Services RESTful avec Jersey

Laurent Réveillère
http://www.labri.fr/perso/reveille/

Résumé

Dans ce TP vous allez développer des servlets un peu plus compliquées que lors du TP précédent. Pour cela, nous allons utiliser deux bibliothèques qui sont Jersey et Jackson. La première permet d'utiliser des annotations java pour développer les servlets plus facilement, et la seconde permet de sérialiser et déserialiser des objets Java en JSON (JavaScript Object Notation). Nous allons faire une application relativement simple qui gère un carnet d'adresses.

1 Développement avec Eclipse

Pour développer des servlets, il est conseillé d'utiliser un IDE tel que *Eclipse for Java EE Developers* (dernière version en date : *Neon*). Nous allons voir comment créer une application Web Jersey sous Eclipse et comment la déployer sur notre serveur Tomcat qui se trouve sur Morpheus.

Prérequis. Comme nous allons développer sur notre poste local puis déployer notre application Web sur notre serveur, il faut faire attention d'avoir la même version du JDK Java. Vous devriez avoir la version 7 sur votre serveur, donc il faut également installer cette version en local sur votre poste. Pour éviter de gérer à la main les jar à ajouter à notre projet, nous allons utilisé Maven. Si vous voulez tester votre application en local, vous pouvez simplement installer tomcat sur votre poste client.

Création de l'application. Dans Eclipse, créer un nouveau projet de type *Dynamic web project*, appelé `HelloWorld`. Dans *Configuration*, sélectionner la version 1.8 de Java. Sur le dernier écran du wizard, cocher la case pour générer le fichier `web.xml`. Faire ensuite un clic droit sur le projet et sélectionner *Configure* puis *Configure to Maven Project*. Modifier le fichier `pom.xml` généré en ajoutant les dépendances suivantes après la fermeture du bloc `build`.

```
<dependencies>
   <dependency>
      <groupId>org.glassfish.jersey.containers</groupId>
      <artifactId>jersey-container-servlet</artifactId>
      <version>2.22.2</version>
   </dependency>
</dependencies>
```

Créer ensuite une classe nommé *HelloWorld* dans le package `fr.enseirb.t2` en utilisant le générateur d'Eclipse. Modifier le code de la classe `HelloWorldRessource` comme suit.

```
@Path("/hello")
public class HelloWorldResource {
    @GET
    @Produces("text/plain")
    public String getHelloWorld() {
        return "Hello World from text/plain";
    }
}
```

Modifier ensuite le fichier `web.xml` en ajoutant les informations suivantes pour que Jersey enregistre automatiquement toutes les ressources trouvées.

```
<servlet>
   <servlet-name>javax.ws.rs.core.Application</servlet-name>
</servlet>

<servlet-mapping>
   <servlet-name>javax.ws.rs.core.Application</servlet-name>
   <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Remarque. Il est possible de configurer plus finement les ressources disponibles en sous classant `Application` ou `RessourceConfig`. Voir `https://jersey.java.net/documentation/latest/user-guide.html` pour plus d'informations.

Test et Déploiement. Si vous avez installé un serveur Tomcat, vous pouvez tester votre application sur votre poste client. Pour déployer votre application sur votre serveur, la première étape consiste à exporter le projet dans une archive *.war* (pour *Web application ARchive*). Pour cela, faire un clic droit sur le projet et aller dans le menu *Export*. Enregistrer l'archive n'importe où sur votre machine. Pour ajouter une archive war à un serveur Tomcat, nous pouvons utiliser le gestionnaire d'application Tomcat accessible ici : `http://`*`serveur-tomcat`*`/manager/`. Cette page va vous demander un identifiant et un mot de passe. Comme c'est la première fois que vous utilisez le manager, vous n'avez pas encore configuré les droits d'accès à celui-ci. Cliquez sur *"annuler"* dans la fenêtre qui demande les identifiants. Un message d'erreur, ainsi que des instructions relatives à la configuration du serveur devraient s'afficher. Suivez ces instructions et redémarrez votre serveur.
Vous pouvez maintenant accéder à l'interface du manager et déployer l'archive war en utilisant l'interface prévue à cet effet. Tester votre application !

2 JSON et Jackson

Nous allons maintenant produire des données au format JSON. Pour ce faire, nous allons modifier un peu la configuration de notre projet, en précisant que nous souhaitons utiliser Jackson comme *provider* json. Pour ce faire, ajouter la dépendance suivante dans le fichier `pom.xml`.

```
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId>
  <version>2.22.2</version>
</dependency>
```

Créer ensuite une classe `User` contenant deux champs : `firstname` et `lastname`. Créer une classe `UserRessource` permettant de récupérer un utilisateur au format JSON.

Personnalisation du document JSON Pour personnaliser le format du document généré, utiliser différentes annotations fournies pas Jackson comme par exemple `@JsonProperty` pour modifier le nom d'un élément.

Configuration du provider JSON Pour configurer Jackson, nous devons créer une classe étendant `RessourceConfig` en lieu et place de la découverte automatique fournit par Jersey. Pour ce faire, créer la classe suivante et modifier le fichier `web.xml` en conséquence.

```
@ApplicationPath("/")
public class ContactManager extends ResourceConfig {

   public ContactManager() {
      // Register resources and providers using package-scanning.
      packages("fr.enseirb.t2");

      register(MyObjectMapperProvider.class);

      // Enable Tracing support.
      property(ServerProperties.TRACING, "ALL");
   }
}
```

La classe `MyObjectMapperProvider` doit ressembler à cela :

```
@Provider
public class MyObjectMapperProvider implements ContextResolver<ObjectMapper> {

   final ObjectMapper defaultObjectMapper;

   public MyObjectMapperProvider() {
       defaultObjectMapper = createDefaultMapper();
   }

   @Override
       public ObjectMapper getContext(final Class<?> type) {
       return defaultObjectMapper;
   }

   private static ObjectMapper createDefaultMapper() {
      final ObjectMapper result = new ObjectMapper();
      result.enable(SerializationFeature.INDENT_OUTPUT);

      return result;
   }
}
```

Aller plus loin. Ajouter les méthodes permettant d'ajouter et modifier un utilisateur ainsi que d'obtenir la liste de tous les utilisateurs.

3 Utilisation de MongoDB

Nous allons voir dans cette section comment utiliser MongoDB qui est une base de donnée orientée document. Dans ce type de base, on stocke directement des documents au format json binaire (bjson).

Mise en place Installer mongodb (`sudo apt-get install mongodb`) puis ajouter la dépendance suivante dans le fichier `pom.xml` pour avoir le connecteur mongodb en Java.

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.2.2</version>
</dependency>
```

Modifier la classe `User` précédente pour sauvegarder les utilisateurs dans la base de donnée. Ajouter une méthode `GET` pour lister tous les utilisateurs et une méthode `POST` pour ajouter un nouvel utilisateur. Voici ci-dessous le code nécessaire pour ajouter l'utilisateur `User user` dans la base.

```
try {
   mongoClient = new MongoClient();
   MongoDatabase db = mongoClient.getDatabase("myBase");
   MongoCollection<Document> collection = db.getCollection("myCollection");

   ObjectMapper mapper = new ObjectMapper();
   String jsonString = mapper.writeValueAsString(user);
   Document doc = Document.parse(jsonString);
   collection.insertOne(doc);
   return "Utilisateur " + user.getFirstname() + " " + user.getLastname() + " added
       successfully.";
} catch (Exception e) {
   e.printStackTrace();
} finally{
   mongoClient.close();
}
```