Cours 3 : Java EE & applications web

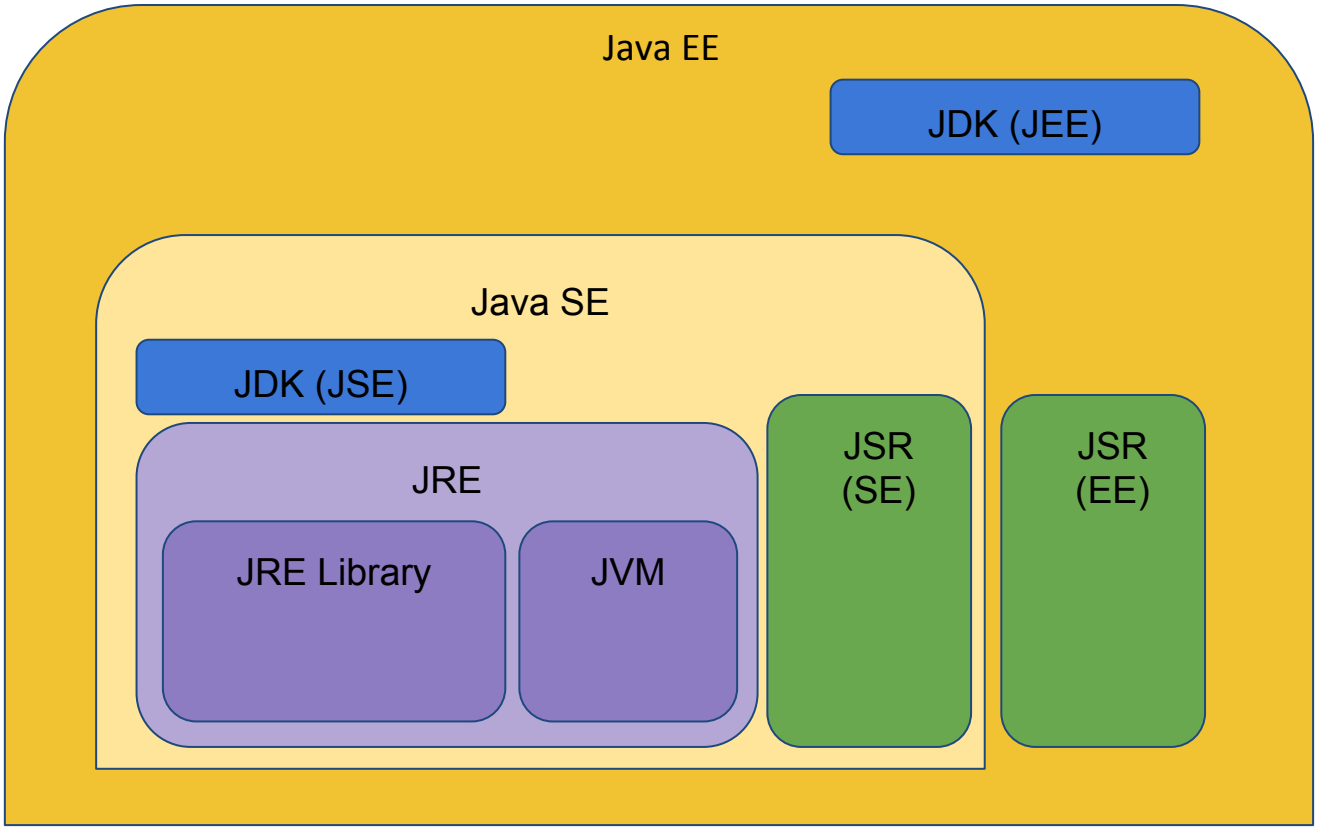
Excilys
Développeurs de passion

Excilys
Développeurs de passion

SOMMAIRE

Présentation de Java EE

Présentation de Java EE



Présentation de Java EE

- Plateforme Java SE
  - JVM (Machine virtuelle Java)
  - Compilateur
  - Ensemble de bibliothèque standard
    - Ces bibliothèques existent en plusieurs implémentations
      - ➢ pour différents systèmes d'exploitations
      - ➢ pour différents appareils (Carte réseau, carte graphique, etc).

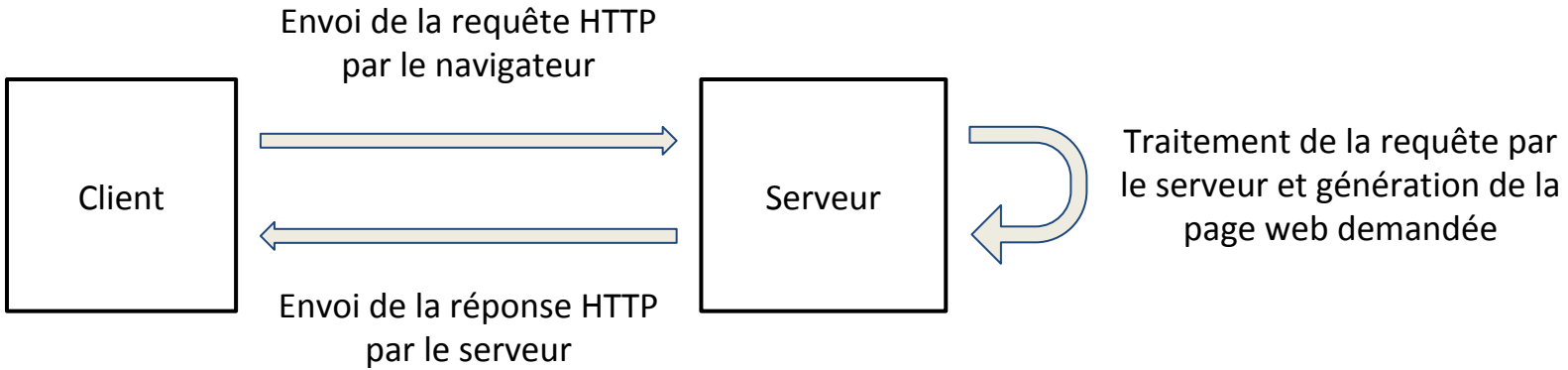
Présentation de Java EE

- JSE : *Java Platform, Standard Edition*
  - Différentes API (JDBC, JAXP etc.).
- JRE : *Java Runtime Environment*
  - La JVM et les bibliothèques nécessaires.
- JDK : *Java Development Kit*
  - L'ensemble des bibliotèque de bases pour le développement.
- JSR : *Java Specifications Requests*
  - L'ensemble des spécifications régissant l'évolution de Java. (Format html)

- Pour résumer Java EE c'est quoi ?
  - une plate-forme (*Java EE Platform*), incluant Java SE + bibliothèques logicielles
  - une suite de tests (*Java EE Compatibility Test Suite*) pour vérifier la compatibilité
  - un serveur d'application de référence conforme au spécifications Java EE dans sa totalité : GlassFish
  - un catalogue de bonnes pratiques (*Java EE BluePrints*)

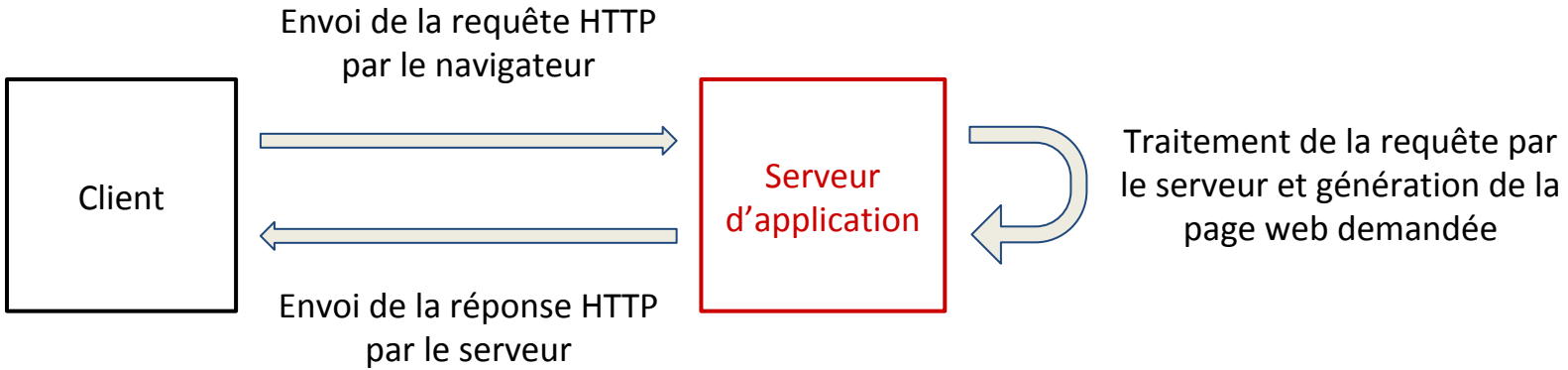
Les applications web avec Java EE

- Traitement d'une requête HTTP (HyperText Transfer Protocol)



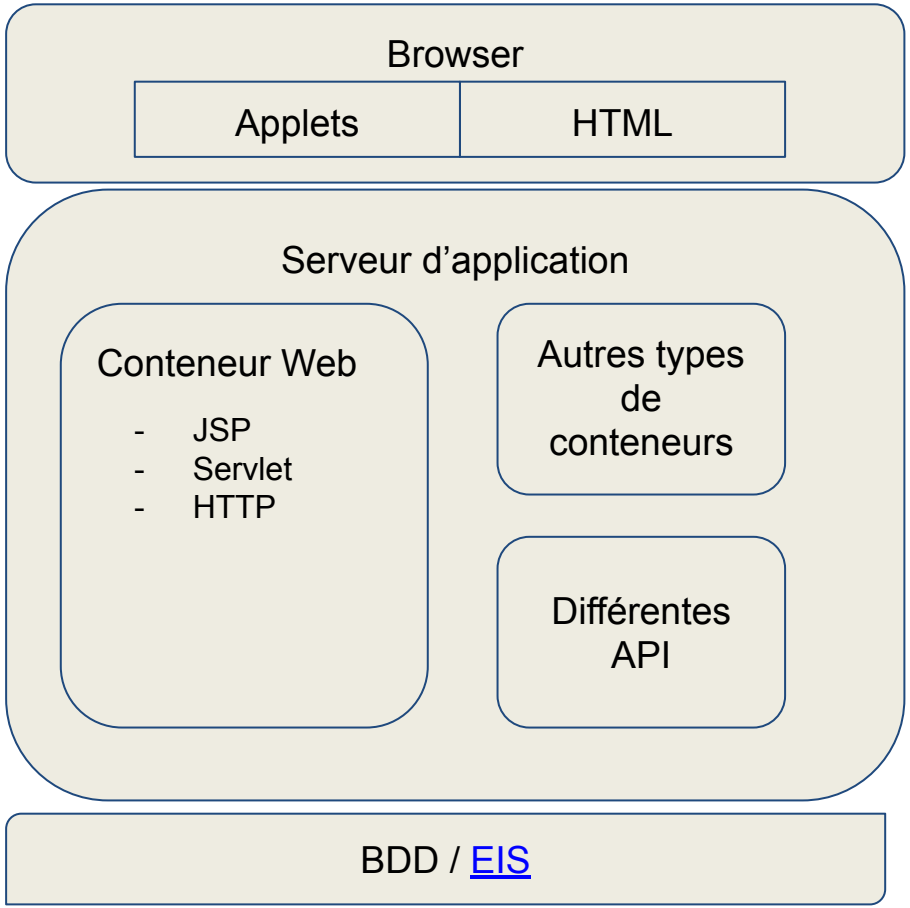
- Ce qu'on va développer à l'aide de Java EE se situe au niveau du serveur

- Traitement d'une requête HTTP (HyperText Transfer Protocol)

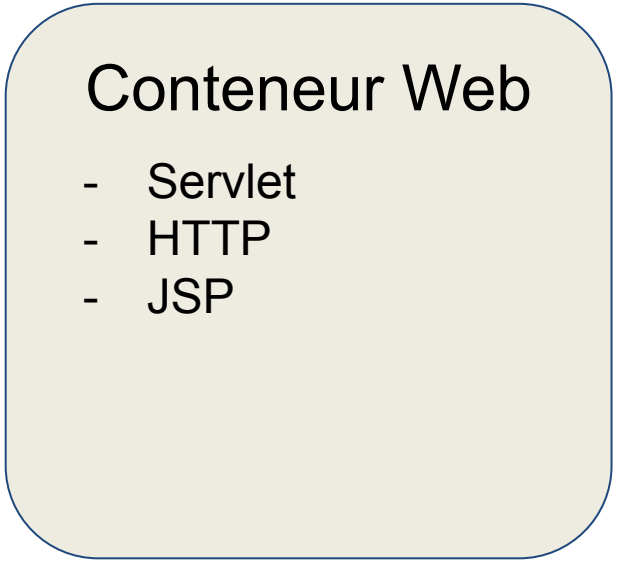


Serveur d'application

- GlassFish
  - Développé par Oracle & la communauté
- WildFly
  - Anciennement JBoss
  - développé par Red Hat



- Tomcat : Conteneur Web
  - Catalina est le conteneur de Servlets
  - Coyote est le connecteur HTTP
  - Jasper est le compilateur JSP

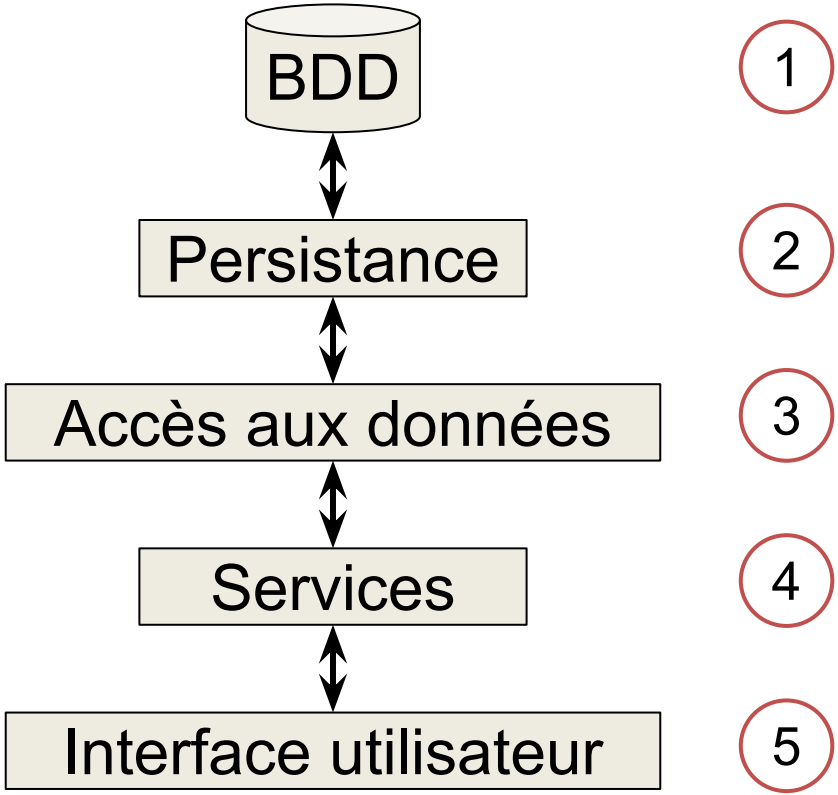


Architecture d’une application web

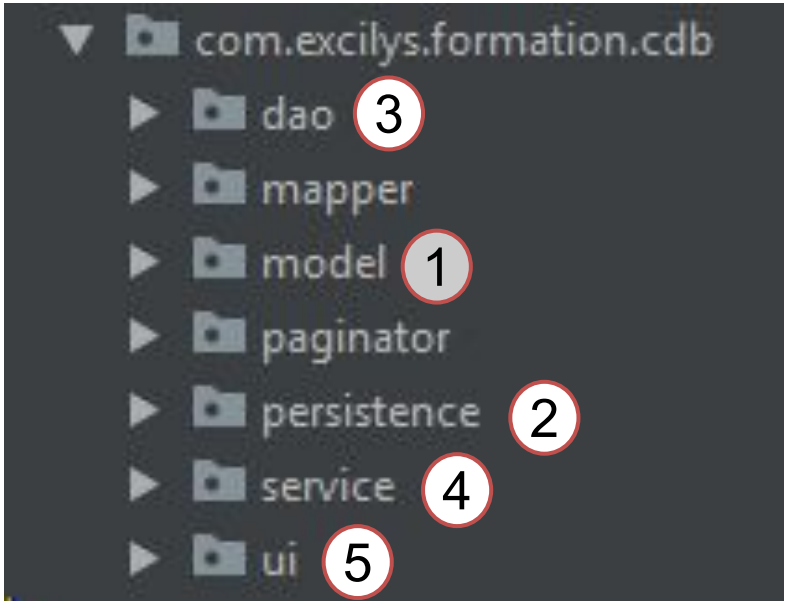
- On peut structurer une application web selon un **trois couches** distinctes (**architecture 3-tiers**) :
  - La couche de **données** ⇒ concerne le stockage et les mécanismes d'accès aux données afin de les utiliser au niveau des traitements ;
  - La couche de **traitement** ⇒ concerne les tâches à réaliser sur les données et les traitements suite à une action de l'utilisateur (authentification par exemple) ;
  - La couche de **présentation** ⇒ concerne l'affichage des données et les interactions avec l'utilisateur.

- On peut raffiner le découpage précédent :

1. Le **Système de gestion de base de données** ⇒ stockage des données utilisées par l'application ;
2. La couche de **persistance** ⇒ gère le mécanisme de sauvegarde et de restauration des données ;
3. La couche d'**accès aux données** ⇒ gère la manipulation des données, quelque soit le SGBD utilisé ;
4. La couche de **services**, ou couche **métier** ⇒ gère la logique de l'application et les traitements à appliquer aux données ;
5. La couche d'**interface utilisateur** ⇒ gère l'affichage des données du service et les actions de l'utilisateur.

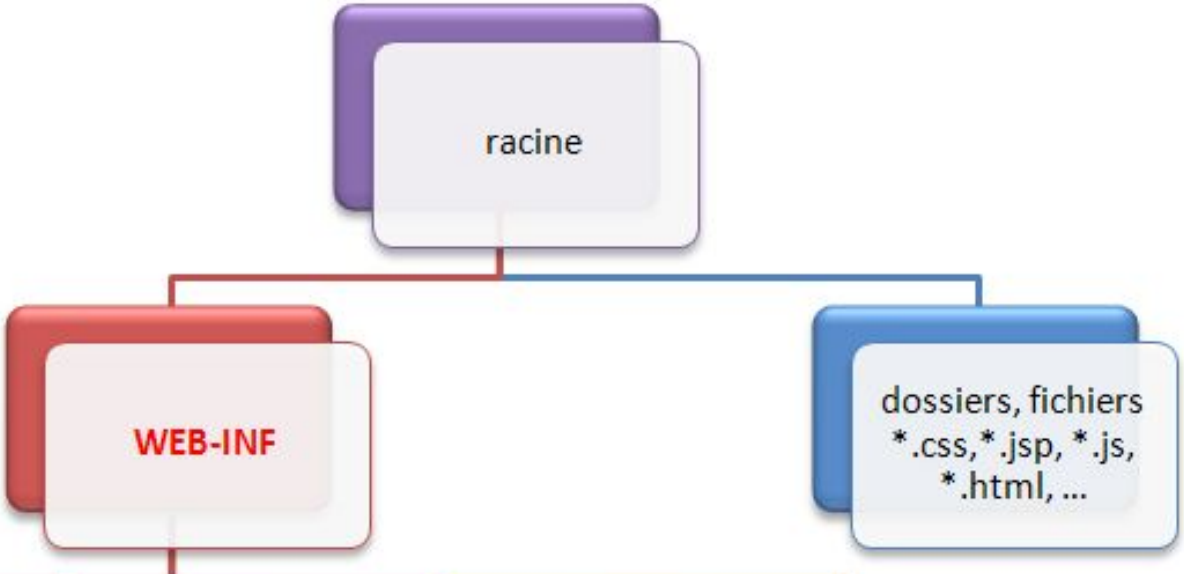

BDD
Persistance
Accès aux données
Services
Interface utilisateur
1
2
3
4
5

- Pour respecter cette architecture, on utilise généralement des **packages** correspondant à chaque couche de l’application.

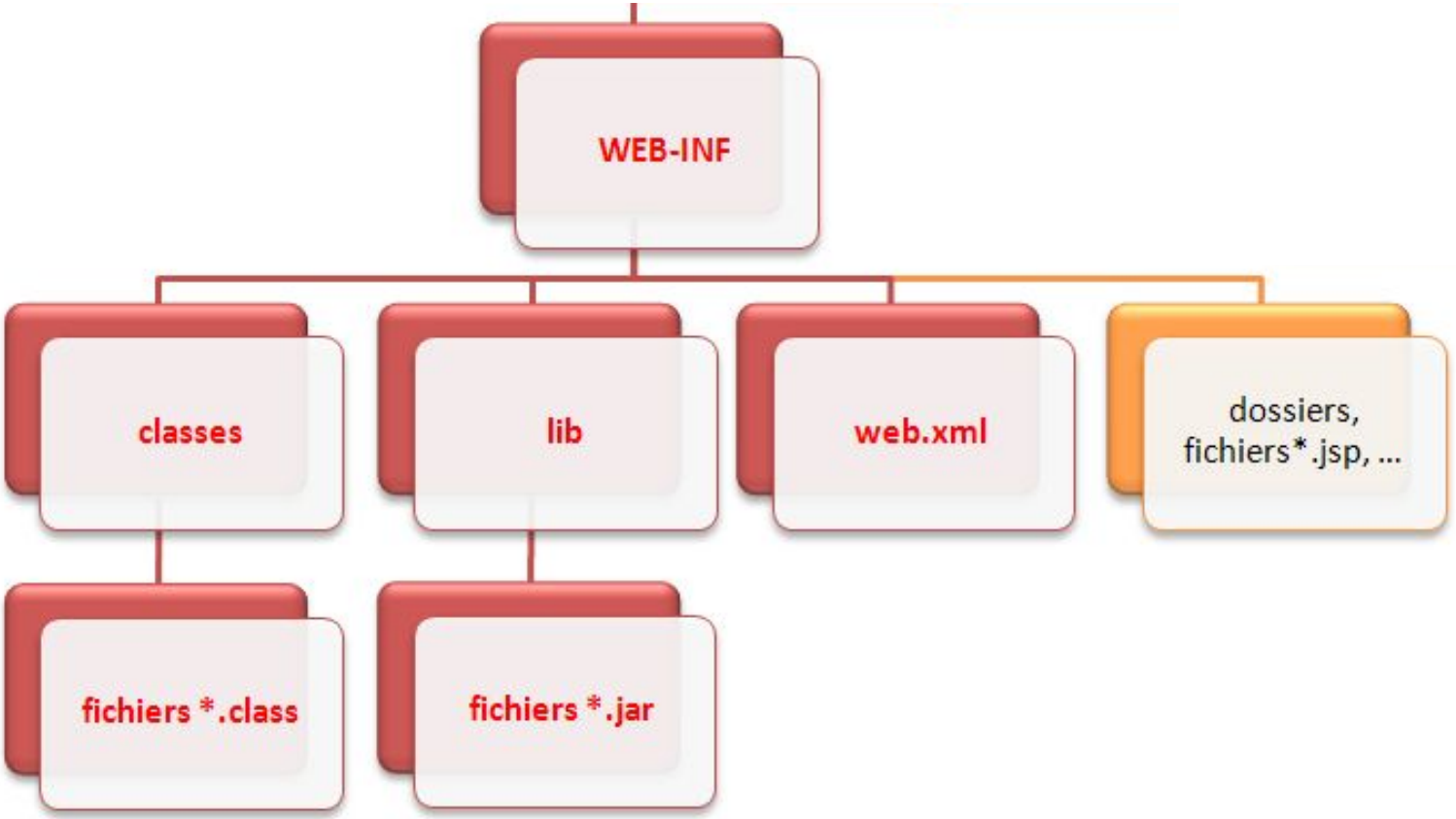


Structure d’une application Java EE

- Les spécifications de Java EE définissent la structure de dossiers que doit respecter toute application Java EE.

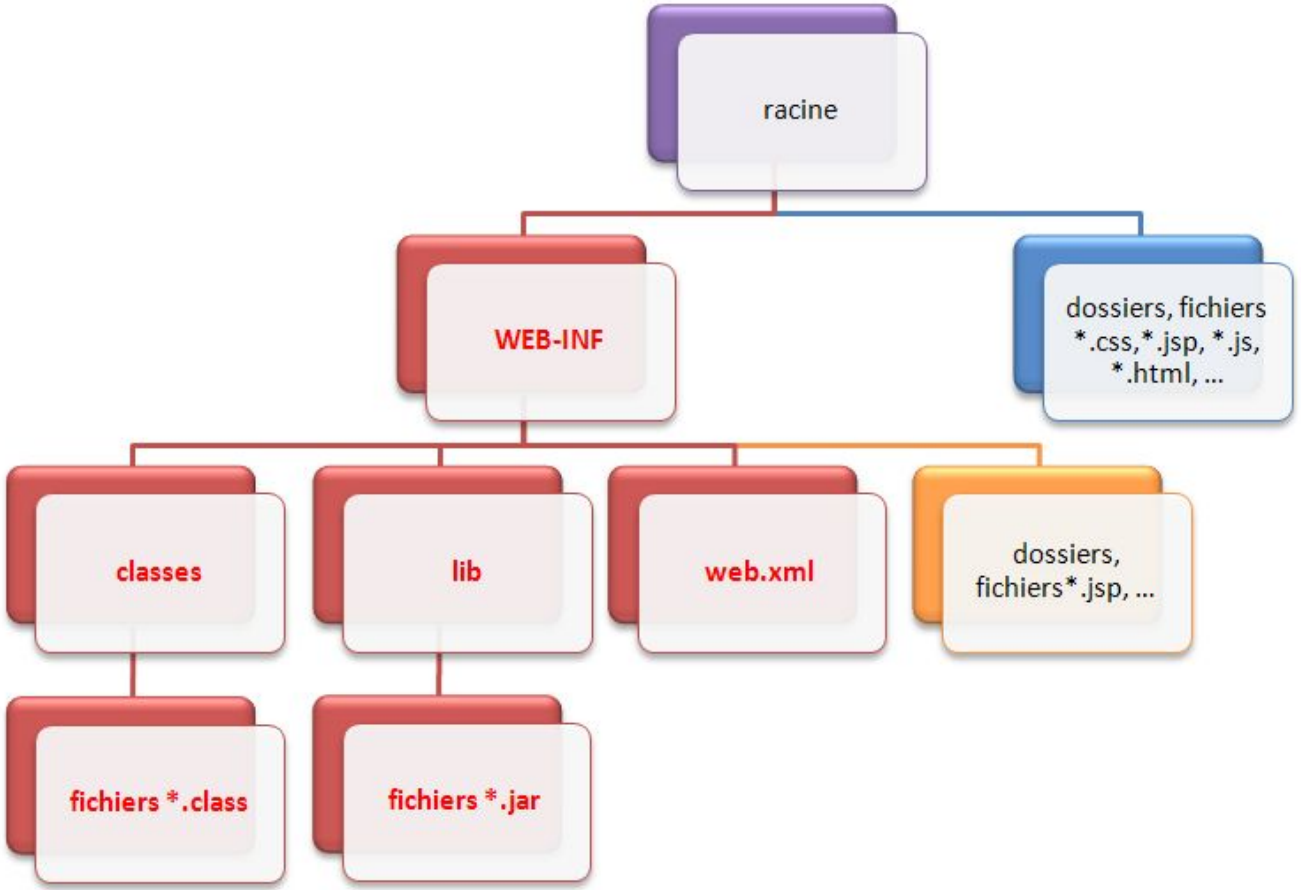


La structure standard...



Structure d’une application Java EE

La structure standard...



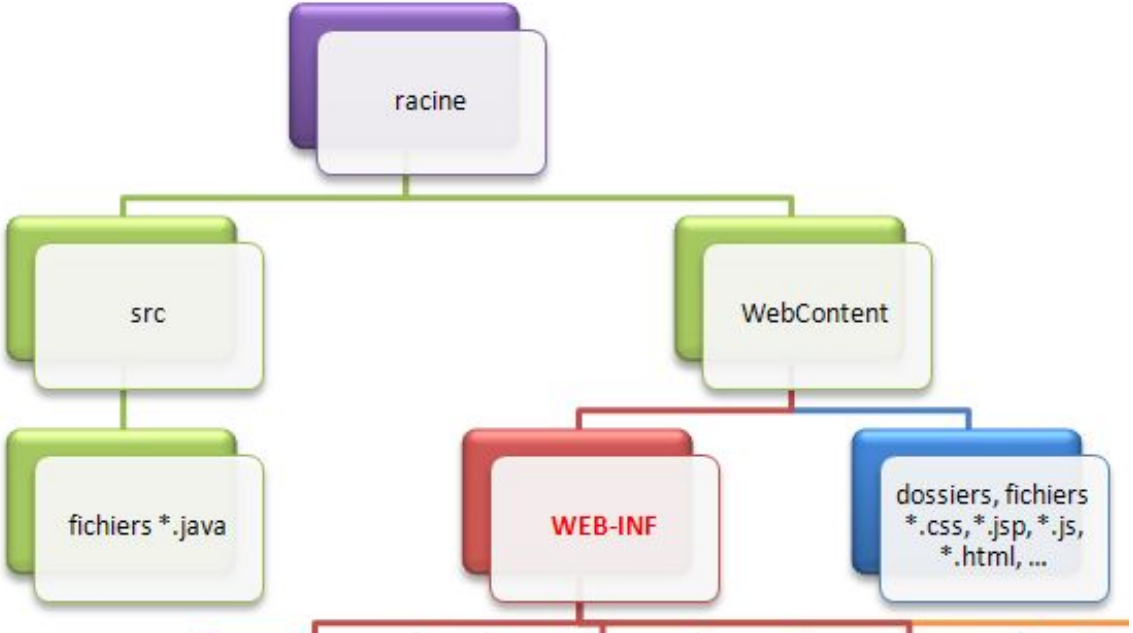
https://zestedesavoir.com/media/galleries/248/19608d2d-5217-4994-838c-c7aee2af1ce2.png

- En violet : racine de l'application, dossier portant le nom du projet et contenant l'intégralité des dossiers et des fichiers de l'application.
- En rouge : le dossier **WEB-INF**, obligatoire, contenant le fichier de configuration de l'application (**web.xml**), les **classes compilées**, les bibliothèques nécessaires pour l'application (fichiers **.jar**).

- En orange : **dossiers et fichiers "personnels"** (facultatifs) placés dans le dossier WEB-INF. Ils ne seront **pas accessibles directement** par le client.
- En bleu : **dossiers et fichiers "personnels"** (facultatifs) placés à la racine de l'application. Ils seront **accessibles directement** par le client.

- Si votre application ne respecte pas cette structure de dossiers, le serveur d’application ne sera pas en mesure de la déployer. Elle ne pourra pas fonctionner.

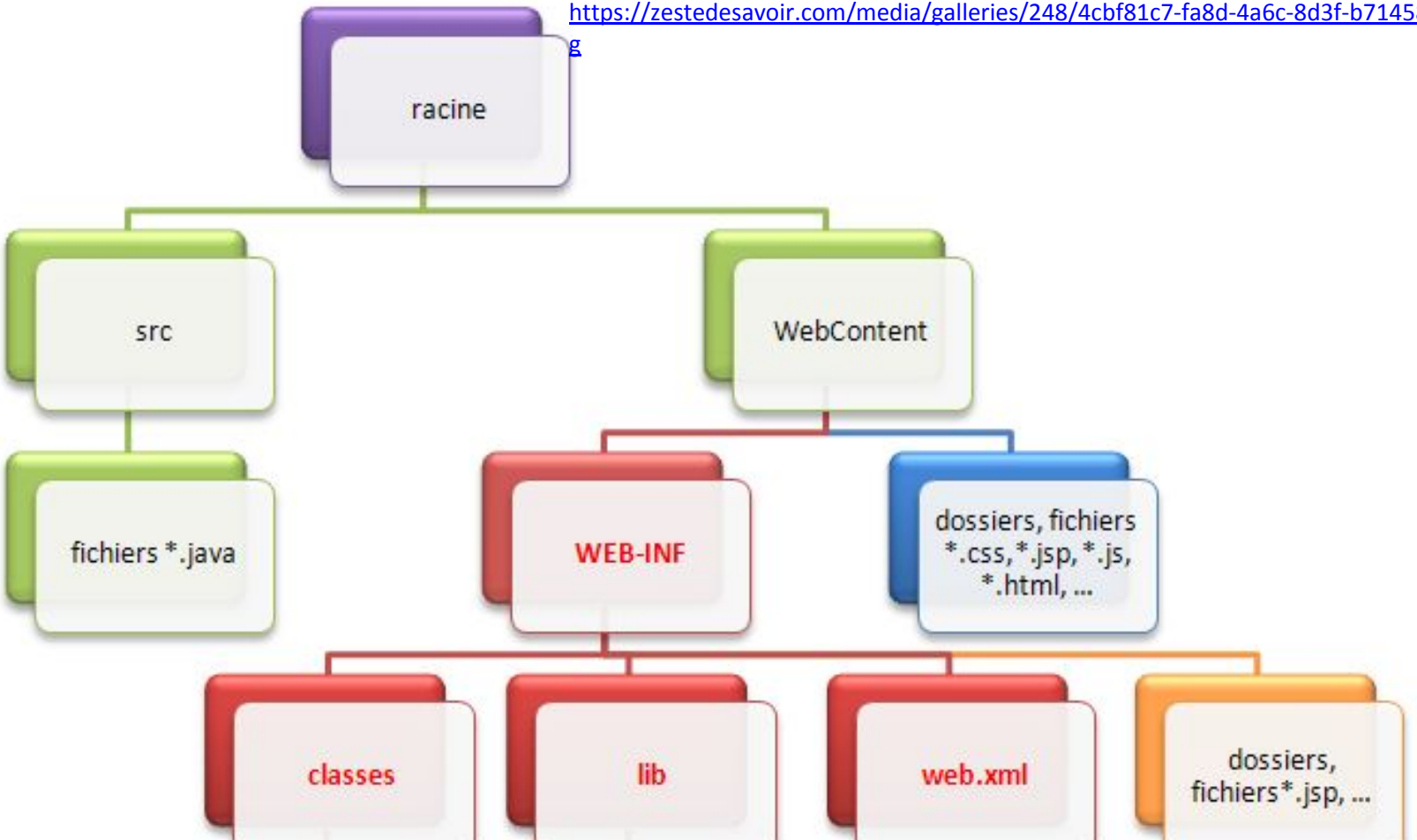
- Eclipse ne fait jamais comme tout le monde, et adapte la structure de dossiers de référence pour une application Java EE de la manière suivante :



Structure d'une application Java EE

La structure selon Eclipse...

https://zestedesavoir.com/media/galleries/248/4cbf81c7-fa8d-4a6c-8d3f-b714586db07d.png

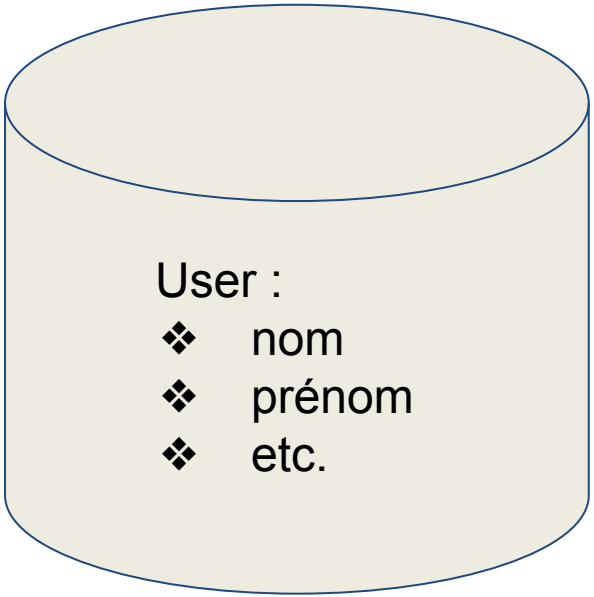


- *Si notre application Java EE ne respecte pas la structure de dossiers de référence, comment faire pour l'utiliser ?*
- Eclipse se débrouille tout seul !
  - Si on lance l'application directement à partir d'Eclipse, tout est géré automatiquement.
  - Si on veut lancer l'application ailleurs, il faudra utiliser l'outil d'export fourni par Eclipse.

Construction d’une application web avec Java EE

1. Modèle de données, JavaBeans
2. JDBC
3. Singleton
4. DAO
5. Services
6. Servlets
7. JSP
8. Transmission de données

Nous avons une **base de données** contenant une table « User »



On crée une classe Java qui aura les mêmes attributs que la table et qui sera chargée de **représenter les instances** de la table dans notre programme Java. C'est le **modèle**.

```java
public class User {
    private String nom;
    private String prenom;
}
```

- Un composant **JavaBean** est une simple classe Java qui respecte les conventions suivantes :
  - La classe doit être **publique** (optionnellement Serializable) ;
  - La classe doit posséder un **constructeur sans paramètre** ;
  - La classe doit respecter le principe d'**encapsulation** :
    - les variables d'instance doivent être privées et être accessibles via des accesseurs et mutateurs ;
    - les accesseurs et mutateurs doivent respecter les conventions de nommage (get/is/set suivi du nom de la variable, avec CamlCase).
  - La classe ne doit pas être déclarée `final`.

Dans le cas de notre classe modèle « User »

```
public class User {
      private String nom;
      private String prenom;
      // Le constructeur par défaut
      public User() { }
      // Un constructeur avec paramètres si on le
souhaite
      public User(String nom, String prenom) { }
      // Getters
      public String getNom() {
          return nom;
      }
      public String getPrenom() {
          return prenom;
      }
      // ....
      // Setters
      public void setNom(String nom) {
          this.nom = nom;
      }
      public void setPrenom(String prenom) {
          this.prenom = prenom;
      }
}
```

- Pour gérer la **persistance des données**, il faut pouvoir se connecter à la base de données.
- On utilise l'API *Java DataBase Connectivity* (**JDBC**)
  - API appartenant à Java SE (et non Java EE) ;
  - Elle permet d'établir une connexion avec un SGBD, l'envoi de requêtes SQL à partir de l'application, et le traitement des données et erreurs retournées par le SGBD.

- Il existe de nombreux SGBD, et chacun a son mode de communication propre.
- JDBC dispose de nombreux **drivers** lui permettant de **communiquer avec ces SGBD**.
- Il suffit d'indiquer quel driver utiliser lorsqu'on initie une connexion à un SGBD.

```
try {
 Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
   // Gestion des erreurs
}
```

- On se connecte à un SGBD à l'aide de son URL et des identifiants de connexion.
  - L'URL se construit selon le modèle suivant : `jdbc:mysql://host:port/nomDeLaBase`
  - On appelle la méthode `getConnection` avec les paramètres

```
Connection connexion = DriverManager.getConnection(url, user, password);
```

  - Lorsqu'on a terminé, il faut **fermer la connexion**

```
connexion.close();
```

Généralement, on encapsule l'objet `connexion` dans un objet respectant le design pattern **Singleton**.

- On ne souhaite avoir qu'une seule instance gérant les connexions dans toute l'application.
- On souhaite que tous les clients passent par cette connexion afin d'éviter de potentiels conflits.

Singleton

```
public class Singleton {
    /*
     * Unique instance du Singleton pour l'ensemble du programme.
     */
    private static Singleton instance;
    /*
     * Constructeur privé, qui ne peut être appelé qu'à l'intérieur de
     * la classe.
     * Cela empêche de créer une instance de Singleton n'importe où
     * dans le programme à l'aide de new.
     */
    private Singleton() {}
```

Singleton

```
    /*
     * Méthode permettant de renvoyer une référence vers l'instance
     * unique du Singleton.
     * C'est le seul moyen pour obtenir un Singleton dans tout le
     * programme.
     */
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- On pourrait dès à présent faire des requêtes sur la base de données, à partir de la couche métier.
  Mais cela créerait trop de dépendance entre la couche métier et la couche de persistance.
- On va donc utiliser des éléments intermédiaires pour gérer l'**accès aux données** : les **DAO** (Data Access Object).

- Les DAO…
  - Encapsulent la logique liée à la base de données ;
  - Respectent le **CRUD** ;
  - Rendent le **code modulable** en séparant l'accès aux données et leur traitement depuis la couche métier ;
  - Suivent généralement le design pattern « **Singleton** ».

- Le **CRUD** = les 4 opérations de base pour la persistance
  - **Create** : ajout de nouvelles données ;
  - **Read** : lecture de données ;
  - **Update** : modification de données ;
  - **Delete** : suppression de données.

Pour implémenter le design pattern DAO, il faut :

- Créer une **interface Java** qui définit les méthodes que le DAO devra implémenter. C'est cette interface qui sera utilisée au niveau métier.
- Créer une classe qui **implémente l'interface**. C'est dans cette classe que seront seront effectués les accès à la base de données.

- L'accès aux données via JDBC se fait à l'aide de requêtes appliquées à un objet de classe `Statement`.
- On crée un `Statement` de la façon suivante :

```
Statement statement = connexion.createStatement();
```

- On peut ensuite écrire des requêtes et les exécuter...
  - avec `statement.executeQuery()` pour une requête SELECT ;
  - avec `statement.executeUpdate()` pour les autres requêtes.
- Le résultat d'une requête SELECT est un `ResultSet`. Pour une requête autre, le résultat est un `int`.

- Vous trouverez beaucoup de détails dans **la Javadoc** : https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html
- Lorsqu'on a terminé de traiter une requête, il faut penser à **fermer les objets** `Statement` et `ResultSet` à l'aide de leur méthode `close()`.
- De manière générale, il est conseillé d'utiliser des blocs `try...catch` lorsqu'on fait une requête avec JDBC.

Les **Services** font le lien entre la persistance (les DAO) et la couche présentation de l'application. (Singleton)

- Ils font appel aux DAO
  - Définissent généralement des méthodes dont les noms sont identiques à ceux des méthodes des DAO.
- Ils offrent la possibilité de traiter les données
  - Vérification des variable `null` ;
  - Mapping des types de données complexes ;
  - etc ...

Une servlet = du code qui s'exécute côté serveur.

- Elle est le centre névralgique de l’application.
- Elle reçoit une requête du client, elle effectue des traitements et renvoie le résultat.
- Une servlet peut être invoquée plusieurs fois en même temps pour répondre à plusieurs requêtes simultanées.

On dit **UNE** servlet. :-)

Une Servlet est une simple classe Java qui est capable de traiter n'importe quel type de couple requête/réponse.

Dans le cas des requêtes HTTP, on verra le plus souvent 2 types de requêtes effectuées par le client :

- GET : Récupérer une ressource
- POST : Soumettre une ressource

Mais il en existe bien d'autres !

Lorsqu'un serveur HTTP reçoit une requête, il crée deux objets qu'il transmet au conteneur de Servlets :

- `HttpServletRequest`
  - Contient toutes les informations relatives à la requête HTTP envoyée par le client.
- `HttpServletResponse`
  - Correspond à la réponse HTTP qui sera renvoyée au client par le serveur. L'objet est initialisé, mais pourra être rempli/complété lors de son passage dans la servlet avant d'être renvoyé par le serveur.

- Pour créer une servlet, il suffit d'implémenter l'interface `javax.servlet.Servlet`.
- Pour créer une servlet HTTP, il suffit d'hériter de la classe abstraite `HttpServlet` qui implémente déjà l'interface `Servlet`, et de définir les méthodes manquantes.

```
package com.excilys.servlet;

import javax.servlet.http.HttpServlet;

public class UserServlet extends HttpServlet {
}
```

On doit notamment définir au moins une méthode `doGet()` ou `doPost()`, correspondant à des points d'entrée de l'application web dépendant du type de requête envoyée par le client (GET ou POST).

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        doGet(request, response);
}
```

Une fois que les servlets ont été écrites :

- Pour que l'application sache quelles servlets utiliser, il faut les déclarer dans le fichier `web.xml` situé dans le dossier de l'application web `WEB-INF/`.
- Il faut fournir un *mapping URL* à la servlet. Celui ci permet de déterminer quelle servlet sera appelée en fonction de l'URL de la requête reçue.

Dans le cas de notre servlet pour les « Users » :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">
     <display-name>nom_de_l'application</display-name>
     <servlet>
          <servlet-name>UserServlet</servlet-name>
          <servlet-class>chemin_de_packages.UserServlet</servlet-class>
     </servlet>
     <servlet-mapping>
          <servlet-name>UserServlet</servlet-name>
           <url-pattern>/user</url-pattern>
     </servlet-mapping>
</web-app>
```

On définit dans la servlet les services par lesquels on souhaite interagir avec notre base de données.

```
import javax.servlet.http.*;
import com.excilys.service.UserService;

public class UserServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws ServletException, IOException {
        UserService userService = UserService.getInstance();
        userService.getNumber();
    }
}
```

Les JSP permettent d'introduire du code Java dans des tags prédéfinis à l'intérieur d'une page HTML.

Une JSP est habituellement constituée :

- de tags et balises HTML ;
- de tags JSP (tagLibs) ;
- de scriptlets (code Java intégré à la JSP via des balises spécifiques à JSP).

Les fichiers JSP possèdent par convention l'extension `.jsp`.

Syntaxe des balises spécifiques aux JSP :

- Commentaire `<%-- --%>`
- Déclaration de variable / méthode `<%! %>`
- Permettant d'afficher une valeur `<%= %>`
- Directive spéciale `<%@ %>`
- Scriptlet `<% %>`

```
<!-- Exemple de balise spéciale pour choisir l'encodage texte de la page -->
<%@ page pageEncoding="UTF-8" %>
```

Les **Expressions Languages** (EL) permettent d'afficher la valeur d'une expression : `${expr}`

Leur utilisation offre les possibilités suivantes :

- Utilisation directement dans le HTML (pas de tag JSP) ;
- Permet d'évaluer les expressions booléennes ;
- Permet de tester si une expression vaut `null` ;
- Permet de manipuler les **JavaBeans** plus facilement : `${user.nom}` appelle automatiquement le getter `getNom()` de la classe User (d'où l'intérêt de respecter la convention JavaBean)

Java Server Page

exemple.jsp

```
<html>
  <head>
      <title>Une première JSP</title>
  </head>
  <body>
      <% for (int i =  0 ; i < 2 ; i++) { %>
          <p>Bonjour le monde ! <%= i %></p>
      <% } %>
  </body>
</html>
```

Servlet

```
out.write("<html>\n");
out.write("  <head>\n");
out.write("    <title>JSP Page</title>\n");
out.write("  </head>\n");
out.write("  <body>\n");
out.write("    <p>Bonjour le monde ! 0</p>\n");
out.write("    <p>Bonjour le monde ! 1</p>\n");
out.write("  </body>\n");
out.write("</html>\n");
```

exemple.html

```
<html>
   <head>
       <title>JSP Page</title>
   </head>
   <body>
       <p>Bonjour le monde ! 0</p>
       <p>Bonjour le monde ! 1</p>
   </body>
</html>
```

Pourquoi utiliser les JSP ?

⇒ Permet d'avoir des pages dynamiques !

⇒ Cela permet de séparer le design front pur du reste du code Servlet.

⇒ Meilleur respect du design pattern *MVC*

On sait maintenant manipuler les données sur la page qui va s'afficher grâce à la JSP.
Comment récupérer les données de la servlet dans la JSP ?

On les ajoute en tant qu'attribut de la requête

```
request.setAttribute("nom", valeur);
```

et on peut ensuite les récupérer dans la JSP

```
<% request.getAttribute("nom"); %>
```

On peut aussi récupérer des paramètres de la requête envoyée par le client, soit dans la servlet, soit dans la JSP :

- Par exemple dans la requête GET suivante :
  http://exemple.fr/path_servlet?param1=true&param2=0

```
request.getParameter("param1");
// dans l’exemple, renvoie true
```

- Cela fonctionne aussi pour les requêtes POST.

Références

- Le site web de J.-M. Doudoux : http://www.jmdoudoux.fr/java/dej/
- Les définitions Wikipedia de Java EE, Java SE, Java Platform…
- https://zestedesavoir.com/tutoriels/591/creez-votre-application-web-avec-java-ee/
- https://www.mistra.fr/tutoriels-java/tutoriel-jee.html
- Bien sûr le site de la JavaDoc EE 7 : https://docs.oracle.com/javaee/7/api/

Excilys
Développeurs de passion