

Applications Web dynamiques



Daniel Hagimont

<https://www.google.fr/search?q=daniel+hagimont+home+page>

1

L'objectif a été de fournir des technologies logicielles permettant de faciliter le développement des applications Web dynamiques.

Orientation vers des solutions plus flexibles

- PHP
 - ◆ Balises spécifiques au sein d'une page HTML
 - ◆ Langage de script exécuté coté serveur
- Servlet
 - ◆ Programme Java coté serveur générant une page HTML
- JSP
 - ◆ Balises spécifiques au sein d'une page HTML
 - ◆ Bonne intégration à Java
- Architectures MVC et multi-tiers
 - ◆ Découpage par préoccupation (Modèle MVC)
 - ◆ Séparation et multiplication des tiers : architecture scalable

2

Pour faciliter le développement de ces applications, on s'est orienté vers des solutions plus flexibles et modulaires.

On a vu qu'un programme dans un serveur Web génère une page HTML sur son STDOUT. Hors, très souvent, le programme doit générer la même page, mais avec seulement quelques petites variations (des variables avec des valeurs différentes qui doivent être affichées dans la page). Il a donc été proposé des pages HTML dynamiques avec des bouts de script à l'intérieur. Et lorsqu'on veut générer la page, on passe des variables en paramètre et on exécute les bouts de script dans la page, ces scripts étant chargés de mettre à jour la page avec ces variables.

C'est cette notion de pages dynamiques que l'on retrouve dans PHP. Les pages dynamiques (pages PHP) sont exécutées coté serveur avec les variables à afficher et la page générée est retournée au client.

Les servlets sont des programmes Java exécutés dans un serveur Web (un peu comme des CGI, mais avec une API de plus haut niveau plus facile à utiliser).

Les JSP sont l'équivalent des pages PHP, mais dans le monde Java.

Enfin, une approche s'est généralisée : on essaie de séparer les principales préoccupations. Par exemple, on sépare la partie présentation (génération des pages Web) de la partie gérant les données de l'application (modèle MVC). Egalement, on sépare les serveurs gérant des données (des bases de données) des serveurs gérant des pages Web, et ces serveurs peuvent être répliqués pour obtenir des architectures passant à l'échelle (résistant à une forte charge).

A ne pas confondre avec

■ Applets

- ◆ Programme Java
- ◆ Référencé depuis une page HTML
- ◆ Stocké sur le serveur Web
- ◆ Chargé et exécuté par le client (browser) web

■ JavaScript

- ◆ Langage de script
- ◆ Balises spécifiques au sein d'une page HTML
- ◆ Chargé et exécuté par le client (browser) web
 - ▶ Encore que Javascript est de plus en plus utilisé coté serveur également

3

Le slide précédent passait en revue des technologies (servlet, JSP, PHP) s'exécutant coté serveur et générant des pages Web retournées au client (navigateur).

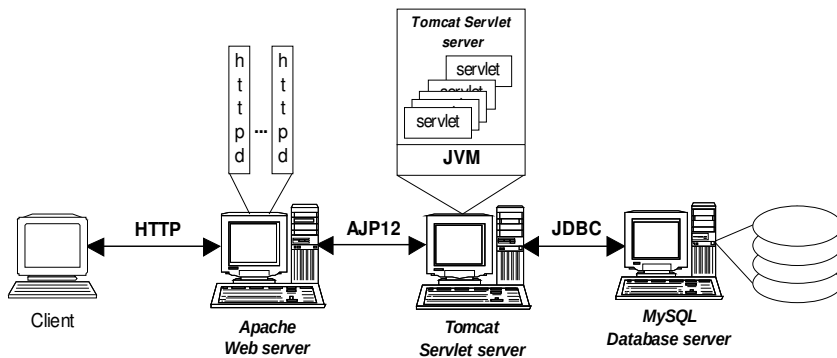
Il ne faut pas confondre avec des technologies qui s'exécutent dans le navigateur (coté client).

Les Applets sont des programmes Java référencés depuis des pages HTML. Ces pages et les Applets sont stockées coté serveur, mais chargées dans le navigateur et les Applets s'exécutent dans le navigateur. Cette technologie a été inventée par Sun Microsystems (inventeur de Java) pour réaliser des traitements qui modifient dynamiquement et coté client l'affichage dans le navigateur. Cette technologie est un peu tombée en désuétude avec l'essor de JavaScript.

JavaScript est un langage de script (mais ce n'est pas du Java, ça y ressemble juste un peu) dont du code peut être inséré au sein d'une page HTML à l'aide de balises spécifiques. Lorsque la page HTML est chargée dans le navigateur, et lorsque des interactions ont lieu (par exemple un click sur un bouton), du code JavaScript peut être exécuté en réaction à l'événement, dans le navigateur, et peut modifier la page HTML.

Servlets Java

- S'exécute dans un "Servlet Container" sur une JVM
 - ♦ Tomcat ou Jetty
 - ♦ Soit utilisé comme un serveur intégré (web + servlets)
 - ♦ Soit utilisé séparément du serveur Web



4

Les servlets sont des programmes Java exécutés dans un serveur Web (un peu comme des CGI, mais avec une API de plus haut niveau plus facile à utiliser).

Une servlet est programmée comme une classe Java qui s'exécute dans un environnement d'exécution (appelé servlet container) dans une JVM.

Les exemples les plus connus de servlet containers sont Tomcat et Jetty.

On peut utiliser ces serveurs (Tomcat ou Jetty) comme des serveurs Web intégrés jouant à la fois le rôle de serveur Web et servlet container.

On peut aussi, suivant la philosophie multi-tiers, séparer le traitement des pages Web statiques (avec un serveur Web comme Apache) et le traitement des pages Web dynamiques (avec un serveur Tomcat). Le serveur Apache reçoit alors des requêtes HTTP. Il traite les requêtes aux pages statiques. Il retransmet les requêtes dynamiques vers Tomcat.

Servlet HTTP - API

```
public void init()  
protected void doGet(HttpServletRequest request, HttpServletResponse response)  
protected void doPost(HttpServletRequest request, HttpServletResponse response)
```

■ Paramètres

- ♦ HttpServletRequest : permet de manipuler la requête reçue
- ♦ HttpServletResponse : permet de générer la réponse

■ Remarques

- ♦ Attention, ces méthodes peuvent être exécutées en concurrence
- ♦ Ces méthodes peuvent appeler des BD : JDBC

5

Une servlet est une classe dans laquelle on peut implanter principalement 3 méthodes.

init() est exécutée à l'initialisation de la servlet.

doGet() est exécuté à la réception d'une requête HTTP GET.

doPost() est exécuté à la réception d'une requête HTTP POST.

Les paramètres permettent de manipuler la requête reçue et de générer la réponse.

Attention, ces méthodes peuvent être exécutées en concurrence, donc gare aux variables partagées.

Ces méthodes peuvent appeler des bases de données dans le cadre d'une application complexe. Pour cela, JDBC (Java Database Connectivity) est une API Java implantée par tous les fournisseurs de BD permettant de faire des requêtes aux bases de données (en local ou à distance).

Exemple 1/3

```
<html> <head><title>Directory</title></head><body>
<h1>Enter a person</h1>
<form action= "Directory" method="post">
  firstname<input type="text" name="firstname"><br/>
  lastname<input type="text" name="lastname"><br/>
  email<input type="text" name="email"><br/>
    <input type="submit" name="op" value="add">
    <input type="submit" name="op" value="list">
</form>
</body></html>
```



Enter a person

firstname

lastname

email

6

Voyons un petit exemple (un annuaire).

Un formulaire permet de :

- soit saisir une personne (nom, prénom, email) puis de cliquer sur le bouton "add".
- soit de cliquer sur le bouton "list" et alors on affiche le contenu de l'annuaire.

On peut remarquer que le formulaire, lorsqu'il est validé, envoie une requête HTTP POST à une servlet dont le nom est "Directory".

Remarquons aussi que si l'utilisateur choisit "add" un paramètre op=add sera envoyé (et similairement op=list dans l'autre cas).

Exemple 2/3

```
@WebServlet("/Directory")
public class Directory extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        try {
            String db_url = "jdbc:hsqldb:hsqldb://localhost/xdm";
            String db_user = "sa";
            Class.forName("org.hsqldb.jdbcDriver");
            Connection con = DriverManager.getConnection(db_url, db_user, null);

            response.setContentType("text/html");
            out.print("<html><body><h1>Directory</h1>");

            String op = request.getParameter("op");
            if (op.equals("list")) {
                Statement stmt = con.createStatement();
                ResultSet res = stmt.executeQuery("SELECT * FROM directory");
                while(res.next()) {
                    out.print("<p>" + res.getString("firstname") + " " + res.getString("lastname")
                        + " " + res.getString("email") + "</p>");
                }
            }
        }
    }
}
```

7

Voici le code de la servlet qui doit soit ajouter une personne, soit lister le contenu de l'annuaire.

Cette servlet utilise JDBC pour émettre des requête à la base de données.

La classe est annotée (@WebServlet) avec le path dans l'URL permettant d'appeler la servlet.

La classe implante la méthode doPost()

La variable "out" permet d'écrire du HTML (avec des println()) retourné au navigateur (un peu comme les écritures sur STDOUT pour un CGI).

Il y a ensuite 4 lignes permettant d'initialiser la connexion avec la BD.

On retourne ensuite l'entête MIME (text/html) indiquant que l'on retourne du HTML. Ensuite, on génère le début du code HTML.

On peut accéder aux paramètres de la requête avec request.getParameter(...).

En fonction du paramètre "op" :

- si c'est "list" : on exécute une requête (SELECT) sur la BD pour récupérer le contenu de l'annuaire et on affiche chaque ligne reçue avec des out.print()

Exemple 3/3

```
        } else {
            PreparedStatement ps = con.prepareStatement("INSERT INTO directory VALUES(?,?,?)");
            ps.setString(1, request.getParameter("firstname"));
            ps.setString(2, request.getParameter("lastname"));
            ps.setString(3, request.getParameter("email"));
            ps.executeUpdate();
            out.print("Person was added");
        }
        out.print("</body></html>");
    } catch (Exception ex) {
        ex.printStackTrace(out);
    }
}
```

8

si c'est "add" : on exécute une requête (INSERT) sur la BD pour ajouter la nouvelle personne. Les paramètres de cette personne sont récupérés avec `request.getParameter(...)`. On génère dans la page HTML un petit message (Person was added).

Dans les 2 cas, on ferme les balises ouvertes (body et html).

On peut voir que les servlets, c'est un peu comme des CGI, sauf que c'est en Java. On doit faire des `out.print(...)` chaque fois que l'on veut générer du HTML.

Session

- Notion de session

- ◆ Une requête dépend du résultat des requêtes précédentes
- ◆ Ex : caddy

- Création de session

```
HttpSession HttpServletRequest.getSession ()  
HttpSession HttpServletRequest.getSession(boolean create)
```

- Utilisation de la session

```
Object getAttribute(String name)  
Enumeration getAttributeNames()  
long getCreationTime()  
int getMaxInactiveInterval()  
void invalidate()  
void removeAttribute(String name)  
void setAttribute(String name, Object value)  
void setMaxInactiveInterval(int interval)
```

9

Les servlets fournissent une notion de session qui permet de relier plusieurs requêtes successives d'un même utilisateur. Les sessions résolvent le même problème que les cookies qu'on a vu auparavant (souvenez vous de l'exemple du caddy).

Lorsqu'on reçoit une requête, on peut appeler `getSession(true)` qui va créer une session si elle n'existe pas, ou retourner la session si elle existe.

Cet objet session est associé à l'utilisateur et sera toujours le même pour cet utilisateur.

Pour gérer un objet caddy, il suffit de l'enregistrer dans l'objet session.

La classe `HttpSession` fournit des méthodes d'enregistrement (pour enregistrer un objet quelconque sous un nom) et consultation (pour retrouver l'objet enregistré sous un nom). Voir les méthodes `getAttribute()` et `setAttribute()`.

Notez les autres méthodes permettant de supprimer une session ou de fixer sa durée de vie si elle n'est pas utilisée.

NB : a priori ces sessions sont implantées à l'aide de cookies (vus précédemment), mais ce n'est pas la seule technique pour implanter les sessions.

Cookies

■ Création / initialisation

```
Cookie(java.lang.String name, java.lang.String value)
void setValue(java.lang.String newValue)
void setMaxAge(int expiry)
void setDomain(java.lang.String pattern)
java.lang.String getValue()
java.lang.String getDomain()
int getMaxAge()
```

■ A l'exécution

```
Cookie[] HttpServletRequest.getCookies()
HttpServletResponse.addCookie(javax.servlet.http.Cookie)
```

10

Les servlets permettent également de gérer des cookies directement.

On peut créer et initialiser un cookie, puis ajouter (addCookie) ce cookie à une réponse.

On peut aussi récupérer la liste des cookies d'une requête (getCookies).

Packaging et déploiement d'une servlet

- Un répertoire par application
 - ◆ Pages web (html)
 - ◆ Répertoire "WEB-INF"
 - ▶ Répertoire "classes" : les classes des servlets
 - ▶ Répertoire "lib" : les librairies (jar) par exemple driver jdbc
- Création d'un fichier WAR (jar)
- Copie dans \$TOMCAT_HOME/webapps
- Le fichier WAR est expansé

11


Lorsqu'on a programmé une servlet, cette-ci doit être packagée dans une archive (JAR) qu'on suffixe par convention par .war

Dans cette archive, il doit y avoir :

- un répertoire pour l'application
- dans ce répertoire, les pages HTML (statiques) qu'on utilise
- dans ce répertoire, un sous répertoire WEB-INF
- dans ce sous-répertoire WEB-INF, un sous répertoire classes contenant les classes compilées des servlets
- dans ce sous-répertoire WEB-INF, un sous répertoire lib contenant d'éventuelles librairies (par exemple le JDBC driver)

Quand on copie l'archive dans le répertoire webapps de Tomcat, Tomcat le détecte, expande l'archive et installe les servlets.

Servlet - Bilan



- Facile à programmer
 - ◆ Programmation en Java
 - ◆ Manque l'insertion de code dans les pages HTML
- Mélange entre aspects présentation (génération HTML), code métier et code d'accès aux données persistantes (à priori pas très MVC ...)
 - ◆ But séparer ces aspects

12

Un premier bilan montre que les servlets sont plus faciles à programmer que des CGI.

Mais comme dit auparavant, il est fastidieux de générer toutes les pages Web (des tonnes de `out.print(...)`) et il manque la possibilité d'écrire des pages HTML incluant des bouts de code permettant d'inclure des données dans la page HTML.

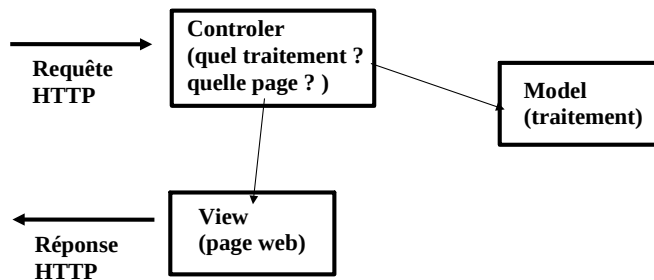
De plus, quand on ne programme qu'avec des servlets, tous les aspects (gestion de l'affichage, gestion des données) sont mélangés dans le code des servlets.

Le modèle MVC vise à séparer les aspects Modèle (code métier de l'application), Vue (code de présentation des pages HTML) et Contrôle (décodage des requêtes pour choisir les traitements à effectuer).

Dans la suite, le but sera de séparer ces aspects.

Modèle MVC

- Model View Controller
- Séparation entre
 - ♦ Le contrôleur : servlet qui aiguille les requêtes
 - ♦ Le Modèle : les classes (beans) qui traitent les données
 - ♦ La vue : pages JSP pour l’affichage à l’écran



13

Pour implanter le modèle MVC, on sépare 3 parties dans le code de l’application Web (au lieu d’avoir tout le code dans une servlet) :

Le Contrôleur est la partie du code qui reçoit la requête HTTP, extrait les paramètres, et en fonction de ces paramètres, décide d’appeler une fonction spécifique de l’application (de la partie Modèle). Le Contrôleur est implémenté par une servlet.

Le Modèle est la partie du code qui implante les fonctions métiers de l’application. Il s’agit en général de gérer les données de l’application. Par exemple, dans un site de réservation de billets d’avion, cette partie gère les réservations, l’état de remplissage des avions etc. Il ne s’agit pas de présentation des pages HTML (partie Vue) ou de traitement des paramètres de la requête (partie Contrôleur).

La Vue est la partie présentation de l’application et est chargée de présenter les résultats en générant des pages HTML.

Le Contrôleur reçoit donc la requête HTTP, appelle le Modèle pour faire le traitement correspondant à la requête (et récupère un résultat), puis appelle la Vue pour générer la page Web présentant les résultats.

JSP (Java Server Page)

- Langage de script (proche de Java)
 - ♦ Générer des pages dynamiques
 - ♦ Intégré dans des pages web
 - ♦ Compilé dynamiquement en servlet
- Interaction avec des classes Java



14

Alors que le Contrôleur est implémenté par une servlet, la Vue est implantée par des JSP (Java Server Pages).

Une JSP est une page HTML dans laquelle on peut introduire des séquences de script (en Java) à l'aide de balises spécifiques (balises JSP).

Une page JSP est compilée en servlet qui, à l'exécution, génère la page HTML résultat à retourner au client. Les balises HTML de la JSP sont laissées telles quelles dans la page résultat. Par contre les balises JSP sont exécutées et ce que ces balises affichent est inséré dans la page résultat.

Un petit exemple

<pre><%@ page language="Java" %> <html> <head> <title>First.jsp</title> </head> <body> <h1>Nombres de 1 à 10</h1> <% for(int i=1; i<=10; i++) { out.println(i + "
"); } %> </body> </html></pre>	<pre><html> <head> <title>First.jsp</title> </head> <body> <h1>Nombres de 1 à 10</h1> 1 2 3 4 10 </body> </html></pre>
---	---

15

Voici un petit exemple de page JSP.

Une balise en début de page indique que c'est une page JSP.

On peut voir que c'est une page HTML avec des balises HTML classiques, ici `<html>`, `<title>` ou `<h1>`.

On peut ajouter dans la page HTML une portion de script, parenthésée avec `< %` et `%>`

Dans cet exemple, le script affiche (en utilisant la variable prédéfinie "out") les entiers de 1 à 10.

Lorsque cette JSP est retournée au client, le script est exécuté et la page résultat est retournée au client (page résultat à droite).

Les directives

- `<%@ directive attribut1="valeur" attribut2="valeur" ... %>`
- 3 directives possibles :
 - ♦ page : informations relatives à la page
 - ▶ `<%@ page import="..." %>`
 - ▶ `<%@ page errorPage="..." %>`
 - ▶ `<%@ page isThreadSafe="..." %>`
 - ▶ ...
 - ♦ include : fichiers à inclure littéralement (file)
 - ▶ `<%@ include file="..." %>`
 - ♦ taglib : permet d'utiliser des librairies de tags personnalisés
 - ▶ `<%@ taglib uri="..." prefix="..." %>`

16

On distingue dans les JSP les directives, les déclarations et les scripts.

Les directives ne sont pas à proprement parler du code, mais plutôt un paramétrage du comportement de l'environnement d'exécution de la JSP.

La première directive que nous avons déjà vue est la première ligne du slide précédent, indiquant qu'il s'agit d'une page JSP (c'est une directive page).

Le premier type de directive (page) permet d'adapter le comportement de la page :

- notamment pour ajouter des imports de packages Java
- notamment pour indiquer une page à afficher en cas d'erreur dans l'exécution de la JSP
- ou encore pour indiquer si la page JSP peut être exécutée de façon concurrente

Le second type de directive (include) permet d'inclure un fichier dans la JSP.

Le troisième type de directive (taglib) permet d'ajouter une taglib, une librairie définissant de nouvelles balises JSP.

Les déclarations

- `<%! declaration %>` variables et méthodes globales à la page
- Exemple

```
<%!  
String Chaine = "bonjour";  
int Numero = 10 ;  
public void jspInit() {  
    // instructions  
}  
%>
```

17

Les déclarations permettent d'ajouter dans une JSP des déclarations de variables ou de méthodes.

Les scripts Java

- Du code Java : `<% code Java %>`
- Des évaluations d'expression : `<%= expression %>`
- Des variables prédéfinies

```
<%@ page language="Java" %>
<html><head><title>First.jsp</title>
</head><body>
<h1>Nombres de 1 à 10</h1>
<% for(int i=1; i<=10; i++) { %>
  <%= i %> <br/>
<% } %>
</body>
</html>
```

■ Variables prédéfinies

- ◆ `HttpServletRequest request`
- ◆ `HttpServletResponse response`
- ◆ `HttpSession session`
- ◆ `ServletContext application`
- ◆ `PrintWriter out`
- ◆ `ServletConfig config`
- ◆ ...

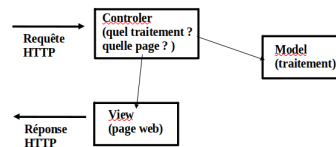
18

Et enfin, les scripts permettent d'insérer un traitement dynamique dans les pages HTML avec les balises `< %` et `%>` (on a déjà vu cet exemple précédemment).

Notons que dans ces scripts, vous disposez de variables prédéfinies qui sont les mêmes variables que l'on trouve dans une servlet. C'est normal, car une JSP est compilée en une servlet. On trouve :

- request et response : en général, une requête HTTP est reçue par une servlet (le Contrôleur) qui appelle du code de la partie Modèle, puis appelle la JSP correspondant à la Vue (et cette JSP est compilée en une servlet). Donc la servlet du Contrôleur appelle la servlet de la JSP et elle lui passe les variables request et response. La JSP a donc accès à ces variables et peut donc voir par exemple les paramètres de la requête HTTP.
- session : de même, si on gère une session dans la servlet du Contrôleur, la session est aussi accessible dans la JSP.
- application : dans une servlet, on a une variable qui correspond à l'application. Elle est unique, même si la servlet est répliquée pour gérer le passage à l'échelle (parallélisme). Et on peut notamment enregistrer des objets dedans (comme on le faisait avec une session)
- out : dans une servlet, on a une variable out qui permet de générer du code HTML pour générer la page Web. On a également cette variable dans une JSP. Cela permet au script de générer du contenu qui est inséré dans la page à l'exécution (à l'endroit où est le script).

Lien HTML/Servlet/JSP



- Une page HTML peut référencer une servlet

- ◆ Dans un formulaire

```
<form action="Directory" method="post">
</form>
```

- Une servlet peut référencer une page JSP

```
RequestDispatcher disp = request.getRequestDispatcher("page.jsp");
disp.forward(request, response);
```

- Passage de paramètre entre le servlet et la JSP

```
request.setAttribute("key", value);
value = (ValueType)request.getAttribute("key");
```

```
// enregistrement dans request, session, ou application
```

19

En présentant le modèle MVC, je disais que le Contrôleur reçoit une requête HTTP, puis décide d'appeler une méthode du Modèle pour récupérer des résultats. Et enfin, le Contrôleur appelle la Vue pour présenter ces résultats.

Comment se passe cet enchaînement.

Tout d'abord, le navigateur du client peut faire appel à la servlet du Contrôleur depuis un formulaire HTML permettant de saisir des paramètres.

La servlet du Contrôleur qui a reçu la requête HTTP peut faire appel au Modèle. Le Modèle est du code Java et nous verrons plus tard comment il est structuré. Je suppose ici que l'appel au Modèle a permis de récupérer des résultats sous forme d'objets Java.

Ensuite la servlet du Contrôleur peut appeler la JSP pour générer la page Web à retourner au client. Ceci se fait en utilisant un objet RequestDispatcher comme indiqué ci-dessus (pour appeler une JSP page.jsp).

Cette JSP a besoin des objets résultats pour les afficher. Pour cela, on peut utiliser l'objet request qui est visible à la fois dans la servlet du Contrôleur et dans la JSP. L'objet request permet d'enregistrer et récupérer des objets (des attributs). Le mode d'emploi est simple :

- la servlet du Contrôleur enregistre un objet résultat :

```
Request.setAttribute("monresultat", theresultat);
```

- la JSP récupère cet objet résultat pour l'afficher

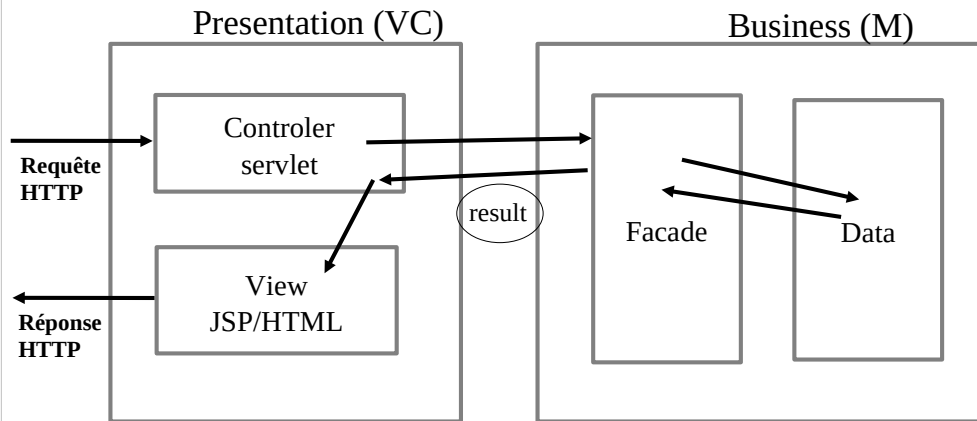
```
< %
```

```
ResultatType r = (ResultatType)request.getAttribute("monresultat") ;
```

```
%>
```

Notons que cet enregistrement d'attributs dans request peut aussi se faire dans session ou application (ça reste plus longtemps).

Architecture souhaitée



20

Voici un schéma d'architecture montrant la structuration visée.

On voit bien que la partie frontale à gauche (qu'on appelle aussi souvent front-end) correspond au servlet container qui héberge les servlets du Contrôleur et de la Vue (les JSP étant compilées en servlet).

La partie à droite (souvent appelée back-end) correspond au Modèle aussi appelé code métier, car cela implante la logique de l'application indépendamment du fait que ce soit une application Web ou pas.

La partie métier est en général composée de deux parties :

- la Facade qui fournit l'interface d'utilisation de la partie métier et son implémentation. Dans mon exemple précédemment évoqué d'un site de réservation de billets d'avion, on aura des méthodes pour réserver, annuler, consulter etc.
- la partie Data correspond aux objets que l'on stocke. On aura dans mon exemple, des trajets d'avions, des villes, des réservations etc. La partie Data est en général implantée par une base de données.

Exemple : les données manipulées (Data)

```
public class Compte {  
    private int num;  
    private String nom;  
    private int solde;  
  
    public Compte() {}  
  
    public Compte(int num, String nom, int solde) {  
        this.num = num; this.nom = nom; this.solde = solde;  
    }  
  
    public String toString() {  
        return "Compte [num="+num+", nom="+nom+", solde="+solde+"]";  
    }  
  
    // setters and getters
```

21

Pour vous donner une vue d'ensemble, je déroule un petit exemple de gestion de comptes bancaires.

Dans cet exemple, la partie Data sera simplement une table d'objets, alors que dans la réalité, ça devrait être une base de données (ce que nous verrons plus loin dans le cours).

Les données manipulées sont des comptes bancaires. On a ici une simple classe Compte.

Exemple : implantation de la Facade

```
public class Facade {  
    private Map<Integer, Compte> comptes = new Hashtable<Integer, Compte>();  
  
    public void addCompte(Compte c) {  
        comptes.put(c.getNum(), c);  
    }  
  
    public Collection<Compte> consulterComptes() {  
        return comptes.values();  
    }  
  
    public Compte consulterCompte(int num) throws RuntimeException {  
        Compte c = comptes.get(num);  
        if (c == null) throw new RuntimeException("Compte introuvable");  
        return c;  
    }  
}
```

22

Vous avez ici l'implantation de la Facade, donc une classe qui fournit l'interface d'utilisation (depuis le Contrôleur) du code métier, mais aussi l'implantation de la gestion des comptes bancaires.

Ici la Facade gère une table (Hashtable) d'objets Compte (car je ne fais pas de bases de données pour le moment).

La Facade implante ensuite les méthodes métiers. On peut ajouter un compte, consulter les comptes, consulter un compte ...

Exemple : implantation de la Facade

```
public void debit(int num, int montant) throws RuntimeException {
    Compte c = consulterCompte(num);
    if (c.getSolde() < montant) throw new RuntimeException("Solde insuffisant");
    c.setSolde(c.getSolde() - montant);
}

public void credit(int num, int montant) {
    Compte c = consulterCompte(num);
    c.setSolde(c.getSolde() + montant);
}

public Facade() {
    addCompte(new Compte(1, "dan", 2000));
    addCompte(new Compte(2, "alain", 4000));
    addCompte(new Compte(3, "luc", 6000));
}
}
```

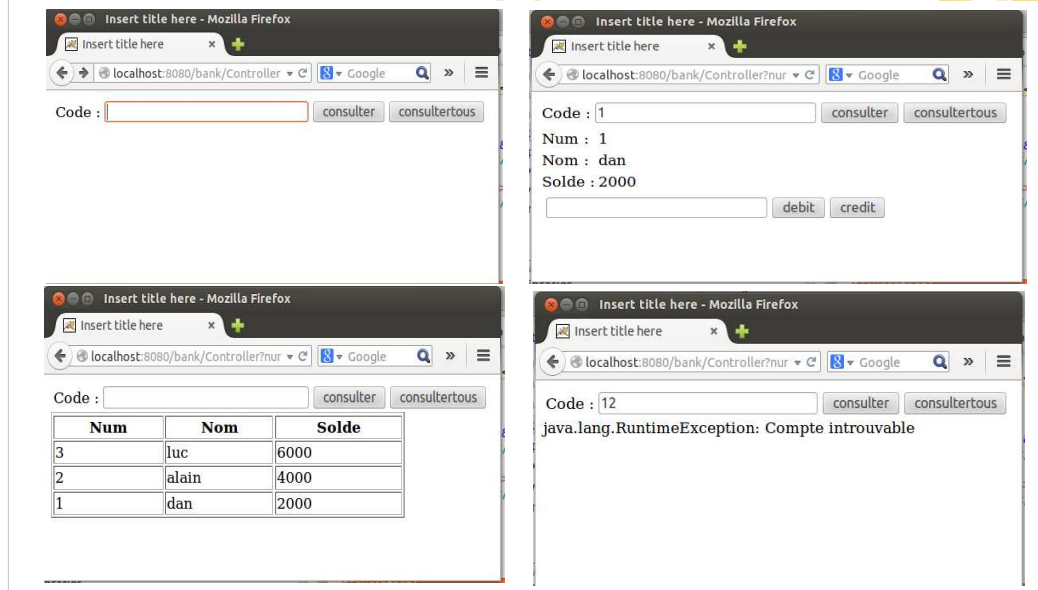
23

... effectuer un débit ou un crédit.

Pour pouvoir tester, je me crée un état avec des comptes dans le constructeur (pas très réaliste, mais juste pour pouvoir tester).

Maintenant, on a fini la partie Modèle (ou métier).

Exemple : schéma d'exécution



Voici ce que l'on a envie d'avoir à l'exécution.

(Gauche, Droite Haut, Bas)

HG : la première page qui permet de choisir une opération :

- consulter si on a tapé le code (numéro) d'un compte
- consultertous qui permet d'afficher la liste de tous les comptes

HD : la page atteinte quand on fait consulter. Elle affiche les données du compte, et permet de faire une opération : débit ou crédit

BG : la page atteinte si on fait consultertous

BD : la page si on a tapé un mauvais numéro de compte

Exemple : le controleur (servlet)

```
@WebServlet("/Controller")
public class Controller extends HttpServlet {

    private Facade facade = new Facade();

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        try {
            String action=request.getParameter("action");
            if (action.equals("consulter")) {
                int num=Integer.parseInt(request.getParameter("num"));
                request.setAttribute("num", num);
                request.setAttribute("compte", facade.consulterCompte(num));
            }
            if (action.equals("consultertous")) {
                request.setAttribute("comptes", facade.consulterComptes());
            }
        }
    }
}
```

25

On a ici l'implantation du Contrôleur.

On voit qu'on instancie la Facade. Ce n'est pas très propre, nous verrons plus tard pourquoi et qu'il y a une autre façon de faire.

Le Contrôleur traite des requêtes HTTP GET.

La première chose qu'il fait est de récupérer le paramètre action qui indique ce qu'il doit faire. Nous verrons plus tard que ce paramètre op a été donné par le formulaire HTML qui fait l'appel à cette servlet.

Conformément à notre application, action peut prendre quatre valeurs : consulter, consultertous, debit et credit.

Si action vaut consulter, on récupère le paramètre num, puis on prépare les paramètres à passer à la JSP :

- on passe en paramètre le numéro du compte saisi, en faisant request.setAttribute(...). Ca permettra à la JSP d'afficher ce numéro même si on a une exception dans l'appel à la Facade.

- on appelle la méthode consulter() sur la Facade, qui retourne un objet Compte. Cet objet Compte est passé en paramètre à la JSP de la même façon.

Si action vaut consulter tous, on appelle la Facade pour récupérer la liste des comptes et on la passe en paramètre à la JSP.

Exemple : le controleur (servlet)

```
if ((action.equals("debit")) || (action.equals("credit"))) {
    int num=Integer.parseInt(request.getParameter("num"));
    int montant=Integer.parseInt(request.getParameter("montant"));
    request.setAttribute("num", num);
    if (action.equals("debit")) facade.debit(num, montant);
    else facade.credit(num, montant);
    request.setAttribute("num", null);
}
} catch (Exception ex) {
    request.setAttribute("exception", ex.getMessage());
}
}
request.getRequestDispatcher("Banque.jsp").forward(request, response);
}
```

26

Si action vaut debit ou credit, on récupère les paramètres num et montant, puis on prépare les paramètres à passer à la JSP :

- on passe en paramètre le numéro du compte en faisant request.setAttribute(...). Ca permettra à la JSP d'afficher ce numéro si on a une exception.
- on appelle la méthode debit() ou credit() sur la Facade. Et si on n'a pas eu d'exception, on annule l'enregistrement du numéro pour ne plus l'afficher.

En cas d'exception, le message associé à l'exception est passé en paramètre à la JSP pour pouvoir l'afficher.

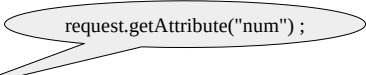
Dans ce petit exemple, je n'ai qu'une seule JSP, mais on aurait pu découper en plusieurs JSP.

Dans tous les cas (valeurs de action), on renvoie l'unique JSP (Banque.jsp).

Exemple : la vue (JSP)

```
<%@ page language="java" import="pack.*, java.util.*" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

<form action="Controller" method="get">
  <table>
    <tr>
      <td> Code :</td>
      <td><input type="text" name="num" value="{num}"></td>
      <td><input type="submit" name="action" value="consulter"></td>
      <td><input type="submit" name="action" value="consulter tous"></td>
    </tr>
  </table>
</form>
```



27

Voici le début de la JSP.

On peut noter l'import du package `bk` (contenant ma classe `Compte`) et du package `java.util` (pour `Hashtable`).

La premier formulaire permet de saisir le numéro de compte et de choisir l'option consulter ou consulter tous. Ce formulaire est toujours affiché dans toutes les pages qui peuvent être présentées (voir les captures d'écran présentées précédemment).

Notons que dans le champs de saisie du numéro de compte, on place le numéro de compte trouvé dans `request.getAttribute(...)`. C'est pour afficher ce numéro dans le cas où on a une exception (et l'exception est affichée plus bas).

La notation `$(nom)` est équivalente à `request.getAttribute("nom")`. C'est plus court et on l'utilise très souvent.

Exemple : la vue (JSP)

```
<% if (request.getAttribute("compte") != null) { %>

<table>
  <tr><td> Num :</td><td>${compte.num}</td></tr>
  <tr><td> Nom :</td><td>${compte.nom}</td></tr>
  <tr><td> Solde :</td><td>${compte.solde}</td></tr>
</table>

<form action="Controller" method="get">
  <table>
    <tr>
      <td><input type="hidden" name="num" value="${num}"></td>
      <td><input type="text" name="montant"></td>
      <td><input type="submit" name="action" value="debit"></td>
      <td><input type="submit" name="action" value="credit"></td>
    </tr>
  </table>
</form>

<% } %>
```

28

Dans la suite de la JSP, on veut afficher un compte si on en a consulté un. Pour savoir cela, on regarde si un objet Compte a été enregistré par le Contrôleur dans request.

Si c'est le cas, on affiche les caractéristiques du compte.

Puis on affiche le formulaire permettant de faire des opérations debit ou credit sur le compte. Ce formulaire permet de saisir le montant, puis de cliquer sur un bouton debit ou credit. Notons qu'une variable de type hidden est définie, permettant de passer le numéro de compte à la servlet Contrôleur lorsque ce formulaire est validé (avec debit ou credit).

Exemple : la vue (JSP)

```
<% Collection<Compte> l = (Collection<Compte>)request.getAttribute("comptes");
    if (l != null) {
    %>

    <table border="1" width="80%">
    <tr> <th>Num</th><th>Nom</th><th>Solde</th></tr>

    <% for (Compte c : l) { %>
    <tr><td><%=c.getNum() %></td><td><%=c.getNom() %></td><td><%=c.getSolde() %></td></tr>
    <% } %>

    </table>

    <% } %>

    ${exception}

    </body>
    </html>
```

29

Enfin, si on trouve une liste de comptes enregistrée dans request, on affiche la liste des comptes sous la forme d'un tableau HTML.

Notons que les JSP permettent de mixer du code HTML et du script JSP (l'accolade de la boucle étant même fermée plus loin).

A la fin, on affiche le message de l'exception (s'il y en a un).

Notons qu'on utilise les tableaux HTML dans toute la JSP pour formater l'affichage.

Servlet/JSP - Bilan

- Modèle MVC donne une séparation claire entre
 - ◆ Modèle (programmes Java)
 - ◆ Vue (page JSP)
 - ◆ Contrôleur (servlet)
- Modèle
 - ◆ Le code métier de l'application avec une Facade et des données
- Vue
 - ◆ Des pages HTML et JSP correspondant aux pages affichées
- Contrôleur
 - ◆ Reçoit des requêtes HTTP et récupère les paramètres
 - ◆ Appelle le code métier
 - ◆ Passe les paramètres pour la JSP
 - ◆ Renvoie une page JSP pour la présentation des résultats

30

Le bilan de cette partie du cours est le suivant.

Le modèle MVC permet une séparation claire entre le code métier de l'application (Modèle), la présentation des pages (Vue) et l'orchestration de l'ensemble (Contrôleur).