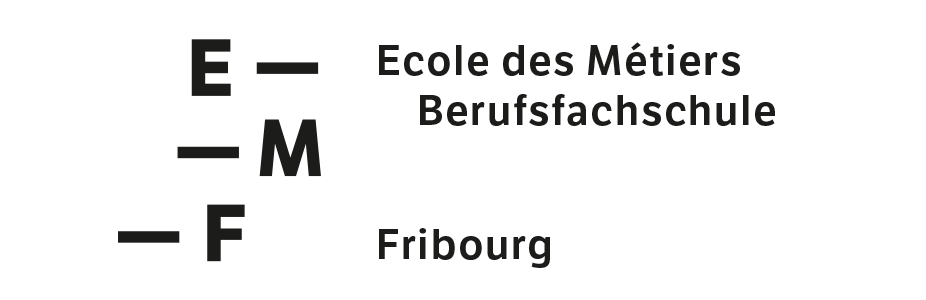
Module 133

Réaliser des applications web session-handling

Illan Angel

Rapport personnel

Module du 17.03.2025 – 15.04.2025

Table des matières

[Introduction 4](#_Toc194300917)

[Tests technologiques selon les exercices 4](#_Toc194300918)

[Installation et Hello World 4](#_Toc194300919)

[Implémentation 4](#_Toc194300920)

[Réponse aux questions 4](#_Toc194300921)

[Comment le projet est-il lancé et comment TOURNE-t-il 4](#_Toc194300922)

[C'est quoi le build et le run de Java ? 4](#_Toc194300923)

[Quel outil a-t-on utiliser pour build le projet ? 5](#_Toc194300924)

[Y a-t-il un serveur web ? 5](#_Toc194300925)

[Quelle version de java est utilisée ? 5](#_Toc194300926)

[S’IL y a un serveur web, quelle version utilise-t-il ? 5](#_Toc194300927)

[Conteneurisation 5](#_Toc194300928)

[Implémentation 5](#_Toc194300929)

[Mise en place 5](#_Toc194300930)

[Installation de maven 6](#_Toc194300931)

[Enventuels problème 7](#_Toc194300932)

[Réponse aux questions 7](#_Toc194300933)

[Pourquoi faire un container pour une application Java ? 7](#_Toc194300934)

[Y’a-t-il un server Web ? Ou se trouve-t-il 7](#_Toc194300935)

[A quoi faut-il faire attention (pensez aux version) 7](#_Toc194300936)

[Création d'un projet Spring Boot 8](#_Toc194300937)

[Classe Controller 8](#_Toc194300938)

[@RestController 8](#_Toc194300939)

[@GetMapping 9](#_Toc194300940)

[@PostMapping 9](#_Toc194300941)

[@PutMapping 9](#_Toc194300942)

[@RequestParam 9](#_Toc194300943)

[@RequestBody 9](#_Toc194300944)

[Résumé des annotations utilisées 10](#_Toc194300945)

[Test avec Postman 10](#_Toc194300946)

[GET 10](#_Toc194300947)

[POST 10](#_Toc194300948)

[PUT 11](#_Toc194300949)

[Connexion à la DB JDBC 11](#_Toc194300950)

[Connexion à la DB JPA 14](#_Toc194300951)

[Implémentation 14](#_Toc194300952)

[Questions 15](#_Toc194300953)

[Connexion à la DB JPA avec DTO 15](#_Toc194300954)

[Pourquoi un SkieurDTO et pas de PaysDTO ? 15](#_Toc194300955)

[Explication des différentes couches : 15](#_Toc194300956)

[Gestion des sessions 16](#_Toc194300957)

[Implémentation 16](#_Toc194300958)

[Documentation API avec Swagger 17](#_Toc194300959)

[Implémentation 17](#_Toc194300960)

[Hébergement 17](#_Toc194300961)

[Ex9Application.java 17](#_Toc194300962)

[com.example.ex9.controller : UserController.java 18](#_Toc194300963)

[com.example.ex9.models: User 18](#_Toc194300964)

[Applications.properties 19](#_Toc194300965)

[pom.xml : 19](#_Toc194300966)

[Auto-évaluations et conclusions 21](#_Toc194300967)

[Ce que j’ai appris 21](#_Toc194300968)

[Ce que j’ai aimé 22](#_Toc194300969)

[Ce que j’ai moins aimé 22](#_Toc194300970)

[Conclusion 22](#_Toc194300971)

# Introduction

# Tests technologiques selon les exercices

## Installation et Hello World

### Implémentation

Tous d’abord j’ai installé les extensions suivantes :

* Docker
* Extension Pack for Java
* Spring Boot Extension Pack

Puis j’ai démarré mon projet sur WSL : (>Connect to WSL)

Nous avons ensuite installé le JDK via le termine de commande et les commandes suivantes :

sudo apt install openjdk-17-jdk –y

Puis j’ai démarré le projet :

* Localhost:8080
* Localhost:8080/greeting

### Réponse aux questions

#### Comment le projet est-il lancé et comment TOURNE-t-il

Le projet est lancé en utilisant la commande suivante :

/usr/bin/env /usr/lib/jvm/java-17-openjdk-amd64/bin/java @/tmp/cp\_dxckg50z2exzf41xucecfdxog.argfile com.example.restservice.RestServiceApplication

Cela utilise l'exécution de Java via le JDK OpenJDK 17.

L'argument @/tmp/cp\_dxckg50z2exzf41xucecfdxog.argfile spécifie un fichier contenant les arguments nécessaires pour démarrer l'application. Le projet utilise Spring Boot, comme indiqué par la sortie, et démarre une application RestServiceApplication dans un contexte de serveur web intégré. Le serveur démarre sur le port 8080 avec le chemin contextuel par défaut /.

#### C'est quoi le build et le run de Java ?

* Build : Le processus de build en Java consiste à compiler les fichiers sources Java en bytecode exécutable par la JVM. Le build génère un fichier exécutable (généralement un fichier JAR ou WAR).
* Run : Exécuter l'application signifie lancer le fichier JAR ou WAR sur la JVM pour qu'il fonctionne, ce qui est ce qui est fait ici avec la commande indiquée.

#### Quel outil a-t-on utiliser pour build le projet ?

Avec Maven comme outil de build, voici quelques informations supplémentaires :

Maven est utilisé pour gérer les dépendances et le cycle de vie du projet, en automatisant les étapes de compilation, de test, et de création du fichier JAR ou WAR. Le fichier principal de configuration de Maven est pom.xml, où les dépendances, les plugins et les configurations du projet sont définis.

Commande Maven pour build : Pour créer le projet avec Maven, la commande suivante est utilisée :

mvn clean install

Cela nettoie le répertoire de build précédent (clean), puis construit le projet et installe le fichier JAR/WAR dans le répertoire target (install).

Commande Maven pour run : Pour exécuter l'application Spring Boot, Maven peut être utilisé avec le plugin spring-boot-maven-plugin :

mvn spring-boot:run

Cela démarre l'application sans avoir à exécuter directement le fichier JAR.

#### Y a-t-il un serveur web ?

Oui, un serveur web est utilisé dans ce projet. La sortie des logs indique que Tomcat est utilisé comme serveur web intégré :

Tomcat initialized with port 8080 (http)

Le serveur Tomcat est intégré dans l'application Spring Boot et est configuré pour écouter sur le port 8080.

#### Quelle version de java est utilisée ?

La version de Java utilisée dans ce projet est Java 17, comme indiqué dans les logs :

Starting RestServiceApplication using Java 17.0.14

#### S’IL y a un serveur web, quelle version utilise-t-il ?

Le serveur web utilisé est Apache Tomcat version 10.1.24, comme indiqué dans les logs :

Starting Servlet engine: [Apache Tomcat/10.1.24]

## Conteneurisation

### Implémentation

#### Mise en place

Pour commencer, on crée un fichier DockerFile : CTRL+SHIFT+P -> Docker: Add Docker Files To Workspace…

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

Ensuite on choisit : Java, complete/pom.xml, port 8080

Puis on doit avoir un DockerFiles ressemblant à :

# Utilisation de l'image openjdk

FROM openjdk:17-jdk-alpine

VOLUME /tmp

ARG JAVA\_OPTS

ENV JAVA\_OPTS=$JAVA\_OPTS

COPY complete/target/rest-service-complete-0.0.1-SNAPSHOT.jar complete.jar

EXPOSE 8080

ENTRYPOINT exec java $JAVA\_OPTS -jar complete.jar

Attention : vérifiez bien le chemin d’accès, dans la consigne c’est target/… mais le dossier target ce trouve dans complete/.

#### Installation de maven

Il faut installer maven sur notre WSL :

Sudo apt install maven

Et pour finir Build le projet avec maven pour avoir le .jar du DockerFiles :

cd /complete #projet à build

mvn clean package

On fait un clic droit sur DockerFile -> Build Images

Dans l'onglet Docker de VSCode : faite un clic droit sur l'image -> "Run"

Une image contenant texte, capture d’écran, Police, diagramme

Le contenu généré par l’IA peut être incorrect.

#### Enventuels problème

##### Connexion a Docker

Il faut faire attention à ce que vous soyez bien connecté à Docker :



##### Maven pas installé

Pour vérifier si Maven est installer :

mvn -v

### Réponse aux questions

#### Pourquoi faire un container pour une application Java ?

Utiliser un container Docker pour une application Java présente plusieurs avantages :

* Portabilité : Docker permet d'emballer l'application avec toutes ses dépendances (y compris la JVM, les bibliothèques et la configuration), ce qui garantit qu'elle fonctionnera de manière identique sur n'importe quel environnement (local, de développement, de test, de production).
* Isolation : Docker permet de séparer l'application Java des autres applications et services qui peuvent être exécutés sur la même machine, ce qui évite des conflits de dépendances ou de ressources.
* Facilité de déploiement : Une image Docker peut être facilement déployée sur n'importe quel système compatible Docker, réduisant ainsi la complexité de l'installation et de la configuration.
* Scalabilité : Les conteneurs Docker sont légers et peuvent être déployés rapidement et à grande échelle, ce qui est idéal pour les environnements où vous avez besoin de scalabilité.
* Automatisation du déploiement : En utilisant Docker, on peut automatiser le processus de déploiement, ce qui est crucial pour les équipes CI/CD (Intégration Continue/Déploiement Continu).

#### Y’a-t-il un server Web ? Ou se trouve-t-il

Oui, un serveur web est présent dans l'application, et il est intégré via Spring Boot.

* Serveur intégré : Spring Boot intègre un serveur web embarqué, généralement Tomcat, ce qui signifie que l'application ne nécessite pas un serveur externe comme Apache ou Nginx pour fonctionner.
* Localisation du serveur : Ce serveur est inclus dans le conteneur Docker via l'image de base OpenJDK. Dans ce cas, le serveur Tomcat démarre sur le port 8080 par défaut, comme mentionné dans le Dockerfile et les logs de l'application.

Le serveur Tomcat est lancé à partir de l'application Spring Boot et est exposé sur le port 8080, comme spécifié dans le Dockerfile :

EXPOSE 8080

#### A quoi faut-il faire attention (pensez aux version)

Il est essentiel de prêter attention aux versions des différents composants dans Docker et Java :

Version de l'image Docker OpenJDK : Il est important de s'assurer que l'image Docker utilisée pour le JDK correspond à la version de Java nécessaire pour l'application. Dans cet exercice, vous devez utiliser OpenJDK 17 pour qu'il soit compatible avec votre application Java.

Exemple de ligne dans le Dockerfile :

FROM openjdk:17-jdk-alpine

Version du fichier JAR : Le projet Java doit être compilé au préalable avant d'être copié dans l'image Docker. Vous devez vous assurer que le JAR que vous copiez dans l'image est le bon, par exemple :

COPY target/rest-service-complete-0.0.1-SNAPSHOT.jar complete.jar

Version de Tomcat : Même si Spring Boot gère Tomcat automatiquement, il est bon de savoir que Tomcat 10.1.24 est utilisé par défaut. Assurez-vous que la version de Tomcat utilisée est compatible avec votre application.

Variables d'environnement : Lorsque vous configurez des options Java (comme JAVA\_OPTS), vous devez veiller à ce que celles-ci soient adaptées aux besoins de votre application (par exemple, la mémoire allouée à la JVM).

## Création d'un projet Spring Boot

### Classe Controller

@RestController

public class Controller {

    // Handler pour GET

    @GetMapping("/getExample")

    public String getExample(@RequestParam(value = "name", defaultValue = "World") String name) {

        return String.format("Hello, %s!", name);

    }

    // Handler pour POST

    @PostMapping("/postExample")

    public String postExample(@RequestBody String body) {

        return "Received POST request with body: " + body;

    }

    // Handler pour PUT

    @PutMapping("/putExample")

    public String putExample(@RequestBody String body) {

        return "Received PUT request with body: " + body;

    }

}

### @RestController

Cette annotation indique que la classe est un contrôleur REST. Elle est une combinaison de @Controller et @ResponseBody, ce qui signifie que les méthodes de cette classe retournent directement les réponses HTTP (souvent au format JSON ou texte), sans avoir besoin de passer par des vues (comme avec un contrôleur MVC traditionnel).

@RestController

public class Controller { ... }

### @GetMapping

Cette annotation est utilisée pour marquer une méthode comme un handler pour les requêtes HTTP GET. Elle est une version spécialisée de l'annotation @RequestMapping qui est utilisée pour gérer les requêtes GET sur un chemin spécifique.

@GetMapping("/getExample")

public String getExample(@RequestParam(value = "name", defaultValue = "World") String name) { ... }

### @PostMapping

Cette annotation marque la méthode comme étant un handler pour les requêtes HTTP POST. Elle permet de gérer les requêtes POST envoyées vers un chemin spécifique.

@PostMapping("/postExample")

public String postExample(@RequestBody String body) { ... }

### @PutMapping

Cette annotation marque la méthode comme étant un handler pour les requêtes HTTP PUT. Elle permet de gérer les requêtes PUT envoyées vers un chemin spécifique.

@PutMapping("/putExample")

public String putExample(@RequestBody String body) { ... }

### @RequestParam

Cette annotation permet d'extraire les paramètres de la requête GET. Elle est utilisée pour accéder aux paramètres de la requête HTTP envoyée. Ici, elle permet de récupérer le paramètre name de l'URL. Si ce paramètre est absent, la valeur par défaut "World" est utilisée.

@RequestParam(value = "name", defaultValue = "World") String name

### @RequestBody

Cette annotation est utilisée pour extraire le corps de la requête HTTP dans les méthodes POST et PUT. Elle permet de lier directement le corps de la requête au paramètre de la méthode. Dans ton exemple, elle est utilisée pour récupérer le corps du POST ou PUT envoyé dans la requête HTTP.

@RequestBody String body

### Résumé des annotations utilisées

* @RestController : Indique que la classe est un contrôleur REST.
* @GetMapping : Gère les requêtes HTTP GET.
* @PostMapping : Gère les requêtes HTTP POST.
* @PutMapping : Gère les requêtes HTTP PUT.
* @RequestParam : Extrait un paramètre de la requête GET.
* @RequestBody : Extrait le corps de la requête HTTP (POST ou PUT).

Ces annotations permettent de facilement gérer les différentes méthodes HTTP et les paramètres associés dans un contrôleur Spring.

### Test avec Postman

#### GET

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

#### POST

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

#### PUT

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

## Connexion à la DB JDBC

Pour commencer on créer un contenaire MySql pour notre base de donnée :

#Création du répertoire sur la machine locale qui contiendra les données de MySQL

mkdir -p /opt/mysql

#Démarrage du container MySQL

docker run --name mysql -d -p 3308:3306 -e MYSQL\_ROOT\_HOST=% -e MYSQL\_ROOT\_PASSWORD=emf123 -v /opt/mysql:/var/lib/mysql mysql/mysql-server:8.0

Ensuite on récupère la DB et on la mets dans WorkBench et on ajoute dans notre contrôleur une fonctionnalité getPays pour récupérer les pays (grâce à la logique dans WrkDB :

Controller :

@GetMapping("/getPays")

public List<String> getPays() {

    return wrkDB.getPays();

}

WrkDB ; on ajoute les identifications de connexion :

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

final String user = "root";

final String pw =  "emf123";//

boolean result = false;

 Requête exécuter pour récupérer les pays (getPays) :

ps = dbConnexion.prepareStatement("SELECT Nom FROM t\_pays");

Ensuite on ajout des drivers JDBC dans le pom.xml avec un clic droit sur le pom.xml et « Add starter », puis MySQL :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Ceci a dû ajouter au fichier pom.xml le dependency suivante :

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

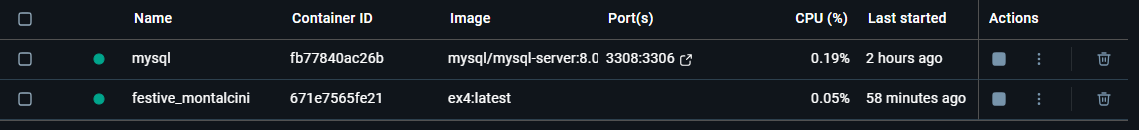
<scope>runtime</scope>

</dependency>

Dans le backend, vous pouvez maintenant vous connecter à la base de données en utilisant les credentials :

'mysql:host=localhost;port=3308;dbname=nomDB;charset=utf8', 'root', 'emf123'

Pour finir il faut créer une connexion docker en y ajoutant notre container MySQL et celui de notre exercice (création du dockerFile -> voir étape de l’exercice 4). Pour savoir le nom il faut aller dans docker :





Et exécuter les commandes suivantes dans un terminal :

docker network create asgard

docker network connect asgard mysql

docker network connect asgard nomContainerBackend

Pour finir on test avec Postman notre getPays :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

## Connexion à la DB JPA

### Implémentation

Pour la réalisation de cet exercice, j’ai dû modifier le fichier .properties et le pom.xml

.properties

spring.application.name=ex5

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.jdbc.batch\_size=20

spring.datasource.url=jdbc:mysql://localhost:3308/133ex5

spring.datasource.username=root

spring.datasource.password=emf123

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.hikari.maximum-pool-size=10

spring.datasource.hikari.minimum-idle=2

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=false

spring.jpa.properties.hibernate.cache.use\_query\_cache=false

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.orm.jdbc.bind=TRACE

### Questions

@Autowired : Cette annotation permet à Spring d’injecter automatiquement le bean SkieurRepository dans le contrôleur, ce qui simplifie l'accès aux méthodes de gestion des entités Skieur.

@ManyToOne : Cette annotation définit une relation entre l'entité Skieur et l'entité Pays. Un skieur appartient à un pays.

FetchType.EAGER : Cela signifie que les pays associés aux skieurs seront toujours récupérés en même temps que les skieurs. Si vous testez avec FetchType.LAZY, le pays ne sera récupéré que lorsqu'il est explicitement demandé.

## Connexion à la DB JPA avec DTO

### Pourquoi un SkieurDTO et pas de PaysDTO ?

SkieurDTO est nécessaire car nous exposons directement les informations des skieurs dans l'API, et ces informations peuvent être manipulées ou filtrées avant d'être envoyées au client.

PaysDTO n'est pas nécessaire, car le pays d'un skieur est uniquement une référence dans l'entité Skieur et nous n'avons pas besoin d'exposer l'entité Pays entière dans l'API. Nous envoyons simplement le nom du pays dans le SkieurDTO.

#### Explication des différentes couches :

Modèles : Ce sont des classes qui représentent les entités de la base de données, comme Skieur et Pays.

Repository : Ce sont des interfaces qui permettent de gérer les opérations CRUD (Create, Read, Update, Delete) sur les entités en utilisant Spring Data JPA.

DTO : Ce sont des objets qui sont utilisés pour transférer des données entre différentes couches de l'application. Les DTOs sont souvent utilisés pour ne transmettre que les données nécessaires et éviter l'exposition directe des entités.

Services : Les services contiennent la logique métier. Ils gèrent les interactions entre le contrôleur et les repositories, et la transformation des données entre entités et DTOs.

Contrôleurs : Les contrôleurs sont responsables de la gestion des requêtes HTTP. Ils exposent les API REST et appellent les services pour traiter les données.

## Gestion des sessions

### Implémentation

Pour cet exercice, j’ai tout d’abord recréé un exercice, j’y ai ensuite ajouté le « userController » donnée par l’exercice :

package com.example.ex6.controller;

import jakarta.servlet.http.HttpSession;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/user")

public class UserController {

@PostMapping("/login")

public ResponseEntity<String> login(HttpSession session, @RequestParam String username, @RequestParam String password) {

// Ici, vérifiez les identifiants de l'utilisateur

// Ceci est juste un exemple simple, ne stockez jamais le mot de passe en clair comme ceci dans la session

if ("user".equals(username) && "pass".equals(password)) {

session.setAttribute("username", username);

session.setAttribute("visites", 0);

return ResponseEntity.ok("Connecté");

} else {

return ResponseEntity.badRequest().body("Échec de la connexion");

@asdfas }

}

@GetMapping("/visites")

public ResponseEntity<String> visites(HttpSession session) {

if (session.getAttribute("username") != null) {

Integer visites = (Integer) session.getAttribute("visites");

visites++;

session.setAttribute("visites", visites);

return ResponseEntity.ok("Nombre de visites: " + visites);

} else {

return ResponseEntity.badRequest().body("Non connecté");

}

}

@PostMapping("/logout")

public ResponseEntity<String> logout(HttpSession session) {

session.invalidate();

return ResponseEntity.ok("Déconnecté");

}

}

J’ai dû modifier mon application.properties car je n’avais pas besoins d’une base de données, j’ai donc ajouté cette ligne de code pour que mon code ne recherche pas automatiquement une base de données :

spring.application.name=ex7

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

J’ai ensuite effectué les tests suivant avec postman :

* Login
* Logout
* getVisites

## Documentation API avec Swagger

### Implémentation

Dans cet exercice, nous avons mis en place l’outils de documentation swager-API, pour cela j’ai simplement copier-coller l’exercice précédent.

Puis,, j’ai simplement ajouté les dépendances nécessaires :

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.3.0</version>

</dependency

## Hébergement

Pour cet exercice, nous devions tester dockerHub, pour cela nous avons tous d’abord mis en place une structure basique avec un ControllerUser, un fichier « application.properties »correctement configurer ainsi qu’un pom.xml et toutes la gestions de la base de données avec un jpa et un repository, voici les différents fichiers :

### Ex9Application.java

package com.example.ex9;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Ex9Application {

public static void main(String[] args) {

SpringApplication.run(Ex9Application.class, args);

}

}

### com.example.ex9.controller : UserController.java

package com.example.ex9.controller;

import org.springframework.web.bind.annotation.\*;

import org.springframework.http.ResponseEntity;

@RestController

@RequestMapping("/user")

public class UserController {

@PostMapping("/login")

public ResponseEntity<String> login(@RequestParam String username, @RequestParam String password) {

// Logique de vérification de l'utilisateur

if ("user".equals(username) && "pass".equals(password)) {

return ResponseEntity.ok("Utilisateur connecté");

} else {

return ResponseEntity.badRequest().body("Échec de la connexion");

}

}

@PostMapping("/logout")

public ResponseEntity<String> logout() {

// Logique pour déconnecter l'utilisateur (ici une simulation)

return ResponseEntity.ok("Utilisateur déconnecté");

}

}

### com.example.ex9.models: User

package com.example.ex9.models;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String password;

// Getters and setters

}

com.example.ex9.repository: UserRepository :

package com.example.ex9.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.example.ex9.models.User;

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

Optional<User> findByUsername(String username);

}

### Applications.properties

spring.application.name=ex9

spring.datasource.url=${DATABASE\_URL:jdbc:mysql://localhost:3306/dbprojet}

spring.datasource.username=root

spring.datasource.password=emf123

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

### pom.xml :

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 <https://maven.apache.org/xsd/maven-4.0.0.xsd>">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.4.3</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>ex5</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>ex5</name>

<description>Demo project for Spring Boot</description>

<url/>

<licenses>

<license/>

</licenses>

<developers>

<developer/>

</developers>

<scm>

<connection/>

<developerConnection/>

<tag/>

<url/>

</scm>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<!-- Dépendance pour les tests, enlevée ou ignorée -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.3.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<annotationProcessorPaths>

<path>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</path>

</annotationProcessorPaths>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

<!-- Ajout du plugin pour ignorer les tests -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version>

<configuration>

<skipTests>true</skipTests>

</configuration>

</plugin>

</plugins>

</build>

</project

Une fois cette structure mis en place, j’ai créé un dockerFiles que j’ai directement modifié de la façon suivante :

FROM openjdk:17-jdk-alpine

VOLUME /tmp

ARG JAVA\_OPTS

ENV JAVA\_OPTS=$JAVA\_OPTS

COPY target/ex5-0.0.1-SNAPSHOT.jar ex9.jar

EXPOSE 8080

ENTRYPOINT exec java $JAVA\_OPTS -jar ex9.jar

ENV DATABASE\_URL=jdbc:mysql://host.docker.internal:3306/dbprojet

# 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 ex9.jar

J’ai ensuite réalisé les 3 étapes suivantes :

* Créez un compte sur dockerhub : <https://hub.docker.com>
* Créez un repository public sur votre compte
* Créez un token : <https://hub.docker.com/settings/security>

Une fois ces éléments terminés nous pouvons disposer de notre image à tout moment sur dockerHUB.

# Auto-évaluations et conclusions

Au terme de cette expérience, voici les trois points qui ressortent le plus clairement pour moi : ce que j'ai appris, ce que j'ai aimé et ce que j'ai moins aimé.

## Ce que j’ai appris

Globalement, l'expérience n'a pas été une véritable découverte pour moi, car j'avais déjà une bonne maîtrise de plusieurs des outils utilisés, notamment Spring Boot, Docker et WSL. Ces technologies étaient déjà familières, ce qui fait que j'ai pu les aborder rapidement sans avoir à les apprendre en profondeur. Cependant, j'ai tout de même appris quelques éléments intéressants. Parmi ceux-ci, Swagger était une nouveauté pour moi. Bien que j'en aie entendu parler, je n'avais jamais eu l'occasion de l'utiliser concrètement, et j'ai pu découvrir son fonctionnement et son utilité pour documenter les API de manière interactive et intuitive. Une autre notion que j'ai apprise concerne la gestion des sessions HTTP, que je n'avais pas encore abordée de cette manière. C'était donc un aspect précieux que j'ai pu explorer et qui m'a permis de mieux comprendre certains mécanismes sous-jacents au développement web.

## Ce que j’ai aimé

Ce que j'ai le plus apprécié dans cette expérience, c'est de pouvoir travailler sur le développement web. Le web a toujours été une partie que j'aime particulièrement dans la programmation, et l'opportunité de développer des applications en ligne m'a enthousiasmé. J'ai aussi particulièrement aimé l'utilisation de Docker. Ce sont des outils puissants et complets qui facilitent grandement le développement et la gestion d'environnements isolés. Docker permet de gagner un temps précieux, surtout lorsqu'il s'agit de gérer des configurations complexes et de garantir que les applications fonctionnent de manière identique sur toutes les machines. Ces deux éléments ont vraiment enrichi l'expérience et ont rendu le travail agréable et dynamique.

## Ce que j’ai moins aimé

En revanche, il y a eu quelques aspects qui m'ont moins satisfait. Tout d'abord, j'ai trouvé qu'il y avait très peu de théorie utile fournie pour aborder les exercices. L'absence d'explications théoriques détaillées sur les concepts derrière les exercices a souvent laissé place à une forme de flou, notamment lorsqu'on avait besoin de mieux comprendre certaines notions. En plus, la disponibilité du professeur n'était pas toujours optimale : lorsqu'on rencontrait des difficultés, il n'était pas toujours facile d'obtenir de l'aide rapidement, ce qui a rendu certaines parties du travail plus complexes. Enfin, j'ai également constaté que certaines données des exercices étaient fausses, ce qui a entraîné quelques erreurs dans la réalisation des tâches et a, par moments, conduit à une perte de temps inutile. Cela a quelque peu freiné l'avancée des travaux et créé des zones d'incertitude dans la gestion des exercices.

## Conclusion

En résumé, cette expérience a été enrichissante dans son ensemble, même si certains aspects ont pu être améliorés. J'en ressors avec des connaissances supplémentaires, en particulier sur Swagger et la gestion des sessions HTTP, et avec une appréciation renforcée pour les outils comme Docker. Toutefois, un meilleur équilibre entre théorie et pratique, ainsi qu'une assistance plus disponible, auraient rendu l'expérience encore plus constructive et fluide.