



Par N. Chenfour

# **Table des Matières**

Chapitre 1. Présentation du framework Struts 2.....	4
Chapitre 2. Réalisation d'une application Struts 2.....	11
Chapitre 3. Actions et résultats.....	21
Chapitre 4. Configuration par Convention.....	42
Chapitre 5. Les Intercepteurs .....	54
Chapitre 6. Bibliothèque de balises Struts 2 .....	66



# Chapitre 1. Présentation du framework Struts 2

## 1.1 Description du framework Struts

**Struts** (<http://struts.apache.org/>), ou encore « **Apache Struts** », est un framework open source géré par le groupe Jakarta de la communauté Apache. Il s'agit de l'une des premières implémentations du modèle MVC2 (MVC préalablement), fournissant dans sa version 2 un contrôleur sous forme d'un filtre et un modèle de raisonnement basé sur les actions.

Il a été créé par Craig McClanahan qui est passé à la fondation Apache en 2000 en sous-partie du projet **Jakarta**.

Le framework Struts se base la technologie des Servlets (d'une manière transparente, plus exactement un filtre) et celle des JSP en les étendant et en donnant accès à des objets améliorant l'approche de ces dernières. Cela débouche sur une meilleure structuration du code d'une application web Java ; ce qui offre ainsi une grande flexibilité du code.

Struts s'est bien démarqué depuis 2006 avec sa version 2 ; résultat d'une fusion entre le projet Struts et le projet WebWork, par le fait qu'il n'impose aucune implémentation d'interface ou extension de classe du framework. Les objets qu'il gère sont des objets Java ordinaires (ou encore des POJO « *Plain Old Java Objects* »). Les classes d'action ainsi réalisées (sauf si le développeur intègre délibérément les API Struts) sont totalement indépendantes du framework.

Par ailleurs, le framework Struts offre une utilisation assez remarquable du design pattern « chaîne de responsabilité » à travers ses intercepteurs qui offre une grande extensibilité du code.

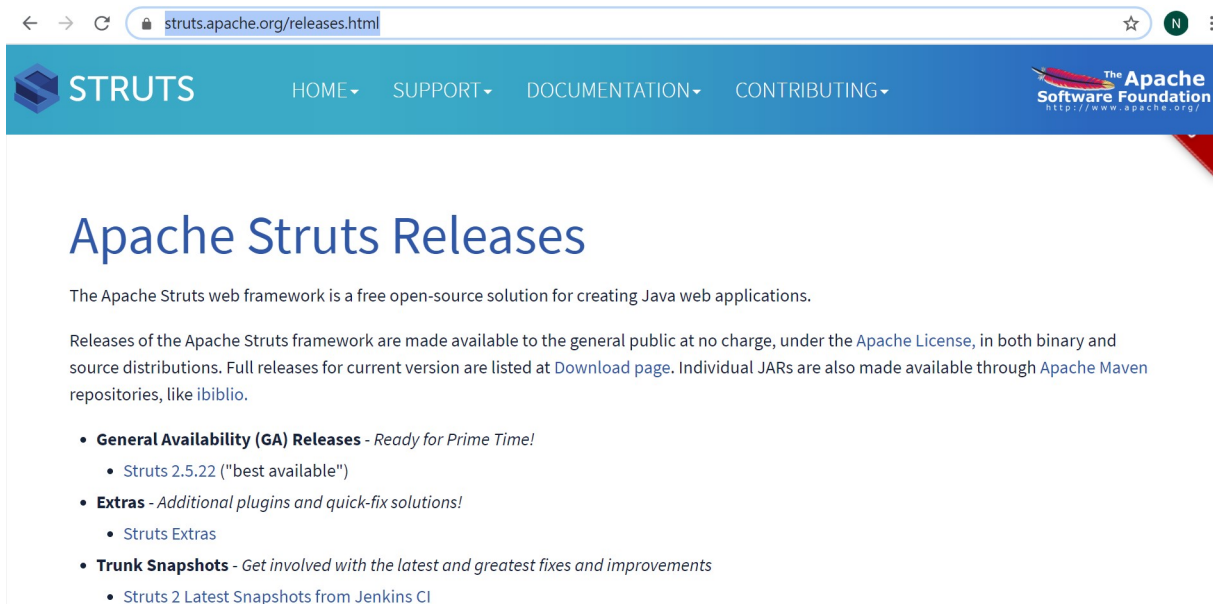
Dans ce document, nous détaillerons les fonctionnalités les plus importantes du framework Struts.

## 1.2 Historique des versions

<b>Version</b>	<b>Date</b>
<b>Struts 2.5.22</b>	<b>29/11/2019</b>
Struts 2.5.20	14/01/2019
Struts 2.5.18	26/09/2018
Struts 2.5.17	22/08/2018
Struts 2.5.16	16/03/2018
Struts 2.5.14	23/11/2017
Struts 2.5.13	05/09/2017
Struts 2.5.12	06/07/2017
Struts 2.5.10	03/02/2017
Struts 2.5.8	19/12/2016
Struts 2.5.5	21/10/2016
Struts 2.5.2	07/07/2016
Struts 2.5.1	18/06/2016
<b>Struts 2.5</b>	<b>09/05/2016</b>
<b>Struts 2.3.1</b>	<b>12/12/2011</b>
Struts 2.2.3	07/09/2011
<b>Struts 2.2.1</b>	<b>16/08/2010</b>
Struts 2.1.6	05/01/2009
<b>Struts 2.0.6</b>	<b>22/02/2007</b>
Struts 2.0.5 Beta	09/02/2007
<b>Struts 2.0.1</b>	<b>10/10/2006</b>
Struts 1.3.5 Beta	18/08/2006
Struts 1.2.9	22/03/2006
Struts 1.2.8	25/11/2005
Struts 1.2.7	26/05/2005
Struts 1.2.4	19/09/2004
Struts 1.2.2	31/08/2004
Struts 1.1	30/06/2003
<b>Struts 1.0</b>	<b>15/06/2001</b>

On peut trouver plus détails sur l'historique des « releases » à travers le lien suivant :

<https://struts.apache.org/releases.html>



The screenshot shows the Apache Struts Releases page. The browser address bar displays `struts.apache.org/releases.html`. The page header includes the Struts logo and navigation links: HOME, SUPPORT, DOCUMENTATION, and CONTRIBUTING. The main heading is "Apache Struts Releases". Below it, a paragraph states: "The Apache Struts web framework is a free open-source solution for creating Java web applications." Another paragraph explains: "Releases of the Apache Struts framework are made available to the general public at no charge, under the Apache License, in both binary and source distributions. Full releases for current version are listed at Download page. Individual JARs are also made available through Apache Maven repositories, like ibiblio." A bulleted list follows:

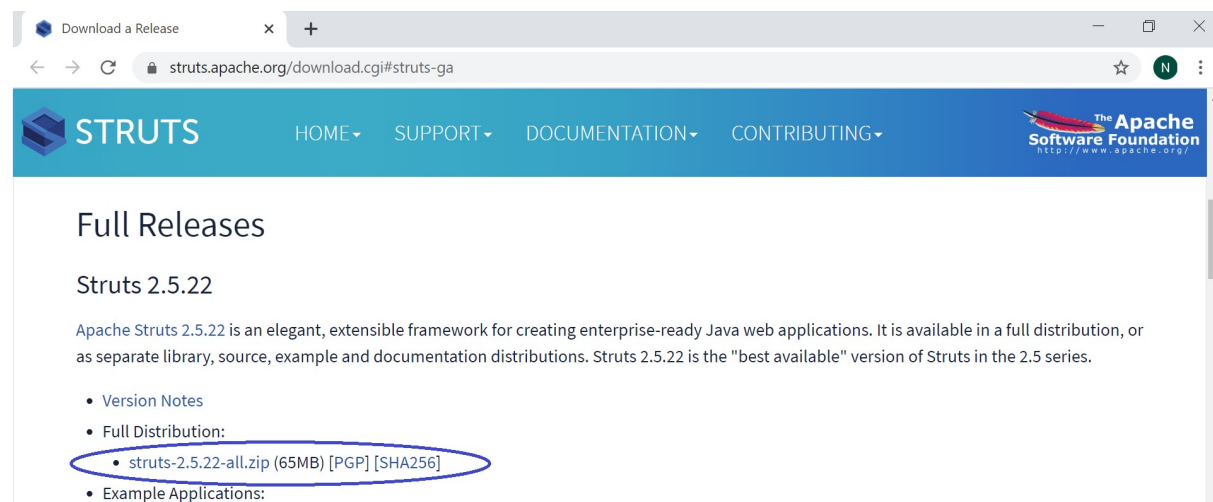
- **General Availability (GA) Releases** - Ready for Prime Time!
  - Struts 2.5.22 ("best available")
- **Extras** - Additional plugins and quick-fix solutions!
  - Struts Extras
- **Trunk Snapshots** - Get involved with the latest and greatest fixes and improvements
  - Struts 2 Latest Snapshots from Jenkins CI

On y trouve aussi la dernière version à télécharger :

Struts 2.5.22

**Attention** à la compatibilité des versions avec le JDK. Par exemple, certaines version antérieures (Struts 2.5.13) ne sont pas compatibles avec le JDK 10 et ultérieurs. La version **2.5.13** fonctionne avec le **JDK 1.8** !! et présente des Bugs de fonctionnement avec le **JDK 10**.

Télécharger la version Struts 2.5.22 sur le même lien, on obtient la page suivante :




The screenshot shows the Apache Struts Download page. The browser address bar displays `struts.apache.org/download.cgi#struts-ga`. The page header is identical to the previous screenshot. The main heading is "Full Releases". Below it, the heading "Struts 2.5.22" is followed by a paragraph: "Apache Struts 2.5.22 is an elegant, extensible framework for creating enterprise-ready Java web applications. It is available in a full distribution, or as separate library, source, example and documentation distributions. Struts 2.5.22 is the 'best available' version of Struts in the 2.5 series." A bulleted list follows:

- Version Notes
- Full Distribution:
  - **struts-2.5.22-all.zip (65MB) [PGP] [SHA256]**
- Example Applications:

## 1.3 Installation de Struts 2














Struts est livré sous forme d'un fichier zip installable (par décompression) sous forme de plusieurs répertoires et librairies jar

 struts-2.5.22-all.zip	64 785 Ko
---	-----------










Depuis la version 2.0, le framework a apporté plusieurs modifications fonctionnelles et la structure de ses packages a aussi été nettement révisée.

struts-2.5.22	
apps	
docs	
lib	
src	

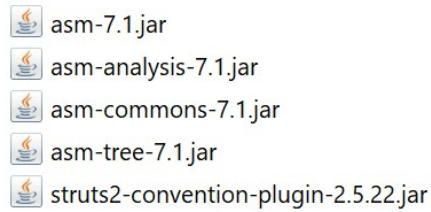
  

Nom	Modifié le
 asm-7.1.jar	18/10/2019 20:19
 asm-analysis-7.1.jar	18/10/2019 20:19
 asm-commons-7.1.jar	18/10/2019 20:19
 asm-tree-7.1.jar	18/10/2019 20:19
 bsh-2.0b4.jar	18/10/2017 15:50
 classworlds-1.1.jar	18/10/2017 16:29
 commons-beanutils-1.9.4.jar	06/09/2019 09:37
 commons-collections-3.2.2.jar	05/05/2016 09:36
 commons-digester-2.1.jar	18/10/2017 16:25
 commons-fileupload-1.4.jar	29/12/2018 16:58
 commons-io-2.6.jar	26/02/2018 08:44
 commons-lang-2.4.jar	18/10/2017 19:16
 commons-lang3-3.8.1.jar	19/10/2018 09:32

91 librairies sont disponibles dans le répertoire lib ; mais il est nécessaire de remarquer que pour réaliser des applications Struts ordinaire, on n'a pas besoin de toutes ces librairies, nous pouvons faire la sélection suivante :

-  commons-fileupload-1.4.jar
-  commons-io-2.6.jar
-  commons-lang3-3.8.1.jar
-  commons-logging-1.2.jar
-  freemarker-2.3.28.jar
-  javassist-3.20.0-GA.jar
-  log4j-api-2.12.1.jar
-  ognl-3.1.26.jar
-  struts2-core-2.5.22.jar

On peut aussi ajouter les librairies suivantes pour activer la configuration par convention et par annotations :



On pourra remarquer la grande richesse des librairies et une ouverture à plusieurs autres frameworks et technologies dont on peut citer :

- Spring
- JSF
- Portlets
- Logging
- JUnit
- JSON
- ...



La bibliothèque « **struts2-convention-plugin-2.5.22.jar** » ne sera nécessaire que si on utilise la « **Convention Plugin** » et les **annotations** comme politique de configuration, au lieu de struts.xml (voir plus loin pour plus de détail : Chapitre « Convention Plugin et Annotations »).



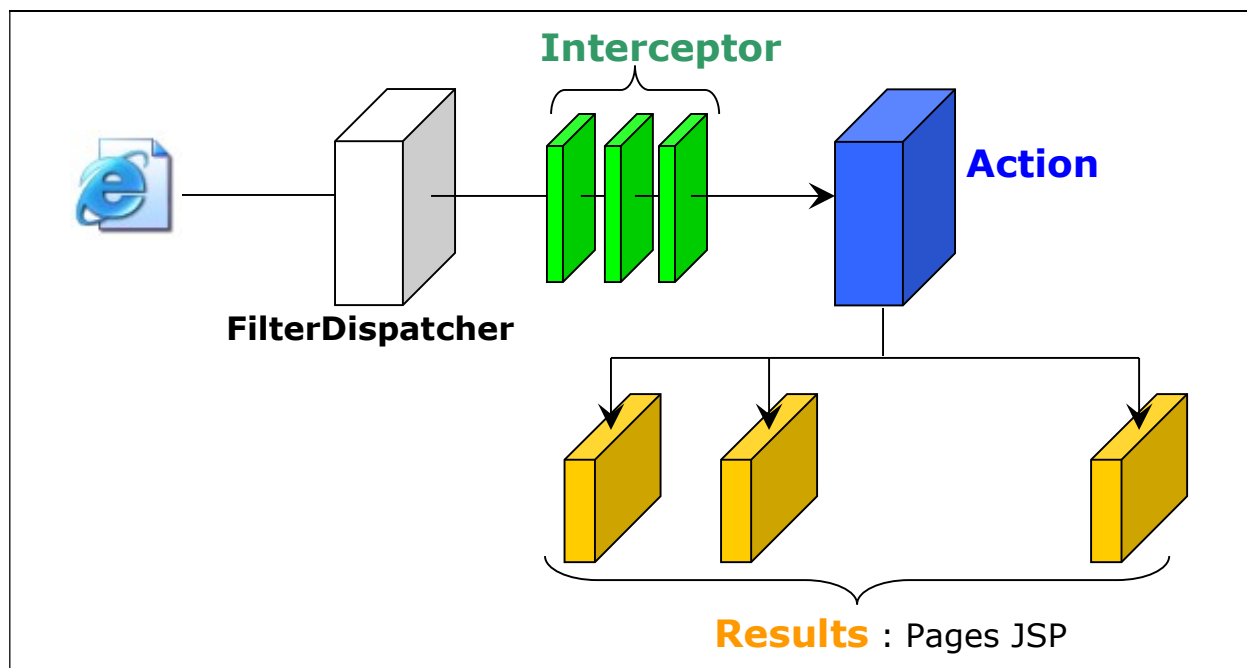
## 1.4 Caractéristiques générales du Framework Struts 2

- Support du développement web Java EE : Servlets et JSP.
- Favorisation des POJO (Plain Old Java Objects)
- Configuration Libre basée sur XML
- Une base très puissante de Design Patterns : MVC2, IoC, Interceptors, ....
- Une grande bibliothèque d'APIs
- Des balises utiles pour la réalisation des vues
- support de la programmation orientée Aspects (AOP)
- Internationalisation
- Support du Web 2.0 pour des interfaces web riches et interactives
- Journalisation automatique « logging »
- Intégration avec le framework Spring.
- Favorisation des tests unitaires (Développement piloté par les tests : TDD)



# Chapitre 2. Réalisation d'une application Struts 2

## 2.1 Principe de base d'une application Struts2



### Interceptors

Ce sont des objets de classes permettant d'intercepter la requête pour effectuer des pré/post traitements sur celle-ci. Les pré-traitements sont plus fréquents (d'où notre schéma). Exp : intercepter les données d'un formulaire et les injecter d'une manière appropriée dans l'action.

### Action

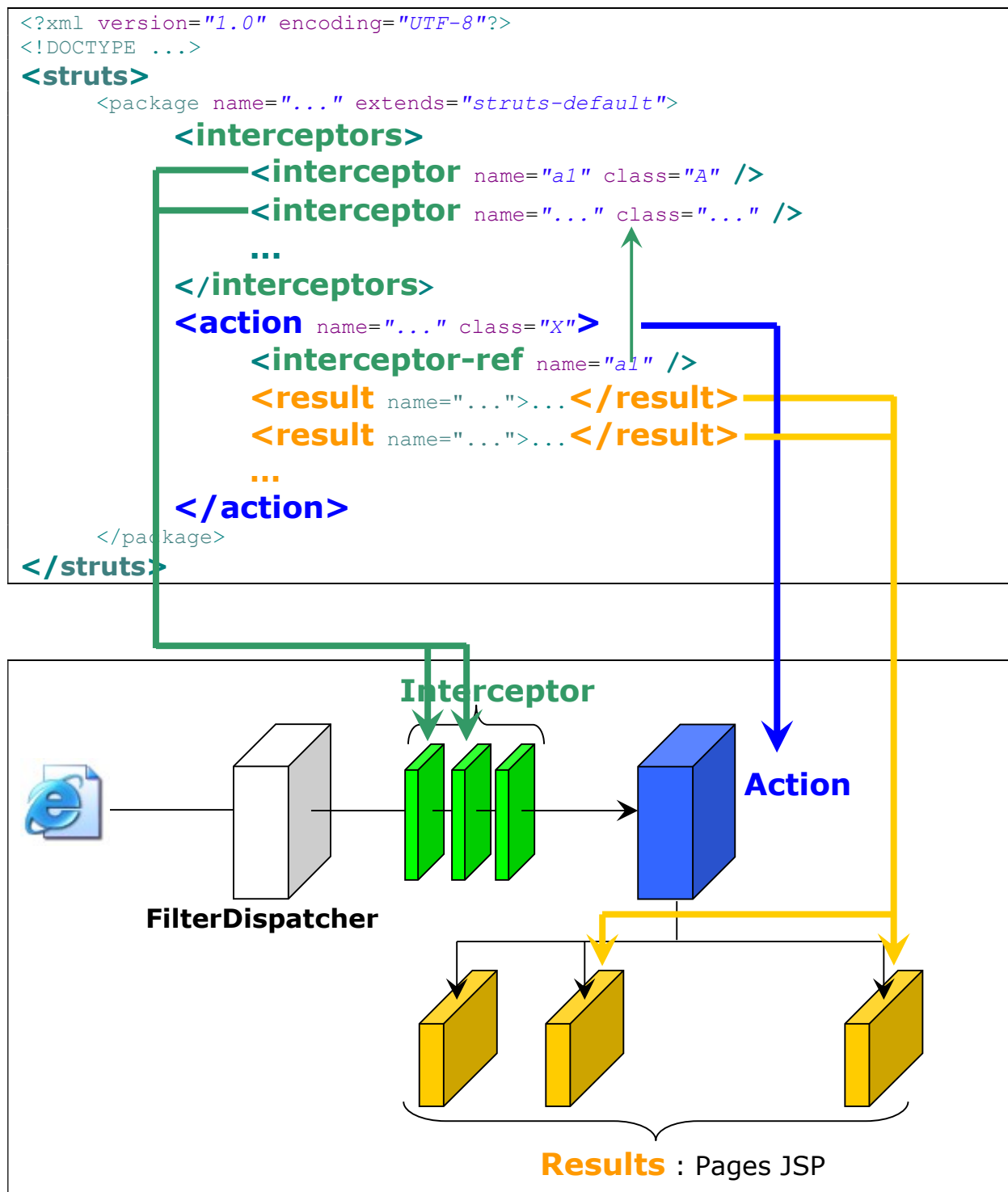
C'est dans cette classe que se passe l'action de traitement d'une requête donnée. Avec la possibilité (mais pas obligatoirement) d'une Action différente par requête différente.

### Results

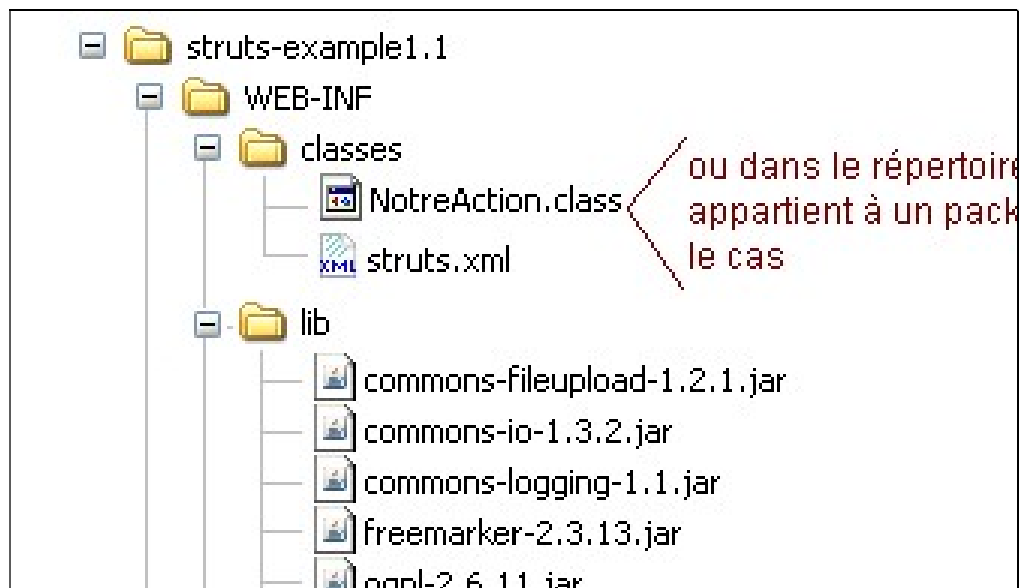
Ce sont les pages JSP qui se chargeront de l'affichage du résultat. Elles ont accès à l'action et à tous les autres objets habituels : session, in, out, ...

## 2.2 Configuration habituelle

Pour la configuration du contexte, une première solution consiste à utiliser un fichier « **struts.xml** » qui devrait être définie dans la racine de l'application : soit dans le répertoire « src » sous eclipse et répertoire classes sous Tomcat.



## 2.3 Structure arborescente d'une application Struts 2



### 1. Le descripteur de déploiement « web.xml »

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <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>
</web-app>
```

- Les éléments les plus importants dans le fichier « **web.xml** » sont **<filter>** et **<filter-mapping>** utilisés pour la configuration du **FilterDispatcher**.
- Le **FilterDispatcher** est la **servlet** de base du framework Struts 2, il permet de traiter toutes les requêtes en entrée, il permet l'accès aux éléments de base (ConfigurationManager, ActionMapper, ObjectFactory ...) du framework pour traiter les requêtes.

## 2. Le fichier de configuration « struts.xml »

```
<?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="nomDuPackage" extends="struts-default">
        <action name="NotreAction" class="NotreAction">
            <result>NotreResult.jsp</result>
        </action>
    </package>
</struts>
```

- Dans cet exemple, il n'y a pas d'interceptor.
- Le nom du package est arbitraire
- Cette configuration signifie : après exécution de l'action « NotreAction » (en « success ») passer la main à la JSP : « NotreResult.jsp »

## 3. L'action « NotreAction.java »

```
public class NotreAction {
    private String unePropriete;

    public String execute() {
        unePropriete = "Framework Struts2 Test";
        return "success";
    }

    public String getUnePropriete() {
        return unePropriete;
    }
}
```

- On remarque qu'il s'agit d'une classe POJO.
- La classe d'action doit comporter obligatoirement (et au moins) une **méthode d'action** (sans paramètre et retournant un String). Par défaut la méthode doit avoir le nom « **execute()** » et retourne obligatoirement (par défaut) la chaîne « **success** ».
- Il est possible de donner un autre nom à la **méthode d'action**, et qu'elle retourne toute autre chaîne à condition d'associer la configuration adéquate dans le fichier « **struts.xml** » (voir plus loin).

#### 4. La vue « NotreResult.jsp »

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>Premier Exemple Struts 2 :
    <s:property value="unePropriete" />
    </h1>
</body>
</html>
```

- La JSP "Result" a la possibilité d'accéder à l'action et donc à ses « getters » pour récupérer les données calculées dans celle-ci. La balise <s:property ...> donne ainsi accès à la propriété « unePropriete » par l'intermédiaire de son accesseur : « getUnePropriete ».
- La balise <s:property ...> fait partie d'une multitude de balises de la bibliothèque de balises Struts 2.
- Il serait possible d'accéder aux propriétés de l'action en utilisant la méthode classique de passage par l'objet **request** de la classe « **HttpServletRequest** » (voir plus loin)

## 2.4 Développement d'une application Struts 2 avec Eclipse

### Etape N° 1 :

Créer un projet web dynamique « Dynamic Web Project »

### Etape N° 2 :

Ajouter les lignes marquées au descripteur de déploiement

« web.xml » :

```
<?xml version="1.0" encoding="UTF-8"?>

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

    <display-name>struts2.2</display-name>

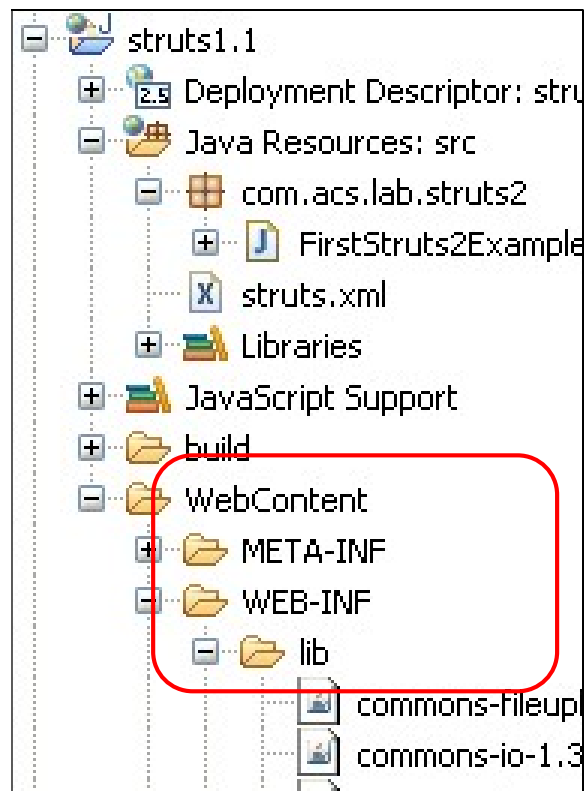
    <filter>
        <filter-name>Struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```



### Etape N° 3 :

Ajouter les librairies utiles au répertoire « lib » de « WEB-INF »



### Etape N° 4 :

Créer le fichier de configuration « **struts.xml** » dans le répertoire « **src** » avec 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="struts2.2" extends="struts-default">
        <action
            name="FirstStruts2Example"
            class="com.technos.labs.struts2.FirstStruts2Example"
            >

            <result>FirstStruts2Example.jsp</result>

        </action>
    </package>
</struts>
```

## Etape N° 5 :

Créer l'action « FirstStruts2Example.java » :

```
package com.technos.labs.struts2;

public class FirstStruts2Example {
    private String name;

    public String execute() {
        setName("Framework Struts2");
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

## Etape N° 6 :

Créer la vue :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <body>
        <h1>Premier Exemple Struts 2 :
        <s:property value="name" /></h1>
    </body>
</html>
```

## Etape N° 7 :

Enfin, on pourra créer un fichier index :

```
<html>
<body>
  <h2>Liste d'exemples Struts 2</h2>
  <a href="FirstStruts2Example">
    Premier Exemple Struts 2
  </a><br/>
</body>
</html>
```

## Remarque :

Il est aussi possible d'utiliser le plugin « **Alveole Studio MVC Web Project** » qui est à peine à sa version « 0.7.4 » (Novembre 2010) ; téléchargeable depuis eclipse, avec l'URL de mise à jour ci-dessous, à l'aide de l'option « Help/Software updates/Find and Install... »

**<http://mvcwebproject.sourceforge.net/update/>**



# Chapitre 3. Actions et résultats

## 3.1 Structure du fichier de configuration « struts.xml »

Le fichier « **struts.xml** » permet de déclarer l'ensemble des entités constituant l'application web et de définir les relations entre elles. Dans ces formes les plus complètes, le fichier « struts.xml » a la structure suivante :

```
<?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>
    <constant name="..." value="..." />
    ...
    <bean class="..." name="..." scope="..." type="..." />
    ...
    <include file="..." />
    ...
    <package name="..." extends="struts-default" namespace="...">
        <interceptors>
            <interceptor name="..." class="...">
                <param name="...">...</param>
            </interceptor>
            ...
            <interceptor-stack name="...">
                <interceptor-ref name="..." />
            </interceptor-stack>
            ...
        </interceptors>
        <action name="..." class="..." method="...">
            <param name="...">...</param>
            <interceptor-ref name="..." />
            ...
            <result name="..." type="...">...</result>
            ...
        </action>
    </package>
    ...
</struts>
```

On pourra constater que le fichier contient généralement 4 blocs possibles :

- **constant**
- **bean**
- **include**
- **package**

Le bloc package est le plus important et il permet de déclarer 3 types d'entités :

- **interceptor**
- **action**
- **result**

Aussi nous débutons par les packages :

### 3.1 Les packages

Les packages offrent un moyen pour regrouper les actions, les résultats (results), et les intercepteurs (interceptors) dans une unité de configuration logique. Le package pourra être étendu par la suite ce qui donnera au nouveau package la possibilité d'accéder aux intercepteurs et aux autres paramètres de configuration définis dans le package père.

L'élément **<package>** a un seul attribut obligatoire « **name** » permettant de l'identifier en vue de le réutiliser (l'étendre par exemple). Les autres attributs sont optionnels :

Attribut	Description
<b>name</b>	identificateur du package permettant de le référencer par les packages définis plus bas.
<b>extends</b>	Hérite des éléments d'un package défini plus haut et dont le nom (name) sera fournie comme valeur de celui-ci
<b>namespace</b>	La valeur du namespace devra préfixer le nom de l'action dans l'URL : http://.../LeNameSpace/L'Action
<b>abstract</b>	Si le package est déclaré abstrait, alors pas de définition d'actions (seulement des intercepteurs)

### Exemple :

```
<struts>
  <package name="Struts2-Exemple01" extends="struts-default">
    ...
  </package>
</struts>
```

On Remarquera le besoin d'étendre le package par défaut « **struts-default** » (se trouvant dans le fichier de configuration par défaut « struts-default.xml » de l'API « struts2-core-2.2.1.jar ») pour pouvoir hériter de la configuration par défaut de struts (intercepteurs et autre).

### 📁 Les namespaces

La notion de namespace permet de rassembler les actions en modules logiques contenant chacun un paquet d'actions accessibles par l'intermédiaire d'un préfixe différent qui est la valeur du namespace :

<http://.../LeNameSpace/L'Action>

Les actions d'un même module auront le même préfixe.

### Exemple :

```
<struts>
  <package
    name="nomDuPackage"
    extends="struts-default"
    namespace="/NS1"
  >
    <action name="NotreAction" ...>
      ...
    </action>
    ...
  </package>
</struts>
```

L'accès à l'action « **NotreAction** » sera fait à l'aide de l'URL suivante :

<http://localhost:8080/exemple1.0/NS1/NotreAction>

En supposant que Tomcat tourne sur la machine locale et écoute sur le port 8080 ; Et que le nom du contexte est « *exemple1.0* ».

## 3.2 Les Actions

Les actions sont configurées dans le fichier struts.xml et définies sous forme de classes dans le répertoire « WEB-INF/classes » :

### 🚩 3.2.1 Configuration d'une action

```
<struts>
  <package name="Struts2-Exemple01" extends="struts-default">

    <action
      name="FirstStruts2Example"
      class="com.technos.labs.struts2.FirstStruts2Example"
    >

      <result>FirstStruts2Example.jsp</result>

    </action>

  </package>
</struts>
```

Cette configuration simple signifie :

- Suite à chaque nouvelle requête : **"FirstStruts2Example"**
- Instancier un objet de la classe :  
**com.technos.labs.struts2.FirstStruts2Example**
- Appeler la méthode d'action « **public String execute()** » de l'objet instancié.
- Donner la main à la vue « **FirstStruts2Example.jsp** » si la méthode « **execute** » retourne la chaîne **"success"**.

### 🚩 3.2.2 Définition de l'action

La classe d'action est dans ce cas définie comme suit :

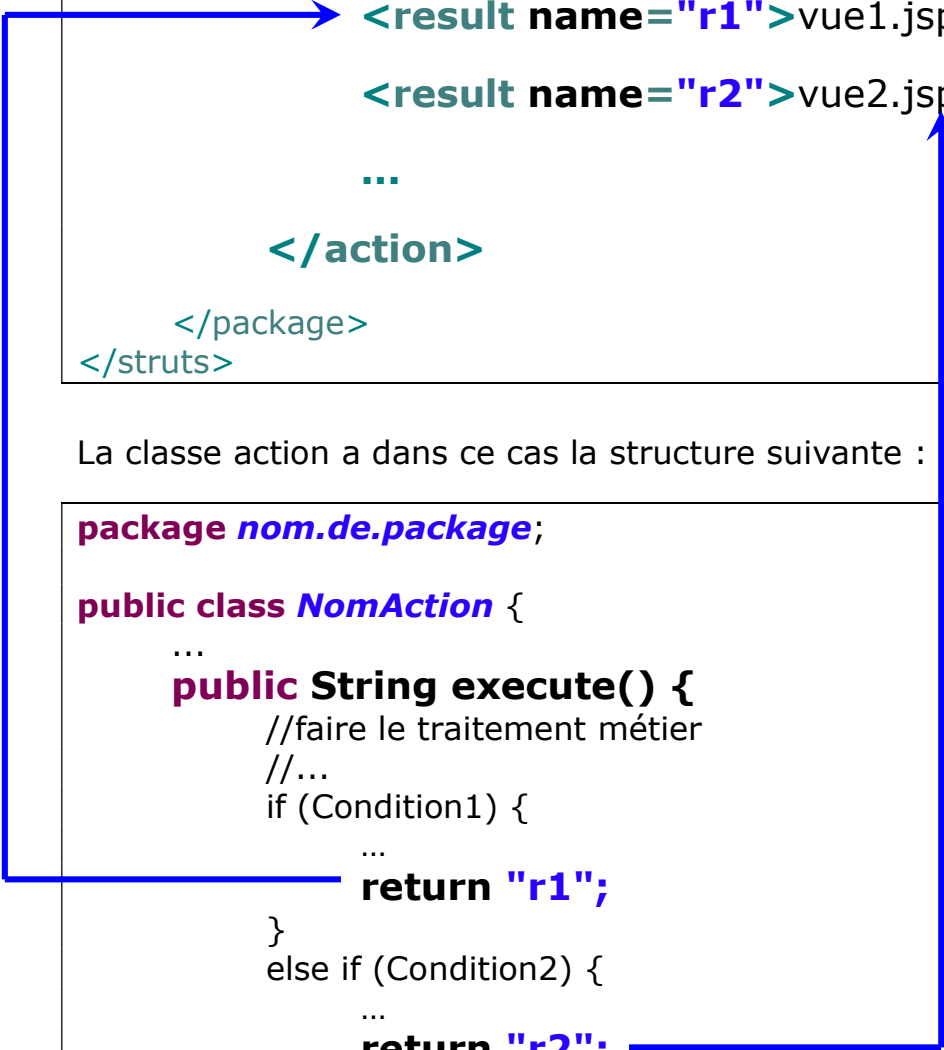
```
package com.technos.labs.struts2;

public class FirstStruts2Example {
    ...
    public String execute() {
        //faire le traitement metier
        //...
        return "success";
    }
    ...
}
```



### 🚩 3.2.3 Configuration des vues

```
<struts>
  <package name="Struts2-Exemple01" extends="struts-default">
    <action
      name="NomAction"
      class="nom.de.package.NomDeClasseAction"
    >
      <result name="r1">vue1.jsp</result>
      <result name="r2">vue2.jsp</result>
      ...
    </action>
  </package>
</struts>
```



La classe action a dans ce cas la structure suivante :

```
package nom.de.package;

public class NomAction {
  ...
  public String execute() {
    //faire le traitement métier
    //...
    if (Condition1) {
      ...
      return "r1";
    }
    else if (Condition2) {
      ...
      return "r2";
    }
    ...
  }
  ...
}
```

### 🚩 3.2.4 Méthodes d'action

```
<struts>
  <package name=" Struts2-Exemple02" extends="struts-default">
    <action
      name="NomAction1"
      class="nom.de.package.NomDeClasseAction"
      method="methode1"
    >
      <result>vue1.jsp</result>
    </action>
    <action
      name="NomAction2"
      class="nom.de.package.NomDeClasseAction"
      method="methode2"
    >
      <result>vue2.jsp</result>
    </action>
    ...
  </package>
</struts>
```

La classe action a dans ce cas la structure suivante :

```
package nom.de.package;

public class NomAction {
  ...
  public String methode1() {
    //faire le traitement métier
    //...
    return "success";
  }

  public String methode2() {
    //faire le traitement métier
    //...
    return "success";
  }
  ...
}
```

### 🚩 3.2.5 Méthodes génériques (Wildcard Methods)

Plusieurs actions peuvent partager un modèle de configuration commun. Par exemple l'action « **ajouterClient** » appelle la méthode **ajouter()** de la classe d'action **Client**. L'action « **supprimerClient** » appelle la méthode **supprimer()** de la même classe d'action **Client**, etc.

#### **Solution : mapping générique (Wildcard Mapping)**

```
<action
  name="*Client"
  class="com.tehnos.labs.struts2.Client"
  method="{1}"
>
...
</action>
```

Dans ce cas, faire référence au nom d'action "**ajouterClient**" entraînera l'exécution de la méthode « **ajouter()** » sur l'instance de la classe d'Action Client. De la même manière, faire référence au nom d'action "**supprimerClient**" entraînera l'exécution de la méthode « **supprimer()** » sur l'instance de la classe d'Action Client.

#### **Remarque :**

Le nom du résultat (la vue) peut aussi utiliser la notation "**{1}**" pour faire le mapping.

#### **Exemple :**

```
<result>{1}Response.jsp</result>
```

Dans ce cas, l'exécution de la méthode « **ajouter** » par exemple ce terminera par l'invocation de la JSP : **ajouterResponse.jsp**

Une autre possibilité pour réaliser le « Wildcard Mapping » consiste à postfixer le nom de l'action par le nom de la méthode (au lieu de le préfixer comme dans la méthode précédente). Dans ce cas, il serait nécessaire de séparer le nom de la méthode d'action, représenté par le symbole \*, par un blanc souligné par rapport au mot de début.

```
<action
  name="Client_*"
  class="com.technos.labs.struts2.Client"
  method="{1}"
>
  ...
</action>
```

### 👉 3.2.5 Actions génériques par défaut (Wildcard Default)

La technique du « wildcard » peut être utilisée pour donner une liberté de programmer l'exécution de certaines JSPs sans définition propre à chaque JSP. Une définition générique par défaut pourra être programmée de la manière suivante :

```
<action name="*">
  <result>/ {1}.jsp</result>
</action>
```

#### **Remarques :**

1. Un appel à l'url suivante par exemple : le lien "**Test**" exécutera la méthode « **execute()** » de la classe d'action par défaut « **ActionSupport** » du package « **com.opensymphony.xwork2** » du framework struts, ensuite il y aura exécution de la JSP « **Test.jsp** ».
2. On remarque que dans le cas général (pas de Wildcard), si l'attribut « **class** » de la balise « **action** » n'est pas précisé, c'est la classe

**ActionSupport** qui jouera le rôle de l'action par l'intermédiaire de sa méthode « **execute()** ».

3. On remarque aussi que la classe **ActionSupport** peut être utilisée comme classe mère des actions utilisateur, dans ce cas on héritera de quelques possibilités offertes par celle-ci dont la méthode « **execute()** » qui retourne la chaîne "**success**", ainsi que la méthode « **input()** » qui retourne la chaîne "**input**".
4. Afin de définir une autre classe comme la classe d'action par défaut à la place de **ActionSupport**, on ajoutera l'élément suivant comme sous balise de l'élément package :

```
<default-class-ref class="nom.de.la.nouvelle.DefaultActionClasse" />
```

### 👉 3.2.6 Action par défaut

Habituellement, si le framework ne peut pas résoudre un nom d'action (pas de mapping associé), une erreur se déclenche. Il est cependant possible de créer une action par défaut qui sera appelée à chaque fois qu'il n'y a pas de mapping associé. Cette solution est réalisée à l'aide de l'élément **<default-action-ref>** qui devrait figurer une seule fois comme sous balise de l'élément **<package>** :

```
<package name="..." extends="struts-default">

    <default-action-ref name="actionPardefaut" />
    ...
    <action name="actionPardefaut">
        <result>/UnderConstruction.jsp</result>
    </action>
    ...
</package>
```

### 3.3 Les Vues

Ce sont les JSP associées sous forme de « Result » aux différentes actions configurées dans le fichier « struts.xml ». Ceci est réalisé par l'intermédiaire de la sous balise <result> de l'élément <action>

#### **Syntaxe :**

```
<package name="..." extends="struts-default">
    ...
    <action name="..." class="...">
        <result name="..." type="...">
            /NomDeLaJSP.jsp
        </result>
    </action>
    ...
</package>
```

Précéder le nom de la JSP par l'intermédiaire du symbole « / » signifie que la recherche de la JSP est réalisée dans la racine du contexte. Si le symbole « / » n'est pas utilisé, la recherche est effectuée dans le répertoire ayant comme nom celui du namespace (s'il y a un namespace).

#### 👉 **3.3.1 Nom d'un resultat : l'attribut « name »**

L'attribut « **name** » de la balise **<result>** ayant par défaut la valeur « **success** », permet la sélection de la JSP en fonction de la chaîne retournée par la méthode d'action. La liste des chaînes de retour prédéfinies est la suivante :

```
String SUCCESS      = "success";
String NONE          = "none";
String ERROR         = "error";
String INPUT         = "input";
String LOGIN         = "login";
```

Ces constantes sont définies dans l'interface « **Action** » du package «**com.opensymphony.xwork2** ». L'interface **Action** est implémentée par la classe **ActionSupport**.

### 👉 3.3.2 l'attribut « type »

L'attribut « **type** », aussi optionnel, permet de définir le type du résultat. La valeur par défaut est « **dispatcher** » qui signifie de donner la main à la ressource web (la JSP) précisée comme valeur de l'élément <result> (entre <result> et </result>).

Les valeurs suivantes sont possibles (les plus intéressantes seront étudiées plus tard) :

- **Chain**
- **Dispatcher**
- FreeMarker
- HttpHeaders
- **Redirect**
- **Redirect Action**
- Stream
- Velocity
- XSL
- PlainText
- Tiles

### 👉 3.3.3 Paramètres de la balise <result>

L'élément <**result**>, correspondant à une classe se dérivant de l'interface « **Result** », dispose d'une sous balise <**param**> permettant d'injecter des valeurs dans des propriétés de l'objet Result. L'une des propriétés les plus utilisées est la propriété « **location** » qui est déduite automatiquement de la configuration habituelle :

```
<result name=" ... " type=" ... " >/NomDeLaJSP.jsp</result>
```

Qui est donc équivalente à la configuration suivante, utilisant la balise <**param**>:

```
<result name=" ... " type=" ... " >  
    <param name="location">/NomDeLaJSP.jsp</param>  
</result>
```

### 👉 3.3.4 Résultats globaux

```
<package name="..." extends=" struts-default ">
  <global-results>
    <result name="R01">/Page01.jsp</result>
    <result name="R02">/Page02.jsp</result>
    ...
  </global-results>
  ...
</package>
```

### 👉 3.3.5 Chaînage des résultats

```
<package name="p01" extends="struts-default">
  <action name="a01" class="...">
    <result type="chain">a02</result>
  </action>

  <action name="a02" class="...">
    <result type="chain">
      <param name="actionName">a03</param>
      <param name="namespace">/p02</param>
    </result>
  </action>
</package>

<package name="p02" extends="struts-default" namespace="/p02">
  <action name="a03" class="...">
    <result>R03.jsp</result>
  </action>
</package>
```



### 📌 3.3.6 Redirection de resultats (Redirect result)

```
<package name="p01" extends="struts-default" namespace="/p01">
  <action name="a01" class="...">
    <result name="r01" type="redirect">
      <param name="location">r01.jsp</param>
      <param name="p1">valeurP1</param>
      <param name="p2">valeurP2</param>
      ...
    </result>
  </action>
</package>
```

### 🚩 3.3.7 Redirect Action Result

```
<package name="p01" extends="struts-default">
  <action name="a01" class="...">
    <result type="redirectAction">
      <param name="actionName">a02</param>
      <param name="namespace">/p02</param>
    </result>
  </action>
</package>

<package name="p02" extends="struts-default" namespace="/p02">
  <action name="a02" class="...">
    <result>r02.jsp</result>
    <result name="error" type="redirectAction">error</result>
  </action>

  <action name="error" class="...">
    <result>error.jsp</result>
  </action>
</package>
```

```
<package name="p03" extends="struts-default" namespace="/p03">
  <action name="a03" class="...">
    <result name="r01" type="redirect-action">
      <param name="actionName">a04</param>
      <param name="namespace">/p04</param>
      <param name="p01">valeurP1</param>
      <param name="p02">valeurP2</param>
      ...
    </result>
  </action>
</package>
```

### 📌 3.3.8 Résultats dynamiques

```
public class NomAction {  
    private String nextAction;  
    ...  
    public String getNextAction() {  
        return nextAction;  
    }  
    ...  
    public String execute() {  
        ...  
        return "next";  
    }  
}
```

```
<action name="NomAction" class="NomAction">  
    <result name="next" type="redirectAction">${nextAction}</result>  
</action>
```

### 3.4 La balise <constant> :

Struts 2 permet de définir des constantes de configuration (utilisées par le framework) dans le fichier de configuration struts.xml. Exemple : **<constant name="struts.enable.SlashesInActionNames" value="true"/>**. Ces constantes peuvent être aussi définies dans le fichier de configuration struts.properties ou encore dans le descripteur de déploiement web.xml.

Syntaxe :

```
<struts>
    ...
    <constant name="struts.action.extension" value="action" />
    ...
</struts>
```

La même constante pourra être configurée dans le fichier « **struts.properties** » dans le répertoire « **WEB-INF/classes** » :

```
struts.action.extension = action
```

Ou encore dans le fichier « **web.xml** » du répertoire « **WEB-INF** » :

```
<web-app>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    <init-param>
      <param-name>struts.action.extension</param-name>
      <param-value>action</param-value>
    </init-param>
  </filter>
  ...
</web-app>
```

**Remarque :** Par défaut les constantes sont définies dans le fichier default.properties fourni avec le framework struts 2 (voir annexe A).

## 3.5 La balise <bean>

La balise <bean> a un attribut obligatoire "class", permettant ainsi de spécifier la classe depuis laquelle le bean sera instancié. Les autres attributs sont optionnels :

Attribut	Description
<b>class</b>	Le nom de la classe du bean
<b>type</b>	Le nom de l'interface mère dont dérive la classe
<b>name</b>	Un identificateur du bean
<b>scope</b>	La portée du bean. Pouvant prendre les valeurs suivantes : default, singleton, request, session, thread
<b>static</b>	Valeurs possibles : true ou false. Injection via les méthodes statiques. Si type est précisé alors static devrait avoir la valeur false.
<b>optional</b>	Si le bean est optionnel

Voici la liste de quelques beans déclarés dans le fichier de configuration struts « struts-default.xml ». La liste entière est fournie en annexe B.

```
<bean class="com.opensymphony.xwork2.ObjectFactory"
name="xwork" />
<bean type="com.opensymphony.xwork2.ObjectFactory"
name="struts" class="org.apache.struts2.impl.StrutsObjectFactory" />

<bean type="com.opensymphony.xwork2.ActionProxyFactory"
name="xwork"
class="com.opensymphony.xwork2.DefaultActionProxyFactory"/>
<bean type="com.opensymphony.xwork2.ActionProxyFactory"
name="struts"
class="org.apache.struts2.impl.StrutsActionProxyFactory"/>

<bean
type="com.opensymphony.xwork2.conversion.ObjectTypeDeterminer"
name="tiger"
class="com.opensymphony.xwork2.conversion.impl.DefaultObjectTypeDet
erminer"/>
...
```

## 3.6 La balise <include>

L'élément <include> permet d'inclure un autre fichier dans lequel on dispose d'une configuration séparée. Le fichier à inclure devrait avoir la même structure que celle du fichier de configuration « struts.xml ».

Cet élément peut être utilisé autant de fois que nécessaire. Il permettra alors de structurer la configuration d'une application web sous forme de modules décrit chacun dans un fichier séparé (par exemple un package par fichier), plutôt que de mettre la configuration de tous les modules (ou packages) dans le même fichier « struts.xml ».

### **Syntaxe :**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC...>
<struts>

  <include file="module1.xml"/>
  <include file="module2.xml"/>

  <package ...>
    ...
  </package>

</struts>
```

Avec « module1.xml » et « module2.xml » se trouvant dans le même répertoire et ayant la structure suivante :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC...>
<struts>

  <package ...>
    ...
  </package>

</struts>
```

## 3.7 Injection de dépendances

L'injection de dépendances et en principe réalisée par l'intermédiaire des intercepteurs, devant donc être traité dans le chapitre 5. Mais, vu sa grande importance nous donnons un aperçu dessus dans cette partie. L'injection de dépendance va nous permettre de résoudre le problème suivant :

Problème :

Comment faire connaître à l'action les valeurs saisies dans un formulaire ?

Une solution :

### **Injection de dépendances**

Comment ?

Soit un champ de formulaire dont le nom est par exemple « **field** ». Il suffira de définir dans l'action un « **setter** » pour le champ; soit dans ce cas :

```
void setField(TypeDuFiled field)
```

Qui va appeler cette méthode ?

C'est le framework Struts (c'est pour cela qu'il est framework !), et ceci par l'intermédiaire d'un **intercepteur** approprié ; il s'agit de l'intercepteur prédéfinie :

### **ParametersInterceptor**

## Formulaire

```
<form action="UneAction" method="POST">
  ...
  <input type="text" size="15" name="field">
  ...
</form>
```

## La classe d'action :

```
public class UneAction {
    private type field;

    public void setField(type field) {
        this.field = field;
    }

    public void execute() {
        ...
        return "success";
    }

    public type getField() {
        return field;
    }
}
```

## La vue

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <body>
    ...
    <s:property value="field" /></h1>
    ...
  </body>
</html>
```





# Chapitre 4. Configuration par Convention

## 4.1 Principe de la technique de configuration par convention « **Convention Plugin** »

La technique de «Convention Plugin» n'a été intégrée à Struts qu'à partir de la version 2.1, soit au début de l'année 2009. La technique se base sur les conventions de nommage des classes d'action, des packages et des résultats pour leur localisation. La technique utilise aussi les annotations comme solution de configuration au lieu du fichier struts.xml.

Pour pouvoir utiliser la « **Convention plugin** », il faut intégrer le fichier jar « **struts2-convention-plugin-2.2.1.jar** » à votre application Struts, c.à.d au répertoire « **WEB-INF/lib** »

La technique de « convention plugin » permettra alors de trouver les actions sans avoir à les mapper par l'intermédiaire du fichier « struts.xml ». Par défaut, le principe est de chercher toutes les classes qui implémentent l'interface « **com.opensymphony.xwork2.Action** » (donc par exemple celles qui étendent la classe « **ActionSupport** » du même package), ainsi que toutes les classes dont le nom se termine par le mot « **Action** » dans des packages bien déterminés : ceux nommés **struts**, **struts2**, **action** ou **actions**.

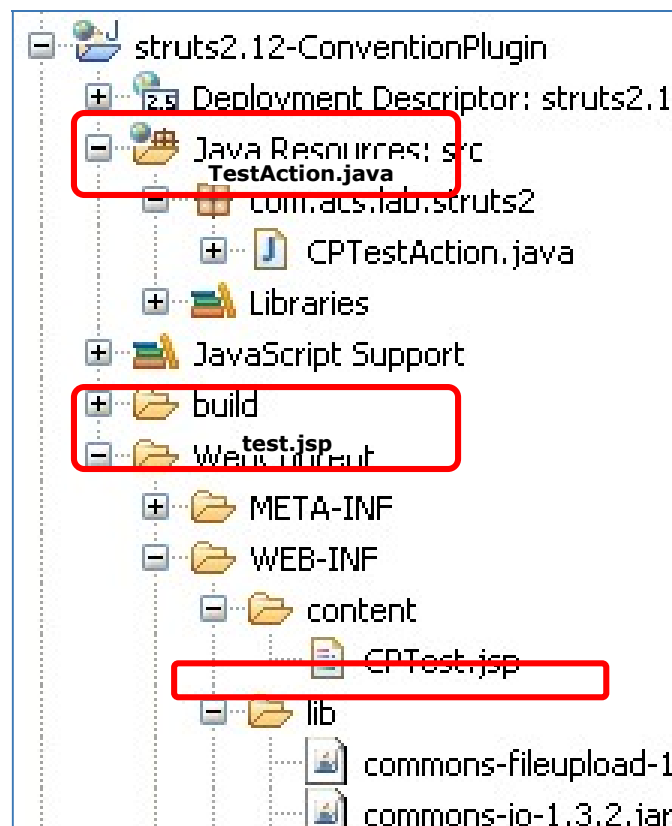
Les paragraphes suivants donneront plus de détails sur les différentes règles de convention.

## 4.2 Règles de convention par défaut

Si le nom de l'action est : « **nom** »

- 🚩 La classe d'action devrait avoir le nom « **NomAction** »
- 🚩 Dans l'un des packages : **struts**, **struts2**, **action**, **actions**
- 🚩 Méthode d'action par défaut « **public String execute()** »
- 🚩 Dans le cas de la valeur de retour « **success** » :  
Enchaînement vers le résultat (la vue) nommé :  
« **nom-success.jsp** » ou « **nom.jsp** »  
se trouvant dans le répertoire « **/WEB-INF/content** »
- 🚩 Dans le cas d'une valeur de retour différente « **retour** » :  
Enchaînement vers le résultat (la vue) nommé :  
« **nom-retour.jsp** » ou « **nom.jsp** »  
se trouvant dans le répertoire « **/WEB-INF/content** »

**Exemple : Nom d'action "test" (minuscule !)**



### 👉 La classe d'action :

```
package com.technos.labs.struts2;

public class TestAction {

    public String execute() {
        System.out.println(">>> EXECUTE");
        return "success";
    }
}
```

### 👉 La vue :

**test.jsp (minuscule !)**

Dans le répertoire : **WEB-INF/content**

### 👉 Exécution de l'action : le fichier index.html

```
<html>
<body>
    <h2>Exemple Convention Plugin</h2>
    <a href="test">Convention Plugin TestAction</a> <br/>
</body>
</html>
```

### **Remarque :**

Si le nom de la classe d'action est composé de plus qu'un Mot :

**NomAvecPlusDunMotAction**

Alors le nom de l'action devrait être :

« **nom-avec-plus-dun-mot** »

Et le nom de la JSP, pareil :

« **nom-avec-plus-dun-mot.jsp** »

## 4.3 Configuration

Les opérations de mapping et de recherche de ressources par la technique de « **Convention Plugin** » sont contrôlées par un certain nombre de constantes prédéfinies. Ces constantes sont configurables

```
<constant name="struts.convention.action.suffix" value="Controller"/>
<constant name="struts.convention.action.mapAllMatches" value="true"/>
<constant name="struts.convention.default.parent.package" value="rest-default"/>
<constant name="struts.convention.result.path" value="/WEB-INF/content/">
```

Le tableau suivant contient la liste de toutes ces constants avec leurs valeurs par défaut

Constante	Valeur par défaut
struts.convention.action.disableJarScanning	true
struts.convention.action.packages	
struts.convention.result.path	/WEB-INF/content/
struts.convention.result.flatLayout	true
struts.convention.action.suffix	Action
struts.convention.action.disableScanning	false
struts.convention.action.mapAllMatches	false
struts.convention.action.checkImplementsAction	true
struts.convention.default.parent.package	convention-default
struts.convention.action.name.lowercase	true
struts.convention.action.name.separator	-
struts.convention.package.locators	action,actions,struts,struts2
struts.convention.package.locators.disable	false
struts.convention.exclude.packages	org.apache.struts.*, org.apache.struts2.*, org.springframework.web.struts.*, org.springframework.web.struts2.*, org.hibernate.*
struts.convention.package.locators.basePackage	
struts.convention.relative.result.types	dispatcher,velocity,freemarker
struts.convention.redirect.to.slash	true

## 4.4 Convention Plugin par annotations :

### 4.4.1 Annotation : @Action

Si le nom de l'action est : « **x** »

👉 « **/x** » est l'url associée, par annotation, à une méthode d'action d'une classe postfixée par le mot Action « **...Action** » se trouvant dans l'un des packages : **struts, struts2, action, actions** :

👉 L'annotation est réalisée respectant la syntaxe suivante :

```
@Action (value="/x")  
public String uneMethode() {  
    ...  
    return "success";  
}
```

👉 La méthode sera exécutée après instanciation de la classe qui la contient

👉 Dans le cas de la valeur de retour « success » :  
    enchaînement vers le résultat (la vue) nommé :  
    « **x-success.jsp** » ou « **x.jsp** »  
    se trouvant dans le répertoire « /WEB-INF/content »

👉 Pareil, dans le cas d'une valeur de retour différente « **retour** » :  
    enchaînement vers le résultat (la vue) nommé :  
    « **x-retour.jsp** » ou « **x.jsp** »  
    se trouvant dans le répertoire « **/WEB-INF/content** ».

## 4.4.2 Annotation : @Result

- ✎ Si on veut exécuter, comme résultat, une page jsp non déterminée par convention, alors on devrait utiliser l'annotation suivante :

```
@Action (  
    value="/x",  
    results = {@Result  
                (name="success", location="uneAutre.jsp")  
            }  
)  
public String uneMethode() {  
    ...  
    return "success";  
}
```

- ✎ Le nom du résultat (name) devrait coïncider avec l'une des valeurs de retour de la méthode
- ✎ La méthode d'action peut avoir un nom quelconque (autre que le nom par défaut execute)

### Exemple :

```
package com.technos.labs.struts2;

import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Result;

public class BonjourAction {
    private String name;

    @Action (
        value="/test",
        results = {@Result (name="success", location="reponse.jsp")}
    )
    public String run() {
        setName("Bonjour" );
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

👉 Dans ce cas, l'accès à l'action est réalisée à l'aide de l'url "test" :

Exemple : <a href="**test**">...</a>

👉 la vue est : <**reponse.jsp**>

👉 Le « **name** » doit obligatoirement avoir dans ce cas la valeur :  
« **success** » puisque la méthode retourne « **success** ».

👉 On remarque aussi que le nom de la méthode, puisqu'il y a annotation, n'est pas obligatoirement « **execute** ». dans cet exemple nous avons choisi arbitrairement le nom « run() »



## 4.5 Annotation : @Actions

Une même méthode d'action peut être accessible par l'intermédiaire de plusieurs URLs :

```
package com.technos.labs.struts2;

import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;

public class TestAction {
    @Actions({
        @Action("/url1"),
        @Action("/url2")
    })
    public String execute() {
        return "success";
    }
}
```

## 4.6 Actions multiples

```
public class TestAction {
    @Action("/url1")
    public String a1() {
        ...
        return "success";
    }

    @Action("/url2")
    public String a2() {
        ...
        return "success";
    }
}
```

## Remarques:

- 👉 Les URLs peuvent être des chemins complets, exemple :  
/stock/ChercherProduit
- 👉 Le symbole « / » au début de l'URL signifie que la recherche de la ressource est réalisée relativement à la racine du contexte.
- 👉 Si on omet le symbole « / » la recherche est effectuée relativement au namespace.

## 4.7 Les namespaces

```
@Namespace("/p1")
public class TestAction {
    @Action("a1")
    public String a1() {
        ...
        return "success";
    }

    @Action("/u2/a2")
    public String a2() {
        ...
        return "success";
    }
}
```

Dans ce cas l'accès aux actions est réalisé comme suit :

- La première action : <a href="**p1/a1**">...</a>

Avec comme ressource « **WEB-INF/content/p1/a1.jsp** »

- La deuxième action : <a href="**u2/a2**">...</a>

Avec comme ressource « **WEB-INF/content/u2/a2.jsp** »

## 4.8 Annotation @Result

Il existe 2 types de résultats : Résultats globaux (Global Results) et résultats locaux (Local Results). Les résultats globaux sont annotés au niveau de la classe et ils sont associés à toutes les méthodes d'action sans résultat spécifié. Les résultats locaux sont associés directement aux méthodes d'action.

### **Exemple :**

```
@Results({
    @Result(name="error", location="error.jsp")
})
public class TestAction {
    @Action(
        value="a1",
        results={
            @Result(
                name="success",
                location="http://struts.apache.org",
                type="redirect"
            )
        }
    )
    public String a1() {
        ...
        return "success";
    }

    @Action("/u2/a2")
    public String a2() {
        ...
        return "success";
    }
}
```

## 📌 Passage de paramètres aux résultats

```
@Action(  
    value="a1",  
    results={  
        @Result(  
            name="success",  
            type="httpheader",  
            params={"p1", "v1", "p2", "v2", ...}  
        )  
    }  
)
```

### 4.9 Annotation ResultPath

Permet de changer le chemin par défaut (**WEB-INF/content**) dans lequel les vues sont stockées. Cette annotation opère au niveau de la classe :

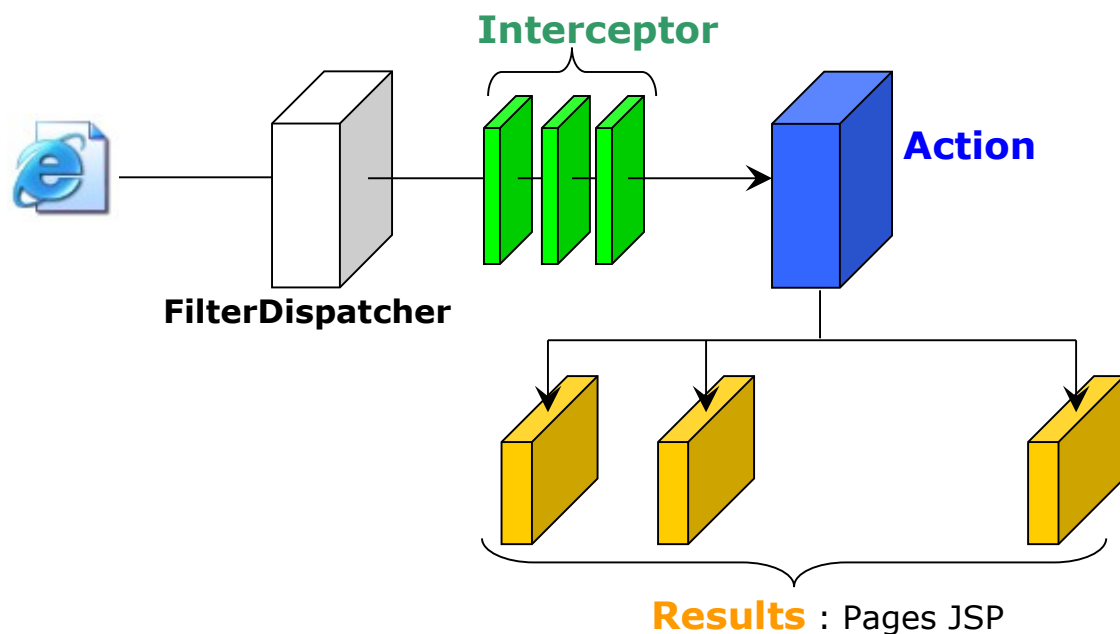
```
@ResultPath("/WEB-INF/NouveauDossier")  
public class TestAction {  
    ...  
}
```



# Chapitre 5. Les Intercepteurs

## 5.1 Principe de base

Avant d'expliquer le principe des Intercepteurs (Interceptors), il faut comprendre le cycle de vie d'une action qui se déroule en six étapes :



1. La requête est tout d'abord reçue par le framework, à l'aide du contrôleur de Struts « StrutsPrepareAndExecuteFilter » du package « org.apache.struts2.dispatcher.ng.filter ». Le framework détermine les composantes reliées à la requête : Interceptors, action et results. Et ce conformément au contenu du fichier « struts.xml » ou aux décisions prises par la Convention Plugin. Les interceptors sont créés une fois pour toutes au niveau session, alors que l'action est instanciée à ce stade même et à chaque nouvelle requête une nouvelle instance est créée.

2. La requête passe ensuite par une série d'intercepteurs qui sont programmés pour fournir une liste de prétraitements avant que la requête n'arrive finalement à l'action.
3. L'action est ensuite invoquée par l'intermédiaire de l'une de ses méthodes d'action.
4. Le vue « est enfin » appelée. Elle pourra accéder à l'action pour récupérer des données calculées par celle-ci.
5. La requête passe de nouveau par la liste des intercepteurs pour offrir la possibilité d'appliquer un post-traitement ou tout traitement de nettoyage pouvant être programmé en ce niveau.
6. en fin la réponse HTML est retournée au client final.

## 5.2 Liste d'intercepteurs prédéfinis

Ci-dessous une liste d'intercepteurs issus directement de la classe « AbstractInterceptor » ; Mais, il existe bien d'autres qui sont des classes filles des classes suivantes :

ActionAutowiringInterceptor  
AliasInterceptor  
AnnotationParameterFilterIntereptor  
ChainingInterceptor  
ClearSessionInterceptor  
ConversionErrorInterceptor  
CookieInterceptor  
CreateSessionInterceptor  
ExceptionMappingInterceptor  
FileUploadInterceptor  
I18nInterceptor  
LoggingInterceptor  
MethodFilterInterceptor ➔ ParametersInterceptor  
ModelDrivenInterceptor  
ParameterFilterInterceptor

ParameterRemoverInterceptor  
ProfilingActivationInterceptor  
RolesInterceptor  
ScopedModelDrivenInterceptor  
ScopeInterceptor  
ServletConfigInterceptor  
StaticParametersInterceptor  
TimerInterceptor

## 5.3 Intercepteur avec Injection de Dépendances

### 5.3.1 Injection des paramètres de formulaire

Nous avons déjà vu (fin du chapitre 3) que les paramètres d'un formulaire sont injectés dans l'action à l'aide de setters appropriés qu'il faut prévoir dans la classe d'action. Le composant qui se charge d'appeler ces setters est un intercepteur qui fait donc l'opération en prétraitement avant que la méthode d'action ne soit exécutée. L'intercepteur en question est le « **ParametersInterceptor** ».

### 5.3.2 Injection basée sur les interfaces « Interface Injection »

L'intercepteur « **ServletConfigInterceptor** » est conçu pour injecter toutes les propriétés dont une action peut avoir besoin, à condition qu'elle en soit consciente (**aware**), c'est-à-dire qu'elle doit implémenter l'interface appropriée. On site :



- ServletContextAware
- ServletRequestAware
- ServletResponseAware
- ParameterAware
- RequestAware
- SessionAware
- ApplicationAware
- PrincipalAware

Par la suite quelques exemples d'injections utiles basées sur cet intercepteur :

#### 👉 Injection d'un objet HttpServletRequest

- Interface : **ServletRequestAware**
- Méthode à implémenter :

```
public void setServletRequest(HttpServletRequest request)
```

#### **Exemple :**

```
public class UneAction implements ServletRequestAware {

    private HttpServletRequest request;

    public void setServletRequest(HttpServletRequest request) {
        this.request = request;
    }

    public String execute() throws Exception {
        return "success";
    }

}
```

### 📌 Injection d'un objet HttpServletResponse

- Interface : « **ServletResponseAware** »
- Méthode à implémenter :

```
void setServletResponse(HttpServletResponse response)
```

### 📌 Injection d'un objet ServletContext

- Interface : « **ServletContextAware** »
- Méthode à implémenter :

```
void setServletContext(ServletContext context)
```

### 📌 Injection des attributs d'une Session

- Interface : « **SessionAware** »
- Méthode à implémenter :

```
void setSession(Map<String, Object> session)
```

### 📌 Injection des paramètres (autre solution)

- Interface : « **ParameterAware** »
- Méthode à implémenter :

```
void setParameters(Map<String, String[]> parameters)
```

## 5.4 Le ModelDrivenIntercepteur

➔➔ Injection dans le modèle récupéré par l'intercepteur

```
public class Product {  
    private String id,  
    private String name;  
    private double price;  
    private double amount;  
  
    ...  
}
```

```
<form action="ProductAction" method="POST">  
  Id   : <input type="text" size="15" name="id"> <br />  
  Designation : <input type="text" size="25" name="name"> <br />  
  PU   : <input type="text" size="10" name="price"> <br />  
  QS   : <input type="text" size="6" name="amount"> <br />  
  
  <input type="submit" value="Enregistrer">  
</form>
```

```
public class ProductAction implements ModelDriven<Product> {  
    private Product product;  
  
    public ProductAction() {  
        product = new Product();  
    }  
  
    public Product getModel() {  
        return product;  
    }  
  
    ...  
}
```

## 5.5 Réalisation de nouveaux intercepteurs

La réalisation d'un intercepteur exige le passage par les étapes suivantes :

1. Réaliser la classe intercepteur qui devrait implémenter l'interface « **Interceptor** » du package « **com.opensymphony.xwork2.interceptor** » :

```
public interface Interceptor {  
    public void init();  
    public void destroy();  
    public String intercept(ActionInvocation a);  
}
```

Ceci peut être réalisé par extension de la classe abstraite « **AbstractInterceptor** » du même package qui implémente l'interface « **Interceptor** » en fournissant une implémentation vide des deux méthodes **init()** et **destroy()**.

2. Fournir une implémentation de la méthode **intercept(...)**. Son paramètre « **ActionInvocation** » donne accès aussi bien à l'action qu'au résultat (la vue). Les deux méthodes suivantes sont disponibles :

- **getAction() : Object**
- **getResult() : Result**

Il est alors possible de communiquer la donnée qu'on veut à l'action une fois récupérée.

### **Exemple 1 :**

```
package com.technos.labs.struts2;

import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class TestInterceptor extends AbstractInterceptor {

    public String intercept(ActionInvocation invocation)
        throws Exception {

        Action1 a = (Action1)invocation.getAction();
        a.setData(...);

        return invocation.invoke();
    }
}
```

En supposant que l'on dispose d'une action qui s'appelle « Action1 » qui offre une méthode setData(...).

3. Configuration : l'intercepteur doit ensuite être configuré dans le fichier « struts.xml »

```
<struts>
  <package name="..." extends="struts-default">
    <interceptors>

      <interceptor name="interceptor1"
        class="com.technos.labs.struts2.TestInterceptor" />

    </interceptors>

    <action name="Action1"
      class="com.technos.labs.struts2.Action1">

      <interceptor-ref name="interceptor1" />

      <result>/result.jsp</result>
    </action>
  </package>
</struts>
```

### **Remarque :**

L'extension du package « struts-default » permet de charger les intercepteurs par défaut.

### **Exemple 2 :**

#### **Code de l'intercepteur :**

Un intercepteur qui gère une base de données « stock » en communiquant une référence sur celle-ci à l'action :

```
package com.technos.labs.struts2;

import java.util.HashMap;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class InventoryInterceptor extends AbstractInterceptor {
    private HashMap<String, Product> inventory;

    public InventoryInterceptor() {
        inventory = new HashMap<String, Product>();
    }

    public String intercept(ActionInvocation invocation) throws Exception
    {
        ProductAction action = (ProductAction)invocation.getAction();
        action.setInventory(inventory);
        return invocation.invoke();
    }
}
```

### **Code de l'action :**

Celle-ci fourni 3 méthodes d'action :

- insert
- select
- list

```
public class ProductAction implements ModelDriven<Product> {  
    private HashMap<String, Product> inventory;  
    private int id;  
    private List<Product> result;  
    private Product product;  
  
    public ProductAction() {  
        product = new Product();  
    }  
    public String select() {  
        product = inventory.get(product.getId());  
        if (product==null) return "error";  
        else return "success";  
    }  
    public String list() {  
        result = inventory.values();  
        return "success";  
    }  
    public Product getModel() {  
        return product;  
    }  
    public Product getProduct() {  
        return product;  
    }  
    public List<Product> getResult() {  
        return result;  
    }  
    public void setInventory(HashMap<String, Product> inventory) {  
        this.inventory = inventory;  
    }  
}
```

## Configuration :

```
<struts>
  <package name="Struts2.5" extends="struts-default">
    <interceptors>
      <interceptor name="inventoryInterceptor"
        class="com.technos.labs.struts2.InventoryInterceptor"
      />
      <interceptor-stack name="config">
        <interceptor-ref name="inventoryInterceptor" />
        <interceptor-ref name="defaultStack" />
      </interceptor-stack>
    </interceptors>

    <default-interceptor-ref name="config" />

    <action name="productSelect"
      class="com.technos.labs.struts2.ProductAction" method="select"
    >
      <result>/selectResponse.jsp</result>
      <result name="error">/Error.jsp</result>
    </action>
    <action name="productList"
      class="com.technos.labs.struts2.ProductAction" method="list"
    >
      <result>/List.jsp</result>
    </action>
  </package>
</struts>
```





# Chapitre 6. Bibliothèque de balises Struts 2

On peut généralement répartir les balises Struts en 3 catégories qui seront traitées dans ce chapitre : « Data tags », « Control tags » et « UI Tags ».

## 6.1 Balises de traitement des données « Data Tags »

Nous choisissons une liste représentative de balises, ce qui nous permettra d'avoir un aperçu général sur l'utilisation des Data Tags.

📌 Balise <property>

### **- Utilisation générale :**

```
<s:property  
    value="nom de la propriété"  
    default="valeur par défaut si on reçoit null"  
/>
```

Permet généralement de récupérer une propriété de l'action (via son getter)

### **Exemple :**

*Dans l'action :*

```
public String getName() {  
    return name;  
}
```

*Dans la JSP result :*

```
<s:property value="name" />
```

### **Exemple 2 :**

*Dans l'action*

```
private Product product;  
  
public String execute() {  
    product = new Product("C01", "Clavier", 200, 10);  
  
    return "success";  
}  
  
public Product getProduct() {  
    return product;  
}
```

*Dans la JSP result :*

```
<s:property value="product" />  
<s:property value="product.name" />  
<s:property value="product.price" />
```

### **- Récupération d'un attribut :**

*Soit dans l'action :*

```
Product p = new Product("C01", "Clavier", 200, 10)  
ServletContext.getRequest().setAttribute("p01", p);
```

*Solution classique (JSP)*

```
<%  
    Product p = (Product)request.getAttribute("p01");  
%>
```

*Ou bien :*

```
<% = request.getAttribute("p01") %>
```

*Solution Struts*

```
<s:property value="#attr.p01" />
```

📌 Balise <action>

Inclusion du résultat d'exécution d'une action :

```
<s:action name="NomD'action" executeResult="true" />
```

📌 Balise <bean>

```
<s:bean name="nom.de.la.Classe" var="nomDuBean">  
    <s:param name="p1" value="v1" />  
    ...  
</s:bean>
```

➔ Création d'un bean

## 📌 Balise <date>

*Soit dans l'action :*

```
private Date date1 = new Date();  
private GregorianCalendar date2= new GregorianCalendar();
```

*Dans la JSP result :*

```
<s:date name="date1" format="dd/MM/yyyy"/><br/>  
<s:date name="date2" format="dd/MM/yyyy"/><br/>
```

## 📌 Balise <include>

### **Exemples :**

```
<s:include value="V01.jsp" />
```

```
<s:include value="V01.jsp">  
  <s:param name="param1" value="value2" />  
  <s:param name="param2" value="value2" />  
</s:include>
```

```
<s:include value="V01.jsp">  
  <s:param name="param1">value1</s:param>  
  <s:param name="param2">value2</s:param>  
</s:include>
```

## 📌 Autres balises à voir :

```
a, debug, i18n, push, set, text, url
```

## 6.2 Balises de contrôles « Control Tags »

👉 Les balises <if>, <elseif> et <else>

**Exemple :**

```
<s:if test="product.name=='clavier'">
    ...
</s:if>
<s:elseif test="product.name=='ecran'">
    ...
</s:elseif>
<s:else>
    ...
</s:else>
```

👉 La balise <iterator>

*Soit l'action :*

```
public class NotreAction {
    private Vector<Product> products = new Vector<Product>();

    public String execute() {
        products.add(new Product("C01", "Clavier", 200, 10));
        products.add(new Product("C02", "Ecran", 1200, 10));
        return "success";
    }
    public Vector<Product> getProducts() {
        return products;
    }
}
```

Dans la JSP result :

```
<s:iterator value="products">
    <li>  <s:property value="name"/> :
          <s:property value="price"/>
</s:iterator>
```

👉 Autres balises à voir :

append , generator , merge , sort , subset

## 6.3 Balises d'interfaces « UI Tags »

👉 Balise <select>

Soit la classe d'action :

```
public class NotreAction {
    private Vector<String> cities = new Vector<String>();
    public String execute() {
        cities.add("Casablanca");
        cities.add("Fès");
        cities.add("Rabat");
        return "success";
    }
    public Vector<String> getCities() {
        return cities;
    }
}
```

*Dans la JSP result :*

```
<s:select list="cities"></s:select>
```

📌 Autres balises :

checkbox , checkboxlist , combobox , doubleselect , head , file , form ,  
hidden , label , optiontransferseselect , optgroup , password , radio , reset ,  
submit , textarea , textfield , token , updownselect, tree , treenode