133 - Réaliser une application web en Session-Handling

|  |
| --- |
| Rapport personnel / Doc Projet |

Version 1 du 20.06.2024  
Date de création : 21.03.2024

Naël Currat

Début du module :  
21.03.2024  
Fin du module :  
03.05.2024

Type du module : Développement

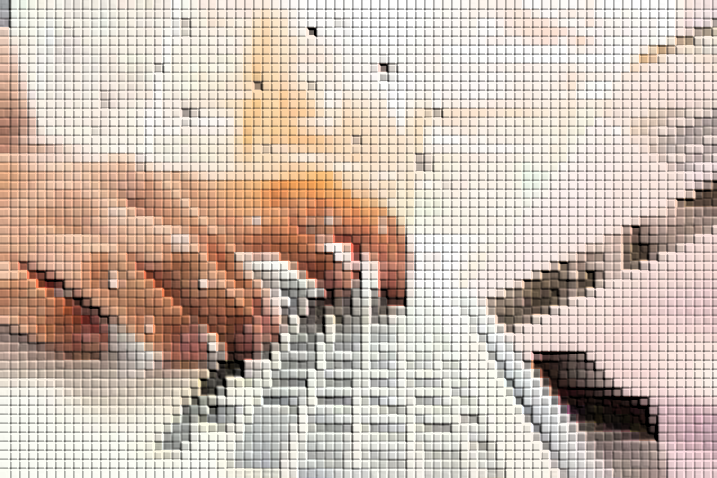


Table des matières

[1 Introduction 4](#_Toc165813888)

[2 Tests technologiques selon les exercices 5](#_Toc165813889)

[2.1 Installation et Hello World 5](#_Toc165813890)

[2.1.1 Questions 6](#_Toc165813891)

[2.2 Conteneurisation 7](#_Toc165813892)

[2.2.1 Questions 8](#_Toc165813893)

[2.3 Création d'un projet Spring Boot 9](#_Toc165813894)

[2.4 Connexion à la DB JDBC 11](#_Toc165813895)

[2.5 Connexion à la DB JPA 14](#_Toc165813896)

[2.5.1 Questions 16](#_Toc165813897)

[2.6 Connexion à la DB JPA avec DTO 17](#_Toc165813898)

[2.6.1 Questions 17](#_Toc165813899)

[2.7 Gestion des sessions 17](#_Toc165813900)

[2.8 Documentation API avec Swagger 19](#_Toc165813901)

[3 Projet 20](#_Toc165813902)

[3.1 Description 20](#_Toc165813903)

[3.2 Projet mis sur Dockerhub 20](#_Toc165813904)

[3.3 Créer la DB sur le serveur Docker 20](#_Toc165813905)

[4 Diagrammes 22](#_Toc165813906)

[4.1 Diagramme classe conception 22](#_Toc165813907)

[4.2 Reverse code Implémentation 22](#_Toc165813908)

[5 Bases de données 24](#_Toc165813909)

[5.1 Modèles Workbench MySQL 24](#_Toc165813910)

[6 Implémentation des applications <Le client Ap1> 25](#_Toc165813911)

[6.1 Une descente de code client 25](#_Toc165813912)

[7 Implémentation de l'application <API Gateway> 26](#_Toc165813913)

[7.1 Une descente de code APIGateway 26](#_Toc165813914)

[7.1.1 Service 26](#_Toc165813915)

[7.1.2 Controller 26](#_Toc165813916)

[8 Implémentation des applications <API élève1> 28](#_Toc165813917)

[8.1 Une descente de code de l'API REST 28](#_Toc165813918)

[8.1.1 Service 28](#_Toc165813919)

[8.1.2 Controller 28](#_Toc165813920)

[9 Hébergement 29](#_Toc165813921)

[9.1 Héberger API et DB 29](#_Toc165813922)

[9.2 Infrastructure en local 29](#_Toc165813923)

[10 Installation du projet complet avec les 5 applications 31](#_Toc165813924)

[11 Tests de fonctionnement du projet 32](#_Toc165813925)

[4) Auto-évaluation et conclusion 33](#_Toc165813926)

[11.1 Auto-évaluation et conclusion de Nael 33](#_Toc165813927)

# 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 |

# Tests technologiques selon les exercices

## 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

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

4) Se connecter avec VScode en Ubuntu



5) Installer les extensions nécessaires

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement

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)

### Questions

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

1) Le projet a été lancé en utilisant Java 17.0.10 avec le PID 10330. Le répertoire de travail est /home/nael42/gs-rest-service.

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

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

4) Le WebApplicationContext intégré de Spring a été initialisé.

5) Le DispatcherServlet ‘dispatcherServlet’ de Spring a été initialisé.

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

C’est Maven. L’extension suivante :

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

Description générée automatiquement

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

Oui, c’est un serveur Tomcat

**Quelle version de java est utilisée ?**

OpenJDK 17

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

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

Tomcat Version 10.1.16

## Conteneurisation

Il faut installer maven avec cette commande :

sudp apt install maven

Premièrement, nous allons devoir build le projet java avec maven

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

Description générée automatiquement

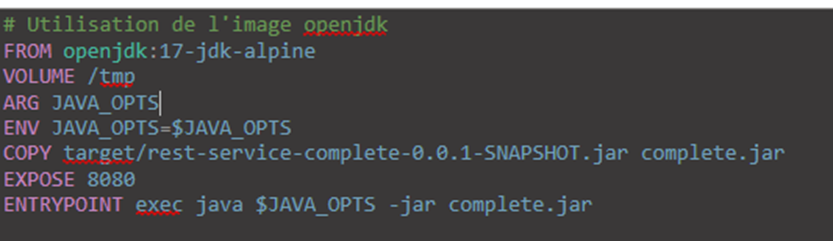
Ensuite il faut créer le fichier Docker. Nous allons faire Ctrl + shift + P.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

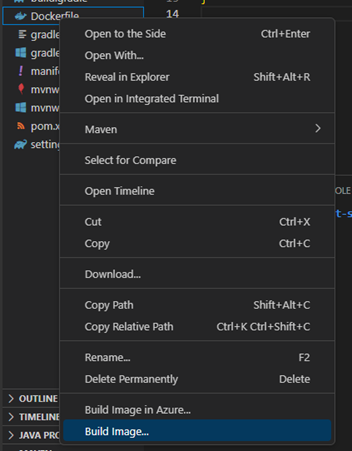
Choisissez ensuite "Java", le port 8080 et le fichier pom.xml.

La version de JDK est 17.

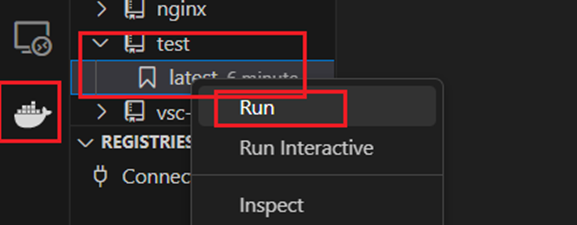


Créer l’image docker

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/>

Une image contenant texte, Police, ligne, nombre

Description générée automatiquement

### Questions

**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.

Une image contenant texte, Police, ligne, nombre

Description générée automatiquement

(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.

Une image contenant texte, Police, logo, conception

Description générée automatiquement

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 ?**

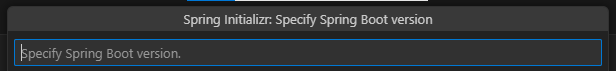
JDK 17

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

Tomcat Version 10.1.16

## 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 librarie 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 |
| **@CompoenentScan** | 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 :

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

Description générée automatiquement

PUT :

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

Description générée automatiquement

GET :

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

Description générée automatiquement

## Connexion à la DB JDBC

Voici comment créer une DB :

1) 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 :

image

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

docker network create asgard

docker network connect asgard mysql

Voici comment retourne 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)

Une image contenant texte, nombre, logiciel, Police

Description générée automatiquement

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

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

Description générée automatiquement

**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 containeur 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=CET";

final String user = "root";

final String pw = "emf123";//

boolean result = false;

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

## 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

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

Description générée automatiquement

(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;

)

### Questions

**À quoi sert l'annotation @Autowired dans vos controlleur 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, rééssayer 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

## Connexion à la DB JPA avec DTO

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 (**dans application.properties**) :

spring.datasource.url=${DATABASE\_URL:jdbc:mysql://localhost:3306/133ex6}

2) en docker (**dans le dockerfile)** :

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 |

## 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);

}

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

Description générée automatiquement

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");

}

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

Description générée automatiquement

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");

}

}

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

Description générée automatiquement

## Documentation API avec Swagger

La documentation d’une API est une tache lourde et ennuyante. C’est pourquoi Swagger nous créer 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>

# Projet

## Description

Notre projet est un quizz qui comporte des questions et 4 réponses par question dont une correcte. Le joueur doit se loger 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.

## Projet mis sur 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

## 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

# Diagrammes

## Diagramme classe conception

Voici mon diagramme de conception, avant de commencer le code de mon projet :

Une image contenant texte, capture d’écran, Rectangle, Parallèle

Description générée automatiquement

## Reverse code Implémentation

Voici le reverse du code de mon REST1 après l’implémentation :

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

Description générée automatiquement

On peut voir qu’il y a pas mal de différences entre les deux, mais on retrouve aussi des similitudes comme le Controller et les Beans.

# Bases de données

## Modèles Workbench MySQL

Voici le modèle de ma base de données (APIRest1) :

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

Description générée automatiquement

Ma DB contient 2 tables :

**Réponse :** Gère les réponses, indique si la réponse est correcte ou non et appartient à une question.

**Question :** Gère les questions qui ont un énoncé.

# Implémentation des applications <Le client Ap1>

## Une descente de code client

# Implémentation de l'application <API Gateway>

## Une descente de code APIGateway

### Service

    public ResponseEntity<String> getQuestions() {

        String url = base\_url + "/getQuestions"; // 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");

    }

Cette méthode va faire un appel au REST1 et va renvoyer une réponse pour le controller.

### Controller

    @GetMapping("/getQuestions")

    public ResponseEntity<String> getAllQuestions() {

        try {

            // Appelle la méthode du service

            ResponseEntity<String> response = rest1.getQuestions();

            // 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());

        }

    }

On appel la méthode du service REST1 de l’APIGateway et il va rendre une réponse que le client va devoir traiter.

# Implémentation des applications <API élève1>

## Une descente de code de l'API REST

### Service

Voici ma méthode getAllQuestions dans mon service :

public String getAllQuestions() {

    List<Reponse> reponses = (List<Reponse>) reponseRepository.findAll();

    Map<String, List<String>> questionReponses = new HashMap<>();

    for (Reponse reponse : reponses) {

        String question = reponse.getQuestion().getEnoncer();

        if (!questionReponses.containsKey(question)) {

            questionReponses.put(question, new ArrayList<>());

        }

        questionReponses.get(question).add(reponse.getReponse());

    }

    JSONObject json = new JSONObject(questionReponses);

    return json.toString();

}

C’est une méthode qui ajoute dans un MAP toutes les questions et leurs réponses, et qui retourne toutes ces données en format JSON.

### Controller

Voici le suivi de cette méthode dans le controller :

@GetMapping("/getQuestions")

    public @ResponseBody String getQuestionsReponses() {

        return reponseService.getAllQuestions();

    }

On fait un GET qui va chercher la méthode dans le service.

# Hébergement

## Héberger API et DB

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/dbprojet}

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

## Infrastructure en local

Lancement REST1 sur 8081:

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

docker run --network reseauDocker --name rest1 -p **8081**:8080/tcp -d **projet133-rest1**:latest

Lancement de l’APIGTW:

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

# Installation du projet complet avec les 5 applications

Notre APIGateway, nos DB et nos deux REST sont héberger, donc nous avons réussi à mettre en place toute la partie Backend, mais ils nous restent encore les clients à mettre en place.

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

Description générée automatiquement

# Tests de fonctionnement du projet

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

1) Lancer mon REST 1 sur un container tournant sur le port 8081 :8080

2) Run java sur mon APIGateway

3) Tester la méthode du REST1

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

Description générée automatiquement

4) Tester la méthode depuis l’APIGateway



# Auto-évaluation et conclusion

## Auto-évaluation et conclusion de Nael

**Point positif :** Je trouve que j’ai pu bien progresser durant ce module en posant des questions et en documentant les choses que j’ai compris. Il nous restera un bout de projet à faire en R2 mais dans l’ensemble je trouve que nous avons bien avancé.

**Point négatif :** Je trouve que le temps (jours perdues, …) était un peu limite pour faire les clients. Il y avait également un peu trop de TT.