133 – Réaliser des applications web en session-handling

Rapport personnel

Version 1 du 18 mars 2025

Ribeiro Joël

|  |  |
| --- | --- |
|  | Module du 17.03.2025 au 15.04.2025 |

Table des matières

[1 Introduction 4](#_Toc193707817)

[2 Tests technologiques selon les exercices 4](#_Toc193707818)

[2.1 Installation et Hello World 4](#_Toc193707819)

[2.1.1 Prérequis 4](#_Toc193707820)

[2.1.2 Extensions installées sur VS Code 4](#_Toc193707821)

[2.1.3 Hello World 4](#_Toc193707822)

[2.1.4 Observation 5](#_Toc193707823)

[2.1.4.1 Observez la console pour comprendre comment le projet est lancé et comment il tourne ? 5](#_Toc193707824)

[2.1.4.2 C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ? 5](#_Toc193707825)

[2.1.4.3 Y a-t-il un serveur web ? 5](#_Toc193707826)

[2.1.4.4 Quelle version de java est utilisée ? 5](#_Toc193707827)

[2.1.4.5 S’il y a un serveur web, quelle version utilise-t-il ? 5](#_Toc193707828)

[2.2 Conteneurisation 5](#_Toc193707829)

[2.2.1 Création d’un DockerFile 5](#_Toc193707830)

[2.2.2 Création de l’image docker 6](#_Toc193707831)

[2.2.3 Lancement du container docker 7](#_Toc193707832)

[2.2.4 Observation 7](#_Toc193707833)

[2.2.4.1 Pourquoi faire un container pour une application Java ? 7](#_Toc193707834)

[2.2.4.2 Y a-t-il un serveur web ? Ou se trouve t'il ? 7](#_Toc193707835)

[2.2.4.3 A quoi faut-il faire attention (pensez au versions!) ? 8](#_Toc193707836)

[2.3 Création d’un projet Spring Boot 8](#_Toc193707837)

[2.3.1 Création du projet 8](#_Toc193707838)

[2.3.2 Création du contrôleur REST 8](#_Toc193707839)

[2.3.3 Lancement de l’application 9](#_Toc193707840)

[2.3.4 Observation 9](#_Toc193707841)

[2.3.4.1 Quelles sont les annotations utilisée (commencent par @) dans votre controller ? 9](#_Toc193707842)

[2.4 Connexion à la DB JDBC 9](#_Toc193707843)

[2.4.1 Container MySQL 9](#_Toc193707844)

[2.4.2 Connexion JDBC 9](#_Toc193707845)

[2.4.3 Test avec un Run en local dans VSC 10](#_Toc193707846)

[2.4.4 Test avec un container pour l’application et la MySQL 10](#_Toc193707847)

[2.5 Connexion à la DB JPA 11](#_Toc193707848)

[2.5.1 Création du projet 11](#_Toc193707849)

[2.5.2 Créer le fichier application.properties 11](#_Toc193707850)

[2.5.3 Créer le @Entitiy Model 11](#_Toc193707851)

[2.5.4 Créer le Repository 12](#_Toc193707852)

[2.5.5 Créer le Controller 12](#_Toc193707853)

[2.5.6 Tester l’application 13](#_Toc193707854)

[2.5.7 Ajouter les pays 14](#_Toc193707855)

[2.5.8 Observation 16](#_Toc193707856)

[2.5.8.1 À quoi sert l'annotation @Autowired dans vos controlleur pour les Repository ? 16](#_Toc193707857)

[2.5.8.2 A quoi sert l'annotation @ManyToOne dans l'entité skieur ? 16](#_Toc193707858)

[2.5.8.3 Sur la même ligne, quel FetchType est utilisé et pourquoi, rééssayer avec le FetchType LAZY et faites un getSkieur. 17](#_Toc193707859)

[2.6 Connexion à la DB JPA avec DTO 17](#_Toc193707860)

[2.6.1 Utilisation de DTO 18](#_Toc193707861)

[2.6.2 Utilisation de Service 19](#_Toc193707862)

[2.6.3 Lancement de l’application 20](#_Toc193707863)

[2.6.4 Observations 21](#_Toc193707864)

[2.6.4.1 Pourquoi dans ce cas, on retrouve un SkierDTO et pas de PaysDTO ? 21](#_Toc193707865)

[2.6.4.2 Expliquez dans votre rapport à quoi servent les model, les repository, les dto, les services et les controlleurs en vous basant sur le code donné. 22](#_Toc193707866)

[2.7 Gestion des sessions 22](#_Toc193707867)

[2.7.1 Réalisation 22](#_Toc193707868)

[2.7.2 Observation 23](#_Toc193707869)

[2.7.2.1 Décrivez dans votre rapport l'utilisation de HTTPSession. Illustrez le fonctionnement des sessions avec des printscreen de vos appels Postman à ces 3 méthodes. 23](#_Toc193707870)

[2.8 Documentation API avec Swagger 25](#_Toc193707871)

[2.8.1 Documenter avec Swagger 25](#_Toc193707872)

[2.8.2 Tester sa documentation 25](#_Toc193707873)

[2.9 Hébergement java 25](#_Toc193707874)

[2.9.1 Utilisez des variables qui seront remplacées dans le DockerFile 25](#_Toc193707875)

[2.9.2 Envoyer son image docker su dockerhub 26](#_Toc193707876)

[3 Conclusion 27](#_Toc193707877)

[3.1 Ce que j’ai appris, ce qui est important 27](#_Toc193707878)

[3.2 Ce que j’ai apprécié 28](#_Toc193707879)

[3.3 Mes propositions d’amélioration 28](#_Toc193707880)

[3.4 Conclusion personnelle 28](#_Toc193707881)

[3.4.1 Points forts 28](#_Toc193707882)

[3.4.2 Points faibles 28](#_Toc193707883)

# Introduction

# Tests technologiques selon les exercices

## Installation et Hello World

### Prérequis

Il faut installer WSL via powershell puis le docker-desktop à partir des liens donnés dans l’exercice :

<https://learn.microsoft.com/fr-fr/windows/wsl/install>

<https://docs.docker.com/desktop/wsl/#download>

### Extensions installées sur VS Code

Voici les extensions installées sur VS Code pour le bon fonctionnement de notre exercice :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

### Hello World

Pour créer une première application java, nous allons utiliser l’exemple Spring.

Il faut donc cloner le projet disponible sur GitHub : <https://spring.io/guides/gs/guides-with-vscode>

Ensuite depuis le terminal sur le WSL il faut faire les commandes suivantes :

sudo apt install openjdk-17-jdk -y

echo 'export JAVA\_HOME=/usr/lib/jvm/java-17-openjdk-amd64' >> ~/.bashrc

source ~/.bashrc

echo $JAVA\_HOME

S’il y a une erreur lors de l’exécution de la première ligne il faut faire la commande suivante, cela peut régler le problème

sudo apt update

Cela a permis d’installer le Jdk 17, une fois cela terminé il faut lancer le programme. Si tout se passe bien 2 pages web devrait être accessible à ces endroits :

* [localhost:8080](http://localhost:8080/)
* [localhost:8080/greeting](http://localhost:8080/greeting)

### Observation

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

L'application Spring Boot démarre avec Java 17.

Elle utilise Tomcat 10.1.24 comme serveur web embarqué.

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

Elle écoute sur le port 8080 et utilise le profil "default".

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

Oui, un serveur Tomcat est intégré à l'application.

#### Quelle version de java est utilisée ?

Java 17.0.14

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

Tomcat 10.1.24.

## Conteneurisation

### Création d’un DockerFile

On commence par créer le fichier « DockerFil » puis on fait CTRL + SHIFT + P et on sélectionne « Docker : Add Docker Files to Workspace… » :  
Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

On choisit ensuite « Java », le port 8080 ainsi que le fichier pom.xml :

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

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

Voici l’interieur du fichier Docker configurer :

# 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

### Création de l’image docker

On execute ces commandes avant de mettre en place l’image :

cd complete

sudo apt install maven

mvn clean package

On fait un clic droit sur le fichier « DockerFile » puis on sélectionne « Build image » :

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

Le contenu généré par l’IA peut être incorrect.

### Lancement du container docker

Dans l'onglet Docker de VSCode : faite un clic droit sur l'image puis « Run » :

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

Le contenu généré par l’IA peut être incorrect.

### Observation

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

Conteneuriser une application Java permet de garantir la portabilité et la cohérence de l’environnement d’exécution. Avec Docker, l’application Java peut s’exécuter sur n’importe quel système disposant de Docker, sans se soucier des configurations spécifiques du système hôte (versions de Java, dépendances, variables d’environnement, etc.). De plus, cela facilite le déploiement, l’évolutivité et la gestion des versions.

#### Y a-t-il un serveur web ? Ou se trouve t'il ?

Oui, l’application Spring Boot embarque un serveur web, souvent **Tomcat** par défaut. Celui-ci est inclus dans le JAR généré lors du build et s’exécute automatiquement lorsque l’application est lancée. Le serveur est accessible via le **port 8080**, qui est exposé dans le Dockerfile (EXPOSE 8080).

#### A quoi faut-il faire attention (pensez au versions!) ?

* Il faut s’assurer que la version de l’image OpenJDK utilisée dans Docker est compatible avec la version de Java utilisée pour compiler l’application. Par exemple, si l’application a été compilée avec Java 17, il faut utiliser une image OpenJDK 17 (FROM openjdk:17-jdk-alpine).
* Vérifier que les dépendances utilisées dans pom.xml sont compatibles avec cette version de Java.
* Faire attention aux ports : s’assurer qu’aucun autre service n’utilise déjà le port 8080 sur la machine hôte.
* S’assurer que le chemin du fichier JAR est correct dans le Dockerfile, et que le projet a bien été buildé avant (mvn package).
* Tester le conteneur sur différents environnements pour éviter des erreurs dues aux différences entre les configurations locales et celles du conteneur.

## Création d’un projet Spring Boot

### Création du projet

On commence par faire un CTRL + SHIFT + P puis on écrit « Spring Initializr : Create a Maven Project… »

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

Le contenu généré par l’IA peut être incorrect.

On choisit ces deux extensions :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

### Création du contrôleur REST

On crée un package ctrl puis on crée un controleur puis on y définit les méthodes GET, POST et PUT :

@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;

    }

}

### Lancement de l’application

Après avoir lancé l’application, on se rend sur le localhost:8080/getExample :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

### Observation

#### Quelles sont les annotations utilisée (commencent par @) dans votre controller ?

Les annotations utilisées sont :

@RestController, @GetMapping, @PostMapping, @PutMapping.

## Connexion à la DB JDBC

### Container MySQL

Pour créer la DB, on commence par démarrer un nouveau container qui contiendra le serveur MySQL avec ces commandes depuis le terminal du WSL :

#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

### Connexion JDBC

Sur la base de l'exercice 3, on ajoute une fonctionnalité getPays à notre contrôleur.

Cette fonctionnalité doit aller rechercher dans la DB (à disposition dans le OneNote) la liste des pays.

On utilise le WrkDB à disposition dans le OneNote pour la connexion jdbc.

package ex4.ex4.ctrl;

import ex4.ex4.wrk.WrkDB;

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

import java.util.List;

@RestController

public class Controller {

    private final WrkDB wrkDB = new WrkDB("3308", "bd\_kitzbuehl"); // Mets le bon nom de ta DB

    @GetMapping("/getPays")

    public List<String> getPays() {

        return wrkDB.getPays(); // Appelle WrkDB pour récupérer la liste des pays

    }

    // 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;

    }

}

On ajoute ensuite les drivers JDBC dans le pom.xml : faire un clic droit sur pom.xml et "Add starter". Choisissez d'ajouter le driver MySQL :

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

Le contenu généré par l’IA peut être incorrect.

Ceci a dû être ajouté à votre fichier pom.xml :

<dependency>

<groupId>com.mysql</groupId>

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

<scope>runtime</scope>

</dependency>

### Test avec un Run en local dans VSC

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'

### Test avec un container pour l’application et la MySQL

Afin d'être capable de se connecter depuis le container backend sur la base de données MySQL contenue dans le container MYSQL, il faut créer un nouveau réseau Docker et connecter ces deux containeurs sur ce réseau. Voici les commandes à exécuter sur le CLI de WSL :

docker network create asgard

docker network connect asgard mysql

docker network connect asgard nomContainerBackend

Puis dans le backend, on change localhost en host.docker.internal pour l'accès à la DB :

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

## Connexion à la DB JPA

### Création du projet

On commence par créer notre projet avec les librairies suivantes :

* + Spring Web : comme la description l’indique, permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints.
  + Lombok : c’est une librarie pour optimiser certaines classes et gagner du temps
  + Spring Data JPA : c'est la librairie pour utiliser la persistance (JPA)
  + Mysql Driver SQL: driver pour Mysql

### Créer le fichier application.properties

On créer le fichier application.properties dans le « src/main/resources/application.properties » :

spring.application.name=ex5

spring.jpa.hibernate.ddl-auto=update

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.jpa.show-sql=true

### Créer le @Entitiy Model

On doit créer le model d'entité comme dans main/java/com/example/ex5/model/Skieur.java :

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table (name = "t\_skieur")

public class Skieur {

    @Id

    @Column(name = "PK\_Skieur", length = 50)

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Integer id;

    @Column(name = "Nom", length = 50)

    private String name;

    @Column(name = "Position", length = 50)

    private Integer position;

    public Integer getId() {

        return id;

    }

    public void setId(Integer id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public Integer getPosition() {

        return position;

    }

    public void setPosition(Integer position) {

        this.position = position;

    }

}

### Créer le Repository

On crée le repository pour gérer les opérations CRUD sur l’entité Skieur :

import org.springframework.data.repository.CrudRepository;

import com.example.ex5.model.Skieur;

// This will be AUTO IMPLEMENTED by Spring into a Bean called SkieurRepository

// CRUD refers Create, Read, Update, Delete

public interface SkieurRepository extends CrudRepository<Skieur, Integer> {

}

### Créer le Controller

On crée le contrôleur qui permet d’interagir avec la base de données via des requêtes HTTP :

package com.example.ex5.controller;

import com.example.ex5.model.Skieur;

import com.example.ex5.model.SkieurRepository;

import java.util.ArrayList;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

@RestController

public class Controller {

    @Autowired // This means to get the bean called skieurRepository which is auto-generated by Spring, we will use it to handle the data

    private SkieurRepository skieurRepository;

    @PostMapping(path="/addSkieur")

    public @ResponseBody String addNewSkieur (@RequestParam String name, @RequestParam Integer position) {

        // @ResponseBody means the returned String is the response, not a view name

        // @RequestParam means it is a parameter from the GET or POST request

        Skieur newSkieur = new Skieur();

        newSkieur.setName(name);

        newSkieur.setPosition(position);

        skieurRepository.save(newSkieur);

        return "saved";

    }

    @GetMapping(path="/getSkieur")

    public @ResponseBody Iterable<Skieur> getAllUsers() {

        // This returns a JSON or XML with the users

        return skieurRepository.findAll();

    }

}

### Tester l’application

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

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

### Ajouter les pays

Pour ajouter une nouvelle entité pays, on la créée et on la liée à Skieur avec une relation @ManyToOne :

@ManyToOne(fetch = FetchType.EAGER)

    @JoinColumn(name = "FK\_pays")

    private Pays pays;

Puis on insère le model Pays.java et PaysRepository.java :

package ex5.ex5.model;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table (name = "t\_pays")

public class Pays {

    @Id

    @Column(name = "PK\_Pays", length = 50)

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Integer id;

    @Column(name = "Nom", length = 50)

    private String name;

    public Integer getId() {

        return id;

    }

    public void setId(Integer id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}

package ex5.ex5.model;

import org.springframework.data.repository.CrudRepository;

import ex5.ex5.model.Pays;

// This will be AUTO IMPLEMENTED by Spring into a Bean called PaysRepository

// CRUD refers Create, Read, Update, Delete

public interface PaysRepository extends CrudRepository<Pays, Integer> {

}

Puis on ajoute les méthodes dans le contrôleur pour ajouter et afficher les pays :

package ex5.ex5.ctrl;

import ex5.ex5.model.Pays;

import ex5.ex5.model.PaysRepository;

import ex5.ex5.model.Skieur;

import ex5.ex5.model.SkieurRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

@RestController

public class Controller {

    @Autowired // This means to get the bean called skieurRepository which is auto-generated by Spring, we will use it to handle the data

    private SkieurRepository skieurRepository;

    @Autowired

    private PaysRepository paysRepository;

    @PostMapping(path="/addSkieur")

    public @ResponseBody String addNewSkieur (@RequestParam String name, @RequestParam Integer position, @RequestParam Pays pays) {

        // @ResponseBody means the returned String is the response, not a view name

        // @RequestParam means it is a parameter from the GET or POST request

        Skieur newSkieur = new Skieur();

        newSkieur.setName(name);

        newSkieur.setPosition(position);

        newSkieur.setPays(pays);

        skieurRepository.save(newSkieur);

        return "saved";

    }

    @GetMapping(path="/getSkieur")

    public @ResponseBody Iterable<Skieur> getAllUsers() {

        // This returns a JSON or XML with the users

        return skieurRepository.findAll();

    }

    @PostMapping(path="/addPays")

    public @ResponseBody String addNewPays (@RequestParam String name) {

        // @ResponseBody means the returned String is the response, not a view name

        // @RequestParam means it is a parameter from the GET or POST request

        Pays newPays = new Pays();

        newPays.setName(name);

        paysRepository.save(newPays);

        return "saved";

    }

    @GetMapping(path="/getPays")

    public @ResponseBody Iterable<Pays> getAllPays() {

        // This returns a JSON or XML with the users

        return paysRepository.findAll();

    }

}

Dans Postman :

Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

### Observation

#### À quoi sert l'annotation @Autowired dans vos controlleur pour les Repository ?

L'annotation @Autowired est utilisée dans les contrôleurs pour injecter automatiquement les dépendances de Spring, notamment les Repository.

Dans notre cas, elle permet à Spring d’initialiser automatiquement les instances de SkieurRepository et PaysRepository afin qu’elles puissent être utilisées sans avoir besoin de les instancier manuellement avec new.

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

L'annotation @ManyToOne est utilisée pour établir une relation "Plusieurs à Un" entre Skieur et Pays.

Cela signifie qu'un pays peut être associé à plusieurs skieurs, mais chaque skieur n'a qu'un seul pays.

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

Actuellement, le FetchType utilisé est EAGER, Spring JPA charge immédiatement l'entité Pays lorsque l'on récupère un Skieur donc cela permet d'éviter les problèmes de chargement différé lorsque l'on souhaite afficher immédiatement les détails du pays d’un skieur.

## Connexion à la DB JPA avec DTO

L’entité Skieur est la même que dans l’exercice 5, mais avec « FetchType.LAZY » pour optimiser le chargement des données.

package com.example.ex6.model;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.FetchType;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

import jakarta.persistence.JoinColumn;

import jakarta.persistence.ManyToOne;

import jakarta.persistence.Table;

@Entity

@Table (name = "t\_skieur")

public class Skieur {

    @Id

    @Column(name = "PK\_Skieur", length = 50)

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Integer id;

    @Column(name = "Nom", length = 50)

    private String name;

    @Column(name = "Position", length = 50)

    private Integer position;

    @ManyToOne(fetch = FetchType.LAZY)

    @JoinColumn(name = "fk\_pays")

    private Pays pays;

    public Integer getId() {

        return id;

    }

    public void setId(Integer id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public Integer getPosition() {

        return position;

    }

    public void setPosition(Integer position) {

        this.position = position;

    }

    public Pays getPays() {

        return pays;

    }

    public void setPays(Pays pays) {

        this.pays = pays;

    }

}

### Utilisation de DTO

Le DTO est utilisé pour exposer seulement les données nécessaires et éviter les problèmes de Lazy Loading.

package com.example.ex6.dto;

public class SkieurDTO {

    private Integer id;

    private String name;

    private Integer position;

    private String paysNom; // Nom du pays pour simplifier, vous pouvez inclure d'autres propriétés si nécessaire

    // Constructeurs, getters et setters

    public SkieurDTO() {}

    public SkieurDTO(Integer id, String name, Integer position, String paysNom) {

        this.id = id;

        this.name = name;

        this.position = position;

        this.paysNom = paysNom;

    }

    public Integer getId() {

        return id;

    }

    public void setId(Integer id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public Integer getPosition() {

        return position;

    }

    public void setPosition(Integer position) {

        this.position = position;

    }

    public String getPaysNom() {

        return paysNom;

    }

    public void setPaysNom(String paysNom) {

        this.paysNom = paysNom;

    }

}

### Utilisation de Service

Le Service gère la logique métier et transforme un Skieur en SkieurDTO avant de l’envoyer au contrôleur.

package com.example.ex6.service;

import com.example.ex6.dto.SkieurDTO;

import com.example.ex6.model.Pays;

import com.example.ex6.model.Skieur;

import com.example.ex6.repository.PaysRepository;

import com.example.ex6.repository.SkieurRepository;

import jakarta.transaction.Transactional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

@Service

public class SkieurService {

    private final SkieurRepository skieurRepository;

    private final PaysRepository paysRepository;

    @Autowired

    public SkieurService(SkieurRepository skieurRepository, PaysRepository paysRepository) {

        this.skieurRepository = skieurRepository;

        this.paysRepository = paysRepository;

    }

    public Iterable<SkieurDTO> findAllSkieurs() {

        Iterable<Skieur> skieurs = skieurRepository.findAll();

        List<SkieurDTO> skieurDTOs = new ArrayList<>();

        for (Skieur skieur : skieurs) {

            SkieurDTO skieurDTO = new SkieurDTO(

                    skieur.getId(),

                    skieur.getName(),

                    skieur.getPosition(),

                    skieur.getPays() != null ? skieur.getPays().getNom() : null);

            skieurDTOs.add(skieurDTO);

        }

        return skieurDTOs;

    }

    @Transactional

    public String addNewSkieur(String name, Integer position, Integer paysId) {

        Pays pays = paysRepository.findById(paysId).orElse(null);

        if (pays == null) {

            return "Pays not found";

        }

        Skieur newSkieur = new Skieur();

        newSkieur.setName(name);

        newSkieur.setPosition(position);

        newSkieur.setPays(pays);

        skieurRepository.save(newSkieur);

        return "Saved";

    }

}

### Lancement de l’application

Voici un exemple d’utilisation de l’application :

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

Le contenu généré par l’IA peut être incorrect.

### Observations

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

L’entité Skieur contient une relation @ManyToOne vers Pays. En utilisant un DTO, on peut éviter que JPA charge tout l’objet Pays en EAGER, ce qui ralentirait la requête. Pour Pays, il n’est pas nécessaire d’avoir un DTO, car la table n’a pas de relation complexe.

Pays est une entité simple qui ne pose pas de problème en Lazy Loading, alors que Skieur peut provoquer des problèmes de performance si mal géré.

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

Model : Représente les tables de la base de données. Chaque entité JPA est mappée à une table SQL. Exemple : Skier.java.

Repository : Interface qui étend JpaRepository, permet d'accéder aux données sans écrire de SQL. Exemple : SkierRepository.

DTO : Objet simplifié servant à transporter les données entre le service et l’extérieur (API), sans exposer l’entité directement.

Service : Contient la logique métier et fait le lien entre Repository et Controller. Il transforme aussi les entités en DTO.

Controller : Expose les endpoints de l’API (via @RestController), appelle le service, et retourne les résultats (souvent sous forme de DTO).

## Gestion des sessions

### Réalisation

On ajoute un nouveau contrôleur qui va permettre la gestion des sessions, avec un login, un logout et la possibilité de voir combien de personne visite le site actuellement :

package com.example.demo.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(@RequestParam String username,

@RequestParam String password,

HttpSession session) {

// Ici on ne vérifie pas le mot de passe, c’est une simulation

session.setAttribute("username", username);

session.setAttribute("visites", 0);

return ResponseEntity.ok("Session créée pour l'utilisateur : " + username);

}

@PostMapping("/logout")

public ResponseEntity<String> logout(HttpSession session) {

session.invalidate();

return ResponseEntity.ok("Session supprimée !");

}

@GetMapping("/visites")

public ResponseEntity<String> visites(HttpSession session) {

Object visitesAttr = session.getAttribute("visites");

if (visitesAttr == null) {

return ResponseEntity.status(401).body("Aucune session active. Veuillez vous connecter.");

}

int visites = (int) visitesAttr;

visites++;

session.setAttribute("visites", visites);

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

}

}

### Observation

#### Décrivez dans votre rapport l'utilisation de HTTPSession. Illustrez le fonctionnement des sessions avec des printscreen de vos appels Postman à ces 3 méthodes.

Pour tester le login sur postman avec <http://localhost:8080/user/login?username=user&password=pass> :

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

Le contenu généré par l’IA peut être incorrect.

Pour regarder le nombre de visite actuelle sur le site avec <http://localhost:8080/user/visites> :

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

Le contenu généré par l’IA peut être incorrect.

Pour tester le logout sur postman avec <http://localhost:8080/user/logout> :

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

Le contenu généré par l’IA peut être incorrect.

## Documentation API avec Swagger

### Documenter avec Swagger

On ajoute cette « dependency » dans le pom.xml :

 <dependency>

      <groupId>org.springdoc</groupId>

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

      <version>2.3.0</version>

   </dependency>

### Tester sa documentation

Pour tester la documentation, on se rend sur [Swagger UI](http://localhost:8080/swagger-ui/index.html) , ce qui nous permet de voir les différentes méthodes et leur type :

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

## Hébergement java

### Utilisez des variables qui seront remplacées dans le DockerFile

Dans l’application.properties, on ajoute cette ligne :

spring.datasource.url=${DATABASE\_URL:jdbc:mysql://localhost:3308/133ex6}

Ainsi que cette ligne dans le Dockerfile :

ENV DATABASE\_URL=jdbc:mysql://host.docker.internal:3308/133ex6

### Envoyer son image docker su dockerhub

On crée un repository public via <https://hub.docker.com> :

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

Le contenu généré par l’IA peut être incorrect.

Puis on crée un token via <https://hub.docker.com/settings/security> :



Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Pour continuer, on se rend sur VSCode et on insère la commande suivante :

docker login -u ribeiroj

On insère ensuite le mot de passe et lorsqu’il est validé, cela s’affiche comme ça :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Avec ces lignes de commandes suivantes, je tag et push l’image sur dockerhub :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

# Conclusion

## Ce que j’ai appris, ce qui est important

Ce projet m’a permis de découvrir et d’expérimenter plusieurs aspects essentiels du développement d’applications web modernes avec Java et Spring Boot. J’ai appris à créer des API REST avec des annotations comme @RestController, @GetMapping, @PostMapping, différentes manières de connecter une application à une base de données via JDBC, via JPA et DTO. Gérer les sessions utilisateur avec HttpSession et suivre les visites. Documenter une API avec Swagger. Héberger un projet en ligne via DockerHub.

## Ce que j’ai apprécié

J’ai apprécié les exercices qui m’ont permis de faire étapes par étapes, on faisait également les exercices à notre vitesse.

## Mes propositions d’amélioration

Commencer par un modèle d’organisation, pour éviter de tous mélanger (controllers, services, models).

Apprendre plus sur Docker car on n’a presque pas utilisé depuis le début de notre formation.

## Conclusion personnelle

### Points forts

Les exercices étaient simple et facile, j’ai pas eu de difficulté à les faire.

### Points faibles

Beaucoup de difficulté à mettre en place toute l’infrastructure avant de commencer les exercices.