Introduction à la plate-forme Java Enterprise Edition

année 2012-2013

Agenda

Qu’est-ce la plateforme J2EE

Les Servlets et applications web

Les JSP

Le Modèle MVC (Architecture en couches)

L’Accès aux données via l’interface universelle « JDBC »

Modèle-Vue-Contrôleur

Développer une application Web est simple

Développer une application Web structurée et facile à maintenir l'est beaucoup moins

Le patron "Modèle-Vue-Contrôleur" (MVC), dû à Adele Goldberg (Xerox) pour SmallTalk, a été proposé pour la conception des applications Web

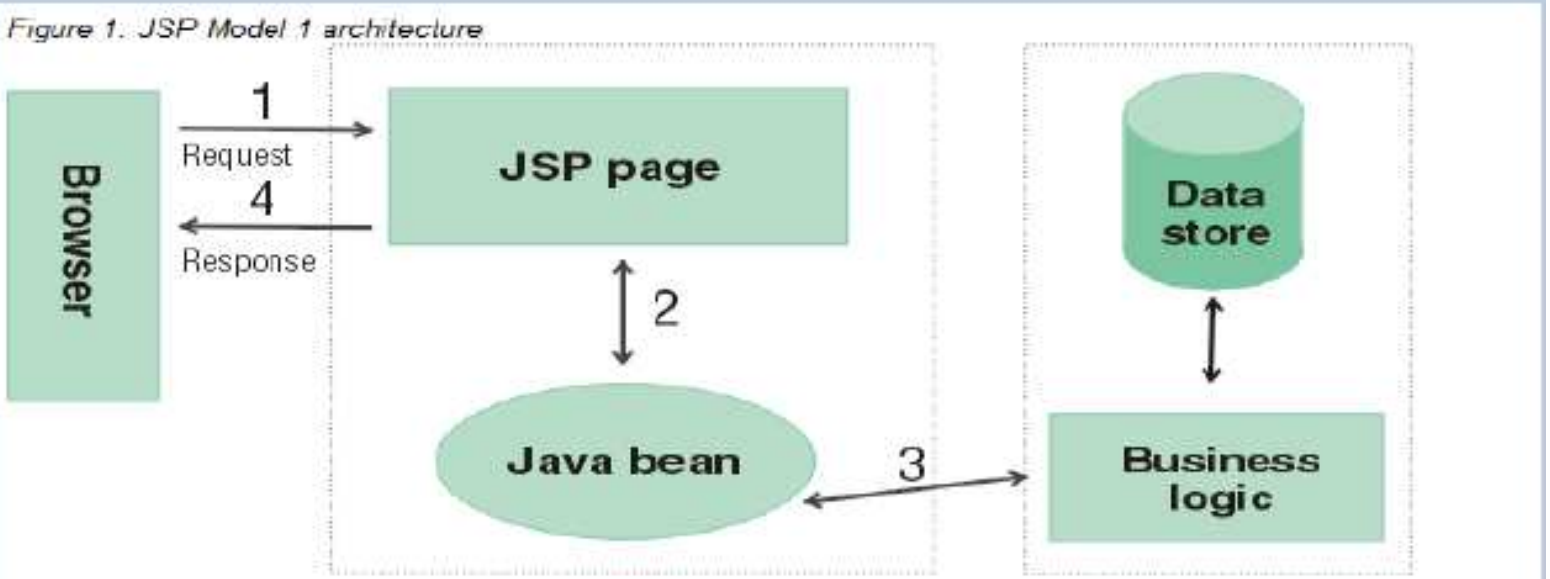
Une architecture MVC se divise en trois parties distinctes :

- Modèle : ensemble d'objets qui représentent la logique métier de l'application.
- Vue : objet ou groupe d'objets qui fabrique ce que voit l'utilisateur (page HTML)
- Contrôleur : objet qui définit la nature des réactions de l'interface utilisateur aux introductions de données. Il transforme l'information issue de la couche Modèle en une forme compréhensible par la couche Vue et traite toutes les décisions de telle sorte que la vue affichée corresponde bien à l'action choisie par l'utilisateur

MVC

MVC

Figure 1. JSP Model 1 architecture



1. The browser sends a request to a JSP page.
2. The JSP page communicates with a Java bean.
3. The Java bean is connected to a database.
4. The JSP page responds to the browser.

MVC

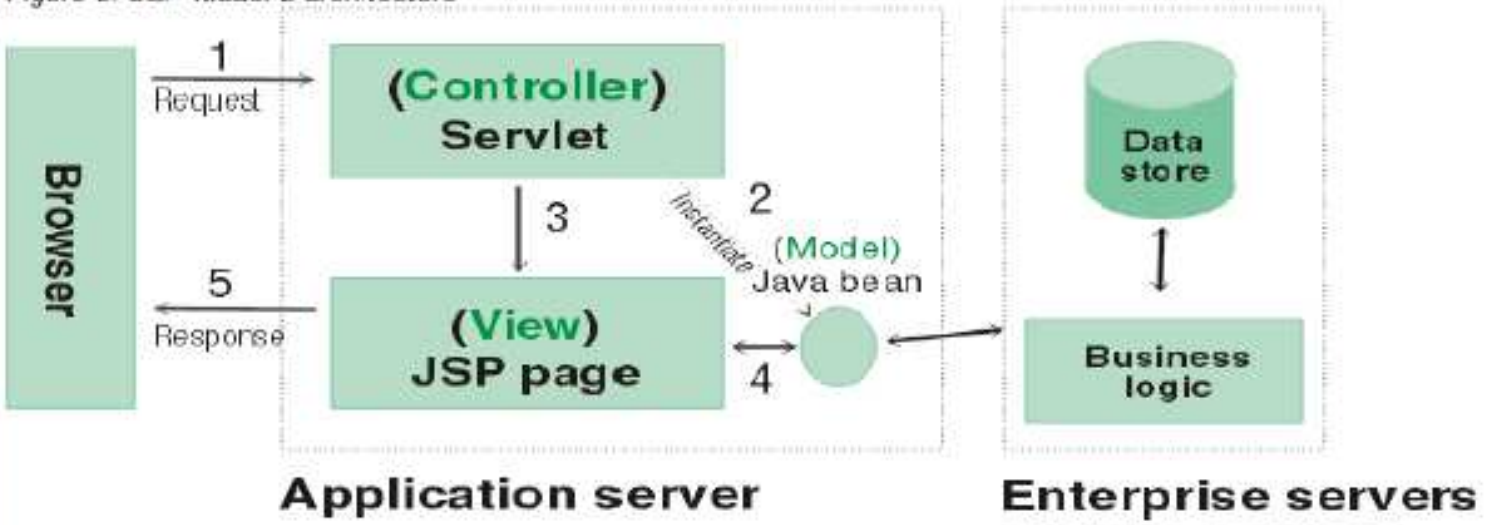
Le Model (MVC1)

- Cette architecture est valable pour de petites applications
- Elle présente des problèmes de
  - maintenance : la logique est mélangée à la présentation,
  - les pages sont dès lors plus compliquées, moins lisibles et moins maintenables
  - réutilisation : puisque la logique est placée dans la présentation, on ne sait pas l'utiliser ailleurs
  - sécurité : risque de dévoiler dans la page, des données sensibles liées à l'accès à la base de données

MVC

Le Model 2 (MVC 2)

Figure 2. JSP Model 2 architecture



1. The browser sends a request to a servlet.
2. The servlet instantiates a Java bean that is connected to a database.
3. The servlet communicates with a JSP page.
4. The JSP page communicates with the Java bean.
5. The JSP page responds to the browser.

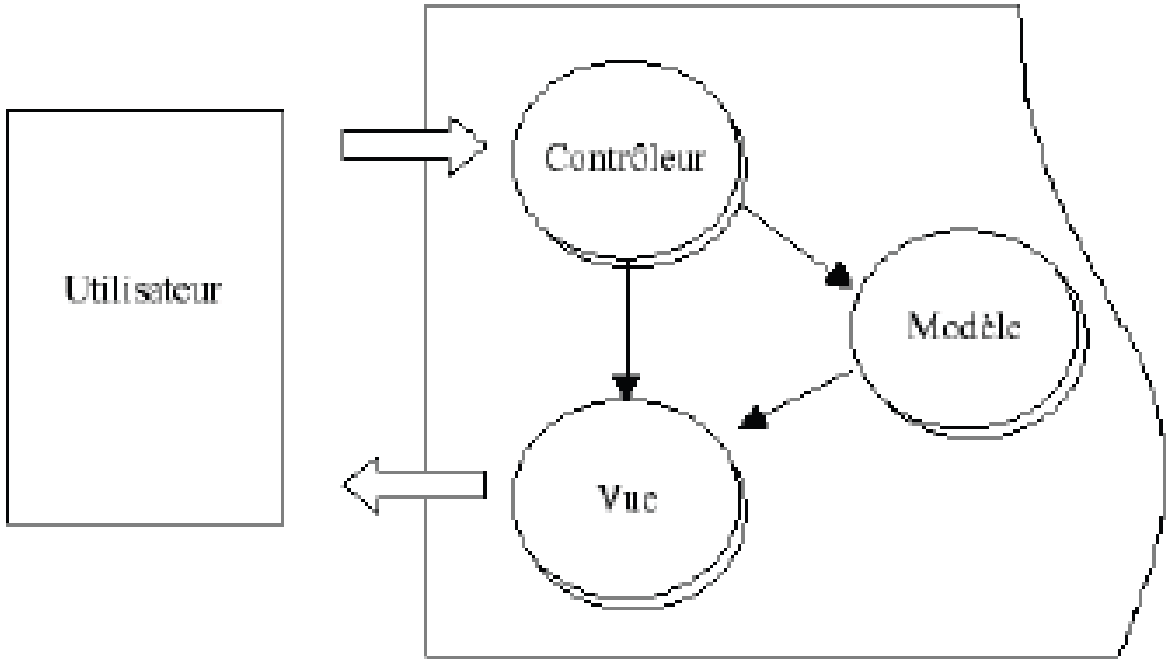
Table 2 presents criteria to help you determine when Model 1 or Model 2 is likely to be more appropriate:

MVC

Le Model 2 (MVC 2)

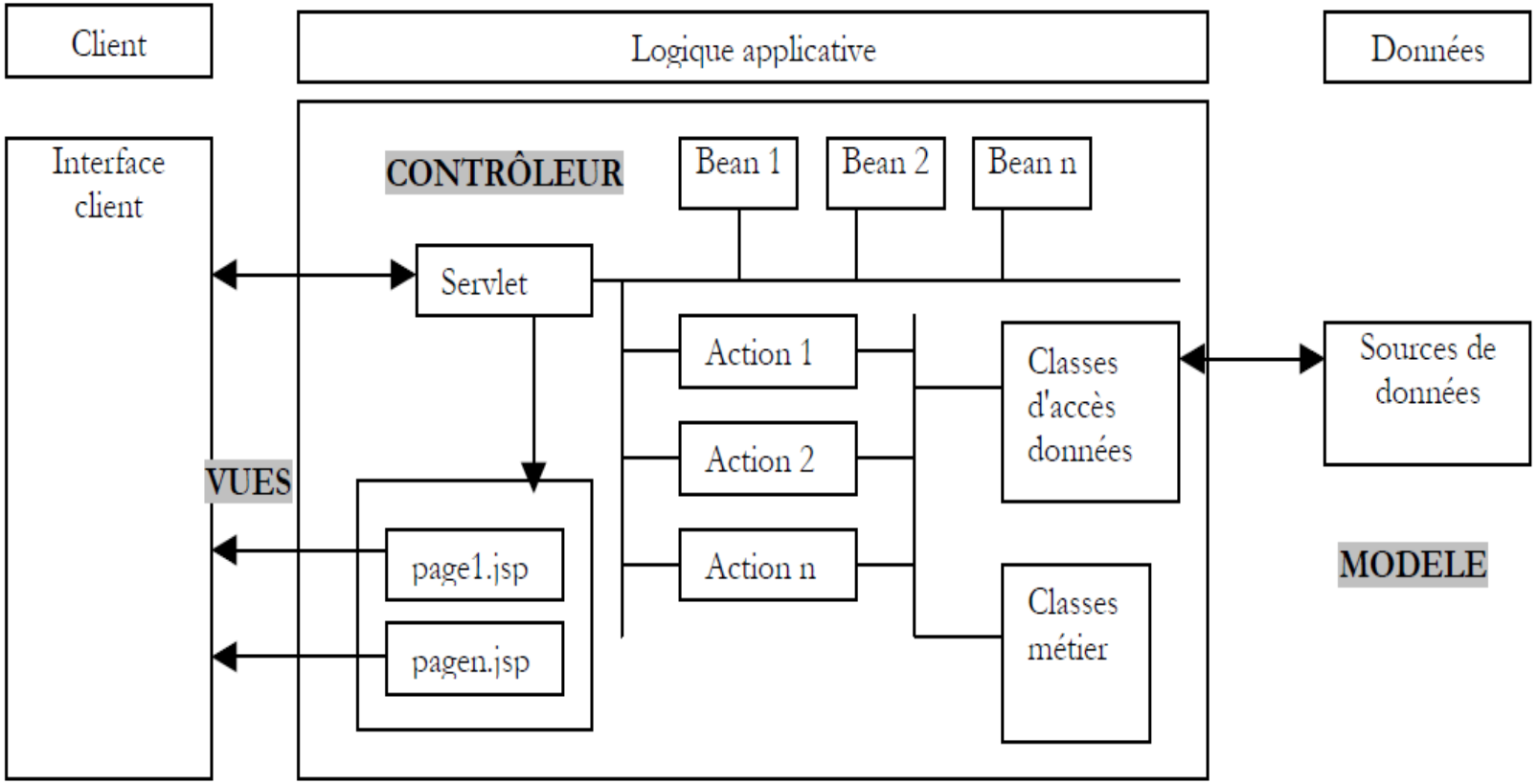
- Permet de développer/d'accéder à une application entreprise.
- L'utilisation de javabeans par les pages JSP y est moins nécessaire.
- Les problèmes précédents sont résolus par la séparation de la logique (servlet) et de la présentation (page JSP).

Modèle-Vue-Contrôleur

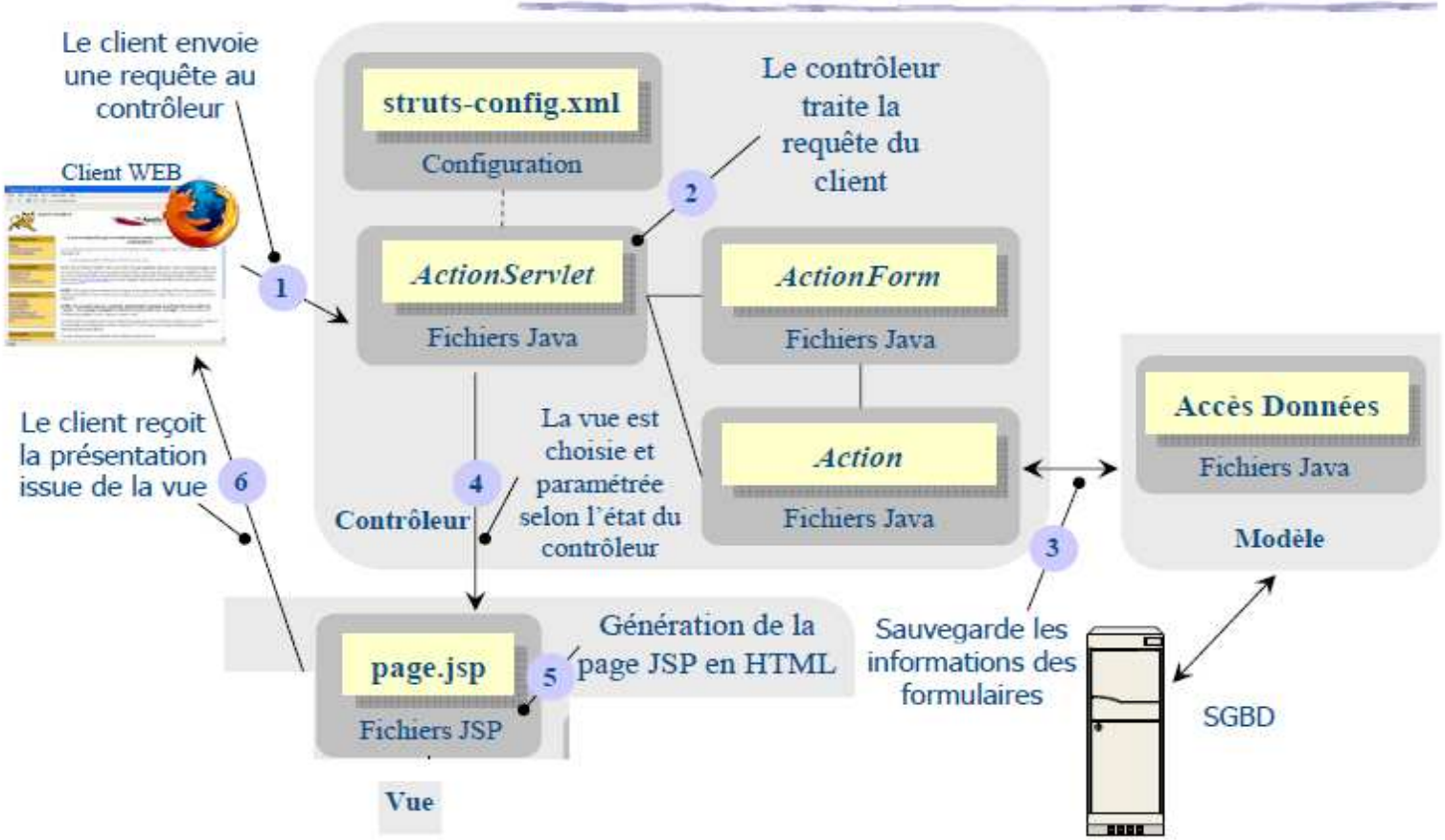


Struts

L'architecture MVC utilisée par STRUTS est la suivante :



Architecture et composants struts



Struts

- La bibliothèque Struts est un framework qui permet de construire des applications Web respectant le modèle d'architecture MVC
- Struts est un projet soutenu par l'Apache Software Foundation. Site de référence *struts.apache.org*
- Logique de fonctionnement
  - la structure de l'application Web est décrite dans *struts-config.xml*
  - l'utilisation de Servlets est transparente via des classes adaptées
  - les pages JSP exploitent des balises personnalisées de Struts.

Contrôleur

- Le contrôleur est le cœur de l'application Web. Toutes les demandes du client transitent par lui
- Il est défini par une Servlet générique de type *ActionServlet* fournie par l'API de Struts
- Le contrôleur prend les informations dont il a besoin dans le fichier *struts-config.xml*
- Si la requête du client contient des paramètres, ceux-ci sont transmis dans un objet de type *ActionForm*
- Selon l'état retourné par l'*ActionForm* précédent, le contrôleur traite une action spécifique par un objet de type *Action*

Contruire une applicaton web avec Struts

- Il est nécessaire de modifier le fichier web.xml pour déclarer struts
- Struts est atteint par toutes les URL's se qui se terminent par le suffixe ".do"

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

Le contrôleur est défini par la Servlet générique *ActionServlet*

En paramètre de la Servlet le fichier *struts-config.xml*

web.xml

les urls terminent par ".do"

Le fichier de configuration struts-config.xml

- Le fichier gérant la logique de l'application Web s'appelle *struts-config.xml*
- Il est placé dans le répertoire WEB-INF au même niveau que *web.xml*
- Il décrit essentiellement trois éléments
  - les objets (*ActionForm*) associés aux formulaires JSP
  - les actions à réaliser suite aux résultats des objets *ActionForm* (*Action*)
  - les ressources éventuelles suites à des messages
- Le fichier de configuration est un fichier XML
  La balise de départ est *<struts-config>*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
          "http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    ...
</struts-config>
```

Description de fonctionnement de l'architecture de l'application Web

Struts : Action

- Une action est un traitement obtenu suite au passage de la requête au contrôleur
- Nous distinguons deux sortes de requête client
  - requête sans paramètre issue par exemple d'une re-direction
  - requête avec paramètres issue par exemple d'un formulaire
- Les actions sont décrites dans la balise *<action-mappings>* au moyen de la balise *<action>*
- Selon le type de requête (avec ou sans paramètre) différents attributs de la balise *<action>* sont à renseigner

Struts : Action

- Dans le cas d'une requête sans paramètre le rôle du contrôleur est de relayer la demande du client à une URL
- La balise *<action>* dispose alors des attributs suivants
  - *String path* : définit le nom de l'URL (suffixe « .do » implicite)
  - *String type* : définit le nom de la classe *Action* qui doit traiter la demande. Utilisez la classe *org.apache.struts.actions.ForwardAction* dans ce cas précis de re-direction
  - *String parameter* : le nom de l'URL à qui doit être relayée la demande

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
          "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<struts-config>
    <action-mappings>
        <action
            path="/nom.do"
            parameter="/vues/formulaire.jsp"
            type="org.apache.struts.actions.ForwardAction" />
    </action-mappings>
</struts-config>
```

Quand le client transmet l'URL « .../nom.do » au contrôleur celui-ci redirige vers « /vues/formulaire.jsp »

Il s'agit d'une re-direction

Struts : Action

- Épisode 1 : appel du formulaire de saisie du nom et de l'age

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
          "http://struts.apache.org/dtds/struts-config_1_2.dtd">
<struts-config>
    <action-mappings>
        <action
            path="/formulaire"
            parameter="/vues/formulaire.jsp"
            type="org.apache.struts.actions.ForwardAction" />
    </action-mappings>
</struts-config>
```

Le formulaire est défini dans la page « formulaire.jsp »

Le client envoie la requête suivante

C'est la page *formulaire.jsp* qui est retournée au client

Struts : Action

- Dans le cas d'une requête avec paramètres le rôle du contrôleur est double
  - transmettre les informations dans un objet de type *ActionForm*
  - réaliser une action spécifique (autre qu'une simple redirection)
- La balise *<action>* dispose, en plus des attributs déjà étudiés, des attributs suivants
  - *String scope* : les valeurs du formulaire sont stockées en session
  - *String name* : référence le nom d'une section *<form-bean>*
  - *String validate* : indique si la méthode *validate* de l'objet *ActionForm* doit être appelée ou non
  - *String input* : indique la vue qui sera appelée s'il y a erreur dans l'objet *ActionForm*

Struts : Action

- Les formulaires sont déclarés dans la balise *<form-beans>* au moyen de la balise *<form-bean>*
- La balise *<form-bean>* possède les attributs suivants
  - *String name* : nom du formulaire de la page JSP
  - *String type* : classe *ActionForm* qui stocke les paramètres du Bean

```
<?xml version="1.0" encoding="ISO-8859-1"?>
...
<struts-config>
    <form-beans>
        <form-bean name="formulaire"    type="monpackage.ClassActionForm" />
    </form-beans>
    <action-mappings>
        <action
            path="/nom"
            name="formulaire"
            scope="session"
            validate="true"
            input="/pageerreurs.do"
            parameter="/vues/formulaire1.jsp"
            type="org.apache.struts.actions.ForwardAction" />
    </action-mappings>
</struts-config>
```

Quand le client transmet l'URL « .../nom.do » au contrôleur celui-ci redirige vers « /vues/formulaire1 .jsp » si aucun problème

Dans le cas où les paramètres sont mauvais le contrôleur redirige vers «/pageerreurs.do »

Struts : Action

- Épisode 2 : envoie d'une requête de type POST du formulaire

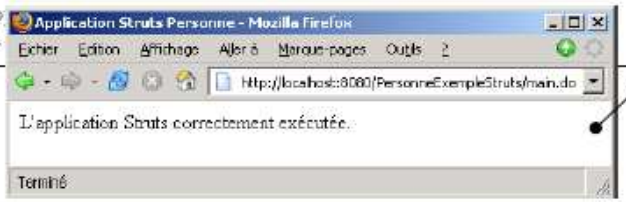
Trois actions et un formulaire sont actuellement définis

Le formulaire est défini par la valeur « formPersonne »

```
...
<struts-config>
    <form-beans>
        <form-bean name="formPersonne" type="monpackage.FormulaireBean" />
    </form-bean>

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

Les valeurs sont stockées dans *monpackage.FormulaireBean*

Si les données sont correctes direction « /vues/main.jsp » sinon direction « /erreurs.do »



Struts :Action

- Épisode 2 (suite) : envoie d'une requête issue du formulaire

```
<%@ taglib uri="htmlstruts" prefix="html" %>
...
<body>
<center>
<h2>Personne - Formulaire</h2><hr>
<html:form action="/main" >
<table>
    <tr>
        <td>Nom</td>
        <td><html:text property="nom" size="20" /></td>
    </tr>
    <tr>
        <td>Age</td>
        <td><html:text property="age" size="3"/></td>
    </tr>
</table>
<table>
    <tr>
        <td><html:submit value="Envoyer" /></td>
        <td><html:reset value="Retablir" /></td>
        <td><html:button property="btnEffacer" value="Effacer" onclick="effacer()"/></td>
    </tr>
</table>
</html:form>
...
```

Bibliothèque de balises personnalisées Struts:HTML

L'action du formulaire est d'appeler la ressource « /main » associée

Les deux paramètres transmis en paramètre de la requête

Struts : ActionForm

- L'objectif d'un objet de type *ActionForm* est de stocker les informations issues d'un formulaire
- Les classes de type ActionForm devront donc hériter de la classe *ActionForm* du package *org.apache.struts.action*
- C'est le contrôleur via la Servlet qui se charge de créer les instances des objets de type *ActionForm*
- Pour chaque propriété de la classe un attribut et deux méthodes doivent être définis
  - un modifieur pour affecter une valeur à l'attribut
  - un accesseur pour obtenir la valeur de l'attribut correspondant

Struts : ActionForm

- Hormis le but de stocker les propriétés des formulaires, les objets de type *ActionForm* s'occupent aussi de l'aspect sémantique des données
- La méthode *validate* s'occupe de vérifier la validité des attributs de l'objet Bean
- *ActionErrors validate(ActionMapping, HttpServletRequest)*
  - le paramètre *ActionMapping* est un objet « image » de la configuration de l'action en cours stockée dans *struts-config.xml*
  - le paramètre *HttpServletRequest* est la requête du client transmise par la Servlet de contrôle
  - le retour *ActionErrors* permet de retourner des messages erreurs au client
- La classe *ActionForm* dispose également d'autres méthodes
  - *ActionServlet getServlet()* : retourne la Servlet qui gère le contrôle
  - *reset(ActionMapping, HttpServletRequest)* : initialise les propriétés

Struts : ActionForm

- Un objet de type *ActionMapping* permet d'extraire les informations contenu dans le fichier *struts-config.xml*
- Il possède des méthodes associées

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

  - *String getType()* : pour accéder au contenu de l'attribut *type*
  - *String getInput()* : pour accéder au contenu de l'attribut *input*
- Un objet *ActionErrors* permet d'ajouter des erreurs et l'ajout se fait par la méthode
  - *add(String, ActionMessage)* : où le premier paramètre correspond à la clé et le second au message d'erreur

Épisode 3 : stocker les informations du formulaire

```
public class Formulaire     extends ActionForm {
    private String nom = null;
    private String age = null;

    public String getNom() {
        return nom;
    }

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

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

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

La classe du framework Struts qui gère les classes de type ActionForm associés aux formulaires

Les deux attributs modélisant les propriétés de la classe

Les modifieurs et accesseurs pour traiter et modifier les propriétés

Épisode 3 (suite) : stocker et valider les info. du formulaire

```
public class Formulaire     extends ActionForm {
    ... // Lié à la modélisation des propriétés
    public ActionErrors validate(ActionMapping arg0, HttpServletRequest arg1) {
        ActionErrors erreurs = new ActionErrors();
        if (nom == null || nom.trim().equals("")) {
            erreurs.add("nomvide", new ActionMessage("formulaire.nom.vide"));
        }
        if (age == null || age.trim().equals("")) {
            erreurs.add("agevide", new ActionMessage("formulaire.age.vide"));
        } else {
            try {
                int mon_age_int = Integer.parseInt(age);
                    if (mon_age_int < 0) {
                    erreurs.add("ageincorrect",
                        new  ActionMessage("formulaire.age.incorrect"));

                    }
                } catch (Exception e) {
                    erreurs.add("ageincorrect",
                    new ActionMessage("formulaire.age.incorrect",age));
                }
        }
        return erreurs;
    }
```

Au début *erreurs* est vide donc pas d'erreur

Ajout des erreurs selon « l'arrivage »

Depuis la nouvelle version 1.2, il faut utiliser *ActionMessage* et non *ActionError*

Struts : ActionForm et ActionErrors

- Les messages d'erreurs stockés dans un objet *ActionErrors* et retournés par la méthode *validate* sont transmis au contrôleur
- Si *validate* vaut « true » et que l'objet *ActionErrors* n'est pas *null* le contrôleur redirige vers la vue de l'attribut *input*

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

- Les erreurs sont affichées dans la vue JSP au moyen de la balise personnalisée *<errors>* de la bibliothèque Struts-HTML

```
<%@ taglib uri="htmlstruts" prefix="html" %>
<html:errors/>
```

- La balise *<errors>* n'affiche pas les messages mais des identifiants présents dans un fichier ressource qui **doit** être référencé dans struts-config.xml

Struts : ActionForm et ActionErrors

- Pour déclarer un fichier ressource dans le fichier configuration *struts-config.xml* utiliser la balise *<message-resources>*
  - *String parameter* : nom du fichier ressource
  - *boolean null* : *true* affiche *null*, *false* affiche *???key???*
  - *String key* : à utiliser quand il y a plusieurs fichiers ressources

De préférence à la fin du fichier *struts-config.xml*

- Le fichier ressource doit porter comme extension *.properties*
  - Exemple de fichier : toto.properties
- Le fichier ressource doit être placer obligatoirement dans un sous-répertoire de /WEB-INF/classes. Exemples :
  - /WEB-INF/classes/toto.properties
  - /WEB-INF/classes/
- Pour choisir le fichier ressource, utilisez l'attribut *bundle* dans la balise *<errors>* en indiquant le nom de la clé

Struts : ActionForm et ActionErrors

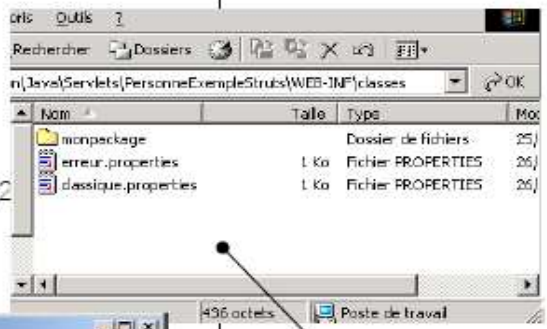
➤ Épisode 4 : gérer les erreurs sémantiques du formulaire

```
    ...
    <message-resources parameter="erreur" null="false" key="erreur" />
    <message-resources parameter="classique" null="false" key="classique" />
</struts-config>
```

Fichier « erreur.properties »

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

```
<%@ taglib uri="/WEB-INF/tlds/struts-l

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



Emplacement des fichiers properties

Struts : Action

- Nous avons pour l'instant utilisé simplement la classe *ForwardAction* qui ne permet que de traiter des re-directions sans de réels traitements métiers

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

- De manière à pouvoir réaliser des actions plus complexes (modification du modèle, création de nouveaux Bean, ...) nous dérivons explicitement la classe *Action*
- Cette classe possède la méthode *execute* appelée par le constructeur de l'application Web si aucune erreur ne s'est produite

Struts : Action

- *ActionForward execute(ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)*
  - le paramètre *ActionMapping* est un objet image de la configuration de l'action en cours stockée dans *struts-config.xml*
  - le paramètre *ActionForm* correspond au Bean qui stocke l'information du formulaire
  - le paramètre *HttpServletRequest* est la référence de la requête
  - le paramètre *HttpServletResponse* est la référence de la réponse
  - le retour *ActionForward* est un objet pour identifier la destination prochaine que le contrôleur choisira
- Il faut modifier également *struts-config.xml* en ajoutant au corps de la balise *<action>* la balise *<forward>*
  - *String name* : étiquette pour la re-direction
  - *String path* : chemin de re-direction

Struts : Action

- Épisode 5 : améliorer le traitement des actions du contrôleur

```
public class FormulaireAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest req, HttpServletResponse res) throws Exception {
        Formulaire     formulaire = (Formulaire)form;

        req.setAttribute("nom", formulaire.getNom());
        req.setAttribute("age", formulaire.getAge());

        return mapping.findForward("response");
    }
}
```

Grâce au paramètre *ActionForm* on a accès au contenu du Bean

L'objet requête de la Servlet est modifié en ajoutant deux attributs issus du Bean

On indique ici que la prochaine re-direction se fera dans « response »

```
...
<action
    path="/main" name="formPersonne" scope="session" validate="true"
    input="/erreurs.do" parameter="/vues/main.jsp"
    type="monpackage.FormulaireAction">
        <forward name="response" path="/reponse.do" />
</action>
...
```

Ajout dans le corps de cette action de la balise *<forward>*

L'étiquette « response » indique une nouvelle page cible

Struts : Action

- Épisode 5 (suite) : réponse positive

```
<%@ taglib uri="htmlstruts" prefix="html" %>
<html>
    <head>
        <title>Personne</title>
    </head>
    <body>
        <h2>Personne - Reponse</h2><hr>
    <table>
        <tr>
            <td>Nom</td><td>${nom}
        </tr>
        <tr>
            <td>Age</td><td>${age}
        </tr>
    </table>
    <br>
    <html:link page="/formulaire.do">
        Retour au formulaire
    </html:link>
    </body>
</html>
```

Utilisation des EL dans la page JSP puisque deux attributs ont été définis dans la classe *FormulaireAction* (scope = request)

La balise *<link>* permet de retourner facilement un lien hypertexte

JDBC

Java Database Connectivity

JDBC

JDBC est une API Java (ensemble de classes et d'interfaces défini par SUN et les acteurs du domaine des BD) permettant d'accéder aux bases de données à l'aide du langage Java via des requêtes SQL. Cette API permet d'atteindre de manière quasi-transparente des bases Sybase, Oracle, Informix, ... avec le même programme Java JDBC.

En fait cette API est une spécification de ce que doit implanter un constructeur de BD pour que celle ci soit interrogeable par JDBC. De ce fait dans la programmation JDBC on utilise essentiellement des références d'interface (`Connection`, `Statement`, `ResultSet`, ...).

Sun et les constructeurs de BD se sont charger de fournir (vendre ou donner) des classes qui implémentent les interfaces précitées qui permettent de soumettre des requêtes SQL et de récupérer le résultat.

Par exemple Oracle fournit une classe qui, lorsqu'on écrit

```
Statement stmt =
conn.createStatement();
```

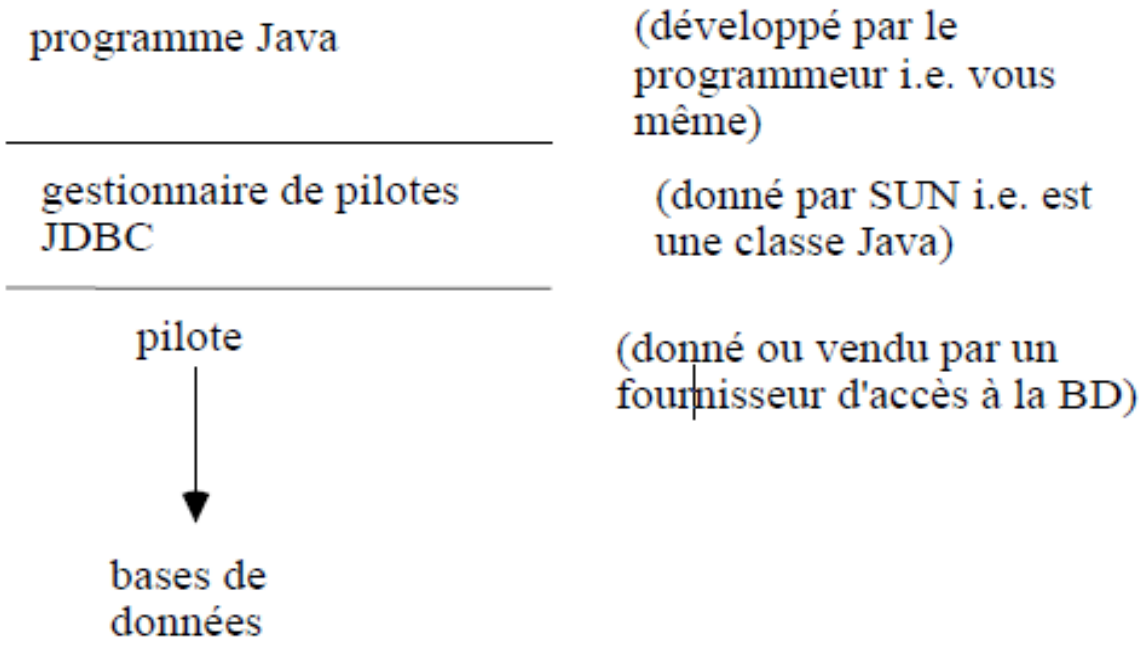
retourne un objet concret (de classe `class OracleStatement implements Statement` par exemple) qui est repéré par la référence `stmt` de l'interface `Statement`.

Pilotes (drivers)

L'ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un gestionnaire de bases de données particulier est appelé un **pilote JDBC**.
Les protocoles d'accès aux BD étant propriétaires il y a donc plusieurs drivers pour atteindre diverses BD.

Parmi les interfaces, l'une d'entre elles, l'interface `Driver`, décrit ce que doit faire tout objet d'une classe qui implémente l'essai de connexion à une base de données. Entre autre, un tel objet doit obligatoirement s'enregistrer auprès du `DriverManager` et retourner en cas de succès un objet d'une classe qui implémente l'interface `Connection`.

Architecture JDBC



Les 4 types de pilotes

Les pilotes sont classés en quatre types :

Ceux dits natifs qui utilisent une partie écrite dans du code spécifique non Java (souvent en langage C) et appelé par ces implantations : les pilotes de classe 1 et 2. Ces pilotes sont rapides mais doivent être présent sur le poste client car ne peuvent pas être téléchargés par le ClassLoader de Java (ce ne sont pas des classes Java mais plutôt des librairies dynamiques). Ils ne peuvent donc pas être utilisés par des applets dans des browsers classiques.

JDBC

Ceux dits 100% Java qui interrogent le gestionnaire de base de données avec du code uniquement écrits en Java : les pilotes de classe 3 et 4.
Ces pilotes peuvent alors être utilisés par des applets dans des browsers classiques.
Plus précisément :
pilote de classe 1 : pilote jdbc:odbc
pilote de classe 2 : pilote jdbc:protocole spécifique et utilisant des méthodes natives.
pilote de classe 3 : pilote écrit en Java jdbc vers un middleware qui fait l'interface avec la BD
pilote de classe 4 : pilote écrit en Java jdbc:protocole de la BD. Accède directement à l'interface réseau de la BD.

Structure d'un programme JDBC

Un code JDBC est de la forme :
- recherche et chargement du driver approprié à la BD.
- établissement de la connexion à la base de données.
- construction de la requête SQL
- envoi de cette requête et récupération des réponses
- parcours des réponses.

Syntaxe

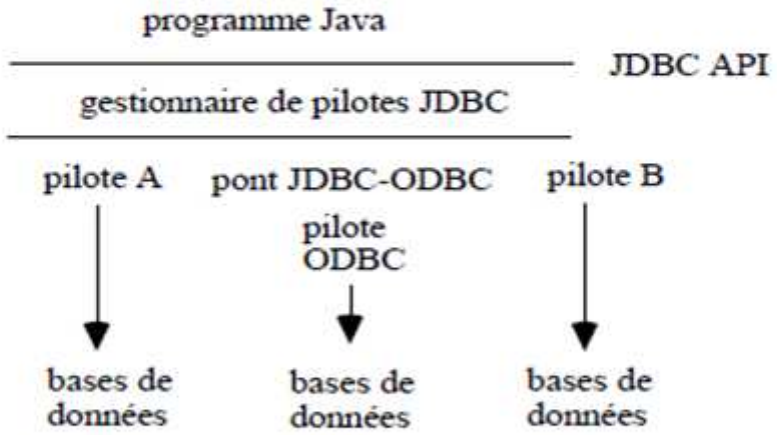
```
Class.forName(" org.apache.derby.jdbc.ClientDriver ");
Connection conX = DriverManager.getConnection( "jdbc:derby://localhost;1527/contact","nbuser","nbuser" );
Statement stmt = conX.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c ... FROM ...
WHERE ...");

while (rs.next()) {
...
// traitement
}
```

La programmation avec JDBC

On utilise le paquetage `java.sql`. La plupart des méthodes lèvent l'exception `java.sql.SQLException`.

Chargement du pilote

On commence un programme JDBC en chargeant dans le programme, le pilote approprié pour la BD. Comme le programme peut interroger divers types de BD il peut avoir plusieurs pilotes. C'est au moment de la connexion que sera choisi le bon pilote par le `DriverManager`.
On a donc une architecture :



Syntaxe :

```
Class.forName(" org.apache.derby.jdbc.ClientDriver ");
```

La connexion

On ouvre une connexion avec une des méthodes `DriverManager.getConnection(...)` qui retourne un objet d'une classe qui implémente l'interface `Connection`. Ces méthodes contiennent comme premier argument une "URL JDBC". Elles recherchent le pilote adapté pour gérer la connexion avec la base repérée par cette URL.

Une URL JDBC doit commencer par `jdbc`. Le second argument est le protocole sous jacent dans lequel le pilote traduit. Le troisième argument est un identificateur de base de données. La syntaxe d'une URL JDBC est donc:

`jdbc:`*<sous-protocole>*`:`*<baseID>*

la partie *baseID* est propre au *sous-protocole*.

Les arguments suivants de `Driver.getConnection(...)` sont des informations nécessaires à l'ouverture de la connexion souvent un nom de connexion à la base suivi d'un mot de passe.

```
Connection conX = DriverManager.getConnection(URLjdbc,
"dbUser1", "pwuser1");
```

La classe `DriverManager`

Cette classe est une classe qui ne contient que des méthodes statiques.
Elle fournit des méthodes qui sont des utilitaires pour gérer l'accès aux bases de données par Java et les différents drivers JDBC à l'intèrieur d'un programme Java.

Finalement on ne créée ni ne récupère d'objet de cette classe.

Requêtes au bases de données

Syntaxiquement en Java on utilise `executeQuery(...)` si la requête est une requête SELECT et `executeUpdate(...)` si la requête est une requête d'action.

Récupération des résultats (suite)

Les résultats des requêtes SELECT ont été mis dans un `ResultSet`. Cet objet récupéré modélise le résultat qui peut être vu comme une table. Par exemple
SELECT nom, prenom, adresse FROM Personnes
retourne un `ResultSet` qui modélise :

Nathalie	Berthe	Paris
Patrick	Dupont	Paris
Terence	Field	Paris
.	.	.
.	.	.
.	.	.

Un pointeur géré par Java jdbc permet de parcourir tout le `ResultSet` et est initialisé : il est automatiquement positionné avant la première ligne de résultat.

On parcourt alors tout le `ResultSet` pour avoir l'ensemble des réponses à la requête. La boucle de parcours est :

```
while( rs.next() ) {
// traitement
}
```

Récupération des résultats (fin)

Les colonnes demandées par la requête sont numérotées à partir de 1 (culture IBM-SQL et non pas culture Unix-C).

Remarque
La numérotation est relative à l'ordre des champs de la requête et non pas l'ordre des champs de la table interrogée (évidemment).

De plus les colonnes sont typées en type SQL. Pour récupérer une valeur de colonne il faut donc indiquer le numéro de la colonne ou son nom et faire la conversion de type approprié. Par exemple :

```
Statement stmt = ...;
ResultSet rs = stmt.executeQuery(SELECT nom, prenom, age,
date FROM LaTable");
String leNom = rs.getString( 1 ) );
String lePrenom = rs.getString( 2 );
int lage =  rs.getInt ( 3 );
Date laDate = rs.getDate( 4);
```

On peut aussi désigner les colonnes par leur nom dans la BD (c'est moins rapide mais plus lisible) et réécrire :

```
ResultSet rs = stmt.executeQuery("SELECT nom, prenom,
age, date FROM LaTable");
String leNom = rs.getString( "nom") );
String lePrenom = rs.getString( "prenom" );
int lage =  rs.getInt ( "age" );
Date laDate = rs.getDate( "date ");
```

Java

Les correspondances entre type SQL et type Java sont données par les spécifications JDBC. En voici certaines :

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Conclusion : finalement comment cela fonctionne ?

Après l'instruction

```
Connection conX = DriveManager.getConnection(URLjdbc,
"dbUser1", "pwuser1");
```

Chargement du pilote de la base de données

```
Class.forName("piloteToBD");
```

Ce pilote permet de faire une connexion à la base de données et retourner un objet d'une classe qui implémente l'interface Connection.

Le fonctionnement (suite)

Par la suite cet objet repéré par `conX` va retourner, à la suite de l'instruction,

```
Statement smt = conX.createStatement();
```

Enfin l'instruction

```
ResultSet rs = smt.executeQuery( "SELECT * FROM ' utilisateur " );
```

demande le lancement de la méthode `executeQuery(...)` sur un objet d'une classe qui implémente `Statement` et qui est propre à la base et cela fonctionne et est universel (pour la troisième fois ;-)).

PreparedStatement

Lors de l'envoi d'une requête pour exécution 4 étapes doivent être faites :

- analyse de la requête
- compilation de la requête
- optimisation de la requête
- exécution de la requête

PreparedStatement

Lors de l'envoi d'une requête pour exécution 4 étapes doivent être faites :
- analyse de la requête
- compilation de la requête
- optimisation de la requête
- exécution de la requête

et ceci même si cette requête est la même que la précédente !! Or les 3 premières étapes ont déja été effectuées dans ce cas.

Les bases de données définissent la notion de requête préparée, requête où les 3 premieres étapes ne sont effectuées qu'une seule fois. JDBC propose l'interface `PreparedStatement` pour modéliser cette notion. Cette interface dérive de l'interface `Statement`.

PreparedStatement : Syntaxe

Même sans paramêtres, les `PreparedStatement` ne s'utilisent pas comme des `Statement`.
Au lieu d'écrire :

```
Statement smt = conX.createStatement();
ResultSet rs = smt.executeQuery( "SELECT * FROM Livres" );
```

on écrit :

```
PreparedStatement pSmt = conX.prepareStatement("SELECT
* FROM Livres" );
ResultSet rs = pSmt.executeQuery();
```

à savoir la requête est décrite au moment de la "construction" pas lors de l'exécution qui est lancée par `executeQuery()` sans argument.

Requêtes paramétrées

On utilise donc les `PreparedStatement` et on écrit des requêtes de la forme :

```
SELECT nom FROM Personnes WHERE age > ?
AND adresse = ?
```

Puis on utilise les méthodes `setType(numéroDeLArgument, valeur)` pour positionner chacun des arguments.
Les numéros commencent à 1 dans l'ordre d'apparition dans la requête.
On a donc un code comme :

```
PreparedStatement pSmt = conX.prepareStatement("SELECT
nom FROM Personnes WHERE age > ? AND adresse = ?" );
pSmt .setInt(1, 22);
pSmt .setString(2, "Turin");
ResultSet rs = pSmt.executeQuery();
```

JSTL

- 'Depuis la version *JSP 1.2*
- Spécification développée par le groupe d'experts *JSR 52*
- Collection de Tag Librairies personnalisées qui implémentent la plupart des fonctions communes aux pages web:
  - Itérations et conditions (core)
  - Formatage des données (format)
  - Manipulation de XML (xml)
  - Accès au bases de données (sql)
- Utilisation du langage EL (Expression Language)
- Avantages
  - Code simple, lisible et facile à maintenir
  - Le concepteur de page est libéré de code Java
  - Évite au développeur d'écrire à chaque fois les fonctions de bases.

EL: Expression Language

- Les expressions d'EL s'écrivent sous les formes :
  - `${expression}` ou
  - `#{expression}`
- Toutefois dans certaines circonstances (par exemple dans les tags personnalisés (custom tags)), la notation `${expression}` implique une évaluation immédiate.
- Tandis qu'alors, `#{expression}` signifie que l'évaluation de l'expression sera différée (jusqu'à l'exécution du tag).

JSTL

- Un identificateur dans *EL* fait référence à une variable retournée par l'appel de `pageContext.findAttribute(identificateur)` et qui est dans la portée (scope): `page`, `request`, `session` ou `application`.

  `${ var } = pageContext.getAttribute("var")`

- Objets implicites:
  - `pageScope`, `requestScope`, `sessionScope`, `applicationScope`

- Accès au paramètres d'une requête *HTTP* via `param` (objet de type Map) et `paramValue`

- Un objet implicite `pageContext` qui donne accès aux propriétés associées au contexte de la page JSP

Objets prédéfinis

`applicationScope` : `Map` contenant les paires (nom, valeur) des variables de portée (scope) application

`sessionScope` : idem de portée session

`requestScope` : idem de portée request

`pageScope` : idem de portée page

`cookie` : idem pour les cookies

`initParam` : idem pour les paramètres d'initialisation

`param` : `Map` contenant les paires (nom, 1ère valeur) des paramètres

`header` : idem pour les headers de request.

`paramValues`: `Map<String,String[]>` contenant les paires (nom, tableau de toutes les valeurs) des paramètres

`headerValues` : idem pour les headers

`pageContext` : le `PageContext`

- Les expressions d'EL peuvent être utilisée dans
  - le corps de la page (template text)

```
<table border="1">
<tr><td>Hello ${param[nom]}</td>
```

L'opérateur [] pour accéder au objets de type Map, Array et List :
Ex : param["p1"] <=> param.get("p1")

Les opérateurs

Par préséance décroissante :

```
[] .
() (les parenthèses)
- (unaire) not ! empty
* / div % mod
+ - (binaire)
< > <= >= lt gt le ge
== != eq ne
&& and
|| or
```

Les opérateurs `!` et `not` sont identiques.

De même pour

- `/` et `div`
- `%` et `mod`
- `<` et `lt`
- `>` et `gt`
- `<=` et `le`
- `>=` et `ge`
- `==` et `eq`
- `!=` et `ne`
- `&&` et `and`
- `||` et `or`

JSTL

Les opérateurs `.` et `[]` sont interchangeables :

On peut écrire

`${user.nom}` ou `${user["nom"]}`

On les utilise pour l'accès aux champs et l'accès au éléments des Maps :

`${param["nom"]}` ou `$param.nom`

Accès aux variables

On peut accéder aux variables stockées dans les différentes portées comme suit :

`${sessionScope.caddie}` en précisant la portée ou

`${caddie}` sans préciser la portée. Dans ce cas, EL utilise le `pageContext` pour la chercher d'abord dans la page, puis la request, puis la session et finalement l'application, renvoyant la première trouvée.

Les propriétés peuvent être emboîtées :

Si `caddie` contient une `Collection articles`, on peut écrire `${caddie.articles[0].prix}` pour accéder au premier article (dans l'ordre de `l'Iterator`).

Désactiver EL

- Dans une page :

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

- Dans un groupe de pages, via web.xml :

```
<jsp-config>
    <jsp-property-group>
            <url-pattern>/pasDEL/</url-pattern>
            <el-ignored>true</el-ignored>
    </jsp-property-group>
</jsp-config>
```

- On peut préciser seulement une page dans <url-pattern> :

```
<url-pattern>pasdel.jsp</url-pattern>
```

ou toutes avec `*.jsp`

Désactiver les scriptlets

A l'inverse, on peut interdire les scriptlets dans une, un groupe ou toutes les pages via le fichier web.xml :

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>
            /pasDeScripts/
            < !--*.jsp -- >
            < !--/test.jsp -- >
        </url-pattern>
        <scripting-invalid>
          true
        </scripting-invalid>
    </jsp-property-group>
</jsp-config>
```

JSTL

Depuis la version 1.0 parue en 2002, nous pouvons utiliser une librairie de tags qui permettent d'éviter l'écriture de scriptlets dans les pages jsp.

Les 4 tag libraries

- c.tld
  - core : logique de base
- fmt.tld
  - formatage et internationalisation
- sql.tld
  - accès aux bases de données
- x.tld
  - traitement de fichier xml

Installation de JSTL

- Downloadez JSTL à l'adresse http://jakarta.apache.org/taglibs. Suivez Downloads et choisissez Standard 1.1 Taglib
- Le site est redirigé vers : http://tomcat.apache.org/taglibs/index.html
- Les download : http://tomcat.apache.org/taglibs/standard/
- By Maven : http://jstl.java.net/download.html
- Décompressez l'archive.
- On peut alors travailler de deux manières
  - Placer standard.jar et jstl.jar dans le WEB-INF/lib de votre projet : Ils seront alors contenu dans votre fichier .war
  - ou placer standard.jar et jstl.jar dans la directory lib de votre serveur d'application. Pour JBoss 4.2.1.GA, c'est inutile, il contient déjà jstl. Vous n'avez donc rien à faire. Veillez néanmoins à ce que ces deux jar soient dans votre classpath (le Build Path de votre projet dans Eclipse). Pour JBoss 4.2.1.GA, vous devez seulement ajouter %JBOSS_HOME%/server/default/deploy/jboss-web.deployer/jstl.jar dans le Build Path. (il contient les deux jar de JSTL)

Utilisation de JSTL

- Pour que votre projet puisse utiliser JSTL, il faut encore préciser dans votre web.xml où se trouvent les fichier .tld de JSTL
- Ouvrez standard.jar avec WinRar. Si vous travaillez avec JBoss, ouvrez leur version de jstl.jar avec WinRar. dans les deux cas déplacez-vous dans META-INF et extrayez les fichiers c.tld , fmt.tld, sql.tld et x.tld. Placez les dans la directory WEB-INF/tld de votre application.
- Ces 4 tld correspondent aux 4 tag libraries de jstl

Fonction de base : La librairie Core

La librairie Core

- On la déclarre avec
- `<%@ taglib uri= "http://java.sun.com/jstl/core" prefix ="c" %>`
- Dans web.xml on ajoute la déclaration de cet uri

```
<jsp-config>
    <taglib>
        <taglib-uri> http://java.sun.com/jstl/core
        </taglib-uri>
        <taglib-location>
       /WEB-INF/tld/c.tld
        </taglib-location>
    </taglib>
</jsp-config>
```

Fonctions de base

- Affichage

```
<c: out  value=" expression " /> → <%=  expression %>
```

- Affectation

```
<c:set  value="value" var=" varName " scope=" application " />
<% pageContext.setAttribute("varName",value,SCOPE) %>
```

- Exception java.lang.Throwable

```
<c:catch [var="varName"] >
    actions a surveiller
</c:catch>

<% try{
    actions à surveiller
    }catch(Throwable varName){}
%>
```

Les conditions

1- simple *if(cond)*

```
<c:if test="${user.visitCount = = 1}">
  <c:out value="Première visite.Bienvenue!" />
</c:if>
```

```
<% if(user.visitCount == 1){ %>
<%= "Prmière visite.Bienvenue" %>
<% } %>
```

2-choix multiple *if/else*

```
<c:choose>
<c:when test="${count == 0}">
   Pas de visite!
</c:when>
<c:otherwise>
<c:out value="${count}"/> visiteurs.
</c:otherwise>
</c:choose>
```

```
<% If(count == 0){ %>
<%= Votre compte est vide %>
<% }else{ %>
<%= count+"visiteurs" %>
<% } %>
```

Les itérations avec la boucle *for/while* *en JSP*

```
<%@page import="java.util.*" %>
. . . .
<% Member user = null;
   Collection users = session.getAttribute("members");
   Iterator it =  users.iterator();
   while(it.hasNext()){
        user = (Member) it.next();
%>
<%= "nom:  "+user.getName() %>
<% } %>
```

Les itérations avec la boucle ***for/while***

forEach

```
<c:forEach var="user" items="sessionScope.members" [begin] [end] [step]>
 <c:out value="nom:  ${user.name}" />
</c:forEach>
```

Librairie de Formattage

- On la déclarre avec
- `<%@ taglib uri= "http://java.sun.com/jstl/fmt" prefix ="fmt" %>`
- Dans web.xml on ajoute la déclaration de cet uri

```
<jsp-config>
    <taglib>
        <taglib-uri> http://java.sun.com/jstl/fmt
        </taglib-uri>
        <taglib-location>
       /WEB-INF/tld/fmt.tld
        </taglib-location>
    </taglib>
</jsp-config>
```

Librairie de Formatage : Internationalisation (i18n) - 1

- <fmt:setLocale value="…" [scope="…"]/>

  où value est par exemple "en_US", "fr_FR" ou "fr_BE" ou simplement "en", "fr", …

- On peut définir une Locale par défaut dans web.xml

```
<context-param>
 <param-name>
        javax.servlet.jsp.jstl.fmt.locale
 </param-name>
 <param-value>en</param-value>
</context-param>
```

- Idem avec fallbackLocale si le Locale demandé n'est pas supporté par le serveur

Librairie de Formatage : Internationalisation (i18n) - 2

- ResourceBundle :
- ```
  ResourceBundle mesTrads =
  ResourceBundle.getBundle("MesTraductions", Locale.FRENCH);
  ```
- ```
  public class MesTraductions extends PropertyResourceBundle
  {
    MesTraductions() {
   super(new
   FileInputStream("mestrad.properties");
    }
  }
  ```
- mestrad.properties
  - firstName=prénom
  - lastName=nom

Librairie de Formatage : Internationalisation (i18n) - 3

- <fmt:setBundle baseName="…" [var="…] [scope="…"]/>
- baseName indique le nom de la classe du ResourceBundle : ex : "MesTraductions" y compris le nom de package.
- var, si présent, sert à stocker le bundle dans une variable qu'on précisera lors d'un emploi futur
- Si var n'est pas présent, on définit ainsi le ResourceBundle par défaut
- Il suffit de placer le fichier MesTraductions_fr.properties ou MesTraductions_fr_FR.properties dans la directory src de votre projet. Le reste est automatique.

Librairie de Formatage : Internationalisation (i18n) - 4

- On peut préciser un bundle par défaut dans web.xml
- <context-param>

```
<param-name>
    javax.servlet.jsp.jstl.fmt.localizationContext
</param-name>
<param-value>messages.MesTraductions</param-value>
</context-param>
```

Librairie de Formatage : Internationalisation (i18n) - 5

- <fmt:message key="…" [bundle="…"] [var="…"] [scope="…"]/>
- Sans var, affiche le message correspondant à la clé indiquée. La recherche de cette clé se fait dans le ResourceBundle par défaut sauf si l'attribut bundle est présent. Il indique le contenu d'une variable définie avec <setBundle … var="…"/>
- Si var est présent, le message est placé dans la variable.

Librairie de Formatage : Internationalisation (i18n) - 6

- Si le message attend des paramètres, on utilise
- <fmt:param value="objet" />
  - <fmt:message key="${clef}">
    <fmt:param value="${prenom}"/>
    <fmt:param value="${nom}"/>
    </fmt:message>
  - Dans le fichier .properties on a clef=Bonjour {0} {1}

Librairie de Formatage : Internationalisation (i18n) - 7

- <fmt:bundle baseName="…" [prefix="…"]>
- définit un bundle localement
- baseName, comme dans setBundle
- prefix pour être employé dans les fmt:message
  - <fmt:bundle baseName="MesTraductions">
    <fmt:message key="trad.nom"/>
    <fmt:message key="trad.age"/>
    </fmt:bundle>

Librairie de Formatage : Internationalisation (i18n) - 8

- Avec un préfixe :
  - <fmt:bundle baseName="MesTraductions" prefix="trad" >
    <fmt:message key="nom"/>
    <fmt:message key="age"/>
    </fmt:bundle>

SQL

- *Faire des requêtes*
- *Accès au résultat simplifié*
- *Faire des mises à jour*
- *Faire des transactions*

La librairie SQL - 1

» <sql:setDataSource

{dataSource="*dataSource*" |

url="*jdbcUrl*"

[driver="*driverClassName*"]

[user="*userName*"]

[password="*password*"]}

[var="*varName*"]

[scope="{page|request|session|application}"]/>

» Si on utilise dataSource, il faut y mettre une entrée JNDI qui sera préfixée par java:comp/env/

» sinon on utilise l'url, le driver, le user et le password comme en JDBC

JSTL

Data source est de type `Javax.sql.DataSource`

```
<%@page import="java.sql.*,javax.sql.*" %>
<% Connection con = dataSource.getConnection;
   Statement stm = con.createStatement();
   ResultSet customers = stm.executeQuery("SELECT * FROM customers
                                    WHERE country = ' France '
                                          ORDER BY lastname");
%>
<table>
<% while(customers.next()){ %>
<tr>
     <td><%= customers.getString("lastName") %></td>
     <td><%= customers.getString("lastName") %></td>
     <td><%= customers.getString("lastName") %></td>
</tr>
<% } %>
</table>
```

La librairie SQL - 2

» *Syntaxe 1: Sans body*

```
<sql:query sql="sqlQuery"
var="varName" [scope="{page|request|session|application}"]
[dataSource="dataSource"]
[maxRows="maxRows"]
[startRow="startRow"]/>
```

JSTL

Data source est de type `Javax.sql.DataSource`

```
<sql:query var="customers" dataSource="${dataSource}">
SELECT * FROM customers
WHERE country = 'Algeria'
ORDER BY lastname
</sql:query>

<table>
<c:forEach var="row" items="${customers.rows}">
<tr>
<td><c:out value="${row.lastName}"/></td>
<td><c:out value="${row.firstName}"/></td>
<td><c:out value="${row.address}"/></td>
</tr>
</c:forEach>
</table>
```

La librairie XML

- définit des actions
  - de type core
  - de type contôle
  - de type transformation
- nécessite xalan.jar
  - soit dans WEB-INF/lib de votre projet
  - soit dans une directory chargée par le serveur d'application (dans JBoss, il se trouve dans %JBOSS_HOME%\lib\endorsed)

JSTL

- **<x:parse>** parse un document XML par sa DTD
- **<x:out>** Évalue une expression Xpath et affiche le résultat
- **<x:transform>** applique les transformations d'une feuille de style XSLT sur un document XML

La librairie XML

» Considérons le fichier livre.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<livre>
   <titre>
Professional JSP 2, 4th Edition
</titre>
   <auteur>Brown et. al</auteur>
   <isbn>1-59059-513-0</isbn>
   <publié>December 2005</publié>
   <éditeur>Apress</éditeur>
<url>
http://www.apress.com/book/bookDisplay.html?bID=464
</url>
</livre>
```

La librairie XML

- On pourra parser ce fichier en faisant

```
<c:import url="livre.xml" var="url"/>

<x:parse doc="${url}" var="livre"/>
```

- Puis on accédera aux données du fichier à l'aide de la variable livre et par navigation :

```
<x:out select="$livre/livre/titre"/>

<x:out select="$livre/livre/auteur"/>
```