TUTORIEL
Servlets et pages JSP avec Eclipse et Tomcat

serge.tahe@istia.univ-angers.fr

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

I.1 Objectifs

On se propose ici de découvrir la programmation web en Java par une série de tests pratiques sur ordinateur. Si les copies d'écran ci-dessous ont été faites sous windows, les tests eux, pourraient être faits indifféremment sous Windows ou Linux. A la fin de cette série de manipulations qui devrait durer 5 à 6 h, le lecteur devrait avoir acquis les concepts de base de la programmation web en java.

Afin d'avoir la compréhension de ce qui est fait, l'usage du polycopié "Introduction à la programmation web en Java" est nécessaire. Des conseils de lecture sont donnés avant la plupart des tests à réaliser.

I.2 Les outils

Nous utiliserons pour ces tests les outils suivants :

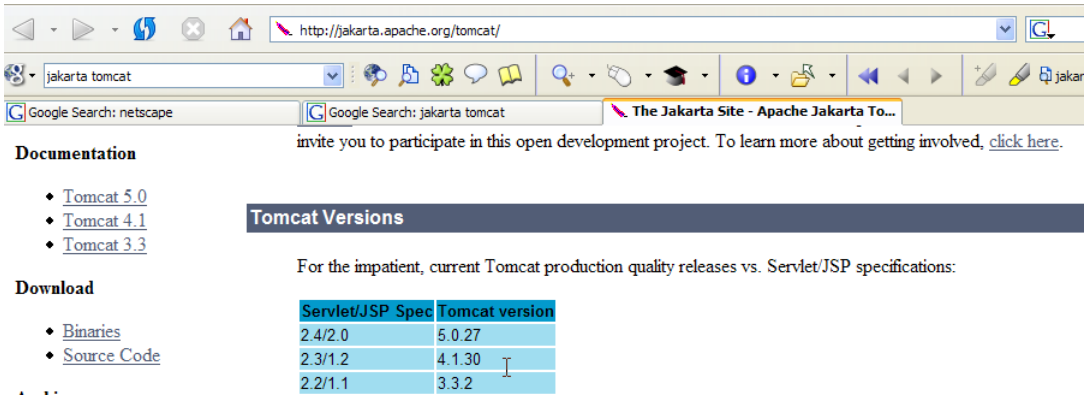
- le serveur web TOMCAT (http://jakarta.apache.org/tomcat/),
- l'outil de développement java ECLIPSE (http://www.eclipse.org/) avec les plugins suivants :
  - XmlBuddy pour gérer les documents XML (http://xmlbuddy.com/)
  - Tomcat de Sysdeo (http://www.sysdeo.com/eclipse/tomcatPlugin.html) pour gérer Tomcat à partir d'Eclipse
- un navigateur (IE, NETSCAPE, MOZILLA, OPERA, ...) :

Ce sont des outils gratuits. De nombreux outils libres peuvent être utilisés dans le développement Web :

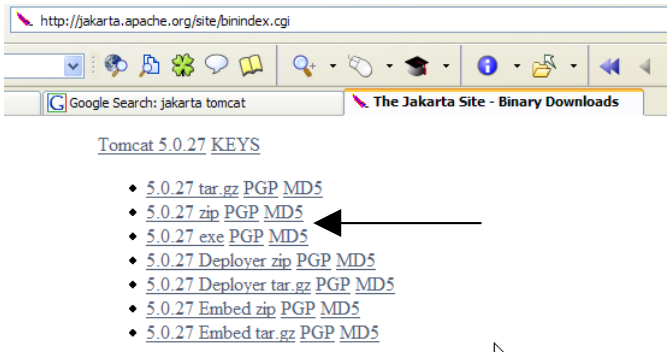
IDE JAVA	Jbuilder Foundation	http://www.borland.com/jbuilder/foundation/index.html
	Eclipse	http://www.eclipse.org/
Bibliothèque JAVA	Struts	http://struts.apache.org/
	Spring	http://www.springframework.org
SGBD	MySQL	http://www.mysql.com/
	Postgres	http://www.postgresql.org/
	Firebird	http://firebird.sourceforge.net/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

I.3 Le conteneur de servlets Tomcat 5

Pour exécuter des servlets, il nous faut un conteneur de servlets. Nous présentons ici l'un d'eux, Tomcat 5 disponible à l'url **http://jakarta.apache.org/tomcat/**. Nous indiquons la démarche (août 2004) pour l'installer pour le lecteur intéressé par l'installer sur son poste personnel.

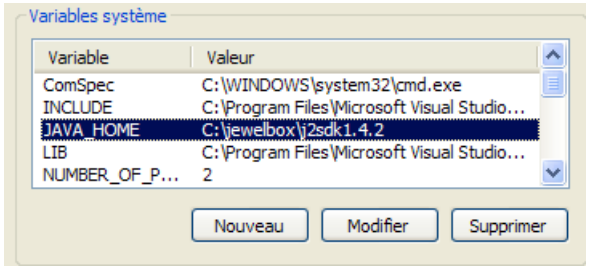


Pour télécharger le produit, on suivra le lien [Binaries] ci-dessus. On arrive alors à une page rassemblant tous les binaires des différents sous-projets du projet Jakarta de l'Apache Software Foundation. Nous choisissons celui de Tomcat 5 :

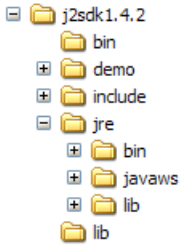


On pourra prendre le .zip ou le .exe destiné à la plate-forme windows. Le .exe est conseillé car il vient avec un assistant d'installation. Tomcat est une application Java et a donc besoin d'une machine virtuelle java (JVM) sur la plate-forme d'installation. Comme beaucoup d'applications Java, Tomcat 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] ->

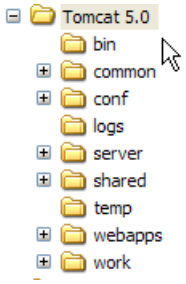


Ici JAVA_HOME désigne un JDK de Sun dont l'arborescence est la suivante :



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 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

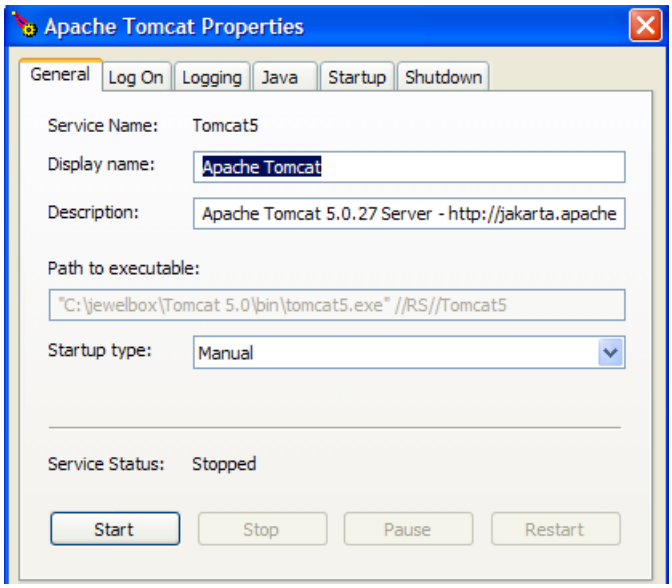
dans un dossier choisi par l'utilisateur, que nous appellerons désormais **<tomcat>**. L'arborescence de ce dossier pour la version Tomcat 5.0.27 ci-dessus est la suivante :



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 :

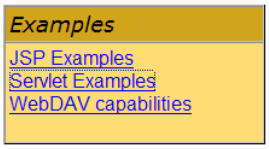
Une icône est insérée dans la barre des tâches en bas à droite de l'écran :

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

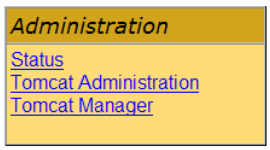


Les boutons [Start - Stop - Pause] - Restart nous permettent de lancer - arrêter - relancer le serveur. Nous lançons le serveur par [Start] puis avec un navigateur nous demandons l'url http://localhost:8080. Nous devons obtenir une page analogue à la suivante :

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 :



I.4 Déploiement d'une application web au sein du serveur Tomcat

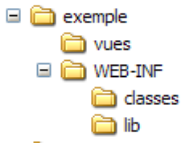
Lectures : chapitre 1, chapitre 2 : 2.3.1, 2.3.2, 2.3.3

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 :

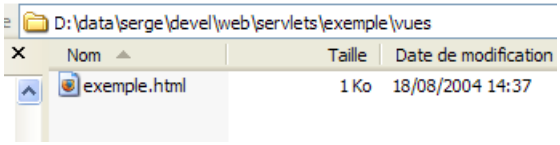
`classes`	elles seront placées dans le dossier <webapp>\WEB-INF\classes
`archives java`	elles seront placées dans le dossier <webapp>\WEB-INF\lib
`vues, ressources (.jsp, .html, ...)`	elles seront placées dans le dossier <webapp> ou des sous-dossiers mais en général en-dehors du sous-dossier WEB-INF

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

Construisons l'application web dont l'arborescence est 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 :

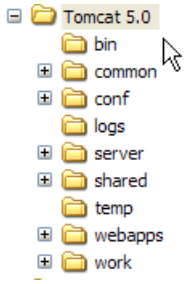
```
<html>
```

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

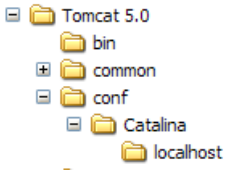
Si on charge ce fichier dans un navigateur, on obtient la page suivante :

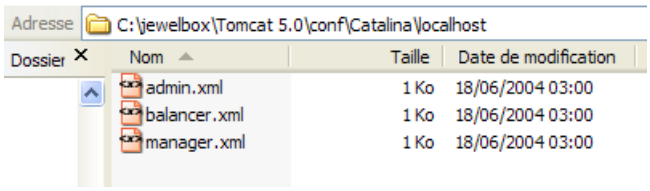
L'URL affichée par le navigateur montre que la page n'a pas été délivrée par un serveur web mais directement chargée par le navigateur. Nous voulons maintenant qu'elle soit disponible via le serveur web Tomcat.

Revenons sur l'arborescence du répertoire <tomcat> :

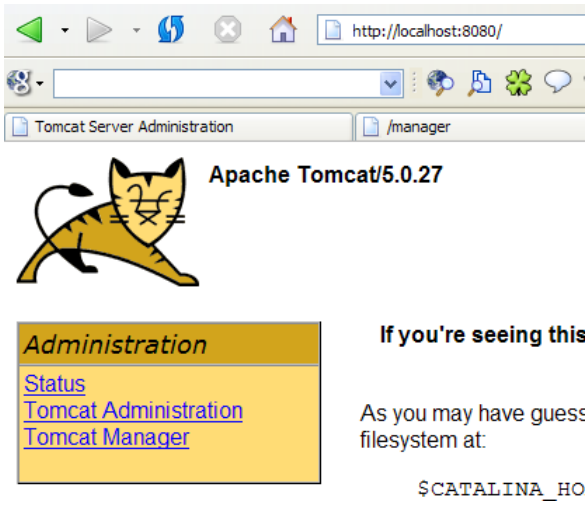


La configuration des applications web déployées au sein du serveur Tomcat se fait à l'aide de fichiers XML placés dans le dossier <tomcat>\conf\Catalina\localhost :

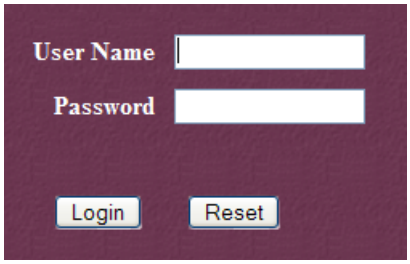


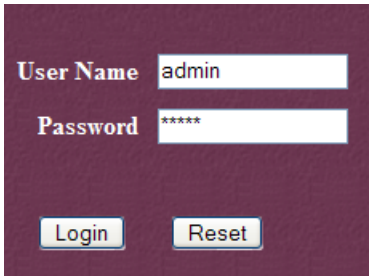


Ces fichiers XML peuvent être créés à la main car leur structure est simple. Plutôt que d'adopter cette démarche, nous allons utiliser les outils web que nous offre Tomcat. Sur sa page d'entrée http://localhost:8080, le serveur nous offre des liens pour l'administrer :

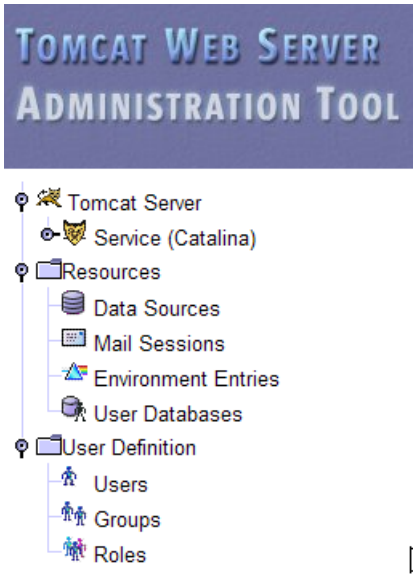


Le lien [Tomcat Administration] nous offre la possibilité de configurer les ressources mises par Tomcat à la disposition des applications web déployée en son sein. Un exemple classique est un pool de connexions à une base de données. Suivons le lien. Nous obtenons une page d'identification :





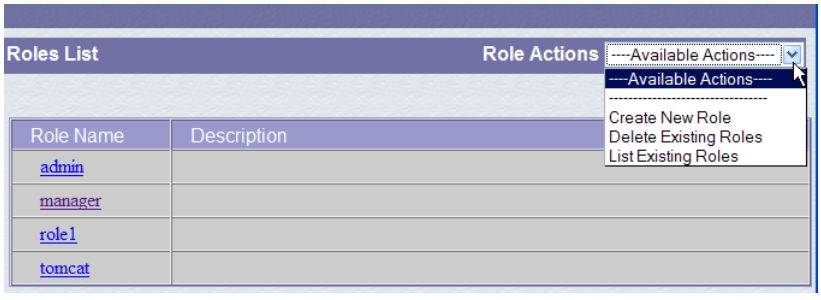
Ici, il faut redonner les informations que nous avons données au cours de l'installation de Tomcat. Dans notre cas, nous donnons le couple admin/admin. Le bouton [Login] nous amène à la page suivante :



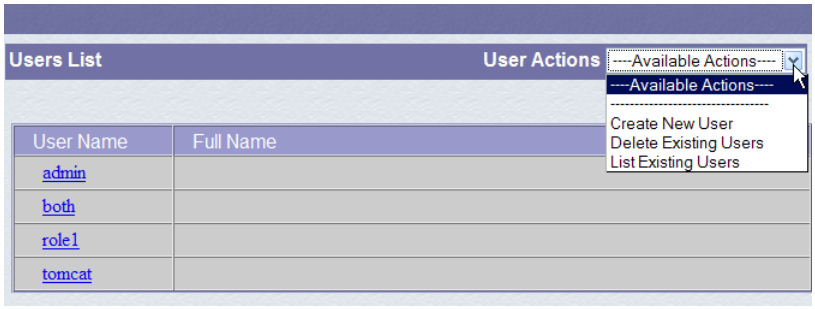
Cette page permet à l'administrateur de Tomcat de définir

- 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/administrateurs de 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)

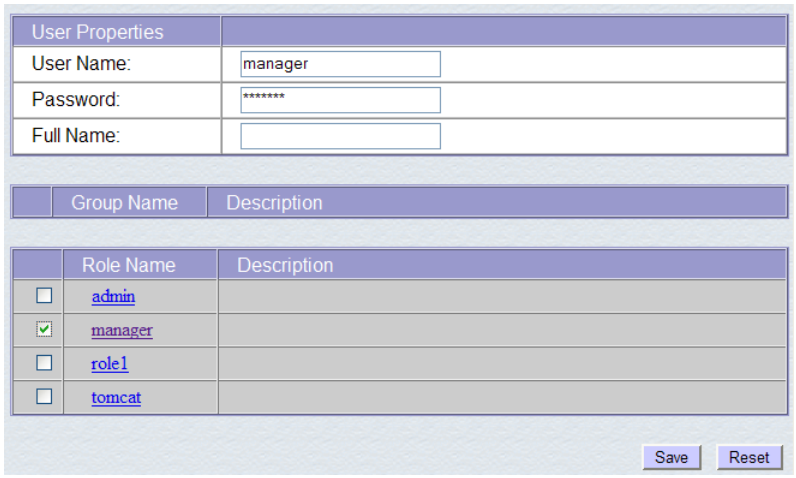
Suivons le lien [Roles] ci-dessus :



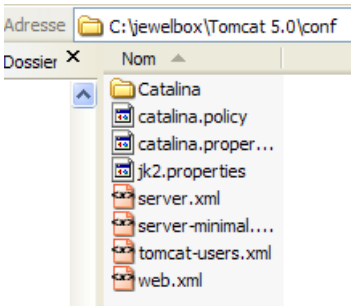
Un rôle permet de définir ce que peut faire ou ne pas faire un utilisateur ou un groupe d'utilisateurs. On associe un rôle à certains droits. Chaque utilisateur est associé à un ou plusieurs rôles et dispose des droits de ceux-ci. Le rôle [manager] ci-dessus donne le droit de gérer les applications web 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 :

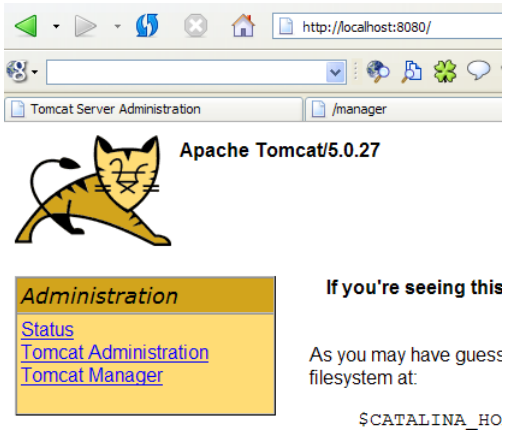


dont le contenu est le suivant :

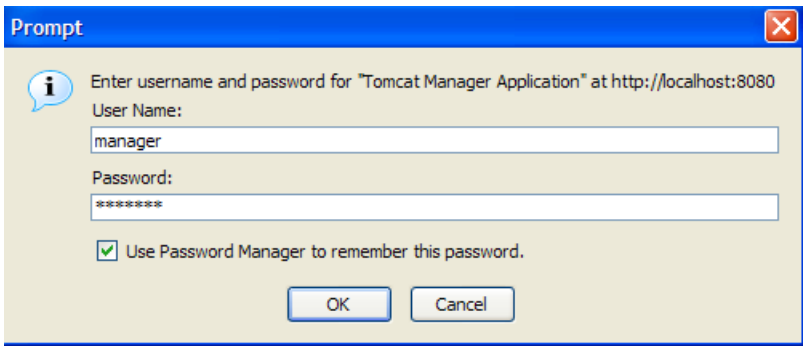
```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user 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**.

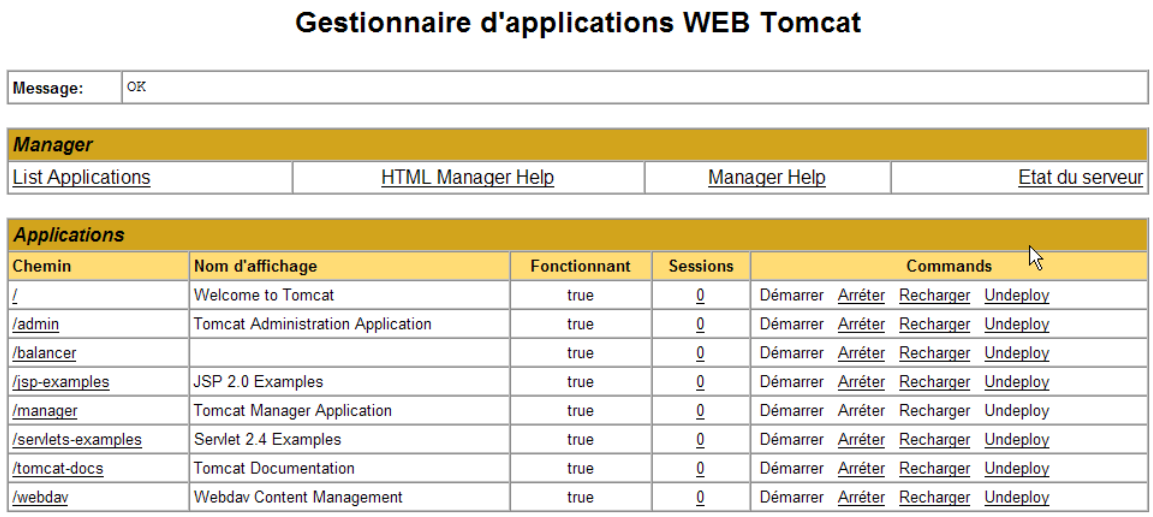
Revenons maintenant à la page d'entrée [http://localhost:8080] et suivons le lien [Tomcat Manager] :



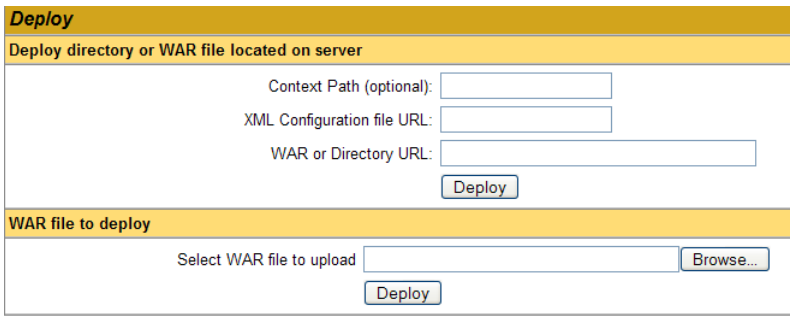
Nous obtenons alors une page d'authentification. Nous nous identifions comme manager/manager, c.a.d. l'utilisateur de rôle [manager] que nous venons de créer. En effet, seul un utilisateur ayant ce rôle peut utiliser ce lien :



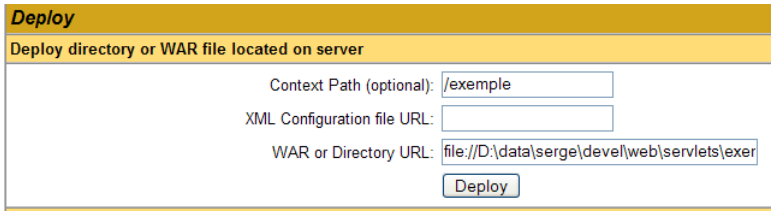
Nous obtenons une page listant les applications actuellement déployées dans Tomcat :



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

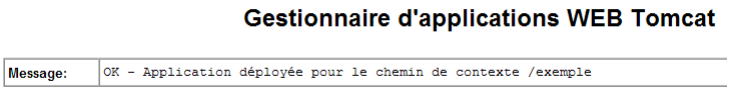


Ici, nous voulons déployer au sein de Tomcat, l'application **exemple** que nous avons construite précédemment. Nous le faisons de la façon suivante :

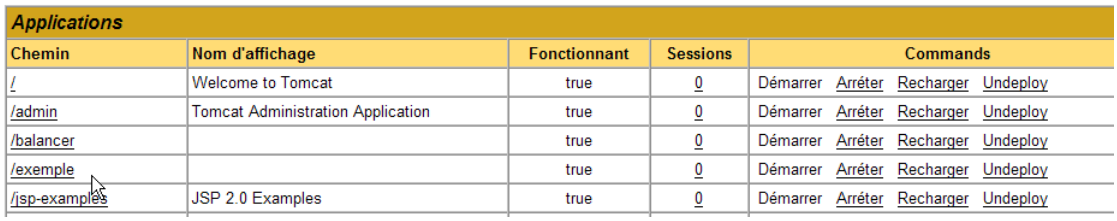


`Context Path`	/exemple	le nom utilisé pour désigner l'application web à déployer
`Directory URL`	file:// D:\data\serge\devel\web\servlets\exemple	le dossier de l'application web

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 :



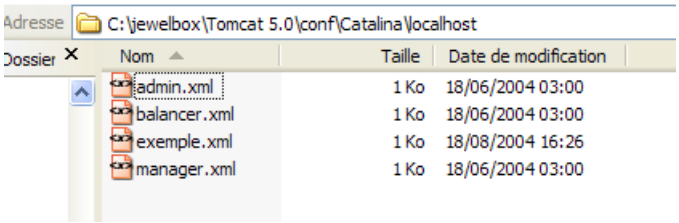
et la nouvelle application apparaît dans la liste des applications déployées :



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 :

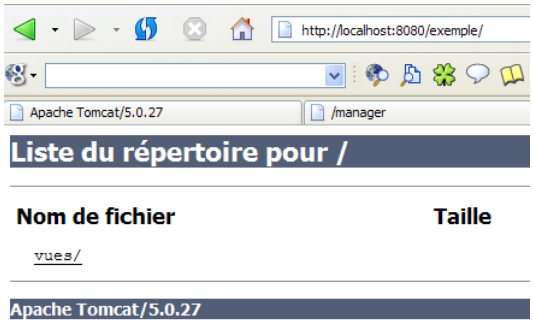
```
<?xml version='1.0' encoding='utf-8'?>
<Context docBase="D:/data/serge/devel/web/servlets/exemple" path="/exemple">
</Context>
```

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

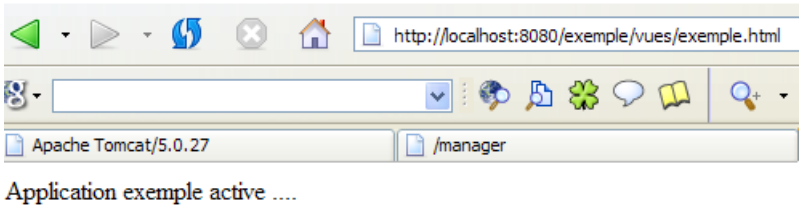
1. écrire à la main le descripteur XML de l'application à déployer
2. mettre ce descripteur dans le dossier <tomcat>\conf\Catalina\localhost
3. arrêter et relancer Tomcat

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/ :

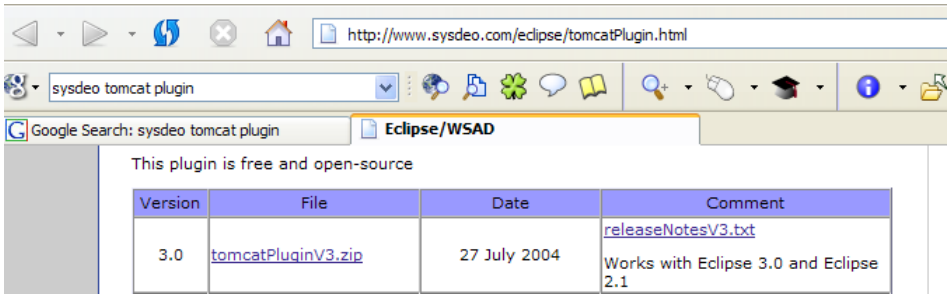


Nous obtenons une liste du contenu du dossier associé au contexte /exemple. Cependant le dossier WEB-INF d'une application web n'est lui jamais présenté. Nous pouvons demander la page exemple.html via l'url http://localhost:8080/exemple/vues/exemple.html :

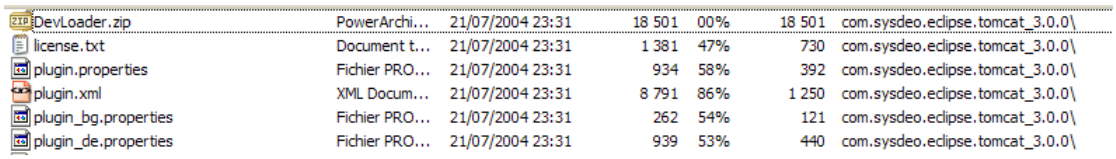


I.5 Installation de plugins Eclipse

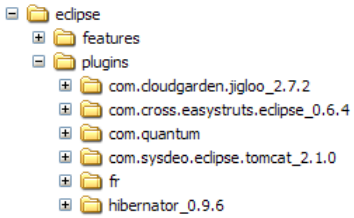
Nous allons avoir besoin dans la suite de plugins pour l'IDE Eclipse, notamment le plugin qui va permettre de gérer Tomcat depuis Eclipse. Nous présentons ici une méthode possible d'installation d'un plugin Eclipse. Nous téléchargeons le plugin Tomcat de Sysdeo :



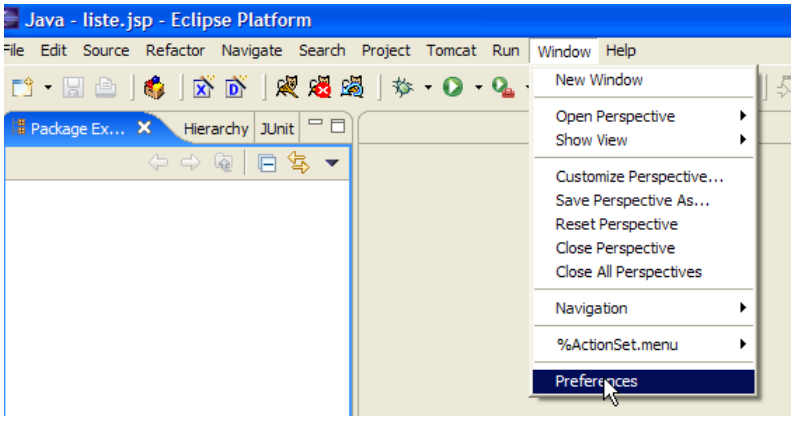
Nous obtenons un zip dont la structure (partielle) est la suivante :

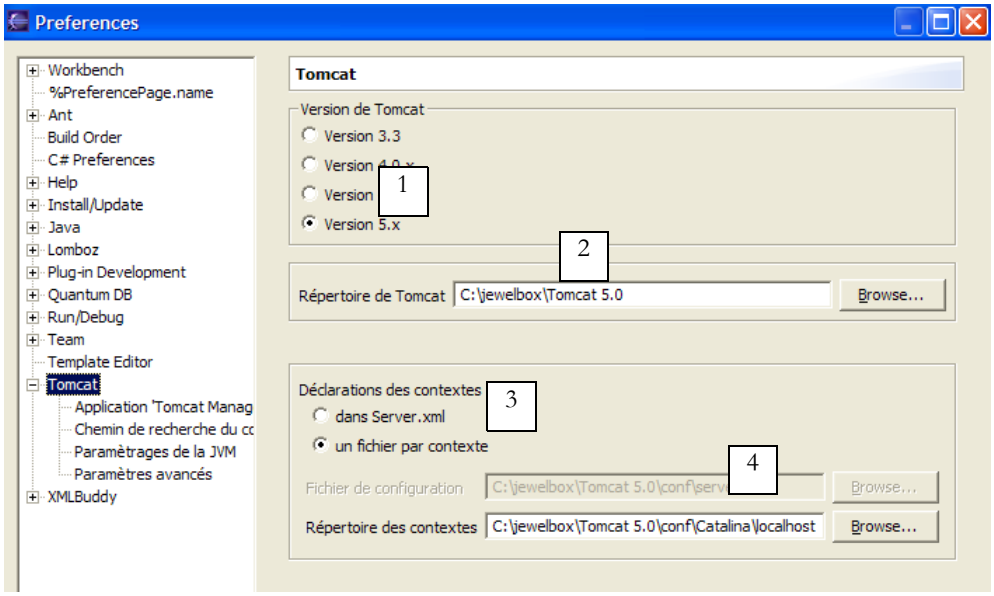


Le contenu du fichier zip est installé dans [<eclipse>\plugins] où <eclipse> est le dossier d'installation d'Eclipse :



Ceci fait, nous pouvons lancer Eclipse qui va alors intégrer la présence d'un nouveau plugin. Avant de pouvoir utiliser le plugin Tomcat il nous faut le configurer. Cela se fait par l'option de menu [Window/Preferences] :





On définit :

1. la version de Tomcat installée sur le poste
2. le dossier où il a été installé
3. le mode de déclaration des contextes des applications web
4. le répertoire des contextes

Les plugins utiles à installer pour le développement web en Java sont les suivants :

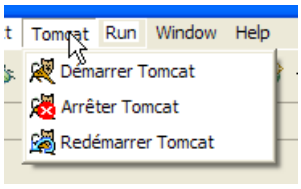
- XmlBuddy pour la gestion des documents Xml (http://xmlbuddy.com/)
- Lomboz pour la gestion des documents JSP (http://www.objectlearn.com/)

I.6 Lancer-Arrêter Tomcat depuis Eclipse

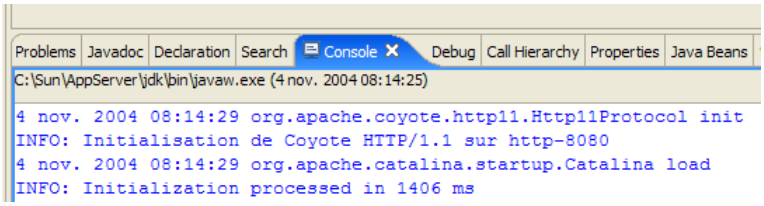
Pour gérer Tomcat depuis Eclipse, nous disposons de trois boutons dans la barre d'outils :

De gauche à droite : Lancer Tomcat - Arrêter Tomcat - Relancer Tomcat (Arrêt+Redémarrage)

On dispose également d'un menu :



Lançons Tomcat. Le serveur est lancé et écrit des logs dans la fenêtre [Console] :



La compréhension de ces logs demande une certaine habitude. Nous ne nous apesantirons pas dessus pour le moment. Il est cependant important de regarder que ces logs ne signalent pas d'erreurs de chargement de contextes. En effet, lorsqu'il est lancé, Tomcat va chercher à charger le contexte des applications dont il trouve les définitions soit dans le fichier [<tomcat>\conf\server.xml] soit dans le dossier [<tomcat>\conf\Catalina\localhost]. Charger le contexte d'une application implique d'exploiter le fichier [web.xml] de l'application et de charger une ou plusieurs classes initialisant celle-ci. Plusieurs types d'erreurs peuvent alors se produire :

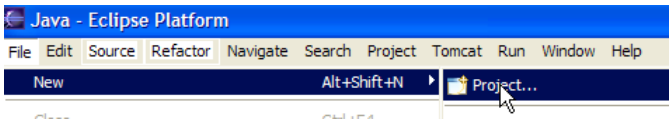
- le fichier [web.xml] est syntaxiquement incorrect. C'est l'erreur la plus fréquente. Il est conseillé d'utiliser un outil capable de vérifier la validité d'un document XML lors de sa construction. Ce sera le rôle du plugin XmlBuddy que nous avons adjoint à Eclipse.
- certaines classes à charger n'ont pas été trouvées. Elles sont cherchées dans [WEB-INF/classes] et [WEB-INF/lib]. Il faut en général vérifier la présence des classes nécessaires et l'orthographe de celles déclarées dans le fichier [web.xml].

Pour vérifier la présence de Tomcat, prenez un navigateur et demandez l'url [http://localhost:8080] :

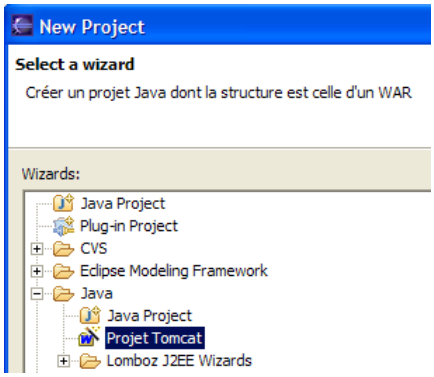
I.7 Développement d'une application web avec Eclipse/Tomcat

I.7.1 Création du contexte de l'application

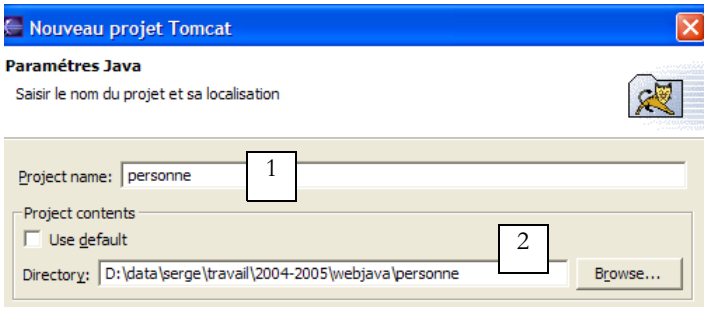
Nous sommes maintenant prêts à développer une première application web avec Eclipse/Tomcat. Nous reprenons une démarche analogue à celle utilisée pour créer une application web sans Eclipse. Eclipse lancé, nous créons un nouveau projet :



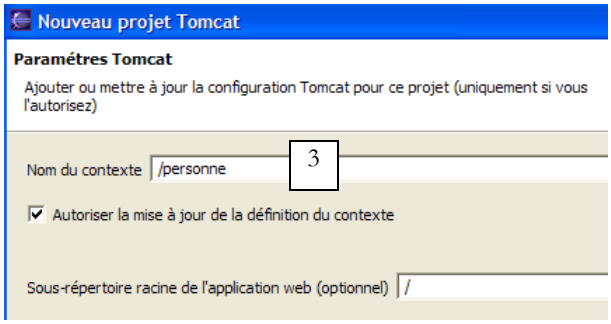
que nous définissons comme un projet Tomcat :



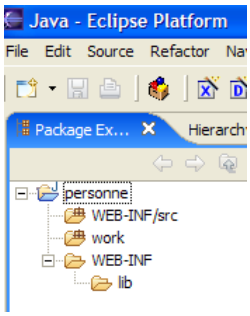
La première page de l'assistant de création nous précisons le nom du projet [1] et son emplacement [2] :



La seconde page de l'assistant nous demande de définir le contexte [3] de l'application :



Une fois l'assistant validé, Eclipse crée le projet Tomcat avec l'arborescence suivante :

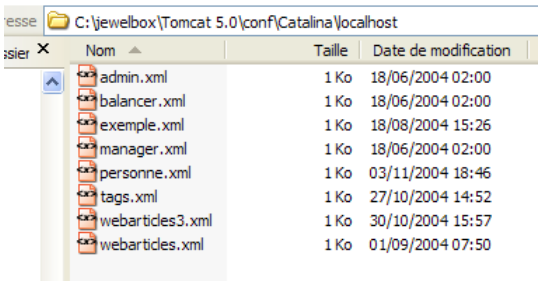


WEB-INF/src : contiendra le code Java des classes de l'application

WEB-INF/classes (non représenté) : contiendra les .class des classes compilées ainsi qu'une copie de tous les fichiers autres que .java placés dans WEB-INF/src. Une application web utilise fréquemment des fichiers dits "ressource" qui doivent être dans la même arborescence que les classes, c.a.d. dans WEB-INF/classes. On les place alors dans le dossier WEB-INF/src sachant qu'Eclipse les recopiera automatiquement dans WEB-INF/classes.

WEB-INF/lib : contiendra les archives .jar dont a besoin l'application web.
work : contiendra le code .java des servlets générées à partir des pages JSP de l'application.

Après la création du projet Tomcat, nous pouvons constater qu'un nouveau fichier [personne.xml] est apparu dans le dossier dans [<tomcat>\conf\Catalina\localhost] :



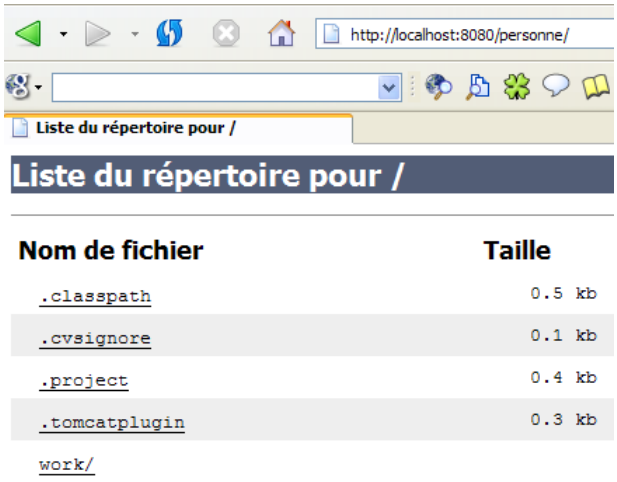
Son contenu est le suivant :

```
<Context path="/personne" reloadable="true" docBase="D:\data\serge\travail\2004-2005\webjava\personne"
workDir="D:\data\serge\travail\2004-2005\webjava\personne\work" />
```

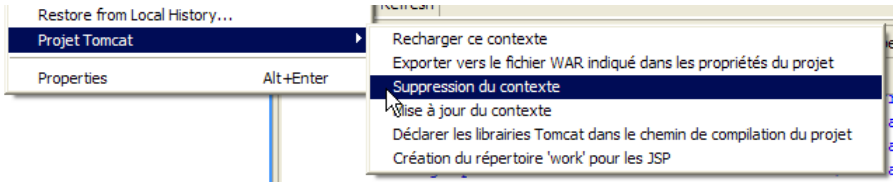
Si nous lançons Tomcat et examinons ses logs dans la fenêtre [Console] d'Eclipse, nous constatons que la nouvelle application a été prise en compte :

```
INFO: Processing Context configuration file URL file:C:\jewelbox\Tomcat 5.0\conf\Catalina\localhost\personne.xml
4 nov. 2004 08:14:37 org.apache.catalina.startup.ContextConfig applicationConfig
```

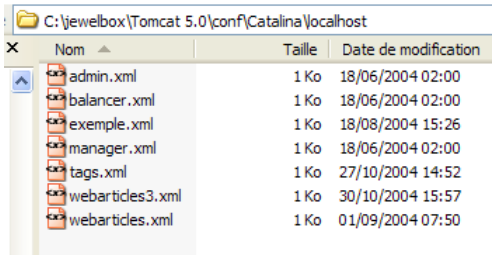
Nous pouvons le vérifier. Demandons le contexte /personne via l'url **http://localhost:8080/personne** :



Le plugin [Tomcat] nous permet de supprimer un contexte au sein du serveur Tomcat. Pour cela, dans Eclipse cliquez droit sur le projet et prenez l'option [Projet Tomcat/Suppression du contexte] :



Nous pouvons constater que le fichier [<tomcat>\conf\Catalina\localhost\personne.xml] a disparu :

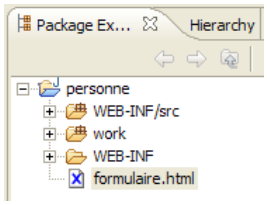


Pour recréer le contexte, on prendra l'option [Projet Tomcat/Mise à jour du contexte]. Le fichier [personne.xml] réapparaît alors dans le dossier [<tomcat>\conf\Catalina\localhost] et les logs de Tomcat indiquent que le chargement du nouveau contexte a eu lieu :

```
4 nov. 2004 09:08:42 org.apache.catalina.core.StandardHostDeployer install
INFO: Processing Context configuration file URL file:C:\jewelbox\Tomcat 5.0\conf\Catalina\localhost\personne.xml
4 nov. 2004 09:08:42 org.apache.catalina.startup.ContextConfig applicationConfig
INFO: Le fichier web.xml de l'application est absent, utilisation des paramètres par défaut StandardEngine[Catalina].Star
```

I.7.2 Création d'un document HTML

Nous créons maintenant un document HTML statique [formulaire.html] dans le dossier [personne] :

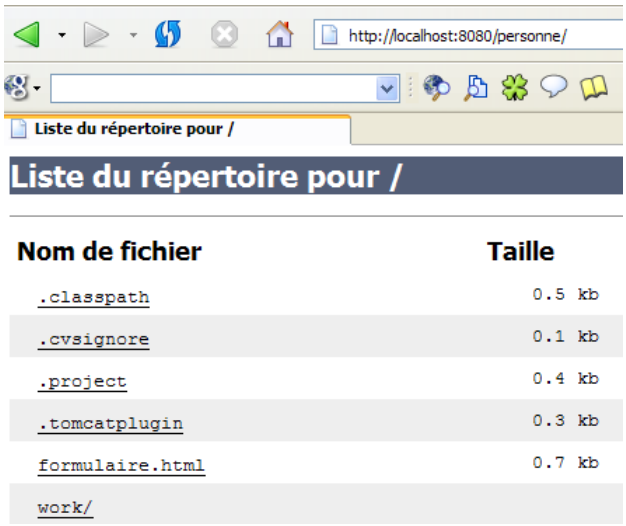


Pour le créer, cliquez droit sur le projet [personne] puis prenez l'option [New/File] et appelez [formulaire.html] le nouveau fichier. Ce document HTML affichera un formulaire demandant le nom et l'âge d'une personne. Son contenu sera le suivant :

```
<html>
  <head>
    <title>Personne - formulaire</title>
  </head>
  <body>
    <center>
      <h2>Personne - formulaire</h2>
```

```
      <hr>
      <form action="" method="post">
       <table>
          <tr>
            <td>Nom</td>
            <td><input name="txtNom" value="" type="text" size="20"></td>
          </tr>
          <tr>
            <td>Age</td>
            <td><input name="txtAge" value="" type="text" size="3"></td>
          </tr>
        </table>
        <table>
          <tr>
            <td><input type="submit" value="Envoyer"></td>
            <td><input type="reset" value="Retablir"></td>
            <td><input type="button" value="Effacer"></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

Sauvegardez ce document dans le dossier <personne>. Lancez Tomcat. Nous allons tout d'abord vérifier que le contexte [/personne] existe bien au sein de Tomcat. Avec un navigateur demandez l'URL **http://localhost:8080/personne** :



On obtient une page qui liste le contenu du dossier physique associé au contexte [/personne]. On y voit [formulaire.html] accessible via un lien. Le premier test à faire lors du déploiement d'une application web est de vérifier, comme nous l'avons fait ici, que le contexte de l'application est accessible. S'il ne l'est pas, il est inutile d'aller plus loin. Que faire si cette étape échoue ? Prenons le cas suivant où nous faisons une faute d'orthographe sur le nom du contexte :

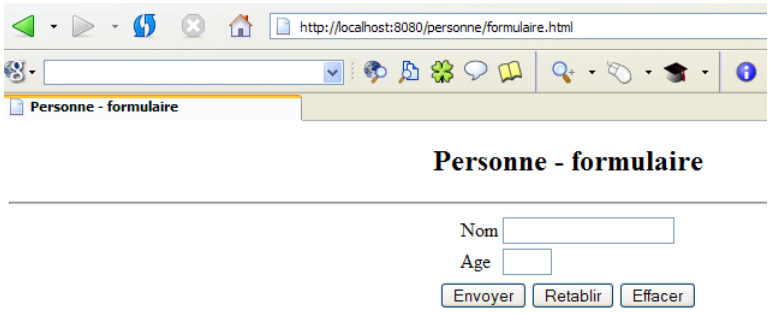
Lorsqu'un contexte est inaccessible, on vérifiera les points suivants :

- l'orthographe du contexte
- les logs de Tomcat dans Eclipse. Ils peuvent signaler une erreur lors du chargement du contexte de notre application. Ci-dessous, on voit les logs liés au traitement du fichier [personne.xml] :

```
INFO: Processing Context configuration file URL file:C:\jewelbox\Tomcat 5.0\conf\Catalina\localhost\personne.xml
4 nov. 2004 08:14:37 org.apache.catalina.startup.ContextConfig applicationConfig
```

- le contenu du fichier [web.xml] qui définit l'application web.

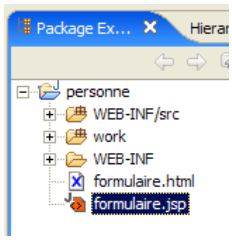
Une fois la page du contexte /personne obtenu, nous pouvons suivre le lien [formulaire.html] :



I.8 Une page JSP

Lectures : chapitre 1, chapitre 2 : 2.2, 2.2.1, 2.2.2, 2.2.3, 2.2.4

Nous dupliquons notre document **formulaire.html** dans un fichier **formulaire.jsp** dans le même dossier



puis nous transformons le texte de [formulaire.jsp] la façon suivante :

```
<%
  // on récupère les paramètres
  String nom=request.getParameter("txtNom");
  if(nom==null) nom="inconnu";
  String age=request.getParameter("txtAge");
  if(age==null) age="xxx";
%>

<html>
  <head>
    <title>Personne - formulaire</title>
  </head>
  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
       <form action="" method="post">
        <table>
          <tr>
            <td>Nom</td>
              <td><input name="txtNom" value="<%= nom %>" type="text" size="20"></td>
           </tr>
           <tr>
            <td>Age</td>
              <td><input name="txtAge" value="<%= age %>" type="text" size="3"></td>
           </tr>
        </table>
        <table>
         <tr>
           <td><input type="submit" value="Envoyer"></td>
             <td><input type="reset" value="Rétablir"></td>
             <td><input type="button" value="Effacer"></td>
          </tr>
        </table>
      </form>
    </center>
```

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

Le document initialement statique est maintenant devenu dynamique par introduction de code Java. Pour ce type de document, nous procèderons toujours comme suit :

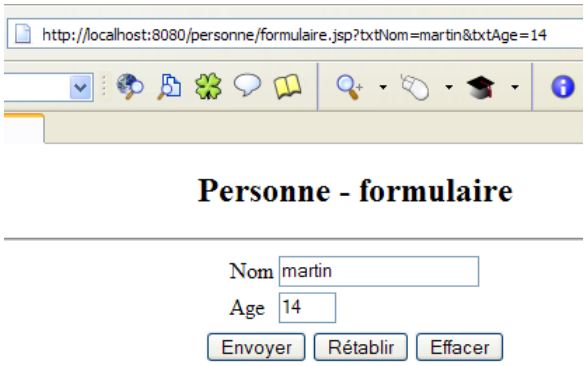
- nous mettons du code Java dès le début du document pour récupérer les paramètres nécessaires au document pour s'afficher. Ceux-ci seront le plus souvent dans l'objet **request**. Cet objet représente la requête du client. Celle-ci peut passer au travers de plusieurs servlets et pages JSP qui ont pu l'enrichir. Ici, elle nous arrivera directement du navigateur.
- le code HTML se trouve à la suite. Il se contentera le plus souvent d'afficher des variables calculées auparavant dans le code Java au moyen de balises **<%= variable %>**. On notera ici que le signe = est collé au signe %. C'est une cause fréquente d'erreurs.

Que fait le document dynamique précédent ?

- il récupère dans la requête, deux paramètres appelés [txtNom] et [txtAge]. S'il ne les trouve pas, il leur donne des valeurs par défaut.
- il affiche la valeur de ces deux paramètres dans le code HTML qui suit

Faisons un premier test. Avec un navigateur, demandons l'URL **http://localhost:8080/personne/formulaire.jsp** :

Le document **formulaire.jsp** a été appelé sans passage de paramètres. Les valeurs par défaut ont donc été affichées. Maintenant demandons l'URL **http://localhost:8080/personne/formulaire.jsp?txtNom=martin&txtAge=14** :



Cette fois-ci, nous avons passé au document **formulaire.jsp**, les paramètres **txtNom** et **txtAge** qu'il attend. Il les a donc affichés. On sait qu'il y a deux méthodes pour passer des paramètres à un document web : GET et POST. Dans les deux cas, les paramètres passés se retrouvent dans l'objet prédéfini **request**. Ici, ils ont été passés par la méthode GET.

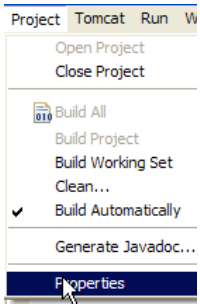
I.9 Une servlet

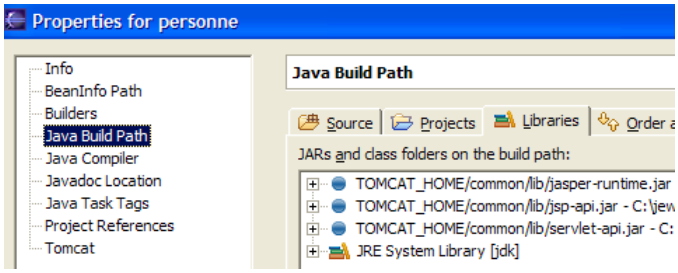
Lectures : chapitre 1, chapitre 2 : 2.1, 2.1.1, 2.1.2, 2.3.1

I.9.1 Configuration d'Eclipse

Nous nous proposons ici de créer une servlet faisant la même chose que la page JSP précédente. Une servlet est une classe Java. Nous l'appellerons **ServletFormulaire**. Une servlet dérive de la classe **javax.servlet.http.Servlet**. Les classes **javax.servlet.*** ne sont pas partie du Java standard et ne sont donc pas trouvées par défaut dans un projet Java sous Eclipse. Il nous faut configurer le projet afin que son [ClassPath] inclue les dossiers contenant les bibliothèques .jar des classes nécessaires au développement web. Sous Eclipse, cliquons droit sur le nom du projet [personne] et accédons à ses propriétés ou bien prenons le menu

[Project/Properties] puis l'option [Java Build Path] de ces propriétés. Cette option fixe les dossiers et bibliothèques à explorer pour trouver toutes les classes nécessaires à l'application.

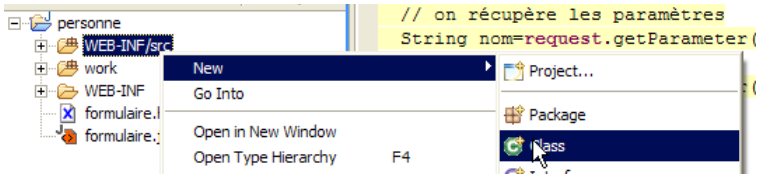




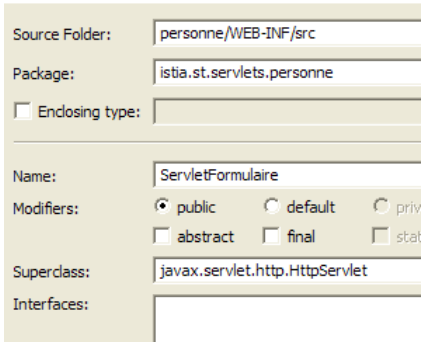
Nous constatons que dans le [Java Build Path] de l'application se trouvent des archives situées dans l'arborescence de Tomcat. Celles-ci contiennent les classes du paquetage [javax.servlet] dont nous avons besoin. C'est parce que nous créé un projet [Java/Tomcat] que ces archives ont été automatiquement ajoutées au [Java Build Path] de l'application.

I.9.2 Écriture de la servlet

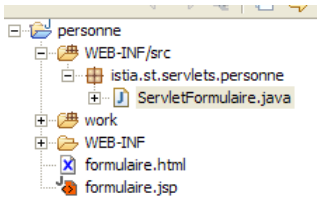
Nous sommes maintenant prêts pour écrire la classe **servletFormulaire**. Sous Eclipse, cliquons droit sur le dossier [WEB-INF/src] et prenons l'option de création d'une classe :



puis définissons les caractéristiques de la classe à créeer :



Vous adapterez le nom du paquetage en remplaçant [st] par votre nom. Après validation de l'assistant, le projet est modifié de la façon suivante :



La classe [ServletFormulaire] a été créée avec un squelette de code :

```
package istia.st.servlets.personne;

import javax.servlet.http.HttpServlet;

public class ServletFormulaire extends HttpServlet {

}
```

Nous complétons ce code avec le contenu suivant :

```
package istia.st.servlets.personne;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletFormulaire
  extends HttpServlet {

  // paramètres d'instance
  private String defaultNom = null;
  private String defaultAge = null;

  //init
  public void init() {
    // on récupère les paramètres d'initialisation de la servlet
    ServletConfig config = getServletConfig();
    defaultNom = config.getInitParameter("defaultNom");
    if(defaultNom==null) defaultNom="NNNNNNNNNNNNNNN";
    defaultAge = config.getInitParameter("defaultAge");
    if(defaultAge==null) defaultAge="AAA";
  }

  //GET
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {

    // on récupère les paramètres du formulaire
    String nom = request.getParameter("txtNom");
    if (nom == null) {
      nom = defaultNom;
    }
    String age = request.getParameter("txtAge");
    if (age == null) {
      age = defaultAge;
    }
    // on affiche le formulaire
    response.setContentType("text/html");
    PrintWriter out=response.getWriter();
    out.println(
                "<html>"+
                  "<head>"+
                    "<title>Personne - formulaire</title>"+
                  "</head>"+
                  "<body>"+
                    "<center>"+
                      "<h2>Personne - formulaire</h2>"+
                      "<hr>"+
                      "<form action='' method='post'>"+
                        "<table>"+
                          "<tr>"+
                            "<td>Nom</td>"+
                            "<td><input name='txtNom' value='"+nom+"' type='text' size='20'></td>"+
                          "</tr>"+
                          "<tr>"+
                            "<td>Age</td>"+
                            "<td><input name='txtAge' value='"+ age +"' type='text' size='3'></td>"+
                          "</tr>"+
                        "</table>"+
                        "<table>"+
                          "<tr>"+
                            "<td><input type='submit' value='Envoyer'></td>"+
                            "<td><input type='reset' value='Rétablir'></td>"+
                            "<td><input type='button' value='Effacer'></td>"+
                          "</tr>"+
                        "</table>"+
                      "</form>"+
                    "</center>"+
                  "</body>"+
                "</html>"
      );
  }

  /**
   * @param request la requête HTTP du client
   * @param response la réponse HTTP qu'on va construire
   */
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
    // on passe la main au GET
    doGet(request, response);
  }
}
```

A la simple lecture de la servlet, on peut constater tout de suite qu'elle est beaucoup plus complexe que la page JSP correspondante. C'est une généralité : une servlet n'est pas adaptée pour générer du code HTML. Ce sont les pages JSP qui sont faites pour cela. Nous aurons l'occasion d'y revenir. Explicitons quelques points importants de la servlet ci-dessus :

- lorsqu'une servlet est appelée pour la 1ère fois, sa méthode **init** est appelée. C'est le seul cas où elle est appelée.
- si la servlet a été appelée par la méthode HTTP GET, la méthode **doGet** est appelée pour traiter la requête du client.
- si la servlet a été appelée par la méthode HTTP POST, la méthode **doPost** est appelée pour traiter la requête du client.

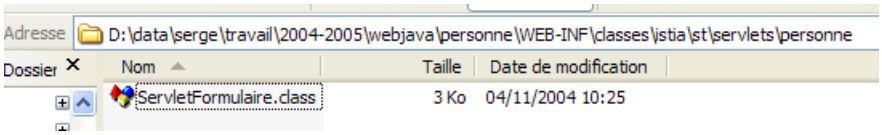
La méthode **init** sert ici à récupérer des valeurs appelées **defaultNom** et **defaultAge**. Pour l'instant, on ne sait pas où elle les récupère. Ce sont des valeurs de configuration de la servlet qui ne changent pas au fil des cycles requête-réponse. La méthode **init** exécutée au chargement initial de la servlet est donc le bon endroit pour les récupérer.

La méthode **doPost** renvoie à la méthode **doGet**. Cela signifie que le client pourra envoyer indifféremment ses paramètres par un POST ou un GET.

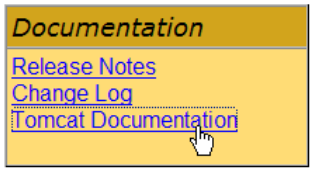
La méthode **doGet** :

- reçoit deux paramètres **request** et **response**. **request** est un objet représentant la totalité de la requête du client. Elle est de type **HttpServletRequest** qui est une interface. **response** est de type **HttpServletResponse** qui est également une interface. L'objet **response** sert à envoyer une réponse au client.
- **request.getparameter("param")** sert à récupérer dans la requête du client la valeur du paramètre de nom **param**.
- **response.getWriter()** sert à obtenir un flux d'écriture vers le client
- **response.setContentType(String)** sert à fixer la valeur de l'entête HTTP **Content-type**. On rappelle que cet entête indique au client la nature du document qu'il va recevoir. Le type **text/html** indique un document HTML.

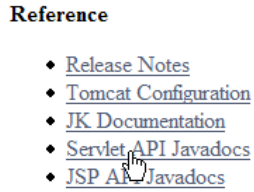
La compilation de cette servlet va produire un fichier .class dans le dossier [WEB-INF/classes] :



Le lecteur est invité à consulter l'aide Java sur les servlets. On pourra pour cela s'aider de Tomcat. Sur la page d'entrée de Tomcat 5, on a un lien [Documentation] :



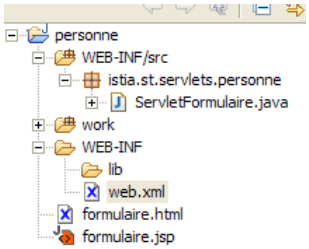
Ce lien mène à une page que le lecteur est invité à explorer. Le lien sur la documentation des servlets est le suivant :



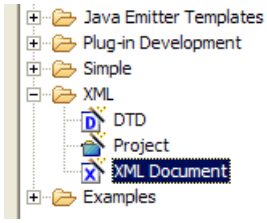
I.9.3 Déploiement de la servlet dans Tomcat

Lectures : chapitre 2 : 2.3, 2.3.1, 2.3.2, 2.3.3, 2.3.4

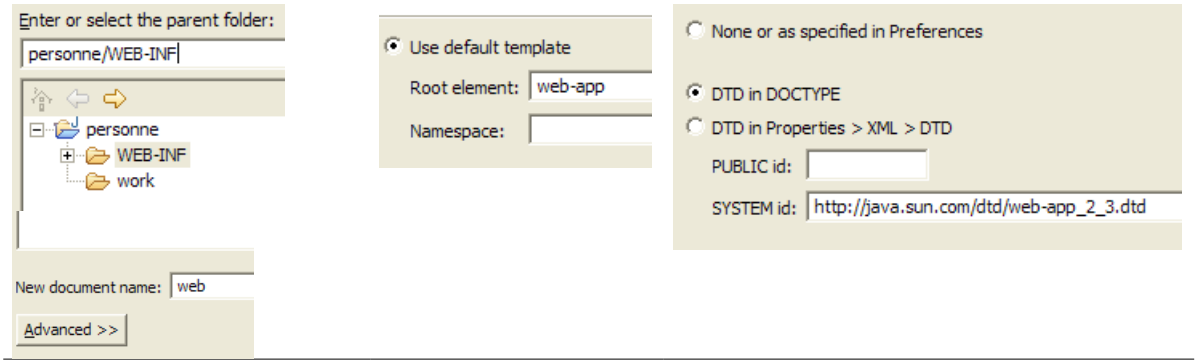
Une fois la servlet précédente correctement compilée par Eclipse, il faut la faire prendre en compte par le serveur Tomcat. Pour cela, il nous faut écrire le fichier de configuration [web.xml] de l'application. Ce fichier est à placer dans le dossier **WEB-INF** de l'application :

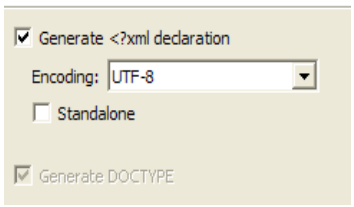


Pour le construire, nous allons utiliser un assistant. Nous cliquons droit sur le dossier [WEB-INF] et prenons l'option [New/Other] pour arriver à la première page d'un assistant :



Nous prenons l'option [XML/XML Document] puis les options suivantes :



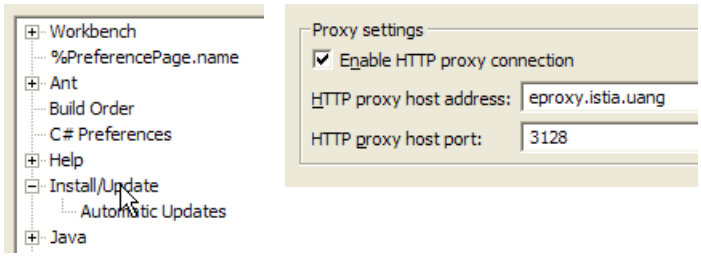


La plupart des options prises ci-dessus sont les options proposées par défaut. Le fichier [web.xml] généré par l'assistant est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

Le fichier reprend ce qui a été positionné dans l'assistant. Aussi la plupart du temps, peut-on se contenter de créer un fichier [web.xml] par copier/coller. L'icône associée au fichier [web.xml] indique que son contenu est pris en charge par le plugin [XmlBuddy] :

Le contenu d'un fichier XML est le plus souvent contrôlé par un document appelé une DTD (Document Type Definition) qui définit les balises que peut contenir le document ainsi que la hiérarchie de celles-ci à l'intérieur du document. Ici la DTD utilisée se trouve à l'url [http://java.sun.com/dtd/web-app_2_3.dtd] comme l'indique l'entête [DOCTYPE] du fichier [web.xml] précédent. Afin que XMlBuddy ait accès à cette DTD, le poste doit avoir accès au réseau. Dans les salles de l'ISTIA, les postes n'ont un accès HTTP au réseau Internet que via une machine proxy de nom [eproxy.istia.uang] et de port [3128]. Il faut s'assurer que le proxy d'Eclipse est correctement positionné. Pour cela, on utilise l'option [Window/Preferences/Install-Update] :

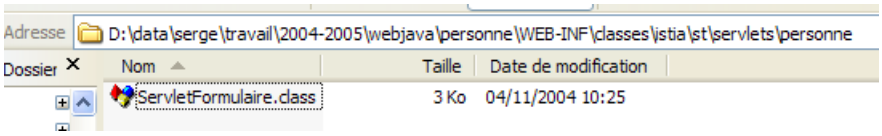


Ceci fait, XmlBuddy va pouvoir signaler les erreurs trouvées dans [web.xml]. C'est une aide indispensable car il est facile de faire des erreurs telles que celle d'oublier de fermer une balise. Le fichier **web.xml** de notre application **personne** sera le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>formulairepersonne</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletFormulaire</servlet-class>
    <init-param>
      <param-name>defaultNom</param-name>
      <param-value>inconnu</param-value>
    </init-param>
    <init-param>
      <param-name>defaultAge</param-name>
      <param-value>XXX</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>formulairepersonne</servlet-name>
    <url-pattern>/formulaire</url-pattern>
  </servlet-mapping>
</web-app>
```

Les points principaux de ce fichier de configuration sont les suivants :

- ce qui précède la balise <web-app> est à reprendre intégralement dans tout fichier de configuration
- la configuration se fait entre les balises **<web-app>** et **</web-app>**
- la configuration d'une servlet se fait entre les balises <**servlet**> et </**servlet**>. Une application peut comporter plusieurs servlets et donc autant de sections de configuration <**servlet**>...</**servlet**>.
- la balise <**servlet-name**> fixe un nom à la servlet - peut être quelconque
- la balise <**servlet-class**> donne le nom complet de la classe correspondant à la servlet. Vous donnerez le nom exact de votre classe qui sera sans doute différent de celui déclaré ci-dessus. Tomcat ira chercher cette classe dans le dossier [WEB-INF/classes] de l'application :



- la balise <**init-param**> sert à passer des paramètres de configuration à la servlet. Ceux-ci sont généralement lus dans la méthode **init** de la servlet car les paramètres de configuration de celle-ci doivent être connus dès son premier chargement.
- la balise <**param-name**> fixe le nom du paramètre et <**param-value**> sa valeur.
- la balise <**servlet-mapping**> sert à associer une servlet (**servlet-name**) à un modèle d'URL (**url-pattern**). Ici le modèle est simple. Il dit qu'à chaque fois qu'une URL aura la forme /**formulaire**, il faudra utiliser la servlet **formulairepersonne**, c.a.d. la classe [istia.st.servlets.ServletFormulaire].

I.9.4 Test de la servlet

Nous en savons assez pour faire un test. Lançons le serveur Tomcat si besoin est. S'il était déjà lancé, les modifications faites au fichier [web.xml] ont du l'amener à recharger le contexte de l'application. Ce point est affiché dans les logs :

```
4 nov. 2004 10:57:14 org.apache.catalina.core.StandardContext reload
INFO: Le rechargement de ce contexte a démarré
4 nov. 2004 10:59:56 org.apache.catalina.startup.HostConfig restartContext
INFO: restartContext(/personne)
```

Demandons l'URL [http://localhost:8080/personne/formulaire] avec un navigateur. Nous demandons ici la ressource [/formulaire] du contexte [/personne]. Le fichier [web.xml] de ce contexte indique que la ressource [/formulaire] est assurée par la servlet de nom [/formulairePersonne]. Dans le même fichier, il est indiqué que cette servlet est la classe [istia.st.servlets.ServletFormulaire]. C'est donc à cette classe que Tomcat va confier le traitement de la requête client. Si la classe n'était pas déjà chargée, elle le sera. Elle restera ensuite en mémoire pour les futures requêtes.

L'url [http://localhost:8080/personne/formulaire] donne le résultat suivant :

Nous obtenons les valeurs par défaut du nom et de l'âge. Demandons maintenant l'URL [http://localhost:8080/personne/formulaire?txtNom=tintin&txtAge=30] :

Nous obtenons bien les paramètres passés dans la requête.

I.10 Coopération servlets et pages JSP

Lectures : chapitre 2 : 2.3.7

I.10.1 La servlet

Nous avons vu que la servlet était mal adaptée pour générer du code HTML alors que la page JSP elle, était bien adaptée. Par la suite, nous construirons nos applications de la façon suivante :

- les pages renvoyées en réponses aux demandes des clients seront générées uniquement par des documents JSP. Ceux-ci seront paramétrés, ce qui leur donnera leur aspect dynamique.
- la logique de traitement des requêtes du client et le calcul des paramètres nécessaires à l'affichage des réponses sera assuré par une ou plusieurs servlets.

Pour exemple, reprenons celui de la servlet précédente [ServletFormulaire] et enlevons-lui toute la génération du code HTML de la réponse. Appelons cette nouvelle servlet **ServletFormulaire2**. Elle sera construite dans le même projet [personne] que précédemment ainsi que toutes les servlets à venir. Son code est le suivant :

```
package istia.st.servlets.personne;

import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletFormulaire2 extends HttpServlet {
  // paramètres d'instance
  private String defaultNom = null;
  private String defaultAge = null;

  //init
  public void init() {
    // on récupère les paramètres d'initialisation de la servlet
    ServletConfig config = getServletConfig();
    defaultNom = config.getInitParameter("defaultNom");
    if(defaultNom==null) defaultNom="NNNNNNNNNNNNNNN";
    defaultAge = config.getInitParameter("defaultAge");
```

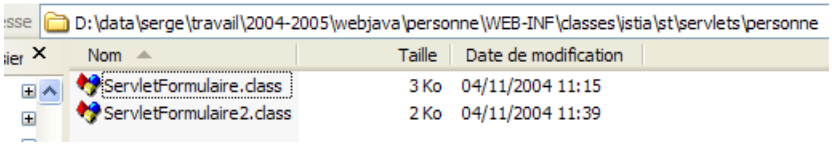
```
    if(defaultAge==null) defaultAge="AAA";
  }

  //GET
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException {

    // on récupère les paramètres du formulaire
    String nom = request.getParameter("txtNom");
    if (nom == null) {
      nom = defaultNom;
    }
    String age = request.getParameter("txtAge");
    if (age == null) {
      age = defaultAge;
    }
    // on affiche le formulaire
    request.setAttribute("nom",nom);
    request.setAttribute("age",age);
    getServletContext().getRequestDispatcher("/formulaire2.jsp").forward(request,response);
  } //GET

  //POST
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException {
    // on passe la main au GET
    doGet(request, response);
  }

}
```

Seule la partie génération de la réponse a changé :

```
    // on affiche le formulaire
    request.setAttribute("nom",nom);
    request.setAttribute("age",age);
    getServletContext().getRequestDispatcher("/formulaire2.jsp").forward(request,response);
```

La réponse est confiée à la page JSP **formulaire2.jsp**. Celle-ci a deux paramètres **nom** et **age**. La servlet met ces deux valeurs dans la requête à l'aide de la méthode **setAttribute**, requête qu'elle passe ensuite à la page **formulaire2.jsp**.

Construire la classe [ServletFormulaire2] en suivant le même cheminement que pour la classe [ServletFormulaire]. Une fois la compilation de la classe réussie, une nouvelle classe apparaît dans [WEB-INF/classes] :



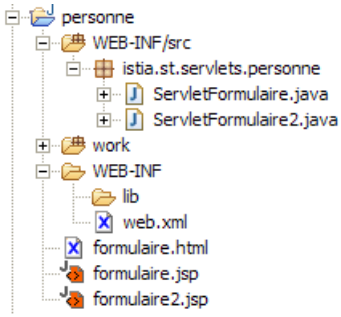
I.10.2 La page JSP

La page JSP **formulaire2.jsp** est la suivante :

```
<%
  // on récupère les valeurs nécessaire à l'affichage
  String nom=(String)request.getAttribute("nom");
  String age=(String)request.getAttribute("age");
%>

<html>
  <head>
    <title>Personne - formulaire</title>
  </head>
  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
       <form action="" method="post">
        <table>
          <tr>
            <td>Nom</td>
              <td><input name="txtNom" value="<%= nom %>" type="text" size="20"></td>
           </tr>
           <tr>
            <td>Age</td>
              <td><input name="txtAge" value="<%= age %>" type="text" size="3"></td>
           </tr>
```

```
        </table>
        <table>
         <tr>
           <td><input type="submit" value="Envoyer"></td>
            <td><input type="reset" value="Rétablir"></td>
            <td><input type="button" value="Effacer"></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

On voit que la page JSP commence par récupérer les deux attributs dont elle a besoin pour s'afficher : **nom** et **age**. Une fois qu'elle les a, la réponse au client est envoyée. Construisez cette page JSP et mettez là sous le dossier <personne> :



I.10.3 Déploiement de l'application

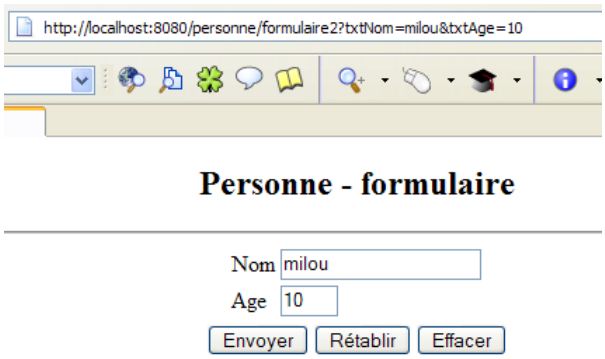
Le fichier de configuration [web.xml] est modifié de la façon suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app SYSTEM "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>formulairepersonne</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletFormulaire</servlet-class>
    <init-param>
      <param-name>defaultNom</param-name>
       <param-value>inconnu</param-value>
    </init-param>
    <init-param>
      <param-name>defaultAge</param-name>
      <param-value>XXX</param-value>
    </init-param>
  </servlet>

  <servlet>
    <servlet-name>formulairepersonne2</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletFormulaire2</servlet-class>
    <init-param>
      <param-name>defaultNom</param-name>
       <param-value>inconnu</param-value>
    </init-param>
    <init-param>
      <param-name>defaultAge</param-name>
       <param-value>XXX</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>formulairepersonne</servlet-name>
    <url-pattern>/formulaire</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>formulairepersonne2</servlet-name>
    <url-pattern>/formulaire2</url-pattern>
  </servlet-mapping>

</web-app>
```

Nous avons conservé l'existant et ajouté :

- une section <**servlet**> pour définir la nouvelle servlet **ServletFormulaire2**
- une section <**servlet-mapping**> pour lui associer l'URL /**formulaire2**

Lancez le serveur Tomcat si besoin est. S'il était déjà actif, l'ajout de nouvelles classes ainsi que la modification du fichier [web.xml] a du provoquer le rechargement du contexte <personne>. Nous sommes maintenant prêts pour les tests. Nous demandons l'URL **http://localhost:8080/personne/formulaire2?txtNom=milou&txtAge=10** :



Nous obtenons le même résultat que précédemment mais la structure de notre application est plus claire : une servlet qui contient de la logique applicative et délègue à une page JSP l'envoi de la réponse au client. Nous procèderons désormais toujours de cette façon.

I.11 Une application multi-pages

Lectures : expressions régulières dans polycopié "Introduction au langage Java"

I.11.1 Introduction

Nous allons construire une application autour du formulaire [nom, âge] précédent. L'application sera composée :

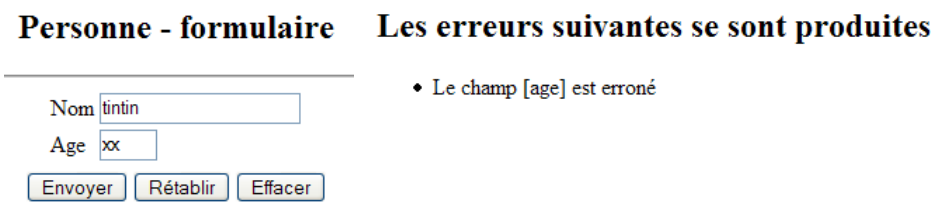
- d'une servlet **main1**. C'est elle qui assurera toute la logique de l'application.
- de trois pages JSP : **formulaire1.personne.jsp**, **reponse1.personne.jsp**, **erreurs1.personne.jsp**

Le fonctionnement de l'application est le suivant. Elle est accessible via l'URL **http://localhost:8080/personne/main1**. A cette url, on obtient un formulaire fourni par la page **formulaire1.personne.jsp** :

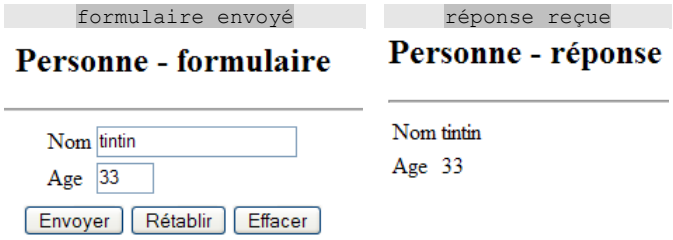
L'utilisateur remplit le formulaire et appuie sur le bouton [Envoyer] de type **submit**. Le bouton [Rétablir] est de type **reset**, c.a.d. qu'il remet le document dans l'état où il a été reçu. Le bouton [Effacer] est de type **button**. Nous verrons son rôle ultérieurement. L'utilisateur doit fournir 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 **reponse1.personne.jsp**. Voici des exemples :

formulaire envoyé	réponse reçue
Personne - formulaire Nom Age Envoyer Rétablir Effacer	**Les erreurs suivantes se sont produites** • Le champ [nom] n'a pas été rempli • Le champ [age] est erroné

formulaire envoyé	réponse reçue



Si l'utilisateur envoie des données correctes, l'application lui envoie une réponse au moyen de la page JSP **reponse1.personne.jsp**.



I.11.2 Configuration

Le fichier **web.xml** de l'application **/personne** est complété par les sections suivantes :

```
  <servlet>
    <servlet-name>personne1</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletPersonne1</servlet-class>
    <init-param>
      <param-name>urlMain</param-name>
       <param-value>/personne/main1</param-value>
    </init-param>
    <init-param>
      <param-name>urlReponse</param-name>
       <param-value>/reponse1.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreurs</param-name>
       <param-value>/erreurs1.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlFormulaire</param-name>
       <param-value>/formulaire1.personne.jsp</param-value>
    </init-param>
  </servlet>
....
  <servlet-mapping>
    <servlet-name>personne1</servlet-name>
    <url-pattern>/main1</url-pattern>
  </servlet-mapping>
```

Les deux sections <**servlet**> et <**servlet-mapping**> précédentes ne sont pas l'une à la suite de l'autre. La section <**servlet-mapping**> est avec les autres sections <**servlet-mapping**> derrière toutes les sections <**servlet**>.

Que dit ce fichier de configuration ?

- l'URL de la servlet et celles des trois pages JSP font l'objet chacune d'un paramètre de configuration. Cela permet de changer leurs noms sans avoir à recompiler l'application.
- l'application appelée **personne1** est accessible via l'URL /**main1** (servlet-mapping)

I.11.3 Les codes

La servlet principale est la classe **ServletPersonne1** suivante :

```
package istia.st.servlets.personne;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class ServletPersonne1
  extends HttpServlet {

  // paramètres d'instance
  String urlMain = null;
  String urlFormulaire = null;
  String urlReponse = null;
  String urlErreurs = null;
  ArrayList erreurs = new ArrayList();

  //init
  public void init() {
    // on récupère les paramètres d'initialisation de la servlet
    ServletConfig config = getServletConfig();
    urlMain = config.getInitParameter("urlMain");
    if (urlMain == null) {
      erreurs.add("Le paramètre [urlMain] n'a pas été initialisé");
    }
    urlFormulaire = config.getInitParameter("urlFormulaire");
    if (urlFormulaire == null) {
      erreurs.add("Le paramètre [urlFormulaire] n'a pas été initialisé");
    }
    urlReponse = config.getInitParameter("urlReponse");
    if (urlReponse == null) {
      erreurs.add("Le paramètre [urlReponse] n'a pas été initialisé");
    }
    urlErreurs = config.getInitParameter("urlErreurs");
  }

  /**
   * @param request requête du client
   * @param response réponse à construire
   */
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException {

    // on vérifie comment s'est passée l'initialisation de la servlet
    if(urlErreurs==null) throw new ServletException("Le paramètre [urlErreurs] n'a pas été initialisé");
    if(erreurs.size()!=0){
      // on passe la main à la page d'erreurs
      request.setAttribute("erreurs",erreurs);
      getServletContext().getRequestDispatcher(urlErreurs).forward(request,response);
      //fin
      return;
    }
    // on récupère les paramètres
    String nom=request.getParameter("txtNom");
    String age=request.getParameter("txtAge");
    // des paramètres ?
    if(nom==null || age==null){
      // on envoie le formulaire
      request.setAttribute("nom","");
      request.setAttribute("age","");
      request.setAttribute("urlAction",urlMain);
      getServletContext().getRequestDispatcher(urlFormulaire).forward(request,response);
      return;
    }//
    // vérification des paramètres
    ArrayList erreursAppel=new ArrayList();
    nom=nom.trim();
    if(nom.equals("")) erreursAppel.add("Le champ [nom] n'a pas été rempli");
    if(! age.matches("^\\s*\\d+\\s*$")) erreursAppel.add("Le champ [age] est erroné");
    if(erreursAppel.size()!=0){
      // on envoie la page d'erreurs
      request.setAttribute("erreurs",erreursAppel);
      getServletContext().getRequestDispatcher(urlErreurs).forward(request,response);
      return;
    }//if
    // les paramètres sont corrects - on envoie la page réponse
    request.setAttribute("nom",nom);
    request.setAttribute("age",age);
    getServletContext().getRequestDispatcher(urlReponse).forward(request,response);
    return;
  }

  /**
   * @param request requête du client
   * @param response réponse à construire
   */
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException {
    // on passe la main au GET
```

```
    doGet(request, response);
  }
}
```

La page **formulaire1.personne.jsp** :

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

<html>
  <head>
    <title>Personne - formulaire</title>
  </head>
  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
       <form action="<%= urlAction %>" method="post">
        <table>
          <tr>
            <td>Nom</td>
              <td><input name="txtNom" value="<%= nom %>" type="text" size="20"></td>
           </tr>
           <tr>
            <td>Age</td>
              <td><input name="txtAge" value="<%= age %>" type="text" size="3"></td>
           </tr>
          <tr>
        </table>
        <table>
         <tr>
           <td><input type="submit" value="Envoyer"></td>
             <td><input type="reset" value="Rétablir"></td>
             <td><input type="button" value="Effacer"></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

La page **reponse1.personne.jsp** :

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

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

La page **erreurs1.personne.jsp** :

```
<%@ page import="java.util.ArrayList" %>

<%
```

```
  // on récupère les données
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <ul>
      <%
        for(int i=0;i<erreurs.size();i++){
          out.println("<li>" + (String) erreurs.get(i) + "</li>\n");
         }//for
       %>
    </ul>
  </body>
</html>
```

Les points principaux de la servlet sont les suivants :

- la méthode **init** de la servlet essaie de récupérer les quatre URL du fichier de configuration (urlMain, urlReponse, urlFormulaire, urlErreurs). Pour chacune des trois premières, si elle échoue un message d'erreur est ajouté à une liste d'erreurs (ArrayList). Cette liste d'erreurs sera fournie ultérieurement à la page **erreurs1.personne.jsp**. Si on échoue à récupérer la valeur du paramètre **urlErreurs**, cette variable prend la valeur **null**.
- les méthodes **doPost** et **doGet** font la même chose
- la méthode **doGet** commence par vérifier s'il y a eu des erreurs à l'initialisation :
  - elle vérifie si **urlErreurs** est resté à **null**. Si oui, une exception est lancée. En effet, cette erreur ne peut être signalée normalement au client puisqu'on n'a pas de page pour l'afficher.
  - si **urlErreurs** est non **null** et la liste des autres erreurs non vide alors celle-ci est affichée par la page JSP **urlErreurs**.
  - s'il n'y a pas d'erreurs d'initialisation, **doGet** vérifie s'il y a les paramètres attendus (txtNom, txtAge) dans la requête. Si ce n'est pas le cas, le formulaire est affiché au moyen de la page JSP **urlFormulaire**. On lui transmet un nom et un age vides ainsi que l'URL (**urlAction**) à laquelle devront être renvoyés les valeurs du formulaire lorsque l'utilisateur appuiera sur le bouton [Envoyer].
  - s'il y a des paramètres, leur validité est vérifiée au moyen d'expression régulières. Si des erreurs sont détectées, leur liste est transmise à la page JSP **urlErreurs**.
  - s'il n'y a pas d'erreurs, le nom et l'âge récupérés sont transmis à la page JSP **urlReponse**.
- la méthode **doGet** sert aussi bien à fournir le formulaire vide qu'à traiter ses valeurs lorsque l'utilisateur appuie sur le bouton [Envoyer].

Les points principaux de la page **formulaire1.reponse.jsp** sont les suivants :

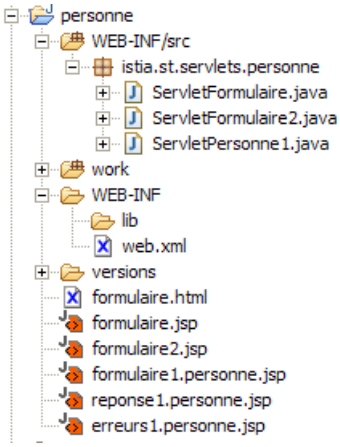
- la page reçoit en atribut la valeur **urlAction** qui est l'URL à laquelle seront envoyées les valeurs du formulaire lorsque l'utilisateur appuiera sur le bouton [Envoyer]. **urlAction** pointe sur la servlet principale. Celle-ci sert donc à la fois à fournir le formulaire vide et à en traiter les valeurs.

Les points principaux de la page **erreurs1.reponse.jsp** sont les suivants :

- la page reçoit sous la forme d'un objet **ArrayList** la valeur **erreurs** qui est la liste des erreurs à afficher.
- pour afficher celles-ci, on est amené à écrire du code Java dans le corps HTML de la page. On doit toujours viser à réduire celui-ci au minimum pour ne pas polluer le code HTML. Nous verrons ultérieurement qu'il existe des solution permettant du zéro Java dans les pages JSP.

I.11.4 Déploiement

Le déploiement de l'application est celui devenu maintenant habituel :



I.11.5 Tests

Si Tomcat était actif lorsque vous avez fait les modifications précédentes (classes et web.xml), il a du recharger le contexte<personne> et le signaler dans ses logs :

```
7 nov. 2004 10:28:18 org.apache.catalina.startup.HostConfig restartContext
INFO: restartContext(/personne)
7 nov. 2004 10:28:18 org.apache.catalina.core.StandardContext reload
INFO: Le rechargement de ce contexte a démarré
```

Au besoin, rechargez manuellement le contexte [clic droit sur projet/Projet Tomcat/Recharger le contexte] ou relancez Tomcat. Ceci fait, on pourra reprendre les tests montrés en exemple au début de cette section. On pourra en rajouter. On pourra par exemple supprimer la présence d'un des quatre paramètres de configuration **urlXXX** dans **web.xml** et voir le résultat.

I.12 Gestion d'une session

Lectures : chapitre 3 : 3.1, 3.2, 3.3, 3.4

I.12.1 Introduction

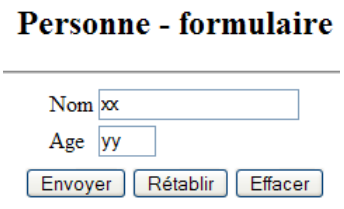
Nous nous proposons maintenant d'ajouter à notre application une gestion de session. Rappelons les points suivants :

- le dialogue HTTP client-serveur est une suite de séquences demande-réponse déconnectées entre elles
- la session sert de mémoire entre différentes séquences demande-réponse d'un même utilisateur. S'il y a N utilisateurs, il y a N sessions.

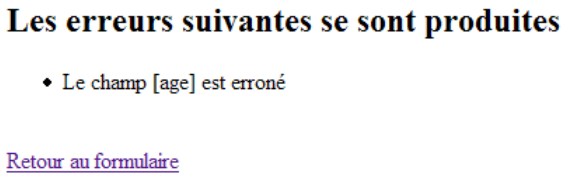
La séquence d'écran suivante montre ce qui est maintenant désiré dans le fonctionnement de l'application :

Echange n° 1

formulaire envoyé



réponse reçue



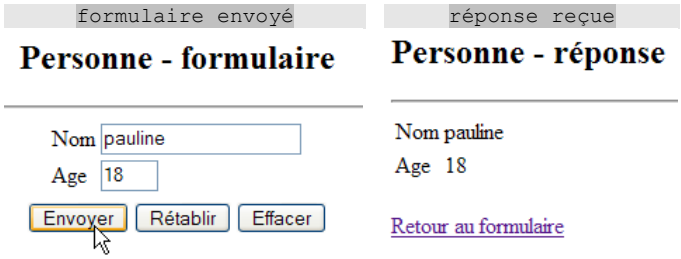
La nouveauté vient du lien de retour au formulaire qui a été rajouté dans la page d'erreurs.

Echange n° 2

Dans l'**échange n° 1**, l'utilisateur a donné pour le couple (nom,âge) les valeurs (xx,yy). Si au cours de l'échange, le serveur a eu connaissance de ces valeurs, à la fin de l'échange il les "**oublie**". Or on peut constater que lors de l'**échange n° 2** il est capable de réafficher leurs valeurs dans sa réponse. C'est la notion de **session** qui permet au serveur web de mémoriser des données au fil des échanges successifs client-serveur. Au cours de l'échange n°1, le serveur doit mémoriser dans la session le couple (nom,age) que le client lui a envoyé afin d'être capable de l'afficher au cours de l'échange n° 2.

Voici un autre exemple de mise en oeuvre de la session entre deux échanges :

Echange n° 1



La nouveauté vient du lien de retour au formulaire qui a été rajouté dans la page de la réponse.

Echange n° 2

I.12.2 Configuration

Le fichier **web.xml** de l'application **/personne** est complété par les sections suivantes :

```
 <servlet>
   <servlet-name>personne2</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletPersonne2</servlet-class>
    <init-param>
      <param-name>urlMain</param-name>
       <param-value>/personne/main2</param-value>
    </init-param>
    <init-param>
      <param-name>urlReponse</param-name>
       <param-value>/reponse2.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreurs</param-name>
       <param-value>/erreurs2.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlFormulaire</param-name>
       <param-value>/formulaire2.personne.jsp</param-value>
    </init-param>
 </servlet>
....
 <servlet-mapping>
   <servlet-name>personne2</servlet-name>
    <url-pattern>/main2</url-pattern>
 </servlet-mapping>
```

La nouvelle application appelée **personne2** est accessible via l'URL /**main2** (servlet-mapping)

I.12.3 Les codes

La servlet principale est la classe **ServletPersonne2**. Elle ne diffère de la classe **ServletPersonne1** que par sa méthode **doGet** et l'ajout de l'import de la classe **HttpSession** :

```
package istia.st.servlets.personne;

import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;


public class ServletPersonne2
  extends HttpServlet {

...

  /**
   * @param request requête du client
   * @param response réponse à construire
   */
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException,
ServletException {

    // on vérifie comment s'est passée l'initialisation de la servlet
    if(urlErreurs==null) throw new ServletException("Le paramètre [urlErreurs] n'a pas été initialisé");
    if(erreurs.size()!=0){
      // on passe la main à la page d'erreurs
      request.setAttribute("erreurs",erreurs);
      getServletContext().getRequestDispatcher(urlErreurs).forward(request,response);
      //fin
      return;
    }
    // on récupère les paramètres
    String nom=request.getParameter("txtNom");
    String age=request.getParameter("txtAge");
    // des paramètres ?
    if(nom==null || age==null){
      // on envoie le formulaire
      request.setAttribute("urlAction",urlMain);
      getServletContext().getRequestDispatcher(urlFormulaire).forward(request,response);
      return;
    }//
     // on mémorise les deux données dans la session
     HttpSession session=request.getSession();
     session.setAttribute("nom",nom);
     session.setAttribute("age",age);
    // vérification des paramètres
    ArrayList erreursAppel=new ArrayList();
    nom=nom.trim();
    if(nom.equals("")) erreursAppel.add("Le champ [nom] n'a pas été rempli");
    if(! age.matches("^\\s*\\d+\\s*$")) erreursAppel.add("Le champ [age] est erroné");
    if(erreursAppel.size()!=0){
      // on envoie la page d'erreurs
      request.setAttribute("erreurs",erreursAppel);
      request.setAttribute("urlRetour",urlMain);
      getServletContext().getRequestDispatcher(urlErreurs).forward(request,response);
      return;
    }//if
    // les paramètres sont corrects - on envoie la page réponse
    request.setAttribute("nom",nom);
    request.setAttribute("age",age);
    request.setAttribute("urlRetour",urlMain);
    getServletContext().getRequestDispatcher(urlReponse).forward(request,response);
    return;
  }
...
```

Les points à noter :

- les paramètres [nom] et [age], s'ils existent, sont enregistrés dans la session courante. Ils y seront disponibles pour le formulaire **formulaire2.personne.jsp**.

- un nouvel attribut **urlRetour** est transmis aux pages **urlErreurs** et **urlReponse**. Il servira à générer le lien de retour vers le formulaire. Ce lien de retour pointe vers la servlet principale **/personne/main2**. Cette servlet recevant une requête sans paramètres la transmettra au formulaire qui lui récupèrera dans la session le couple (nom,age) s'il y a été placé auparavant.

La page **formulaire2.personne.jsp** est identique à la page **formulaire1.personne.jsp,** si ce n'est qu'elle récupère ses paramètres [nom] et [age] dans la session plutôt que dans la requête.

```
<%
 // on récupère les paramètres nom et age dans la session
  String nom=(String)session.getAttribute("nom");
  if(nom==null) nom="";
  String age=(String)session.getAttribute("age");
  if(age==null) age="";
  // l'attribut urlAction est lui récupéré dans la requête
  String urlAction=(String)request.getAttribute("urlAction");
%>

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

La page **reponse2.personne.jsp** est identique à la page **reponse1.personne.jsp** à quelques détails près :

- un nouvel attribut **urlRetour** est récupéré dans la requête
- cet attribut est utilisé pour générer le lien de bas de page

```
<%
 // on récupère les données nom, age et urlRetour nécessaire à l'affichage de la page
  String nom=(String)request.getAttribute("nom");
  String age=(String)request.getAttribute("age");
  String urlRetour=(String)request.getAttribute("urlRetour");
%>

<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>
     <a href="<%= urlRetour %>">Retour au formulaire</a>
  </body>
</html>
```

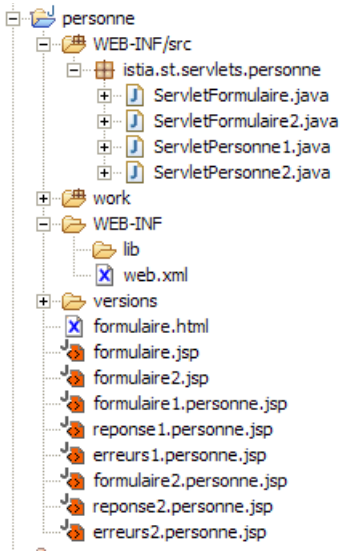
La page **erreurs2.personne.jsp** est identique à la page **erreurs1.personne.jsp** aux détails près suivants :

- l'attribut **urlRetour** est récupéré dans la requête
- puis il est utilisé pour générer le lien de bas de page

```
<%@ page import="java.util.ArrayList" %>

<%
 // on récupère les données
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
  String urlRetour=(String)request.getAttribute("urlRetour");
%>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
     <ul>
```

```
    <%
      for(int i=0;i<erreurs.size();i++){
        out.println("<li>" + (String) erreurs.get(i) + "</li>\n");
       }//for
     %>
   </ul>
   <br>
    <a href="<%= urlRetour %>">Retour au formulaire</a>
  </body>
</html>
```

I.12.4 Déploiement



I.12.5 Tests

On pourra reproduire les tests présentés au début de cette nouvelle version.

I.13 Scripts côté navigateur

Lectures : chapitre 1 : 1.6, 1.61, 1.6.2, chapitre 9 : 9.2

I.13.1 Le nouveau formulaire

Notre application vérifie côté serveur la validité des paramètres nom et âge du formulaire. Il doit toujours en être ainsi. Une application web ne doit jamais faire l'hypothèse que ses clients vérifient les données qu'ils lui envoient. Elle doit au contraire toujours supposer qu'on peut lui envoyer des données erronées parfois volontairement.

Lorsque le client est un navigateur, on peut inclure dans la page HTML qu'on envoie à celui-ci des scripts qui vérifieront les données à envoyer au serveur avant de les envoyer réellement. Si celles-ci s'avèrent invalides, elles ne sont alors pas envoyées au serveur et les erreurs sont signalées à l'utilisateur du navigateur. On a ainsi gagné un aller-retour client-serveur inutile.

Dans notre application personne la seule page JSP où il est utile de placer des scripts est celle du formulaire. La page appelée maintenant **formulaire3.personne.jsp** devient la suivante :

```
<%
  // on récupère les paramètres nom et age dans la session
  String nom=(String)session.getAttribute("nom");
  if(nom==null) nom="";
  String age=(String)session.getAttribute("age");
  if(age==null) age="";
  // l'attributs urlAction est lui récupéré dans la requête
  String urlAction=(String)request.getAttribute("urlAction");
%>

<html>
  <head>
    <title>Personne - formulaire</title>
    <script language="javascript">
    // ------------------------------
```

```
    function effacer(){
      // on efface les champs de saisie
        with(document.frmPersonne){
          txtNom.value="";
          txtAge.value="";
        }//with
      }//effacer
    // ------------------------------
      function envoyer(){
      // vérification des paramètres avant de les envoyer
        with(document.frmPersonne){
          // le nom ne doit pas être vide
          champs=/^\s*$/.exec(txtNom.value);
          if(champs!=null){
            // le nom est vide
            alert("Vous devez indiquer un nom");
            txtNom.value="";
            txtNom.focus();
            // retour à l'interface visuelle
            return;
          }//if
          // l'âge doit être un entier positif
          champs=/^\s*\d+\s*$/.exec(txtAge.value);
          if(champs==null){
            // l'âge est incorrect
            alert("Age incorrect");
            txtAge.focus();
            // retour à l'interface visuelle
            return;
          }//if
          // les paramètres sont corrects - on les envoie au serveur
          submit();
        }//with
      }//envoyer
    </script>
  </head>

  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
      <form action="<%= urlAction %>" method="post" name="frmPersonne">
       <table>
         <tr>
           <td>Nom</td>
            <td><input name="txtNom" value="<%= nom %>" type="text" size="20"></td>
          </tr>
          <tr>
           <td>Age</td>
            <td><input name="txtAge" value="<%= age %>" type="text" size="3"></td>
          </tr>
         <tr>
        </table>
        <table>
         <tr>
            <td><input type="button" value="Envoyer" onclick="envoyer()"></td>
           <td><input type="reset" value="Rétablir"></td>
            <td><input type="button" value="Effacer" onclick="effacer()"></td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>
```

Les points importants :

- un nom a été donné au formulaire : **<form ... name="frmPersonne">**. Ce nom est utilisé dans les scripts côté navigateur
- le bouton [Envoyer] autrefois de type **submit** est devenu de type **button**. Lorsqu'on clique dessus, la fonction javascript **envoyer**() est appelée.
- de même lorsqu'on clique sur le bouton [Effacer] lui aussi de type **button**, la fonction javascript **effacer**() est appelée.
- les fonctions javascript **envoyer**() et **effacer**() ont été placées entre les balises **<script>** et **</script>** du document HTML. Le lecteur est invité à lire le chapitre 9 du polycopié pour une première introduction à Javascript et notamment à ses expressions régulières.

I.13.2 Configuration et déploiement

Les sections suivantes sont ajoutées au fichier **web.xml** :

```
<servlet>
    <servlet-name>personne3</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletPersonne2</servlet-class>
    <init-param>
      <param-name>urlMain</param-name>
       <param-value>/personne/main3</param-value>
    </init-param>
    <init-param>
      <param-name>urlReponse</param-name>
       <param-value>/reponse2.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreurs</param-name>
       <param-value>/erreurs2.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlFormulaire</param-name>
       <param-value>/formulaire3.personne.jsp</param-value>
    </init-param>
  </servlet>
.........
  <servlet-mapping>
    <servlet-name>personne3</servlet-name>
    <url-pattern>/main3</url-pattern>
  </servlet-mapping>
```

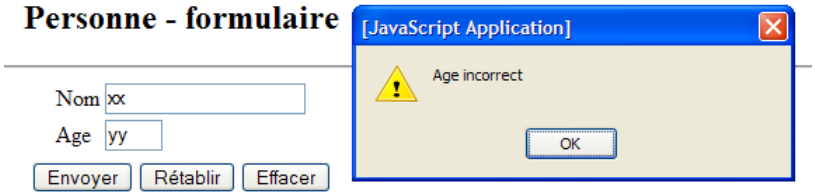
Les changements apportés sont les suivants :

- la page JSP du formulaire s'appelle désormais **formulaire3.personne.jsp**.
- l'application s'appelle **personne3** et est disponible via l'URL **/personne/main3**.

Rien d'autre ne change.

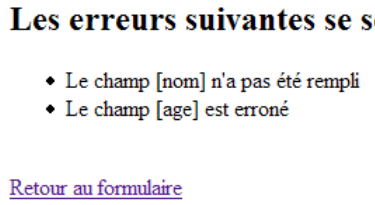
On suivra la méthode de déploiement des cas précédents pour déployer cette nouvelle application.

I.13.3 Tests



I.14 Utiliser des bibliothèques de balises

Considérons la vue [erreurs2.personne.jsp] qui affiche une liste d'erreurs :



Il y a plusieurs façons d'écrire une telle page. Nous ne nous intéressons ici qu'à la partie affichage des erreurs. Une première solution est d'utiliser du code Java comme il a été fait :

```
<%@ page import="java.util.ArrayList" %>

<%
  // on récupère les données
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
  String urlRetour=(String)request.getAttribute("urlRetour");
%>

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

```
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <ul>
      <%
        for(int i=0;i<erreurs.size();i++){
          out.println("<li>" + (String) erreurs.get(i) + "</li>\n");
         }//for
       %>
    </ul>
    <br>
    <a href="<%= urlRetour %>">Retour au formulaire</a>
  </body>
</html>
```

La page JSP récupère la liste des erreurs dans la requête et l'affiche à l'aide d'une boucle. La page mélange code HTML et code Java, ce qui peut être problématique si la page doit être maintenue par un infographiste qui en général ne comprendra pas le code Java. Pour éviter ce mélange, on utilise des bibliothèques de balises qui vont apporter des possibilités nouvelles aux pages JSP. Avec la bibliothèque de balises JSTL, la vue précédente devient la suivante :

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
<%@ page isELIgnored="false"%>
<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
    <ul>
      <c:forEach var="erreur" items="${erreurs}">
        <li><c:out value="${erreur}"/></li>
      </c:forEach>
    </ul>
    <br>
     <a href="<c:url value="${urlRetour}"/>">Retour au formulaire</a>
  </body>
</html>
```

La balise

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

signale l'utilisation d'une bibliothèque de balises dont la définition se trouve dans le fichier [/WEB-INF/c.tld]. Ces balises seront utilisées dans le code de la page, préfixées du mot **c** (prefix="c"). On peut utiliser tout préfixe de son choix. Ici **c** signifie [core]. Les préfixes permettent d'utiliser plusieurs bibliothèques de balises qui pourraient avoir les mêmes noms pour certaines balises. L'utilisation du préfixe lève l'ambiguïté.

La balise

```
<%@ page isELIgnored="false"%>
```

demande l'évaluation des éléments dynamiques ${identificateur} de la page. Sa présence n'est pas toujours nécessaire. Elle dépend à la fois de la DTD utilisée dans [web.xml] et de la version de Jakarta Taglibs utilisée. Dans certaines configurations, l'évaluation des éléments dynamiques ${identificateur} est faite par défaut, dans d'autres cas non. Il faut alors mettre la balise précédente dans le code JSP.

La nouvelle page n'a plus de code Java. La boucle d'affichage des erreurs a été remplacée par le texte suivant :

```
      <c:forEach var="erreur" items="${erreurs}">
        <li><c:out value="${erreur}"/></li>
      </c:forEach>
```

- la balise **<c:forEach>** sert à délimiter une boucle
- la balise **<c:out ...>** sert à écrire une valeur

La balise **<c:forEach>** a ici deux attributs :

- **items="${erreurs}"** indique la collection d'objets sur laquelle il faut itérer. Ici, la collection est l'objet **erreurs**. Où celui-ci est-il trouvé ? La page JSP va chercher un attribut s'appelant "erreurs" successivement et dans l'ordre dans :
  - l'objet [request] qui représente la requête transmise par le contrôleur : **request.getAttribute("erreurs")**
  - l'objet [session] qui représente la session du client : **session.getAttribute("erreurs")**
  - l'objet [context] qui représente le contexte de l'application web : **context.getAttribute("erreurs")**

La collection désignée par l'attribut **items** peut avoir diverses formes : tableau, ArrayList, objet implémentant l'interface List, ...

- **var="erreur"** sert à donner un nom à l'élément courant de la collection en cours de traitement. La boucle <forEach> va être exécutée successivement pour chaque élément de la collection **items**. A l'intérieur de la boucle, l'élément de la collection en cours de traitement sera donc désigné ici par **${erreur}**.

La balise **<c:out value="..."/>** sert à écrire la valeur désignée par l'attribut **value**. L'opération **<c:out value="${erreur}"/>** insère la valeur de la variable **erreur** dans le texte. Cette variable n'est pas nécessairement une chaîne de caractères. La balise utilise la méthode **erreur.toString()** pour insérer l'objet **erreur**.

Pour en revenir à notre exemple d'affichage des erreurs :

- le contrôleur mettra dans la requête transmise à la page JSP un ArrayList de messages d'erreurs, donc un ArrayList d'objets String : **request.setAttribute("erreurs",erreurs)** où **erreurs** est le ArrayList ;
- à cause de l'attribut **items="${erreurs}"**, la page JSP va chercher un attribut appelé **erreurs**, successivement dans la **requête**, la **session**, le **contexte**. Elle va le trouver dans la requête : **request.getAttribute("erreurs")** va rendre le ArrayList placé dans la requête par le contrôleur ;
- la variable erreur de l'attribut **var="erreur"** va donc désigner l'élément courant du ArrayList, donc un objet String. La **méthode erreur.toString()** va insérer la valeur de ce String, ici un message d'erreur, dans le flux HTML de la page.

Les objets de la collection traitée par la balise <forEach> peuvent être plus complexes que de simples chaînes de caractères. Prenons l'exemple d'une page JSP qui affiche une liste d'articles :

```
    <c:forEach var="article" items="${listarticles}">
      <tr>
        <td><c:out value="${article.nom}"/></td>
        <td><c:out value="${article.prix}"/></td>
        <td><a href="<c:url value="?action=infos&id=${article.id}"/>">Infos</a></td>
      </tr>
    </c:forEach>
```

Ici, [listarticles] est un ArrayList d'objets de type [Article]. Celui-ci est un Javabean avec les champs [id, nom, prix, stockActuel, stockMinimum]. Chacun de ces champs est accompagné de ses méthodes **get** et **set**. L'objet [listarticles] a été placé dans la requête par le contrôleur. La page JSP précédente va le récupérer dans l'attribut **items** de la balise **forEach**. L'objet courant **article** (var="article") désigne donc un objet de type [Article]. Considérons la balise d'écriture suivante :

```
<c:out value="${article.nom}"/>
```

Que signifie ${article.nom} ? En fait diverses choses selon la nature de l'objet **article**. Pour obtenir la valeur de **article.nom**, la page JSP va essayer deux choses :

1. **article.getNom()** - on notera l'orthographe **getNom** pour récupérer le champ **nom** (norme Javabean)
2. **article.get("nom")**

On voit donc que l'objet [article] peut être un bean avec un **champ nom**, ou un dictionnaire avec une **clé nom**. Il n'y a pas de limites à la hiérarchie de l'objet traité. Ainsi la balise

```
<c:out value="${individu.enfants[1].nom}"/>
```

permet de traiter un objet [individu] de type suivant :

```
class Individu{
  private String nom;
  private String prénom;
  private Individu[] enfants;
  // méthodes de la norme Javabean
  public String getNom(){ return nom;}
  public String getPrénom(){ return prénom;}
  public Individu getEnfants(int i){ return enfants[i];}
}
```

Pour obtenir la valeur de **${individu.enfants[1].nom}**, la page JSP va essayer diverses méthodes dont celle-ci qui réussira : **individu.getEnfants(1).getNom()** où **individu** désigne un objet de type **Individu**.

La page JSP d'erreurs utilise une balise JSTL également pour afficher le lien de retour vers le formulaire de l'application :

```
    <a href="<c:url value="${urlRetour}"/>">Retour au formulaire</a>
```

Si [/personne] est le contexte de l'application et [main3] la valeur de la variable [urlRetour], le texte HTML généré par le code ci-dessus sera le suivant :

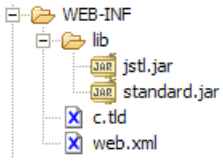
```
    <a href="/personne/main3"/>">Retour au formulaire</a>
```

La balise <c:url ..> préfixe toute url par le contexte de l'application, ici [/personne]. Cela permet de gérer des url indépendamment du contexte. C'est important car le nom d'un contexte peut être amené à changer.

L'utilisation de la bibliothèque de balises JSTL dans les vues JSP de notre application nécessite :

- la présence du fichier **c.tld** dans l'arborescence de l'application web. Ici, nous l'avons placé dans le dossier [WEB-INF] :

- la présence des archives des classes utilisées par la bibliothèque JSTL : **jstl.jar** et **standard.jar**. Ces deux archives sont placées dans [WEB-INF/lib] :



- la présence de la directive suivante dans toutes les pages JSP utilisant la bibliothèque :

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

Une nouvelle application utilisant la bibliothèque JSTL dans les vues amène les modifications suivantes :

web.xml

```
  <servlet>
    <servlet-name>personne4</servlet-name>
    <servlet-class>istia.st.servlets.personne.ServletPersonne2</servlet-class>
    <init-param>
      <param-name>urlMain</param-name>
       <param-value>/main4</param-value>
    </init-param>
    <init-param>
      <param-name>urlReponse</param-name>
       <param-value>/reponse4.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreurs</param-name>
       <param-value>/erreurs4.personne.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlFormulaire</param-name>
       <param-value>/formulaire4.personne.jsp</param-value>
    </init-param>
  </servlet>

...

  <servlet-mapping>
    <servlet-name>personne4</servlet-name>
    <url-pattern>/main4</url-pattern>
  </servlet-mapping>
```

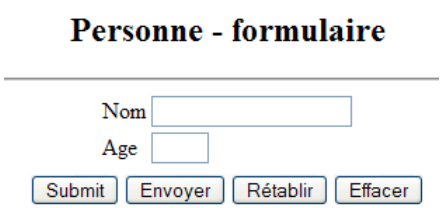
En-dehors des noms d'url qui changent, la principle différence vient de la valeur du paramètre [urlMain]. Celle-ci ne contient plus le nom du contexte /personne, ceci grâce à l'utilisation de la balise JSTL <c:url ...> dans les vues.

Vue formulaire4.personne.jsp

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
<%@ page isELIgnored="false"%>

<html>
  <head>
    <title>Personne - formulaire</title>
    <script language="javascript">
.... (idem version précédente)
    </script>
  </head>
```

```
  <body>
    <center>
      <h2>Personne - formulaire</h2>
       <hr>
       <form method="post" name="frmPersonne" action="<c:url value="${urlAction}"/>">
        <table>
          <tr>
           <td>Nom</td>
             <td><input name="txtNom" value="<c:out value="${nom}"/>" type="text" size="20"></td>
           </tr>
           <tr>
            <td>Age</td>
             <td><input name="txtAge" value="<c:out value="${age}"/>" type="text" size="3"></td>
           </tr>
         <tr>
        </table>
        <table>
         <tr>
            <td><input type="submit" value="Submit"></td>
           <td><input type="button" value="Envoyer" onclick="envoyer()"></td>
             <td><input type="reset" value="Rétablir"></td>
             <td><input type="button" value="Effacer" onclick="effacer()"></td>
           </tr>
        </table>
       </form>
     </center>
  </body>
</html>
```

Nous avons rajouté un bouton de type [submit] au formulaire afin de pouvoir court-circuiter la vérification des données faites par le navigateur lors de l'utilisation du bouton [Envoyer] et d'avoir ainsi accès à la page [erreurs4.personne.jsp].



Vue reponse4.personne.jsp

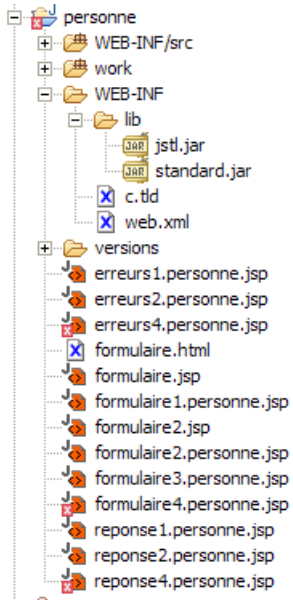
```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
<%@ page isELIgnored="false"%>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Personne - réponse</h2>
     <hr>
     <table>
      <tr>
        <td>Nom</td>
         <td><c:out value="${nom}"/></td>
       </tr>
      <tr>
        <td>Age</td>
         <td><c:out value="${age}"/></td
       </tr>
     </table>
     <br>
     <a href="<c:url value="${urlRetour}"/>">Retour au formulaire</a>
  </body>
</html>
```

Vue erreurs4.personne.jsp

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
<%@ page isELIgnored="false"%>

<html>
  <head>
    <title>Personne</title>
  </head>
  <body>
    <h2>Les erreurs suivantes se sont produites</h2>
```

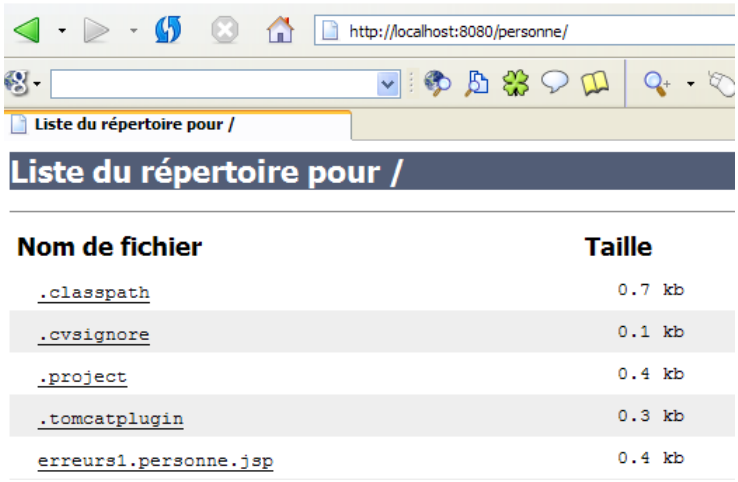
```
    <ul>
      <c:forEach var="erreur" items="${erreurs}">
        <li><c:out value="${erreur}"/></li>
      </c:forEach>
    </ul>
     <br>
     <a href="<c:url value="${urlRetour}"/>">Retour au formulaire</a>
  </body>
</html>
```

Déploiement



I.15 Créer une page d'accueil

Lorsque nous demandons l'url [http://localhost:8080/personne/], nous obtenons actuellement le résultat suivant (vue partielle) :



Nous obtenons le contenu du dossier de l'aplication. Ce n'est pas une bonne chose. Ici, un internaute peut obtenir le contenu du fichier [erreurs1.personne.jsp] et de façon générale de tous les fichiers qui seront exposés dans la vue ci-dessus. Il est possible d'éviter cela au moins de deux façons :

1. configurer Tomcat pour qu'il n'affiche pas le contenu du dossier
2. faire en sorte que lorsque le contexte est demandé, on affiche une page dite d'accueil plutôt que la liste ci-dessus.

Nous développons ici la deuxième solution. On ajoute au fichier [web.xml] une nouvelle balise :

```
.....
  <welcome-file-list>
    <welcome-file>/index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

La balise <welcome-file-list> sert à définir la liste des vues à présenter lorsqu'un client demande le contexte de l'application. Il peut y avoir plusieurs vues. La première trouvée est présentée au client. Ici nous n'en avons qu'une [index.jsp]. Celle-ci se contente de rediriger le client vers une autre URL :

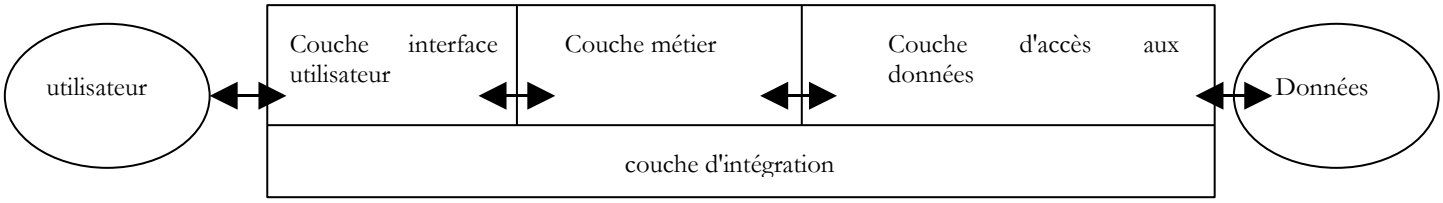
```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>

<c:redirect url="/main4"/>
```

La redirection se fait au moyen de la balise JSTL **<c:redirect .../>**. Le lecteur est invité à tester cette nouvelle version lui-même car ici les copies d'écran ne peuvent montrer la redirection à l'oeuvre.

I.16 Développement MVC

I.16.1 Introduction

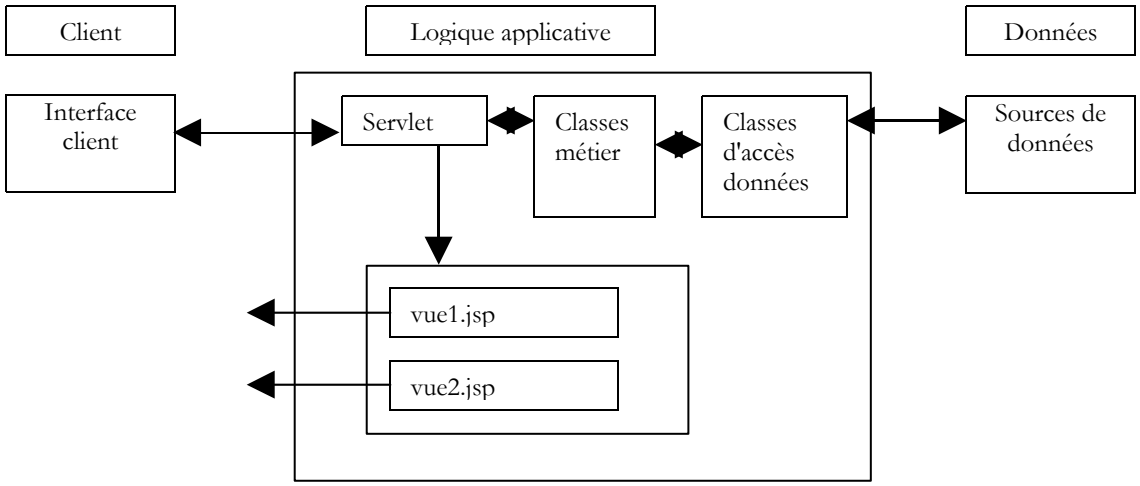
Dans le développement logiciel en général et dans le développement web en particulier, on cherche à séparer nettement les couches présentation, traitement et accès aux données. L'architecture d'une application web est ainsi souvent la suivante :



Une telle architecture, appelée 3-tiers ou à 3 niveaux cherche à respecter le modèle MVC (Model View Controller) :

- l'interface utilisateur contient le V (la vue) et le C (le contrôleur)
- les couches métier et d'accès aux données forment le M (Modèle)

Pour une architecture web J2EE, le schéma à trois couches peut être détaillé de la façon suivante :

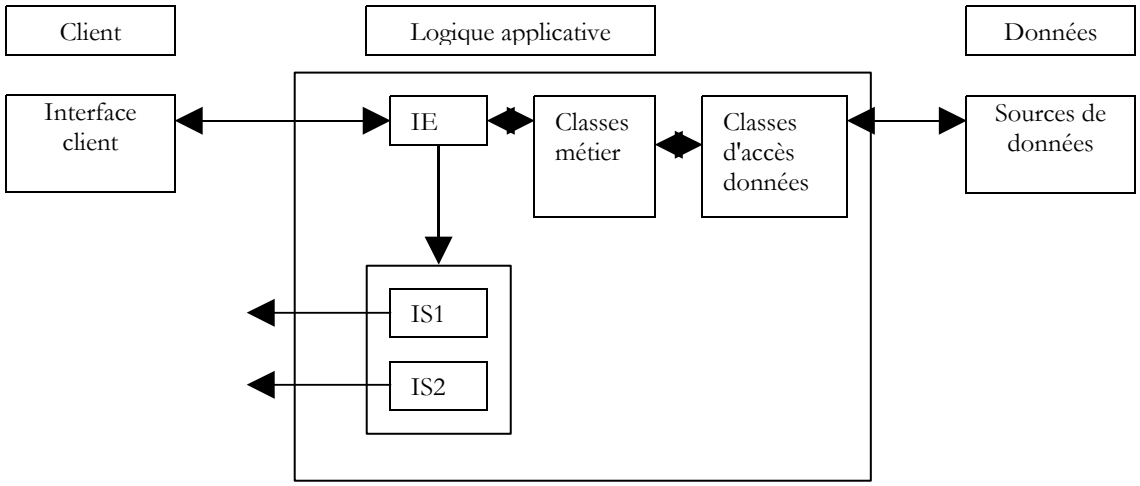


`M=modèle`	les classes métier, les classes d'accès aux données et la source de données
`V=vues`	les pages JSP
`C=contrôleur`	la servlet de traitement des requêtes clientes

L'**interface utilisateur** est ici un navigateur web mais cela pourrait être également une application autonome qui via le réseau enverrait des requêtes HTTP au service web et mettrait en forme les résultats que celui-ci lui envoie. La **logique applicative** est constituée des scripts traitant les demandes de l'utilisateur, ici la servlet Java. La **source de données** est souvent une base de données mais cela peut être aussi un annuaire LDAP ou 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.

- On mettra la logique métier de l'application dans des classes Java séparées de la servlet qui contrôle le dialogue demande-réponse. Ainsi le bloc [Logique applicative] ci-dessus sera constitué des éléments suivants :



Dans le bloc [Logique Applicative], on pourra distinguer

- le bloc [IE=Interface d'Entrée] qui est la porte d'entrée de l'application. Elle est la même quelque soit le type de client.
- le bloc [Classes métier] qui regroupe des classes Java nécessaires à la logique de l'application. Elles sont indépendantes du client.
- le bloc des générateurs des pages réponse [IS1 IS2 ... IS=Interface de Sortie]. Chaque générateur est chargé de mettre en forme les résultats fournis par la logique applicative pour un type de client donné : code HTML pour un navigateur ou un téléphone WAP, code XML pour une application autonome, ...

Ce modèle assure une bonne indépendance vis à vis des clients. Que le client change ou qu'on veuille faire évoluer sa façon de présenter les résultats, ce sont les générateurs de sortie [IS] qu'il faudra créer ou adapter.

- Dans une application web, l'indépendance entre la couche présentation et la couche traitement peut être améliorée par l'utilisation de feuilles de style. Celles-ci gouvernent la présentation d'une page Web au sein d'un navigateur. Pour changer cette présentation, il suffit de changer la feuille de style associée. Il n'y a pas à toucher à la logique de traitement.

- Dans le diagramme ci-dessus, les classes métier passent par des classes intermédiaires pour accéder aux données. Ainsi les classes métier forment le noyau dur de l'application insensible aux changements de présentation ou d'accès aux données.

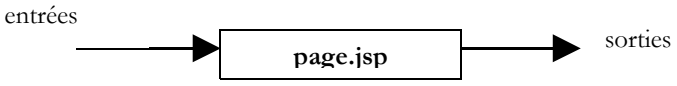
I.16.2 Une démarche de développement MVC en Java

Nous proposons ici une démarche possible pour le développement d'applications web java selon le modèle MVC précédent :

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 soit dans la requête (request) ou la session (session).
- 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/C/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/C/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.16.3 Pour approfondir le développement web MVC

Plusieurs outils du monde "open source" ont systématisé l'approche MVC pour le développement web en Java :

- **Struts** - c'est l'outil le plus connu. La plupart des développements MVC en java sont faits actuellement avec la démarche et les bibliothèques Struts.
- **Spring** - c'est un outil plus récent qui ne se limite pas au développement MVC. Son intérêt principal est qu'il apporte une aide dans les trois couches [web - métier - dao], du développement web. Struts lui se cantonne à la seule couche web.

Table des matières

I.1 Objectifs 1
I.2 Les outils 1
I.3 Le conteneur de servlets Tomcat 5 1
I.4 Déploiement d'une application web au sein du serveur Tomcat 4
I.5 Installation de plugins Eclipse 10
I.6 Lancer-Arrêter Tomcat depuis Eclipse 11
I.7 Développement d'une application web avec Eclipse/Tomcat 12
I.7.1 Création du contexte de l'application 12
I.7.2 Création d'un document HTML 14
I.8 Une page JSP 16
I.9 Une servlet 17
I.9.1 Configuration d'Eclipse 17
I.9.2 Écriture de la servlet 18
I.9.3 Déploiement de la servlet dans Tomcat 20
I.9.4 Test de la servlet 22
I.10 Coopération servlets et pages JSP 23
I.10.1 La servlet 23
I.10.2 La page JSP 24
I.10.3 Déploiement de l'application 25
I.11 Une application multi-pages 26
I.11.1 Introduction 26
I.11.2 Configuration 27
I.11.3 Les codes 27
I.11.4 Déploiement 30
I.11.5 Tests 31
I.12 Gestion d'une session 31
I.12.1 Introduction 31
I.12.2 Configuration 32
I.12.3 Les codes 33
I.12.4 Déploiement 35
I.12.5 Tests 35
I.13 Scripts côté navigateur 35
I.13.1 Le nouveau formulaire 35
I.13.2 Configuration et déploiement 37
I.13.3 Tests 37
I.14 Utiliser des bibliothèques de balises 37
I.15 Créer une page d'accueil 42
I.16 Développement MVC 43
I.16.1 Introduction 43
I.16.2 Une démarche de développement MVC en Java 44
I.16.3 Pour approfondir le développement web MVC 46