

TP Programmation Web

Création d'une mini appli web

Présentation du TP :

Le but du TP est de manipuler les concepts de base d'une application web : backend java hébergé dans Tomcat, webservices REST, accès à une base de données.

Sommaire :

1 Configuration eclipse	1
1.1 Projet	1
1.2 Serveur	1
2 Création d'une mini application bibliothèque.....	2
2.1 Configuration webservice de l'application	2
2.2 Contrôleur	2
2.3 Objet métier	3
2.4 DAO (data access object).....	3
2.5 Appel du DAO dans le contrôleur	3
2.6 Affichage dans index.html	3
2.7 Connexion BD	4
2.8 Recherche d'un livre	4
2.9 Historique de recherche	5
2.10 Formulaire de création	5
3 Packaging.....	6

1 Configuration eclipse

1.1 Projet

Dans le workspace créer un nouveau projet de type « dynamic web project » :

- menu File / new > dynamic web project
- Dans l'écran « dynamic web projet » : project name = **TPBookstore**
Vérifier que le « target runtime » est bien Apache Tomcat v10.1
Sélectionner la valeur 4.0 pour « dynamic web module version »
cliquer sur next.
- Dans l'écran « java » : cliquer sur next
- Dans l'écran « web module » : cliquer sur next.
Puis cliquez sur Finish.

1.2 Serveur

- Aller dans la vue Servers (Window / show view / Others / servers
- Ajouter le projet **TPBookstore** dans le conteneur web de Tomcat :

- clic droit sur l'instance de serveur, choisir « add and remove... »
- Ajouter le projet TPBookstore dans la liste de droite, puis finish.
- Récupérer les bibliothèques nécessaires au projet et les copier dans /src/main/webapp/WEB-INF/lib

2 Création d'une mini application bibliothèque

2.1 Configuration webservice de l'application

- Dans eclipse il y a des configurations de fenêtres et menus personnalisés pré enregistrées (perspective). Afficher la perspective « java » : menu Window / Perspective / Open perspective / Java
- Créer un package nommé org.tutorial (File / new / Package ou bien icône dédiée dans la barre d'outils)
- Créer une classe **AppConfig** :

```
package org.tutorial;
import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;


@ApplicationPath("/api")
public class AppConfig extends Application {
}
```

Ceci à pour but de déclarer un chemin racine pour tous les webservices.

2.2 Contrôleur

- Créer une classe **BookController**
- Définir un chemin pour cette ressource en mettant l'annotation `@Path("/book-management")` avant la déclaration `public class BookController`
- Créer une méthode `hello()` :

```
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/hello")
public String hello() {
    return "Hello World!";
}
```

- Démarrer le serveur 
- Tester la méthode hello depuis le navigateur : <http://localhost:8080/TPBookstore/api/book-management/hello>
- La page doit afficher Hello World !

Eléments de l'url :

localhost = serveur

8080 : port

TPBookstore : racine de contexte de l'application web

api : chemin de l'application défini pour les webservices

book-management : chemin défini pour le contrôleur **BookController**

hello : chemin de la ressource

2.3 Objet métier

- Créer une classe Java nommée **Book** contenant les attributs :

```
private int id;  
private String title;  
private String author;
```

- Générer les getter/setter et un constructeur prenant ces paramètres.
 - Astuce 1 (pour les fans de clics) : clic droit dans la classe / menu source / generate getters and setters
 - Astuce 2 (pour les fans du clavier) : commencez à créer une méthode en tapant « getT » puis control+space : sélectionner « getTitle() : getter for title. »
Ceci ne marche que si la méthode n'existe pas encore.

2.4 DAO (data access object)

a) Interface et implémentation

- Créer une interface **BookDAO** définissant la méthode:
`List<Book> findByAll()`
- Créer la classe **BookDAOMockImpl** qui implémente **BookDAO**
- Implémenter la méthode `findByAll` en construisant une liste de quelques livres « en dur » :
 - création d'un objet de type `List`,
 - ajout d'instances de **Book**.

L'intérêt de cette classe est de pouvoir bouchonner (*mock* en anglais) les accès à la base de données pour se concentrer sur les autres couches de l'application.

2.5 Appel du DAO dans le contrôleur

- Créer un nouvel attribut et une nouvelle méthode dans le **BookController**

```
private BookDAO bookDAO = new BookDAOMockImpl();
```

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
@Path("/books")  
public String getBooks() {  
  
    List<Book> books = bookDAO.findByAll();  
  
    GsonBuilder builder = new GsonBuilder();  
    Gson gson = builder.create();  
    String json = gson.toJson(books);  
    return json;  
}
```

- Tester : redémarrer le server et lancer
<http://localhost:8080/TPBookstore/api/book-management/books>

2.6 Affichage dans index.html

- Récupérer le fichier `index.html` donné en exemple et le copier dans `/src/main/webapp/`

- Tester <http://localhost:8080/TPBookstore/> pour voir la liste des livres s'afficher.

2.7 Connexion BD

a) DataSource

- Récupérer la classe **DBManager** qui fournit des méthodes de connexion à la base de données.
- Copier le fichier config.properties à la racine de src/main/java

b) DAO

- Créer une classe **BookDAOImpl** qui implémente **BookDAO**
- Implémenter findByAll pour aller requêter la base :
 - Récupérer une connexion de type java.sql.Connection par la méthode :
`DBManager.getInstance().getConnection()`
 - Créer un java.sql.Statement à partir de cette connexion en utilisant:
`connexion.createStatement()`
 - Exécuter la requête SQL et récupérer un java.sql.ResultSet :
`rs = statement.executeQuery(« select ... ») ;`
(voir le fichier books.sql pour la structure de la table)
 - Itérer sur le resultSet :
`while (rs.next()) {`
 - Récupérer les colonnes, par exemple :
`String title = rs.getString("title");`
 - Construire la liste de résultats en instanciant des objets Book avec les propriétés récupérées de la base.

c) contrôleur

- Dans **BookController** changer l'implémentation de **bookDAO** en **BookDAOImpl** :
`private BookDAO bookDao = new BookDAOImpl();`
- Redémarrer le serveur et rafraichir l'affichage de <http://localhost:8080/TPBookstore/>

2.8 Recherche d'un livre

a) DAO

- Dans **BookDAO** créer la méthode
`List<Book> findByTitle(String searchText);`
- Implémenter cette méthode dans **BookDAOImpl** par une méthode similaire à findByAll, à la requête SQL près ... (requête affichant les livres dont le titre contient le paramètre « searchText »)

Question : Une classe du projet ne compile plus, laquelle ? Pourquoi ? Que faire ? ...

b) Formulaire web

- Ajouter dans la page
 - un input text, id= "`input_title`"
 - un bouton avec un onClick= « `searchByTitle()` »
 - l'implémentation de la méthode javascript nommée `searchByTitle()` :

```
var title = ... //à vous de récupérer la valeur du champ texte
var url = "/TPBookstore/api/book-management/books?title="+title;
loadBooks(url);
```

c) contrôleur

- modifier dans **BookController** la méthode `getBooks` :
`public String getBooks(@QueryParam("title") String title)`

La valeur du paramètre « title » est automatiquement injectée dans la variable `title`.

- Modifier le contenu de cette méthode pour appeler :
 - `bookDAO.findAll()` s'il n'y a pas de paramètre
 - `bookDAO.findByTitle` s'il y a un paramètre

Redémarrer le serveur et rafraichir la page web pour tester la recherche par titre.

2.9 Historique de recherche

On va utiliser la session pour stocker l'historique de recherche.

- Créer une nouvelle méthode dans **BookController**

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/books-session")
public String getBooksWithSessionHistory(@Context HttpServletRequest request,
@QueryParam("title") String title) {

    HttpSession session = request.getSession();
    List<String> queries = (List<String>) session.getAttribute("queries");
    if (queries == null) {
        queries = new ArrayList<>();
        session.setAttribute("queries", queries);
    }
    queries.add(title);
    System.out.println("liste des recherches stockées en session :");
    queries.stream().forEach(x -> System.out.println("-" + x));
    return getBooks(title);
}
```

- Modifier `index.html` pour que le lien vers le backend pointe vers cette nouvelle méthode
- Rafraichir et tester des recherches, vérifier que la log du serveur affiche la liste des recherches

2.10 Formulaire de création

- Dans `index.html` rajouter un formulaire pour créer un nouveau livre :
 - Action = `/TPBookstore/api/book-management/createbook`
 - Method : POST
 - Un input text `name="book_title"`
 - Un input text `name="book_author"`
 - Un bouton submit
- Dans **BookController** ajouter la méthode :

```
@POST
@Path("/createbook")
@Consumes("application/x-www-form-urlencoded")
public void createBook(@FormParam("book_title") String bookTitle,
@FormParam("book_author") String bookAuthor) {
    Book book = new Book(0, bookTitle, bookAuthor);
    System.out.println("new book: " + book);
}
```

- Rafraîchir la page index.html, saisir des valeurs et vérifier dans la log du serveur si un livre a été créé.

Question : comment est affichée une instance de Book ? Comment rendre l'affichage plus lisible ?

- Enrichir la classe **Book** pour que l'affichage soit plus lisible
- Retester pour voir la différence de log

3 Packaging

a) Export de l'application

Pour pouvoir être installée dans un serveur d'application, l'application a besoin d'être packagée selon les normes JEE.

- Exporter le projet au format war : dans eclipse fichier/exporter/war
- Choisir la destination, nommer le fichier biblio.war et valider

b) Analyse de la structure

- Le war est en fait un fichier zip. Le dézipper et analyser la structure des répertoires

Dans l'arborescence rechercher où se trouvent les classes compilées.