

133 - Réaliser une application web en Session-Handling

Rapport Personnel

Date de création : 21.03.2024
Version 1 du 05.05.2024

Sergio Yaguachi

Module du 21.03.2024
au 03.05.2024



<h2>Table de matières</h2>

1	Introduction	3
2	Conclusion	Erreur ! Signet non défini.
2.1	Ce que j'ai appris.....	Erreur ! Signet non défini.
2.2	Ce que j'ai aimé.....	Erreur ! Signet non défini.
2.3	Mes axes d'amélioration.....	Erreur ! Signet non défini.

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.

Si dans un exercice on a un problème de droit :

```
chmod +x mvnw
```

2 Introduction et contexte du project

Notre projet est un quizz qui comporte des questions et 4 réponses par question dont une correcte. Le joueur doit se logger pour pouvoir jouer et en répondant juste au question son score augmente. A la fin du quizz il peut voir son score et un historique des autres scores qu'il a pu faire auparavant. Il y a aussi un mode admin pour ajouter ou supprimer des questions/réponses.

3 Tests technologiques selon les exercices

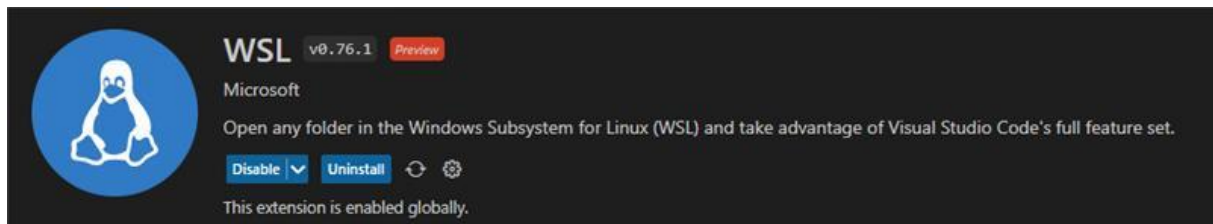
3.1 Installation et Hello World

Pour réaliser l'installation de Hello World :

- 1) Il faut installer WSL / Visual Studio Code / Docker et Ubuntu
- 2) Installer WSL sur Windows powershell (il faudra créer un compte)

wsl --install

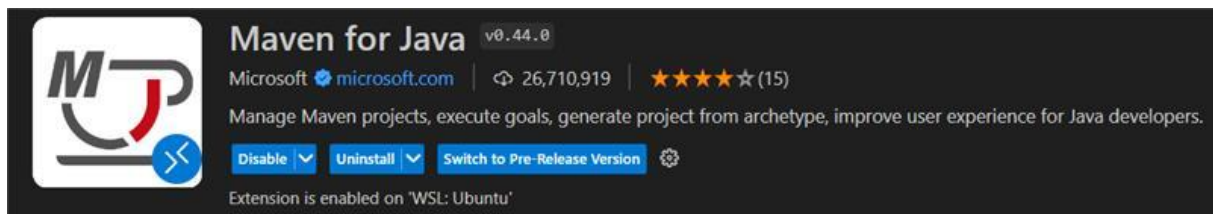
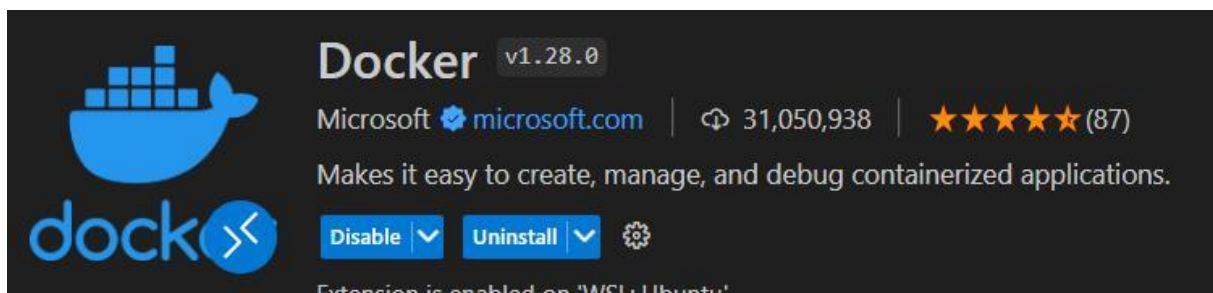
- 3) Installer l'extension WSL



- 4) Se connecter avec VScode en Ubuntu

Connect to WSL using Distro...

- 5) Installer les extensions nécessaires





6) Créer un projet dans l'invite de commande Ubuntu

git clone <https://github.com/spring-guides/gs-rest-service.git>

7) Ouvrir le projet sur VSCode et installer jdk 17 sur le projet

sudo apt update

sudo apt install openjdk-17-jdk

8) Lancer le projet (si cela ne fonctionne pas, chercher la notification use Maven)

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

Le projet a été lancé en utilisant Java 17.0.6 avec le PID 10330. Le répertoire de travail est /home/sergio42/gs-rest-service.

Aucun profil actif n'a pas été défini, donc le profil par défaut a été utilisé.

Tomcat a été initialisé avec le port 8080 (http).

Le service Tomcat a été démarré.

Le moteur de servlet Apache Tomcat/10.1.16 a été démarré.

Le WebApplicationContext intégré de Spring a été initialisé.

L'initialisation du Root WebApplicationContext a été achevée en 485 ms.

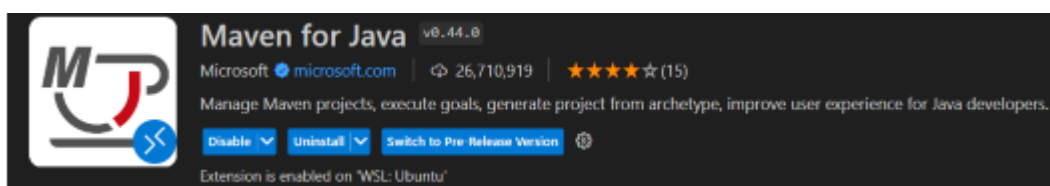
Tomcat a démarré sur le port 8080 (http) avec le chemin de contexte "".

L'application RestServiceApplication a été démarrée en 0.914 secondes (le processus fonctionne depuis 1.103 secondes).

Le DispatcherServlet 'dispatcherServlet' de Spring a été initialisé.

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

MAVEN. L'extension suivant.



- Y a-t-il un serveur web ?

Oui, c'est un serveur Web

- Quelle version de java est utilisée ?

OpenJDK 17

- Si il y a un serveur web, quelle version utilise-t-il ?

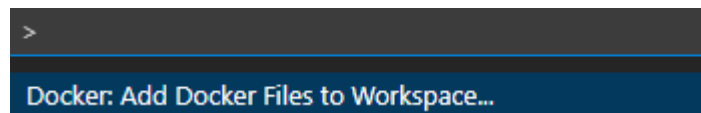
Tomcat 10.1.16

3.2 Conteneurisation

Il faut installer maven avec cette commande :

```
sudo apt install maven
```

Ensuite il faut créer le fichier Docker. Nous allons faire Ctrl + shift + P.



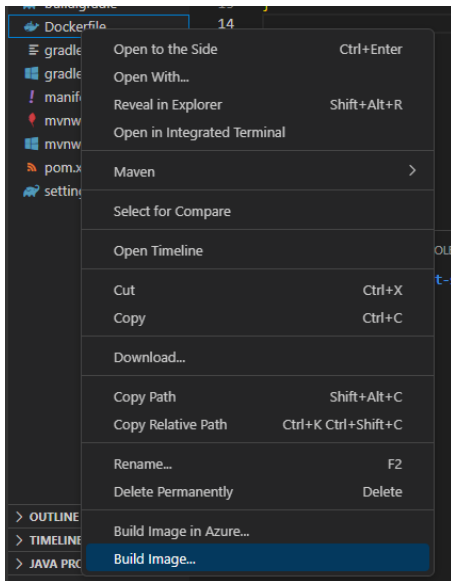
Choisissez ensuite "Java", le port 8080 et le fichier pom.xml.

La version de JDK est 17.

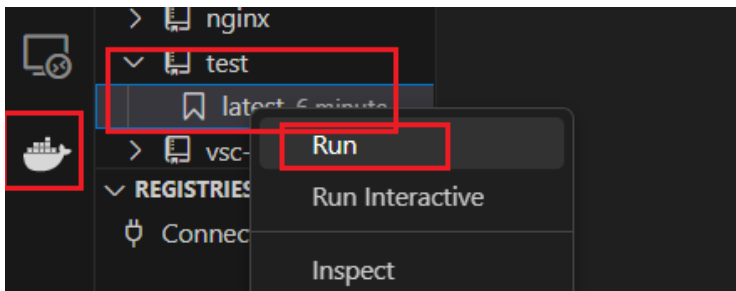
```
# Utilisation de l'image openjdk
FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAVA_OPTS
ENV JAVA_OPTS=$JAVA_OPTS
COPY target/rest-service-complete-0.0.1-SNAPSHOT.jar complete.jar
EXPOSE 8080
ENTRYPOINT exec java $JAVA_OPTS -jar complete.jar
```

Créer l'image docker

Création de l'image docker : Faites un clic droit sur le fichier DockerFile -> "Build image".



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




Sur l'application Docker desktop on peut cliquer sur l'URL : <http://localhost:8080/>



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

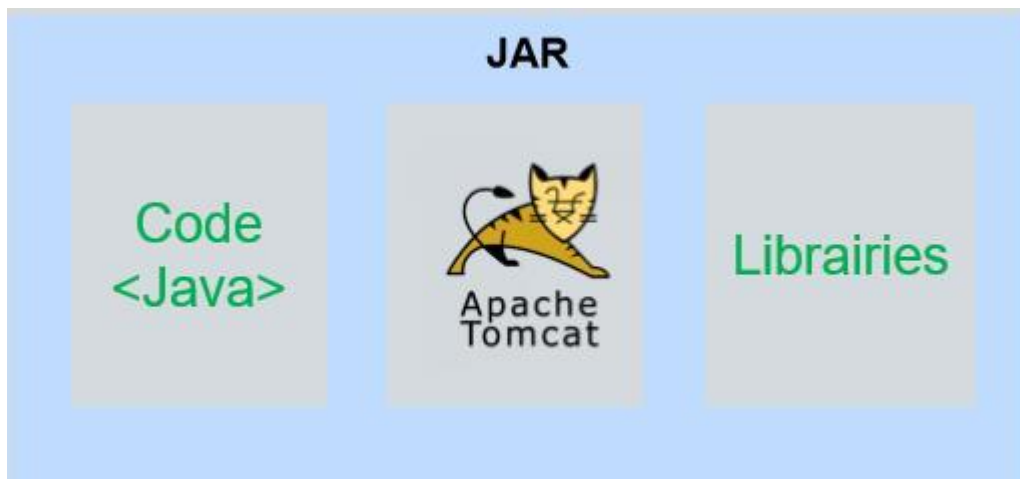
Le projet est lancé grâce à une image Docker que l'on run pour avoir un conteneur Docker qui tourne sur Docker Desktop.

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)
<input type="checkbox"/>	nervous_sutherland aedf5d8f7658		complete:l Running	0.21%	8080:8080

(Le conteneur)

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

Le build permet de créer un fichier jar contenant le code java, le serveur web et les librairies. L'outil **maven** est utilisé pour le build.



Le run permet juste de lancer l'application.

Y a-t-il un serveur web ?

Oui, un serveur Tomcat

Quelle version de java est utilisée ?

JDK 17

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

Tomcat Version 10.1.16

3.3 Création d'un projet Spring Boot

Dans cet exercice, j'ai créé un projet avec Spring Boot de la manière suivante :

1) Options à choisir lors de la création du projet en Spring Boot

JAR	Déploiement de type JAR
Spring Web	Librairie qui permet de faire du RESTful, ce qui correspond à notre API pour exposer des Endpoint.

Lombok	C'est une librairie pour optimiser certaines classes et gagner du temps
--------	---

2) Annotation importantes dans Spring Boot

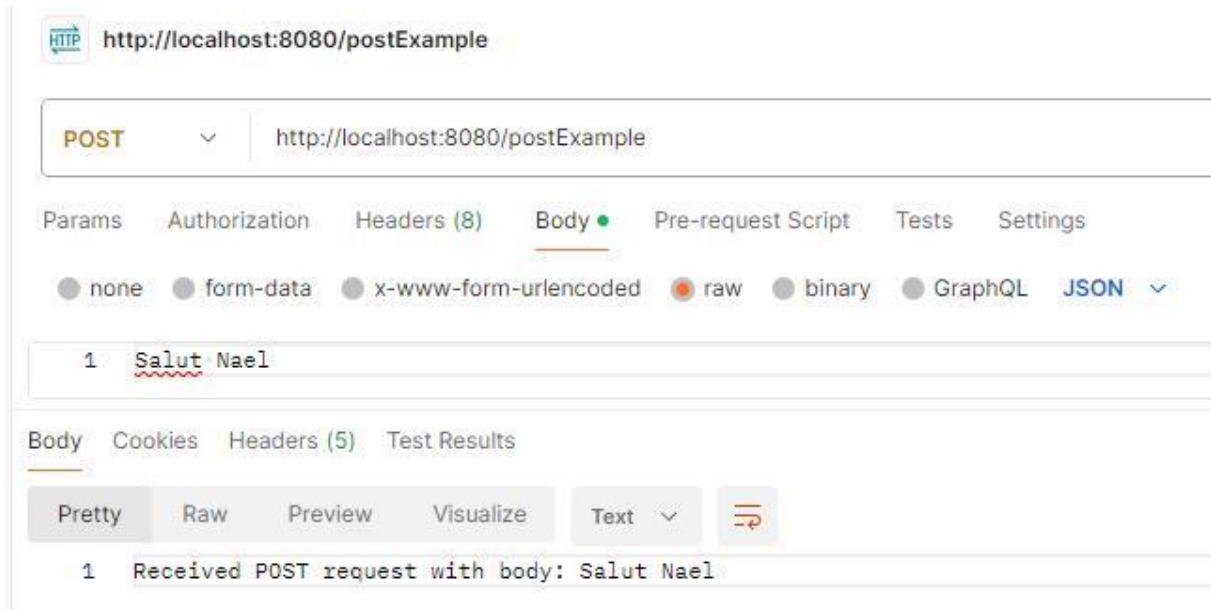
@SpringBootApplication	Toute classe comme point de démarrage d'une application
@configuration	Indique à Spring de considérer la classe sur laquelle elle est apposée comme étant un bean dans lequel d'autres beans peuvent être déclarés
@ComponentScan	Indique à Spring de scanner le package et les sous-packages passés en paramètre pour rechercher des classes qu'elle pourrait injecter comme beans dans l'application
@EnableAutoConfiguration	Détecter et d'injecter comme bean, certaines classes contenues dans les dépendances du classpath de votre application

3) Créer un controller (avec PUT, GET, POST)

```
@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;
    }
}
```

4) Tester les requêtes sur Postman

POST :



HTTP **POST** `http://localhost:8080/postExample`

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ☐

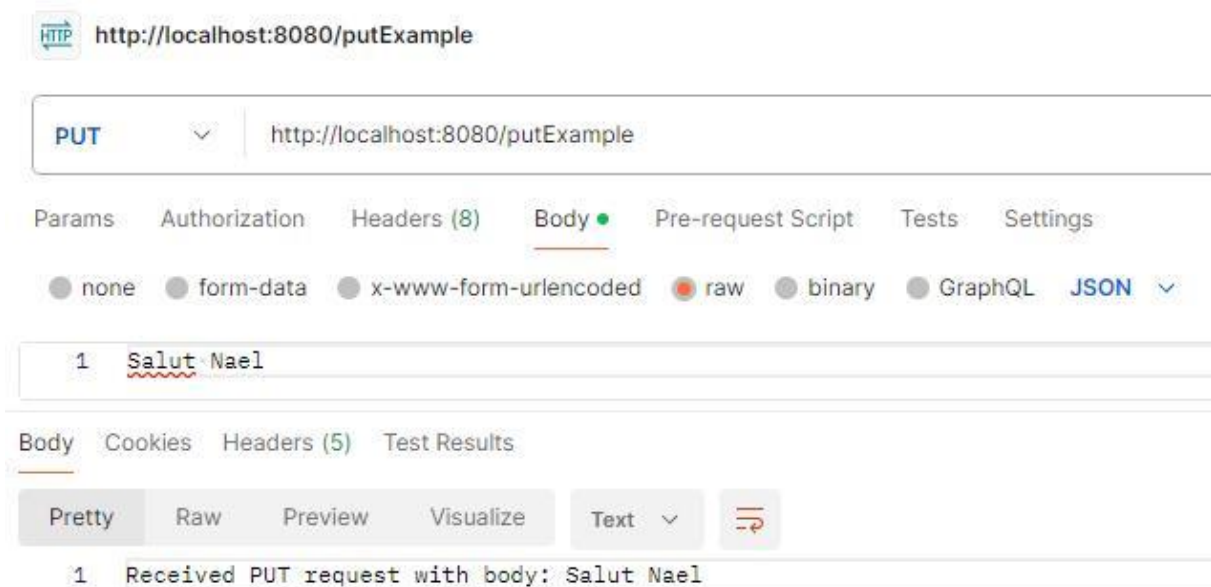
1 Salut Nael

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text ☐

1 Received POST request with body: Salut Nael

PUT :



HTTP **PUT** `http://localhost:8080/putExample`

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ☐

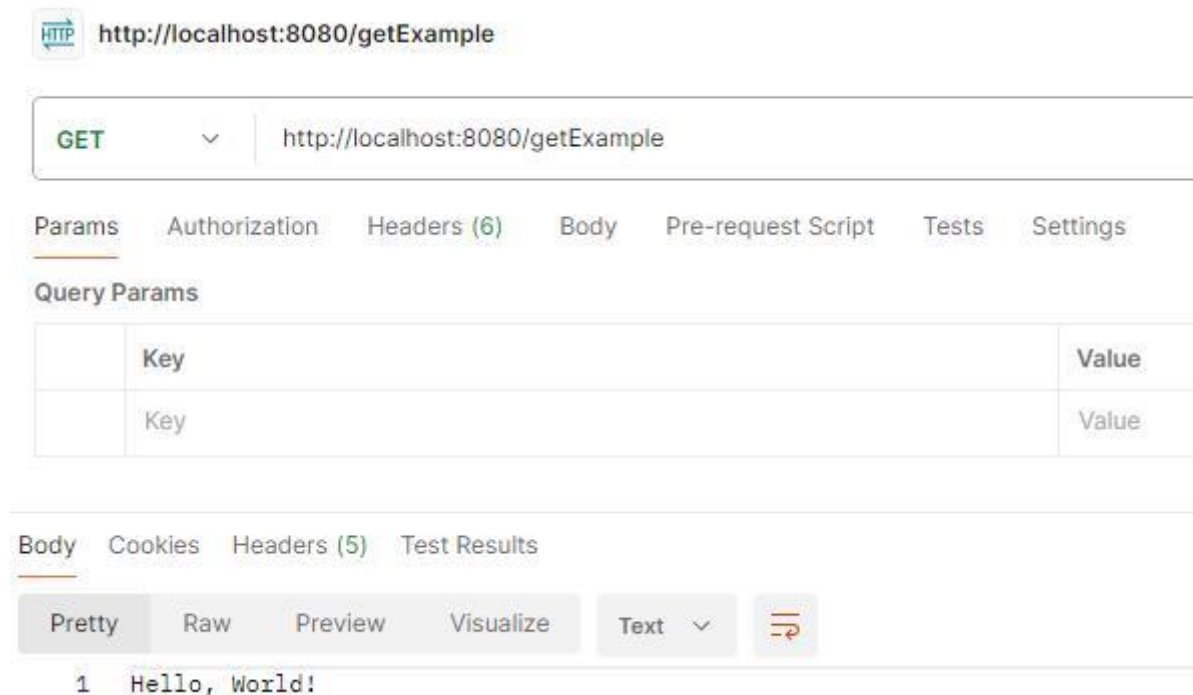
1 Salut Nael

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text ☐

1 Received PUT request with body: Salut Nael

GET :



3.4 Connexion à la DB JDBC

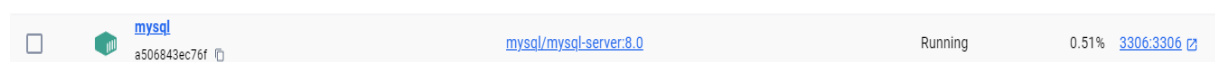
Voici comment créer une DB.

Il faut d'abord créer un nouveau container qui contiendra le serveur MySQL. Voici les commandes à exécuter.

```
#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 3306:3306 -e MYSQL_ROOT_HOST=% -e MYSQL_ROOT_PASSWORD=emf123 -v
/opt/mysql:/var/lib/mysql mysql/mysql-server:8.0
```

Voici l'image créée



Voici les commandes pour se connecter depuis le container backend sur la base de données MySQL.

```
docker network create asgard
```

```
docker network connect asgard mysql
```

Voici comment retourner l'ArrayList en format Json.

```
@GetMapping("/getPays")
public String getPays() {
    ArrayList<String> pays = wrk.getPays();
    ObjectMapper mapper = new ObjectMapper();
    try {
        String json = mapper.writeValueAsString(pays);
        return json;
    } catch (Exception e) {
        e.printStackTrace();
        return "Erreur";
    }
}
```

Pour appeler la méthode getPays() du worker il faut la instancier.

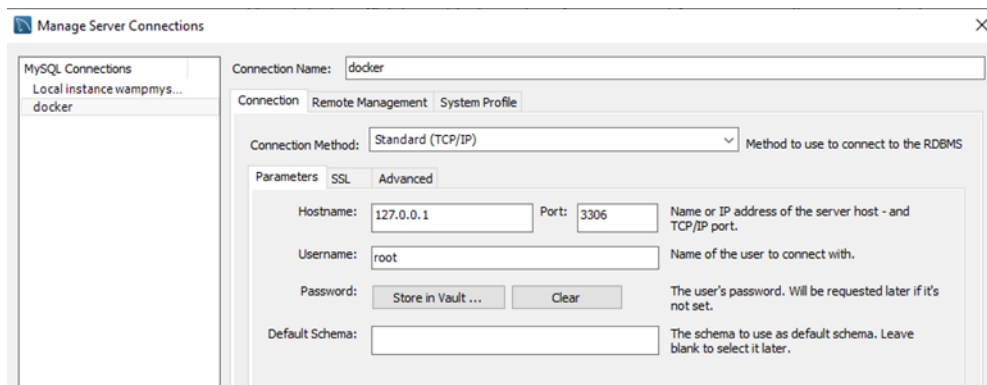
```
private WrkDB wrk;

public Controller (){
    wrk = new WrkDB("3306", "bd_kitzbuehl");
}
```

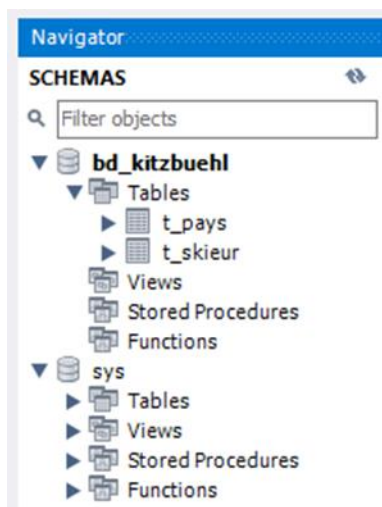
Il faut bien préciser le port et le nom de la DB dans les paramètres.

Voici la marche à suivre pour réaliser la connexion à la DB en JDBC :

- 1) Créer une connexion docker sur notre container MYSQL (avec Workbench)



- 2) Une fois connecter, importer la base de données dans la connexion Workbench



Lorsque l'on veut se connecter en Localhost :

- 1) Run le projet
- 2) Configurer la connexion dans le WrkDB en localhost

```
final String url = "jdbc:mysql://localhost:" + port + "/" + dbName + "?serverTimezone=CET";
final String user = "root";
final String pw = "emf123";
boolean result = false;
```

- 3) Tester dans le navigateur / Postman (méthode getPays)

Lorsque l'on veut se connecter sur le container docker qui tourne :

- 1) Créer le conteneur docker et le lancer (build projet/image docker/run)
- 2) Configurer la connexion dans le WrkDB avec docker

```
final String url = "jdbc:mysql://host.docker.internal:" + port + "/" + dbName +
"?serverTimezone=CEST";
final String user = "root";
final String pw = "emf123";//
boolean result = false;
```

3) Tester dans le navigateur / Postman (méthode getPays)

3.5 Connexion à la DB JPA

Voici comment réaliser une connexion à la DB en JPA :

1) Créer un projet en Spring Boot avec ces librairies

Spring Web	Permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints
Lombok	Optimiser certaines classes et gagner du temps
Spring Data JPA	Utiliser la persistance (JPA)
Mysql Driver SQL	Driver pour Mysql

2) Pour ce qui va être de l'accès à la DB, c'est dans le fichier **src/main/resources/application.properties**

```
spring.application.name=ex5
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/133ex5
spring.datasource.username=root
spring.datasource.password=emf123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
```

3) Il faut créer l'entity model dans le fichier **main/java/com/example/ex5/model/Skieur.java**

```
package com.example.ex5.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.EAGER)
@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;
}
}
```

4) Créer son interface dans **main/java/com/example/ex5/model/SkieurRepository.java**

```
package com.example.ex5.model;
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> {
}
```

5) Créer un controller qui va nous permettre d'ajouter et d'aller prendre tous les skieurs **main/java/com/example/ex5/controller/Controller.java**

```
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();
    }
}

```

6) Tester l'ajout d'un skieur

(Faire de même avec les pays, le skieur aura une fk_pays qui sera afficher comme ceci dans la classe Pays.java :

```

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "fk_pays")
private Pays pays;

)

```

3.5.1 Question

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

L'annotation @Autowired dans Spring est utilisée pour l'injection automatique de dépendances. Elle permet à Spring de résoudre et d'injecter automatiquement les dépendances de bean collaborant dans votre bean.

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

L'annotation `@ManyToOne` est utilisée dans le contexte de JPA (Java Persistence API) pour établir une relation de plusieurs à un entre deux entités.

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

Dans cette ligne, le FetchType utilisé est EAGER. Cela signifie que lorsque vous chargez une entité, toutes ses associations marquées avec FetchType.EAGER sont également chargées en même temps.

Avec FetchType.LAZY, l'association Pays ne sera chargée que lorsque vous y accéderez pour la première fois. Cela peut améliorer les performances en évitant de charger des données inutiles

3.6 Connexion à la DB JPA avec DTO

```
chmod +x mvnw
```

Ce projet est le même que le 5 à quelques exceptions près, il va nous permettre de pouvoir se connecter en local et en docker sans changer la ligne de connexion grâce à une variable :

1) en local :

```
spring.datasource.url=${DATABASE_URL:jdbc:mysql://localhost:3306/133ex6}
```

2) en docker :

```
ENV DATABASE_URL=jdbc:mysql://host.docker.internal:3306/133ex6
```

Questions

Les DTO (Data Transfer Objects) sont des objets simples utilisés dans la programmation logicielle pour transférer des données entre les couches d'une application sans exposer les entités de la base de données ou la logique métier directement

L'utilisation de DTO's vous permet de contrôler précisément les données exposées par votre API, tout en évitant les problèmes liés au chargement paresseux et à la sérialisation des entités JPA/Hibernate. Cette approche vous donne également la flexibilité d'optimiser les performances de votre application en ne chargeant que les données nécessaires.

Model	C'est le bean
Repository	C'est comme une interface du bean
Controller	C'est la classe qui permet de faire des opérations

3.7 Gestion des sessions

Login :

```
@PostMapping("/login")
public ResponseEntity<String> login(@RequestParam String username, @RequestParam String password,
HttpSession session) {
    // Ici, vous pouvez ajouter la logique d'authentification
    session.setAttribute("username", username);
    session.setAttribute("visites", 0);
    return ResponseEntity.ok("Session créée avec succès pour l'utilisateur : " + username);
}
```

```
}

```

POST http://localhost:8080/user/login

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Key	Value	Description	Bulk Edit
<input checked="" type="checkbox"/> username	Sergio		
<input checked="" type="checkbox"/> password	101010		
<input type="checkbox"/> pays	1		
Key	Value	Description	

Body Cookies (1) Headers (6) Test Results

Status: 200 OK Time: 19 ms Size: 295 B Save as example

Pretty Raw Preview Visualize Text

1 Session créée avec succès pour l'utilisateur : Sergio

Par défaut, les sessions HTTP ont une durée de vie qui est configurée dans Tomcat. La valeur par défaut est de 30 minutes. Cela signifie que si le client n'envoie pas de nouvelles requêtes pendant cette période, la session sera supprimée par le serveur.

Dans le code, utilisez `session.invalidate()` pour supprimer la session côté serveur.

Logout :

```
@PostMapping("/logout")
public ResponseEntity<String> logout(HttpSession session) {
    session.invalidate();
    return ResponseEntity.ok("Déconnecté");
}
```

POST http://localhost:8080/user/logout

Params Authorization Headers (8) Body Pre-request Script Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Key	Value
<input type="checkbox"/> username	Sergio
<input type="checkbox"/> password	101010
<input type="checkbox"/> pays	1
Key	Value

Body Cookies (1) Headers (5) Test Results

Pretty Raw Preview Visualize Text

1 Déconnecté

Visiteurs :

Les sessions enregistrent les informations de l'utilisateur et son activité sur un site web dans un fichier situé dans un répertoire temporaire sur le serveur

```
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");  
}  
}
```

GET http://localhost:8080/user/visites

Params Authorization Headers (9) Body Pre-request Script Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Key	Value
<input checked="" type="checkbox"/> username	Sergio
<input checked="" type="checkbox"/> password	101010
<input type="checkbox"/> pays	1
Key	Value

Body Cookies (1) Headers (5) Test Results

Pretty Raw Preview Visualize Text

```
1 Nombre de visites: 1
```

3.8 Documentation API avec Swagger

La documentation d'une API est une tâche lourde et ennuyeuse. C'est pourquoi Swagger nous crée une documentation complète et ordonnée.

Il faut rajouter ce code dans le pom.xml :

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

Ensuite, il faut build le projet, le run et regarder notre documentation sur cette url :

<http://localhost:8080/swagger-ui/index.html>

4 Analyse à faire complètement avec EA -> à rendre uniquement le fichier EA

Fichier EA github

5 Conception à faire complètement avec EA -> à rendre uniquement le fichier EA

Fichier EA github

6 Dockerhub

Voici la marche à suivre pour mettre son projet sur Dockerhub :

1) Créer un Dockerfile dans le projet

```
FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAVA_OPTS
ENV JAVA_OPTS=$JAVA_OPTS
COPY target/rest1-0.0.1-SNAPSHOT.jar rest1.jar
EXPOSE 8080
ENTRYPOINT exec java $JAVA_OPTS -jar rest1.jar
ENV DATABASE_URL=jdbc:mysql://host.docker.internal:3306/rest1
```

2) Créer un compte, un répertoire et un token sur dockerhub (il faut garder le token copier quelque part, car on en aura besoin)

<https://hub.docker.com>

<https://hub.docker.com/settings/security>

3) Faire le tag et push son image sur dockerhub

```
docker tag imageDocker:latest username/nomRepo:latest
docker push username/nomRepo:latest
```

4) Run l'image, il doit aller la chercher sur dockerhub (8081 est le port de mon APIRest)

```
docker run --rm -d -p 8081:8080/tcp username/nomRepo:latest
```

6.1 Créer la DB sur le serveur Docker

Tout d'abord, il faut créer le schéma de la DB sur Workbench et générer le script SQL (Forward Engineer).

1) Se connecter au serveur Docker avec les identifiants (sur Ubuntu)

```
ssh nomAdmin@adresseIPServeur
```

On peut se mettre en mode superadmin

```
sudo -i
```

2) 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 3306:3306 -e MYSQL_ROOT_HOST=% -e MYSQL_ROOT_PASSWORD=emf123 -v /opt/mysql:/var/lib/mysql mysql/mysql-server:8.0
```

3) Créer le fichier sql du schéma depuis le local vers le container docker

```
nano schema.sql  
docker cp /root/schema.sql ad1f9742645e:/root/schema.sql
```

4) Entrer dans l'image docker

```
docker exec -it idImage bash
```

5) Accéder à MYSQL

```
mysql -u root -p
```

6) Appliquer la source comme étant le script sql

```
source /root/schema2.sql
```

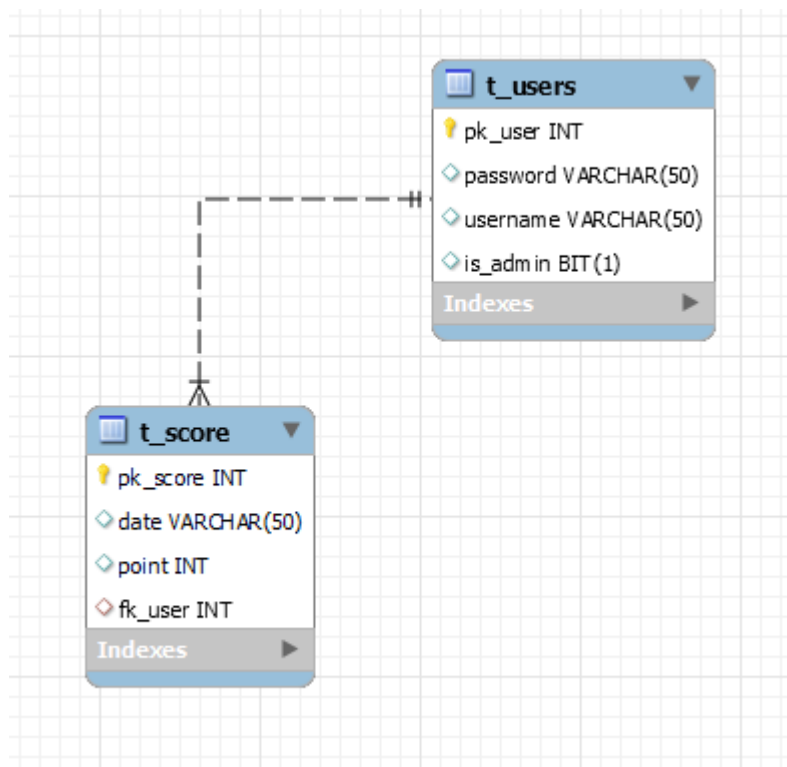
7) Sélectionner la database

```
SHOW DATABASES;  
USE nom_db ;  
SHOW TABLES ;
```

8) Faire les requêtes pour mettre des données dans les tables

7 Bases de données

7.1 Modèles WorkBench MySQL



Voici mon modèle qui contient 2 tables qui dit qu'un user peut avoir plusieurs scores mais un score peut avoir seulement un User.

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

6.1) Une descente de code client

9 Implémentation de l'application <API Gateway>

9.1 Une descente de code APIGateway

Voici la méthode `getScoresUsers()`.

```
public ResponseEntity<String> getScoresUsers() {
    String url = baseUrl + "/scoreUsers"; // Ajustez selon l'API que vous appelez

    // Effectuer l'appel API et recevoir la réponse
    ResponseEntity<String> response = restTemplate.getForEntity(url,
String.class);

    // Supposons que l'API renvoie un statut 200 avec un corps contenant
    // {"status":"success", "data": [{...}]} en cas de succès
    if (response.getStatusCode().is2xxSuccessful()) {
        String responseBody = response.getBody();
        if (responseBody != null) {
            return ResponseEntity.ok(responseBody);
        }
    }
    // Si quelque chose ne va pas, renvoyer une erreur
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
        .body("Erreur lors de la récupération des questions");
}
```

Ce code va faire appel à la méthode `scoreUser` qui se trouve dans le `Rest2`, on va utiliser le lien `url` pour accéder.

Ensuite on fait appel à la méthode dans le `controller`.

```
GetMapping("/getScoresUser")
public ResponseEntity<String> getAllScoresUser() {
    try {
        // Appelle la méthode du service
        ResponseEntity<String> response = rest2.getScoresUsers();

        // Vérifie si la réponse est réussie (code d'état 200)
        if (response.getStatusCode().is2xxSuccessful()) {
            // Retourne HTTP 200 avec le corps de la réponse en cas de succès
            return ResponseEntity.ok(response.getBody());
        } else {
            // Retourne HTTP 400 avec un message d'erreur en cas d'échec
            return ResponseEntity.badRequest().body("Échec de la
récupération des questions");
        }
    } catch (Exception e) {
        // Retourne HTTP 400 avec un message d'erreur en cas d'exception
    }
}
```

```
        return ResponseEntity.badRequest().body("Erreur : " +  
e.getMessage());  
    }  
}
```

10 Implémentation des applications et <API élève2>

10.1 Une descente de code de l'API REST

Il faut afficher les scores d'un User en format JSON. Dans mon fichier ScoreService.java qui se trouve dans le dossier service de mon Rest2 j'ai fait cette méthode.

```
@Transactional
public String getAllScores() {
    List<Score> scores = (List<Score>) scoreRepository.findAll();
    Map<String, List<Integer>> userScores = new HashMap<>();

    for (Score score : scores) {
        String username = score.getUser().getUsername();
        if (!userScores.containsKey(username)) {
            userScores.put(username, new ArrayList<>());
        }

        userScores.get(username).add(score.getPoint());
    }

    JSONObject json = new JSONObject(userScores);
    return json.toString();
}
```

Ce code va me permettre d'afficher les Scores d'un User dans une HashMap et le formater en JSON. J'ai ajouté cette librairie dans le fichier pomp.xml pour le JSON

```
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20210307</version>
</dependency>
```

Voici le code de mon controller et j'utilise la méthode getAllScores().

```
@GetMapping("/scoreUsers")
public @ResponseBody String showScore() {
    return scoreService.getAllScores();
}
```

J'utilise un Get. Voici dans le Postman le test qui montre les scores d'un user en format JSON.

GET

⌵

http://localhost:8080/scoreUsers

Params ● Authorization Headers (8) Body ● Pre

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐

	Key	Value
<input checked="" type="checkbox"/>	point	10
<input checked="" type="checkbox"/>	fk_user	1
	Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text ⌵

1 [{"carlos": [10]}

11 Hébergement

Dans le point précédent, nous avons déjà créer la database sur le serveur docker. Nous allons maintenant héberger les APIRest sur le serveur docker.

1) Ajouter cette ligne dans le application.properties

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

2) Ensuite cette ligne dans le dockerfile

```
ENV DATABASE_URL=jdbc:mysql://mysql:3306/rest1
```

3) Il faut pousser son image sur dockerhub à nouveau (mettre à jour)

```
docker tag nomImage:latest username/nomRepo:latest  
docker push username/nomRepo:latest
```

4) On va se connecter au serveur docker, y créer un réseau et le lier au container mysql

```
docker network create reseauDocker  
docker network connect reseauDocker mysql
```

(mysql = nom container mysql)

5) Supprimer container Rest déjà existant (s'il y en a un)

```
docker ps -a  
docker stop containerid /// docker rm containerid
```

6) Supprimer image Rest déjà existante (s'il y en a une)

```
docker images  
docker rmi containerid
```

7) Lancer le container sur le port voulue (dans ce cas 8081)

```
docker run --network reseauDocker --name nomImageACreer -p  
8081:8080/tcp -d username/nomRepo:latest
```

11.1 Infrastructure en local

Lancement REST2 sur 8082:

- 1) Build maven (package)
- 2) Build sur DockerFile
- 3) Lancement en local

```
docker run --network reseauDocker --name rest2 -p 8082:8080/tcp -d  
projet133-rest2:latest
```

Lancement de l'APIGTW:

- 1) Build maven (package)
- 2) Run java (8080)

12 Installation du projet complet avec les 5 applications

13 Tests de fonctionnement du projet

Nous avons testé notre backend en local uniquement, voici la manière dont nous avons procédé :

- 1) Lancer mon REST 2 sur un container tournant sur le port 8082 :8080
- 2) Run java sur mon APIGateway
- 3) Tester la méthode du REST2

GET ▼ <http://localhost:8082/scoreUsers>

Tester la méthode depuis l'APIGateway

GET ▼ <http://localhost:8080/scoreUsers>

14 Auto-évaluations et conclusions

14.1 Auto-évaluation et conclusion de Sergio

J'ai trouvé que j'ai bien avancé dans le module et bien compris tout ce que je faisais. J'ai posé toutes mes questions au prof et cela m'a aidé pour avancer dans mon projet et comprendre mieux. Ce que j'ai aimé dans ce module c'est la partie projet en groupe, j'ai trouvé cela très utile et très efficace de travailler à deux pour s'entre aider entre nous. Ce que j'ai moins aimé c'est les Test technologiques et la réalisation de diagramme.