Application WEB : les bonnes pratiques

Table des matières

1 Structure des applications WEB **1**

1.1 Réorganisation des pages JSP 2

2 Architecture MVC (Modèle, Vue, Contrôleur) **3**

2.1 MVC et couche métier indépendante 3

2.2 Les vues 4

2.3 Le modèle 4

2.4 Le contrôleur 4

2.5 Enchaînement 5

2.6 Démarche M.V.C. 5

3 La démarche M.V.C. pour l'annuaire **6**

3.1 Représentation d'un utilisateur 6

3.2 Configurer les pages JSP 6

3.3 La page « `/index.jsp` » 7

3.4 La vue « `/loginForm.jsp` » 7

3.5 La vue « `/home.jsp` » 8

3.6 La vue « `/message.jsp` » 8

3.7 Interface vers les vues 9

3.8 La couche métier 9

3.9 Déclarer la servlet contrôleur 9

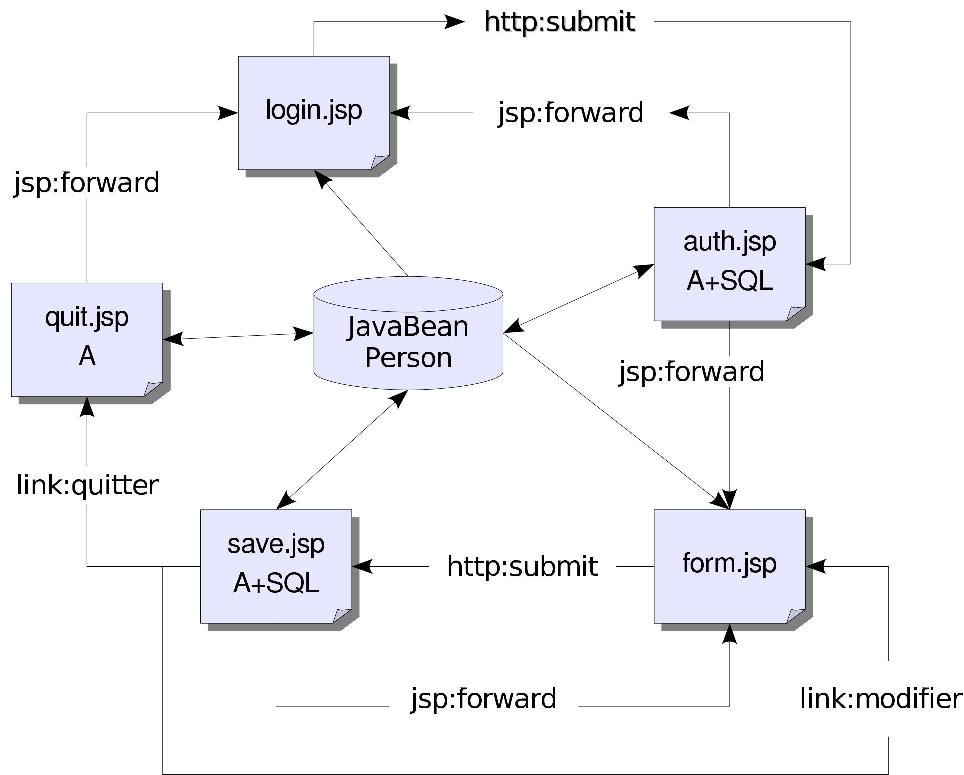
3.10 Le contrôleur de notre application 10

4 Quelques derniers conseils **15**

1 Structure des applications WEB

Une application de modification d'un annuaire :

Description fonctionnelle : Un utilisateur se connecte au système (page `login.jsp`) en donnant un nom et un mot de passe. La soumission se connecte sur la page `auth.jsp` qui vérifie l'identité. En cas d'erreur, un *forward* est effectué sur la page `login.jsp`. En cas de réussite, le Bean est chargé depuis la base et un *forward* est effectué vers la page `form.jsp`. Cette dernière se charge d'afficher un formulaire de modification. La validation se connecte sur la page `save.jsp`. cette dernière enregistre les données du formulaire dans le bean est valide ce dernier en vérifiant la conformité des données entrées par l'utilisateur. Si les données sont valides, la page `save.jsp` sauvegarde le bean et affiche un message indiquant la réussite. Si les données ne sont pas valides, la page `save.jsp` effectue un *forward* vers la page `form.jsp` afin que l'utilisateur corrige les erreurs.



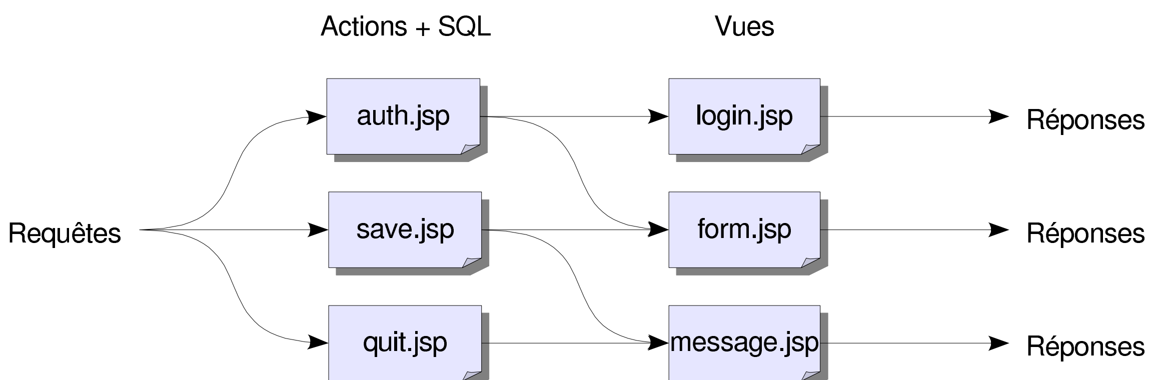
Dans cet exemple, les symboles « A » indiquent des actions (du code Java qui agit sur les données) et « SQL » des accès à la couche de stockage. Les flèches qui arrivent sur le JabaBean indiquent une modification et celles qui partent une utilisation. A ce stade nous pouvons faire les remarques suivantes :

- Les actions sont réparties dans trois pages. Cela ne facilite pas la maintenance du logiciel et son évolutivité (éparpillement de la logique applicative).
- Les pages JSP ne sont pas homogènes : certaines se contentent de construire une page HTML alors que d'autres sont responsables de la logique applicative.
- Il existe deux points d'entrée (`auth.jsp` et `save.jsp`). Chacun doit vérifier avec soin la nature des données fournis en entrée (paramètres de la requête). Un seul point d'entrée aurait facilité la conception d'un code sécurisé.

Il est très difficile de développer des applications complexes et/ou importantes en adoptant cette structure éclatée. Nous allons donc proposer une organisation plus rationnelle pour les applications WEB :

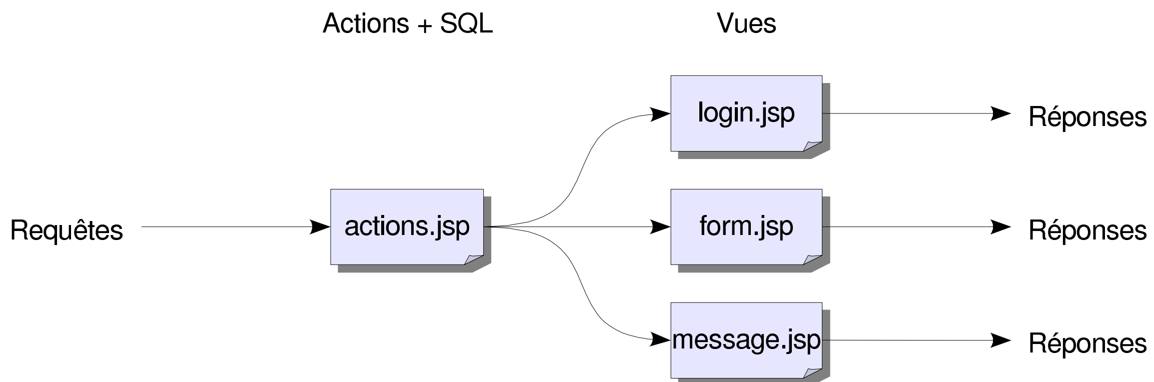
1.1 Réorganisation des pages JSP

Il faut distinguer les pages qui agissent et celles qui produisent les réponses :



Pour simplifier et améliorer la qualité de l'application, il est nécessaire d'effectuer cette séparation. Ce ne sont pas les mêmes personnes qui s'occupent de ces pages (par exemple un graphiste pour les vues).

Il faut une entrée unique :

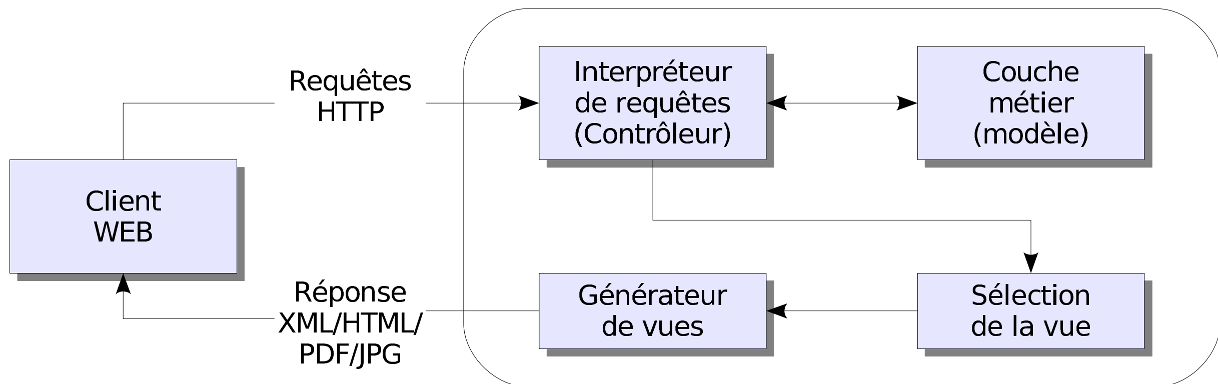


Pour améliorer la qualité et la sécurité, il est nécessaire de mettre en place une porte d'entrée unique qui traite toutes les demandes.

2 Architecture MVC (Modèle, Vue, Contrôleur)

Trois parties :

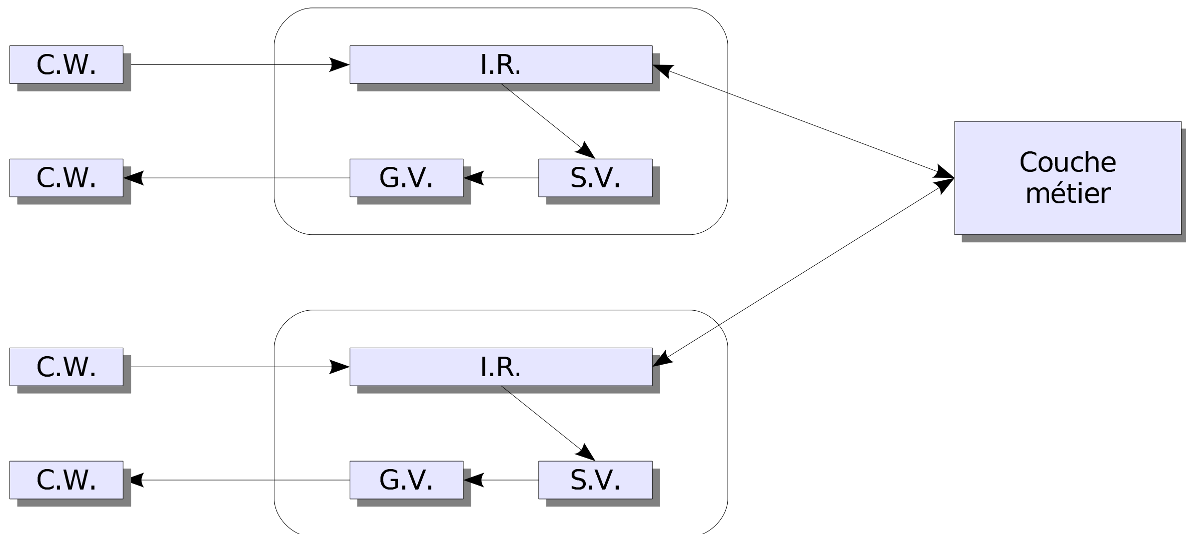
- Le **contrôleur** a la charge de traiter les requêtes du client.
- Le **modèle** gère les données de l'application.
- La **vue** construit le résultat d'une requête.



C'est une architecture concue pour les I.H.M. (Interfaces Homme/Machines).

2.1 MVC et couche métier indépendante

Plusieurs applications peuvent interagir avec la même couche métier :



2.2 Les vues

• Les vues sont basées sur des technologies de production de contenu (JSP, XML/XSLT, XSL-FO, PDF, XQuery).

• Les vues ne modifient pas les données (problème avec JSP).

• Une vue peut être interprétée comme un processus de fabrication à partir de données fournies en entrée (pas de logique applicative).

2.3 Le modèle

• Le modèle à la charge de représenter les données du domaine et de fournir les méthodes permettants l'accès et la modification.

• Le modèle est indépendant de la logique applicative.

• Si une couche métier existe, elle représente le modèle.

• Dans une architecture orientée service, le modèle est l'ensemble des services du système d'information.

2.4 Le contrôleur

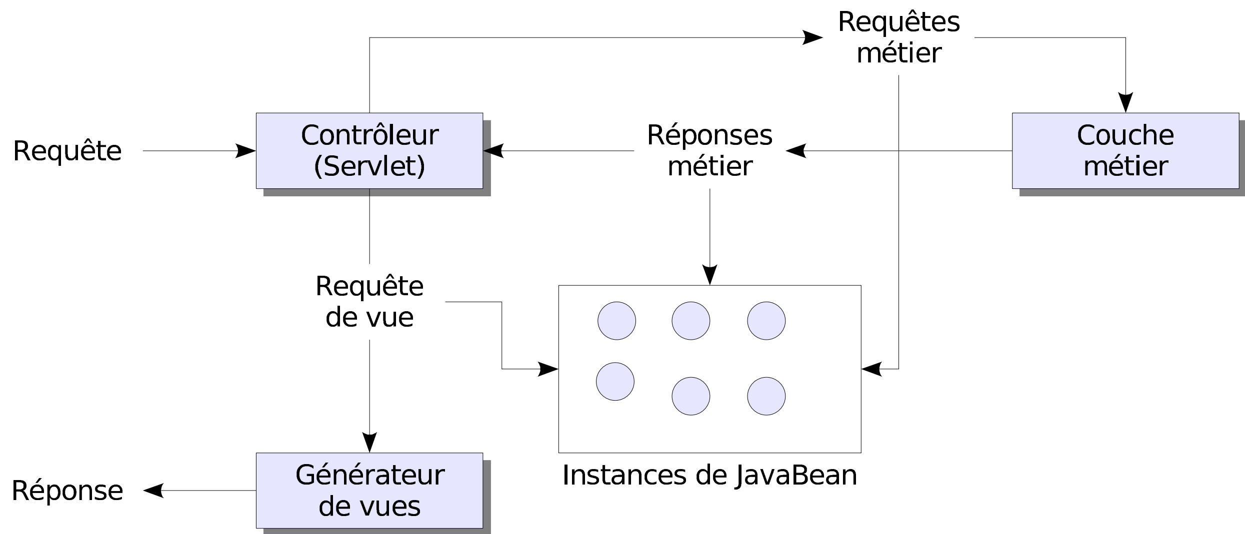
• Le contrôleur est basé sur des technologies de traitement de requêtes (Servlet, C#, C++, Ruby, Python).

• Le contrôleur peut être unique ou multiple.

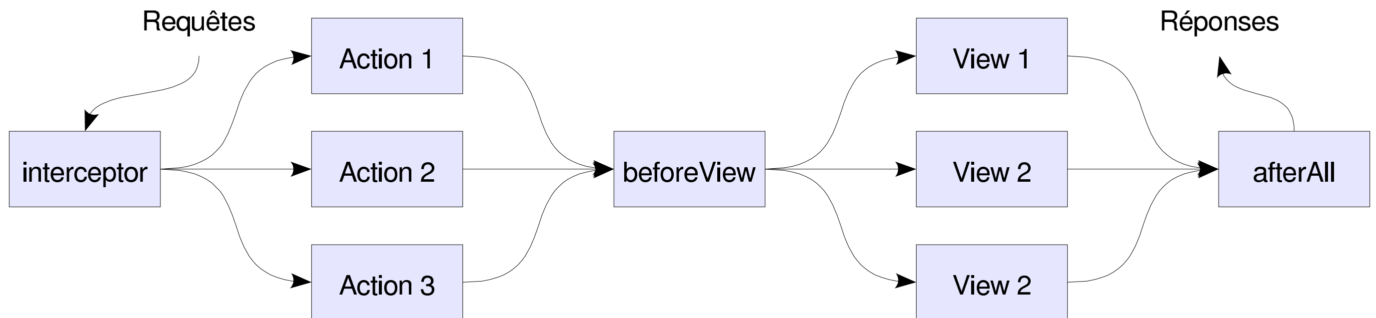
• Le contrôleur est responsable de la logique applicative.

• Il transforme les requêtes utilisateur en requêtes métier (vérification des données entrantes).

• Il choisit la vue et lui fournit les données.



2.5 Enchaînement



• L'étape**beforeAll** traite toutes les requêtes (contrôle de la machine cliente, de l'authentification, ouverture de session JPA, etc...)

• L'**action** traite une demande spécifique et choisie la vue.

• L'étape **beforeView** traite la demande de vue (vérification, aiguillage en fonction du contexte).

• Les **Vues** produisent les résultats.

• L'étape **afterAll** assure la fermeture des ressources allouées (session, temps d'exécution).

2.6 Démarche M.V.C.

Quatre étapes :

1. Dresser la liste des vues (maquette HTML),
   - nom de la vue,
   - type de données généré (XHTML/XML/PDF),
   - paramètres (nom, type, obligatoire/facultatif),
   - requêtes engendrées par la vue,

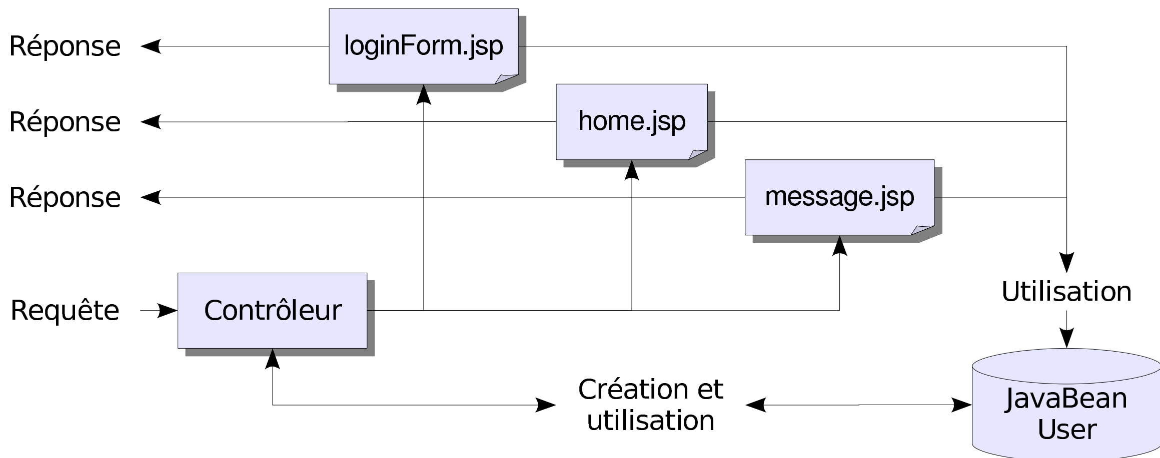
2. Implantation et test des vues.

3. Dresser la liste des requêtes
   - nom de la requête (URL),
   - paramètres (nom, type, obligatoire/facultatif),
   - contexte d'utilisation,
   - vues utlisées.

4. Implantation et test des requêtes.

3 La démarche M.V.C. pour l'annuaire

Architecture :



3.1 Représentation d'un utilisateur

Chaque utilisateur de l'application va être représenté par une instance d'un javaBean de porté session :

```
package fr.exemple.values;

public class User {

    // properties
    private String id = "";
    private String name = "";
    private String password = "";
    private Boolean auth = false; // important !

    public User() {
    }

    // getters and setters
    ...

}
```

3.2 Configurer les pages JSP

Nous ajoutons dans le fichier `web.xml` :

```
<jsp-config>
    <jsp-property-group>
        <description>Toutes les pages</description>
        <url-pattern>*.jsp</url-pattern>
        <page-encoding>UTF-8</page-encoding>
        <include-prelude>/prelude.jsp</include-prelude>
        <include-prelude>/copyright.jsp</include-prelude>
        <include-coda>/coda.jsp</include-coda>
    </jsp-property-group>
</jsp-config>
```

Cette configuration va inclure les pages /prelude.jsp et /copyright.jsp au début et /coda.jsp à la fin de chaque page JSP.

Le code de la page /prelude.jsp :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url var="charte" value="/charte.css" />

<head>
    <title>Annuaire en ligne</title>
    <link rel="stylesheet" type="text/css" href="${charte}" />
</head>
<body>
```

Le code de la page /copyright.jsp :

```
<!-- (c) JL Massat 2007 -->
```

Le code de la page /coda.jsp :

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

3.3 La page « /index.jsp »

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:redirect url="/login.do" />
```

Cette page JSP donne la réponse HTTP suivant :

```
HTTP/1.1 302 Déplacé Temporairement
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=D96D1EA6FA9E99A50E383C7E9E0B0995; Path=/cours-JSP
Location: http://monserveur.fr:8080/cours-JSP/login.do;jsessionid=D96D1EA6FA9E..
Content-Type: text/html;charset=UTF-8
Content-Length: 0
Date: Thu, 17 Dec 2009 11:18:59 GMT
Connection: close
```

Dans cet exemple, le contexte de l'application WEB est /cours-JSP et la balise <c:redirect.../> se charge de transformer l'URL locale /login.do en l'URL globale équivalente.

3.4 La vue « /loginForm.jsp »

Paramètres :
- message de type String (facultatif),
- user de type User (obligatoire),

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url var="login" value="/login.do" />

<h1>Authentification</h1>
<p><c:out value="${message}" default="Identifiez-vous :"/></p>

<form method="POST" action="${login}">
    <label>Login :</label>
    <input name="id" type="text"
        value="<c:out value="${user.id}"/>"/>
    <label>Password : </label>
    <input name="password" type="password"
        value="<c:out value="${user.password}"/>"/>
    <br/>
    <label>Validation :</label>
    <input name="ok" type="submit" value="Ok"/>
</form>
```

3.5 La vue « /home.jsp »

Paramètres :
- user de type User (obligatoire),
- users de type java.util.Set<User> (obligatoire),

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url var="logout" value="/logout.do" />
<c:url var="editForm" value="/editForm.do" />

<h1>Bienvenue <c:out value="${user.name}"/></h1>

<p>Vous pouvez <a href="${logout}">nous quitter</a> ou
<a href="${editForm}">modifier vos informations</a>.</p>

<p>Utilisateurs :</p>
<ul><c:forEach var="u" items="${users}">
    <li>${u.name}</li>
</c:forEach></ul>
```

Requêtes engendrées :
- /logout.do (pas de paramètre)
- /editForm.do (pas de paramètre)

3.6 La vue « /message.jsp »

Paramètre :
- message de type String (obligatoire),

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url var="index" value="/index.jsp" />

<h1><c:out value="${message}"/></h1>
<p><a href="${index}">Retour</a>.</p>
```

Requête engendrée :
- /index.jsp (pas de paramètre)

3.7 Interface vers les vues

```
package fr.exemple.webapp;

import java.util.Set;
import javax.servlet.http.HttpServletRequest;
import fr.exemple.values.User;

public class Views {

    String loginForm(HttpServletRequest request, User user, String message) {
        request.setAttribute("user", user);
        request.setAttribute("message", message);
        return ("/loginForm.jsp");
    }

    String home(HttpServletRequest request, User user, Set<User> users) {
        request.setAttribute("user", user);
        request.setAttribute("users", users);
        return ("/home.jsp");
    }

    String message(HttpServletRequest request, String message) {
        request.setAttribute("message", message);
        return ("/message.jsp");
    }
}
```

3.8 La couche métier

Un exemple de couche métier :

```
package fr.exemple.business;
import fr.exemple.values.User;

public class Business {

    void authenticate(User p) throws BusinessException;

    void updatePerson(User p) throws BusinessException;

}
```

3.9 Déclarer la servlet contrôleur

Avant de décrire le contrôleur, nous devons le déclarer dans le fichier `web.xml` :

```
<!-- Déclaration de la servlet contrôleur -->
<servlet>
   <servlet-name>Cont</servlet-name>
   <servlet-class>fr.exemple.webapp.MyControler</servlet-class>
   <init-param>
      <description>Répertoire de stockage</description>
      <param-name>dataDir</param-name>
      <param-value>/tmp/donnees</param-value>
   </init-param>
   <load-on-startup>2</load-on-startup>
</servlet>

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

Ce contrôleur gère les URL qui se terminent par `.do` et possède un paramètre d'initialisation (`dataDir`).

3.10 Le contrôleur de notre application

```
package fr.exemple.webapp;

import ...

public class MyControler extends HttpServlet {

    private Business business = null;
    final private Set<User> users = new HashSet<User>();
    final private Views views = new Views();

    final private String loginUrl = "/login.do";
    final private String logoutUrl = "/logout.do";

    // démarrage et arret du contrôleur
    ...

    // traitement des sessions
    ...

    // traitement des erreurs et des actions
    ...

    }
```

Démarrage et arret du contrôleur :

```
// Démarrage du contrôleur : création de la couche métier
@Override
final public void init(ServletConfig c) throws ServletException {
    Business b = new Business();
    b.setDataDir(c.getInitParameter("dataDir"));
    b.init();
    business = b;
}

// Arret du contrôleur
@Override
final public void destroy() {
    business.close();
}
```

Obtenir l'utilisateur associé à la session :

```
public User getUser(HttpServletRequest request) {
    HttpSession session = request.getSession();
    synchronized (session) {
        Object o = session.getAttribute("user");
        if (o instanceof User) {
            return (User) o;
        }
        User newUser = new User();
        session.setAttribute("user", newUser);
        // espionner la session
        session.setAttribute("spy", new Spy(newUser));
        return (newUser);
    }
}
```

Surveiller la session :

```
private class Spy implements HttpSessionBindingListener {
    final User user;

    public Spy(User user) {
        this.user = user;
    }

    // traitement d'une nouvelle session
    public void valueBound(HttpSessionBindingEvent ev) {
        users.add(user);
    }

    // traitement d'une fin de session
    public void valueUnbound(HttpSessionBindingEvent ev) {
        users.remove(user);
    }

}
```

Intercepteur :

```
// Interception d'une requête
public String beforeAll(HttpServletRequest request) {
    // pour obtenir le temps de calcul
    request.setAttribute("timer", System.currentTimeMillis());
    // vérifier l'authentification
    if (!request.getServletPath().equals(loginUrl)) {
        User user = getUser(request);
        if (user.getAuth() == false) {
            String msg = "Vous devez vous authentifier";
            return views.loginForm(request, user, msg);
        }
    }
    return null;
}

// Interception après vue
public void afterAll(HttpServletRequest request) {
    Long start = (Long) request.getAttribute("timer");
    Long duration = System.currentTimeMillis() - start;
    debug("time : ", duration.toString(), "ms");
}
```

Après l'action et avant la vue :

```
// Interception avant vue
public String beforeView(String view, HttpServletRequest request) {
    return (view);
}
```

Traitement des erreurs :

```
// Interception d'une erreur
public String error(HttpServletRequest request,
        HttpServletResponse response, Throwable e) {
    String msg = "erreur interne : " + e.getMessage();
    return views.message(request, msg);
}
```

Action spécifique de login :

```java
public String doLogin(HttpServletRequest request) {
    User user = getUser(request);
    // si l'utilisateur est connu
    if (user.getAuth())
        return views.home(request, user, users);

    // récupérer les données du formulaire
    user.setId(request.getParameter("id"));
    user.setPassword(request.getParameter("password"));
    // si aucune données retour au formulaire
    if (user.getId() == null || user.getPassword() == null)
        return views.loginForm(request, user, null);

    // traiter les données
    if (user.getPassword().length() == 0)
        return views.loginForm(request, user, "Mot de passe vide");
    try {
        business.authenticate(user);
        if (user.getAuth())
            return views.home(request, user, users);
        return views.loginForm(request, user, "mot de passe incorrecte");
    } catch (BusinessException e) {}
    return views.loginForm(request, user, "Utilisateur inconnu");
}
```

Action spécifique de logout :

```java
public String doLogout(HttpServletRequest request) {
    User user = getUser(request);
    user.setAuth(false);
    user.setId("");
    user.setName("");
    user.setPassword("");
    return views.message(request, "A bientôt");
}
```

Le traitement des requêtes HTTP :

```
@Override
final protected void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    try {
        String view = beforeAll(request);
        if (view == null)
            view = processAction(request, response);
        if (view != null)
            view = beforeView(view, request);
        if (view != null)
            processView(view, request, response);
    } catch (Throwable t) {
        String view = error(request, response, t);
        if (view != null)
            try {
                processView(view, request, response);
            } catch (Throwable e) {
                throw new ServletException(e);
            }
    } finally {
        afterAll(request);
    }
}
```

Le mécanisme d'aiguillage :

```
private String processAction(HttpServletRequest request,
        HttpServletResponse response) throws Throwable {
    request.setCharacterEncoding("UTF-8");
    String action = request.getServletPath();
    if (action.equals(loginUrl))
        return doLogin(request);
    if (action.equals(logoutUrl))
        return doLogout(request);
    // autres actions
    throw new ServletException("action " + action + " inconnue.");
}
```

On peut envisager également utiliser le mécanisme d'introspection pour déterminer la bonne méthode.

```
@Action(name = "/backoffice/liste.do")
String doBackOfficeList(HttpServletRequest request,
        HttpServletResponse response) throws ...
```

Le traitement des vues :

```
private void processView(String view, HttpServletRequest request,
        HttpServletResponse response) throws Throwable {
    if (view.endsWith(".jsp") || view.endsWith(".jspx")) {
        // technologie JSP/JSPX
        request.getRequestDispatcher(view).forward(request, response);
    } else if (view.endsWith(".xsl")) {
        // technologie XSLT / JAXP
        processXsltView(view, request, response);
    } else {
        // autres technologies
        throw new IllegalStateException("no view " + view);
    }
}
```

4 Quelques derniers conseils

validation des données :

- Les données en provenance d'un client doivent toujours être vérifiées (présence des paramètres, typage du contenu).
- Ne pas utiliser les champs cachés HTML pour transmettre des données entre deux requêtes : utiliser la session.

Traitement des formulaires :

- Un formulaire doit toujours être présenté à partir de données et d'éventuels messages d'erreur associés à ces données.
- Une validation javascript et/ou AJAX est envisageable, mais elle ne doit pas remplacer une validation côté serveur.

Liens : Utilisez soit

- des liens relatifs (difficile sur les applications importantes),
- des liens absolus générés automatiquement par les balises `<c:url .../>`
- Le contexte utilisé dans le cadre du développement (`/monappli`) ne doit pas apparaître.
- Si vous devez exporter des URL (par exemple dans un courriel), prenez-soin de les construire à partir de la requête.

Charte graphique :

- La charte graphique doit être implantée par des feuilles CSS.
- La charte graphique doit être implantée en dernier mais vous devez prendre soin de prévoir les classes CSS.
- Les ressources statiques (images...) doivent être référencées par la feuille CSS grâce à des URL relatives à la position de la feuille CSS.
- Si la technologie des CSS ne suffit pas, utilisez les balises personnalisées (`.tag`).