client. Ici nous avons utilisé l'action prédéfinie **ForwardAction** de Struts. Sa méthode **execute** se contente de rendre un **ActionForward** pointant sur l'URL indiquée par l'attrinut **parameter** :

```
    <action
      path="/main"
      name="frmPersonne"
      validate="true"
      input="/erreurs.do"
      parameter="/vues/main.html"
      type="org.apache.struts.actions.ForwardAction"
    />
```

Si donc, les données du formulaire sont valides alors l'action **/main** renverra la vue **/vues/main.html** que nous avons déjà utilisée.

III.5.2 La nouvelle classe FormulaireBean

Nous avons déjà créé une première version de la classe **FormulaireBean** chargée de mémoriser les donnée (nom,age) du formulaire **formulaire.personne.jsp**. Cette version ne vérifiait pas la validité des données. Maintenant, nous devons le faire puisque nous avons indiqué dans le fichier **struts-config.xml** que les données du formulaire devaient être vérifiées (**validate="true"**) avant d'être transmises à la servlet **ForwardAction**. Le code de la classe **FormulaireBean** devient le suivant :

```
package istia.st.struts.personne;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class FormulaireBean
  extends ActionForm {
  // nom
  private String nom = null;
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  // age
  private String age = null;
```

```
  public String getAge() {
    return age;
  }

  public void setAge(String age) {
    this.age = age;
  }

  // validation
  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
    // gestion des erreurs
    ActionErrors erreurs = new ActionErrors();
    // le nom doit être non vide
    if (nom == null || nom.trim().equals("")) {
      erreurs.add("nomvide", new ActionError("personne.formulaire.nom.vide"));
      // l'âge doit être un entier positif
    }
    if (age == null || age.trim().equals("")) {
      erreurs.add("agevide", new ActionError("personne.formulaire.age.vide"));
    }
    else {
      // l'âge doit être un entier positif
      if (!age.matches("^\\s*\\d+\\s*$")) {
        erreurs.add("ageincorrect", new ActionError("personne.formulaire.age.incorrect", age));
        // on rend la liste des erreurs
      }
    } //if
    // on rend la liste d'erreurs
    return erreurs;
  }
}
```

La nouveauté réside dans l'écriture de la méthode **validate**. Celle-ci est appelée par le contrôleur Struts **après** que celui-ci ait mis dans les attributs **nom** et **age** de la classe, les valeurs des champs du formulaire de même nom. Elle doit vérifier la validité des attributs **nom** et **age**. Le code ci-dessus est assez simple à comprendre :

- une liste d'erreurs (**ActionErrors erreurs**) vide est créée
- le champ **nom** est vérifié. S'il est vide, une erreur est ajoutée à la liste **erreurs** à l'aide de la méthode **ActionErrors.add("clé", ActionError)**.
- on fait de même si le champ **age** n'est pas un nombre entier positif.
- la méthode **validate** rend au contrôleur Struts la liste d'erreurs (**ActionErrors erreurs**). Si **erreurs** est égal à **null** ou si **erreurs.size()** est égal à 0, le contrôleur considère qu'il n'y a pas eu d'erreurs. Il fera alors exécuter la méthode **execute** de la classe **Action** associée à l'action (**type="org.apache.struts.actions.ForwardAction"**). Sinon, il renverra la vue associée au cas d'erreurs dans le formulaire (**input="/erreurs.do"**).

On ajoute une erreur à la liste **ActionErrors erreurs** par **ActionErrors.add("cléErreur", new ActionError("cléMessage"[,param0, param1, param2, param3]))**. Le premier paramètre "**cléErreur**" sert à désigner de façon unique un élément **ActionError** dans la liste **ActionErrors**, un peu comme dans un dictionnaire. Il peut être quelconque. **ActionError** est un objet qu'on associe à un message d'erreur à l'aide de son constructeur **ActionError(String cléMessage[,String param0, String param1, String param2, String param3])** où **cléMessage** est l'identifiant du message associé à l'erreur et **parami** jusqu'à 4 paramètres facultatifs. L'identifiant **cléMessage** n'est pas quelconque. C'est l'un des identifiants que l'on trouve dans le fichier désigné par la balise **<message-resources>** du fichier **struts-config.xml** :

```
  <message-resources parameter="ressources.personneressources"/>
```

Rappelons que ce fichier (en réalité **WEB-INF/classes/ressources/personneressources.properties**) contient les clés suivantes :

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge [{0}] est incorrect</li>
errors.header=<ul>
errors.footer=</ul>
```

On peut vérifier que les clés de messages utilisées par la méthode **validate** de la classe **FormulaireBean** existent bien dans le fichier ci-dessus. On a utilisé la balise HTML **<li>** pour chaque message d'erreur afin que la balise **<html:errors>** les affiche comme une liste HTML. Nous avons vu que l'objet **ActionError** pouvait être non seulement contruit avec une clé de message mais avec des paramètres supplémentaires :

```
public ActionError(String cléMessage[,String param0, String param1, String param2, String param3])
```

Si un **ActionError** a été construit avec des paramètres **parami** supplémentaires (jusqu'à quatre au maximum), ceux-ci sont accessibles dans le texte du message via la notation {0} à {3}. Ainsi, la méthode **validate** de **FormulaireBean** construit un **ActionError** avec la clé **personne.formulaire.age.incorrect** et le paramètre supplémentaire param0 **age** :

```
erreurs.add("age", new ActionError("personne.formulaire.age.incorrect",age));
```

Le message associé, dans le fichier **.properties** des messages, à la clé **personne.formulaire.age.incorrect** est

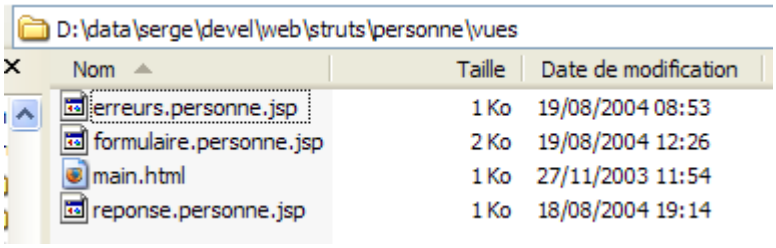
```
personne.formulaire.age.incorrect=<li>L'âge [{0}] est incorrect</li>
```

Le {0} sera remplacé par la valeur de l'âge. Enfin les messages ayant les clés **errors.header** et **errors.footer** seront écrits respectivement avant et après la liste des erreurs. Ici, ces deux clés serviront à inclure les balises HTML **
** et **
** qui doivent entourer les balises **<li>**.

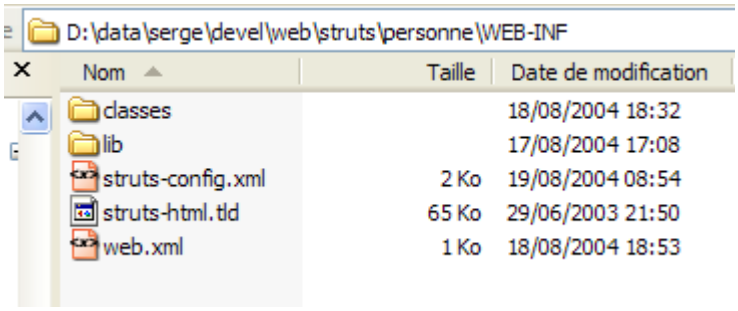
III.6Les tests de validité du formulaire

Nous sommes prêts pour les tests de validité du formulaire. Nous rappelons ci-dessous où doivent être placés les différents composants de l'application :

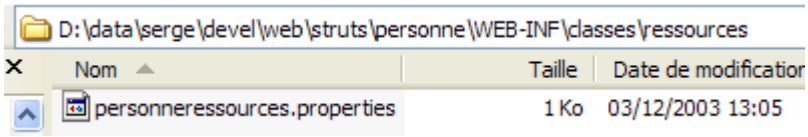
les vues



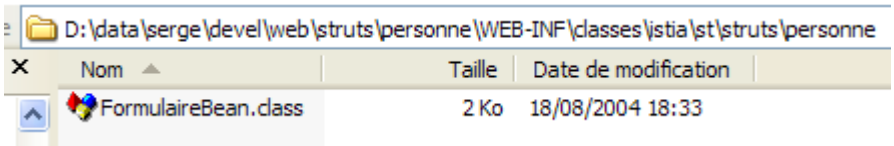
les fichiers de configuration



le fichier des messages



les classes



III.6.1 Test 1

Relançons Tomcat pour qu'il lise les nouveaux fichiers de configuration puis demandons l'URL **http://localhost:8080/strutspersonne/formulaire.do** :

Explications :

- dans **struts-config.html** la section suivante a été exploitée :

```
    <action
      path="/formulaire"
      parameter="/vues/formulaire.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
```

Si nous visualisons le code HTML de la page reçue nous voyons que la balise <**form**> de la page est la suivante :

```
      <form name="frmPersonne" method="post" action="/strutspersonne/main.do">
```

Le bouton [Envoyer] qui est de type **submit** enverra donc les données du formulaire à l'URL **/strutspersonne/main.do**.

III.6.2 Test 2

Utilisons le bouton [Envoyer] en laissant vides les champs de saisie. Nous obtenons la réponse suivante :

Explications :

- comme indiqué plus haut, les données du formulaire ont été envoyées à l'URL **/strutspersonne/main.do**. Les sections suivantes du fichier **struts-config.xml** ont alors été utilisées :

```
    <form-bean
      name="frmPersonne"
      type="istia.st.struts.personne.FormulaireBean"
      scope="session"
    />
....
    <action
      path="/main"
```

```
    name="frmPersonne"
    scope="session"
    validate="true"
    input="/erreurs.do"
    parameter="/vues/main.html"
    type="org.apache.struts.actions.ForwardAction"
  />
```

L'action /**main** a été déclenchée. Elle utilise le formulaire **frmPersonne** (**name="frmPersonne"**). Le contrôleur Struts a donc instancié, si nécessaire, un objet de la classe **FormulaireBean** (**type="istia.st.struts.personne.FormulaireBean"** dans la balise **form-bean**). Il a rempli les attributs **nom** et **age** de cet objet avec les champs **de même nom** du formulaire HTML :

```
      <table>
        <tr>
          <td>Nom</td>
            <td><html:text property="nom" size="20"/></td>
         </tr>
         <tr>
          <td>Age</td>
            <td><html:text property="age" size="3"/></td>
         </tr>
        <tr>
       </table>
```

Ceci fait, le contrôleur Struts a appelé la méthode **validate** de l'objet **FormulaireBean** parce que l'attribut **validate** de l'action **/main** est positionné à **true** dans le fichier de configuration :

```
  <action
    path="/main"
    name="frmPersonne"
    scope="session"
    validate="true"
    input="/erreurs.do"
    parameter="/vues/main.html"
    type="org.apache.struts.actions.ForwardAction"
  />
```

La méthode **validate** de la classe **FormulaireBean** est la suivante :

```
// validation
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
  // gestion des erreurs
  ActionErrors erreurs = new ActionErrors();
  // le nom doit être non vide
  if (nom == null || nom.trim().equals("")) {
    erreurs.add("nomvide", new ActionError("personne.formulaire.nom.vide"));
    // l'âge doit être un entier positif
  }
  if (age == null || age.trim().equals("")) {
    erreurs.add("agevide", new ActionError("personne.formulaire.age.vide"));
  }
  else {
    // l'âge doit être un entier positif
    if (!age.matches("^\\s*\\d+\\s*$")) {
      erreurs.add("ageincorrect", new ActionError("personne.formulaire.age.incorrect", age));
      // on rend la liste des erreurs
    }
  } //if
  // on rend la liste d'erreurs
  return erreurs;
}
```

Puisque les champs [nom] et [age] étaient vides, la méthode **validate** ci-dessus a généré une liste de deux erreurs qu'elle a renvoyée au contrôleur Struts. Parce qu'il y avait des erreurs, le contrôleur a ensuite renvoyé au client la vue associée à l'attribut **input** :

```
  <action
    path="/main"
    name="frmPersonne"
    scope="session"
    validate="true"
    input="/erreurs.do"
    parameter="/vues/main.html"
    type="org.apache.struts.actions.ForwardAction"
  />
```

Pour trouver la vue associée au chemin **/erreurs.do**, il a utilisé la section suivante de son fichier de configuration :

```
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
```

Il a donc au final envoyé la vue **/vues/erreurs.personne.jsp**. Celle-ci a le code suivant :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <html:errors/>
     <html:link page="/formulaire.do">
      Retour au formulaire
    </html:link>
  </body>
</html>
```

La balise <**html:errors**> fait un simple affichage de la liste des messages que le contrôleur Struts lui a envoyée. Elle utilise le fichier des messages indiqué par la balise <**message-resources** > :

```
  <message-resources parameter="ressources.personneressources"/>
```

On y trouve les clés et les messages suivants :

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge [{0}] est incorrect</li>
errors.header=<ul>
errors.footer=</ul>
```

- le message associée à la clé **errors.header** est écrit
- les messages associés aux différentes clés de la liste **ActionErrors** reçue sont écrits
- le message associée à la clé **errors.footer** est écrit

III.6.3 Test 3

Utilisons le lien [Retour au formulaire] de la page d'erreurs. Nous obtenons la page suivante :

Explications :

- le lien [Retour au formulaire] a le code HTML suivant :

```
    <a href="/strutspersonne/formulaire.do">Retour au formulaire</a>
```

Le contrôleur Struts a utilisé la section suivante se son fichier de configuration :

```
<action
  path="/formulaire"
  parameter="/vues/formulaire.personne.jsp"
  type="org.apache.struts.actions.ForwardAction"
/>
```

Il a donc renvoyé la vue **/vues/formulaire.personne.jsp**.

III.6.4 Test 4

Nous remplissons le formulaire suivante puis nous utilisons le bouton [Envoyer] :

Nous obtenons la réponse suivante :

Explications : ce sont les mêmes que pour le test n° 2.

III.6.5 Test 5

Nous utilisons le lien [Retour au formulaire] ci-dessus. Nous obtenons la page suivante :

Nous constatons que nous retrouvons le formulaire dans l'état où nous l'avons validé.

Explications : ce sont celles du test n° 3 avec une information supplémentaire :

- le formulaire HTML affiché a les balises suivantes :

```
      <table>
        <tr>
          <td>Nom</td>
            <td><html:text property="nom" size="20"/></td>
         </tr>
         <tr>
          <td>Age</td>
            <td><html:text property="age" size="3"/></td>
         </tr>
        <tr>
       </table>
```

Les balises <**html:text**> ont deux fonctions :

- lors de l'envoi des valeurs du formulaire du client vers le serveur, les valeurs des champs de saisie du formulaire sont affectées aux champs de même nom de l'objet **FormulaireBean**
- lors de l'envoi par le serveur au client du code HTML du formulaire à afficher, les attributs **value** des champs de saisie associés aux balises <**html:text**> sont initialisés avec les valeurs des champs de même nom de l'objet **FormulaireBean**.

Nous sommes ici dans deux échanges client-serveur différents :

- dans le premier, l'utilisateur a rempli le formulaire et l'a envoyé au serveur
- dans le second, l'utilisateur a utilisé le lien [Retour au formulaire] pour revenir au formulaire.

La seule façon pour que dans le second échange, le formulaire puisse être réaffiché avec ses valeurs d'origine c'est que celles-ci soient placées dans la session du client. C'est ce qui a été demandé dans la section configurant l'action **/main** :

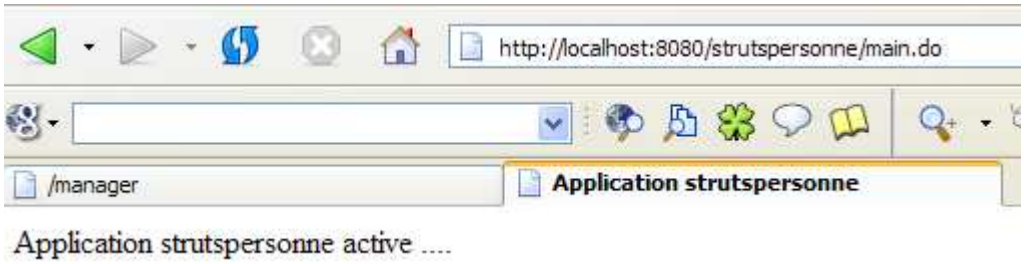
```
   <action
     path="/main"
     name="frmPersonne"
     scope="session"
     validate="true"
     input="/erreurs.do"
     parameter="/vues/main.html"
     type="org.apache.struts.actions.ForwardAction"
   />
```

Si nous avions mis **scope="request"**, les données du formulaire n'auraient pas été stockées dans la session et nous n'aurions alors pas retrouvé ses valeurs dans le second échange.

III.6.6 Test 6

Revenons au formulaire pour entrer cette fois-ci des données valides :

Validons le formulaire. Nous obtenons le résultat suivant :



Explications :

- comme le bouton [Envoyer] envoie les valeurs du formulaire à l'URL **/strutspersonne/main.do**, on retrouve les mêmes explications que dans le test n° 2 jusqu'au retour au contrôleur Struts du résultat **ActionErrors** de la méthode **validate** de **FormulaireBean**. Mais ici, cette liste est vide. Le contrôleur utilise alors une nouvelle partie de la configuration de l'action **/main** :

```
<action
  path="/main"
  name="frmPersonne"
  scope="session"
  validate="true"
  input="/erreurs.do"
  parameter="/vues/main.html"
  type="org.apache.struts.actions.ForwardAction"
/>
```

Le contrôleur Struts crée, si nécessaire, un objet du type indiqué par l'attribut **type.** La méthode **execute** de cette classe est exécutée et doit rendre un objet de type **ActionForward** indiquant la vue que doit envoyer le contrôleur comme réponse au client. Ici l'attribut **type** désigne la classe prédéfinie **ForwardAction**. La méthode **execute** de cette classe ne fait rien et se contente de rendre un objet **ActionForward** pointant sur la vue définie par l'attribut **parameter**, ici la vue **/vues/main.html**. C'est effectivement la vue que le contrôleur a renvoyée.

III.6.7 Test 7

On redemande la vue **/formulaire.do** :

On retouve le formulaire tel qu'on l'a validé. L'explication a déjà été donnée. Par configuration (scope="session") on a demandé que le formulaire reste dans la session. Ses valeurs sont donc conservées au fil des échanges client-serveur.

Nous avons presque terminé. Il nous reste à créer une action véritable pour le cas où les données du formulaire sont valides. Pour l'instant, nous avons utilisé l'action prédéfinie **ForwardAction** pour simplifier notre démonstration.

III.7Nouvelle configuration de l'action /main

Nous ne changeons pas le fichier de configuration **struts-config.xml** actuel si ce n'est pour modifier sa section **/main** de la façon suivante :

```
    <action
      path="/main"
      name="frmPersonne"
      scope="session"
      validate="true"
      input="/erreurs.do"
      type="istia.st.struts.personne.FormulaireAction"
    >
      <forward name="reponse" path="/reponse.do"/>
    </action>
```

L'attribut type désigne maintenant une autre classe appelée **FormulaireAction** que nous aurons à construire. C'est la méthode **execute** de cette classe qui sera exécutée si les données du formulaire **frmPersonne** sont valides. Nous avons indiqué que la méthode **execute** faisait ce qu'elle avait à faire et rendait un objet de type **ActionForward** indiquant la vue que le contrôleur devait renvoyer au client. Il y a souvent plusieurs vues possibles selon le résultat du traitement du formulaire. La liste des différentes vues possibles est listée dans les balises <**forward**> incluses dans la balise <**action**>. La syntaxe d'une telle balise est la suivante :

```
      <forward name="clé" path="/vue" />
```

`clé`	nom quelconque identifiant une vue de façon unique
`vue`	URL de la vue associée à la clé

III.8La classe FormulaireAction

Écrire la classe **FormulaireAction** consiste essentiellement à écrire sa méthode **execute** :

```
package istia.st.struts.personne;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import javax.servlet.ServletException;

public class FormulaireAction extends Action {
```

```
  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse response)
                 throws IOException,ServletException {

    // on a un formulaire valide, sinon on ne serait pas arrivé là
    FormulaireBean formulaire=(FormulaireBean)form;
    request.setAttribute("nom",formulaire.getNom());
    request.setAttribute("age",formulaire.getAge());
    return mapping.findForward("reponse");
  }//execute
}
```

La méthode **execute** reçoit quatre paramètres :

1. **ActionMapping mapping** : un objet "image" de la configuration de l'action en cours d'exécution donc ici une image de la configuration suivante :

```
   <action
     path="/main"
     name="frmPersonne"
     validate="true"
     input="/erreurs.do"
     type="istia.st.struts.personne.FormulaireAction"
   >
     <forward name="reponse" path="/reponse.do"/>
   </action>
```

   Ainsi l'action a accès aux clés associées aux vues qui peuvent être renvoyées au client à l'issue de l'action. La méthode **execute** devra renvoyer l'une de ces clés.
2. **ActionForm form** : l'objet bean dans lequel se trouvent les valeurs du formulaire utilisée par l'action en cours. Ici, c'est l'objet **frmPersonne** de type **FormulaireBean**. Ainsi l'action a accès aux valeurs du formulaire.
3. **HttpServletRequest request** : la requête du client qui a pu être enrichie par différentes servlets. L'action a ainsi accès à tous les paramètres de la requête initiale (request.getParameter) ainsi qu'à tous les attributs rajoutés à cette requête initiale (request.getAttribute). Dans notre exemple, la méthode **execute** enrichit la requête en y ajoutant le nom et l'âge. C'est parfaitement inutile ici puisque ces deux valeurs y sont déjà mais en tant que paramètres et non en tant qu'attributs. Le code est ici pour l'exemple.
4. **HttpServletResponse response** : la réponse qui sera envoyée au client. L'action pourrait enrichir cette réponse. Ici elle ne le fait pas.

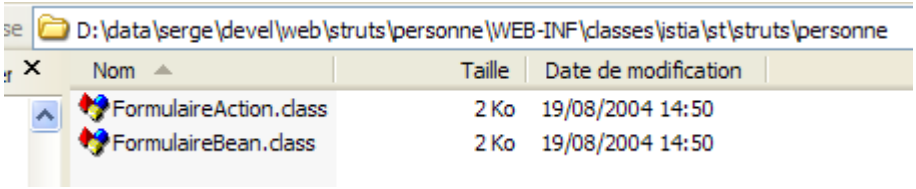
Ici, nous sommes dans un cas particulier. La méthode **execute** n'a quasiment rien à faire. Elle doit simplement indiquer que la vue suivante est la vue **/reponse.do** et mettre dans la requête que cette vue recevra, les informations **nom** et **age** que la vue doit afficher. Elle le fait au moyen de la méthode **findForward** de la class **ActionMapping** qui admet pour paramètre une des clés trouvées dans les balises **forward** de la cconfiguration de l'action. Ici, il n'y a qu'une telle balise :

```
     <forward name="reponse" path="/reponse.do"/>
```

Notre méthode **execute** rend donc un **ActionForward** ayant "**reponse**" comme clé pour indiquer que la vue **/reponse.do** doit être envoyée.

III.9Tests de FormulaireAction

Nous compilons la classe précédente et plaçons le .class généré dans **WEB-INF/classes** :



Nous modifions la vue **/vues/reponse.personne.jsp** :

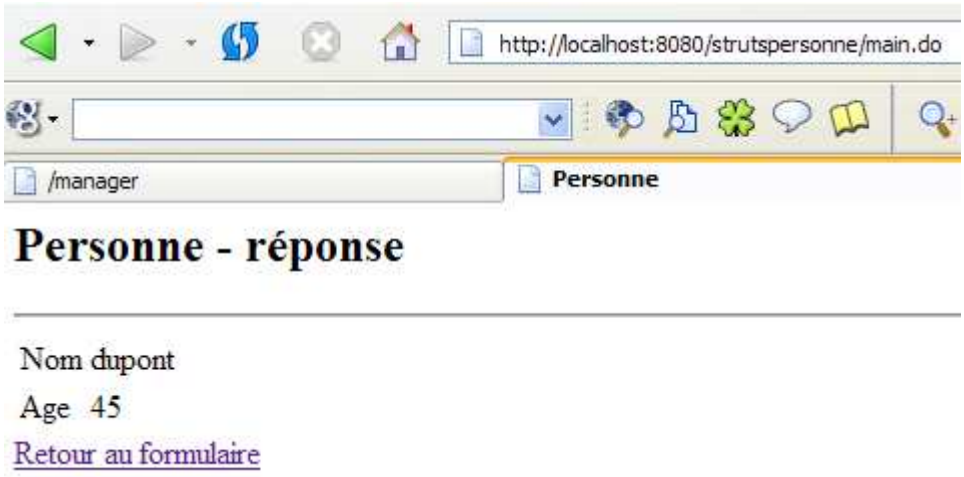
```
<%
  // on récupère les données nom, age
  String nom=(String)request.getAttribute("nom");
  String age=(String)request.getAttribute("age");
%>
```

```
<html>
  <head>
    <title>Personne</title>
```

```
  </head>
  <body>
    <h2>Personne - réponse</h2>
     <hr>
     <table>
       <tr>
         <td>Nom</td>
          <td><%= nom %>
        </tr>
       <tr>
         <td>Age</td>
          <td><%= age %>
        </tr>
     </table>
     <html:link page="/formulaire.do">
       Retour au formulaire
    </html:link>
  </body>
</html>
```

La vue récupère les informations **nom** et **age** dans les attributs de la requête qu'elle reçoit. Nous demandons le formulaire à l'URL **http://localhost:8080/strutspersonne/formulaire.do** puis nous le remplissons :

Nous utilisons le bouton [Envoyer] et nous obtenons la réponse suivante :



Explications :

- on reprendra l'explication donnée pour le test n° 2 pour le début du processus. rappelons la configuration de l'action **/main** :

```
    <action
      path="/main"
      name="frmPersonne"
      scope="session"
```

```
    validate="true"
    input="/erreurs.do"
    type="istia.st.struts.personne.FormulaireAction"
  >
    <forward name="reponse" path="/reponse.do"/>
  </action>
```

- après envoi du formulaire au contrôleur à l'URL **/main.do**, celui-ci a construit ou recyclé un objet **frmPersonne** de type **FormulaireBean** et y a mis les valeurs du formulaire
- la méthode **validate** de l' objet **frmPersonne** a été appelée. Les données étant valides, la méthode **validate** a rendu une liste **ActionErrors** vide.
- un objet **FormulaireAction** a été créé ou recyclé et sa méthode **execute** appelée. Celle-ci a rendu un objet **ActionForward** avec la clé "**reponse".**
- le contrôleur a alors envoyé la vue associée à la clé **reponse**, c.a.d. **/reponse.do** et donc **/vues/reponse.personne.jsp**.
- la vue **reponse.personne.jsp** s'est affichée avec les valeurs mises dans la requête par la méthode **execute** de l'objet **FormulaireAction**.

III.10 Conclusion

Nous avons construit une application complète mais simple. Lorsqu'on la met en oeuvre, on a de nombreuses occasions de faire des erreurs, notamment dans les fichiers XML de configuration de l'application. De prime abord, il peut paraître plus simple de construire cette application sans Struts avec une servlet et des pages JSP. Pour le débutant c'est probablement vrai. Au fil de l'expérience il devient plus simple de développer avec Struts. De nombreuses entreprises imposent la méthodologie Struts pour leurs développements web pour les raisons suivantes :

- Struts respecte le modèle MVC
- lorsque tous les développeurs travaillent de la même façon, la maintenance des applications devient plus simple parce qu'elles ont une architecture standard.

IV Les formulaires dynamiques

IV.1 Déclaration du formulaire dynamique

Nous avons vu que Struts utilisait des objets **ActionForm** pour stocker les valeurs des formulaires HTML traitées par les différentes servlets de l'application. Pour chaque formulaire de notre application, il nous faut construire une classe dérivée de **ActionForm**. Ceci peut devenir vite assez lourd puisque pour chaque champ xx, il nout écrire deux méthodes **setXx** et **getXx**. Struts nous offre la possibilité d'utiliser des formulaires

- dont la structure est déclarée dans le fichier **struts-config.xml** dans la section **<form-beans>**
- qui sont créés dynamiquement par l'environnement Struts selon la structure déclarée

Ainsi la classe utilisée pour stocker les valeurs nom et age de l'application **strutspersonne** pourrait être définie comme suit :

```
<form-beans>
  <form-bean name="frmPersonne" type="org.apache.struts.actions.DynaActionForm">
    <form-property name="nom" type="java.lang.String" initial=""/>
    <form-property name="age" type="java.lang.String" initial=""/>
  </form-bean>
</form-beans>
```

Pour chaque champ du formulaire on définit une balise <**form-property**> avec deux attributs :

- **name** : le nom du champ
- **type** : son type java

Parce que les valeurs d'un formulaire envoyées par un client web sont des chaînes de caractères, les types les plus usités sont **java.lang.String** pour les champs à valeur unique et **java.lang.String[]** pour les champs à valeurs multiples (boîtes à cocher portant le même nom, listes à sélection multiple, ...). La classe **DynactionForm** comme la classe **ActionForm** a une méthode **validate** qui ne fait rien. Pour lui faire vérifier la validité des paramètres du formulaire, il faut la dériver et écrire soi-même la méthode **validate**. C'est ce que nous ferons ici. La déclaration du formulaire sera la suivante :

```
<form-beans>
  <form-bean name="frmPersonne" type="istia.st.struts.personne.PersonneDynaForm">
    <form-property name="nom" type="java.lang.String" initial=""/>
    <form-property name="age" type="java.lang.String" initial=""/>
  </form-bean>
</form-beans>
```

La classe **istia.st.struts.personne.PersonneDynaForm** devra être écrite par nos soins.

IV.2 Écriture de la classe DynaActionForm associée au formulaire dynamique

Ci-dessus, nous avons associé la classe **istia.st.struts.personne.PersonneDynaForm** au formulaire (nom,age). Nous écrivons maintenant cette classe :

```
package istia.st.struts.personne;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class PersonneDynaForm extends DynaActionForm {
  // validation
  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
    // gestion des erreurs
    ActionErrors erreurs = new ActionErrors();
    // le nom doit être non vide
    String nom = (String)this.get("nom");
    if (nom == null || nom.trim().equals("")) {
      erreurs.add("nomvide", new ActionError("personne.formulaire.nom.vide"));
    }
    // l'âge doit être non vide
    String age = (String)this.get("age");
    if (age == null || age.trim().equals("")) {
      erreurs.add("agevide", new ActionError("personne.formulaire.age.vide"));
    }
    else {
      // l'âge doit être un entier positif
      if (!age.matches("^\\s*\\d+\\s*$")) {
        erreurs.add("ageincorrect", new ActionError("personne.formulaire.age.incorrect", age));
        // on rend la liste des erreurs
      }
```

```
    } //if
    // on rend la liste d'erreurs
    return erreurs;
  }
}//classe
```

Les points à noter sont les suivants :

- la classe dérive de **DynaActionForm**
- la méthode **validate** de la classe **DynaActionForm** a été réécrite. Lorsqu'elle est exécutée par le contrôleur Struts, l'objet **PersonneDynaForm** a été construit. Il contient un dictionnaire ayant pour clés les champs du formulaire **nom** et **age** et pour valeurs les valeurs de ces champs. Pour avoir accès à un champ au sein des méthodes de **DynaActionForm** on utilise la méthode **Object get(String nomDuChamp)**. Si on veut fixer une valeur à un champ, on pourra utiliser la méthode **void set(String nomDuChamp, Object valeur)**. On se reportera à la définition de la classe **DynaActionForm** pour daavantage de précisions.
- une fois récupérées les valeurs des champs nom et age du formulaire, la méthode **validate** ne diffère pas de celle qui avait été écrite lorsque le formulaire était associé à un objet **ActionForm**.

IV.3La nouvelle classe FormulaireAction

Le fichier de configuration définit l'action /**main** suivante :

```
...
  <form-beans>
    <form-bean name="frmPersonne" type="istia.st.struts.personne.PersonneDynaForm">
      <form-property name="nom" type="java.lang.String" initial=""/>
      <form-property name="age" type="java.lang.String" initial=""/>
    </form-bean>
  </form-beans>
....
    <action
      path="/main"
      name="frmPersonne"
      scope="session"
      validate="true"
      input="/erreurs.do"
      type="istia.st.struts.personne.FormulaireAction"
    >
      <forward name="reponse" path="/reponse.do"/>
    </action>
```

Cette définition est la même que dans l'application **strutspersonne**. L'action **/main** est réalisée par un objet de type **FormulaireAction**. Cet objet reçevait les valeurs du formulaire **frmPersonne** dans un objet de type **FormulaireBean**. Maintenant il les reçoit dans un objet de type **PersonneDynaForm**. Il nous faut écrire cette nouvelle classe :

```
package istia.st.struts.personne;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import javax.servlet.ServletException;
import istia.st.struts.personne.PersonneDynaForm;

public class FormulaireAction extends Action {

  public ActionForward execute(ActionMapping mapping, ActionForm form,
           HttpServletRequest request, HttpServletResponse response) throws IOException,ServletException {

    // on a un formulaire valide, sinon on ne serait pas arrivé là
    PersonneDynaForm formulaire=(PersonneDynaForm)form;
    request.setAttribute("nom",formulaire.get("nom"));
    request.setAttribute("age",formulaire.get("age"));
    return mapping.findForward("reponse");
  }//execute
}
```

Les points à noter sont les suivants :

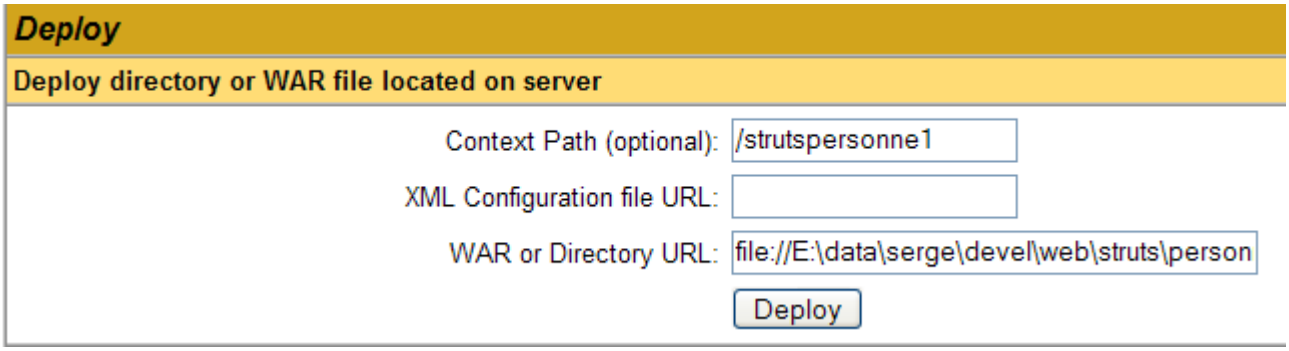
- nous devons importer la classe **PersonneDynaForm** qui contient les données du formulaire pour avoir accès à sa définition
- la méthode **execute** récupère la valeur des paramètres du formulaire avec la méthode **[DynaActionForm].get**.

Vis à vis de la classe **FormulaireAction** de l'application **strutspersonne**, seule la façon d'accéder aux valeurs du formulaire a changé.

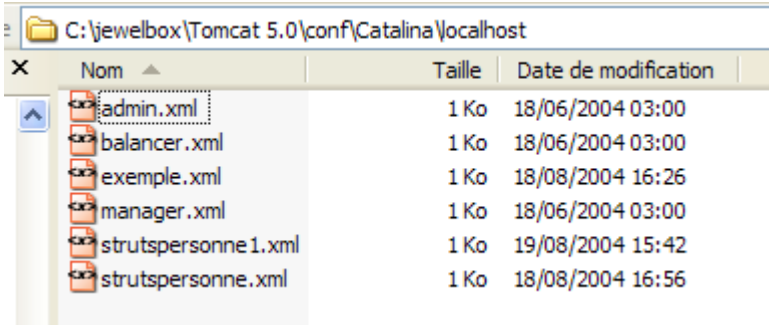
IV.4Déploiement et test de l'application strutspersonne1

IV.4.1 Création du contexte

Nous appelerons **strutspersonne1** cette nouvelle application. Nous la déployons dans Tomcat de la façon suivante (page Tomcat Manager) :



Ce déploiement donne naissance au fichier **strutspersonne1.xml** dans <tomcat>\conf\Catalina\localhost :



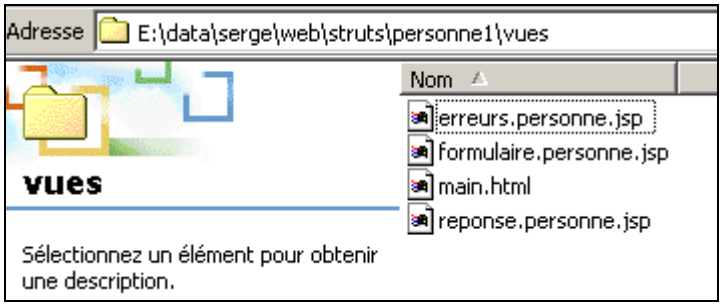
Le contenu de ce fichier est le suivant :

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="E:/data/serge/devel/web/struts/personne1" path="/strutspersonne1">
</Context>
```

On peut vérifier la validité du contexte en demandant l'URL **http://localhost:8080/strutspersonne1/**

IV.4.2 Les vues

On copiera le dossier **vues** de l'application **strutspersonne** dans le dossier de l'application **strutspersonne1**. En effet, les vues n'ont pas changé.

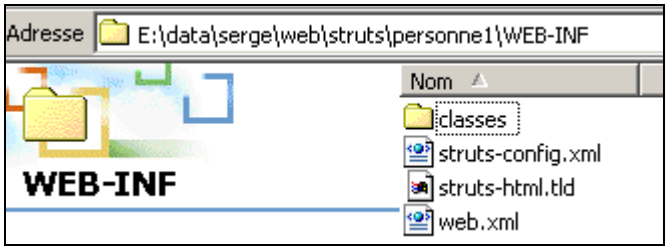


IV.4.3 Compilation des classes

Nous avons deux classes à créer : **PersonneDynaForm** et **FormulaireAction**, la dernière utilisant la première. On les compilera avec un IDE quelconque sans oublier d'ajouter les archives de Struts et Tomcat à celles explorées par le compilateur de l'IDE (cf les exemples du chapitre précédent).

IV.4.4 Le dossier WEB-INF

On copiera le dossier **WEB-INF** de l'application **strutspersonne** dans le dossier de l'application **strutspersonne1**. Quelques fichiers changent :



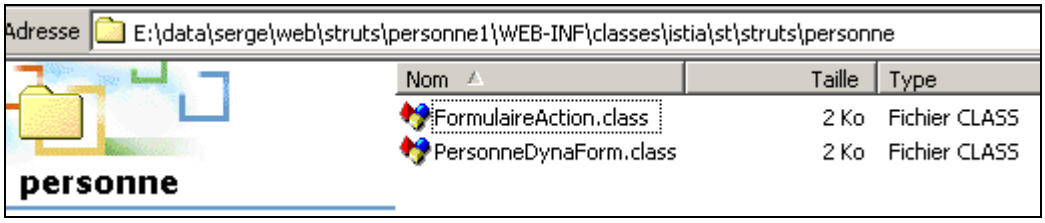
Le fichier de configuration **struts-config.xml** devient le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
```

```
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="frmPersonne" type="istia.st.struts.personne.PersonneDynaForm">
      <form-property name="nom" type="java.lang.String" initial=""/>
      <form-property name="age" type="java.lang.String" initial=""/>
    </form-bean>
  </form-beans>

  <action-mappings>
    <action
      path="/main"
      name="frmPersonne"
      scope="session"
      validate="true"
      input="/erreurs.do"
      type="istia.st.struts.personne.FormulaireAction"
    >
      <forward name="reponse" path="/reponse.do"/>
    </action>
    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/reponse"
      parameter="/vues/reponse.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
    <action
      path="/formulaire"
      parameter="/vues/formulaire.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>

  <message-resources parameter="ressources.personneressources"/>
</struts-config>
```

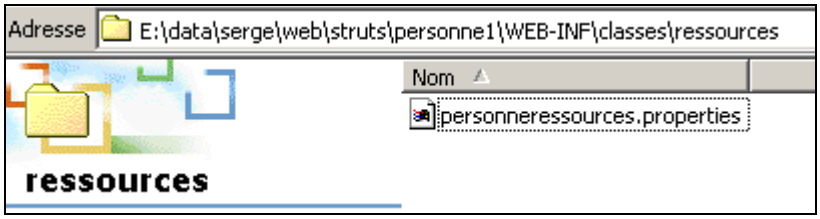
Ce fichier est identique à celui de l'application **strutspersonne** à l'exception de la définition dynamique du formulaire (partie encadrée).

Dans le dossier **WEB-INF/classes**, on mettra les classes compilées :



Dans le dossier **WEB-INF\classes\ressources**, on place le fichier des messages. Il n'a pas changé.

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge [{0}] est incorrect</li>
errors.header=<ul>
errors.footer=</ul>
```



IV.5 Tests

Nous sommes prêts pour les tests. On trouvera ci-dessous quelques copies d'écran que le lecteur est invité à reproduire.

On demande l'URL **http://localhost:8080/strutspersonne1/formulaire.do** :

On utilise le bouton [Envoyer] sans remplir les champs :

On recommence avec une erreur sur le champ age :

On obtient la réponse suivante :

On recommence en mettant cette fois-ci des valeurs correctes :

On obtient la réponse suivante :

IV.6 Conclusion

L'utilisation de formulaires dynamiques facilite l'écriture des classes de type **ActionForm** chargées de stocker les valeurs des formulaires. On peut aller un peu plus loin dans la simplification de la gestion des formulaires. Dans l'application **strutspersonne1**, nous avons créé une classe **PersonneDynaForm** pour vérifier la validité des valeurs (nom,age) du formulaire. Dans la pratique, certaines vérifications reviennent fréquemment : champ non vide, champ vérifiant une certaine expression régulière, champ entier, champ date, ... Ce type de vérification standard peut alors être demandé dans le fichier de configuration **struts-config.html**. Si toutes les vérifications à faire sont "standard", alors il n'y a plus de classe à écrire pour le formulaire. C'est ce que nous voyons maintenant.

V Formulaires dynamiques avec contraintes d'intégrité

Nous allons développer maintenant une nouvelle application appelée **strutspersonne2** utilisant

- un formulaire dynamique comme dans **strutspersonne1**
- un fichier de déclaration de contraintes d'intégrité à vérifier par les champs de ce formulaire dynamique

V.1 Déclaration des contraintes d'intégrité

La classe utilisée pour stocker les valeurs nom et age de l'application **strutspersonne1** était déclarée comme suit :

```
<form-beans>
  <form-bean name="frmPersonne" type="istia.st.struts.personne.PersonneDynaForm">
    <form-property name="nom" type="java.lang.String" initial=""/>
    <form-property name="age" type="java.lang.String" initial=""/>
  </form-bean>
</form-beans>
```

Nous avons du écrire la classe **PersonneDynaForm** pour disposer d'une méthode validate capable de vérifier que les valeurs nom et age du formulaire dynamique étaient valides. Les vérifications à faire étaient de deux sortes :

- les deux champs devaient être non vides
- le champ **age** devait vérifier le masque (expression régulière) **\s*\d+\s***

Ces deux vérifications font partie des vérifications que l'environnement **StrutsValidator** peut effectuer. Cet environnement vient avec Struts et comporte un certain nombre de classes qu'on trouve dans **commons-validator.jar** et **jakarta-oro.jar**. Si vous avez suivi la procédure, expliquée en début de document, d'installation des bibliothèques de Struts au sein de Tomcat, ces bibliothèques sont déjà disponibles à Tomcat. Assurez-vous qu'elles le sont également à Jbuilder. Ceci a également été expliqué en début de document.

La nouvelle déclaration du formulaire dans **struts-config.xml** devient la suivante :

```
<form-beans>
  <form-bean name="frmPersonne" type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="nom" type="java.lang.String" initial=""/>
    <form-property name="age" type="java.lang.String" initial=""/>
  </form-bean>
</form-beans>
```

Il y a donc très peu de différences. La classe associée au formulaire dynamique est maintenant une classe prédéfinie de StrutsValidator : **org.apache.struts.validator.DynaValidatorForm**. Le développeur n'a plus de classe à écrire. Il précise les contraintes d'intégrité que doit vérifier le formulaire dans un fichier XML séparé. Le contrôleur Struts doit connaître le nom de ce fichier. Apparaît pour cela une nouvelle section de configuration dans le fichier **struts-config.xml** :

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
     value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"
   />
</plug-in>
```

La section **<plug-in>** sert à charger une classe externe à Struts. Son attribut principal est **classname** qui indique le nom de la classe à instancier. L'objet instancié peut avoir besoin de s'initialiser. Ceci est fait au moyen de balises **set-property** qui a deux attributs :

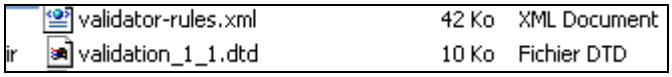
- **property** : le nom de la propriété à iitialiser
- **value** : la valeur de la propriété

Ici, la classe **DynaValidatorForm** a besoin de connaître deux informations :

- le fichier XML qui définit les contraintes d'intégrité standard que la classe sait vérifier.
- le fichier XML définissant les contraintes d'intégrité des différents formulaires dynamiques de l'application

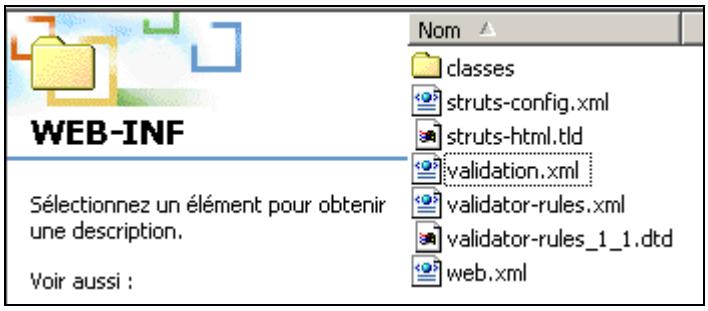
Ces deux informations sont ici données par la propriété **pathnames**. La valeur de cette propriété est une liste de fichiers XML que le validateur chargera :

- **validator-rules.xml** est le fichier définissant les contraintes d'intégrité standard. Il est livré avec Struts. On le trouvera dans **<struts>\lib** accompagné de son fichier DTD de définition :



- **validation.xml** est le fichier définissant les contraintes d'intégrité des différents formulaires dynamiques de l'application. Il est construit par le développeur. Son nom est libre.

Ces deux fichiers peuvent être placés n'importe où sous WEB-INF. Dans notre exemple, ils seront placés directement sous WEB-INF :



V.2 Écriture des contraintes d'intégrité des formulaires dynamiques

Le fichier **validation.xml** contiendra nos contraintes d'intégrité pour les champs **nom** et **age** du formulaire **frmPersonne** de type **org.apache.struts.validator.DynaValidatorForm**. Son contenu est le suivant :

```
<form-validation>
  <global>
    <constant>
      <constant-name>entierpositif</constant-name>
      <constant-value>^\s*\d+\s*$</constant-value>
    </constant>
  </global>
  <formset>
    <form name="frmPersonne">
      <field property="nom" depends="required">
        <arg0 key="personne.nom"/>
      </field>
      <field property="age" depends="required,mask">
        <arg0 key="personne.age"/>
        <var>
          <var-name>mask</var-name>
          <var-value>${entierpositif}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

Les règles d'écriture à respecter sont les suivantes :

- l'ensemble des règles se trouve dans une balise <**form-validation**>
- la balise <**global**> sert à définir des informations à portée globale, c.a.d. valable pour tous les formulaires s'il y en a plusieurs. Ici on a mis en <**global**> des constantes. Une constante est définie par son nom (balise <**constant-name**>) et sa valeur (balise <**constant-value**>). Nous définissons la constante **entierpositif** avec pour valeur l'expression régulière que doit vérifier un nombre entier positif : **^\s*\d+\s*$** (une suite de chiffres éventuellement précédée et/ou suivie d'espaces).
- la balise <**formset**> définit l'ensemble des formulaires pour lesquels il y a des contraintes d'intégrité à vérifier
- la balise <**form name="unFormulaire"**> sert à définir les contraintes d'intégrité d'un formulaire précis, celui dont le nom est donné par l'attribut **name**. Ce nom doit exister dans la liste des formulaires définis dans **struts-config.xml**. Ici le formulaire **frmPersonne** utilisé est défini dans **struts-config.xml** par la section suivante :

```
  <form-beans>
    <form-bean name="frmPersonne" type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="nom" type="java.lang.String" initial=""/>
      <form-property name="age" type="java.lang.String" initial=""/>
    </form-bean>
```

- une balise <**form**> contient autant de balises <**field**> que de contraintes d'intégrité à vérifier pour le formulaire. Une balise <**field**> a les attributs suivants :

- **property** : nom du champ du formulaire pour lequel on définit des contraintes d'intégrité
- **depends** : liste des contraintes d'intégrité à vérifier.
  - les contraintes possibles sont les suivantes : **required** (le champ doit être non vide), **mask** (la valeur du champ doit correspondre à une expression régulière définie par la variable **mask**), **integer** : la valeur du champ doit être entier, **byte** (octet), **long** (entier long), **float** (réel simple), **double** (réel double), **short** (entier court), **date** (la valeur du champ doit être une date valide), **range** (la valeur du champ doit être dans un intervalle donné), **email** : (la valeur du champ doit être une adresse mél valide), ...
  - les contraintes d'intégrité sont vérifiées dans l'ordre de l'attribut **depends**. Si une contrainte n'est pas vérifiée, les suivantes ne sont pas testées.
  - chaque contrainte est liée à un message d'erreur définie par une clé. En voici quelques-unes sous la forme **contrainte (clé)** : required (errors.required), mask (errors.invalid), integer (errors.integer), byte (errors.byte), long (errors.long), ...
  - les messages d'erreurs associés aux clés précédentes sont définies dans le fichier **validator-rules.xml** :

```
   # Struts Validator Error Messages
   errors.required={0} is required.
   errors.minlength={0} can not be less than {1} characters.
   errors.maxlength={0} can not be greater than {1} characters.
   errors.invalid={0} is invalid.

   errors.byte={0} must be a byte.
   errors.short={0} must be a short.
   errors.integer={0} must be an integer.
   errors.long={0} must be a long.
   errors.float={0} must be a float.
   errors.double={0} must be a double.

   errors.date={0} is not a date.
   errors.range={0} is not in the range {1} through {2}.
   errors.creditcard={0} is an invalid credit card number.
   errors.email={0} is an invalid e-mail address.
```

- Nous voyons ci-dessus que les messages sont en anglais. Les messages français équivalents seront placés dans le fichier des messages de l'application, fichier configuré dans la section <messages-resources> du fichier struts-config.xml :

```
  <message-resources
    parameter="ressources.personneressources"
    null="false"
  />
```

L'attribut **parameter** indique que les messages de l'application sont dans le fichier **WEB-INF/classes/ressources/personneressources.properties**.

Les nouveaux messages d'erreurs sont à placer dans ce fichier. Ils sont rajoutés à ceux existant déjà :

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge est incorrect</li>
errors.header=<ul>
errors.footer=</ul>

   # Messages d'erreur de Struts Validator
   # la clé est prédéfinie et ne doit pas être changée
   # le msg d'erreur associé est libre
   # le msg peut avoir jusqu'à 4 paramètres {0} à {3}

   errors.required=<li>Le champ [{0}] doit être renseigné.</li>
   errors.minlength=<li>Le champ [{0}] foit avoir au moins {1} caractère.</li>
   errors.maxlength=<li>Le champ [{0}] ne peut avoir plus de {1} caractères.</li>
   errors.invalid=<li>Le champ [{0}] est incorrect.</li>

   errors.byte=<li>{0} doit être un octet.</li>
   errors.short=<li>{0} doit être un entier court.</li>
   errors.integer=<li>{0} doit être un entier.</li>
   errors.long=<li>{0} doit être un entier long.</li>
   errors.float=<li>{0} doit être un réel simple.</li>
   errors.double=<li>{0} doit être un réel double.</li>

   errors.date=<li>{0} n'est pas une date valide.</li>
   errors.range=<li>{0} doit être dans l'intervalle {1} à {2}.</li>
   errors.creditcard=<li>{0} n'est pas un numéro de carte valide.</li>
   errors.email=<li>{0} n'est pas une adresse électronique valide.</li>

   personne.nom=nom
   personne.age=age
```

Prenons les contraintes d'intégrité du fichier **validation.xml** une à une pour les expliquer :

```
  <formset>
    <form name="frmPersonne">
      <field property="nom" depends="required">
        <arg0 key="personne.nom"/>
      </field>
...
    </form>
  </formset>
```

Rappelons que les contraintes d'intégrité s'appliquent aux champs **nom** et **age** d'un formulaire dynamique défini dans **struts-config.xml** :

```
    <form-bean name="frmPersonne" type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="nom" type="java.lang.String" initial=""/>
      <form-property name="age" type="java.lang.String" initial=""/>
    </form-bean>
```

Il est important que les noms du formulaire et des champs de celui-ci soient identiques dans les deux fichiers. La contrainte d'intégrité sur le champ nom (**property="nom"**) dit que le champ doit être non vide (**depends="required"**). Si ce n'est pas le cas, un objet **ActionError** de clé **errors.required** sera généré. Le message associé à cette clé se trouve dans le fichier **personneressources.properties** :

```
    errors.required=<li>Le champ [{0}] doit être renseigné.</li>
```

On voit que ce message utilise un paramètre **{0}**. La valeur de celui-ci est fixé par la balise <**arg0**> de la contrainte d'intégrité :

```
        <arg0 key="personne.nom"/>
```

Là encore, **arg0** est désigné par une clé qu'on trouve également dans le fichier des messages :

```
    personne.nom=nom
```

Si on met tout cela bout à bout, le message d'erreur généré si le champ **nom** n'est pas rempli est :

```
<li>Le champ [nom] doit être renseigné.</li>
```

Analysons maintenant la seconde contrainte, celle sur le champ **age** :

```
  <global>
    <constant>
      <constant-name>entierpositif</constant-name>
      <constant-value>^\s*\d+\s*$</constant-value>
    </constant>
  </global>
  <formset>
    <form name="frmPersonne">
...
      <field property="age" depends="required,mask">
        <arg0 key="personne.age"/>
        <var>
          <var-name>mask</var-name>
          <var-value>${entierpositif}</var-value>
        </var>
      </field>
    </form>
  </formset>
```

Il y a deux contraintes pour le champ **age** : **required** et **mask**. On peut renouveler l'explication précédente pour la contrainte **required**. On aboutit au fait que le message d'erreur associé à cette contrainte sera :

```
<li>Le champ [age] doit être renseigné.</li>
```

La seconde contrainte est **mask**. Cela signifie que le contenu du champ doit correspondre à un modèle exprimé par une expression régulière. La valeur de cette dernière est définie dans une balise <**var**> définissant une variable portant le nom **mask** (<**var-name**>) qui a pour valeur **${entierpositif}** (<**var-value**>).**entierpositif** est une constante définie dans la section <**global**> du fichier avec pour valeur l'expression régulière **^\s*\d+\s*$**. La contrainte d'intégrité est donc que l'âge doit être une suite d'un ou plusieurs chiffres éventuellement précédée ou suivie d'espaces. Si cette contrainte n'est pas vérifiée, un objet **ActionError** de clé **errors.invalid** sera généré. Dans le fichier des messages, cette clé est associée au message suivant :

```
    errors.invalid=<li>Le champ [{0}] est incorrect.</li>
```

La contrainte doit définir une valeur pour le paramètre {0}. Elle le fait avec la balise <arg0> :

```
        <arg0 key="personne.age"/>
```

Dans le fichier des messages, la clé **personne.age** est associée au message suivant :

```
   personne.age=age
```

Le message d'erreur qui sera généré si la contrainte **mask** n'est pas vérifiée est donc :

```
<li>Le champ [age] est incorrect.</li>
```

V.3 Les classes de l'application

Dans une application Struts, les classes à écrire sont celles des formulaires (**ActionForm** ou dérivée) et celles des actions (**Action** ou dérivée). Dans la nouvelle application **strutspersonne2** il n'y a plus de classe pour le formulaire. Le contenu du formulaire est défini dans **struts-config.xml** et les contraintes d'intégrité associées dans **WEB-INF/validation.xml**. Dans l'application **strutspersonne1**, la classe **FormulaireAction** de traitement du formulaire était la suivante :

```
package istia.st.struts.personne;

....

public class FormulaireAction
  extends Action {
  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse response) throws
IOException,
    ServletException {
    // on a un formulaire valide, sinon on ne serait pas arrivé là
    DynaActionForm formulaire=(DynaActionForm)form;
    request.setAttribute("nom",formulaire.get("nom"));
    request.setAttribute("age",formulaire.get("age"));
    return mapping.findForward("reponse");
  }//execute
}
```

La classe **FormulaireAction** s'attendait à recevoir un formulaire sous la forme d'un objet **DynaActionForm** (code encadré). Or dans le fichier struts-config.xml, la classe du formulaire est définie comme suit :

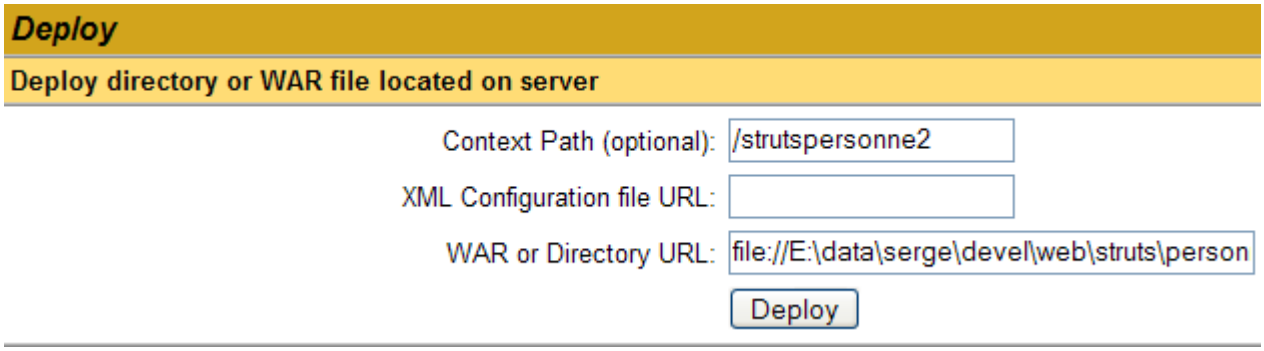
```
    <form-bean name="frmPersonne" type="org.apache.struts.validator.DynaValidatorForm">
...
    </form-bean>
```

Le formulaire sera donc placé dans un objet de type **DynaValidatorForm**. Il se trouve que cette classe est dérivée de la classe **DynaActionForm**. La méthode **execute** de notre classe **FormulaireAction** reste donc valide. Elle n'a pas à être ré-écrite.

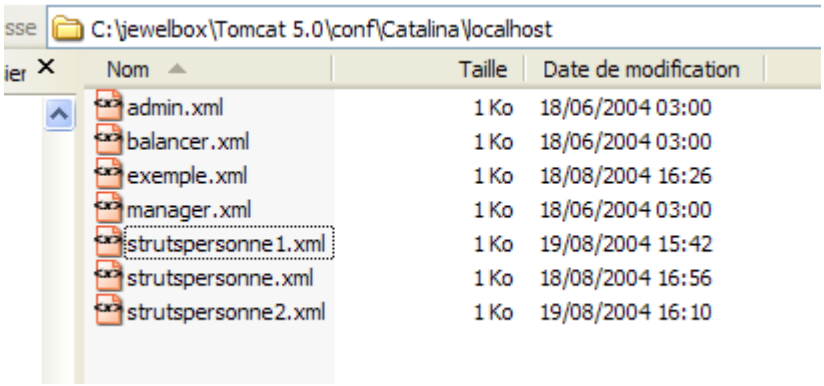
V.4 Déploiement et test de l'application /strutspersonne2

V.4.1 Création du contexte

Nous appellerons **strutspersonne2** cette nouvelle application. Nous la déployons dans Tomcat de la façon suivante (page Tomcat Manager) :



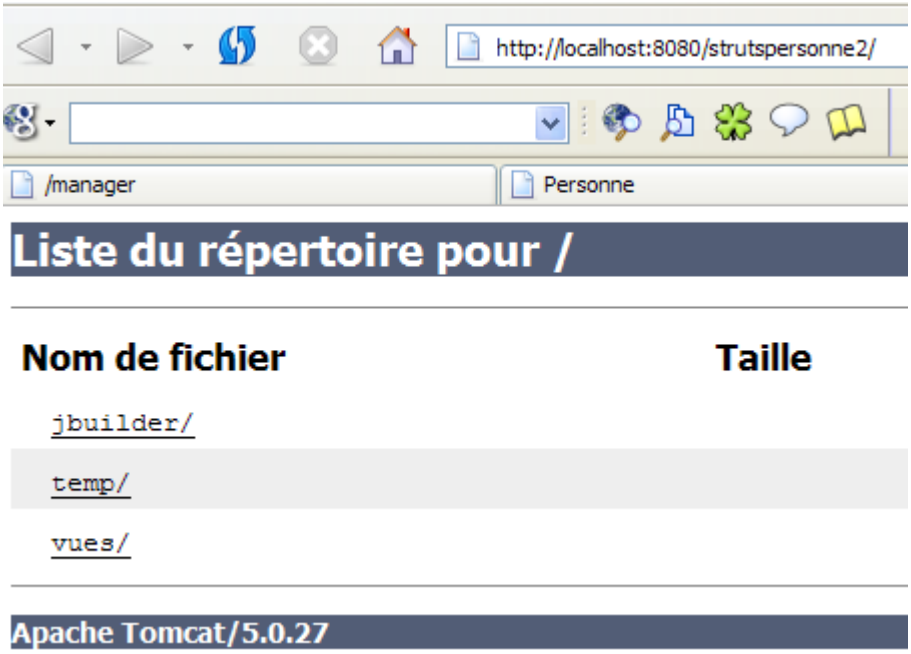
Ce déploiement donne naissance au fichier **strutspersonne2.xml** dans <tomcat>\conf\Catalina\localhost :



Le contenu de ce fichier est le suivant :

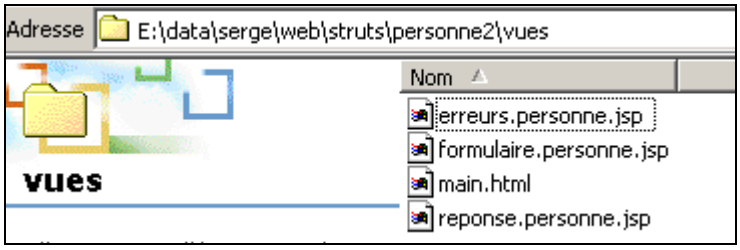
```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="E:/data/serge/devel/web/struts/personne2" path="/strutspersonne2">
</Context>
```

On peut vérifier la validité du contexte en demandant l'URL **http://localhost:8080/strutspersonne2/**



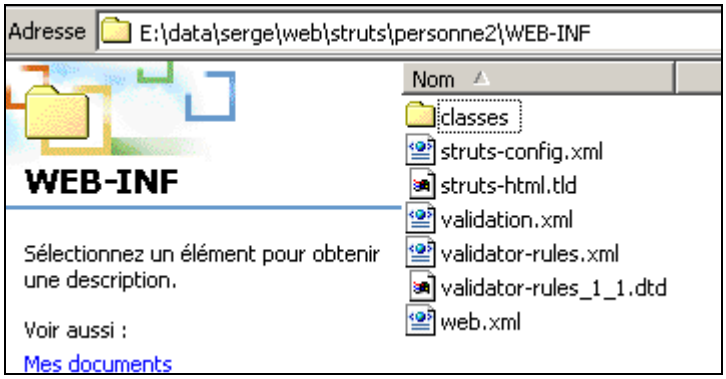
V.4.2 Les vues

On copiera le dossier **vues** de l'application **strutspersonne1** dans le dossier de l'application **strutspersonne2**. En effet, les vues n'ont pas changé.



V.4.3 Le dossier WEB-INF

On copiera le dossier **WEB-INF** de l'application **strutspersonne1** dans le dossier de l'application **strutspersonne2**. Quelques fichiers changent :



Le fichier de configuration **struts-config.xml** devient le suivant :

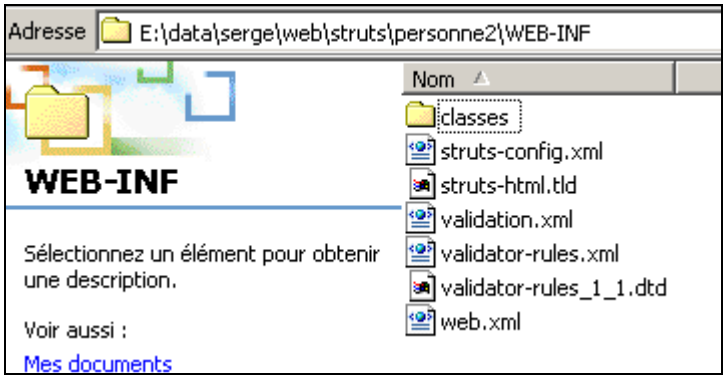
```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="frmPersonne" type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="nom" type="java.lang.String" initial=""/>
      <form-property name="age" type="java.lang.String" initial=""/>
    </form-bean>
  </form-beans>

  <action-mappings>

    <action
      path="/main"
      name="frmPersonne"
      validate="true"
      input="/erreurs.do"
      scope="session"
      type="istia.st.struts.personne.FormulaireAction"
    >
      <forward name="reponse" path="/reponse.do"/>
    </action>

    <action
      path="/erreurs"
      parameter="/vues/erreurs.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

    <action
      path="/reponse"
      parameter="/vues/reponse.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

    <action
      path="/formulaire"
      parameter="/vues/formulaire.personne.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
  </action-mappings>

  <message-resources
    parameter="ressources.personneressources"
     null="false"
   />

  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
      property="pathnames"
       value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"
     />
  </plug-in>

</struts-config>
```

Ce fichier est identique à celui de l'application **strutspersonne1** à l'exception de la définition dynamique du formulaire et de l'insertion du plugin de validation (parties encadrées).

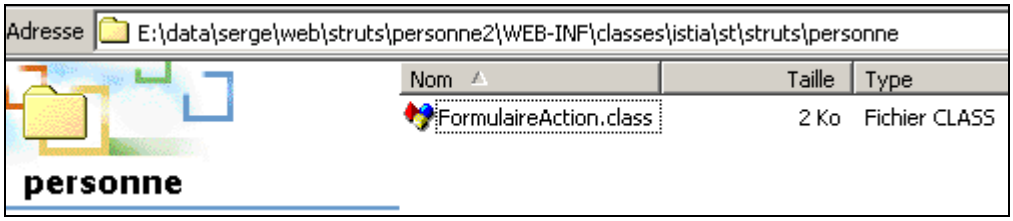
On ajoute dans **WEB-INF** le fichier de validation **validation.xml** suivant :

```
<form-validation>
  <global>
    <constant>
      <constant-name>entierpositif</constant-name>
      <constant-value>^\s*\d+\s*$</constant-value>
    </constant>
  </global>
  <formset>
    <form name="frmPersonne">
      <field property="nom" depends="required">
        <arg0 key="personne.nom"/>
      </field>
      <field property="age" depends="required,mask">
        <arg0 key="personne.age"/>
        <var>
          <var-name>mask</var-name>
          <var-value>${entierpositif}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

On ajoute dans **WEB-INF** les fichiers **validator-rules.xml**, **validator-rules_1_1.dtd** qu'on trouvera dans **<struts>\lib** :



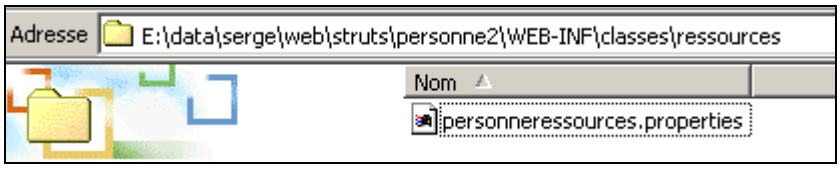
Dans le dossier **WEB-INF/classes**, on n'a plus qu'une seule classe :



Dans le dossier **WEB-INF\classes\ressources**, on trouve le fichier des messages **personneressources.properties** suivant :

```
personne.formulaire.nom.vide=<li>Vous devez indiquer un nom</li>
personne.formulaire.age.vide=<li>Vous devez indiquer un age</li>
personne.formulaire.age.incorrect=<li>L'âge est incorrect</li>
errors.header=<ul>
errors.footer=</ul>

   # Messages d'erreur de Struts Validator
   # la clé est prédéfinie et ne doit pas être changée
   # le msg d'erreur associé est libre
   # le msg peut avoir jusqu'à 4 paramètres {0} à {3}

   errors.required=<li>Le champ [{0}] doit être renseigné.</li>
   errors.minlength=<li>Le champ [{0}] foit avoir au moins {1} caractère.</li>
   errors.maxlength=<li>Le champ [{0}] ne peut avoir plus de {1} caractères.</li>
   errors.invalid=<li>Le champ [{0}] est incorrect.</li>

   errors.byte=<li>{0} doit être un octet.</li>
   errors.short=<li>{0} doit être un entier court.</li>
   errors.integer=<li>{0} doit être un entier.</li>
   errors.long=<li>{0} doit être un entier long.</li>
   errors.float=<li>{0} doit être un réel simple.</li>
```

```
  errors.double=<li>{0} doit être un réel double.</li>

  errors.date=<li>{0} n'est pas une date valide.</li>
  errors.range=<li>{0} doit être dans l'intervalle {1} à {2}.</li>
  errors.creditcard=<li>{0} n'est pas un numéro de carte valide.</li>
  errors.email=<li>{0} n'est pas une adresse électronique valide.</li>

  personne.nom=nom
 personne.age=age
```



V.5 Tests

Nous sommes prêts pour les tests. On trouvera ci-dessous quelques copies d'écran que le lecteur est invité à reproduire.

On demande l'URL **http://localhost:8080/strutspersonne2/formulaire.do** :

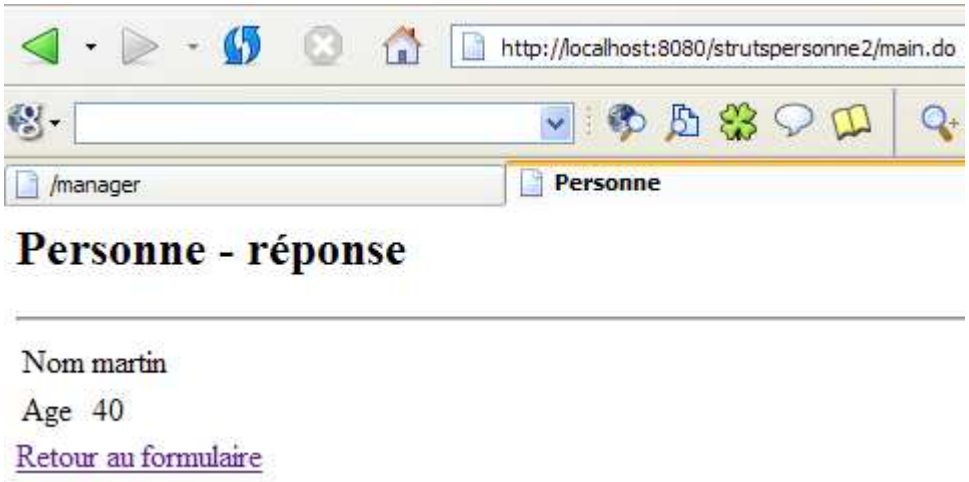
On utilise le bouton [Envoyer] sans remplir les champs :

On recommence avec une erreur sur le champ age :

On obtient la réponse suivante :

On recommence en mettant cette fois-ci des valeurs correctes :

On obtient la réponse suivante :



V.6 Conclusion

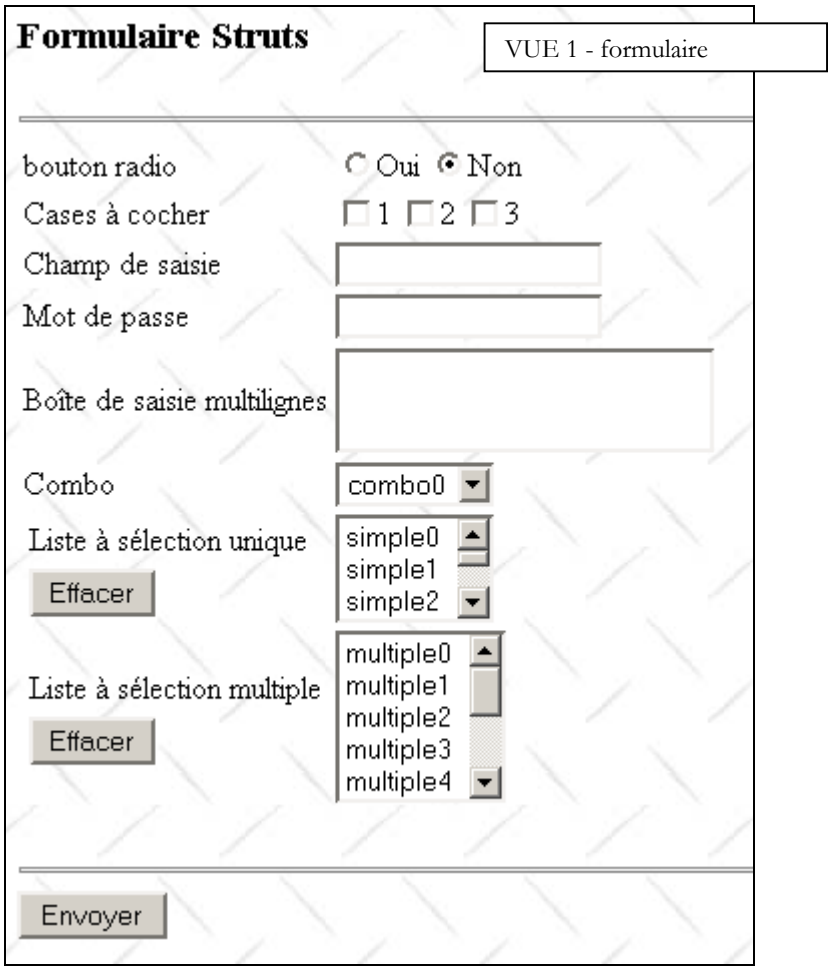
Nous avons montré que l'utilisation de formulaires dynamiques pour lesquelles les contraintes d'intégrité sont "standard" évite la création de classes pour les représenter. Si les contraintes d'intégrité sortent du standard alors on est amené de nouveau à créer des classes pour vérifier ces nouvelles contraintes.

VI Formulaires HTML

Jusqu'à maintenant, nous avons utilisé un unique formulaire ne comprenant que deux champs de saisie. Nous nous proposons ici de créer et traiter un formulaire utilisant les composants graphiques usuels (boutons radio, cases à cocher, champs de saisie, combobox, listes).

VI.1 Les vues de l'application

L'application n'aura que deux vues. La première présente un formulaire vierge :



Cette première vue appelée **formulaire.jsp** nous permettra de mettre en oeuvre différentes balises de la bibliothèque **struts-html**. L'utilisateur remplit le formulaire :

Formulaire Struts

bouton radio — Oui / Non

Cases à cocher — 1 / 2 / 3

Champ de saisie — qqs mots

Mot de passe — ****

Boîte de saisie multilignes — ligne1 ligne2

Combo — combo2

Liste à sélection unique — simple0, simple1, simple2 — Effacer

Liste à sélection multiple — multiple0, multiple1, multiple2, multiple3, multiple4 — Effacer

Envoyer

Le bouton [Envoyer] permet d'avoir une confirmation des valeurs saisies. Ce sera la deuxième vue :

Confirmation des valeurs saisies

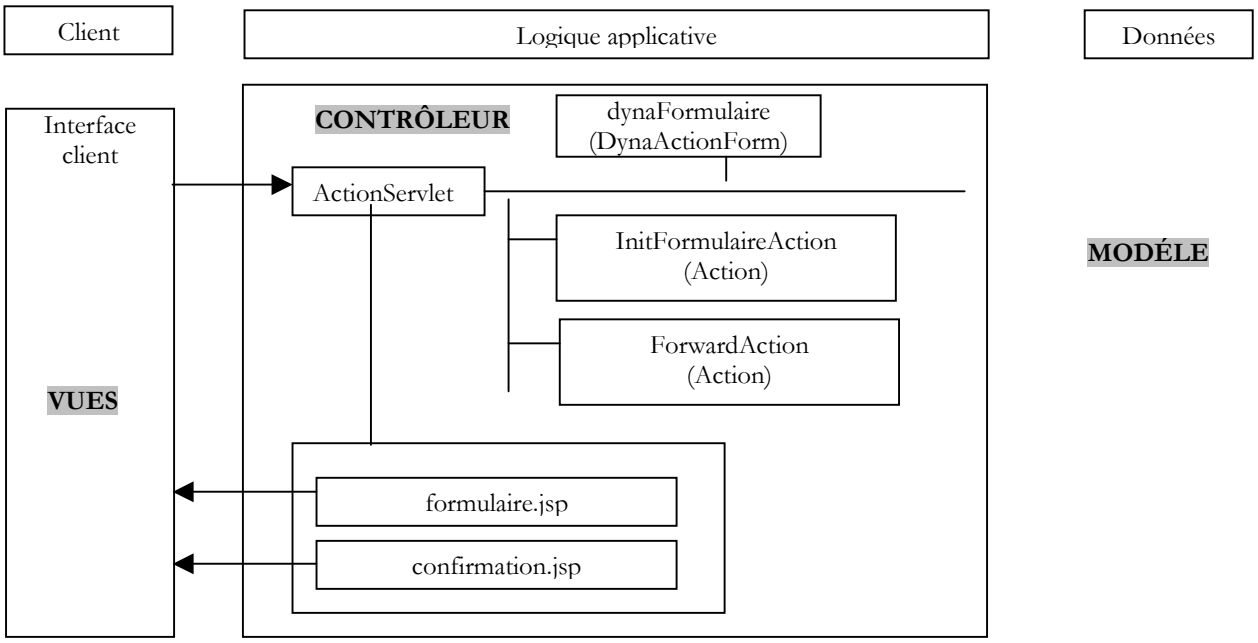
VUE 2 - confirmation

Bouton radio	oui
Case à cocher chk1	on
Case à cocher chk2	off
Case à cocher chk3	on
Champ de saisie	qqs mots
Mot de passe	mdp
Boîte de saisie	ligne1 ligne2
combo	combo2
liste simple	simple1
liste multiple[0]	multiple1
liste multiple[1]	multiple3
liste multiple[2]	multiple4

Retour au formulaire

Cette deuxième vue nous permettra d'utiliser deux autres bibliothèques de balises **struts-bean** et **struts-logic**. Le lien [Retour au formulaire] nous permet de retrouver le formulaire tel que l'on l'a rempli. On retrouve alors la première vue.

VI.2L'architecture de l'application

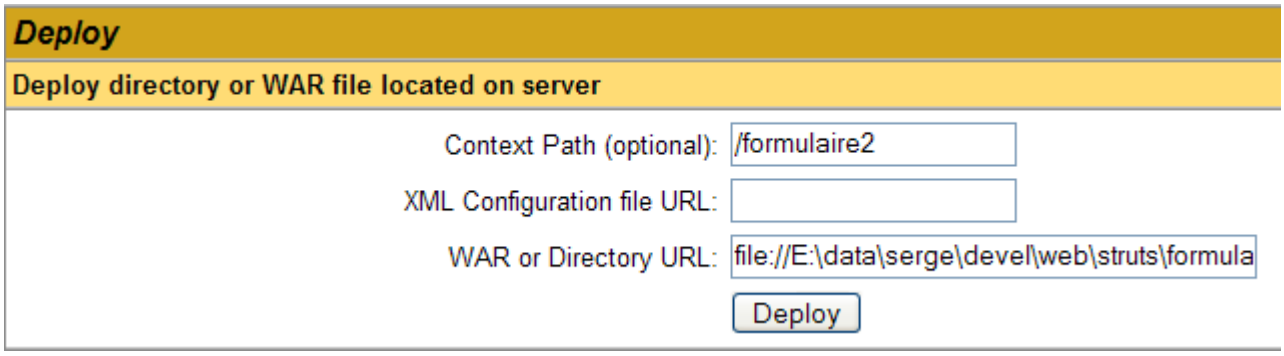


- le formulaire (vue 1) sera représenté par un objet Struts dynamique appelé **dynaFormulaire** de type dérivé de **DynaActionForm**. Il sera affiché par la vue **formulaire.jsp**.
- l'action Struts **InitFormulaireAction** aura pour but d'acquérir les données nécessaires à l'affichage du formulaire
- le formulaire rempli sera traité par une action **ForwardAction** qui se contera de rediriger la requête vers la seconde vue **confirmation.jsp**. Celle-ci se chargera d'afficher les valeurs du formulaire.

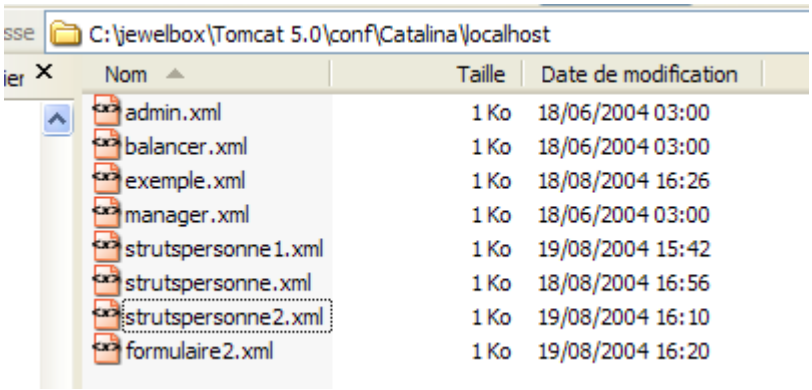
VI.3La configuration de l'application

VI.3.1 Création du contexte

Nous appellerons **/formulaire2** cette nouvelle application. Nous la déployons dans Tomcat de la façon suivante (page Tomcat Manager) :



Ce déploiement donne naissance au fichier **formulaire2.xml** dans <tomcat>\conf\Catalina\localhost :



Le contenu de ce fichier est le suivant :

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="E:/data/serge/devel/web/struts/formulaire2" path="/formulaire2">
</Context>
```

On peut vérifier la validité du contexte en demandant l'URL **http://localhost:8080/formulaire2/**

VI.3.2 Le fichier web.xml

Le fichier de configuration **web.xml** de l'application sera le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

```
</web-app>
```

Il reste donc identique à celui des applications précédemment étudiées.

VI.3.3 Le fichier struts-config.xml

Le fichier **struts-config.xml** sera lui le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
      <form-property name="opt" type="java.lang.String" initial="non"/>
      <form-property name="chk1" type="java.lang.String"/>
      <form-property name="chk2" type="java.lang.String"/>
      <form-property name="chk3" type="java.lang.String"/>
      <form-property name="champSaisie" type="java.lang.String" initial=""/>
      <form-property name="mdp" type="java.lang.String" initial=""/>
      <form-property name="boiteSaisie" type="java.lang.String" initial=""/>
      <form-property name="combo" type="java.lang.String"/>
      <form-property name="listeSimple" type="java.lang.String"/>
      <form-property name="listeMultiple" type="java.lang.String[]"/>
      <form-property name="secret" type="java.lang.String" initial="xxx"/>
      <form-property name="valeursCombo" type="java.lang.String[]" />
      <form-property name="valeursListeSimple" type="java.lang.String[]" />
      <form-property name="valeursListeMultiple" type="java.lang.String[]"/>
    </form-bean>
  </form-beans>

  <action-mappings>
    <action
      path="/confirmation"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      parameter="/vues/confirmation.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

    <action
      path="/init"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      type="istia.st.struts.formulaire.InitFormulaireAction"
    >
      <forward name="afficherFormulaire" path="/vues/formulaire.jsp"/>
    </action>

    <action
      path="/affiche"
      parameter="/vues/formulaire.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

  </action-mappings>

    <message-resources
    parameter="ApplicationResources"
      null="false"/>

</struts-config>
```

On y retrouve les trois grandes sections :
- la déclaration des **formulaires** dans la section <**form-beans**>
- la déclaration des **actions** dans la section <**action-mappings**>
- la déclaration du **fichier de ressources** dans <**message-ressources**>

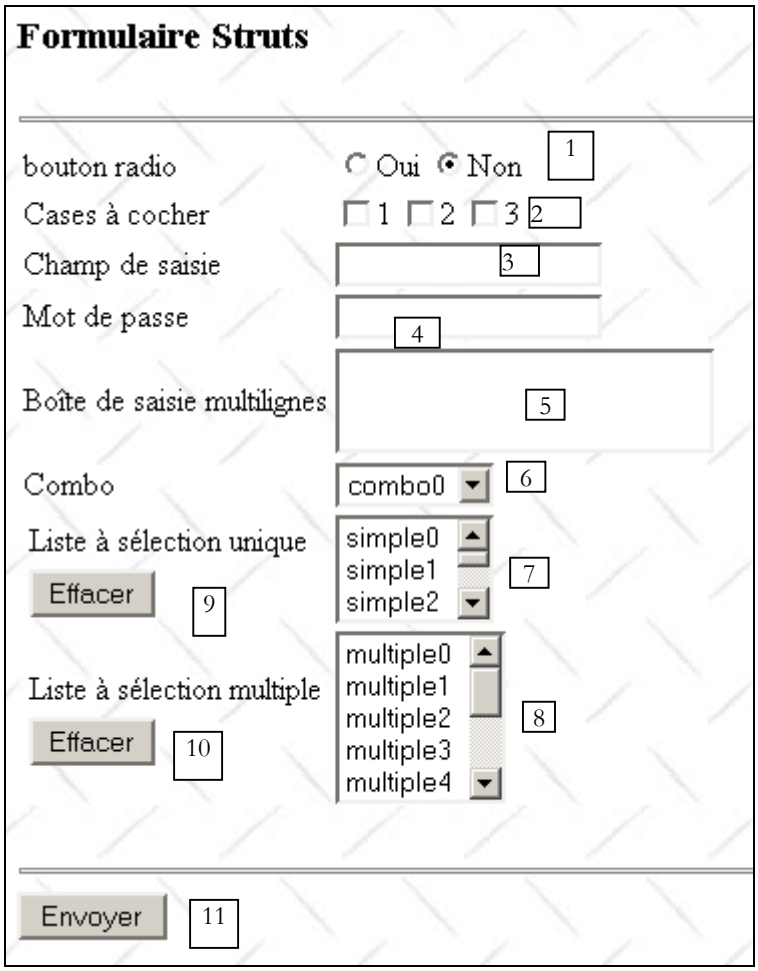
VI.3.4 Les objets (beans) formulaires de l'application

Les objets utilisés pour représenter les formulaires HTML de l'application sont des objets de type **ActionForm** ou dérivé (**DynaActionForm**, **DynaValidatorForm**, ...). On les appelle des beans parce que leur construction suit les règles des JavaBeans.

Il n'y a qu'un bean formulaire dans notre application., appelé **dynaFormulaire** et de type dérivé de **DynaActionForm**. Il sera utilisé dans les situations suivantes :

- contenir les données nécessaires à l'affichage de la vue n° 1
- récupérer les valeurs du formulaire de la vue n°1 lorsque l'utilisateur va le valider (submit)
- contenir les données nécessaires à l'affichage de la vue n° 2

La structure du bean **dynaFormulaire** est intimement lié au formulaire de la vue n° 1. Etudions celui-ci :



N°	Type HTML	Rôle
1	<input name="opt" type="radio" value="oui"> <input name="opt" type="radio" value="non">	groupe de boutons radio liés entre-eux (même nom)
2	<input name="chk1" type="radio" value="on"> <input name="chk2" type="radio" value="on"> <input name="chk3" type="radio" value="on">	groupes de cases à cocher indépendantes (pas le même nom)
3	<input type="text " name="champSaisie" >	un champ de saisie
4	<input type="password " name="mdp" >	un champ mot de passe
5	<textarea name="boiteSaisie">...</textarea>	un champ de saisie multi-lignes
6	<select name="combo" size="1">..</select>	un combo
7	<select name="listeSimple" size="3">..</select>	une liste à sélection unique
8	<select name="listeMultiple" size="3" multiple>..</select>	une liste à sélection multiple
9	<input type="button" value="Effacer" onclick='effacerListe("listeSimple")'>	bouton permettant de désélectionner les éléments sélectionnés dans listeSimple (7)
10	<input type="button" value="Effacer" onclick='effacerListe("listeMultiple")'>	bouton permettant de désélectionner les éléments sélectionnés dans listeMultiple (8)
11	<input type="submit" value="Envoyer">	bouton submit du formulaire
12	<input type="hidden" name="secret" value="...">	un champ caché

Distinguons plusieurs cas :

1. l'objet **dynaFormulaire** est utilisé pour contenir les valeurs du formulaire HTML ci-dessus qui sera posté par le bouton [Envoyer]. Il faut donc qu'il ait les mêmes champs que dans le formulaire HTML. Le type du champ est fixé par la règle suivante :

- si le champ HTML ne fournit qu'une valeur, alors le champ de **dynaFormulaire** sera de type **java.lang.String**
- si le champ HTML fournit plusieurs valeurs, alors le champ de **dynaFormulaire** sera de type **java.lang.String[]**

Dans le formulaire HTML ci-dessus, seul le champ **listeMultiple** est susceptible d'être associé à plusieurs valeurs (celles sélectionnées par l'utilisateur). Donc une première définition de l'objet **dynaFormulaire** serait la suivante :

```
<form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
  <form-property name="opt" type="java.lang.String" initial="non"/>
  <form-property name="chk1" type="java.lang.String"/>
  <form-property name="chk2" type="java.lang.String"/>
  <form-property name="chk3" type="java.lang.String"/>
  <form-property name="champSaisie" type="java.lang.String" initial=""/>
  <form-property name="mdp" type="java.lang.String" initial=""/>
  <form-property name="boiteSaisie" type="java.lang.String" initial=""/>
  <form-property name="combo" type="java.lang.String"/>
  <form-property name="listeSimple" type="java.lang.String"/>
  <form-property name="listeMultiple" type="java.lang.String[]"/>
  <form-property name="secret" type="java.lang.String" initial="xxx"/>
</form-bean>
```

Comment se fera le remplissage de **dynaFormulaire** avec les valeurs du formulaire HTML envoyées par le client web ?

`opt`	le champ **opt** recevra la valeur "**oui**" si le champ HTML **<input type="radio" name="opt" value="oui">** a été coché, la valeur "**non**" si c'est le champ **<input type="radio" name="opt" value="non">** qui a été coché.
`chk1`	le champ **chk1** recevra la valeur "**on**" si le champ HTML **<input name="chk1" type="radio" value="on">** a été coché, rien sinon. Dans ce dernier cas, le champ **chk1** conservera sa précédente valeur.
`chk2`	idem
`chk3`	idem
`champSaisie`	le champ **champSaisie** recevra le texte mis par l'utilisateur dans le champ HTML **<input type="text" name="champSaisie">**. Ce texte peut être éventuellement la chaîne vide.
`mdp`	le champ **mdp** recevra le texte mis par l'utilisateur dans le champ HTML **<input type="password" name="mdp">**. Ce texte peut être éventuellement la chaîne vide.
`boiteSaisie`	le champ **boiteSaisie** recevra le texte mis par l'utilisateur dans le champ HTML **<textarea name="boiteSaisie">...</textarea>**. Ce texte forme une seule chaîne de caractères, formée des lignes tapées par l'utilisateur, séparées les unes des autres par la séquence de caractères "\r\n". Le texte obtenu peut être éventuellement la chaîne vide.
`combo`	le champ **combo** recevra l'option sélectionnée par l'utilisateur dans le champ HTML **<select name="combo" size="1">..</select>**. L'option sélectionnée est celle qui apparaît dans le combo. Si l'option HTML sélectionnée est du type **<option value="XX">YY</option>**, le champ **combo** recevra la valeur "**XX**". Si l'option HTML sélectionnée est du type **<option>YY</option>**, le champ **combo** recevra la valeur "**YY**".
`listeSimple`	le champ **listeSimple** recevra l'option sélectionnée par l'utilisateur dans le champ HTML **<select name="listeSimple" size="..">..</select>** s'il y en a une. S'il n'y en a pas, le champ **listeSimple** ne recevra aucune valeur et gardera sa valeur précédente. La valeur affectée réellement au champ **listeSimple** suit les règles indiquées pour le combo.
`listeMultiple`	le champ **listeMultiple** de type **String[]** recevra les options sélectionnées par l'utilisateur dans le champ HTML **<select name="listeMultiple" size=".." multiple>..</select>** s'il y en a. S'il n'y en a pas, le tableau **listeMultiple** ne recevra aucune valeur et son contenu restera inchangé. Les valeurs affectées réellement au tableau **listeMultiple** suivent les règles indiquées pour le combo.
`secret`	le champ **secret** recevra la valeur **XX** du champ HTML **<input type="hidden" name="secret" value="XX">**. Ce texte peut être éventuellement la chaîne vide.

2. l'objet **dynaFormulaire** est utilisé pour donner le contenu initial de la vue n° 1. Les valeurs des champs précédents seront utilisées aux fins suivantes :

`opt`	devra avoir la valeur "**oui**" ou "**non**" pour indiquer quel bouton radio cocher
`chk1`	si **chk1** a la valeur "**on**" la case à cocher sera cochée sinon elle ne le sera pas
`chk2`	idem
`chk3`	idem
`champSaisie`	la valeur du champ sera affichée dans la zone de saisie **champSaisie**
`mdp`	la valeur du champ sera affichée dans la zone de saisie **mdp**
`boiteSaisie`	la valeur du champ sera affichée dans la zone de saisie **boiteSaisie**
`combo`	la valeur de ce champ indique quel élément du combo doit être sélectionné à l'affichage du formulaire
`listeSimple`	idem
`listeMultiple`	les valeurs du tableau **listeMultiple** indiquent quels éléments de la liste multiple doivent être

sélectionnés à l'affichage du formulaire

`secret`	la valeur du champ sera affectée à l'attribut **value** du champ HTML **secret**.

La vue n° 1 a besoin d'autres informations :

- la liste des valeurs à afficher dans la liste **combo**
- la liste des valeurs à afficher dans la liste **listeSimple**
- la liste des valeurs à afficher dans la liste **listeMultiple**

Il existe plusieurs façons de fournir ces informations à la vue. Des tableaux placés dans la requête passée à la vue feraient par exemple l'affaire. Ici, nous plaçons ces tableaux dans le bean **dynaFormulaire** :

```
    <form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
...
      <form-property name="valeursCombo" type="java.lang.String[]" />
      <form-property name="valeursListeSimple" type="java.lang.String[]" />
      <form-property name="valeursListeMultiple" type="java.lang.String[]"/>
    </form-bean>
```

Le formulaire **dynaFormulaire** sera initialisé par l'action /**init** qui fera appel à un objet dérivé de **Action** appelé **InitFormulaireAction**. C'est cet objet qui se chargera de créer les trois tableaux nécessaires à l'affichage des trois listes et de les mettre dans le bean **dynaFormulaire**. Le fichier de configuration donne à ce bean une portée égale à la session. Ceci entraîne que le contrôleur Struts placera ce bean dans la session. Nous n'aurons ainsi pas besoin de le régénérer entre deux cycles demande-réponse. Aussi l'action /**init** ne sera-t-elle appelée **qu'une fois**.

3. l'objet **dynaFormulaire** est également utilisé pour donner le contenu de la vue n° 2. Celle-ci se contente d'en afficher les valeurs.

VI.3.5 Les actions de l'application

Les actions sont assurées par des objets de type **Action** ou dérivé. La configuration des actions est faite à l'intérieur des balises <**action-mappings**> :

```
  <action-mappings>
    <action
      path="/confirmation"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      parameter="/vues/confirmation.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

    <action
      path="/init"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      type="istia.st.struts.formulaire.InitFormulaireAction"
    >
      <forward name="afficherFormulaire" path="/vues/formulaire.jsp"/>
    </action>

    <action
      path="/affiche"
      parameter="/vues/formulaire.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

  </action-mappings>
```

Notons qu'une action peut ne pas avoir de formulaire associé. C'est le cas ci-dessus, de l'action /**affiche**. Avant de détailler chaque action, rappelons le fonctionnement du couple action-formulaire associé au sein d'une balise <**action**> :

- une action commence avec une requête d'un client web et se termine par l'envoi d'une page réponse. C'est le cycle demande-réponse du client-serveur web. La requête est reçue par le contrôleur Struts de type **ActionServlet** ou dérivé. C'est aussi ce contrôleur qui envoie la réponse.
- le bean du formulaire, de type **ActionForm** ou dérivé est créé s'il n'existe pas déjà. Le contrôleur regarde s'il peut trouver un objet de nom **name** dans la portée indiquée par **scope**. Si oui, il l'utilise. Si non, il le crée et le met dans la portée indiquée par **scope,** associé à l'attribut indiqué par **name**.

Dans l'exemple de l'action /**init** par exemple, le contrôleur fera un **request.getSession().getAttribute("dynaFormulaire")** pour savoir si **dynaFormulaire** a déjà été créé ou non. Si ce n'est pas le cas, il le créera et le mettra dans la session par une instruction du genre **request.getSession().setAttribute("dynaFormulaire",new DynaFormulaire(...))**.

- le contrôleur va également chercher un objet **Action** du type indiqué par l'attribut **type**. S'il ne le trouve pas, il le crée, sinon il l'utilise.
- la méthode **reset** du bean formulaire va être appelée. Celui-ci, sauf lors de sa création initiale, est recyclé. Il contient alors des données qu'on peut vouloir "nettoyer". C'est dans la méthode **reset** du bean **ActionForm** ou dérivé que l'on fera cela.
- si l'action est la cible d'un formulaire posté, alors les valeurs du formulaire qui se trouvent dans la requête du client sont copiées dans les champs de même nom du bean formulaire. On notera que la méthode **reset** a été appelée avant cette recopie.
- si la configuration précise l'attribut **validate="true"**, la méthode **validate** du bean formulaire sera appelée. Celle-ci doit alors vérifier les données du bean. Cette vérification n'a lieu le plus souvent que lorsque le formulaire vient de recevoir de nouvelles données via un formulaire posté et qu'on veut vérifier la validité de ces données. Cette méthode rend au contrôleur une éventuelle liste d'erreurs dans un objet **ActionErrors**.
- si l'objet **ActionErrors** est non vide, le contrôleur fait afficher la vue précisée par l'attribut **input** de l'action.
- si la validation des données n'est pas demandée ou si elle s'est bien passée, le contrôleur fait exécuter la méthode **execute** de l'objet de type **Action** ou dérivé associé à l'action en cours. C'est dans cette méthode que la demande du client web est traitée. La méthode **execute** renvoie un objet **ActionForward** indexé par des clés de type chaîne de caractères. Ces clés sont celles déclarées par les balises **forward** de l'action configurée. Dans notre exemple, l'action /**init** a une seule balise **forward**. Elle associe la clé "**afficherFormulaire**" à la vue **formulaire.jsp**.
- le contrôleur fait afficher la vue à laquelle est associée la clé reçue. Cette vue peut être en fait une action, auquel cas le processus précédent est répété.

L'action /init

```
<action
  path="/init"
  name="dynaFormulaire"
  validate="false"
  scope="session"
  type="istia.st.struts.formulaire.InitFormulaireAction"
>
  <forward name="afficherFormulaire" path="/vues/formulaire.jsp"/>
</action>
```

- l'action /**init** a lieu normalement une fois lors du 1er cycle demande-réponse lorsque l'utilisateur demande l'URL **http://localhost:8080/formulaire2/init.do**
- l'objet **dynaFormulaire** est créé ou recyclé. Il est récupéré (recyclage) ou placé (création) dans la session comme le demande l'attribut **scope**.
- sa méthode **reset** est appelée. Que doit-elle faire ? Normalement, on remet les champs de l'objet **ActionForm** à des valeurs par défaut. Seulement ici, nous ne le ferons pas, car l'objet **dynaFormulaire** est placé dans la session (scope="session"). Les champs de **dynaFormulaire** doivent donc garder leurs valeurs. Quelles sont celles-ci lors de la création initiale de l'objet **dynaFormulaire** ? Il y a deux cas :

  - le champ a une valeur initiale indiquée dans le fichier de configuration :

```
  <form-property name="opt" type="java.lang.String" initial="non"/>
```

  Dans ce cas, le contrôleur Struts créera ce champ avec cette valeur initiale.

  - le champ n'a pas de valeur initiale par configuration : les règles d'initialisation de Java s'appliquent. En général, les champs numériques auront la valeur zéro, les chaînes de caractères auront pour valeur la chaîne vide, les autres objets ayant la valeur **null**.

Regardons la configuration initiale de **dynaFormulaire** :

```
<form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
  <form-property name="opt" type="java.lang.String" initial="non"/>
  <form-property name="chk1" type="java.lang.String"/>
  <form-property name="chk2" type="java.lang.String"/>
  <form-property name="chk3" type="java.lang.String"/>
  <form-property name="champSaisie" type="java.lang.String" initial=""/>
  <form-property name="mdp" type="java.lang.String" initial=""/>
  <form-property name="boiteSaisie" type="java.lang.String" initial=""/>
  <form-property name="combo" type="java.lang.String"/>
  <form-property name="listeSimple" type="java.lang.String"/>
  <form-property name="listeMultiple" type="java.lang.String[]"/>
  <form-property name="secret" type="java.lang.String" initial="xxx"/>
  <form-property name="valeursCombo" type="java.lang.String[]" />
  <form-property name="valeursListeSimple" type="java.lang.String[]" />
  <form-property name="valeursListeMultiple" type="java.lang.String[]"/>
</form-bean>
```

Les valeurs initiales des champs de **dynaFormulaire** après sa création seront les suivantes :

Champ	Valeur initiale
`opt`	"non"
`chk1, chk2, chk3`	chaîne vide
`champSaisie`	chaîne vide
`mdp`	chaîne vide
`boiteSaisie`	chaîne vide
`combo`	chaîne vide
`listeSimple`	chaîne vide
`listeMultiple`	tableau de chaînes vides
`secret`	"xxx"
`valeursCombo, listeSimple, listeMultiple`	tableau de chaînes vides

- on pourrait imaginer que la méthode **reset** de **dynaFormulaire** donne des valeurs aux trois tableaux devant alimenter les trois listes de la vue **formulaire.jsp**. Ce serait possible ici car les données de ces trois tableaux seront générées de façon arbitraire. Néanmoins, le cas le plus usuel est que ces données proviennent du modèle de l'application, le M de MVC. Ici, nous prendrons une position intermédiaire, pour ne pas compliquer l'exemple, en faisant générer ces valeurs par l'action **InitFormulaireAction** donc par le C de MVC.
- il n'y a pas obligation d'écrire une méthode **reset** à **dynaFormulaire**, la classe **ActionForm** dont elle dérive ayant une telle méthode qui ne fait rien (pas d'initialisations).
- une fois la méthode **reset** de **dynaFormulaire** appelée, le contrôleur regarde l'attribut **validate** de l'action. Ici, il a la valeur "false". La méthode **validate** de **dynaFormulaire** ne sera pas appelée.
- l'objet **InitFormulaireAction** est créé ou recyclé s'il existait déjà et sa méthode **execute** lancée. C'est elle qui va affecter des valeurs arbitraires aux trois tableaux de **dynaFormulaire** : **valeursCombo**, **valeursListeSimple** et **valeursListeMultiple**. La méthode rend un **ActionForward** de clé "**afficherFormulaire**".
- le contrôleur fait afficher la vue **/vues/formulaire.jsp** qui a été associée à la clé "**afficherFormulaire**" par une balise **forward** de l'action **/init**.

L'action /confirmation

```
<action
  path="/confirmation"
  name="dynaFormulaire"
  validate="false"
  scope="session"
  parameter="/vues/confirmation.jsp"
  type="org.apache.struts.actions.ForwardAction"
/>
```

- l'action /**confirmation** a lieu lorsque l'utilisateur appuie sur le bouton [Envoyer] de la vue n° 1. Le navigateur "poste" alors au contrôleur Struts le formulaire rempli par l'utilisateur.
- l'objet **dynaFormulaire** est pris dans la session
- sa méthode **reset** est appelée. Une fois qu'elle aura été appelée, le contrôleur Struts va copier les valeurs des champs du formulaire posté par le client dans les champs de même nom de **dynaFormulaire**. Reprenons la liste des champs de ce dernier et voyons comment s'opère cette recopie :

Champ	Code HTML associé	Valeur du champ après recopie des valeurs du formulaire
`opt`	<input type="radio" name="opt" value="oui">Oui <input type="radio" name="opt" value="non" checked="checked">Non	- "oui" ou "non" selon le bouton radio coché
`chk1`	<input type="checkbox" name="chk1" value="on">	- "on" si la case **chk1** a été cochée - garde sa précédente valeur si la case **chk1** n'a pas été cochée
`chk2`	<input type="checkbox" name="chk2" value="on">	- "on" si la case **chk2** a été cochée - garde sa précédente valeur si la case **chk2** n'a pas été cochée
`chk3`	<input type="checkbox" name="chk2" value="on">	- "on" si la case **chk3** a été cochée - garde sa précédente valeur si la case **chk3** n'a pas été cochée
`champSaisie`	<input type="text" name="champSaisie" value="">	- valeur tapée par l'utilisateur dans **champSaisie**
`mdp`	<input type="password" name="mdp" value="">	- valeur tapée par l'utilisateur dans **mdp**
`boiteSaisie`	<textarea name="boiteSaisie"></textarea>	- valeur tapée par l'utilisateur dans **boiteSaisie**
`combo`	<select name="combo">...</select>	- valeur sélectionnée par l'utilisateur dans **combo**
`listeSimple`	<select name="listeSimple" size="3">...</select>	- valeur sélectionnée par l'utilisateur dans **listeSimple**
`listeMultiple`	<select name="listeMultiple" multiple="multiple" size="5">	- tableau de chaînes contenant les valeurs

secret	<input type="hidden" name="secret" value="xxx">>	sélectionnées par l'utilisateur dans **listeMultiple** - "xxx".

On a une difficulté avec les champs qui ne reçoivent pas forcément une valeur dans la requête envoyée par le navigateur. C'est le cas des cases à cocher **chk1** à **chk3** et des deux listes **listeSimple** et **listeMultiple**. Dans ce cas, ces champs gardent leur valeur précédente, celles acquises au cours du cycle demande-réponse précédent.

Raisonnons sur la case à cocher **chk1** par exemple et supposons que dans le précédent cycle demande-réponse, l'utilisateur avait coché cette case. Le navigateur avait alors envoyé dans la chaîne de paramètres de sa requête l'information **chk1="on"**. Le constructeur a donc affecté la valeur "**on**" au champ **chk1** de **dynaFormulaire**. Supposons maintenant que dans le cycle en cours, l'utilisateur ne coche pas la case **chk1**. Dans ce cas, dans la chaîne de paramètres de la nouvelle requête, le navigateur n'envoie pas quelque chose comme **chk1="off"** mais n'envoie rien. Du coup, le champ **chk1** de **dynaFormulaire** va garder sa valeur "**on**" et donc avoir une valeur qui ne reflète pas celle du formulaire validé par l'utilisateur. On utilisera la méthode **reset** de **dynaFormulaire** pour résoudre ce problème. On mettra dans cette méthode les trois champs chk1, chk2, chk3 à "off". Dans notre exemple de chk1, soit l'utilisateur :

- coche la case chk1. Alors le navigateur envoie l'information **chk1="on"** et le champ **chk1** de **dynaFormulaire** va passer à "**on**"
- ne coche pas la case **chk1**. Alors le navigateur n'envoie pas de valeur pour le champ **chk1** qui va garder sa précédente valeur "**off**". Dans les deux cas, la valeur enregistrée dans le champ **chk1** de **dynaFormulaire** est correcte.

Le problème est analogue pour les deux listes **listeSimple** et **listeMultiple**. Si aucune option n'a été sélectionnée dans ces listes, elles ne seront pas présentes dans les paramètres de la requête et vont donc conserver leurs valeurs précédentes. Dans la méthode **reset** de **dynaFormulaire**, nous ré-initialiserons **listeSimple** avec une chaîne de caractères vide et **listeMultiple** avec un tableau de chaînes de taille 0.

- une fois la méthode **reset** de **dynaFormulaire** appelée, le contrôleur recopie dans les champs de **dynaFormulaire** les informations qui lui ont été envoyées dans la requête du client
- un objet **ForwardAction** est créé ou recyclé et sa méthode **execute** appelée. **ForwardAction** est une classe prédéfinie qui rend un objet **ActionForward** pointant sur la vue définie par l'attribut "**parameter**" de l'action, ici **/vues/confirmation.jsp**.
- le contrôleur envoie cette vue. Le cycle est terminé.

L'action /affiche

```
<action
  path="/affiche"
  parameter="/vues/formulaire.jsp"
  type="org.apache.struts.actions.ForwardAction"
/>
```

- l'action /**affiche** est déclenchée par activation du lien [Retour vers le formulaire] de la vue n° 2.
- ici il n'y a pas de formulaire associé à l'action. On passe donc tout de suite à l'exécution de la méthode **execute** d'un objet **ForwardAction** qui va rendre un objet **ActionForward** pointant sur la vue **/vues/formulaire.jsp**.

VI.3.6 Le fichier de messages de l'application

La troisième section du fichier **struts-config.xml** est celui du fichier des messages :

```
<message-resources
  parameter="ApplicationResources"
    null="false"
/>
```

Le fichier **ApplicationResources.properties** est placé dans **WEB-INF/classes**. Il sera vide. Même vide, il doit cependant être déclaré dans le fichier de configuration sinon la bibliothèquede balises **struts-bean** que nous verrons un peu plus loin génère une erreur. Cette bibliothèque est utilisée par la vue **confirmation.jsp**.

VI.4Le code des vues

VI.4.1 La vue formulaire.jsp

Rappelons que cette vue est affichée dans deux cas :
- à l'appel de l'action **/init** lors du 1er cycle demande-réponse
- à l'appel de l'action **/affiche** lors des cycles suivants

Le code de la vue **formulaire.jsp** est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
```

```
<html>
  <head>
  <title>formulaire</title>
  </head>
  <body background='<html:rewrite page="/images/standard.jpg"/>'>
  <h3>Formulaire Struts</h3>
    <hr>
    <html:form action="/confirmation" name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
      <table border="0">
        <tr>
          <td>bouton radio</td>
          <td>
            <html:radio name="dynaFormulaire" property="opt" value="oui">Oui</html:radio>
            <html:radio name="dynaFormulaire" property="opt" value="non">Non</html:radio>
          </td>
        </tr>
        <tr>
          <td>Cases à cocher</td>
          <td>
            <html:checkbox name="dynaFormulaire" property="chk1">1</html:checkbox>
            <html:checkbox name="dynaFormulaire" property="chk2">2</html:checkbox>
            <html:checkbox name="dynaFormulaire" property="chk3">3</html:checkbox>
          </td>
        </tr>
        <tr>
          <td>Champ de saisie</td>
          <td>
            <html:text name="dynaFormulaire" property="champSaisie" />
          </td>
        </tr>
        <tr>
          <td>Mot de passe</td>
          <td>
          <html:password name="dynaFormulaire" property="mdp" />
          </td>
        </tr>
        <tr>
          <td>Boîte de saisie multilignes</td>
          <td>
          <html:textarea name="dynaFormulaire" property="boiteSaisie" />
          </td>
        </tr>
        <tr>
          <td>Combo</td>
          <td>
          <html:select name="dynaFormulaire" property="combo">
              <html:options name="dynaFormulaire" property="valeursCombo"/>
            </html:select>
          </td>
        </tr>
        <tr>
          <td>
            <table>
              <tr>
                <td>Liste à sélection unique</td>
              </tr>
              <tr>
                <td>
                  <input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
                </td>
              </tr>
            </table>
          <td>
          <html:select name="dynaFormulaire" property="listeSimple" size="3">
              <html:options name="dynaFormulaire" property="valeursListeSimple"/>
            </html:select>
          </td>
        </tr>
        <tr>
          <td>
            <table>
              <tr>
                <td>Liste à sélection multiple</td>
              </tr>
              <tr>
                <td>
                  <input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
                </td>
              </tr>
            </table>
          </td>
          <td>
          <html:select name="dynaFormulaire" property="listeMultiple" size="5" multiple="true">
              <html:options name="dynaFormulaire" property="valeursListeMultiple"/>
            </html:select>
          </td>
        </tr>
      </table>
      <html:hidden name="dynaFormulaire" property="secret"/>
      <br>
      <hr>
      <html:submit>Envoyer</html:submit>
    </html:form>
```

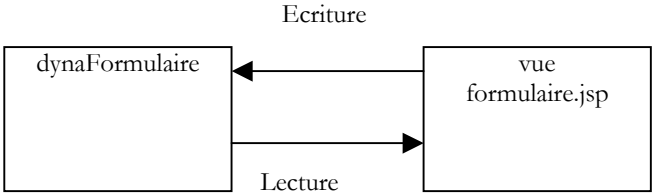
```
  </body>
</html>
```

Cette page JSP utilise des balises provenant de la bibliothèque **struts-html**. Rappelons que pour utiliser une bibliothèque de balises, il faut :

- placer le code de cette bibliothèque quelque part dans l'arborescence de l'application, ici **WEB-INF/struts-html.tld**
- déclarer l'utilisation de cette bibliothèque au début des pages JSP qui l'utilisent :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
```

La vue **formulaire.jsp** utilise des balises que nous allons maintenant expliquer. Pour bien comprendre le fonctonnement de celles-ci, il faut se rappeler que le bean **dynaFormulaire** s'utilise à la fois en **lecture** et **écriture** et qu'il en est de même des balises dela vue **formulaire.jsp** associées aux différents champs de **dynaFormulaire** :



Lorsque le client demande l'url **/init.do** ou **/affiche.do**, la vue **formulaire.jsp** est envoyée. Son contenu provient des champs du bean **dynaFormulaire**. Lorsque le client clique sur le bouton [Envoyer] du formulaire, le navigateur "poste" alors le formulaire HTML à l'action /**confirmation**. Nous avons déjà expliqué ce qui se passait alors et notamment que les champs de **dynaFormulaire** allaient recevoir les valeurs des champs de même nom présents dans la requête HTTP du client. Le comportement en lecture/écriture des différentes balises du formulaire sont expliquées ci-dessous :

- traduction HTML pour l'opération de lecture
- submit pour celle d'écriture

`balise`	<body background="**<html:rewrite page="/images/standard.jpg"/>**">
`traduction HTML`	<body background="/formulaire2/images/standard.jpg">
`explication`	la balise **html:rewrite** permet de s'affranchir du nom de l'application dans des URL. Elle a un attribut : `page` URL à réécrire Ainsi ci-dessus, si on décide d'appeler l'application **formulaire3**, le code de l'attribut background n'a pas à être réécrit. La balise **html:rewrite** génèrera le nouveau code HTML **background="/formulaire3/images/standard.jpg"**

`balise`	**<html:form action="/confirmation" name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">**
`traduction HTML`	<form name="dynaFormulaire" method="post" action="/formulaire2/confirmation.do">"
`explication`	la balise **html:form** permet de générer la balise HTML **form**. Elle a plusieurs attributs : `action` action Struts à laquelle doit être posté le formulaire - doit correspondre à l'une des actions définies dans struts-config.xml `name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devront être placées ou lues les valeurs du formulaire posté. En l'absence de cet attribut, c'est le formulaire associé à l'action définie par l'attribut **action** qui est utilisé. Cette information est trouvée dans le fichier de configuration (attribut **name** de l'action). `type` facultatif - classe Java à instancier pour le formulaire. En l'absence de cet attribut, c'est la classe associée à l'action définie par l'attribut **action** qui est utilisée. Cette information est trouvée dans le fichier de configuration (attribut **type** de l'action). On voit que par défaut, le code HTML généré utilise la méthode **POST**. Dans ce même code HTML, l'URL de l'action a été réécrite pour être préfixée par le nom de l'application et suffixée par .do.

`balise`	**<html:radio name="dynaFormulaire" property="opt" value="oui">Oui</html:radio>** **<html:radio name="dynaFormulaire" property="opt" value="non">Non</html:radio>**
`traduction HTML`	Si le champ **opt** de dynaFormulaire vaut "oui", génère le code HTML <input type="radio" name="opt" value="oui" checked="checked">Oui <input type="radio" name="opt" value="non">Non
`explication`	la balise **html:radio** sert à générer la balise HTML <input type="radio" ...>. Elle admet divers attributs :

	`name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé au bouton radio en lecture et écriture. `value` valeur à donner au champ HTML lorsqu'il sera posté Le texte entre la balise de début et de fin est le texte qui sera affiché à côté du bouton radio.
`submit`	le champ **value** du bouton radio coché est affecté au champ **opt** de **dynaFormulaire**

`balise`	**<html:checkbox name="dynaFormulaire" property="chk1">1</html:checkbox>** **<html:checkbox name="dynaFormulaire" property="chk2">2</html:checkbox>** **<html:checkbox name="dynaFormulaire" property="chk3">3</html:checkbox>**
`traduction HTML`	Si les champs **chk1** et **chk3** de dynaFormulaire valent "on" et si le champ **chk2** vaut "off", génère le code HTML <input type="checkbox" name="chk1" value="on" checked="checked">1 <input type="checkbox" name="chk2" value="on">2 <input type="checkbox" name="chk3" value="on" checked="checked">3
`explication`	la balise **html:checkbox** sert à générer la balise HTML <input type="checkbox" ...>. Elle admet divers attributs : `name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé à la case à cocher en lecture et écriture. `value` valeur à donner au champ HTML lorsqu'il sera posté Le texte entre la balise de début et de fin est le texte qui sera affiché à côté de la case à cocher.
`submit`	Le champ **chk1** de **dynaFormulaire** recevra la valeur du champ value du champ HTML **chk1** si celui-ci est posté, sinon il gardera sa valeur courante. Le champ HTML **chk1** n'est posté que s'il est coché par l'utilisateur.

`balise`	**<html:text name="dynaFormulaire" property="champSaisie" />**
`traduction HTML`	Si le champ **champSaisie** vaut "ceci est un essai", génère le code HTML <input type="text" name="champSaisie" value="ceci est un essai">
`explication`	la balise **html:text** sert à générer la balise HTML <input type="text" ...>. Elle admet divers attributs : `name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé au champ de saisie en lecture et écriture.
`submit`	Le texte saisi par l'utilisateur dans le champ HTML **champSaisie** est affecté au champ **champSaisie** du bean **dynaFormulaire**.

`balise`	**<html:password name="dynaFormulaire" property="mdp" />**
`traduction HTML`	Si le champ **mdp** vaut "azerty", génère le code HTML <input type="password" name="mdp" value="azerty">>
`explication`	la balise **html:password** sert à générer la balise HTML <input type="password" ...>. Elle admet divers attributs : `name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé au champ mot de passe en lecture et écriture.
`submit`	Le texte saisi par l'utilisateur dans le champ HTML **mdp** est affecté au champ mdp du bean **dynaFormulaire**.

`balise`	**<html:textarea name="dynaFormulaire" property="boiteSaisie" />**
`traduction HTML`	Si le champ **boiteSaisie** vaut "ligne1\r\nligne2\r\n", génère le code HTML <textarea name="boiteSaisie">ligne1 ligne2 </textarea>
`explication`	la balise **html:textarea** sert à générer la balise HTML <textarea>...</textarea>. Elle admet divers attributs :

	`name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé au champ de saisie en lecture et écriture.
`submit`	Le texte saisi par l'utilisateur dans le champ HTML **textarea** est affecté au champ textarea du bean **dynaFormulaire**.

`balise`	**<html:select name="dynaFormulaire" property="combo">** **<html:options name="dynaFormulaire" property="valeursCombo"/>** **</html:select>**
`traduction HTML`	Si le champ **combo** vaut "combo2", génère le code HTML <select name="combo"> <option value="combo0">combo0</option> <option value="combo1">combo1</option> <option value="combo2" selected="selected">combo2</option> <option value="combo3">combo3</option> <option value="combo4">combo4</option> </select>
`explication`	la balise **html:select** sert à générer la balise HTML <select>...</select>. Elle admet divers attributs : `name` facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire associé au champ de saisie en lecture et écriture. la balise **html:options** sert à générer les balises HTML <option>...</option> à l'intérieur d'une balise HTML <select>. Il y a diverses façons de préciser comment trouver les valeurs de remplissage du select. Ici, nous avons utilisé les attributs **name** et **property** : `name` facultatif - nom du bean contenant les valeurs des différentes options. En l'absence de cet attribut, c'est le formulaire associé à la balise <**html:form**> qui est utilisé. `property` nom du champ du bean formulaire qui contient les valeurs des options du select. Ce doit être un tableau de chaînes de caractères.
`submit`	Le champ **combo** du bean **dynaFormulaire** reçoit la valeur de l'option sélectionnée par l'utilisateur dans le combo HTML **combo**. Cette valeur dépend de la forme de l'option sélectionnée : <option value="XX">YY</option> : la valeur de l'option est XX <option>YY</option> : la valeur de l'option est YY

`balise`	**<html:select name="dynaFormulaire" property="listeSimple" size="3">** **<html:options name="dynaFormulaire" property="valeursListeSimple"/>** **</html:select>**
`traduction HTML`	Si le champ **listeSimple** vaut "simple1", génère le code HTML <select name="listeSimple" size="3"> <option value="simple0">simple0</option> <option value="simple1" selected="selected">simple1</option> <option value="simple2">simple2</option> ... </select>
`explication`	les mêmes que pour le combo
`submit`	reçoit la valeur de l'option sélectionnée par l'utilisateur dans la liste HTML **listeSimple**. A la différence du combo, un utilisateur peut ne sélectionner aucune option. Dans ce cas, le client n'envoie pas le paramètre **listeSimple** dans sa requête HTTP au serveur. Le champ **listeSimple** de **dynaFormulaire** garde alors sa valeur précédente.

`balise`	**<html:select name="dynaFormulaire" property="listeMultiple" size="5" multiple="true">** **<html:options name="dynaFormulaire" property="valeursListeMultiple"/>** **</html:select>**
`traduction HTML`	Si le champ **listeMultiple** est le tableau {"multiple0","multiple2"}, génère le code HTML <select name="listeMultiple" multiple="multiple" size="5"> <option value="multiple0" selected="selected">multiple0</option> <option value="multiple1">multiple1</option> <option value="multiple2" selected="selected">multiple2</option> <option value="multiple3">multiple3</option> ... </select>

explication	les mêmes que pour le combo
submit	reçoit sous forme d'un tableau de chaînes de caractères, les valeurs des options sélectionnées par l'utilisateur dans la liste HTML **listeMultiple**. Un utilisateur peut ne sélectionner aucune option. Dans ce cas, le client n'envoie pas le paramètre **listeMultiple** dans sa requête HTTP au serveur. Le champ **listeMultiple** de **dynaFormulaire** garde alors sa valeur précédente.

balise	**<html:hidden name="dynaFormulaire" property="secret"/>**
traduction HTML	Si le champ **secret** de **dynaFormulaire** a la valeur "xxx", génère le code HTML <input type="hidden" name="secret" value="xxx">
explication	la balise **html:hidden** sert à générer la balise HTML <input type="hidden" ...>. name : facultatif - nom du bean **ActionForm** ou dérivé dans lequel devra être placée ou lue la valeur du champ. En l'absence de cet attribut, c'est le formulaire associé à la balise **<html:form>** qui est utilisé. property : nom du champ du bean formulaire associé au champ caché. La valeur "xxx" attribuée au champ HTML **secret** vient de la définition du formulaire dans le fichier de configuration. On y a défini une valeur initiale "xxx" pour le champ **secret**.
submit	reçoit la valeur du champ HTML secret. Comme l'utilisateur n'a pas la possibilité de changer cette valeur, le champ **secret** de **dynaFormulaire** recevra ici la valeur **xxx**.

balise	**<html:submit>Envoyer</html:submit>**
traduction HTML	<input type="submit" value="Envoyer">.

La dernière chose à expliquer est le code javascript inclus dans la page JSP et lié aux deux boutons [Effacer] qui désélectionnent les éléments sélectionnés dans les listes **listeSimple** et **listeMultiple** :

```
<input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
<input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
```

Pour comprendre ce code, rappelons comment sont désignés les différents éléments d'un document web au sein d'un code Javascript l'exploitant :

Donnée	Signification
document	désigne la totalité du document web
document.forms	désigne la collection des formulaires définis dans le document
document.forms[i]	désigne le formulaire n° i du document
document.forms["nomFormulaire"]	désigne le formulaire <form> ayant l'attribut **name** égal à **"nomFormulaire"**
document.nomFormulaire	désigne le formulaire <form> ayant l'attribut **name** égal à **"nomFormulaire"**
document.[formulaire].elements	désigne la collection des éléments membres du formulaire désigné par l'expression [formulaire]. Cette collection comprend toutes les balises <input>, <textarea>, <select> du formulaire désigné.
document.[formulaire].elements[i]	désigne l'élément n° i de [formulaire]
document.[formulaire].elements["nomComposant"]	désigne l'élément de [formulaire] ayant l'attribut **name** égal à **nomComposant**
document.[formulaire]. nomComposant	désigne l'élément de [formulaire] ayant l'attribut **name** égal à **nomComposant**
document.[formulaire].[composant].value	désigne la valeur du composant [composant] du formulaire [formulaire] lorsque le code HTML de celui-ci peut avoir un attribut value (<input>, <textarea>)
document.[formulaire].[select].selectedIndex	désigne l'indice de l'option sélectionnée dans une liste. S'utilise en lecture et écriture. Mettre cette propriété à -1 désélectionne tous les éléments de la liste.
document.[formulaire].[select].options	désigne le tableau des options associées à une balise **<select>**
document.[formulaire].[select].options[i]	désigne l'option n° i de la balise <select> indiquée
document.[formulaire].[select].options[i].selected	booléen indiquant si l'option n° i de la balise [select] indiquée est sélectionnée (true) ou non. Peut s'utiliser en lecture et écriture

Revenons sur le code javascript des deux boutons :

```
<input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
<input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
```

Sur l'événement clic sur le bouton, le code associé à l'attribut "**onclick**" est exécuté. Ici, c'est un code en ligne. Le plus souvent, on écrit **onclick="fonction(...)"** où **fonction** est une fonction définie au sein d'une balise **<script language="javascript"> ... </script>**. Que fait le code ci-dessus ? Commentons le code du premier bouton :

`this`	désigne le document web dans lequel se trouve le bouton
`this.form`	désigne le formulaire dans lequel se trouve le bouton
`this.form.listeSimple`	désigne le composant **listeSimple** du formulaire
`this.form.listeSimple.selectedIndex`	désigne l'indice de l'option sélectionnée dans **listeSimple**. Mettre à -1 cette propriété entraîne la désélection de toute option.

VI.4.2 La vue confirmation.jsp

Rappelons que cette vue est affichée à l'issue de l'action **/confirmation**, c.a.d. après que le formulaire contenu dans la vue **formulaire.jsp** ait été posté par le client web. Elle a pour seul but d'afficher les valeurs saisies par l'utilisateur. Son code est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html>
  <head>
    <title>Confirmation</title>
  </head>
  <body background="<html:rewrite page="/images/standard.jpg"/>">
    <h3>Confirmation des valeurs saisies</h3>
     <hr/>
     <table border="1">
      <tr>
         <td>Bouton radio</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="opt"/></td>
       </tr>
      <tr>
         <td>Case à cocher chk1</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="chk1"/></td>
       </tr>
      <tr>
         <td>Case à cocher chk2</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="chk2"/></td>
       </tr>
      <tr>
         <td>Case à cocher chk3</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="chk3"/></td>
       </tr>

      <tr>
         <td>Champ de saisie</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="champSaisie"/></td>
       </tr>
      <tr>
         <td>Mot de passe</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="mdp"/></td>
       </tr>

      <tr>
         <td>Boîte de saisie</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="boiteSaisie"/></td>
       </tr>

      <tr>
         <td>combo</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="combo"/></td>
       </tr>
      <tr>
         <td>liste simple</td>
         <td><bean:write name="dynaFormulaire" scope="session" property="listeSimple"/></td>
       </tr>
      <logic:iterate id="choix" indexId="index" name="dynaFormulaire" property="listeMultiple">
        <tr>
           <td>liste multiple[<bean:write name="index"/>]</td>
           <td><bean:write name="choix"/></td>
         </tr>
      </logic:iterate>
     </table>
    <br>
     <html:link page="/affiche.do">
       Retour au formulaire
```

```
    </html:link>
  </body>
</html>
```

Nous introduisons ici deux nouvelles bibliothèques de balises : **struts-bean** et **struts-logic**. La bibliothèque **struts-bean** permet d'avoir accès à des objets placés dans la requête, la session ou le contexte de l'application. La bibliothèque **struts-logic** permet d'introduire de la logique d'exécution au moyen de balises. Ces deux bibliothèques ne sont nullement indispensables. Comme nous l'avons vu, une page JSP peut :

- récupérer des objets dans la requête (request.getAttribute(...)), la session (session.getAttribute(...), ou le contexte de l'application
- inclure des parties dynamiques dans le code HTML au moyen de variables <%= variable %>
- comporter du code Java <% code Java %>

L'inclusion de code Java dans des pages JSP gêne tous ceux qui voudraient une stricte séparation entre logique applicative (code java) et présentation (utilisation de balises). Aussi a-t-on inventé pour eux des bibliothèques de balises.

Nous allons procéder comme il a été fait pour la vue **formulaire.jsp** et expliquer chacune des balises présentes dans le code de **confirmation.jsp** si elles n'ont pas été déjà rencontrées dans la vue **formulaire.jsp**. Tout d'abord, remarquons que la page commence par déclarer les trois bibliothèques de balises qu'elle va utiliser :

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
```

Nous commentons maintenant les balises du document **formulaire.jsp** :

balise
```
<bean:write name="dynaFormulaire" scope="session" property="opt"/>
```
fonction
écrit une valeur dans le flux HTML courant. La balise **bean:write** admet les attributs suivants :

name : nom de l'objet à utiliser
scope : portée (request, session, context) dans laquelle chercher cet objet
property : champ de l'objet désigné par **name** dont il faut écrire la propriété. Ce champ peut être un objet de tout type. Ce sera la méthode **toString** de l'objet qui sera utilisée pour afficher celui-ci.

Ici, la valeur du champ **opt** de **dynaFormulaire** est écrite. On aura soit "oui" si l'utilisateur a coché le bouton radio ayant l'attribut value="oui", soit "non" s'il a coché le bouton radio ayant l'attribut value="non"

balise
```
<bean:write name="dynaFormulaire" scope="session" property="chk1"/>
```
fonction
écrit la valeur du champ **chk1** de **dynaFormulaire**. On aura soit "on" si l'utilisateur a coché la case, soit "off" sinon. Idem pour **chk2**, **chk3**.

balise
```
<bean:write name="dynaFormulaire" scope="session" property="champSaisie"/>
```
fonction
écrit la valeur du champ **champSaisie** de **dynaFormulaire**, c'est à dire le texte tapé par l'utilisateur dans ce champ. Idem pour **mdp**, **boiteSaisie.**

balise
```
<bean:write name="dynaFormulaire" scope="session" property="combo"/>
```
fonction
écrit la valeur du champ **combo** de **dynaFormulaire**. On aura l'attribut **value** de l'élément <**option**> sélectionné par l'utilisateur.

balise
```
<bean:write name="dynaFormulaire" scope="session" property="listeSimple"/>
```
fonction
écrit la valeur du champ **listeSimple** de **dynaFormulaire**. On aura l'attribut **value** de l'élément <**option**> sélectionné par l'utilisateur s'il y en a eu un. Sinon on aura la chaîne vide.

balise
```
     <logic:iterate id="choix" indexId="index" name="dynaFormulaire" property="listeMultiple">
       <tr>
          <td>liste multiple[<bean:write name="index"/>]</td>
          <td><bean:write name="choix"/></td>
        </tr>
     </logic:iterate>
```
fonction
Ici, nous introduisons des balises de logique. Nous avons affaire à une liste à choix multiple. La valeur du champ **listeMultiple** de l'objet **dynaFormulaire** est un tableau de **String**. En java, nous écririons une boucle. La balise **logic:iterate** nous permet de faire cette même boucle sans écrire de code Java. La balise **logic:iterate** a dans l'exemple les attributs suivants :

name="dynaFormulaire" : nom de l'objet à utiliser
property="listeMultiple" : nom de la propriété qui dans l'objet indiqué par **name** contient la collection qui va être parcourue dans la boucle. Ici cette collection est le tableau des valeurs sélectionnées dans **listeMultiple**. Ce tableau peut être vide.

id="choix" : identificateur désignant l'élément courant du tableau à chaque tour de boucle. Lors de la 1ère itération, **choix** représentera **listeMultiple[0]**, à la seconde **listeMultiple[1]** et ainsi de suite.
indexID="index" : identificateur désignant l'indice de l'élément courant du tableau à chaque tour de boucle. Lors de la 1ère itération, **index** aura la valeur 0, à la seconde la valeur 1 et ainsi de suite.

Le code HTML contenu entre les balises **<logic:iterate ...>** et **</logic:iterate>** est répété pour chaque élément de la collection désignée par le couple (name,property). La partie dynamique de ce code est le suivant :

```
          <td>liste multiple[<bean:write name="index"/>]</td>
          <td><bean:write name="choix"/></td>
```

De ce qui a été dit précédemment, on déduit qu'à l'itération n° i (i>=0), le code HTML généré est équivalent au code suivant :

```
          <td>liste multiple[<%=i%>]</td>
          <td><%=listeMultiple[i]%></td>
```

balise

```
    <html:link page="/affiche.do">
      Retour au formulaire
    </html:link>
```

fonction génère un lien relatif au contexte de l'application, ce qui évite de connaître celui-ci. Le code HTML généré par cette balise est le suivant :

```
<a href="/formulaire2/affiche.do">Retour au formulaire</a>
```

VI.5 Les classes Java

Le fichier de configuration **struts-config.xml** référence deux classes Java :

```
    <form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
...
    <action
      path="/init"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      type="istia.st.struts.formulaire.InitFormulaireAction"
    >
```

La classe **DynaFormulaire** est la classe qui contiendra les valeurs de la vue n° 1 **formulaire.jsp**. La classe **InitFormulaireAction** est la classe chargée d'initialiser le bean **dynaFormulaire** avant son premier affichage.

VI.5.1 La classe DynaFormulaire

Pour contenir les valeurs d'un formulaire, un objet de type **DynaActionForm** est suffisant sauf si on a besoin de redéfinir l'une des méthodes **reset** ou **validate** de cette classe. Ici, la méthode **validate** n'a pas besoin d'être redéfinie puisqu'on ne fait aucune validation de données. Mais la méthode **reset** a elle besoin d'être redéfinie. En effet, les champs de l'objet **DynaFormulaire** vont recevoir leurs valeurs du formulaire posté par le client web. Or certains champs peuvent ne pas recevoir de valeur s'ils ne sont pas présents dans la requête. Cela se produit dans les cas suivants :

- une case à cocher qui n'a pas été cochée par l'utilisateur
- une liste de taille supérieure à 1 ou aucune option n'a été sélectionnée

Pour des formulaires ayant ce type de composants, la méthode **reset** doit

- affecter la valeur "off" au champ associé à la case à cocher
- affecter la chaîne vide au champ associé à une liste à sélection unique
- affecter un tableau de taille nulle de chaînes de caractères au champ associé à une liste à sélection multiple

Ainsi, si ces champs ne reçoivent pas de valeur de la requête, ils conservent la valeur attribuée par **reset**, valeur qui correspond à l'état du composant dans le formulaire validé par l'utilisateur (case non cochée, liste avec aucun élément sélectionné).

Le code de la classe **DynaFormulaire**, classe dérivée de **DynaActionForm** est le suivant :

```
package istia.st.struts.formulaire;

import org.apache.struts.action.DynaActionForm;
import org.apache.struts.action.ActionMapping;
```

```
import javax.servlet.http.HttpServletRequest;

public class DynaFormulaire extends DynaActionForm {
  public void reset(ActionMapping mapping, HttpServletRequest request){
    // reset des cases à cocher - valeur off
    set("chk1","off");
    set("chk2","off");
    set("chk3","off");
    // reset listeSimple - chaîne vide
    set("listeSimple","");
    // reset listeMultiple - tableau vide
    set("listeMultiple",new String[]{});
  }
}
```

VI.5.2 La classe InitFormulaireAction

La classe **InitFormulaireAction** est, dans le fichier **struts-config.xml**, associée à l'action /**init** :

```
    <action
      path="/init"
      name="dynaFormulaire"
      validate="false"
      scope="session"
      type="istia.st.struts.formulaire.InitFormulaireAction"
    >
      <forward name="afficherFormulaire" path="/vues/formulaire.jsp"/>
    </action>
```

L'action /**init** n'est utilisée qu'une fois lors de la construction initiale de l'objet **DynaFormulaire**. Elle a pour but de fournir du contenu aux trois listes du formulaire **combo**, **listeSimple**, **listeMultiple**. Ce contenu est fourni sous forme de trois tableaux, propriétés de l'objet **dynaFormulaire** :

```
    <form-bean name="dynaFormulaire" type="istia.st.struts.formulaire.DynaFormulaire">
      <form-property name="opt" type="java.lang.String" initial="non"/>
...
      <form-property name="valeursCombo" type="java.lang.String[]" />
      <form-property name="valeursListeSimple" type="java.lang.String[]" />
      <form-property name="valeursListeMultiple" type="java.lang.String[]"/>
    </form-bean>
```

Une fois que les tableaux **valeursCombo**, **valeursListeSimple**, **valeursListeMultiple** auront été initialisés par **InitFormulaireAction**, ils n'auront plus besoin de l'être. En effet, l'objet **dynaFormulaire** est placé dans la session et conserve donc sa valeur au fil des cycles demande-réponse. C'est pourquoi l'action /**init** n'est exécutée qu'une fois. Le code de **InitFormulaireAction** est le suivant :

```
package istia.st.struts.formulaire;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;

public class InitFormulaireAction
  extends Action {
  public ActionForward execute(ActionMapping mapping, ActionForm form,
       HttpServletRequest request, HttpServletResponse response) throws IOException,    ServletException {

    // prépare le formulaire à afficher

    // on met les informations nécessaires au formulaire dans son bean
    DynaFormulaire formulaire = (DynaFormulaire) form;
    formulaire.set("valeursCombo", getValeurs(5, "combo"));
    formulaire.set("valeursListeSimple", getValeurs(7, "simple"));
    formulaire.set("valeursListeMultiple", getValeurs(10, "multiple"));
    // on rend la main
    return mapping.findForward("afficherFormulaire");
  } //execute

  // liste des valeurs du combo
  private String[] getValeurs(int taille, String label) {
    String[] valeurs = new String[taille];
    for (int i = 0; i < taille; i++) {
      valeurs[i] = label + i;
    }
    return valeurs;
  }
```

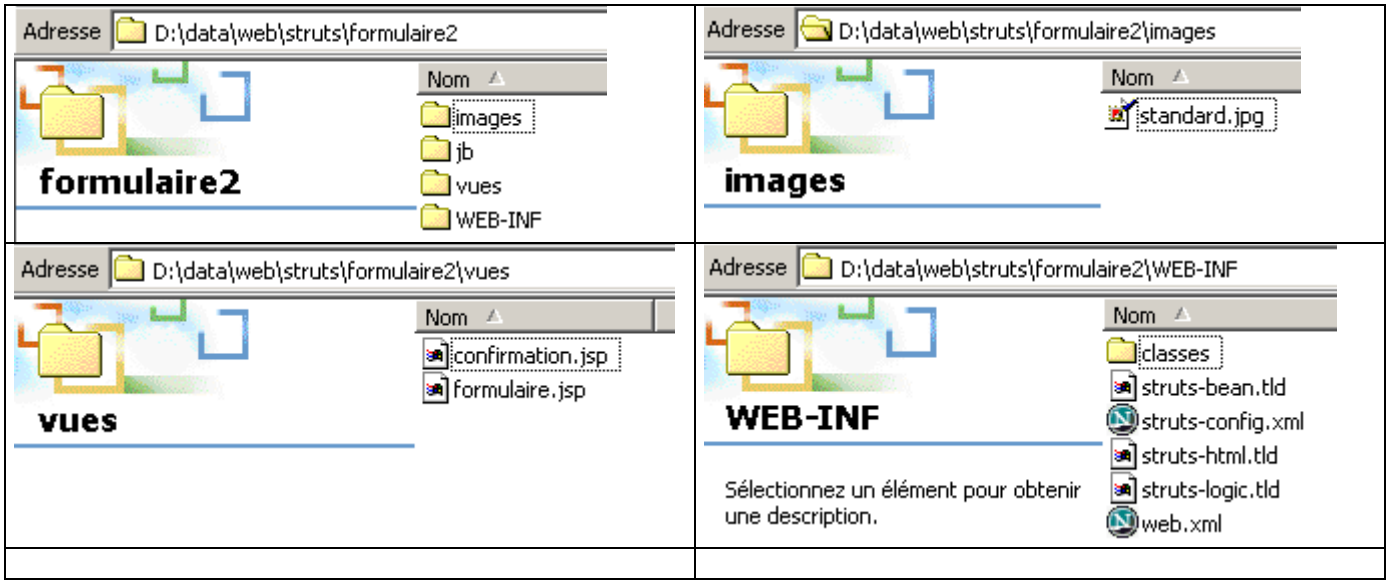
```
}
```

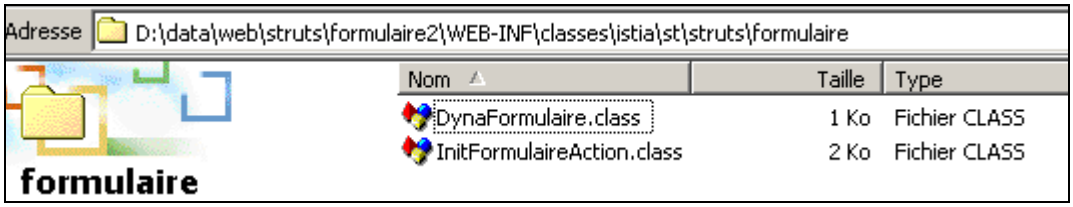
- la classe dérive de la classe **Action**. C'est obligatoire.
- le contrôleur Struts utilise un objet **Action** via sa méthode **execute**. C'est donc elle qu'il faut redéfinir. Cette méthode reçoit les pramètres suivants :
    - **ActionMapping mapping** : un objet représentant la configuration de l'aplication dans struts-config.xml
    - **ActionForm form** : le formulaire associé à l'action s'il y en a un défini dans la configuration de l'action (attribut **name** de l'action).
    - **HttpServletRequest request** : la requête du client
    - **HttpServletResponse** : la réponse au client

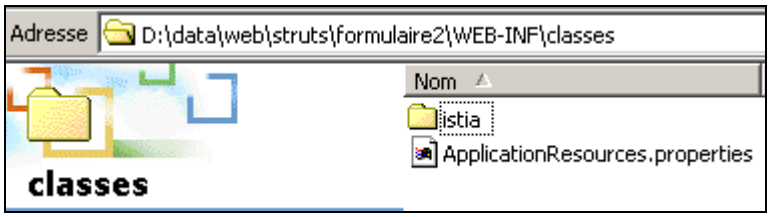
- la classe **InitFormulaireAction** doit initialiser le formulaire **dynaFormulaire**. Celui arrive à la méthode **execute** sous la forme du paramètre **ActionForm form**. Rappelons que **dynaFormulaire** est de type **DynaFormulaire,** classe dérivée de la classe **DynaActionForm** elle-même dérivée de la classe **ActionForm**.
- dans la méthode **execute**, des valeurs sont affectées aux trois champs **valeursCombo**, **valeursListeSimple**, **valeursListeMultiple** grâce à la méthode **set** de la classe **DynaActionForm**. Ces valeurs sont des tableaux arbitraires par souci de simplicité. Remarquons que la méthode **set** affecte une valeur à un champ existant. Elle ne peut pas servir à créer de nouveaux champs. C'est pourquoi, il est nécessaire de définir les trois champs **valeursCombo**, **valeursListeSimple**, **valeursListeMultiple** dans la définition de l'objet **dynaFormulaire** dans **struts-config.xml**.
- la méthode **execute** se termine en renvoyant au contrôleur la clé de la vue à afficher comme réponse au client. Ici, c'est la clé **afficherFormulaire** qui dans le fichier **struts-config.xml** a été associée à la vue **/vues/formulaire.jsp**.

VI.6 Déploiement

L'arborescence de l'application est la suivante :



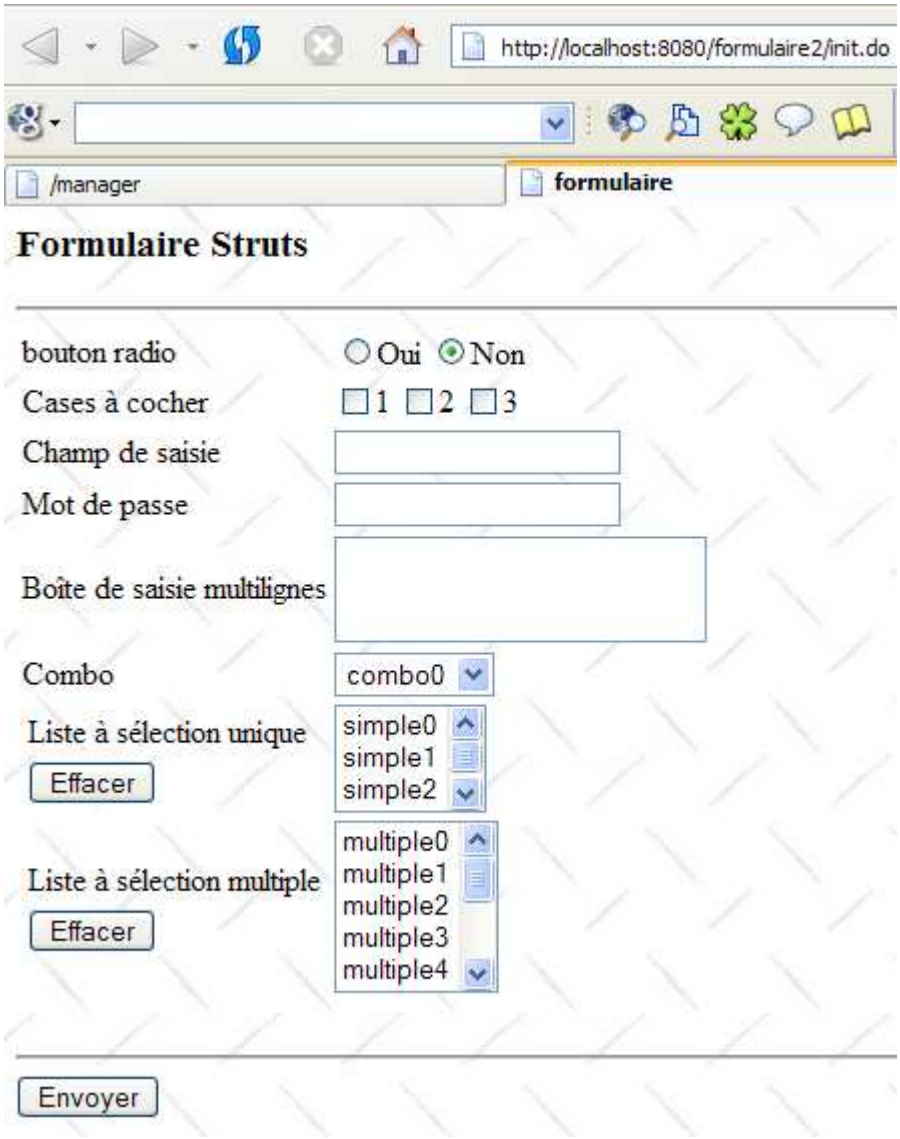




Rappelons que le fichier **ApplicationResources.properties** ci-dessus est nécessaire à la bibliothèque de balises **struts-bean**. On sait que ce fichier contient les messages de l'application. Ceux-ci sont accessibles à la bibliothèque **struts-bean**. Ici, notre application ne définit aucun message. Aussi le fichier **ApplicationResources.properties** existe mais est vide.

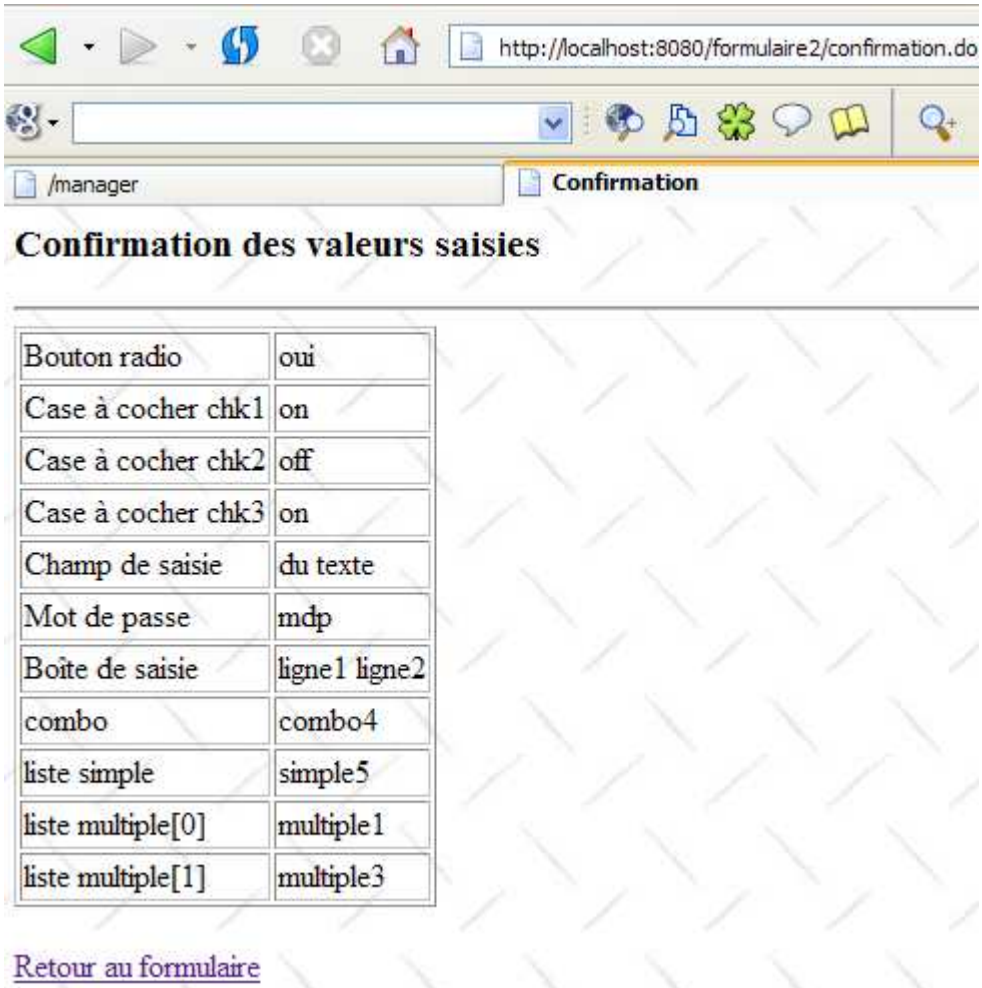
VI.7 Tests

Une fois l'application /formulaire2 correctement déployée, nous la démarrons (Tomcat manager) et demandons l'url http://localhost:8080/formulaire2/init.do :



Nous remplissons le formulaire :

Après utilisation du bouton [Envoyer], nous obtenons le résultat suivant :



Si nous utilisons le lien [Retour au formulaire] ci-dessus, nous retrouvons le formulaire tel qu'il a été saisi :

VI.8 Conclusion

Nous avons détaillé dans cette partie la façon de gérer les différents composants d'un formulaire HTML. Nous pouvons maintenant utiliser des formulaires complexes dans nos applications Struts.

VII Application QuiEst

Nous décrivons ici une application Struts un peu plus sophistiquée que celles qui ont précédé et qui se devaient d'être simples pour être pédagogiques.

VII.1 La classe users

On dispose d'une classe Java mémorisant les informations sur les utilisateurs d'une machine Unix. Celles-ci sont enregistrées dans trois fichiers particuliers :

- **/etc/passwd** : liste des utilisateurs
- **/etc/group** : liste des groupes
- **/etc/aliases** : liste des alias pour le courrier

Le contenu de ces trois fichiers est le suivant :

- /etc/passwd

Les lignes de ce fichier ont la forme suivante :
login:pwd:uid:gid:id:dir:shell
avec

`login`	login de l'utilisateur
`pwd`	son mot de passe crypté
`uid`	son n° d'utilisateur
`gid`	son n° de groupe
`id`	son identité
`dir`	son répertoire de connexion
`shell`	son interpréteur de commandes

Ainsi la ligne d'un utilisateur pourrait être la suivante :
dupond:xg675SDFEkl09:110:57:Guillaume Dupond:/home/iup2-auto/dupond:/bin/bash
L'utilisateur précédent porte le n° 110 et appartient au groupe 57. C'est dans le fichier */etc/group* qu'on trouvera la définition du groupe 57.

- /etc/group

Les lignes de ce fichier ont la forme suivante :
nomGroupe:pwd:gid:membre1,membre2,....
avec

`nomGroupe`	nom du groupe
`pwd`	son mot de passe crypté - le plus souvent ce champ est vide
`gid`	le n° du groupe
`membrei`	logins d'utilisateurs - ce champ peut être vide

Ainsi la ligne du groupe 57 précédent pourrait être la suivante :
iup2-auto::57:
qui indique que le groupe 57 porte le nom *iup2-auto*.

- /etc/aliases

Les lignes de ce fichier ont la forme suivante :
alias:[tab]login
avec

`alias`	alias
`[tab]`	une ou plusieurs tabulations
`login`	login de l'utilisateur auquel appartient l'alias

Ainsi la ligne
guillaume.dupond: dupond

signifie que l'alias *guillaume.dupond* appartient à l'utilisateur de login *dupond*. Rappelons que les alias sont utilisés dans les adresses électroniques. Ainsi si dans l'exemple précédent, la machine Unix s'appelle *shiva.istia.univ-angers.fr*, un courrier adressé à *guillaume.dupond@shiva.istia.univ-angers.fr* sera délivré dans la boîte à lettres de l'utilisateur de login *dupond* de cette machine.

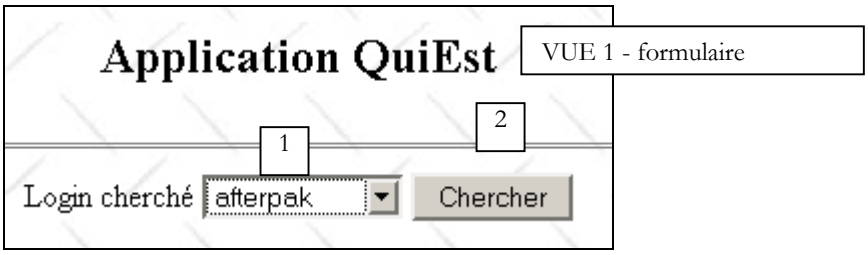
Nous ne nous intéresserons pas ici à la totalité de l'interface de la classe **users** mais simplement à son constructeur et quelques méthodes :

```
import java.io.*;
import java.util.*;

public class users{


  // attributs
  private Hashtable usersByLogin=new Hashtable();    // login --> login, pwd, ..., dir
  private ArrayList erreurs=new ArrayList();         // liste de messages d'erreur

....

  // constructeur
  public users(String usersFileName, String groupsFileName, String aliasesFileName) throws Exception {
     // usersFileName : nom du fichier des utilisateurs avec des lignes de la forme
     // login:pwd:uid:gid:id:dir:shell
     // groupsFileName : nom du fichier des groupes avec des lignes de la forme
     // nom:pwd:numéro:membre1,membre2,..
     // aliasesFileName : nom du fichier des alias avec des lignes de la forme
     // alias:[tab]login
    // construit le dictionnaire usersByLogin
....
  }// constructeur

  // liste des utilisateurs
  public Hashtable getUsersByLogin(){
    return usersByLogin;
  }

  // erreurs
  public ArrayList getErreurs(){
    return erreurs;
  }
```

`usersByLogin`	dictionnaire (Hashtable) dont les clés sont les logins du fichier **passwd**. La valeur associée à la clé est un tableau de chaînes (String [7]) dont les éléments sont les 7 champs de la ligne du fichier **passwd** associée au login. Certains champs peuvent être vides si la ligne a moins de 7 champs.
`erreurs`	liste de messages d'erreurs - vide si pas d'erreurs

VII.2L'application web quiest

S'appuyant sur la classe précédente, on se propose de construire l'application web suivante (page formulaire) :



n°	nom	type HTML	rôle
1	cmbLogins	<select ...>...</select>	présente la liste de tous les logins à propos desquels on peut demander de l'information
2	btnChercher	<input type="submit" ...>	pour lancer la recherche

Lorsque l'utilisateur appuie sur le bouton [Chercher] (2), le login de (1) est demandé à un objet U de type **users**. Si le login existe, on a la réponse suivante (page infos) :

http://localhost:8080/strutsquiest2/infosLogin.do?cmbLogins=afterpak

Mail Home Radio Netscape Search Bookmarks

Application QuiEst - login[afterpak]

Application QuiEst - login[afterpak]

VUE 2 - informations

login	pwd	uid	gid	id	dir	shell
afterpak	4bls0OLOZCqII	1070	156	afterpak	/home/./projets-0102/afterpak	/bin/false

Retour au formulaire

Comme l'URL du navigateur le montre ci-dessus, les paramètres du formulaire sont envoyés au serveur par un GET. On peut donc donner directement au navigateur cette URL paramétrée. C'est ce que nous faisons ici, pour introduire un login qui n'existe pas. On a la réponse suivante (page erreurs) :

http://localhost:8080/strutsquiest2/infosLogin.do?cmbLogins=xx

Mail Home Radio Netscape Search Bookmarks

Application QuiEst - erreurs

VUE 3- erreurs

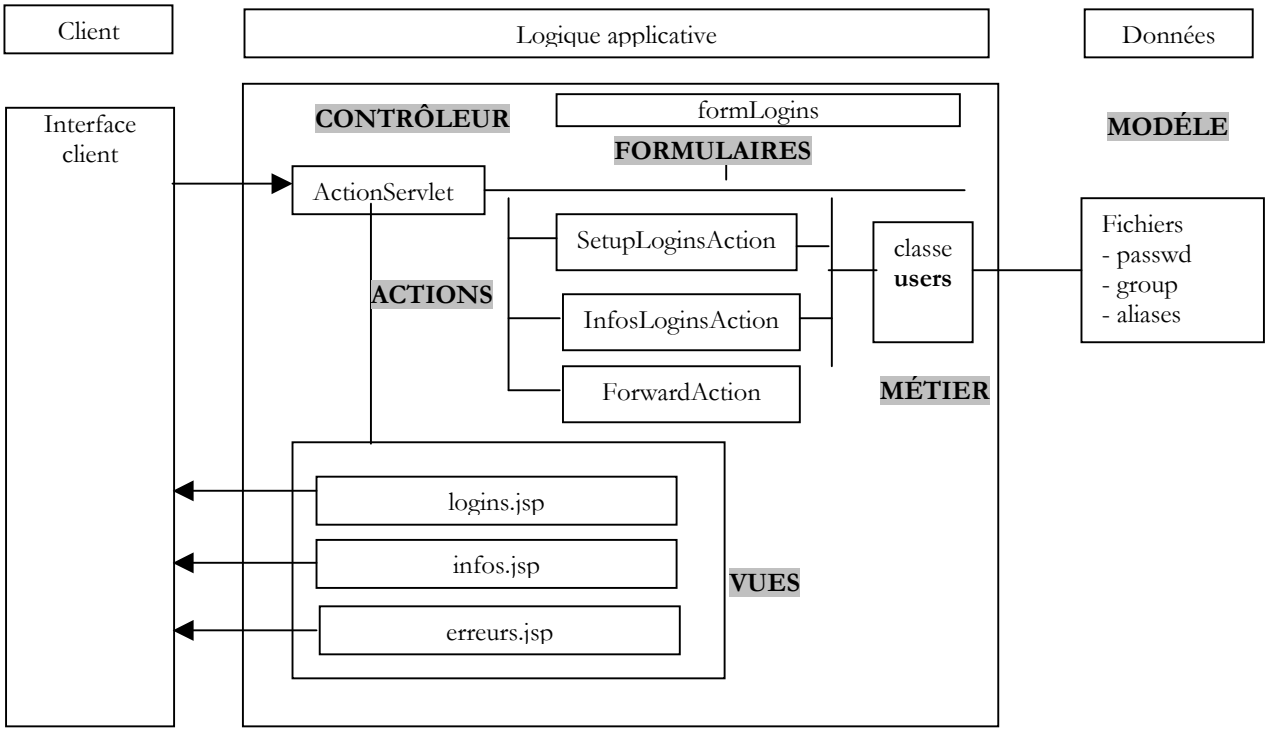
Application QuiEst - Erreurs

Les erreurs suivantes se sont produites

- Le login [xx] n'existe pas

Retour au formulaire

VII.3L'architecture de l'application



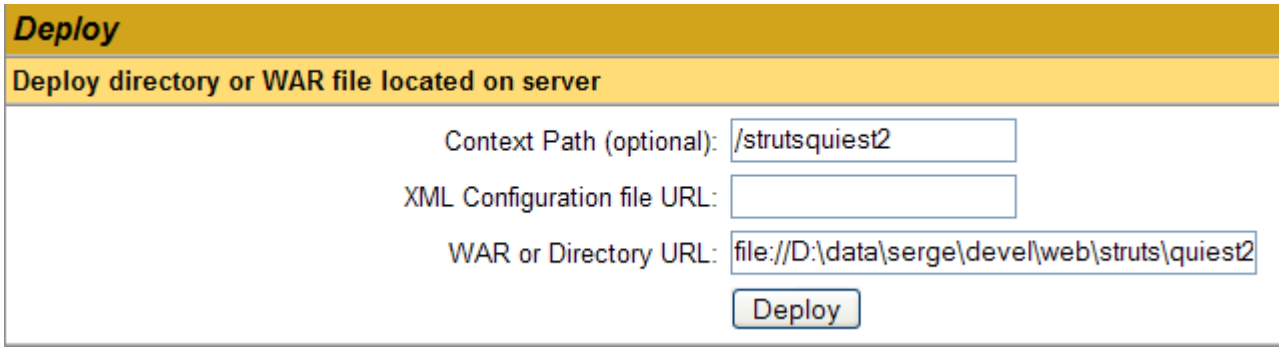
Nous trouvons dans cette architecture les composants suivants :

- les vues :
  - **logins.jsp**, utilisée pour afficher la liste des logins (vue 1)
  - **infos.jsp** utilisée pour afficher les informations liées à un login (vue 2)
  - **erreurs.jsp** utilisée pour afficher une liste d'erreurs (vue 3)
- les formulaires de type **ActionForm** utilisés par les actions :
  - **formLogins** utilisé pour collecter les données du formulaire **logins.jsp**
- les actions :
  - **SetupLoginAction** qui prépare le contenu de **formulaire.jsp** puis affiche cette vue
  - **InfosLoginAction** qui traite le contenu de **logins.jsp** une fois celui-ci envoyé au serveur
  - **ForwardAction** qui traite le lien [Retour vers le formulaire] des vues **infos.jsp** et **erreurs.jsp**
- la classe métier **users** utilisée par les actions pour obtenir leurs données
- le modèle assuré par les trois fichiers plats **passwd**, **group** et **aliases**

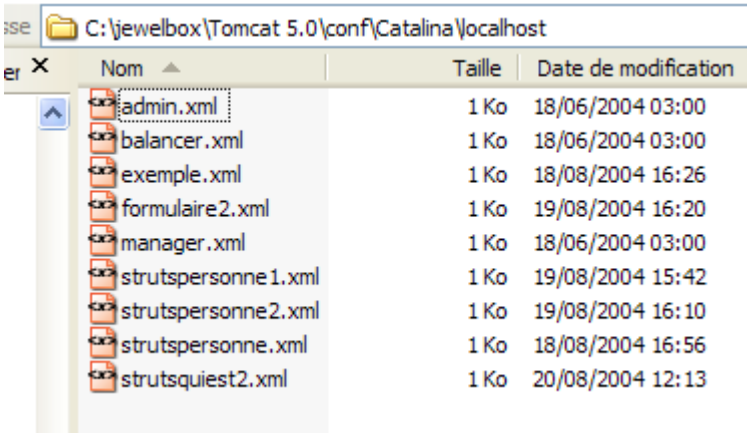
VII.4Configuration de l'application web

VII.4.1 Création du contexte

Nous appellerons **/strutsquiest2** cette nouvelle application. Nous la déployons dans Tomcat de la façon suivante (page Tomcat Manager) :



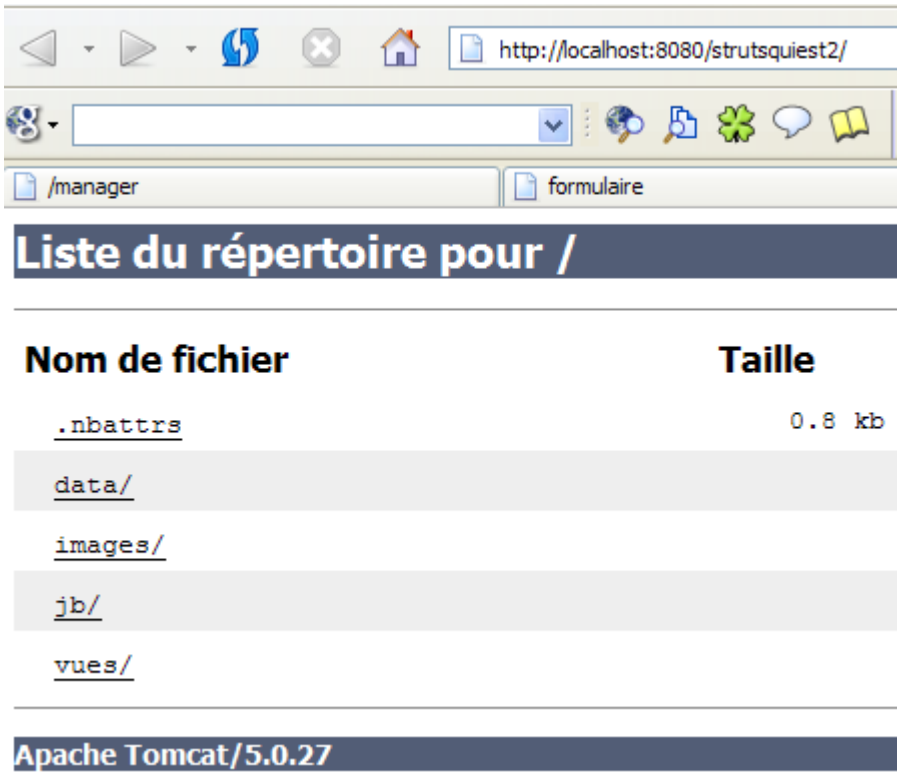
Ce déploiement donne naissance au fichier **strutsquiest2.xml** dans <tomcat>\conf\Catalina\localhost :



Le contenu de ce fichier est le suivant :

```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="D:/data/serge/devel/web/struts/quiest2" path="/strutsquiest2">
</Context>
```

On peut vérifier la validité du contexte en demandant l'URL **http://localhost:8080/strutsquiest2/**



VII.4.2 Le fichier web.xml

Le fichier de configuration **web.xml** de l'application sera le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <servlet>
        <servlet-name>strutsquiest2</servlet-name>
        <servlet-class>istia.st.struts.quiest.Quiest2ActionServlet</servlet-class>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>passwdFileName</param-name>
            <param-value>data/passwd</param-value>
        </init-param>
```

```
        <init-param>
            <param-name>groupFileName</param-name>
            <param-value>data/group</param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>strutsquiest2</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

Ce fichier **web.xml** apporte une nouveauté. Le contrôleur Struts n'est plus **org.apache.struts.action.ActionServlet** mais une classe dérivée qu'on a appelée ici **istia.st.struts.quiest.Quiest2ActionServlet**. Cela va nous permettre de récupérer les deux paramètres d'initialisation que sont **passwdFileName** (localisation du fichier **passwd**) et **groupFileName** (localisation du fichier **group**). Le fichier **aliases** est inutile dans cette application.

VII.4.3 Le fichier struts-config.xml

Le fichier **struts-config.xml** sera le suivant :

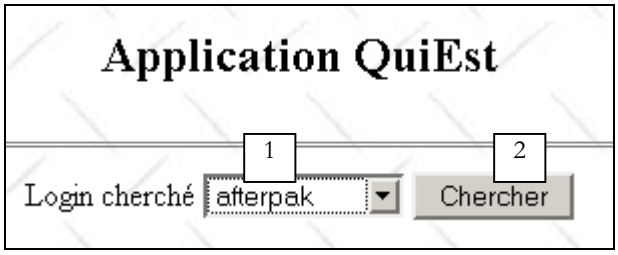
```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="formLogins" type="org.apache.struts.action.DynaActionForm">
      <form-property name="cmbLogins" type="java.lang.String" initial=""/>
    </form-bean>
  </form-beans>

  <action-mappings>

    <action
      path="/init"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.SetupLoginsAction"
    >
      <forward name="afficherLogins" path="/vues/logins.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    </action>

    <action
      path="/infosLogin"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.InfosLoginAction"
    >
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
      <forward name="afficherInfos" path="/vues/infos.jsp"/>
    </action>

    <action
      path="/retourLogins"
      parameter="/vues/logins.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />

  </action-mappings>

    <message-resources
      parameter="istia.st.struts.quiest.ApplicationResources"
     null="false"
   />

</struts-config>
```

On y retrouve les trois grandes sections :
- la déclaration des **formulaires** dans la section <**form-beans**>
- la déclaration des **actions** dans la section <**action-mappings**>
- la déclaration du **fichier de ressources** dans <**message-ressources**>

VII.4.4 Les objets (beans) formulaires de l'application

```
<form-beans>
  <form-bean name="formLogins" type="org.apache.struts.action.DynaActionForm">
    <form-property name="cmbLogins" type="java.lang.String" initial=""/>
  </form-bean>
</form-beans>
```

Il n'y a qu'un bean formulaire dans notre application., appelé **formLogins** et de type dérivé de **DynaActionForm**. Il sera utilisé pour gérer le combo du formulaire de la vue n°1 lorsque l'utilisateur va valider celui-ci. Etudions le :



n°	nom	type HTML	rôle
1	cmbLogins	<select ...>...</select>	présente la liste de tous les logins à propos desquels on peut demander de l'information
2	btnChercher	<input type="submit" ...>	pour lancer la recherche

Distinguons plusieurs cas :

- du client vers le serveur, l'objet **formLogins** est utilisé pour contenir les valeurs du formulaire HTML ci-dessus qui sera posté par le bouton [Envoyer]. Il lui faut donc un champ **cmbLogins** qui recevra la valeur du champ HTML **cmbLogins**, c.a.d le login choisi par l'utilisateur.
- du serveur vers le client, l'objet **formLogins** est utilisé pour fixer l'élément sélectionné dans le combo. Les valeurs de la liste du combo seront elles fournies d'une façon qui sera expliquée ultérieurement.

VII.4.5 Les actions de l'application

Les actions sont assurées par des objets de type **Action** ou dérivés. La configuration des actions est faite à l'intérieur des balises <**action-mappings**> :

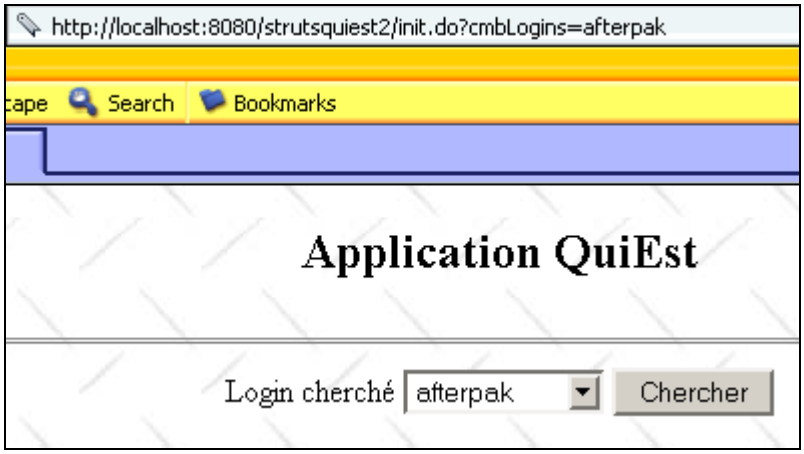
```
<action-mappings>

  <action
    path="/init"
    name="formLogins"
    validate="false"
    scope="session"
    type="istia.st.struts.quiest.SetupLoginsAction"
  >
    <forward name="afficherLogins" path="/vues/logins.jsp"/>
    <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
  </action>

  <action
    path="/infosLogin"
    name="formLogins"
    validate="false"
    scope="session"
    type="istia.st.struts.quiest.InfosLoginAction"
  >
    <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    <forward name="afficherInfos" path="/vues/infos.jsp"/>
  </action>

  <action
    path="/retourLogins"
    parameter="/vues/logins.jsp"
    type="org.apache.struts.actions.ForwardAction"
  />

</action-mappings>
```

L'action /init

```
    <action
      path="/init"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.SetupLoginsAction"
    >
      <forward name="afficherLogins" path="/vues/logins.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    </action>
```

Décrivons le fonctionnement de l'action **/init** :

- l'action /**init** a lieu normalement une unique fois lors du 1er cycle demande-réponse où l'utilisateur demande l'URL **http://localhost:8080/strutsquiest2/init.do**
- l'objet **formsLogins** est créé ou recyclé. Il est récupéré (recyclage) ou placé (création) dans la session comme le demande l'attribut **scope**.
- sa méthode **reset** est appelée. Rappelons que cette méthode ne fait rien par défaut dans les classes **ActionForm** et dérivées. Elle est appelée juste avant la recopie des données de la requête du client dans l'objet **ActionForm** et sert à nettoyer l'objet avant cette recopie. Quelle est ici, la requête du client ? L'action **/init** est déclenchée lorsque l'url demandée est **http://localhost:8080/strutsquiest2/init.do**. Cette URL peut être demandée par un GET ou un POST. Il suffit que l'on mette dans cette requête des paramètres portant le nom de champs de **formLogins** pour que ceux-ci soient initialisés comme le montre l'exemple suivant :



- la requête contient le paramètre **cmbLogins** (*afterpak*). Le contrôleur Struts a donc recopié la valeur de ce paramètre dans le champ **cmbLogins** de **formLogins**. L'action **SetupLoginsAction** s'est ensuite déroulée et s'est achevée par l'affichage de la vue **logins.jsp**. Cette vue a un formulaire dont certains champs reçoivent leur valeur de **formLogins**. Ainsi le champ HTML **select** appelé **cmbLogins** a reçu sa valeur du champ **cmbLogins** (=*afterpak*) de **formLogins**. C'est pourquoi, la liste de logins apparaît positionnée sur le login *afterpak*.
- normalement, l'action **/init** est appelée sans paramètres du client mais rien n'empêche celui-ci de l'invoquer avec des paramètres arbitraires. A l'issue de la phase de "peuplement" du bean **formLogins**, le champ **cmbLogins** peut ainsi avoir une valeur. L'action **SetupLoginsAction** aurait pu être utilisée pour affecter la chaîne vide au champ **cmbLogins** de **formLogins**. Nous ne l'avons pas fait ici.
- une fois le peuplement de **formLogins** opéré, le contrôleur regarde l'attribut **validate** de l'action. Ici, il a la valeur "false". La méthode **validate** de **formLogins** ne sera pas appelée. Nous ne l'écrirons donc pas.
- l'objet **SetupLoginsAction** est créé ou recyclé s'il existe déjà et sa méthode **execute** lancée. Son seul rôle est d'affecter une valeur à un objet appelé **cLogins** qui sera placé dans la session. Cette valeur est le tableau des logins, tableau qui sera demandé à la classe métier **users**. Cette opération peut échouer. C'est pourquoi l'action **/init** peut être suivie de deux vues :
    - la vue **erreurs.jsp** si la classe **users** n'a pu fournir le tableau des logins
    - la vue **logins.jsp** sinon
- le contrôleur fera afficher l'une de ces deux vues
- le cycle demande-réponse de l'action /**init** est terminé.

L'action /infosLogin

```
    <action
      path="/infosLogin"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.InfosLoginAction"
    >
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
      <forward name="afficherInfos" path="/vues/infos.jsp"/>
```

```
    </action>
```

Décrivons le fonctionnement de l'action **/ infosLogin** :

- l'action /**infosLogin** a lieu normalement lorsque l'utilisateur clique sur le bouton [Chercher] de la vue **logins.jsp**. Une requête est alors faite au serveur, fixée par la balise HTML **<form>** de la vue :

```
<html:form name="formLogins" method="get" action="/infosLogin" type="org.apache.struts.action.DynaActionForm">
```

- on voit que la requête est envoyée au serveur par la méthode **get**. L'utilisateur peut donc la taper à la main :

- l'objet **formsLogins** est créé ou recyclé. Il est récupéré (recyclage) ou placé (création) dans la session comme le demande l'attribut **scope**.
- sa méthode **reset** est appelée juste avant la recopie des données de la requête du client dans l'objet **ActionForm**. Elle est normalement de la forme **http://localhost:8080/strutsquiest2/infosLogin.do?cmbLogins=xx** où **xx** est un login choisi dans la liste des logins. Mais ce peut être aussi n'importe quoi si l'utilisateur a utilisé l'URL précédente en passant des paramètres arbitraires. Revenons à une requête normale à l'action /**infosLogin** du genre :
  http://localhost:8080/strutsquiest2/infosLogin.do?cmbLogins=afterpak
- Le contrôleur Struts va affecter une valeur au champ **cmbLogins** de l'objet **ActionForm**. Nous n'aurons donc pas à écrire de méthode **reset** personnalisée pour **formLogins**.
- une fois la méthode **reset** de **formLogins** appelée, le contrôleur recopie les données de la requête du client dans les champs de même nom de **formLogins**. Le champ **cmbLogins** va recevoir une valeur, le login choisi par l'utilisateur (*afterpak*).
- le contrôleur regarde ensuite l'attribut **validate** de l'action. Ici, il a la valeur "false". La méthode **validate** de **formLogins** ne sera pas appelée.
- l'objet **InfosLoginAction** est créé ou recyclé s'il existait déjà et sa méthode **execute** lancée. Son rôle est d'obtenir les informations associées au login **cmbLogins**. Ces informations seront demandées à la classe métier **users**. Cette opération peut échouer (login inexistant par exemple). C'est pourquoi l'action **/infosLogin** peut être suivie de deux vues :
  - la vue **erreurs.jsp** si la classe **users** n'a pu fournir les renseignements demandés
  - la vue **infos.jsp** sinon
- le contrôleur fera afficher l'une de ces deux vues
- le cycle demande-réponse de l'action /**infosLogin** est terminé.

L'action /retourLogins

```
    <action
      path="/retourLogins"
      parameter="/vues/logins.jsp"
      type="org.apache.struts.actions.ForwardAction"
    />
```

- l'action /**retourLogins** est déclenchée par activation du lien [Retour au formulaire] des vues **erreurs.jsp** et **infos.jsp**.

- ici il n'y a pas de formulaire associé à l'action. On passe donc tout de suite à l'exécution de la méthode **execute** d'un objet **ForwardAction** qui va rendre un objet **ActionForward** pointant sur la vue **/vues/logins.jsp**.

VII.4.6 Le fichier de messages de l'application

La troisième section du fichier **struts-config.xml** est celui du fichier des messages :

```
    <message-resources
    parameter="istia.st.struts.quiest.ApplicationResources"
     null="false"
  />
```

Le fichier **ApplicationResources.properties** est placé dans **WEB-INF/classes/istia/st/struts/quiest**. Son contenu est le suivant :

```
errors.header=<ul>
errors.footer=</ul>
parametreManquant=<li>Le paramètre [{0}] n'a pas été initialisé</li>
usersException=<li>Erreur d'initialisation de l'application : {0}</li>
loginInconnu=<li>Le login [{0}] n'existe pas</li>
```

VII.5Les contrôleurs, actions et vues de l'application

Le fichier web.xml référence une classe Java :

```
<web-app>
  <servlet>
    <servlet-name>strutsquiest2</servlet-name>
     <servlet-class>istia.st.struts.quiest.Quiest2ActionServlet</servlet-class>
....
  </servlet>

...
</web-app>
```

Le fichier de configuration **struts-config.xml** référence lui deux classes Java :

```
    <action
      path="/infosLogin"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.InfosLoginAction"
    >
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
      <forward name="afficherInfos" path="/vues/infos.jsp"/>
    </action>

    <action
      path="/init"
      name="formLogins"
      validate="false"
      scope="session"
      type="istia.st.struts.quiest.SetupLoginsAction"
    >
      <forward name="afficherLogins" path="/vues/logins.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    </action>
```

Détaillons ces classes et leurs relations avec les vues de l'application.

VII.5.1 La classe Quiest2ActionServlet

La classe **Quiest2ActionServlet** est dérivée de la classe **ActionServlet**, la classe du contrôleur Struts. Nous dérivons la classe **ActionServlet** pour personnaliser sa méthode **init**. En effet, cette méthode, exécutée une seule fois au moment du chargement initial de la servlet, va nous permettre de construire un objet métier de type **users**. Cet objet n'a en effet besoin d'être construit qu'une seule fois et la méthode **init** est un bon endroit pour faire cette construction. L'objet **users** a besoin de deux fichiers pour se construire, les fichiers **passwd** et **group**. La localisation de ces deux fichiers est passée en paramètres à la servlet dans le fichier **web.xml** de l'application :

```
  <servlet>
    <servlet-name>strutsquiest2</servlet-name>
     <servlet-class>istia.st.struts.quiest.Quiest2ActionServlet</servlet-class>
     <init-param>
      <param-name>config</param-name>
```

```
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>passwdFileName</param-name>
      <param-value>data/passwd</param-value>
    </init-param>
    <init-param>
      <param-name>groupFileName</param-name>
      <param-value>data/group</param-value>
    </init-param>
  </servlet>
```

Le code de la servlet est le suivant :

```
package istia.st.struts.quiest;

import java.util.*;
import javax.servlet.*;
import org.apache.struts.action.*;
import istia.st.users.*;

public class Quiest2ActionServlet
  extends ActionServlet {

  // attributs de la servlet
  private users u = null;
  private ActionErrors erreurs = new ActionErrors();

  //init
  public void init() throws ServletException {

    // ne pas oublier d'initialiser la classe parent
    super.init();

    // variables locales
    final String[] initParams = {
      "passwdFileName", "groupFileName"};
    Properties params = new Properties();

    // on récupère les paramètres d'initialisation de la servlet
    ServletConfig config = getServletConfig();
    String servletPath = config.getServletContext().getRealPath("/");
    for (int i = 0; i < initParams.length; i++) {
      String valeur = config.getInitParameter(initParams[i]);
      if (valeur == null) {
        erreurs.add(ActionErrors.GLOBAL_ERROR, new ActionError("parametreManquant", initParams[i]));
        valeur = "";
      }
      // on mémorise le paramètre
      params.setProperty(initParams[i], valeur);
    } //for
    // retour s'il y a eu des erreurs d'initialisation
    if (erreurs.size() != 0) {
      return;
    }
    // on crée un objet users
    try {
      u = new users(servletPath + "/" + params.getProperty("passwdFileName"),
                    servletPath + "/" + params.getProperty("groupFileName"), null);
    }
    catch (Exception ex) {
      erreurs.add(ActionErrors.GLOBAL_ERROR, new ActionError("usersException", ex.getMessage()));
      return;
    } //catch
    // on récupère la liste des logins
    String[] tLogins = new String[u.getUsersByLogin().size()];
    Enumeration eLogins = u.getUsersByLogin().keys();
    for (int i = 0; i < tLogins.length; i++) {
      tLogins[i] = (String) eLogins.nextElement();
    }
    // on trie les logins
    Arrays.sort(tLogins);
    // on les met dans des beans
    LoginBean[] cLogins = new LoginBean[tLogins.length];
    for (int i = 0; i < tLogins.length; i++) {
      cLogins[i] = new LoginBean(tLogins[i]);
    }
    // on met le tableau des beans dans le contexte de l'application
    getServletContext().setAttribute("cLogins",cLogins);
  } //init
```

```
  // méthode d'accès aux infos privées de la servlet
  public Object[] getInfos() {
    return new Object[] {
      erreurs, u};
  }
}
```

Cette classe a deux attributs privés :

`users u`	instance de la classe métier **users** mettant à la disposition des différentes actions de l'application web, différentes méthodes donnant accès aux utilisateurs et groupes. Cette instance est construite par la méthode **init** de la classe.
`ActionErrors erreurs`	la constructions de l'instance **u** peut échouer. Dans ce cas, les erreurs sont mémorisées dans l'attribut **erreurs**.

Le fonctionnement de la méthode **init** est le suivant :
- tout d'abord, il y a appel à la méthode **init** de la classe parent (**ActionServlet**) pour que celle-ci s'initialise correctement
- puis il y a lecture des paramètres d'initialisation. S'il en manque, l'attribut privé **ActionErrors erreurs** est renseigné.
- si les paramètres d'initialisation sont bien là, un objet **users** est construit. Cette construction peut lancer une exception. Dans ce cas, l'attribut **ActionErrors erreurs** est renseigné sinon l'objet **users** construit est mémorisé dans l'attribut privé **users u**.
- si la construction s'est bien passée, on tire de l'objet créé la liste de tous les logins et on trie ceux-ci. Ce tableau trié est ensuite utilisé pour construire un tableau de beans **cLogins** sur lequel nous allons revenir. Ce tableau est mis dans le contexte de l'application et devient alors disponible pour tous les clients de celle-ci.
- la méthode publique **getInfos** permet d'obtenir les deux attributs privés (u, erreurs) dans un tableau d'objets.

VII.5.2 La classe LoginBean

Cette classe mémorise un login sous forme d'un bean plutôt que sous forme de chaîne de caractères. Sous cette forme, le login devient accessible aux bibliothèques de balises Struts qui travaillent essentiellement avec des beans. Le bean est le suivant :

```
package istia.st.struts.quiest;

public class LoginBean implements java.io.Serializable{
  // bean conteneur d'un login
  private String login;

  //constructeurs
  public LoginBean(){
    // constructeur par défaut
  }
  public LoginBean(String login){
    this.login=login;
  }
  // accesseur
  public String getLogin(){
    return login;
  }
  // initialiseur
  public void setLogin(String login){
    this.login=login;
  }
}
```

Le bean définit un champ privé **login** ainsi que les méthodes **get** et **set** permettant d'y accéder. Par ailleurs, deux constructeur sont fournis.

VII.5.3 La classe SetupLoginsAction

Cette action a pour but de démarrer la session d'un client. Elle n'a lieu qu'une fois et permet de savoir si le contexte de l'application a pu s'initialiser correctement. Son code est le suivant :

```
package istia.st.struts.quiest;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;

public class SetupLoginsAction
  extends Action {

  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse response) throws
IOException,
```

```
    ServletException {
    // prépare le formulaire à afficher

    // on récupère les infos de la servlet contrôleur
    Object[] infos = ( (Quiest2ActionServlet)this.getServlet()).getInfos();

    // y-a-t-il eu des erreurs d'initialisations ?
    ActionErrors erreurs = (ActionErrors) infos[0];
    if (!erreurs.isEmpty()) {
      // on affiche la page d'erreurs
      this.saveErrors(request, erreurs);
      return mapping.findForward("afficherErreurs");
    }

    // pas d'erreurs - on affiche les logins
    return mapping.findForward("afficherLogins");
  }
}
```

Comme pour toutes les actions Struts, le code se trouve dans la méthode **execute**. Celle-ci :
- récupère auprès du contrôleur Struts les informations que celui-ci a mémorisées grâce à sa méthode **init**. C'est la méthode **getServlet**() de la classe **Action** qui permet cela.
- parmi celles-ci se trouve l'attribut **ActionErrors erreurs** du contrôleur. Si cette liste d'erreurs est non vide, elle est placée dans la requête et on demande l'affichage de la vue **erreurs.jsp**.
- si la liste des erreurs est vide, alors on demande ensuite l'affichage de la vue **logins.jsp** qui va afficher cette liste.

VII.5.4 La vue logins.jsp

Rappelons que cette vue est affichée dans deux cas :
- à l'appel de l'action **/init** lors du 1er cycle demande-réponse
- à l'appel de l'action **/retourLogins** lors des cycles suivants

Le code de la vue **logins.jsp** est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
  <title>Quiest - formulaire</title>
  </head>
  <body background="<html:rewrite page="/images/standard.jpg"/>">
  <center>
    <h2>Application QuiEst</h2>
      <hr>
      <html:form name="formLogins" action="/infosLogin" type="org.apache.struts.action.DynaActionForm">
      <table>
        <tr>
          <td>Login cherché</td>
            <td>
            <html:select name="formLogins" property="cmbLogins">
              <html:options collection="cLogins" property="login" labelProperty="login"/>
            </html:select>
            </td>
            <td>
            <html:submit value="Chercher"/>
            </td>
          </tr>
        </table>
      </html:form>
    </center>
  </body>
</html>
```

Explicitons les balises du combo des logins :

```
            <html:select name="formLogins" property="cmbLogins">
              <html:options collection="cLogins" property="login" labelProperty="login"/>
            </html:select>
```

La page **logins.jsp** utilise deux objets pour son affichage :
- le bean **formLogins** : son champ **cmbLogins** est utilisé pour savoir quel élément de la liste doit être sélectionné
- un tableau de beans **cLogins** qui contient la liste des logins à afficher dans le combo. L'objet **cLogins** a éré mis dans le contexte de l'application par la servlet **Quiest2ActionServlet**.

Nous n'avons pas encore rencontré la balise **struts-html** suivante :

```
              <html:options collection="cLogins" property="login" labelProperty="login"/>
```

Elle sert à générer les options d'une balise <select>. Les options sont prises dans l'objet **cLogins** identifié comme une collection. Rappelons que cet objet a été construit comme un tableau de beans de type **LoginBean**, chacun d'eux encapsulant un login. L'objet **cLogins** n'est pas un bean Java, ce qui explique que, dans la balise <html:options>, il ne soit pas référencé sous la forme **name="cLogins"** habituelle, car l'attribut **name** sert à désigner un bean. Le terme "**collection**" désigne un objet Java implémentant une collection : Array, ArrayList, ... Comme pour un bean, la collection **cLogins** est cherchée dans les portées suivantes et ce dans l'ordre : **page, request, session, context**. Ici, la collection **cLogins** sera trouvée dans la portée **context**.
Rappelons qu'une option de la balise HTML *select* a la forme :

<option value="XX">YY</option>

Chacune des options du <select> est générée à partir d'un **LoginBean** de la collection **cLogins**. La valeur XX est générée à partir du champ de **LoginBean** désigné par **property** et la valeur YY par le champ de **LoginBean** désigné par **labelProperty**. Le bean **LoginBean** n'a qu'un champ **login**, qui contient le login à afficher. Aussi les deux attributs **property** et **labelProperty** de la balise **<html:options>** désignent-ils ce champ. Cela aura pour effet de générer des options du genre

```
<option value="unLogin">unLogin</option>
```

VII.5.5 la vue erreurs.jsp

Cette vue est affichée lorsque les actions **/init** ou **/infosLogin** se terminent par une erreur. Son code est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
    <title>Application QuiEst - erreurs</title>
  </head>
  <body background="<html:rewrite page="/images/standard.jpg"/>">
    <h2 align="center">Application QuiEst - Erreurs</h2>
    <hr>
    <h2>Les erreurs suivantes se sont produites</h2>
    <html:errors/>
     <html:link page="/retourLogins.do">
      Retour au formulaire
    </html:link>
  </body>
</html>
```

VII.5.6 Les classes InfosLoginBean et InfosLoginAction

L'action **InfosLoginAction** a pour but de récupérer les informations associées au login choisi par l'utilisateur et de les présenter à celui-ci. Les informations seront rassemblées dans un objet de type **InfosLoginBean** :

```
package istia.st.struts.quiest;

public class InfosLoginBean implements java.io.Serializable{

  // bean détenant l'info nécessaire à la page d'infos
  private String titre;
  private String[] infosLogin;

  // constructeur
  public InfosLoginBean(String titre, String[] infosLogin){
    this.titre=titre;
    this.infosLogin=infosLogin;
  }

  // getters
  public String getTitre(){
    return this.titre;
  }
  public String[] getInfosLogin(){
    return this.infosLogin;
  }
  public String getInfosLogin(int i){
    return this.infosLogin[i];
  }
}
```

La classe précédente est un bean, c.a.d. une classe java où un attribut privé **T unAttribut** est automatiquement associé à deux méthodes privées :

- **void setUnAttribut(T valeur){unAttribut=valeur;}**
- **T getUnAttribut(){ return unAttribut;}**

On notera la syntaxe spéciale des méthodes **get** et **set**. Si l'attribut est un tableau **T[] unAttribut**, on peut créer des méthodes **get** et **set** pour les éléments du tableau :

- **void setUnAttribut(T valeur, int i){unAttribut[i]=valeur;}**
- **T getUnAttribut(int i){ return unAttribut[i];}**

Pour mieux comprendre, détaillons le code de la vue **infos.jsp** qui doit être envoyée à la suite de l'action **InfosLoginAction** :

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html>
  <head>
    <title><bean:write name="infosLoginBean" scope="request" property="titre"/></title>
  </head>
  <body background="<html:rewrite page="/images/standard.jpg"/>">
    <h2><bean:write name="infosLoginBean" scope="request" property="titre"/></h2>
     <hr>
     <table border="1">
       <tr>
         <th>login</th><th>pwd</th><th>uid</th><th>gid</th><th>id</th><th>dir</th><th>shell</th>
       </tr>
       <tr>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[0]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[1]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[2]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[3]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[4]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[5]"/></td>
         <td><bean:write name="infosLoginBean" scope="request" property="infosLogin[6]"/></td>
       </tr>
     </table>
     <br>
     <html:link page="/retourLogins.do">
       Retour au formulaire
     </html:link>
  </body>
</html>
```

Prenons la balise suivante :

```
<bean:write name="infosLoginBean" scope="request" property="titre"/>
```

Elle demande à écrire la valeur du champ **titre** (property) de l'objet **infosLoginBean** (name) mis dans la requête (scope). La valeur à écrire sera obtenue par **request.getAttribute("infosLoginBean").getTitre()**. Il faut donc que la méthode **getTitre** existe dans la classe **InfosLoginBean**. C'est le cas. La balise

```
<bean:write name="infosLoginBean" scope="request" property="infosLogin[0]"/>
```

demande à écrire la valeur de l'élément **infosLogin[0]** de l'objet **infosLoginBean** mis dans la requête. La valeur à écrire sera obtenue par **request.getAttribute("infosLoginBean").getInfosLogin(0)**. Il faut donc que la méthode **getInfosLogin(int i)** existe dans la classe **InfosLoginBean**. C'est le cas.

La classe **InfosLoginAction** a pour but de construire l'objet **InfosLoginBean** précédent à partir d'un login choisi par l'utilisateur. Son code est le suivant :

```
package istia.st.struts.quiest;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import istia.st.users.*;

public class InfosLoginAction
  extends Action {

  public ActionForward execute(ActionMapping mapping, ActionForm form,
                               HttpServletRequest request, HttpServletResponse response) throws
IOException,
    ServletException {

    // doit afficher les infos liées à un login

    // on récupère les infos de la servlet contrôleur
    Object[] infos = ( (Quiest2ActionServlet)this.getServlet()).getInfos();

    // y-a-t-il eu des erreurs d'initialisations ?
```

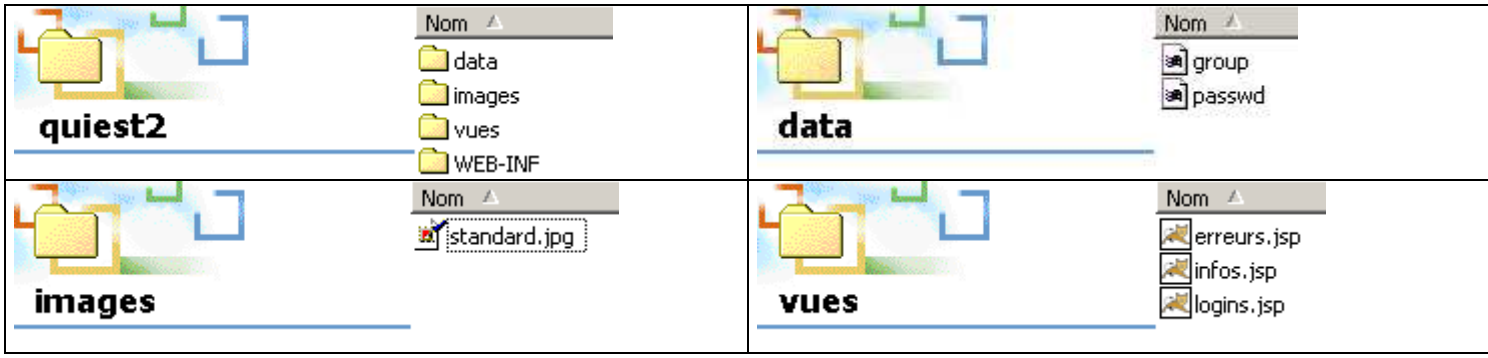
```
    ActionErrors erreurs = (ActionErrors) infos[0];
    if (!erreurs.isEmpty()) {
      this.saveErrors(request, erreurs);
      return mapping.findForward("afficherErreurs");
    }

    // d'abord récupérer ce login
    String login = (String) ( (DynaActionForm) form).get("cmbLogins");

    // a-t-on qq chose ?
    if (login == null) {
      return mapping.findForward("afficherLogins");
    }

    // on a un login - on le cherche
    String[] infosLogin = (String[]) ( (users) infos[1]).getUsersByLogin().get(login);

    // a-t-on trouvé ?
    if (infosLogin == null) {
      // le login n'a pas été trouvé - on affiche la page d'erreurs
      ActionErrors erreurs2=new ActionErrors();
      erreurs2.add(ActionErrors.GLOBAL_ERROR, new ActionError("loginInconnu", login));
      this.saveErrors(request, erreurs2);
      return mapping.findForward("afficherErreurs");
    }

    // le login a été trouvé - on met les infos trouvées dans la requête
    String titre="Application QuiEst - login["+login+"]";
    InfosLoginBean infosLoginBean= new InfosLoginBean(titre,infosLogin);
    request.setAttribute("infosLoginBean",infosLoginBean);
    return mapping.findForward("afficherInfos");
  }
}
```

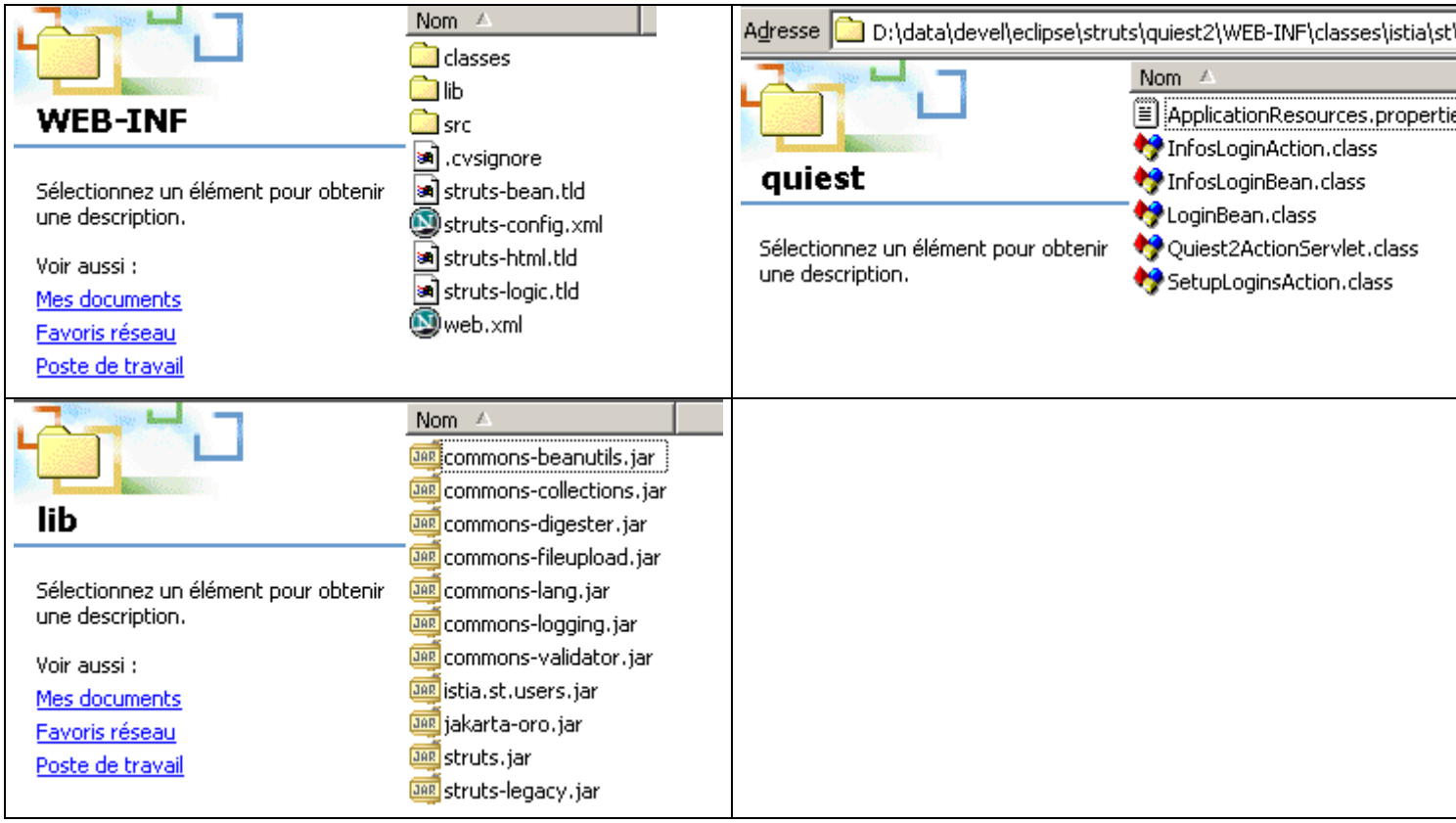
Le fonctionnement de la méthode **execute** est le suivant :

- on récupère les informations collectées par le contrôleur Struts lors de son initialisation. Si celui-ci avait noté des erreurs, l'exécution s'arrête là avec demande d'affichage de ces erreurs.
- on vérifie qu'il y a bien un login. Si l'utilisateur est passé par le formulaire de choix d'un login, le login est bien présent. Mais l'utilisateur peut très bien taper l'url de l'action directement dans son navigateur sans passer de paramètres. S'il n'y a pas de login, on fait réafficher la liste des logins.
- si on a un login, on demande les informations associées à la classe métier **users**. Si celle-ci ne trouve pas le login cherché, la page d'erreurs est affichée. Sinon, un objet **InfosLoginBean** est construit pour y mettre les informations dont a besoin la vue **infos.jsp**. Cet objet est mis dans la requête et on fait afficher la page **infos.jsp**.

VII.6 Déploiement

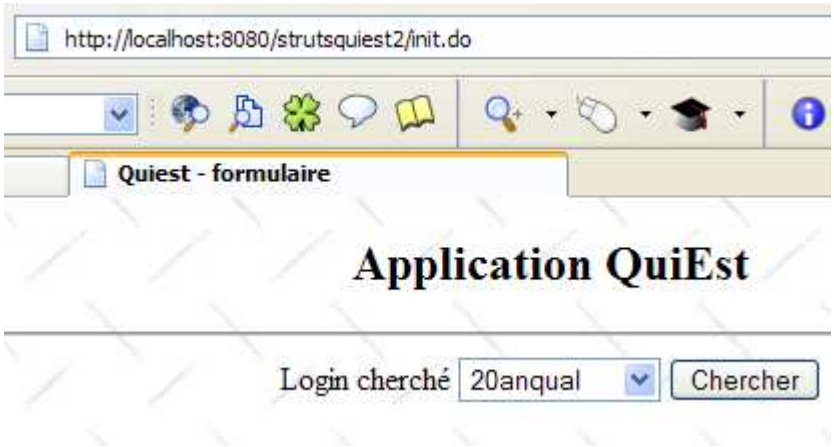
L'arborescence de l'application est la suivante :



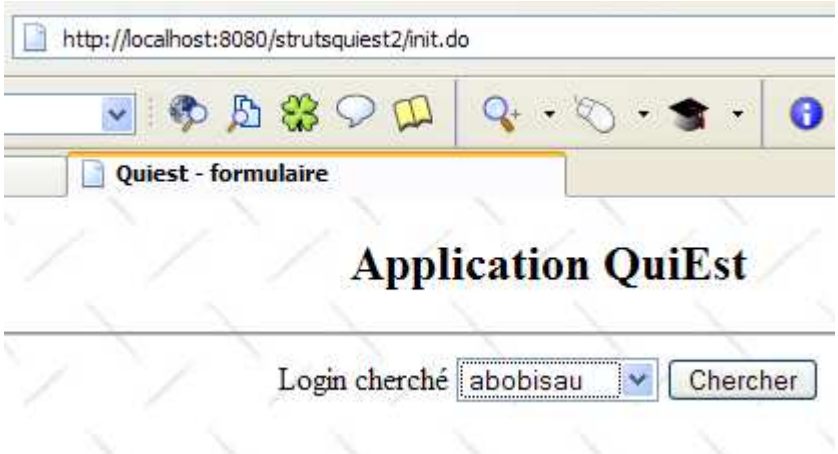


VII.7 Tests

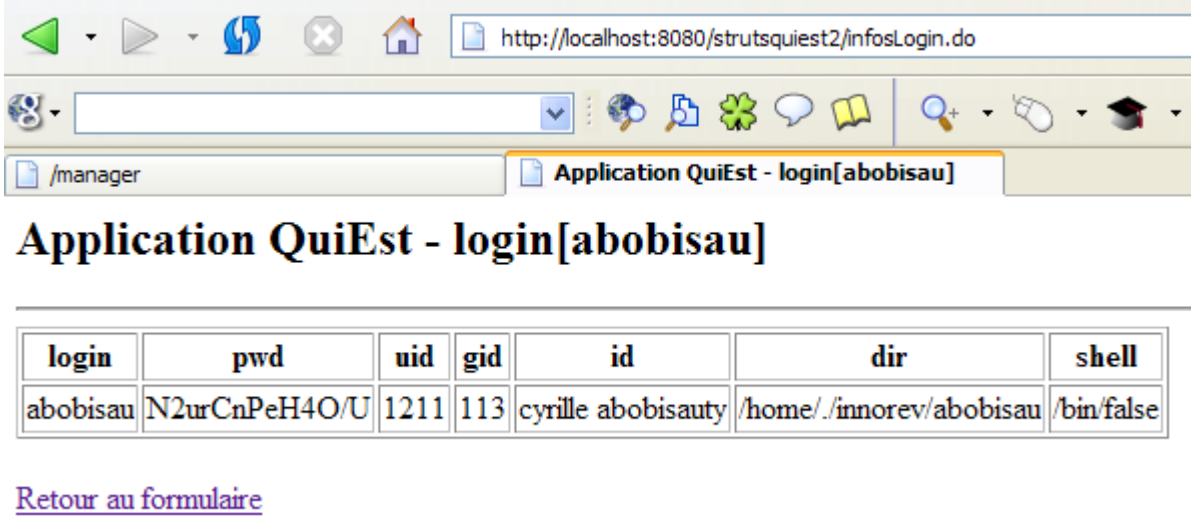
Nous demandons tout d'abord la liste des logins :



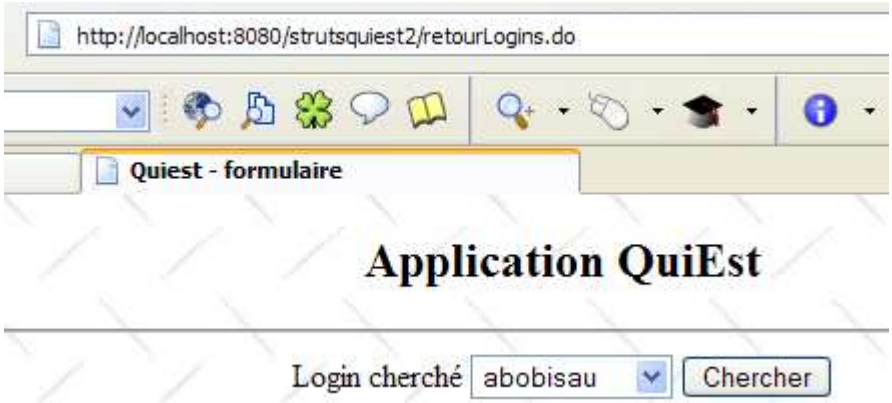
Nous en sélectionnons un en particulier :



Nous utilisons le bouton [Chercher] pour avoir les informations liées à ce login :



Nous utilisons le lien [Retour au formulaire] pour revenir aux logins :



Nous retrouvons le combo tel qu'il a été validé, ceci parce que le bean associé a été placé dans la session. Nous demandons maintenant directement dans l'url, un login inexistant :

VII.8 Conclusion

Nous avons utilisé Struts dans une application réaliste utilisant une classe métier. Nous avons par ailleurs montré qu'il fallait prêter particulièrement attention à la requête envoyée par un client et ne faire aucune hypothèse sur la nature de celle-ci. Une requête peut être quelconque et toute application doit commencer par vérifier la validité de celle-ci.

VIII Création d'un plugin

Il est possible de créer des applications appelées **plugins** qui sont chargées au démarrage d'une application Struts et déchargées lors de son arrêt. Cela permet classiquement d'opérer des initialisations au démarrage de l'application et des libérations de ressources lors de l'arrêt de celle-ci. Ces opérations peuvent être également faites en dérivant la classe **ActionServlet** du contrôleur et c'est ce qui a été fait dans l'application **quiest** précédente. Le plugin est une alternative à cette solution. Le plugin peut être une application plus complexe qu'une simple initialisation de l'environnement. Nous en avons vu un exemple dans la leçon où ont été introduites les notions de règles de validation déclaratives. Un plugin est déclaré dans le fichier de configuration de Struts. Le plugin **ValidatorPlugIn** était ainsi déclaré dans le fichier **struts-config.xml** de l'application qui l'utilisait :

```
  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
      property="pathnames"
       value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"
     />
  </plug-in>
```

Nous décrivons maintenant une application Struts utilisant un plugin.

VIII.1 Configuration de l'application Struts /plugin1

On se propose de construire l'application Struts **/plugin1** configurée de la façon suivante :

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Nous ne nous attarderons pas sur ce fichier qui est standard.

struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

  <plug-in className="istia.st.struts.plugins.MyPlugin">
    <set-property property="passwdFileName" value="data/passwd"/>
    <set-property property="groupFileName" value="data/group"/>
  </plug-in>

</struts-config>
```

Nous n'avons introduit qu'une section, celle pour le plugin. Les attributs de la balise **<plug-in>** utilisés ici sont les suivants :

`className`	nom de la classe du plugin
`<set-property>`	permet d'initialiser le plugin avec des couples (clé, valeur)

Le but de cette application est de montrer comment le plugin peut récupérer ses paramètres d'initialisation et les rendre disponibles aux autres objets partageant le même contexte d'application. Nous ferons appel à une simple vue JSP pour afficher les valeurs d'initialisation du plugin ce qui explique qu'il n'y a aucune action dans le fichier de configuration.

VIII.2 La classe java du plugin

Une classe Java servant de plugin à une application Struts doit implémenter l'interface **org.apache.struts.action.PlugIn**. Cette implémentation consiste à écrire deux méthodes :

- **public void init(ActionServlet servlet, ModuleConfig conf)**
- **public void destroy()**

Lorsque l'application Struts va être chargée, son contrôleur va instancier tous les plugins déclarés dans le fichier *struts-config.xml*. Puis, il exécutera la méthode **init** de chacun d'eux. C'est dans cette méthode que le plugin fait ses initialisations. Ici, nous nous contenterons de lire les paramètres d'initialisation du plugin et de mettre ceux-ci dans le contexte de l'application afin qu'ils soient disponibles pour tous les autres objets de celle-ci. Lorsque l'application va être déchargée, le contrôleur exécutera la méthode **destroy** de chacun des plugins chargés. C'est le moment de libérer des ressources devenues inutiles. Ici, nous n'aurons rien à faire.

Le code de la classe est le suivant :

```
package istia.st.struts.plugins;

import javax.servlet.ServletException;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ModuleConfig;
import org.apache.struts.config.PlugInConfig;

public class MyPlugin implements PlugIn {

  // méthode appelée à la suppression du contexte de l'application
  public void destroy() {
  }

  // méthode appelée lors de la création initiale du contexte de l'application
  public void init(ActionServlet servlet, ModuleConfig conf)
    throws ServletException {
    // nom de la classe de cet objet
    String className=this.getClass().getName();
    // liste des plugins
    PlugInConfig[] pluginConfigs = conf.findPlugInConfigs();
    // exploration des plugins pour trouver celui
    // qui porte le nom de cette classe
    boolean trouvé=false;
    for (int i = 0; ! trouvé && i < pluginConfigs.length; i++) {
      // nom du plugin
      String pluginClassName=pluginConfigs[i].getClassName();
      // si ce n'est pas le bon plugin, on continue
      if(! pluginClassName.equals(className)) continue;
      // c'est le bon - on mémorise ses propriétés dans le contexte
      servlet.getServletContext().setAttribute("initialisations",pluginConfigs[i].getProperties());
      trouvé=true;
    }//for i
  } //init
} //classe
```

A noter les points suivants :

- la méthode **init** reçoit un paramètre appelé ici **conf** de type **ModuleConfig** qui donne accès au contenu du fichier **struts-config.xml**.
- la méthode **[ModuleConfig].findPlugInConfigs()** permet de récupérer toutes les sections **<plug-in>** du fichier de configuration de Struts sous la forme d'un tableau d'objets **PlugInConfig**.
- la classe **PluginConfig** représente une section **<plug-in>** du fichier de configuration. La méthode **[PlugInConfig].getProperties** nous permet d'avoir accès aux éléments **<set-property>** de la section sous la forme d'un dictionnaire **java.util.Map**.
- le dictionnaire des propriétés du plugin est mis dans le contexte de l'application.
- comme il peut y avoir plusieurs plugins, il nous faut trouver celui qui nous intéresse. C'est celui qui porte le même nom que la classe qui s'exécute.

Nous avons préféré ici mettre la totalité du dictionnaire dans le contexte. Nous aurions pu vouloir exploiter celui-ci. La séquence de code suivante illustre une manière de faire :

```
      Map initialisations = pluginConfigs[i].getProperties();
      // itérer sur les entrées du dictionnaire
      Iterator entrées = initialisations.entrySet().iterator();
      while (entrées.hasNext()) {
```

```
        // récupérer l'entrée (clé,valeur) courante
        Map.Entry entrée = (Map.Entry) entrées.next();
        String clé = (String) entrée.getKey();
        String valeur = (String) entrée.getValue();
        // exploiter (clé,valeur)
        //...
      }
```

VIII.3 Le code de la vue infos.jsp

La vue **infos.jsp** est chargée de visualiser le contenu du dictionnaire des propriétés du plugin qui a été mis dans le contexte de l'application. Comme la vue **infos.jsp** fait partie de ce contexte, elle y a accès. Son code est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html>
  <head>
    <title>Plugin</title>
  </head>
  <body>
    <h3>Infos du plugin<br></h3>
    <table border="1">
      <tr>
        <th>Clé</th><th>Valeur</th>
      </tr>
      <logic:iterate id="element" name="initialisations">
        <tr>
          <td><bean:write name="element" property="key"/></td>
          <td><bean:write name="element" property="value"/></td>
        </tr>
      </logic:iterate>
    </table>
  </body>
</html>
```

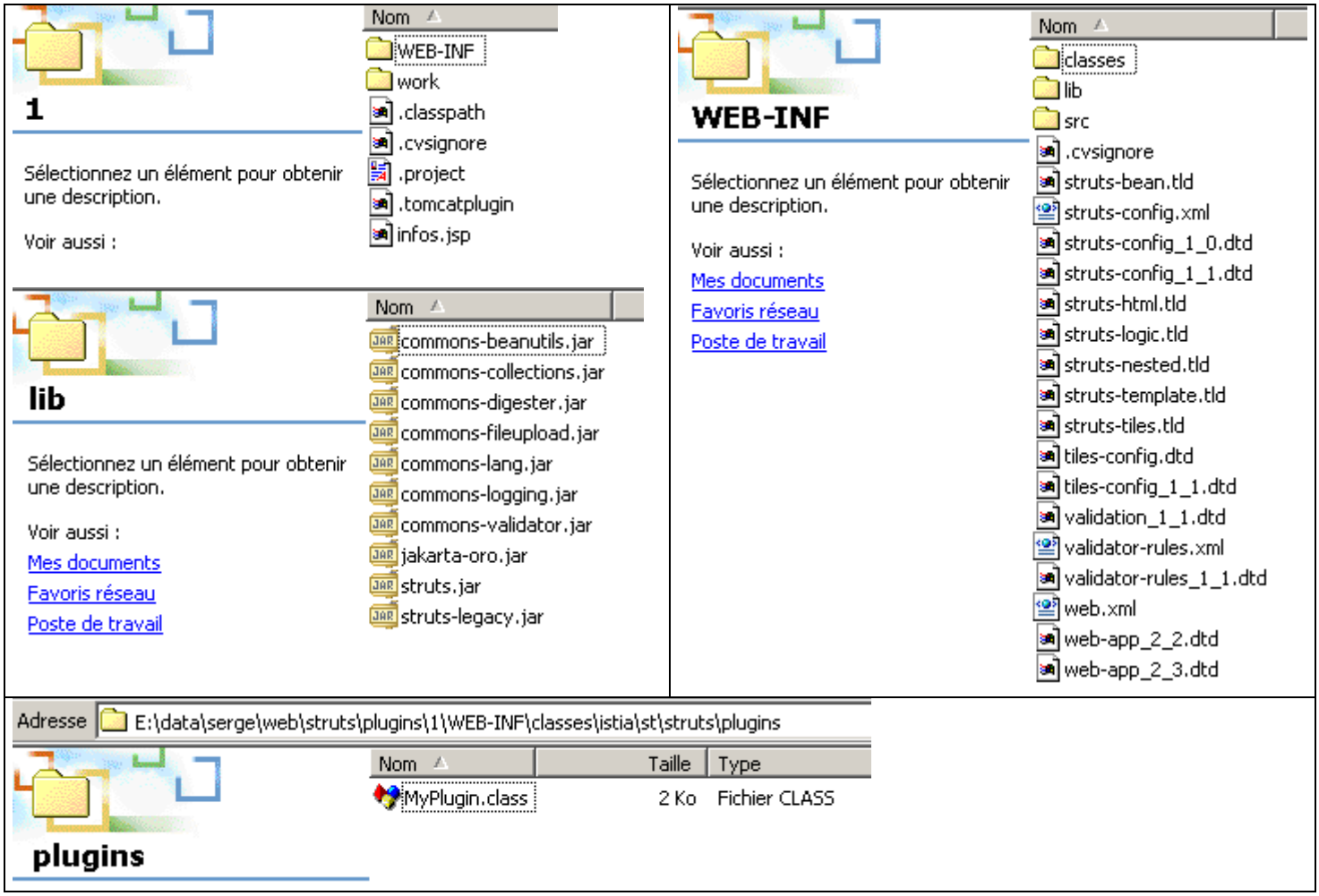
Points à noter :
- nous utilisons les bibliothèques de balises **struts-logic** et **struts-bean**
- la balise **<logic:iterate>** nous permet d'afficher le dictionnaire **initialisations** qui a été mis dans le contexte par la méthode **init** du plugin. L'attribut **name** désigne l'objet sur lequel il faut itérer. C'est quelque chose qui doit ressembler à une collection, un itérateur, ... L'objet est cherché dans toutes les portées (page, request, session, context). Ici il sera trouvé dans la portée **context**. Le paramètre **id** sert à donner un nom à l'élément courant de la collection au fil des itérations. Ici, dans le corps de la balise **<logic:iterate>**, **element** représentera l'élément courant d'un dictionnaire. Celui-ci est représenté par un objet **java.util.Map.Entry** qui est grosso modo un couple (clé, valeur) du dictionnaire.
- dans le corps de la balise **<logic:iterate>**, on fait afficher (**<bean:write>**) le contenu de l'élément courant **element**. Celui-ci est considéré comme un objet à deux propriétés (**property**) : la clé **key** et la valeur **value**. Ces deux propriétés sont affichées.

VIII.4 Déploiement

Le contexte de l'application est défini dans le fichier de configuration **plugin1.xml** dans <tomcat>\conf\Catalina\localhost :

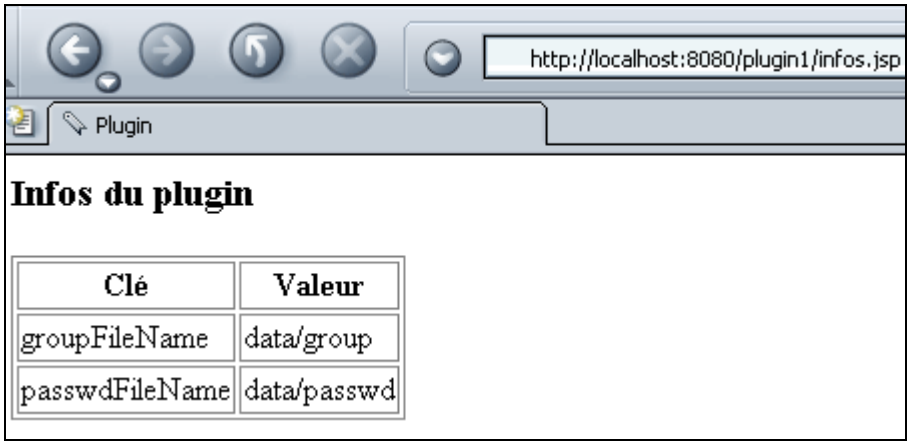
```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="D:/data/serge/devel/eclipse/struts/plugins/1" path="/plugin1">
</Context>
```

L'arborescence de l'application est la suivante :



VIII.5 Les tests

Nous lançons Tomcat et demandons l'URL http**://localhost:8080/plugin1/infos.jsp** :



Nous constatons que la vue **infos.jsp** a bien eu accès aux informations stockées dans le contexte de l'application par le plugin.

VIII.6 Conclusion

Nous avons montré qu'on pouvait initialiser une application Struts à l'aide d'un plugin. Ce peut être une solution alternative à celle de dériver la classe **ActionServlet** pour faire le même travail.

IX Exploitation d'une source de données

Nous nous proposons ici de poser les bases de l'exploitation de bases de données au sein d'une application Struts.

IX.1 L'application Struts /strutslistarticles

Nous souhaitons afficher le contenu d'une table enregistrant les caractéristiques d'articles vendus par un magasin.

Field	Type
code	varchar(4)
nom	varchar(30)
prix	double
stockActuel	int(11)
stockMinimum	int(11)

`code`	code de l'article - clé primaire
`nom`	nom de l'article
`prix`	prix de l'article
`stockActuel`	stock actuel de l'article
`stockMinimum`	stock au-dessous duquel il faut lancer un réapprovisionnement

Le contenu de la table est le suivant :

	code	nom	prix	stockActuel	stockMinimum
1	a100	cartable	20	7	3
2	b100	trousse	5	20	3
3	c100	lot de 6 crayons bille	4	30	10
4	d100	feuilles blanches - lot de 500	11	32	6
5	e100	stylo plume	20	40	17
6	f100	crayons de couleur	5	20	11
7	g100	compas	1	7	10

L'application /**strutslistarticles** nous donnera la même chose dans une page web :

Liste des articles

a100	cartable	20	7	3
g100	compas	1	7	10
f100	crayons de couleur	5	20	11
d100	feuilles blanches - lot de 500	11	32	6
c100	lot de 6 crayons bille	4	30	10
e100	stylo plume	20	40	17
b100	trousse	5	20	3

IX.2 Configuration de l'application Struts /strutslistarticles

Le fichier web.xml de l'application est standard :

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
```

```
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
       <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

</web-app>
```

Le fichier **struts-config.xml** introduit une nouvelle section **<data-sources>** qui permet de déclarer et configurer des sources de données :

struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <data-sources>
    <data-source key="dbarticles">
      <set-property property="driverClass" value="com.mysql.jdbc.Driver"></set-property>
      <set-property property="url" value="jdbc:mysql://localhost/dbarticles"></set-property>
      <set-property property="user" value="admarticles"></set-property>
      <set-property property="password" value="mdparticles"></set-property>
      <set-property property="minCount" value="2"></set-property>
      <set-property property="maxCount" value="5"></set-property>
    </data-source>
  </data-sources>

  <action-mappings>
    <action path="/liste" type="istia.st.struts.articles.ListeArticlesAction">
      <forward name="afficherListeArticles" path="/vues/listarticles.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    </action>
  </action-mappings>

  <message-resources parameter="istia.st.struts.articles.ApplicationResources"
    null="false" />

</struts-config>
```

IX.2.1 Sources de données

La section **<data-sources>** permet de déclarer toutes les sources de données de l'application, chacune d'elles étant décrite par une section **<data-source>**. Cette balise admet plusieurs attributs :

`key`	identifie la source de données lorsqu'il y en a plusieurs.
`driverClass`	le nom de la classe à instancier pour avoir accès à la base de données. Cette classe se trouve généralement dans une bibliothèque de classes (.jar) livrée par l'éditeur du SGBD. Cette bibliothèque contient le pilote JDBC d'accès à la base. La propriété **driverClass** désigne ce pilote. Le fichier .jar des classes d'accès au SGBD sera placé dans le dossier **WEB-INF/lib** de l'application.
`url`	chaîne de connexion à une base précise. En effet, le pilote JDBC permet l'accès à toutes les bases gérées par le SGBD. La propriété **url** nous permet d'indiquer celle que nous allons exploiter.
`user, password`	L'accès aux bases est protégé. Le SGBD gère des utilisateurs identifiés par un login et un mot de passe. Ceux-ci sont précisés dans les attributs **user** et **password** de la balise.
`minCount, maxCount`	Struts va gérer un pool de connexions au SGBD. L'ouverture d'une connexion à un SGBD est une opération coûteuse en ressources temps et mémoire. Plutôt que d'ouvrir et fermer une connexion au SGBD pour chacune des requêtes qui seront faites sur l'application, l'application gère un pool de n connexions où n est dans l'intervalle [minCount, maxCount]. Si lors d'une requête, l'application a besoin d'une connexion : - elle va chercher à en obtenir une dans le pool de connexions. Si elle en trouve une de libre, elle l'utilise. - s'il n'y a pas de connexions libres dans le pool et qu'il y a moins de **maxCount** connexions dans le pool, une

nouvelle connexion est ouverte et mise dans le pool. La requête courante pourra l'utiliser.
- s'il n'y a pas de connexions libres et pas moyen d'en créer une nouvelle, la requête est mise en attente.
- lorsque la requête courante ferme la connexion, celle-ci n'est pas fermée réellement mais rendue au pool.

Ici, nous avons les propriétés suivantes :

`key`	**dbarticles** - c'est le nom sous lequel sera connue la source de données dans le contexte de l'application
`driverClass`	**com.mysql.jdbc.Driver**. Nous utilisons une base MySQL.
`url`	**jdbc:mysql://localhost/dbarticles**. La base s'appelle **dbarticles** et est présente sur la machine locale.
`user, password`	**admarticles, mdparticles**. Cet utilisateur a été défini comme ayant tous les privilèges sur la base **dbarticles**.
`minCount, maxCount`	**2, 5**. Minimum de 2 connexions dans le pool, maximum de 5.

IX.2.2 Actions

Les actions déclarées dans le fichier de configuration de Struts sont les suivantes :

```
  <action-mappings>
    <action path="/liste" type="istia.st.struts.articles.ListeArticlesAction">
      <forward name="afficherListeArticles" path="/vues/listarticles.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
    </action>
  </action-mappings>
```

L'unique action s'appelle **/liste** et est associée à la classe **istia.st.struts.articles.ListeArticlesAction**. Cette action se termine soit par l'affichage :
- de la page des articles (**/vues/listarticles.jsp**)
- de la page des erreurs (**/vues/erreurs.jsp**)

IX.2.3 Le fichier des messages

```
  <message-resources parameter="istia.st.struts.articles.ApplicationResources"
    null="false" />
```

Le fichier des messages **ApplicationResources.properties** sera placé dans le dossier *WEB-INF/classes/istia/st/struts/articles*. Son contenu sera le suivant :

```
# erreurs
errors.header=<ul>
errors.footer=</ul>
erreur.dbarticles=<li>Erreur d'accès à la base des articles</li>
```

Une seule erreur peut se produire, un erreur d'accès à la base. Il y a en fait plusieurs types d'erreurs possibles, toutes rassemblées dans le même message d'erreur. La cause exacte de l'erreur sera néanmoins précisée dans le paramètre **{0}**.

IX.3Les vues

IX.3.1 La vue erreurs.jsp

Cette vue doit afficher une liste d'erreurs. Nous l'avons rencontrée à plusieurs reprises déjà.

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>
  <head>
    <title>Liste d'articles - erreurs</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <html:errors/>
  </body>
</html>
```

Cette vue se contente d'afficher la liste des erreurs avec la balise **<html:errors/>**.

IX.3.2 La vue listarticles.jsp

Cette vue doit afficher le contenu de la table ARTICLES de la base de données. Son code est le suivant :

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<%
  // listarticles : ArrayList dans la requête
  // listArticles(i) : tableau (String[5]) de 5 éléments
%>

<html>
  <head>
    <title>DataSource Struts</title>
  </head>
  <body>
    <h3>Liste des articles</h3>
    <hr>
    <table border="1">
      <logic:iterate id="ligne" name="listArticles">
        <tr>
          <logic:iterate id="colonne" name="ligne">
            <td><bean:write name="colonne"/></td>
          </logic:iterate>
        </tr>
      </logic:iterate>
    </table>
  </body>
</html>
```

L'action **/liste** va mettre le contenu de la table ARTICLES dans un objet *ArrayList* placé dans la requête sous le nom **listArticles**. Chaque élément de l'objet **listArticles** est un tableau de 5 chaînes de caractères représentant les 5 informations (code, nom, prix, stockActuel, stockMinimum) liées à un article dans la table. La vue doit afficher le contenu de l'objet **listArticles** dans un tableau HTML. Elle s'aide pour cela de la balise **<logic:iterate>** :

```
      <logic:iterate id="ligne" name="listArticles">
        <tr>
          <logic:iterate id="colonne" name="ligne">
            <td><bean:write name="colonne"/></td>
          </logic:iterate>
        </tr>
      </logic:iterate>
```

Il y a ici deux itérations. La première boucle sur les éléments de l'objet **ArrayList listArticles**. L'élément courant de **listArticles** est ici appelé **ligne**. L'élément **ligne** représente un objet **String[5]** qu'on parcourt à l'aide de la deuxième itération. L'élément de cette seconde itération est appelé **colonne**. Il représente l'élément courant d'un tableau de chaînes de caractères et est donc une chaîne de caractères (code, nom, prix, stockActuel, stockMinimum). Sa valeur est affichée à l'aide de la balise **<bean:write>**.

La vue utilise des balises des bibliothèques **struts-logic** et **struts-bean**. Il faut donc déclarer l'utilisation de celles-ci :

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
```

IX.4L'action /liste

L'action **/liste** a pour but de mettre dans la requête un objet *ArrayList* représentant le contenu de la table ARTICLES, ceci afin qu'une vue l'exploite. Son code est le suivant :

```
package istia.st.struts.articles;

import java.io.IOException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class ListeArticlesAction extends Action {
```

```
  public ActionForward execute(ActionMapping mapping, ActionForm form,
      HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

    // lit le contenu de la table articles d'une connexion
    // réalisée à l'init du contexte
    // on récupère a source de données dbarticles
    DataSource dataSource = this.getDataSource(request, "dbarticles");
    if (dataSource == null) {
      // la source de données n'a pas pu être créée
      ActionErrors erreurs = new ActionErrors();
      erreurs.add("dbarticles", new ActionError("erreur.dbarticles",
          "La source de données n'a pu être créée"));
      this.saveErrors(request, erreurs);
      return mapping.findForward("afficherErreurs");
    }
    // ici la source de données existe - on l'exploite
    Connection connexion = null;
    Statement st = null;
    ResultSet rs = null;
    String requête = null;
    ArrayList alArticles = new ArrayList();
    // on gère les erreurs
    try {
      // obtenir une connexion
      connexion = dataSource.getConnection();
      // préparer la requête SQL
      requête = "select code, nom, prix, stockActuel, stockMinimum from articles order by nom";
      // l'exécuter
      st = connexion.createStatement();
      rs = st.executeQuery(requête);
      // exploiter les résultats
      while (rs.next()) {
        // enregistrer la ligne courante
        alArticles.add(new String[] { rs.getString("code"),
            rs.getString("nom"), rs.getString("prix"),
            rs.getString("stockactuel"), rs.getString("stockMinimum") });
        // ligne suivante
      } //while
      // libérer les ressources
      rs.close();
      st.close();
    } catch (Exception ex) {
      // des erreurs se sont produites
      ActionErrors erreurs = new ActionErrors();
      erreurs.add("dbarticles", new ActionError("erreur.dbarticles"));
      this.saveErrors(request, erreurs);
      return mapping.findForward("afficherErreurs");
    } finally {
      // on libère la connexion
      try {
        connexion.close();
      } catch (Exception ignored) {
      }
    }
    // c'est bon
    request.setAttribute("listArticles", alArticles);
    return mapping.findForward("afficherListeArticles");
  } //execute
} //classe
```

Le lecteur est sans doute capable de comprendre l'essence du code ci-dessus. Nous ne nous attarderons que sur l'unique nouveauté, l'utilisation d'un objet **DataSource** fourni par le framework Struts. Cet objet *DataSource* représente le pool de connexions configuré par la section **<data-source key="dbarticles">** du fichier de configuration. L'objet *DataSource* a été placé, après sa création, dans le contexte de l'application afin que tous les ojets de ce contexte y aient accès. Il est donc récupéré de la façon suivante :

```
    // on récupère la source de données dbarticles
    DataSource dataSource = this.getDataSource(request, "dbarticles");
```

Nous demandons ici la source de données identifiée par la clé "dbarticles". Si nous récupérons un pointeur *null*, c'est ce que la source de données n'a pu être créée lors de l'initilisation de l'application. Dans ce cas, nous notons l'erreur dans un objet **ActionErrors** et nous passons la main à la page d'erreurs identifiée par la clé "**afficherErreurs**" dans le fichier de configuration :

```
    <action path="/liste" type="istia.st.struts.articles.ListeArticlesAction">
      <forward name="afficherListeArticles" path="/vues/listarticles.jsp"/>
      <forward name="afficherErreurs" path="/vues/erreurs.jsp"/>
```

```
    </action>
  </action-mappings>
```

Une fois la source de données obtenue, nous avons accès au pool des connexions. Une connexion à la base est obtenue par le code :

```
      // obtenir une connexion
      Connection connexion = dataSource.getConnection();
```

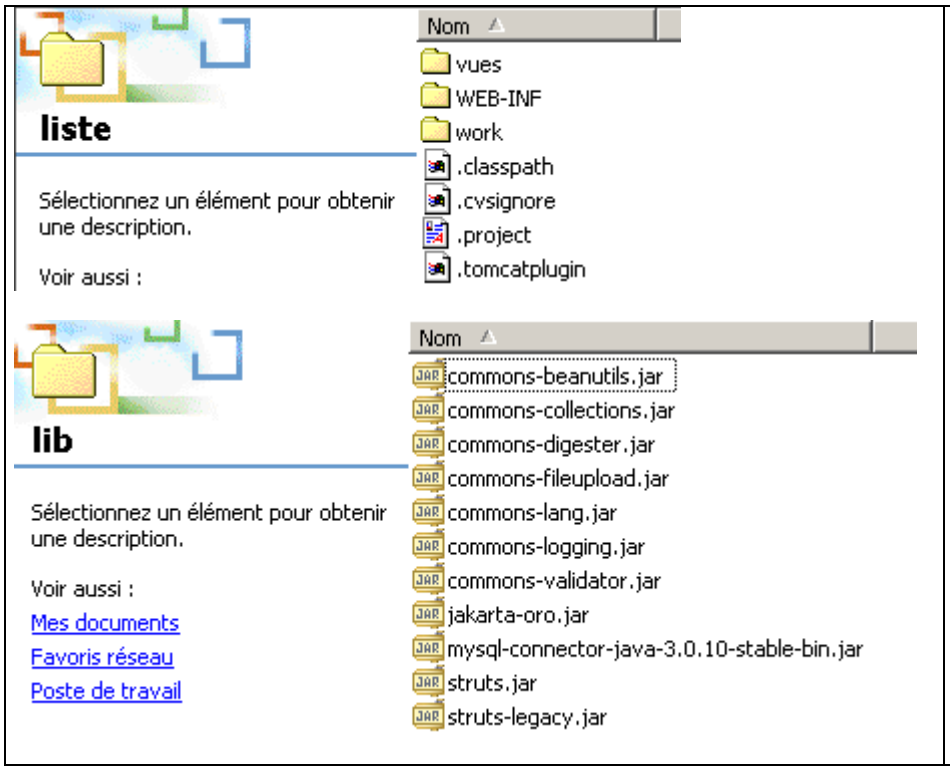
Ici, le pool nous délivrera une connexion recyclée ou créée s'il y avait encore possibilité d'en créer.

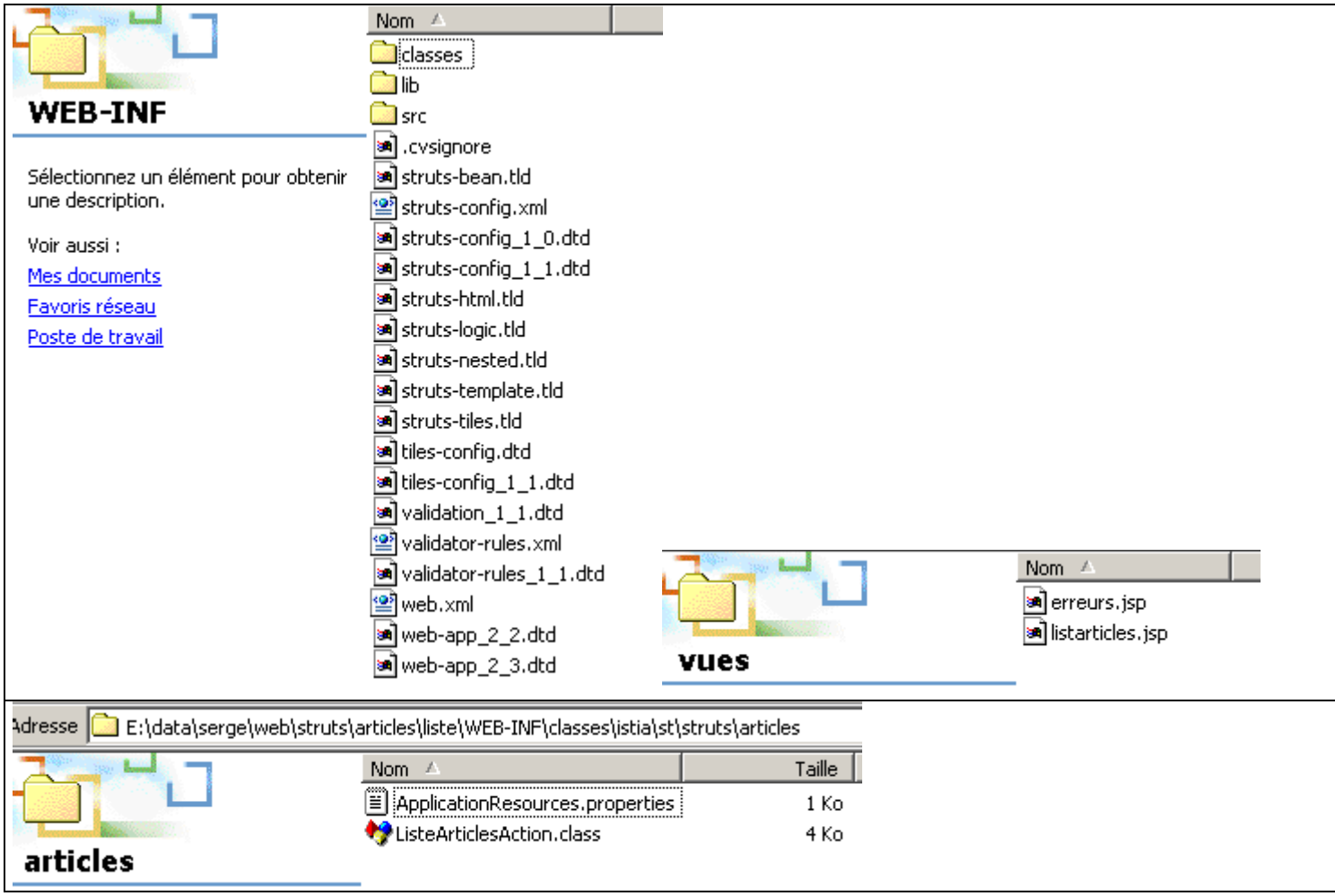
IX.5Déploiement

Le contexte de l'application est défini dans le fichier de configuration **strutslistarticles.xml** de <Tomcat>\conf\Catalina\localhost :

<Context path="/strutslistarticles" reloadable="true" docBase="..." />

L'arborescence de l'application est la suivante :

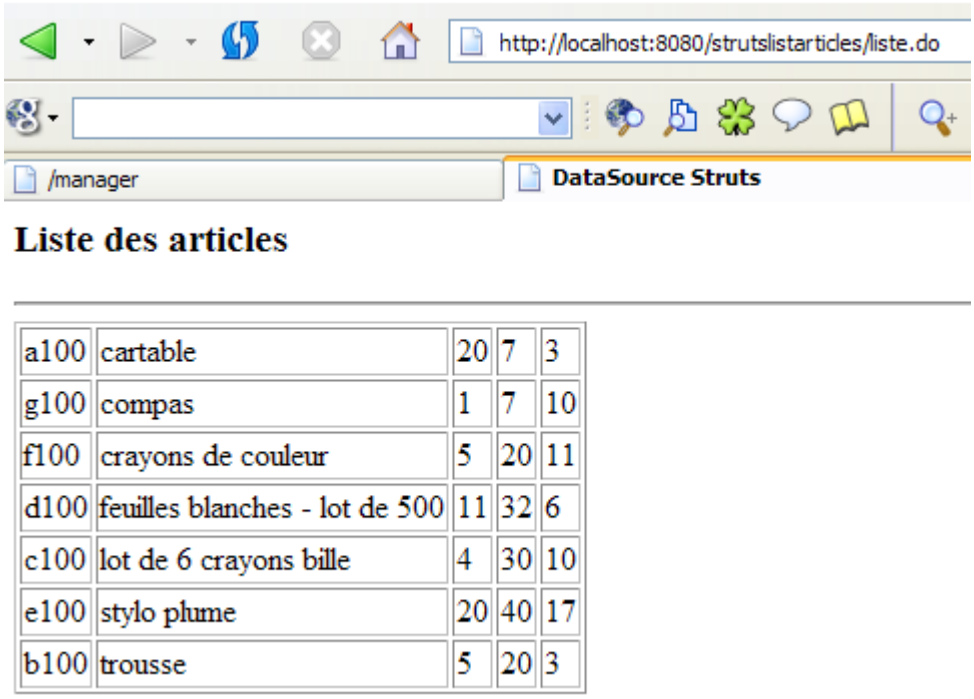




On remarquera la bibliothèque *mysql-connector-java-3.0.10-stable-bin.jar* dans **WEB-INF/lib**. C'est elle qui contient le pilote JDBC de la base MySQL utilisée ici.

IX.6Les tests

Nous lançons l'application **/strutslistarticles** et demandons l'URL **http://localhost:8080/strutslistarticles/liste.do** :



Nous arrêtons le SGBD et demandons la même url :

IX.7 Une seconde source de données

Nous utilisons maintenant une seconde source de données présente dans une base Postgres. Là également, les données sont dans une table ARTICLES avec les mêmes colonnes que précédemment. Son contenu est le suivant :

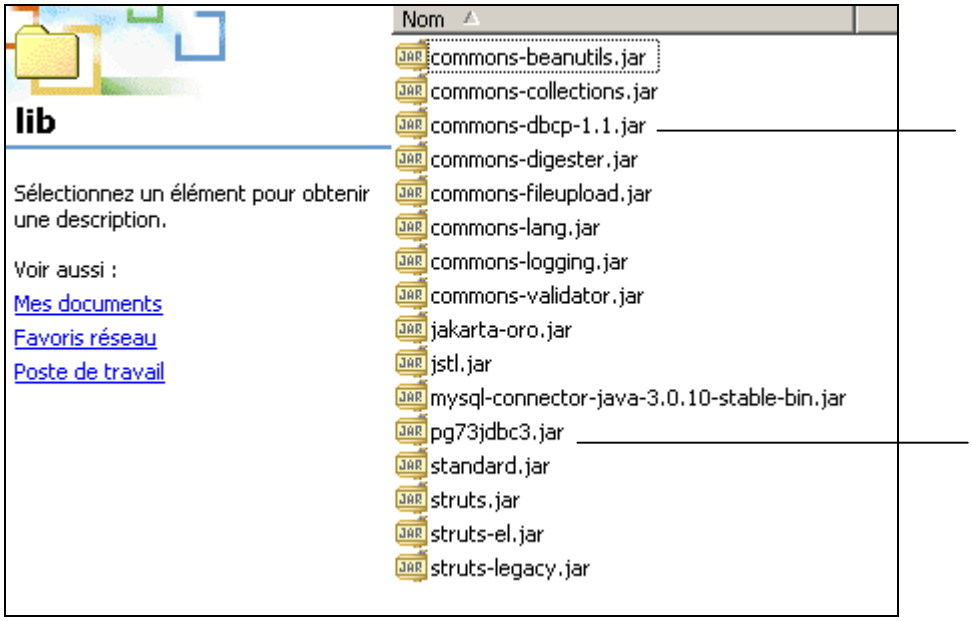
SQL Output (Table: articles): SELECT * FROM articles

code	nom	prix	stockactuel	stockminimum
a100	vélo	500	5	2
b100	pompe	10	3	2
c100	arc	1000	3	1

Cette nouvelle source de données est déclarée dans le fichier de configuration de Struts :

```
<data-sources>
  <data-source key="dbarticles" >
    <set-property property="driverClass" value="com.mysql.jdbc.Driver"></set-property>
    <set-property property="url" value="jdbc:mysql://localhost/dbarticles"></set-property>
    <set-property property="user" value="admarticles"></set-property>
    <set-property property="password" value="mdparticles"></set-property>
    <set-property property="minCount" value="2"></set-property>
    <set-property property="maxCount" value="5"></set-property>
  </data-source>
  <data-source key="pgdbarticles" type="org.apache.commons.dbcp.BasicDataSource">
    <set-property property="driverClassName" value="org.postgresql.Driver" />
    <set-property property="url" value="jdbc:postgresql://localhost/dbarticles" />
    <set-property property="username" value="serge" />
    <set-property property="password" value="serge" />
    <set-property property="maxActive" value="10" />
    <set-property property="maxWait" value="5000" />
    <set-property property="defaultAutoCommit" value="false" />
    <set-property property="defaultReadOnly" value="false" />
  </data-source>
</data-sources>
```

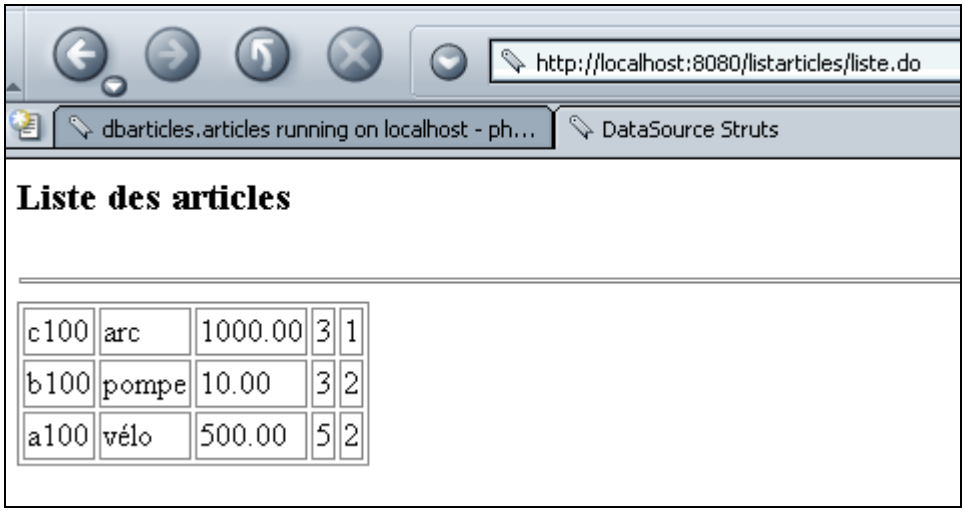
La nouvelle source aura pour identificateur (**key**) **pgdbarticles**. La nouveauté vient de la classe implémentant l'interface **javax.sql.DataSource**. La classe par défaut est **org.apache.struts.util.GenericDataSource** définie dans la bibliothèque **struts.jar**. Cette classe va être délaissée dans les versions futures de Struts et il est conseillé pour la version 1.1 d'utiliser la classe **org.apache.commons.dbcp.BasicDataSource** que l'on trouve dans la bibliothèque **commons-dbcp-1.1.jar**. Celle-ci n'est pas nécessairement livrée avec le paquetage Struts. On peut la trouver à l'url **http://jakarta.apache.org/commons/index.html** et plus particulièrement à l'url **http://jakarta.apache.org/site/binindex.cgi**. Il faut télécharger le produit appelé Commons DBCP. On peut pour windows, télécharger le fichier .zip. Celui-ci contient le code source de la bibliothèque ainsi que le .jar correspondant. Il faut extraire celui-ci de l'archive .zip et le placer dans le dossier WEB-INF/lib de l'application. Il faut également placer dans ce même dossier le pilote du SGBD utilisé :



Pour utiliser cette source, nous changeons une instruction dans le code de l'action **/liste** :

```
    DataSource dataSource = this.getDataSource(request, "pgdbarticles");
```

Nous demandons cette fois, la source de données s'appelant **pgdbarticles**, c.a.d. la source de données Postgres. Il ne nous reste plus qu'à compiler le tout, lancer Tomcat et demander l'URL **http://localhost:8080/listarticles/liste.do** :



IX.8 Une troisième source de données

Nous utilisons maintenant une source de données Firebird. Là également, les données sont dans une table ARTICLES avec les mêmes colonnes que précédemment. Son contenu est le suivant :

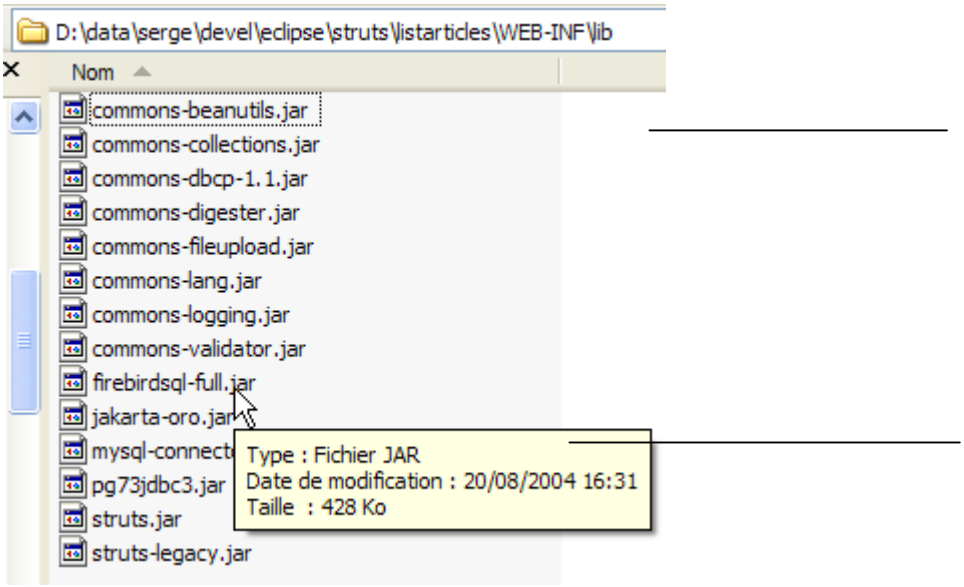
CO...	NOM	PRIX	STOCKACTUEL	STOCKMINIMUM
a100	cartable	20,000	7	3
d100	feuilles blanches - lot de 500	11,000	32	6
f100	crayons de couleur	5,000	20	11
g100	compas	1,000	7	10

Cette nouvelle source de données est déclarée dans le fichier de configuration de Struts :

```
  <data-sources>
    <data-source key="fbdbarticles" type="org.apache.commons.dbcp.BasicDataSource">
      <set-property property="driverClassName" value="org.firebirdsql.jdbc.FBDriver" />
      <set-property property="url"
value="jdbc:firebirdsql:localhost/3050:d:/data/serge/databases/firebird/dbarticles.gdb" />
      <set-property property="username" value="sysdba" />
```

```
      <set-property property="password" value="masterkey" />
      <set-property property="maxActive" value="10" />
      <set-property property="maxWait" value="5000" />
      <set-property property="defaultAutoCommit" value="false" />
      <set-property property="defaultReadOnly" value="false" />
    </data-source>
    <data-source key="dbarticles" >
      <set-property property="driverClass" value="com.mysql.jdbc.Driver"></set-property>
      <set-property property="url" value="jdbc:mysql://localhost/dbarticles"></set-property>
      <set-property property="user" value="admarticles"></set-property>
      <set-property property="password" value="mdparticles"></set-property>
      <set-property property="minCount" value="2"></set-property>
      <set-property property="maxCount" value="5"></set-property>
    </data-source>
</data-sources>
```

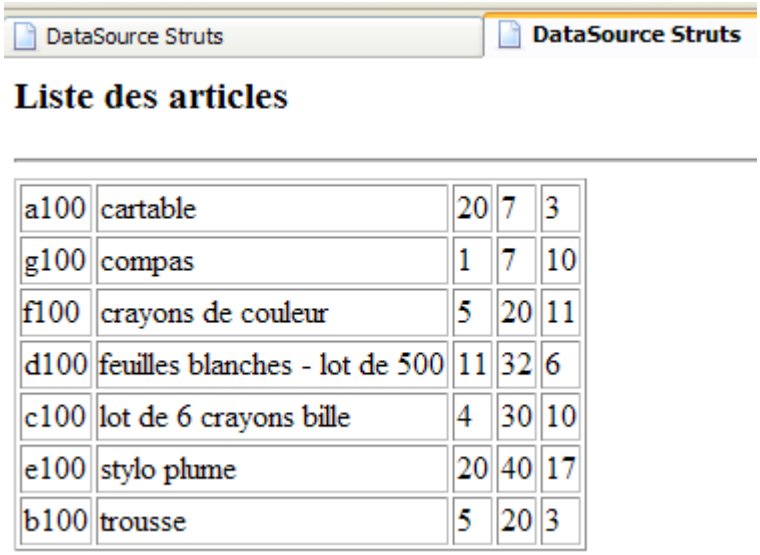
La nouvelle source aura pour identificateur (**key**) **fbdbarticles**. Le SGBD FireBird est disponible à l'url http://firebird.sourceforge.net/. Comme pour le SGBD Postgres, il nous faut mettre le pilote JDBC de Firebird dans WEB-INF/lib :



Pour utiliser cette nouvelle source, nous changeons une instruction dans le code de l'action **/liste** :

```
    DataSource dataSource = this.getDataSource(request, "fbdbarticles");
```

Nous demandons cette fois, la source de données s'appelant **fbdbarticles**, c.a.d. la source de données Firebird. Il ne nous reste plus qu'à compiler le tout, relancer l'application et demander l'URL **http://localhost:8080/listarticles/liste.do** :



IX.9 Conclusion

Nous avons montré diverses méthodes d'accès à une base de données. Remarquons ici que nous n'avons pas respecté l'architecture MVC. En effet, l'action **/liste** fait elle même les requêtes SQL pour accéder aux données. Il aurait été préférable qu'elle s'adresse à une classe métier intermédiaire qui cacherait le fait qu'on va chercher les données dans un SGBD.

I - GENERALITES 2

I.1 OBJECTIFS 2
I.2 LE MODELE MVC 2
I.3 UNE DEMARCHE DE DEVELOPPEMENT MVC A L'AIDE DE SERVLETS ET PAGES JSP 3
I.4 LA DEMARCHE DE DEVELOPPEMENT STRUTS 4

II LES OUTILS DU DEVELOPPEMENT WEB AVEC STRUTS 7

II.1 LA BIBLIOTHEQUE STRUTS 7
II.2 LE CONTENEUR DE SERVLETS TOMCAT 5 9
II.3 DEPLOIEMENT D'UNE APPLICATION WEB AU SEIN DU SERVEUR TOMCAT 12
II.4 DEPLOIEMENT D'UNE APPLICATION STRUTS AU SEIN DU SERVEUR TOMCAT 20
II.5 LES FICHIERS DE LOGS DE TOMCAT 24

III L'APPLICATION /STRUTSPERSONNE 26

III.1 LE FONCTIONNEMENT DE L'APPLICATION 26
III.2 L'ARCHITECTURE STRUTS DE L'APPLICATION 27
III.3 CONFIGURATION DE L'APPLICATION 28
III.4 LES VUES DE L'APPLICATION /STRUTSPERSONNE 29
III.4.1 LA VUE ERREURS.PERSONNE.JSP 29
III.4.2 TEST DE LA VUE ERREURS.PERSONNE.JSP 30
III.4.3 LA VUE REPONSE.PERSONNE.JSP 31
III.4.4 TEST DE LA VUE REPONSE.PERSONNE.JSP 32
III.4.5 LA VUE FORMULAIRE.PERSONNE.JSP 33
III.4.6 LE BEAN ASSOCIE AU FORMULAIRE FORMULAIRE.PERSONNE.JSP 34
III.4.7 COMPILER LES CLASSES NECESSAIRES A L'APPLICATION STRUTS 35
III.4.8 TEST DE LA VUE FORMULAIRE.PERSONNE.JSP 38
III.5 TRAITEMENT DU FORMULAIRE PAR LE CONTROLEUR 40
III.5.1 LE FICHIER STRUTS-CONFIG.XML 40
III.5.2 LA NOUVELLE CLASSE FORMULAIREBEAN 42
III.6 LES TESTS DE VALIDITE DU FORMULAIRE 44
III.6.1 TEST 1 44
III.6.2 TEST 2 45
III.6.3 TEST 3 47
III.6.4 TEST 4 48
III.6.5 TEST 5 48
III.6.6 TEST 6 49
III.6.7 TEST 7 50
III.7 NOUVELLE CONFIGURATION DE L'ACTION /MAIN 51
III.8 LA CLASSE FORMULAIREACTION 51
III.9 TESTS DE FORMULAIREACTION 52
III.10 CONCLUSION 54

IV LES FORMULAIRES DYNAMIQUES 55

IV.1 DECLARATION DU FORMULAIRE DYNAMIQUE 55
IV.2 ÉCRITURE DE LA CLASSE DYNAACTIONFORM ASSOCIEE AU FORMULAIRE DYNAMIQUE 55
IV.3 LA NOUVELLE CLASSE FORMULAIREACTION 56
IV.4 DEPLOIEMENT ET TEST DE L'APPLICATION STRUTSPERSONNE1 57
IV.4.1 CREATION DU CONTEXTE 57
IV.4.2 LES VUES 58
IV.4.3 COMPILATION DES CLASSES 58
IV.4.4 LE DOSSIER WEB-INF 58
IV.5 TESTS 59
IV.6 CONCLUSION 61

V FORMULAIRES DYNAMIQUES AVEC CONTRAINTES D'INTEGRITE 62

V.1 DECLARATION DES CONTRAINTES D'INTEGRITE 62
V.2 ÉCRITURE DES CONTRAINTES D'INTEGRITE DES FORMULAIRES DYNAMIQUES 63
V.3 LES CLASSES DE L'APPLICATION 66
V.4 DEPLOIEMENT ET TEST DE L'APPLICATION /STRUTSPERSONNE2 66
V.4.1 CREATION DU CONTEXTE 66
V.4.2 LES VUES 67
V.4.3 LE DOSSIER WEB-INF 67
V.5 TESTS 70
V.6 CONCLUSION 72

VI FORMULAIRES HTML 73

VI.1 LES VUES DE L'APPLICATION 73
VI.2 L'ARCHITECTURE DE L'APPLICATION 75
VI.3 LA CONFIGURATION DE L'APPLICATION 75
VI.3.1 CREATION DU CONTEXTE 75
VI.3.2 LE FICHIER WEB.XML 76
VI.3.3 LE FICHIER STRUTS-CONFIG.XML 77
VI.3.4 LES OBJETS (BEANS) FORMULAIRES DE L'APPLICATION 77
VI.3.5 LES ACTIONS DE L'APPLICATION 80
VI.3.6 LE FICHIER DE MESSAGES DE L'APPLICATION 83
VI.4 LE CODE DES VUES 83
VI.4.1 LA VUE FORMULAIRE.JSP 83
VI.4.2 LA VUE CONFIRMATION.JSP 89
VI.5 LES CLASSES JAVA 91
VI.5.1 LA CLASSE DYNAFORMULAIRE 91
VI.5.2 LA CLASSE INITFORMULAIREACTION 92
VI.6 DEPLOIEMENT 93
VI.7 TESTS 94
VI.8 CONCLUSION 97

VII APPLICATION QUIEST 98

VII.1 LA CLASSE USERS 98
VII.2 L'APPLICATION WEB QUIEST 99
VII.3 L'ARCHITECTURE DE L'APPLICATION 101
VII.4 CONFIGURATION DE L'APPLICATION WEB 101
VII.4.1 CREATION DU CONTEXTE 101
VII.4.2 LE FICHIER WEB.XML 102
VII.4.3 LE FICHIER STRUTS-CONFIG.XML 103
VII.4.4 LES OBJETS (BEANS) FORMULAIRES DE L'APPLICATION 104
VII.4.5 LES ACTIONS DE L'APPLICATION 104
VII.4.6 LE FICHIER DE MESSAGES DE L'APPLICATION 107
VII.5 LES CONTROLEURS, ACTIONS ET VUES DE L'APPLICATION 107
VII.5.1 LA CLASSE QUIEST2ACTIONSERVLET 107
VII.5.2 LA CLASSE LOGINBEAN 109
VII.5.3 LA CLASSE SETUPLOGINSACTION 109
VII.5.4 LA VUE LOGINS.JSP 110
VII.5.5 LA VUE ERREURS.JSP 111
VII.5.6 LES CLASSES INFOSLOGINBEAN ET INFOSLOGINACTION 111
VII.6 DEPLOIEMENT 113
VII.7 TESTS 114
VII.8 CONCLUSION 116

VIII CREATION D'UN PLUGIN 117

VIII.1 CONFIGURATION DE L'APPLICATION STRUTS /PLUGIN1 117
VIII.2 LA CLASSE JAVA DU PLUGIN 118
VIII.3 LE CODE DE LA VUE INFOS.JSP 119

VIII.4 DEPLOIEMENT **119**
VIII.5 LES TESTS **120**
VIII.6 CONCLUSION **120**

IX EXPLOITATION D'UNE SOURCE DE DONNEES **121**

IX.1 L'APPLICATION STRUTS /STRUTSLISTARTICLES **121**
IX.2 CONFIGURATION DE L'APPLICATION STRUTS /STRUTSLISTARTICLES **121**
IX.2.1 SOURCES DE DONNEES 122
IX.2.2 ACTIONS 123
IX.2.3 LE FICHIER DES MESSAGES 123
IX.3 LES VUES **123**
IX.3.1 LA VUE ERREURS.JSP 123
IX.3.2 LA VUE LISTARTICLES.JSP 123
IX.4 L'ACTION /LISTE **124**
IX.5 DEPLOIEMENT **126**
IX.6 LES TESTS **127**
IX.7 UNE SECONDE SOURCE DE DONNEES **128**
IX.8 UNE TROISIEME SOURCE DE DONNEES **129**
IX.9 CONCLUSION **131**