1. fDu 21.03.2024 au  
   03.05.2024

Rapport personnel

Hoti Flamur

Date de création : 21 mars 2024  
Version 1 du : 28.03.24

Table des matières

1 Introduction 3

2 Tests technologiques selon les exercices 4

2.1 Installation et Hello World 4

2.2 Conteneurisation 4

2.2.1 Création de l'image docker 5

2.3 Création d'un projet Spring Boot 6

2.4 Connexion à la DB JDBC 7

2.5 Connexion à la DB JPA 8

2.6 Connexion à la DB JPA avec DTO 12

2.7 Gestion des sessions 13

2.7.1 Test 15

2.8 Documentation API avec Swagger 16

2.9 Hébergement 17

3 Projet – DH Destinations 18

3.1 Analyse à faire complètement avec EA -> à rendre uniquement le fichier EA 18

3.1.1 Use case client et use case Rest. 18

3.1.2 Activity Diagram d'un cas complet navigant dans les applications avec les explications 18

3.1.3 Sequence System global entre les applications 18

3.2 Conception à faire complétement avec EA -> à rendre uniquement le fichier EA 18

3.2.1 Class Diagram complet avec les explications de chaque application 18

3.3 Bases de données 18

3.3.1 Modèles WorkBench MySQL 18

3.4 Implémentation des applications <Le client Ap1> et <Le client Ap2> 18

3.4.1 Une descente de code client 18

3.5 Implémentation de l'application <API Gateway> 18

3.5.1 Une descente de code APIGateway 18

3.6 Implémentation des applications <API élève1> et <API élève2> 18

3.6.1 Une descente de code de l'API REST 18

3.7 Hébergement 18

3.8 Installation du projet complet avec les 5 applications 18

3.9 Tests de fonctionnement du projet 18

4 Auto-évaluations et conclusions 19

4.1 Auto-évaluation et conclusion de … 19

4.2 Auto-évaluation et conclusion de … 19

# Introduction

# Tests technologiques selon les exercices

## Installation et Hello World

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

* L’application est lancée par la commande Starting RestServiceApplication, l’application tourne en utilisant le Framework Spring Boot et le serveur web Tomcat.

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

* Build : Cette étape consiste à compiler le code source Java en bytecode, qui peut être exécuté par la machine virtuelle Java (JVM).
* Run : Cette étape consiste à exécuter l’application Java. Cela se fait en lançant la JVM et en lui donnant le bytecode à exécuter.

Y a-t-il un serveur web ?

* Oui, Tomcat est actif.

Quelle version de java est utilisée ?

* Java en version 17.0.6

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

* Tomcat en version 10.0.16

## Conteneurisation

J’ai repris l’exercice précédent dans VSCode : ouvert le dossier gs-rest-service/complete qui contient l'application d'exemple.

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

Description générée automatiquement

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

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 gsrestservice.jar

EXPOSE 8080

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

# For Spring-Boot project, use the entrypoint below to reduce Tomcat startup time.

#ENTRYPOINT exec java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar gsrestservice.jar

### Création de l'image docker

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

Description générée automatiquement

Pourquoi faire un container pour une application Java ?

* Les conteneurs peuvent être déployés sur n’importe quelle machine qui a Docker installé, ce qui rend l’application facilement portable entre différents systèmes d’exploitation et environnements.
* Les conteneurs Docker offrent plusieurs avantages pour le déploiement d’applications Java.
* Ils permettent de créer des environnements isolés et reproductibles pour les applications, ce qui facilite le déploiement et la mise à l’échelle.
* De plus, comme les conteneurs Docker encapsulent toutes les dépendances de l’application, ils garantissent que l’application fonctionnera de la même manière sur toutes les machines.

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

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

* Oui, c’est Spring Boot. Il est directement inclus dans l’applications

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

* Faire attention a la version du jdk

sudo apt-get install openjdk-17-jdk

* Faire attention à la version du Dockerfile

FROM openjdk:17-jdk-alpine

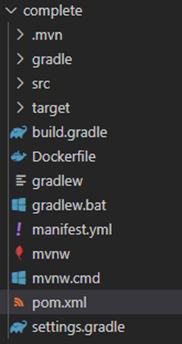
* Faire attention à la version du pom.xml

<properties>

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

</properties>

* Faire attention que le Dockerfile soit au même endroit que le pom.xmlr



## Création d'un projet Spring Boot

Une image contenant texte, capture d’écran, Compositing numérique, Logiciel de jeu vidéo

Description générée automatiquement

@RestController : Cette annotation indique que la classe est un contrôleur REST. Cela signifie que les méthodes de cette classe sont prêtes à traiter les requêtes HTTP entrantes.

@GetMapping("/getExample") : Cette annotation indique que la méthode getExample doit être appelée lorsqu’une requête HTTP GET est envoyée à l’URL “/getExample”. La méthode prend un paramètre name qui a une valeur par défaut de “World” si aucun autre nom n’est fourni dans la requête. Elle renvoie ensuite une chaîne formatée qui dit “Hello, [name]!”.

@PostMapping("/postExample") : Cette annotation indique que la méthode postExample doit être appelée lorsqu’une requête HTTP POST est envoyée à l’URL “/postExample”. La méthode prend le corps de la requête HTTP comme paramètre et renvoie une chaîne qui dit "Received POST request with body: " suivie du contenu du corps de la requête.

@PutMapping("/putExample") : Cette annotation indique que la méthode putExample doit être appelée lorsqu’une requête HTTP PUT est envoyée à l’URL “/putExample”. Comme pour la méthode postExample, cette méthode prend le corps de la requête HTTP comme paramètre et renvoie une chaîne qui dit "Received PUT request with body : " suivie du contenu du corps de la requête.

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

Description générée automatiquement

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

Description générée automatiquement

## Connexion à la DB JDBC

Pour pouvoir se connecter, il faut commencer par ajouter ceci dans pom.xml

<dependency>

<groupId>com.mysql</groupId>

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

<scope>runtime</scope>

</dependency>

Pour se connecter à la base de données on a une classe dédiée :

private Connection dbConnexion;

public boolean openConnexion() {

final String url = "jdbc:mysql://localhost:3306/bd\_kitzbuehl?serverTimezone=CET";

final String user = "root";

final String pw = "emf123";//

boolean result = false;

try {

// nécessaire pour fonctionnement en web

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

} catch (ClassNotFoundException ex) {

System.out.println("Connexion au driver JDBC à échoué!\n" + ex.getMessage());

}

try {

dbConnexion = DriverManager.getConnection(url, user, pw);

// System.out.println("Connection successfull");

result = true;

} catch (SQLException ex) {

System.out.println("Connexion à la BD a échouée!\n" + ex.getMessage());

}

return result;

}

Puis depuis notre Controller on appelle la méthode getPays pour aller chercher la liste de pays et on la convertie en json :

// Handler pour GET Pays

@GetMapping("/getPays")

public String getPays() {

ArrayList<String> pays = wrkDB.getPays();

// Convertir l'ArrayList en JSON

ObjectMapper objectMapper = new ObjectMapper();

String paysJson = null;

try {

paysJson = objectMapper.writeValueAsString(pays);

} catch (Exception e) {

e.printStackTrace(); // Gérer l'exception correctement dans votre application

}

return paysJson;

}

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

Description générée automatiquement

## Connexion à la DB JPA

Cette fois ci la connexion à la base de données est faite depuis le fichier application.properties :

spring.application.name=ex5

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:mysql://host.docker.internal: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

Puis on utilise des repository :

Dans notre exemple, PaysRepository est une interface qui étend CrudRepository. Cette interface fournit des méthodes pour effectuer les opérations CRUD (Create, Read, Update, Delete) sur l'entité Pays. L'interface CrudRepository est fournie par Spring Data et contient des méthodes telles que save, findById, findAll, delete, etc., qui vous permettent d'effectuer ces opérations de base sur votre entité Pays sans avoir à écrire de code spécifique pour chaque opération.

package com.example.ex5.model;

import org.springframework.data.repository.CrudRepository;

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

// CRUD refers Create, Read, Update, Delete

public interface PaysRepository extends CrudRepository<Pays, Integer> {

}

Puis j’ai créé une entity model pour chaque beans :

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;

}

}

Et le travail est fait depuis le Controller :

package com.example.ex5.controller;

import com.example.ex5.model.Pays;

import com.example.ex5.model.PaysRepository;

import com.example.ex5.model.Skieur;

import com.example.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

private SkieurRepository skieurRepository;

@Autowired

private PaysRepository paysRepository;

@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();

}

@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();

}

}

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

* L'annotation @Autowired est utilisée pour effectuer l'injection de dépendances dans Spring. Dans le contexte d'un contrôleur Spring qui utilise des repositories, l'annotation @Autowired est utilisée pour injecter automatiquement une instance du repository dans le contrôleur.
* Cela signifie que lorsque Spring instancie votre contrôleur, il recherchera un bean géré par Spring qui implémente SkieurRepository et injectera automatiquement cette instance dans le champ skieurRepository de votre contrôleur.

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

* L'annotation @ManyToOne est utilisée en Java Persistence API (JPA) pour définir une relation many-to-one entre deux entités dans une base de données relationnelle. Dans le contexte de votre entité Skieur, l'utilisation de @ManyToOne est destinée à définir la relation entre un skieur et son pays.

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

* FetchType.EAGER est utilisé pour charger la relation many-to-one entre un skieur et son pays de manière "hâtive" ou "immédiate". Cela signifie que les données liées à la relation (dans ce cas, le pays du skieur) seront récupérées dès que l'entité principale (dans ce cas, le skieur) est chargée.
* Dans le cas de FetchType.LAZY, lorsque vous récupérez un Skieur, son pays associé ne sera pas immédiatement chargé.

## Connexion à la DB JPA avec DTO

Cet exercice change car on retrouve en plus un répertoire DTO et un répertoire Service par rapport à l’exercice 5 :

DTO :

public SkieurDTO() {}

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

this.id = id;

this.name = name;

this.position = position;

this.paysNom = paysNom;

}

Service:

package com.example.ex6.service;

@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";

}

}

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

* On a utilisé un SkieurDTO dans notre application pour encapsuler les données de Skieur et les transférer entre différentes parties de votre application, ou pour adapter les données de Skieur à d'autres systèmes ou services externes.

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.f

* Model : Les modèles représentent la structure des données et la logique métier de l’application. Ils définissent comment les données sont stockées et manipulées. Dans ce cas, “Pays” et “Skieur” sont des classes représentant respectivement les entités pays et skieur.
* Repository : Les repositories gèrent les opérations de données et encapsulent la logique nécessaire pour accéder aux sources de données. Ils font office d’intermédiaires entre la logique métier de l’application et la source de données. Ici, “PaysRepository” et “SkieurRepository” sont utilisés pour les interactions avec la base de données relatives aux pays et aux skieurs.
* DTO: DTO signifie Data Transfer Object. C’est un objet qui transporte des données entre les processus au sein d’un ensemble bien défini de méthodes pour une tâche métier spécifique. Dans ce contexte, “SkieurDTO” pourrait être utilisé pour transférer des données liées aux skieurs de manière plus efficace que d’envoyer des informations individuelles séparément.
* Services : Les services contiennent la logique métier qui est indépendante de toute implémentation de protocole ou d’interface spécifique (comme HTTP). Ils encapsulent les règles métier et contrôlent comment les entités interagissent entre elles. Ici, nous avons deux classes de service - une pour gérer les opérations liées aux pays (“PaysService”) et une autre pour les skieurs (“SkieurService”).
* Controller : Les contrôleurs gèrent les requêtes entrantes des clients/utilisateurs. Ils reçoivent des entrées des utilisateurs via des interfaces comme GUI ou des points d’accès API ; traitent ces entrées en interagissant avec les modèles ; puis renvoient des réponses aux utilisateurs ou mettent à jour les vues en conséquence.

## Gestion des sessions

HttpSession est une interface dans Servlet qui fournit un moyen d’identifier un utilisateur à travers plusieurs requêtes HTTP. Lorsqu’un utilisateur accède à votre application web pour la première fois, le serveur crée une nouvelle session HTTP pour cet utilisateur. Pour chaque session, le serveur génère un identifiant unique, appelé ID de session, qui est renvoyé au client dans la réponse HTTP.

Dans le Controller j’ai ajouté les 3 méthodes demandées :

@PostMapping(path = "/login")

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

HttpSession session) {

// Vérifiez les identifiants de l'utilisateur

final boolean validCredentials = userService.checkCredentials(username, password);

if (!validCredentials) {

return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Identifiants incorrects");

}

// Stockez le nom d'utilisateur dans la session

session.setAttribute("username", username);

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

if (visites == null) {

visites = 0;

}

session.setAttribute("visites", visites + 1);

return ResponseEntity.ok("Logged in with " + username);

}

@PostMapping(path = "/logout")

public ResponseEntity<String> logout(HttpSession session) {

// Invalidates this session then unbinds any objects bound to it.

session.invalidate();

return ResponseEntity.ok("Déconnexion réussie");

}

@GetMapping(path = "/visites")

public ResponseEntity<Integer> visites(HttpSession session) {

// Récupère le compteur de visites de la session

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

if (visites == null) {

// Si le compteur de visites n'existe pas encore, initialisez-le à 1

visites = 1;

} else {

// Sinon, incrémente le compteur de visites

visites++;

}

// Met à jour le compteur de visites dans la session

session.setAttribute("visites", visites);

return ResponseEntity.ok(visites);

}

### Test

LOGIN :

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

Description générée automatiquement

LOGOUT :

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

Description générée automatiquement

VISITES : avant de se loger

Une image contenant texte, logiciel, Icône d’ordinateur, Page web

Description générée automatiquement

Après qu’on s’est connecté une fois :

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

Description générée automatiquement

## Documentation API avec Swagger

[Swagger UI](http://localhost:8080/swagger-ui/index.html#/controller/login)

Documenter une API peut souvent s'avérer être une tâche ardue et chronophage. Entre la nécessité de décrire minutieusement chaque endpoint, de préciser les paramètres acceptés, les types de réponses attendues, sans oublier les exemples de requêtes et les codes d'erreur, le processus peut rapidement devenir fastidieux.  
L'intégration de Swagger transforme cette corvée en une expérience bien plus agréable et efficace. Grâce à cet outil, la documentation de l'API devient automatique, interactive et surtout, amusante à créer. Non seulement Swagger génère une documentation claire et structurée, mais il offre également une interface utilisateur interactive qui permet de tester les endpoints en temps réel. Ce qui était autrefois une tâche longue et pénible devient soudainement rapide et engageante, permettant aux développeurs de se concentrer sur ce qu'ils font de mieux : coder.

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

Description générée automatiquement

## Hébergement

# Projet – DH Destinations

Dans mon projet le client se retrouve dans une page de voyages, dans laquelle il aura un choix entre plusieurs voyages et s’il voudra il a la possibilité d’en réserver un ou plusieurs, mais pour les réserver il doit s’inscrire.

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

### Use case client et use case Rest.

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

### Sequence System global entre les applications

## Conception à faire complétement avec EA -> à rendre uniquement le fichier EA

### Class Diagram complet avec les explications de chaque application

## Bases de données

### Modèles WorkBench MySQL

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

### Une descente de code client

## Implémentation de l'application <API Gateway>

### Une descente de code APIGateway

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

### Une descente de code de l'API REST

## Hébergement

## Installation du projet complet avec les 5 applications

## Tests de fonctionnement du projet

# Auto-évaluations et conclusions

## Auto-évaluation et conclusion de …

## Auto-évaluation et conclusion de …