Java EE

Beans
Scope
MVC

Rappels

- Les servlets facilitent le traitement des requêtes et réponses HTTP (sans HTML)
- Le JSP aide à séparer la forme (HTML) de la logqiue d‘affaires
- À partir d‘une page JSP, une servlet correspondante est générée automatiquement

- Comment la servlet devrait interagir avec les JSP ?

Rappels

- Importation de classes dans une page JSP :
  - <%@ page import="pack.class" %>
  - <%@ page import="pack.class1,...classN" %>

- Remarques
  - Les classes utilisées par les pages doivent être placées dans le répertoire des classes de l'application Web (*WEB-INF/classes*)
  - Les pages dans '*WEB-INF*' ne sont pas accessibles directement via la barre d'adresse (URL)

JSP + Servlets

- JSP pour faciliter le développement et la maintenance du contenu HTML
- Servlet seuls. OK si l'output est de type binaire. Ex : une image ou il n'y a pas d'output. Ex : redirections.
- Typiquement, la forme/présentation varie selon les interactions utilisateur
- L'utilisation de MVC devient nécessaire

MVC - Situations

- Une même requête peut donner des résultats visuels très différents
- On dispose d'une équipe de développement conséquente avec une partie pour le dev. Web et une autre pour la logique métier
- On a un traitement complexe des données mais une présentation relativement fixe

Contrôleur Web

- Typiquement, un contrôleur principal (FrontController) est le point d'entrée des requêtes clientes
- Ce contrôleur choisit par la suite un contrôleur plus spécifique pour répondre à la requête (LoginController, StatsController, etc.)
- Ce n'est pas vraiment une *Façade* (pas pour simplifier un sous-système) mais on fait aussi usage de la délégation. En réalité, un contrôleur utilise typiquement des Façades.

(Rappel) Pattern architectural MVC en Java

(1) Interaction utilisateur

Ex. : clic de souris, clavier, ouverture d'une page Web

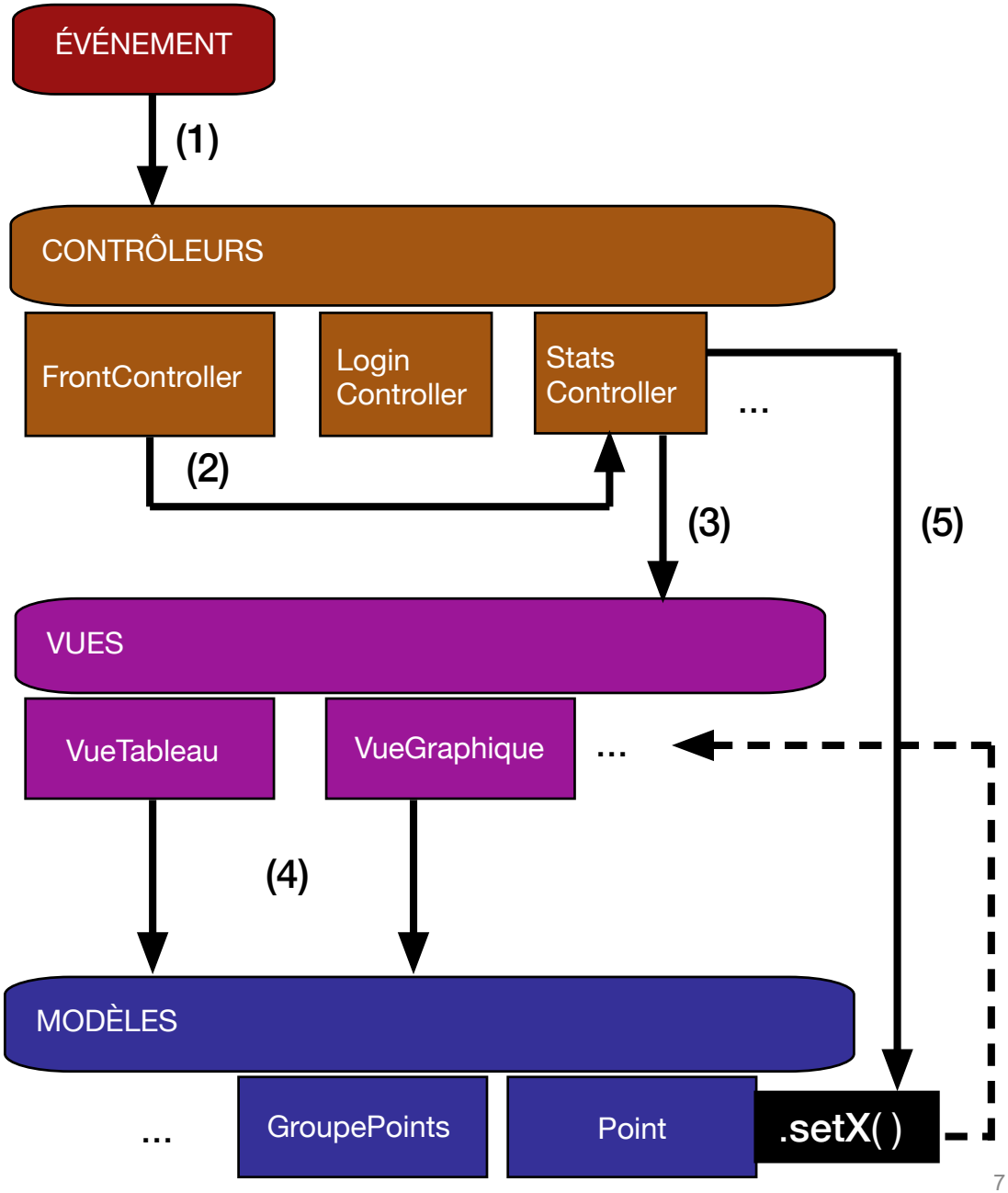
(2) FrontController : Choix du contrôleur pour la requête.

Ex. : Type d'événement, menu ou outil actif, paramètres d'un objet, URL d'une page

(3) Le contrôleur sélectionne les vues.

(4) Chaque vue affiche leur représentation du modèle.

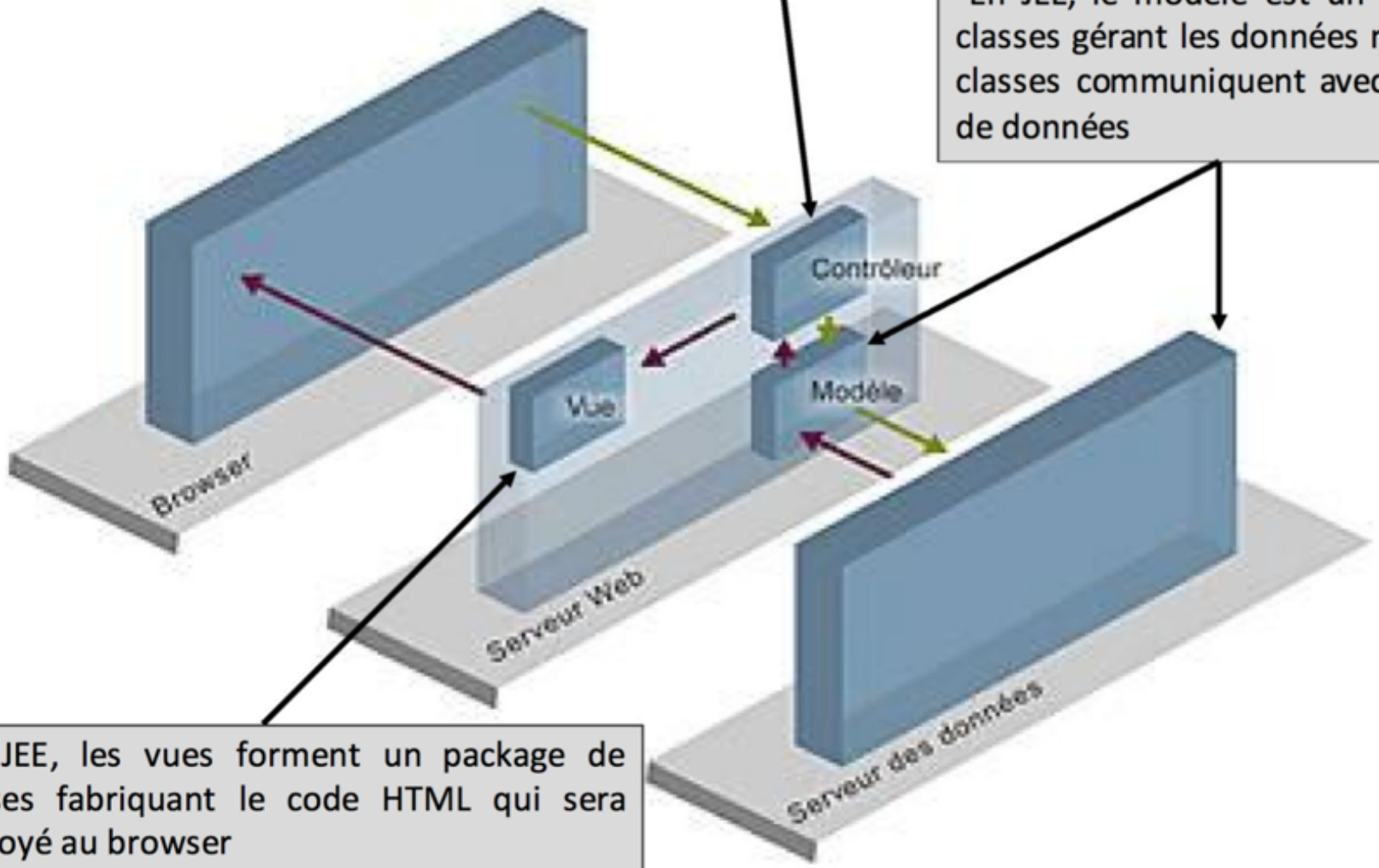
(5) Lorsque le modèle change, les vues sont notifiées via Observer.



Architecture MVC typique en JEE

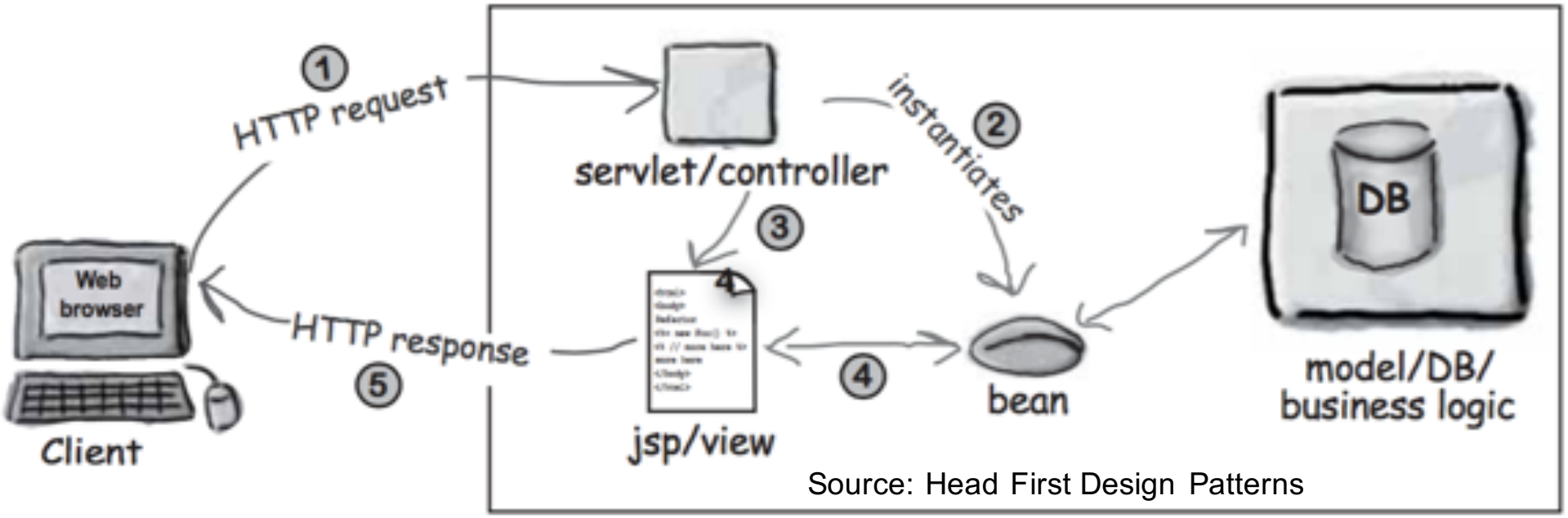
•En JEE, le contrôleur est une servlet JAVA ou une JSP

•En JEE, le modèle est un package de classes gérant les données métiers. Ces classes communiquent avec le serveur de données



•En JEE, les vues forment un package de classes fabriquant le code HTML qui sera renvoyé au browser

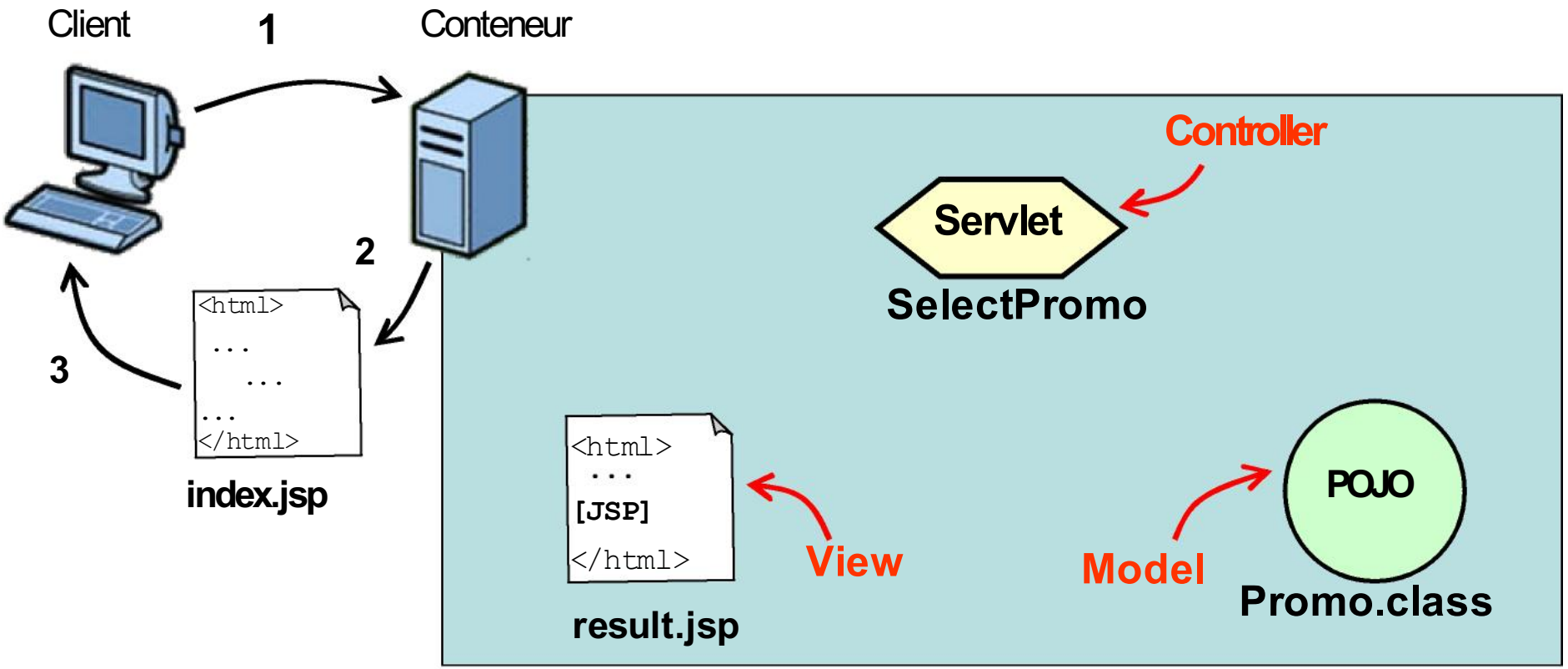
Architecture MVC typique en JEE



Source: Head First Design Patterns

MVC - Étape 1

- Le client récupère un formulaire (index.jsp) pour passer une requête avec paramètres (1, 2, puis 3)

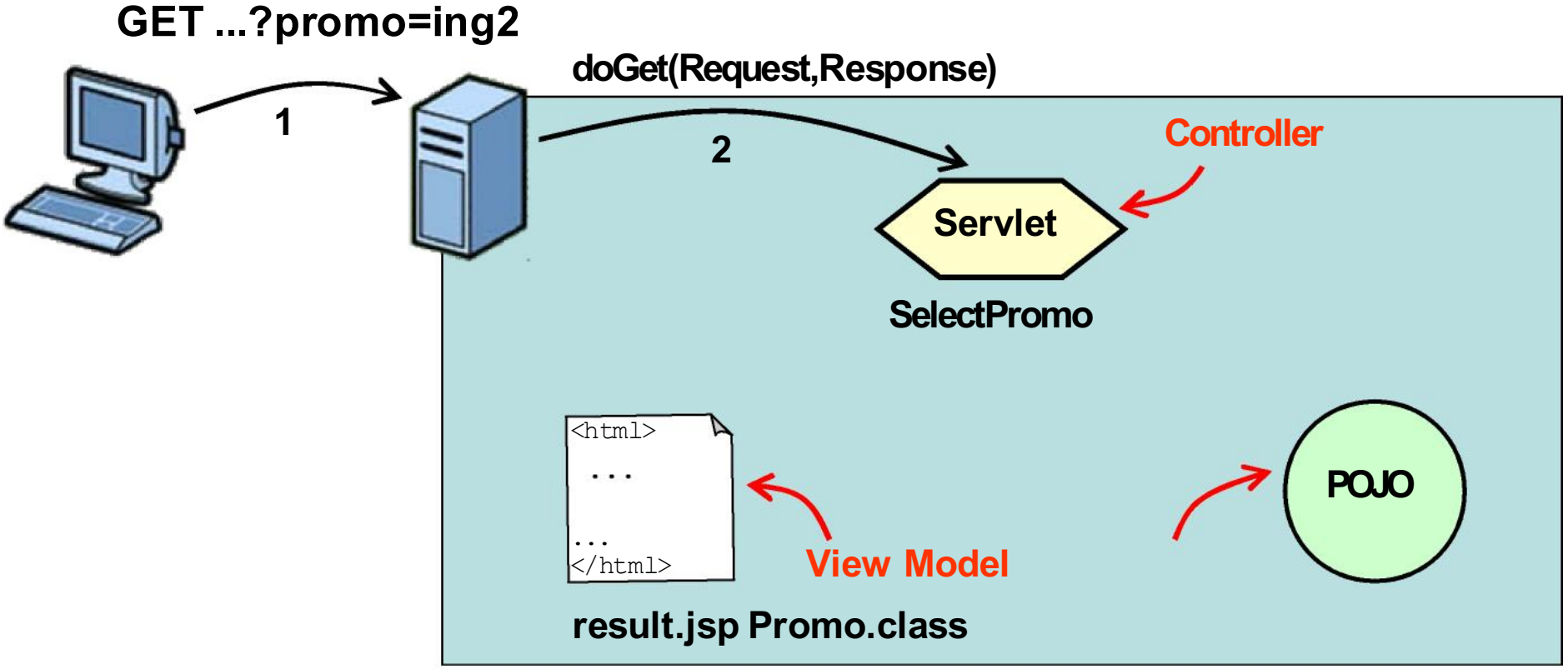


MVC - Formulaire (promo.jsp)

```
<!DOCTYPE html>
<html>
...
<form method="GET" action="/SelectPromo">
Sélectionner la promo à afficher:
 <select name="promo" size="1">
  <option ...>ing1</option>
  <option ...>ing2</option>
  ...
 </select>
 <input type="SUBMIT" />
</form>
...
</html>
```

MVC – Étape 2

- Le client envoie son formulaire (GET/POST avec paramètres)
- Le conteneur transmet au servlet correspondant (le contrôleur)



MVC - Contrôleur : SelectPromo.java

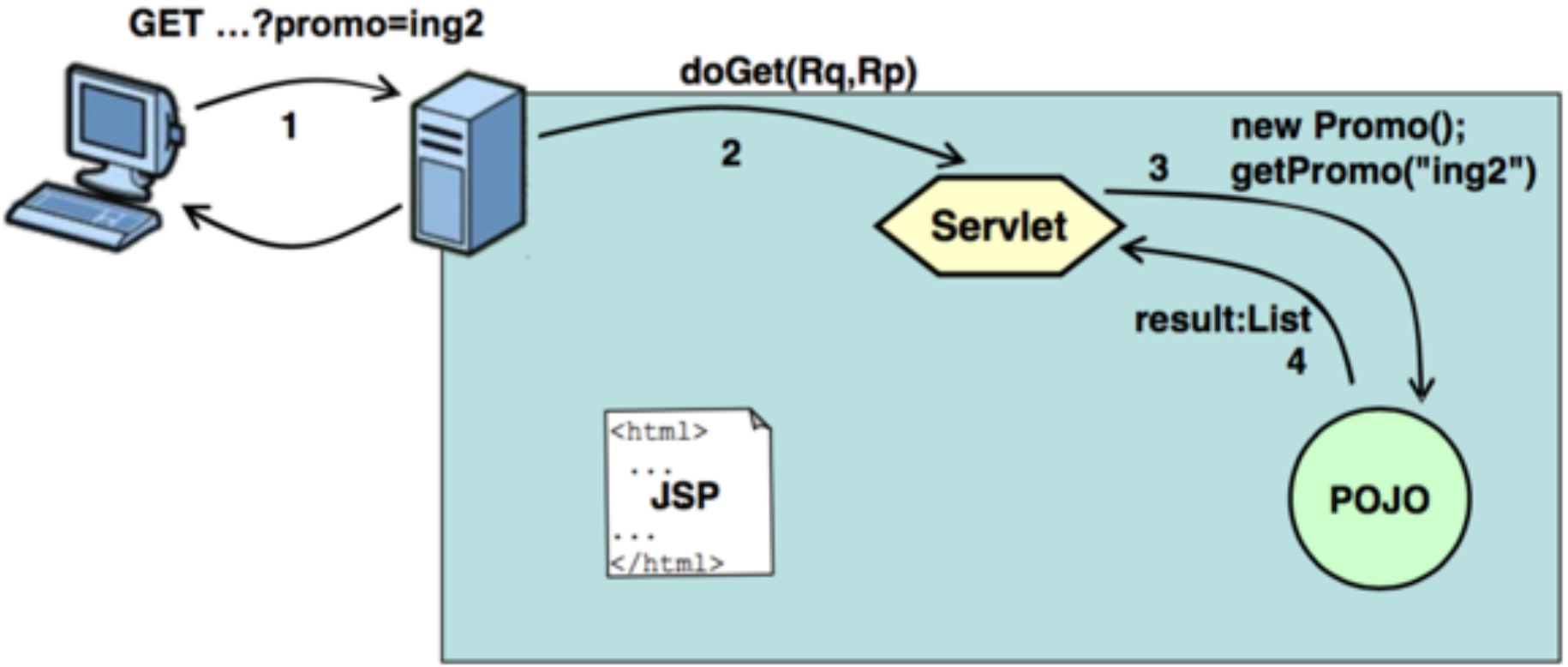
```
package ...;
import javax.servlet.http.HttpServlet;
import javax.servlet.Servlet;

public class SelectPromo extends HttpServlet
                         implements Servlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {

      String promoName =

      request.getParameter("promo");

      //...

}

}
```

MVC – web.xml

```
<web-app …>
 <display-name>MVC</display-name>
  <welcome-file-list>
    <welcome-file>promo.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <description />
    <display-name>SelectPromo</display-name>
    <servlet-name>SelectPromo</servlet-name>
    <servlet-class>pack.SelectPromo</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>SelectPromo</servlet-name>
    <url-pattern>/SelectPromo</url-pattern>
  </servlet-mapping>
</web-app>
```

MVC – Étape 3

- Le servlet *controller* interroge le model sur «ing2»
- Le model retourne au *controller* le résultat correspondant

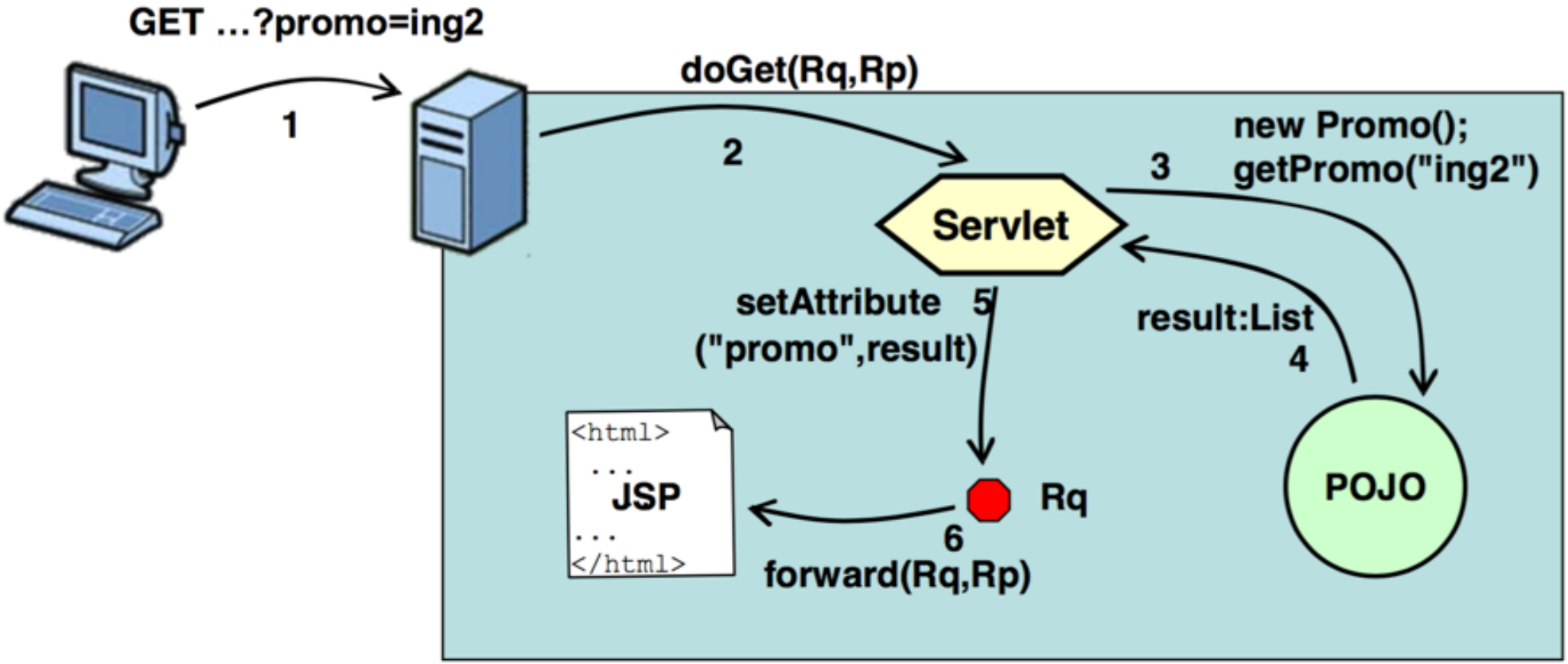


Modèle : Promo.java

```
public class Promo{

  public List<String> getPromo(String promo){
    List<String> promoList

    = new ArrayList<String>();

    if(promo.equals("ing1")){
      promoList.add("Donald Duck");
      promoList.add("Minnie Mouse");
      promoList.add("Pluto"); //...

      } else if

      (promo.equals("ing2")){
      promoList.add("Mickey Mouse");
      promoList.add("Daisy Duck");
      promoList.add("Goofy"); //...
    } else{ return null;}

  return promoList;
  }
}
```

MVC – Étape 4

- Le *controller* utilise les données du *model* pour sa réponse
- Le *controller* transmet sa réponse à la *view* (JSP)

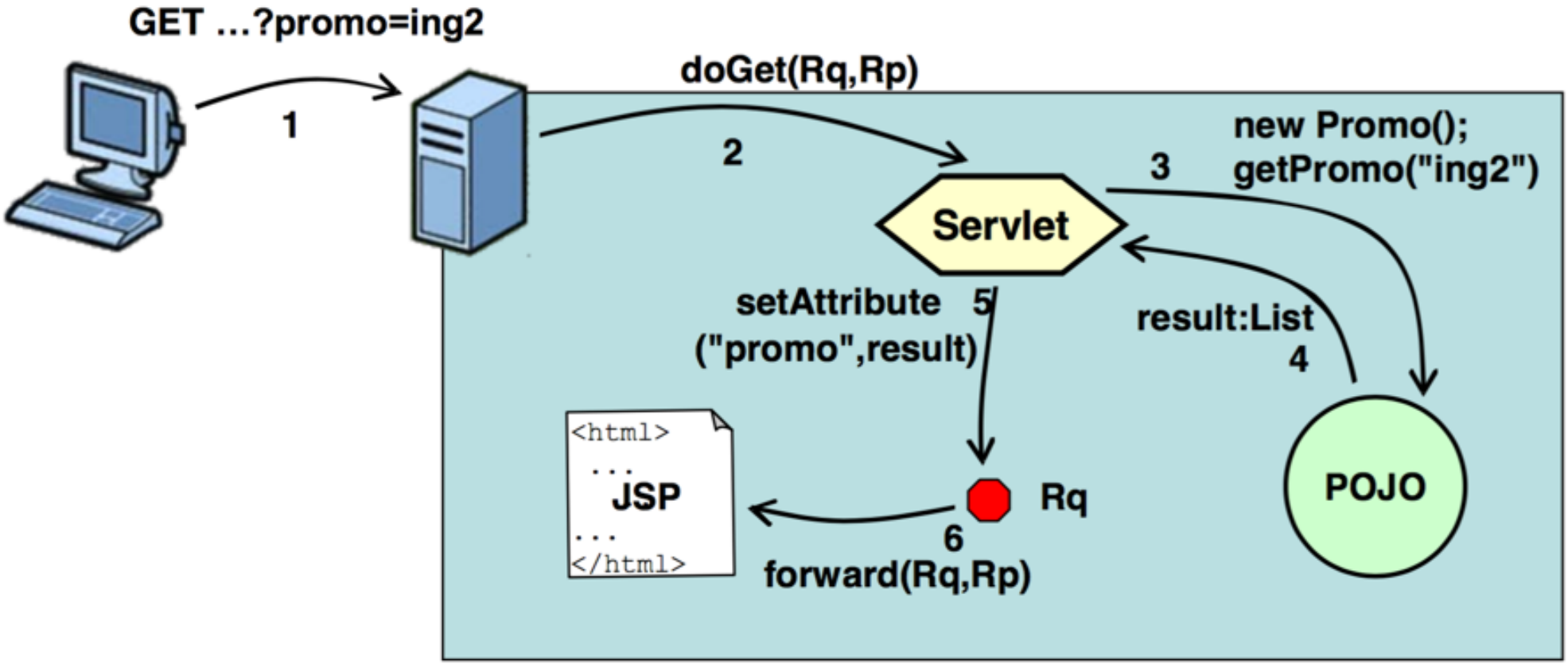


MVC - Contrôleur : SelectPromo.java

```
public class SelectPromo extends HttpServlet ... {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response) {

    String promoName =
    request.getParameter("promo");
    Promo promo = new Promo();
    List<String> result =
    promo.getPromo(promoName);

    request.setAttribute("promo", result);

    RequestDispatcher view =
    request.getRequestDispatcher("result.jsp");

    // Forward la requête à la vue JSP
    view.forward(request, response);

}
}
```

MVC – Étape 5

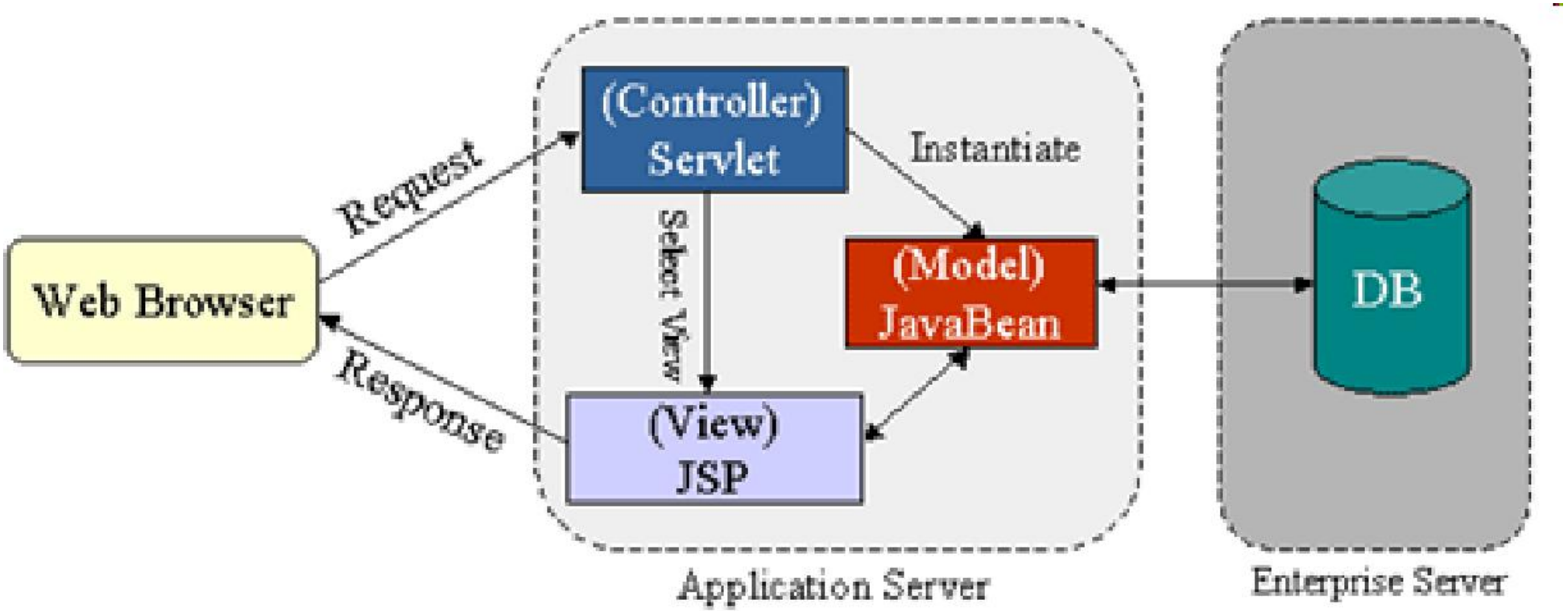
- La JSP (view) traite la réponse transmise par le *controller*
- La page HTML résultante est reçue par le client



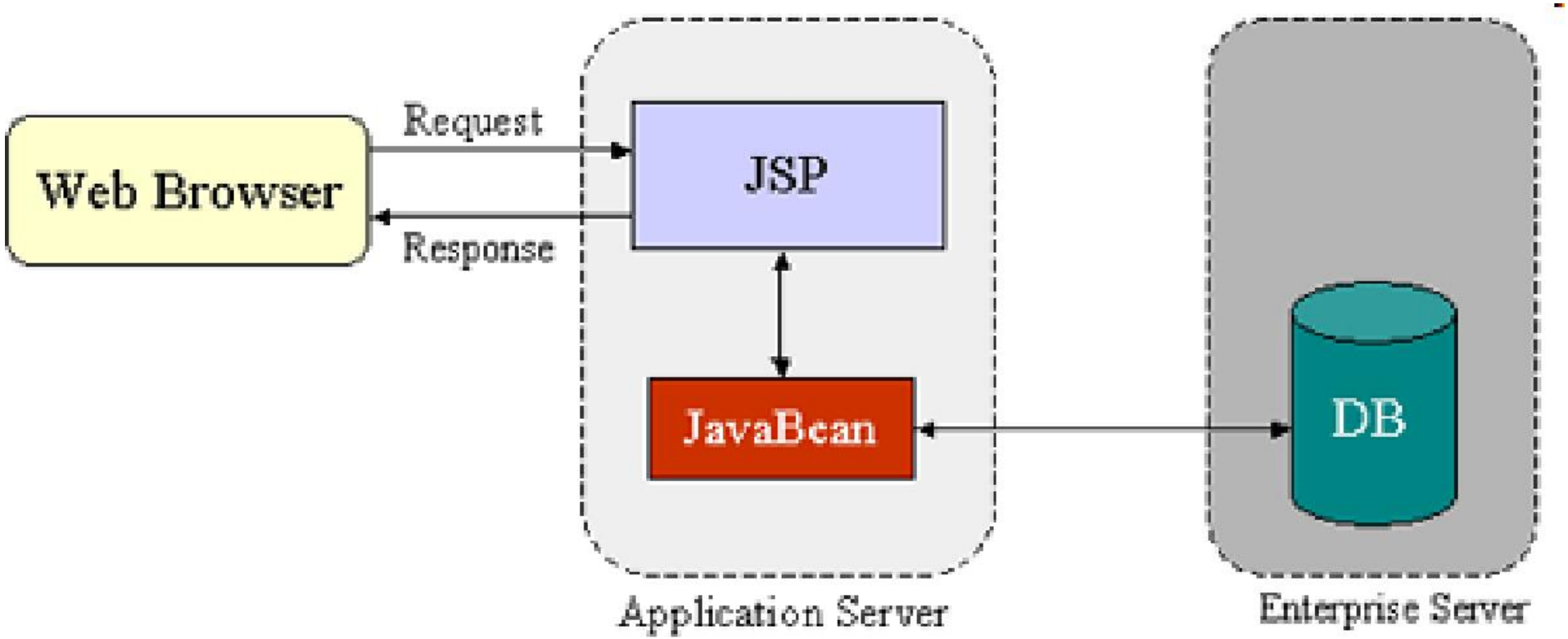
MVC - View : result.jsp

```
<%@ page import="java.util.*" %>

<%@ page language="java"
contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>

<!DOCTYPE html>
<html>
  ...

<%

List<String> promoList = (List<String>)

request.getAttribute("promo");

Iterator it = promoList.iterator();

while (it.hasNext()) {

   out.print("<br />" + it.next());

}

%>
</html>
```

Architecture Web JEE : Model 2 (MVC)



Architecture Web JEE (Sans MVC)



Expression Language (EL)

- Depuis JSP 2.0, ces expressions sont évaluées dynamiquement dans une page JSP
- Permettent de manipuler les données au sein d'une page JSP
- Une EL permet également d'accéder plus simplement aux données et java beans.

Expression Language (EL)

- La syntaxe d'une EL est de la forme suivante : ${ expression }
- La chaîne expression correspond à l'expression à interpréter.
- Peut être composée de plusieurs termes séparés par des opérateurs.

```
${ (10 + 2)*2 }      =>   24
${ a && b }          =>   a ET b
${bean.property}
```

JavaBeans?

Les **JavaBeans** sont des classes Java(POJO) qui suivent certaines conventions:

– Doivent avoir un **constructeur vide** (zéro paramètre)

> On peut satisfaire cette contrainte soit en définissant explicitement un tel constructeur, soit en ne spécifiant aucun constructeur

– Ne doivent **pas avoir d'attributs publics**

> Une bonne pratique réutilisable par ailleurs...

– La valeur des attributs doit être manipulée à travers **des accesseurs** (getters et setters)

> Si une classe possède une méthode getTitle qui retourne un String, on dit que **le bean possède une propriété** String nommée title

> Les propriétés Boolean utilisent isXXX à la place de getXXX

Réf. sur les beans : http://docs.oracle.com/javase/tutorial/javabeans/

Pourquoi faut-il utiliser des accesseurs?

- Dans un bean, on ne peut pas avoir de champs publics
- Donc, il faut remplacer

```
public double speed;
```

Par

```
private double speed;

public double getSpeed() {
    return(speed);
}

public void setSpeed(double newSpeed) {
    speed=newSpeed;
}
```

Pourquoi faut-il faire cela pour **tout** code Java?

Pourquoi faut-il utiliser des accesseurs?

1) On peut imposer des contraintes sur les données

```
public void setSpeed(double newSpeed){
if(newSpeed < 0){
        SendErrorMessage(…);
newSpeed  = Math.abs(newSpeed);  }
speed = newSpeed;
```

Pourquoi faut-il utiliser des accesseurs?

2) On peut changer de représentation interne sans changer d'interface

```
// newSpeed and internal representation are in km units
public void setSpeed(double newSpeed){
       speed = newSpeed;
}

// Now using miles units for internal representation (but
// keeping km units for newSpeed)
public void setSpeed(double newSpeed){
       speed = convertKMToMiles(newSpeed);
}
```

Pourquoi faut-il utiliser des accesseurs?

3) On peut rajouter du code annexe

```
public double setSpeed(double newSpeed){
        speed = newSpeed;
        updateSpeedometerDisplay();
}
```

Exemple MVC : Modèle

```
package model;

        public class PersonneBean {
        private String nom;
        private String prenom;

        public String getNom() {
                return nom;
        }
        public void setNom(String nom) {
                this.nom = nom;
        }
        public String getPrenom() {
                return prenom;
        }
       public void setPrenom(String prenom) {
                this.prenom = prenom;
        }
}
```

- Les Beans doivent être déployés dans le même répertoire que les autres classes Java:

  `WEB-INF/classes/nomPackage`

- Les Beans doivent **toujours** être dans des packages

Exemple MVC : Contrôleur

- En MVC, **les beans sont crées/modifiés par le contrôleur**, en fonction de la requête du client => surtout pas par la vue !

```
import model.PersonneBean;

@WebServlet("/MyServlet")
public class MyServlet extends HttpServlet {
        protected void doGet(...)
        {
                String urlVue = "vue.jsp"; // URL de la vue à appeler
                String nom = request.getParameter("..."); // Récupération du champs nom
                                                          //(par exemple via un formulaire)
                String prenom = request.getParameter("..."); // Récupération du champs prenom
                if(nom == null || nom.trim().equals("")) {
                        // Erreur : nom non saisi
                }
                else if(prenom == null || prenom.trim().equals("")) {
                        // Erreur : prenom non saisi
                }
                else // Cas sans erreur : on traite la requête, et crée les beans
                nécessaires
                {
                        PersonneBean bean = new PersonneBean(); // Instanciation d'un bean
                                                                // de type PersonneBean
                        bean.setNom(nom); // Affectation de la propriété nom
                        bean.setPrenom(prenom); // Affectation de la propriété prenom
                        request.setAttribute("myBean", bean); // Attacher ce bean au
                                                              // scope de requête
                }
                // Forward à la vue:
                request.getRequestDispatcher(urlVue).forward(request, response);
        }
}
```

Exemple MVC : Vue

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Ma Vue</title>
</head>
<body>
        Salut ${myBean.prenom} ${myBean.nom} !
</body>
</html>
```

- En MVC, les beans sont uniquement consultés par la vue, pour faire son affichage.
=> L'Expression Language (${...}) permet de récupérer l'information sur un bean (préalablementp lacé dans un scope par une servlet avec un *setAttribute*):

${nomBean.property}

- *nomBean* est celui défini lors du setAttribute fait par la servlet
- *property* est le nom de la propriété que l'on veut accéder (attention aux normes d'écriture Java => respectez la casse) 13

Un Scope ?

Un scope peut être vu comme un **conteneur de beans** stocké du côté du **serveur** (le client ne peut avoir aucune connaissance sur ces beans)

Il existe 4 types de Scopes, que l'on distingue par la **durée de vie des beans** qui y sont stockés :

- **Requête :** ce scope est créé à chaque requête du client, les beans qu'il contient sont détruit lorsque le serveur à envoyé la réponse au client
- **Session :** ce scope est créé automatiquement par le conteneur JEE pour chaque client qui se connecte au serveur. Les beans qu'il contient ne sont visible que par le client détenant la session, et sont détruit lorsque la session du client se termine (géré par un timeout du côté du serveur)
- **Application :** ce scope est créé automatiquement au lancement du projet JEE sur le serveur. Les beans qu'il contient sont partagés par tous les clients et ne sont détruit que lors de l'arrêt/rechargement du projet JEE sur le serveur.
- **Page :** ce scope restreint la durée de vie des beans aux requêtes POST effectuées sur une page donnée, dès qu'on en sort, les beans sont détruit.

  => Ce scope n'est quasiment pas utilisé

Partage de données sur requête

Servlet

```
ValueObject value = new ValueObject(...);
request.setAttribute("key", value);
RequestDispatcher dispatcher =
   request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="request" />
<jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0 (utilisation des EL)

```
${key.someProperty}
```

Partage de données sur requête: exemple simple

Servlet

```
Customer myCustomer=
  new Customer(request.getParameter("customerID"));
request.setAttribute("customer", myCustomer);
RequestDispatcher dispatcher=
 request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP 1.2

```
<jsp:useBean id="customer" type="somePackage.Customer"
             scope="request" />
<jsp:getProperty name="customer" property="firstName"/>
```

JSP 2.0

```
${customer.firstName}
```

Partage de données sur session

Servlet

```
ValueObject value = new ValueObject(...);
HttpSession session=request.getSession();
session.setAttribute("key", value);
RequestDispatcher dispatcher=
  request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
dispatcher.forward(request, response);
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="session" />
<jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0

```
${key.someProperty}
```

Partage de données sur session: variation

Redirection vers une page au lieu d'un transfert

– Utiliser response.sendRedirect à la place de RequestDispatcher.forward

Différences avec sendRedirect:

– L'utilisateur voit l'URL de la JSP (l'utilisateur ne voit que l'URL du servlet avec RequestDispatcher.forward)

– Deux aller-retour pour le client (au lieu d'un avec forward)

Avantage du sendRedirect

– L'utilisateur peut accéder à la JSP séparément

Possibilité de mettre la JSP en marque-page

Désavantages du sendRedirect

– Deux aller-retour, c'est plus coûteux

– Puisque l'utilisateur peut accéder à la JSP sans passer par le servlet d'abord, les beans qui contiennent les données peuvent ne pas être disponibles (si stockés dans la requête, au deuxième appel ils n'existent plus)

Il faut du code en plus pour détecter cette situation

Partage de données sur application (Rare)

Servlet

```
synchronized(this){
  ValueObjectvalue= new ValueObject(...);
  getServletContext().setAttribute("key", value);
  RequestDispatcher dispatcher=
    request.getRequestDispatcher("/WEB-INF/SomePage.jsp");
  dispatcher.forward(request, response);
}
```

JSP 1.2

```
<jsp:useBean id="key" type="somePackage.ValueObject"
             scope="application" />
  <jsp:getProperty name="key" property="someProperty" />
```

JSP 2.0

```
${key.someProperty}
```