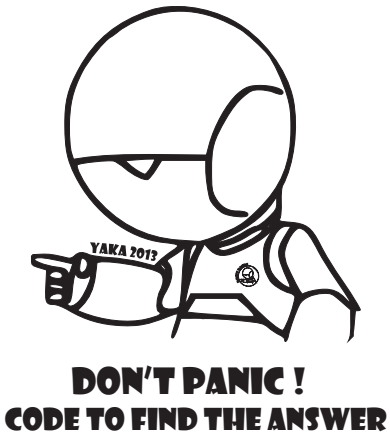
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Struts - TP

Version 1
2012



YAKA 2013 <yaka@epita.fr>

Copyright

Contents

1 Présentation

La dernière fois, nous avons appris à créer des modèles de données à l'aide du framework Hibernate. Aujourd'hui, nous allons découvrir le framework Struts qui vous permettra de créer :

- Des contrôleurs, qui sont destinés à manipuler les données ;
- Des vues, qui permettront à terme d'afficher le résultat de ces manipulations.

Ainsi, les applications web sont découpées en 3 blocs distincts, qui effectuent des opérations différentes, mais complémentaires : c'est le modèle MVC, pour Modèle Vue Contrôleur.

Le MVC est beaucoup utilisé en conception web, aussi un petit résumé s'impose-t-il afin de vous assurer de bien réussir vos futurs entretiens d'embauche ! Synthétiquement, donc :

- Modèle : Votre modèle de données et les routines pour intéragir avec celui-ci : accès aux bases de données, par exemple ;
- Vue : Votre couche de présentation, l'interface. Elle met visuellement en forme le modèle et récupère les entrées de l'utilisateur ;
- Contrôleur : La couche métier. Prend en charge la synchronisation du modèle et de la vue, lance les traitements, commande les deux autres couches.

2 Préparation

2.1 Création du projet

Dans Netbeans, créez un nouveau projet Web : **File -> New Project** et séléctionnez **Java Web -> Web Application**.

- Nommez votre application : **TPStruts** ;
- Assurez-vous que le serveur Glassfish v3 est sélectionné ;
- Sélectionnez **Hibernate 3.2.5** dans la liste des Frameworks pour importer automatiquement les bibliothèques nécessaires ;
- Dans la configuration Hibernate, sélectionnez le profil de connexion créé durant le TP Hibernate ;
- Cliquez sur **Finish** pour créer le projet.

2.2 Configuration

Avant tout, vous devez placer les bibliothèques de Struts dans le projet.

- Téléchargez l'archive **lib_struts.zip** sur l'intranet ;
- Copiez le dossier **lib** contenu dans l'archive dans le répertoire **WEB-INF** ;
- Dans les propriétés du projet, sélectionnez **Libraries**, puis **Add JAR/Folder** ;
- Sélectionnez les bibliothèques précedemment copiées, puis validez.

2.2.1 web.xml

Nous devons maintenant indiquer au serveur d'utiliser Struts pour filtrer les URLs et lui indiquer la page d'accueil.

Pour rappel, le serveur (ici, Glassfish) est l'entité qui va recevoir les connexions des clients et décider quelle page leur renvoyer et quels traitements effectuer. Ici en particulier, Glassfish est capable d'exécuter du code Java présent (comme nous le verrons plus tard) dans les pages. Si nous utilisions un autre serveur, comme Apache, il serait capable d'exécuter un autre type de code (du PHP, par exemple[1]).

Ouvrez le fichier **web.xml** dans le dossier **WEB-INF** et placez-y le contenu suivant (attention aux sauts de lignes en copiant-collant !) :

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

<web-app version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-
   app_2_5.xsd">

    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.
           FilterDispatcher</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <session-config>
  <session-timeout>30</session-timeout>
    </session-config>

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

</web-app>
```

Les premières lignes définissent des namespaces[2] XML utiles pour J2EE. Nous créons un filtre en utilisant la classe **FilterDispatcher**, puis lui indiquons de filtrer toutes les URL. Il est bien sûr possible de mettre une expression rationnelle plus complexe.

Un peu plus de détails ? Bien sûr ! Penchons-nous sur les quatre noeuds :

- Filter : définit le type de filtre qu'utilisera Glassfish, ici celui fournit par Struts mais vous pourriez créer les votres ;
- Filter mapping : pour le filtre Struts, indique une expression rationnelle. Toute requête correspondant à celle-ci seront traitées par le serveur ;

[1] On sait que vous êtes nombreux à aimer ce langage ;)

[2] Comme vous êtes, par la force des choses, devenu des dieux en XML, ce lien ne vous apprendra rien sur ce qu'est un namespace XML : http://www.w3.org/TR/REC-xml-names/

- Session config : un exemple de paramètre que vous pouvez modifier : le temps que dure une session[3]. Vous pouvez évidemment trouver bien d'autres paramètres sur le net ! ;
- Welcome file list : La page sur laquelle un nouveau visiteur (qui n'a pas de session ouverte) sera dirigé en arrivant sur votre site.

Vous pouvez ajouter bien des choses à ce fichier, par exemple des déclarations de *taglibs*, des bibliothèques[4] de tags permettant d'ajouter des pseudo-balises dans votre page HTML pour réaliser des traitements sans écrire de code... À bon entendeur...

2.2.2 struts.xml

Nous allons maintenant créer le fichier qui sera utilisé pour définir les actions de Struts. Une action correspond à du code (contrôleur) qui sera exécuté, et une page (vue) qui sera ensuite renvoyée au client.

Créez le fichier **struts.xml** dans le dossier **src** et placez-y le contenu suivant :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN
   "
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="default" namespace="/" extends="struts-
       default">
      <action name="Exemple" class="com.yaka.struts.action.
         ExempleAction">
          <result>/exemple.jsp</result>
      </action>
    </package>
</struts>
```

Détaillons une fois encore un peu plus la structure de ce code :

- Package permet de séparer divers groupes d'actions en leur donnant un espace de nom (un ensemble d'URL) et une structure d'héritage particulière qui peut être utile pour factoriser et cloisonner votre code ;
- Action déclare l'action elle-même, avec son nom (qui se retrouvera dans l'URL) et la classe Java contenant le code à exécuter ;
- Result donne la vue (page) à retourner après l'action (ici une page JSP). Il est possible de faire autre chose que retourner une page (redirection...).

[3] Le temps pendant dlequel le visiteur sera reconnu comme étant une seule et même personne.
[4] Le premier qui dit librairie...

3 Struts 2 : pas à pas

3.1 Créer une action et une vue

Nous allons écrire une première action Struts 2. Comme dans tout Framework MVC, l'action sera associée à une vue. Les conventions veulent que l'action soit nommée **xxxAction.java** et la vue associée **xxx.jsp**. En J2EE, les pages HTML sont toujours suivies de l'extension **.jsp**.

3.1.1 L'action

Pour commencer, créez un nouveau package dans le repertoire **src** : **com.yaka.struts.action**. Dans ce package, créez une nouvelle classe Java nommée **ExempleAction**.

Placez le code suivant dans le fichier :

```
package com.yaka.struts.action;

import com.opensymphony.xwork2.ActionSupport;

public class ExempleAction extends ActionSupport
{

  private static final long serialVersionUID = 1L;
  private String myMessage;

  @Override
  public String execute() throws Exception
  {
    myMessage = "Voici ma premiere vue Struts !";
    return SUCCESS;
  }

  public String getMyMessage()
  {
    return myMessage;
  }
}
```

La méthode *execute()* est apellée quand l'action est invoquée. L'attribut *myMessage* pourra être utilisé dans la vue associée à l'action grâce au getter.

3.1.2 La vue

Créez un nouveau fichier **exemple.jsp** (**Web -> JSP**) et placez-y le code suivant :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "
   http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF
   -8">
<title>TP Struts : Exemple</title>
```

```
</head>
<body>
        <s:property value="myMessage" />
</body>
</html>
```

La ligne *<%@ taglib prefix="s" uri="/struts-tags" %>* définit le préfixe des tags Struts. Nous verrons plus en détail ce que sont les tags plus loin.

<s:property value="myMessage" /> est un tag qui récupère une proprieté de la classe Action. Comme expliqué plus haut, cette propriété est accessible via son getter.

3.1.3 Mapping de la vue avec l'action

Pour chaque **couple vue / action**, il faut mettre en place le **mapping** dans le fichier de configuration **src/struts.xml**. Nous avons déjà ajouté les lignes suivantes :

```
<action name="Exemple" class="com.yaka.struts.action.
   ExempleAction">
   <result>/exemple.jsp</result>
</action>
```

Nous associons **Exemple** à la classe d'action **ExempleAction** et le résultat à la page JSP **exemple.jsp**.

Maintenant, lorsque l'url *http://localhost:11605/TPStruts/Exemple.action* est appelée, l'action Exemple est apelée.
Il ne nous reste plus qu'à ajouter un lien vers cette action sur la page d'accueil **index.jsp**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "
   http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF
   -8">
<title>TP Struts</title>
</head>
<body>
  <a href="Exemple.action">Exemple</a>
</body>
</html>
```

Cliquez sur **Run Main Project** pour voir le résultat !

3.2 Tags

Les tags de Struts 2 vous permettent d'exploiter la puissance de Struts dans un fichier JSP. Nous allons voir un peu plus en détail ce que sont les tags, et comment les utiliser, mais avant tout nous allons répondre à une question qu'une bonne partie d'entre vous doit certainement se poser[5] : qu'est-ce au juste qu'un fichier JSP ?

Le JSP, pour Java Server Page, est un genre de page web. Tout comme une page PHP permet à un script PHP d'être exécuté par le serveur à l'envoi de la page et d'ajouter à celle-ci du texte (ou des balises, ou même d'écrire du code d'un autre langage type JavaScript), une page JSP permet au serveur d'exécuter du code Java ou d'analyser certain type de tags (rappelez-vous de *taglib* pour réaliser des traitements avant l'envoi d'une page au client.

Il est important, si vous ne connaissez pas bien le développement web, que :

- Vous compreniez que le code Java est exécuté par le **serveur** ;
- Il ne reste pour le client **que** du HTML (et éventuellement du JavaScript ou un autre script client que vous auriez ajouté vous-même) ;
- Vous nous posiez des questions pendant le TP ;).

3.2.1 Utiliser les tags Struts 2 dans une vue

Avant tout, il est nécessaire pour utiliser les tags dans une vue d'ajouter ces deux lignes en haut de la page :

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ taglib prefix="sx" uri="/struts-dojo-tags" %>
```

La première ligne définit le préfixe pour la bibliothèque struts-tag, la seconde pour la bibliothèque de tags Ajax. Les prefixes permettent au serveur de savoir à quelle *taglib* associer un tag.

3.2.2 Tags de données

Les tags de données sont utilisés pour manipuler ou créer des données. Voici la liste des tags de données :

* a
* action
* bean
* date
* debug
* i18n
* include
* param
* push
* set
* text
* url
* property

[5] Non, pas "quand est-ce qu'on mange ?"…

3.2.3 Tags de contrôle

Les tags de contrôle sont utilisés comme des structures de contrôles, comme `if else` ou `iterate`. Avec eux, vous pouvez facilement modifier le contenu de votre page selon des conditions (ce qui n'est pas possible en HTML pur), intérer des collections pour afficher, par exemple, des listes, etc.

Voici la liste des tags de contrôle :

* `if`
* `elseIf`
* `else`
* `append`
* `generator`
* `iterator`
* `merge`
* `sort`
* `subset`

3.2.4 Tags UI

Les Tags UI sont conçus pour exploiter les données fournies par les actions ou les tags de données, et génèrent du code sur les pages HTML, par exemple pour afficher et gérer des composants particuliers, chose très appréciée par les utilisateurs (calendriers...).

Voici la liste des tags UI :

* `actionerror`
* `actionmessage`
* `autocompleter`
* `checkbox`
* `checkboxlist`
* `combobox`
* `component`
* `datetimepicker`
* `div`
* `doubleselect`
* `head`
* `fielderror`
* `file`
* `form`
* `hidden`
* `label`
* `optiontransferselect`
* `optgroup`
* `password`
* `radio`
* `reset`
* `select`
* `submit`
* `table`
* `tabbedPanel`
* `textarea`
* `textfield`

* token
* tree
* treenode
* updownselect

3.2.5 Tags Ajax

* a
* autocompleter
* bind
* datetimepicker
* div
* head
* submit
* tabbedpanel
* textarea
* tree
* treenode

3.3 Créer un formulaire de login en utilisant les tags

Nous allons utiliser quelques tags pour construire un petit module d'authentification : une page de connexion qui redirige vers une page cible. La page cible affichera deux résultats différents en fonction du statut de connexion.

3.3.1 LoginAction and TargetAction

D'abord, nous allons créer deux actions : ajoutez les fichiers suivants dans le package **com.yaka.struts.action** :

- com.yaka.struts.action.LoginAction.java
- com.yaka.struts.action.TargetAction.java

Insérer ce code dans le fichier **LoginAction** :

```
package com.yaka.struts.action;

import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport
{

    private static final long serialVersionUID = 1L;

    @Override
    public String execute() throws Exception
    {
        return SUCCESS;
    }

}
```

Insérer ce code dans le fichier **TargetAction** :

```
package com.yaka.struts.action;

import com.opensymphony.xwork2.ActionSupport;

public class TargetAction extends ActionSupport
{

  private static final long serialVersionUID = 1L;
  private String login;
  private String password;
  private boolean logged;

        @Override
        public String execute() throws Exception
        {
         if (login.equals("yaka") && password.equals("secret"))
        setLogged(true);
         else
        setLogged(false);
           return SUCCESS;
        }

  public String getLogin()
        {
                return login;
  }
  public void setLogin(String login)
        {
    this.login = login;
  }
  public String getPassword()
        {
    return password;
  }
  public void setPassword(String password)
        {
    this.password = password;
  }
  public void setLogged(boolean isLogged)
        {
    this.logged = isLogged;
  }
  public boolean isLogged()
        {
    return logged;
  }
}
```

Nous allons créer deux champs **login** et **password**. Ainsi, dans la cible, nous devrons créer deux attributs nommés `login` et `password` (sans oublier d'implémenter les getters et les setters). Quand le formulaire sera validé, Struts assignera les valeurs des champs dans les attributs automagiquement, en utilisant les getters et setters. Lorsque la variable **logged** contiendra *true*,

la connexion sera considérée comme valide, et uniquement dans ce cas. Nous utiliserons cette variable dans la page **target.jsp**.

3.3.2 login.jsp and target.jsp

Une fois que nos actions sont créées, nous devons leur associer des vues. Le fichier login.jsp contiendra le formulaire de login et la page cible contiendra un test, en utilisant le tag **if** afin de vérifier l'état de la variable **logged**.

Ajoutez les deux pages suivantes :

- login.jsp
- target.jsp

Insérer ce code dans le fichier **login.jsp** :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ taglib prefix="sx" uri="/struts-dojo-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "
   http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF
   -8">
<title>Connexion</title>
<sx:head/>
</head>
<body>
<h1>Please login</h1>
<s:form method="post" validate="true" action="Target" namespace
   ="/">
    <s:textfield label="Login"  name="login"></s:textfield>
    <s:password label="Mot de passe" name="password"></s:
       password>
    <s:submit value="Connexion"></s:submit>
</s:form>
</body>
</html>
```

Insérer ce code dans le fichier **target.jsp** :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ taglib prefix="sx" uri="/struts-dojo-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "
   http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF
   -8">
<title>Acces restreint</title>
<sx:head/>
```

```
</head>
<body>
<h1>Acces restreint</h1>
<s:if test="logged == true">
  <p> Bienvenue <s:property value="login"/>, merci de vous etre
     connecte.</p>
</s:if>
<s:else>
  <p>Erreur de login ! Mauvais login ou mot de passe.</p>
  <a href="Login.action">Retour</a>
</s:else>
</body>
</html>
```

3.3.3 Mapping

Comme pour l'action ExempleAction, il faut maintenant que nous mappions l'action **Login** à la page **login.jsp** et l'action **Target** à la page **target.jsp**.
Ouvrez le fichier **struts.xml** et ajoutez les lignes suivantes dans le package **default** :

```
<action name="Login" class="com.yaka.struts.action.LoginAction
   ">
    <result>/login.jsp</result>
</action>
<action name="Target" class="com.yaka.struts.action.
   TargetAction">
    <result name="input">/login.jsp</result>
    <result>/target.jsp</result>
</action>
```

Remarquez ici l'ajout de l'attribut *name* au nœud result, qui permet de changer la vue retournée selon le résultat de l'action. Il est aussi possible de définir des résultats globaux (pour gérer les erreurs par exemple) plus haut dans l'arborescence.

Attention : L'action **Target** a besoin de connaître la page source pour récupérer correctement les données.
Vous devez maintenant pouvoir tester votre formulaire !

3.4 Validation de formulaire automatique

Il est possible de configurer Struts pour qu'il procède automatiquement à la vérification et à la validation des formulaires. En l'occurence, on désire vérifier si les champs de login et de mot de passe sont renseignés avant d'envoyer le formulaire. Rien de plus simple : il suffit de créer un fichier XML dans le même dossier que la classe **TargetAction.java** qui définira les règles de validation pour chaque champ.

Créez le fichier *TargetAction-validation.xml* dans le package **com.yaka.struts.action** et insérez-y le code suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
```

```
        <field name="login">
            <field-validator type="requiredstring">
                <message>Vous devez entrer un login.</message>
            </field-validator>
        </field>
        <field name="password">
            <field-validator type="requiredstring">
                <message>Vous devez entrer un mot de passe.</message
                   >
            </field-validator>
        </field>
    </validators>
```

Les champs sont désormais vérifiés par Struts, en l'occurence en vérifiant s'ils sont remplis (*requiredstring*). Vous pouvez obtenir un aperçu des validateurs existants à cette adresse : <http://struts.apache.org/2.0.11.2/docs/validation.html>

4 Exercices

Maintenant que vous savez tout de Struts (/o/), nous allons pouvoir créer une petite interface web pour exploiter le travail effectué lors du TP Hibernate.

4.1 Importer votre projet Hibernate

Etant des étudiants sérieux, vous avez terminé et conservé votre travail[6]. Vous pouvez donc vous en servir pour ce TP. Pour cela, copiez simplement les packages développés hier dans le dossier *src* de votre projet Struts. Vous devez écraser le fichier de configuration généré automatiquement par Netbeans par celui que vous avez écrit avant-hier.

Pour tous les exercices suivants, vous devez mettre un lien sur la page d'accueil.

4.2 Coordinate

4.2.1 ListCoordinateAction

Pour commencer, vous devez afficher toutes les coordonnées enregistrées dans la base de données dans un tableau HTML.

Pour faire cela, créez un package **com.yaka.struts.action.coordinate** et placez-y la classe **ListCoordinateAction.java**. Ensuite, créez le fichier de vue **coordinate/listCoordinate.jsp**.

Pour le mapping, vous devez créer un package **coordinate** avec l'espace de nom **coordinate**.

4.3 Platform

4.3.1 ListPlatformAction

Comme pour l'exercice précedent, créez un package **com.yaka.struts.action.platform** et placez-y la classe **ListPlatformAction.java**. Ensuite, créez le fichier de vue **platform/listPlatform.jsp**.

Pour le mapping, vous devez créer un package **platform** avec l'espace de nom **platform**

Un petit indice pour faire cela rapidement ? *s:iterator*.

[6] Non ?

4.3.2 AddPlatformAction & SavePlatformAction

Créez maintenant un formulaire qui stocke une nouvelle plateforme dans la base de données. Comme pour le formulaire de login précedemment créé, écrivez les actions **AddPlatformAction** et **SavePlatformAction** dans le package **com.yaka.struts.action.platform** et leurs vues respectives **addPlatform.jsp** et **savePlatform.jsp**.

Le formulaire doit être composé des champs suivants :

- **Location*** - String ;
- **Manager*** - String.

Vous devrez procéder à la validation automatique des champs avec Struts.

4.4 Driver

4.4.1 ListDriverAction

Vous savez ce qu'il vous reste à faire, non ? ;)

4.4.2 AddDriverAction and SaveDriverAction

Mêmes consignes que pour l'exercice 4.3.2, avec les paramètres suivants :

- Package : **com.yaka.struts.action.driver** ;
- Actions : **AddDriverAction - SaveDriverAction** ;
- JSP : **addDriver - saveDriver** ;
- Validator : **SaveDriverAction-validation.xml**.

Le formulaire doit être composé des champs suivants :

- **Firstname*** - String ;
- **Lastname*** - String ;
- **Licence** - Integer ;
- **Birthday** - Date ;
- **Phone** - Integer (10) ;
- **Address*** - String ;
- **City*** - String ;
- **Zipcode** - Integer (5) ;
- **Platform** - List.

5 Conclusion

Don't worry, code to find the answer !

I think the problem, to be quite honest with you, is that you've never actually known what the question is[7].

[7] Because you have never read the subject of the exams correctly. No offense.