



Rapport personnel du module 133 - Réaliser des applications Web en Session-Handling

Langenegger Max

Début du module le 21.03.2024 , fin de module le 03.05.2023

Création du document le 21.03.2024 12:52:00
Version 1 du 18.04.2024

Table des matières

1	Introduction	4
2	Tests technologiques selon les exercices.....	4
2.1	Installation et Hello World	4
2.1.1	Observez la console pour comprendre comment le projet est lancé et comment il tourne ?	4
2.1.2	C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ?....	4
2.1.3	Quelle version de java est utilisée ?	5
2.1.4	S'il y a un serveur web, quelle version utilise-t-il ?	5
2.2	Conteneurisation.....	5
2.2.1	Pourquoi faire un container pour une application Java ?	5
2.2.2	Y a-t-il un serveur web ? Ou se trouve-t-il ?	5
2.2.3	A quoi faut-il faire attention ?	5
2.3	Création d'un projet Spring Boot	6
2.4	Connexion à la DB JDBC.....	6
2.5	Connexion à la DB JPA	7
2.5.1	À quoi sert l'annotation @Autowired dans vos controlleur pour les Repository ?	7
2.5.2	A quoi sert l'annotation @ManyToOne dans l'entité skieur ?	7
2.6	Connexion à la DB JPA avec DTO	7
2.7	Gestion des sessions.....	7
2.8	Documentation API avec Swagger	9
2.9	Hébergement	9
3	Analyse à faire complètement avec EA -> à rendre uniquement le fichier EA.....	10
3.1	Use case client et use case Rest	11
3.2	Activity Diagram d'un cas complet navigant dans les applications avec les explications	11
3.3	Sequence System global entre les applications	11
4	Conception à faire complètement avec EA -> à rendre uniquement le fichier EA.....	11
4.1	Class Diagram complet avec les explications de chaque application.....	11
5	Bases de données	11
5.1	Modèles WorkBench MySQL.....	11
5.2	Implémentation des applications <Le client Ap1> et <Le client Ap2>	11
5.3	Une descente de code client	11
6	Implémentation de l'application <API Gateway>	11
6.1	Une descente de code APIGateway	11
7	Implémentation des applications <API élève1> et <API élève2>	11

7.1	Une descente de code de l'API REST	11
8	Hébergement	11
9	Installation du projet complet avec les 5 applications	11
10	Tests de fonctionnement du projet	11
11	Auto-évaluations et conclusions	11
11.1	Auto-évaluation et conclusion de	11
11.2	Auto-évaluation et conclusion de	11
12	Conclusion	12
12.1	Ce que j'ai appris	12
12.2	Ce que j'ai aimé	12
12.3	Mes améliorations.....	12

1 Introduction

1 Analyser la donnée, projeter la fonctionnalité et déterminer le concept de la réalisation.

2 Réaliser une fonctionnalité spécifique d'une application Web par Session-Handling, authentification et vérification de formulaire.

3 Programmer une application Web à l'aide d'un langage de programmation compte tenu des exigences liées à la sécurité.

4 Vérifier la fonctionnalité et la sécurité de l'application Web à l'aide du plan tests, verbaliser les résultats et, le cas échéant, corriger les erreurs.

2 Tests technologiques selon les exercices

2.1 Installation et Hello World

2.1.1 Observez la console pour comprendre comment le projet est lancé et comment il tourne ?

Le projet se lance avec Spring Boot sur le port 8080.

```

langeneggerm@STEFA39-14:~/gs-rest-service$ /usr/bin/env /home/langeneggerm/.vscode-server/extensions/redhat.java-1.16.0-linux-x64/jre/17.0.6-linux-x86_64/bin/java @/tmp/cp_c01ajoxna99nnk
j6m1w60m15w.argfile com.example.restservice.RestServiceApplication

:: Spring Boot ::
(v3.2.0)

2024-03-21T14:11:15.691+01:00 INFO 6679 --- [main] c.e.restservice.RestServiceApplication : Starting RestServiceApplication using Java 17.0.6 with PID 6679 (/home/langeneggerm
/gs-rest-service/complete/target/classes started by langeneggerm in /home/langeneggerm/gs-rest-service)
2024-03-21T14:11:15.695+01:00 INFO 6679 --- [main] c.e.restservice.RestServiceApplication : No active profile set, falling back to 1 default profile: "default"
2024-03-21T14:11:16.173+01:00 INFO 6679 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-03-21T14:11:16.178+01:00 INFO 6679 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-03-21T14:11:16.178+01:00 INFO 6679 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.16]
2024-03-21T14:11:16.210+01:00 INFO 6679 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-03-21T14:11:16.211+01:00 INFO 6679 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 485 ms
2024-03-21T14:11:16.401+01:00 INFO 6679 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-03-21T14:11:16.406+01:00 INFO 6679 --- [main] c.e.restservice.RestServiceApplication : Started RestServiceApplication in 0.996 seconds (process running for 1.118)
2024-03-21T14:14:57.727+01:00 INFO 6679 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-03-21T14:14:57.727+01:00 INFO 6679 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-03-21T14:14:57.728+01:00 INFO 6679 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
  
```

```

1 {
2   "id": 1,
3   "content": "Hello, World!"
4 }
  
```

2.1.2 C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ?

Build :

Le build consiste à transformer les fichiers source de l'application en un format exécutable ou déployable. Cela implique généralement la compilation du code et la génération de fichiers JAR ou WAR, ainsi que d'autres tâches telles que la minification, l'optimisation, etc.

Run (Exécution) :

Une fois que le processus de build est terminé et que notre application est sous forme exécutable, on peut la lancer pour qu'elle accomplisse ses fonctions prévues. Cela implique de démarrer l'application ou le service correspondant et de laisser le programme s'exécuter pour répondre aux requêtes des utilisateurs ou effectuer les tâches prévues.

Pour run et build l'outil

2.1.3 Quelle version de java est utilisée ?

Dans les exercices on utilise OpenJdk17

2.1.4 S'il y a un serveur web, quelle version utilise-t-il ?

Oui, Apache Tomcat/10.1.16

2.2 Conteneurisation

2.2.1 Pourquoi faire un container pour une application Java ?

Les conteneurs offrent un environnement isolé qui encapsule l'application ainsi que toutes ses dépendances, y compris les bibliothèques, les dépendances système, etc. Cela garantit que l'application fonctionnera de la même manière quel que soit l'environnement dans lequel elle est déployée, qu'il s'agisse d'un ordinateur de développement, d'un serveur de test ou d'un environnement de production.

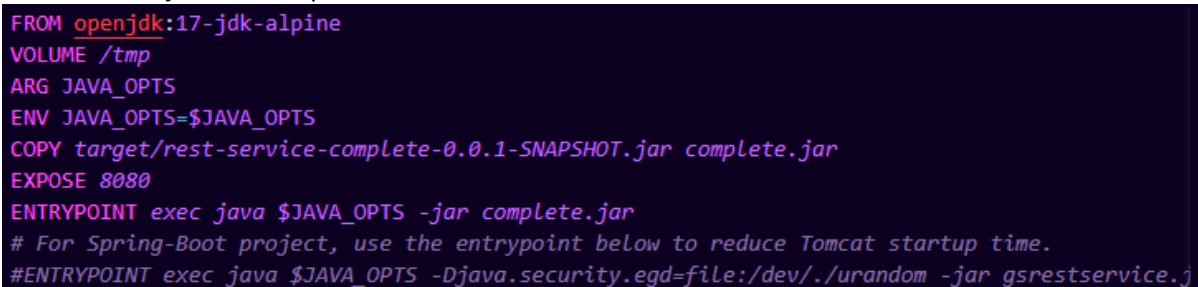
2.2.2 Y a-t-il un serveur web ? Ou se trouve-t-il ?

Il y a un serveur nommé Tomcat qui se trouve compilé dans le jar.

2.2.3 A quoi faut-il faire attention ?

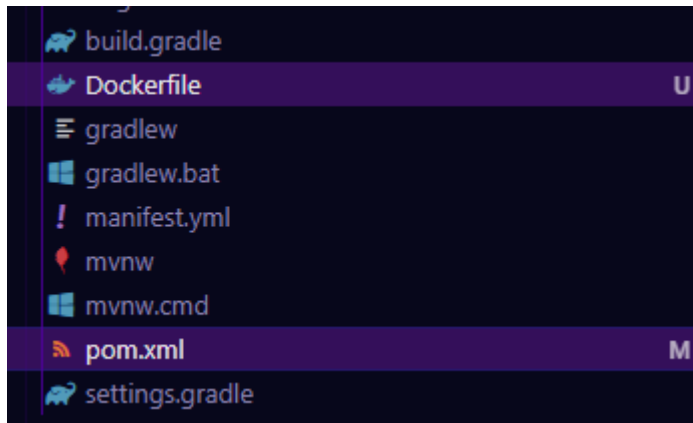
Il y a différents points à faire attention notamment :

La version du jdk installé qui doit être la version 17



```
FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAVA_OPTS
ENV JAVA_OPTS=$JAVA_OPTS
COPY target/rest-service-complete-0.0.1-SNAPSHOT.jar complete.jar
EXPOSE 8080
ENTRYPOINT exec java $JAVA_OPTS -jar complete.jar
# For Spring-Boot project, use the entrypoint below to reduce Tomcat startup time.
#ENTRYPOINT exec java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar gsrestservice.j
```

Mais aussi l'emplacement du fichier Docker qui doit se trouver au même niveau que le fichier pom.xml.



2.3 Création d'un projet Spring Boot

Quelles sont les annotations utilisées (commencent par @) dans votre contrôleur ?

- **GetMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP GET sur des méthodes de gestion spécifiques. Par exemple, `@GetMapping("/home")` mappe les requêtes GET à l'URL `"/home"` à la méthode de gestion correspondante.
- **PostMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP POST sur des méthodes de gestion spécifiques. Par exemple, `@PostMapping("/submit")` mappe les requêtes POST à l'URL `"/submit"` à la méthode de gestion correspondante.
- **PutMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP PUT sur des méthodes de gestion spécifiques. Par exemple, `@PutMapping("/update")` mappe les requêtes PUT à l'URL `"/update"` à la méthode de gestion correspondante.
- **RequestBody** : Cette annotation est utilisée pour lier le corps de la requête HTTP à un objet de domaine dans une méthode de gestion. Le corps de la requête HTTP est passé dans une forme appropriée à la méthode de gestion.
- **RequestParam** : Cette annotation est utilisée pour lier les paramètres de requête à une méthode de gestion. Par exemple, `@RequestParam("id") String id` lie le paramètre de requête `"id"` à la variable String `"id"`.
- **RestController** : Cette annotation est utilisée au niveau de la classe pour indiquer qu'une classe est un contrôleur où les méthodes de gestion renvoient le corps de la réponse directement, et non une vue. Elle est une combinaison de `@Controller` et `@ResponseBody`.

2.4 Connexion à la DB JDBC

Avec cet exercice on interagit avec un DB pour aller chercher des informations. La méthode qu'on utilise pour faire les requêtes c'est JDBC.

Mettre à la place de localhost vu l'existence de deux containers :

```
final String url = "jdbc:mysql://host.docker.internal:" + port + "/" + dbName  
+ "?serverTimezone=CET";
```

Utiliser la ligne suivante sur la méthode de connexion pour que ça marche sur le web :

```
Class.forName("com.mysql.jdbc.Driver");
```

2.5 Connexion à la DB JPA

2.5.1 À quoi sert l'annotation @Autowired dans vos contrôleur pour les Repository ?

L'annotation @Autowired est utilisée dans les contrôleurs Spring pour injecter automatiquement une instance de Repository. Les Repository sont des composants utilisés dans Spring Data JPA pour interagir avec la base de données. L'utilisation de @Autowired permet de demander à Spring d'injecter automatiquement une instance du Repository requis dans le contrôleur, évitant ainsi la nécessité de créer manuellement l'instance du Repository.

2.5.2 A quoi sert l'annotation @ManyToOne dans l'entité skieur ?

Quant à l'annotation @ManyToOne, elle est utilisée pour spécifier une relation many-to-one (plusieurs vers un) entre deux entités dans JPA (Java Persistence API). Elle indique qu'un seul objet de l'entité annotée peut être associé à plusieurs instances d'une autre entité. Cette annotation est généralement utilisée du côté de l'entité qui possède la clé étrangère vers une autre entité.

Sur la même ligne, quel FetchType est utilisé et pourquoi, réessayer avec le FetchType LAZY et faites un getSkieur.

2.6 Connexion à la DB JPA avec DTO

Pourquoi dans ce cas, on retrouve un SkierDTO et pas de PaysDTO ?

Expliquez dans votre rapport à quoi servent les model, les repository, les dto, les services et les contrôleurs en vous basant sur le code donné.

Repo :

Les repository servent de pont entre votre application Spring et la base de données pour effectuer des opérations CRUD dans la base de données. Il utilise la convention de nommage de méthodes de Spring Data JPA pour générer automatiquement les requêtes SQL nécessaires en fonction des méthodes définies dans l'interface.

Service :

Les services permettent d'isoler les différentes parties du modèle pour avoir seulement la logique métier, facilitant ainsi la maintenance, la réutilisabilité et l'évolutivité de l'application. Les services agissent comme une couche intermédiaire qui traite les données, effectue des opérations métier et coordonne les interactions entre les différentes parties de l'application.

DTO :

Les DTO sont des objets utilisés pour transférer des données entre les différentes parties d'une application, souvent entre la couche de présentation (par exemple, les contrôleurs dans une application Spring MVC) et la couche de services ou la couche d'accès aux données.

Controller :

Le Controller permet de faire le pont entre la vue et les différents services de l'application.

2.7 Gestion des sessions

J'ai créé une nouvelle classe appelée ControllerSession et juste au-dessus de sa déclaration j'ai mis le suivant :

```
@RestController
```

```
@RequestMapping(path = "/user")
```

Ensuite j'ai fait 3 méthode. Login, qui a comme paramètres l'username et le mot-de-passe, logout et visites qui retourne le nombre de fois que la session a été visitée.

Login :

```
@PostMapping(path = "/login")
public ResponseEntity<String> login(@RequestParam String username,
    @RequestParam String pwd, HttpSession session) {
    if ((username.equals("test")) && (pwd.equals("test"))) {
        session.setAttribute("username", username);
        session.setAttribute("visites", 0);
        return ResponseEntity.ok("Login successful");
    } else {
        return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid credentials");
    }
}
```

Logout :

```
@PostMapping(path = "/logout")
public ResponseEntity<String> logout(HttpSession session) {
    if (session.getAttribute("username") == null) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("No one
is logged");
    } else {
        session.invalidate();
        return ResponseEntity.ok("Logout successful");
    }
}
```

Visites :

```
@GetMapping(path = "/visites")
public ResponseEntity<String> getVisites(HttpSession session){
    if(session.getAttribute("username") != null){
        int nbrSessions = (int) session.getAttribute("visites");
        nbrSessions += 1;
        session.setAttribute("visites", nbrSessions);
        return ResponseEntity.ok("Visites: " + nbrSessions);
    } else {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("No
session available");
    }
}
```

Il faudra aussi mettre le suivant dans le fichier « application.properties » :

```
spring.datasource.url=${DATABASE_URL:jdbc:mysql://localhost:3306/133ex5}
```

Et ceci dans le docker :

```
ENV DATABASE_URL=jdbc:mysql://host.docker.internal:3306/133ex5
```


Sans ces lignes on ne pourra pas lancer le container sur docker.

Important !!

À chaque fois qu'on veut utiliser la session dans une méthode il faudra qu'on mette en paramètre :

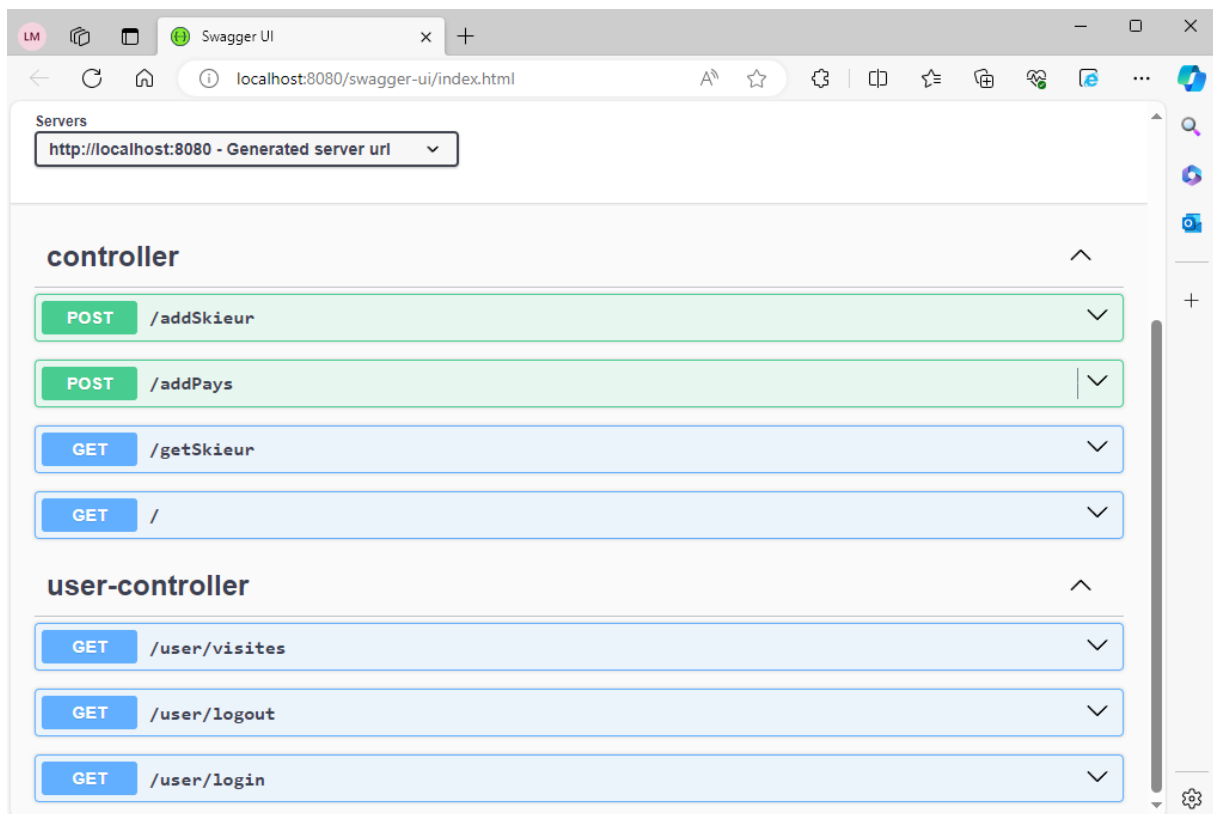
```
HttpSession session
```

2.8 Documentation API avec Swagger

Pour afficher la documentation de notre API, il nous suffit d'ajouter le code ci-dessous dans le fichier pom.xml.

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.3.0</version>
</dependency>
```

Grâce à cela on peut accéder à la page de la doc :



2.9 Hébergement

3 Analyse

3.1 Description du projet

Notre projet sera un magasin de Nerf où nous pourrons acheter différents Nerf. Côté client :

- Acheter des Nerfs
- Les ajouter au panier
- Ajouter un avis
- Voir le catalogue des Nerf

Côté vendeur :

- Gérer le stock des Nerfs
- Ajouter de nouveaux Nerf
- Virer des utilisateurs

3.2 Use case client et use case Rest

3.3 Activity Diagram d'un cas complet navigant dans les applications avec les explications

3.4 Sequence System global entre les applications

4 Conception

4.1 Class Diagram complet avec les explications de chaque application

5 Bases de données

5.1 Modèles WorkBench MySQL

5.2 Implémentation des applications <Le client Ap1> et <Le client Ap2>

5.3 Une descente de code client

6 Implémentation de l'application <API Gateway>

6.1 Une descente de code APIGateway

7 Implémentation des applications <API élève1> et <API élève2>

7.1 Une descente de code de l'API REST

8 Hébergement

9 Installation du projet complet avec les 5 applications

10 Tests de fonctionnement du projet

11 Auto-évaluations et conclusions

11.1 Auto-évaluation et conclusion de ...

11.2 Auto-évaluation et conclusion de ...

12 Conclusion

12.1 Ce que j'ai appris

12.2 Ce que j'ai aimé

12.3 Mes améliorations