Base des applications JavaEE

Développer des applications Web simple avce Servlets, JSP et JDBC

- Java EE
- Servlet API
- JSP/JSTL
- JDBC
- Filtres

Java EE

Qu'est ce que JavaEE ?

Java EE ou J2EE ou Jakarta EE est un ensemble de spécifications de différentes API orientée pour le développement d'applications professionnelles et d'entreprise. Chacune de ces spécifications d'API répond à un besoin courant de ce type d'application, comme par exemple, accèder à une base de donnée, répondre à des requêtes HTTP, etc ...

Les Serveurs d'applications

Les serveurs d'application sont les environnement d'exécutions pour les applications JavaEE. Ils implémentes certaines ou toutes les spécifications d'API. Quelques exemples :

- Référence : Oracle Glassfish
- Commerciaux :
  - WebSphere (IBM)
  - Weblogic (Oracle)
- Open source :
  - Tomcat (TomEE)
  - JBoss/Wildfly

Ainsi, les applications JavaEE ne sont pas exécutées comme les applications JavaSE via une méthode `main`. Elles doivent être packagées puis déployées sur un serveur d'application. Heureusement, notre IDE rend cette tâche triviale.

Servlet API

Les servlets

La Servlet API est une brique fondamentale de la spécification JavaEE. Elle permet de développer des composants gérant des requêtes HTTP. L'instanciation des servlets n'est pas gérées par le programmeur, mais par ce que l'ont appelle le conteneur de servlets. Il s'agit d'une partie du serveur d'application qui orchestre le cycle de vie des servlets ainsi que les threads nécessaires à la gestion des requêtes.

Pour créer une servlet, il suffit de créer une classe qui étend la classe `HttpServlet` et de lui affecter l'annotation `@WebServlet` avec en paramètre la route qui est être gérée par la servlet.

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


}
```

Dans une servlet vous pouvez redéfinir certaines méthodes, notamment `doGet` et `doPost`. Ces méthodes sont appelées par le conteneur lorsqu'une requête HTTP est faite sur la route de la servlet. `doGet` gère les requête avec la méthode HTTP GET et `doPost` gère les requêtes avec la méthode HTTP POST.

```
@WebServlet("/helloworld")
public class HelloWorldServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {


    }


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


    }
}
```

Les paramètres de cetes méthodes, représentent respectivement la requête HTTP et la réponse HTTP.

Objet requête

Paramètres

L'objet requête permet de récupérer des paramètres soit passés dans l'URL de la requête dans le cas d'une requête HTTP GET, ou paramètres d'un formulaire (passés dans le Body de la requête au format FormData) :

```
String monParam = request.getParameter("nomDuParam");
```

Session

L'objet requête permet également de récupérer la session du client. Dans la session, il est possible de stocker des variables et de les récupérer. La session est propre à un client.

```
// Récupérer la session
HttpSession session = request.getSession();

// Ecrire une variable dans la session
session.setAttribute("profil", profil);

// Récupérer une variable de session
ProfilUtilisateur profil = (ProfilUtilisateur) session.getAttribute("profil");
```

Objet réponse

L'objet réponse permet d'écrire du texte dans le Body de la réponse HTTP :

```
response.getOutputStream().println("Hello world");
```

On peut également faire une redirection vers une autre route de l'application :

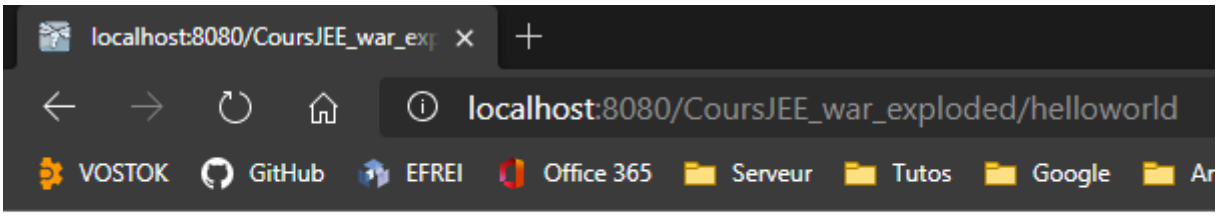
```
response.sendRedirect("/helloworld");
```

Première Servlet

Vous pouvez désormais créer une servlet "Hello world" :

```
@WebServlet("/helloworld")
public class HelloWorldServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        response.getOutputStream().println("Hello world");
    }
}
```

Ensuite lancez le projet, une page web dans votre navigateur devrait apparaître. Ajoutez `/helloworld` à la fin de l'URL et vous devriez voir le message hello world :




```
Hello world
```

Récupération d'un paramètre GET

Pour récupérer un paramètre GET, on utilise la méthode `getParameter` de la requête :

```
@WebServlet("/helloworld")
public class HelloWorldServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        String name = request.getParameter("name");
        response.getOutputStream().println("Hello " + name + "!");
    }
}
```

Relancez votre projet et ajoutez un paramètre GET dans l'URL !

Soumission d'un formulaire

Nous allons maintenant voir comment gérer une soumission de formulaire avec une servlet. Dans le dossier `webapp`, vous devriez trouver une page `index.jsp`. Considérer là comme une simple page HTML, nous verrons les JSP dans la partie d'après. Si elle n'est pas présente créez simplement une page `index.html`. Ecrivez maintenant un formulaire POST simple :

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

  <input type="text" name="monParam"/>
  <button type="submit">Envoyer</button>

</form>
```

Dans ce formulaire, l'action est mappée sur l'URL de notre servlet. Il faut donc que notre servlet implémente la méthode `doPost` pour gérer les requêtes POST. On peut ensuite récupérer le paramètre POST de la même façon que les paramètres GET :

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

  ...

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        String name = request.getParameter("monParam");
        response.getOutputStream().println("Hello " + name + "!");
    }
}
```

JSP/JSTL

Les Java Server Pages

Ecrire une interface HTML dans une string Java n'est pas très pratique. C'est pourquoi il est possible d'utiliser pour ce faire les JSP. Les Java Server Pages sont des templates de vues qui sont rendus coté serveur. Lorsque vous renvoyez une JSP depuis une Servlet, elle va être interprétée et le code HTML résultant sera placé dans le Body de la réponse HTTP. Ces vues sont templatables par du code Java mais c'est une mauvaise pratique car du code Java exprimant de la logique ne devrait pas résider dans une vue. C'est pourquoi il est recommandé d'utilise JSTL, la JSP Standard Tag Library.

Pour créer une JSP qui sera renvoyées par une servlet, créer un fichier `.jsp` à dans le dossier `src/main/webapp/WEB-INF`. Les fichiers de styles et de scripts sont à mettre directement dans le dossier `src/main/webapp` et sont accessibles à partir de l'URL racine du projet.

Il est possible de passer des objets Java à une JSP depuis une servlet :

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

Pour renvoyer une JSP depuis une servlet, utilisez le code suivant :

```
this.getServletContext().getRequestDispatcher("/WEB-INF/maJsp.jsp").forward(request, response);
```

JSP Standard Tag Library

Installation

Pour installer JSTL, ajouter la dépendance au `pom.xml` de votre projet :

```
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
    <version>2.0.0</version>
</dependency>
```

Pour importer la JSTL dans une JSP utilisez les balises suivantes :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Balise de la JSTL

Afficher une variable

La balise `<c:out>` permet d'afficher une variable passée par la Servlet :

```
<c:out value="${ variable }">Valeur par défaut si la variable est null</c:out>
```

Cette balise évalue en fait une expression écrite dans un langage appelé EL pour Expression Language. C'est un langage simple qui permet de faire des petites opérations sur les variables. Il permet d'accèder par leur nom aux variables passés à la JSP par la servlet.

Il permet également d'accèder à un champs d'un objet Java si celui ci possède un Getter correctement nommé en utilisant l'opérateur `.`. EL supporte également les opérateurs arithmétiques et logique ainsi que les expressions ternaires. Le mot clé `empty` permet de vérifier si une valeur est nulle.

Itération sur une liste

La balise `<c:forEach>` permet d'itérer sur les éléments d'une liste :

```
<c:forEach items="${ maListe }" var="monElement" varStatus="status">
   <p>N°<c:out value="${ status.count }" /> : <c:out value="${ titre }" /> !</p>
</c:forEach>
```

Ici la variable `monElement` dansl'attribut `var` de la boucle correspond à l'élément courant de la liste. Quant à la variable `status` (optionnelle) définie dans l'attribut `varStatus` elle permet d'obtenir des informations sur l'élément courant de la boucle et possède des propriétés comme :

- `index` : l'indice de l'élément
- `first` : vrai si l'élément est le premier
- `last` : vrai si l'élément est le dernier

Rendu conditionnel

La balise `<c:if>` permet de faire du rendu conditionnel :

```
<c:if test="${ variable == '1' }">
    C'est vrai !
</c:if>
```

La contenu de la balise n'est rendu que si l'expression EL dans l'attribut `test` est évaluée à `true`.

Rendu conditionnel à choix multiple

La balise `<c:choose>` permet de faire du rendu conditionnel à choix multiples :

```
<c:choose>
    <c:when test="${ variable = '1' }">C'est égal à 1</c:when>
    <c:when test="${ variable = '2' }">C'est égal à 2</c:when>
    <c:when test="${ variable = '3' }">C'est égal à 3</c:when>
    <c:otherwise></c:otherwise>
</c:choose>
```

JDBC

JDBC pour Java DataBase Connectivity est une spécification d'API pour accèder à une base de donnée relationnelle depuis une application Java. Pour se connecter à une base de donnée, il faut un Driver, qui implémente JDBC pour un moteur de base de donnée relationnelle donné. Par exemple il en existe un pour Oracle, un pour MySQL, pour PostgreSQL, etc ...

Installation du Driver

Pour installer le Driver JDBC pour MySQL, ajoutez la dépendance suivante dans votre `pom.xml` :

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.22</version>
</dependency>
```

Pour Oracle utilisez cette dépendance :

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>21.1.0.0</version>
</dependency>
```

N'oubliez pas refresh vos dépendances Maven après avoir modifié votre `pom.xml` !

Connexion à la base de donnée

Pour se connecter à une base de donnée utilisez la méthode static `DriverManager.getConnexion()` en passant en paramètre la chaine de caractère de connexion ainsi que les identifiants.

Pour MySQL :

```
Class.forName("com.mysql.cj.jdbc.Driver")
Connection connexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/nomDeLaBase","root", "mot
```

```
de passe");
```

Si MySQL vous lance une erreur de TimeZone, ajoutez ceci à la fin de l'URL JDBC : `?useLegacyDatetimeCode=false?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC`.

Pour Oracle :

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection connexion = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","root", "mot de passe");
```

Il est recommandé de faire un singleton de l'instance de la connexion à la base de donnée.

Interactions avec la base de donnée

Requêtes

L'objet `Statement` permet d'exécuter des requêtes SQL :

```
Statement statement = connexion.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM Utilisateurs");
```

Le `ResultSet` est un itérateur sur les enregistrements contenues dans la réponse à la requête. On peut l'exploiter en itérant dessus grâce à la méthode `next()` :

```
while (rs.next()) {
  Utilisateur utilisateur = new Utilisateur();
  utilisateur.setPrenom(rs.getString("PRENOM"));
  utilisateur.setNom(rs.getString("NOM"));
  listeUtilisateur.add(utilisateur);
}
```

On peut en ensuite récupérer le contenu de l'enregistrement courant avec des méthodes comme `getString()`, `getInteger()`, etc ... en leur le passant le nom de la colonne dans la base.

Les classes `Statement` et `ResultSet` doivent toutes les deux êtres fermées après leur utilisation car il s'agit de ressources non managées. Pour une syntaxe concise on peut utiliser le "Try With

Resource" :

```
try (Statement statement = this.connection.createStatement();
    ResultSet result = statement.executeQuery("SELECT * FROM Utilisateurs");) {
     while (result.next()) {
        Utilisateur utilisateur = new Utilisateur();
        utilisateur.setPrenom(rs.getString("PRENOM"));
        utilisateur.setNom(rs.getString("NOM"));
        listeUtilisateur.add(utilisateur);
     }
} catch (SQLException e) {
    System.err.println("Erreur à l'exécution de la requête SQL");
}
```

Requêtes DML

Pour modifier des données dans la base avec des commandes `INSERT`, `UPDATE` ou `DELETE` on utilise la méthode `executeUpdate()` de la classe `Statement` :

```
int modifiedRows = statement.executeUpdate("DELETE FROM Utilisateurs WHERE NOM = 'Shepard'");
```

`executeUpdate()` retourne le nombre de lignes modifiées par la requête.

SQL Dynamique

Il est possible de faire des requêtes préparées en utilisant la classe `PreparedStatement` :

```
PreparedStatement preparedStatement = connexion.prepareStatement("DELETE FROM Utilisateurs WHERE NOM = ?");
```

On peut ensuite affecter des paramètres :

```
preparedStatement.setString(1,"Shepard");
```

Enfin, on exécute le `PreparedStatement` en utilise soit `executeUpdate()` soit `executeQuery()` selon qu'il s'agisse d'une commande DML ou d'une requête.

Filtres

Les filtres permettent d'intercepter des requêtes HTTP avant qu'elles arrives à leur Servlet afin d'exécuter de la logique. Cela est notamment utile pour restreindre l'accès à certaines routes à des utilisateurs non authentifiés ou ne possèdant pas le bon rôle.

Créer un filtre

Pour créer un filter, il faut créer une classe qui implémente l'interface `Filter` et annotées avec `@WebFilter`. En paramètre de l'annotation, il faut passer le paramètre urlPatterns, pour préciser les routes qui seront filtrée par ce filtre.

```
@WebFilter(urlPatterns = "/admin/*")
public class AdminFilter implements Filter {



    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws ServletException,
IOException {

    }

    public void init(FilterConfig config) throws ServletException {

    }

      public void destroy() {
    }

}
```

Par exemple ce filtre va intercepter les requêtes sur toutes les routes qui commencent par "/admin/".

Actions dans un filtre

Dans un filtre on peu récupérer la requête et la réponse HTTP par un simple cast :

```
var request = (HttpServletRequest) req;
var response = (HttpServletResponse) resp;
```

On peut ensuite accèder à la session pour vérfier des informations ou faire une redirection. Pour autoriser la poursuite de la requête normalement utilisez :

```
chain.doFilter();
```

Et pour retourner un code HTTP particulier (401 Unauthorized ou 403 Forbidden) et arrêter là la requête :

```
response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
return;
```