1. Du 21.03.2024 au 03.05.2024

133 - Développer des applications WEB

Rapport personnel

Hoti Flamur

HOTI

Date de création : 21 mars 2024
Version 1 du : 06.05.24

Table des matières

HOTI

1 Introduction

HOTI

2 Tests technologiques selon les exercices

2.1 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 ?

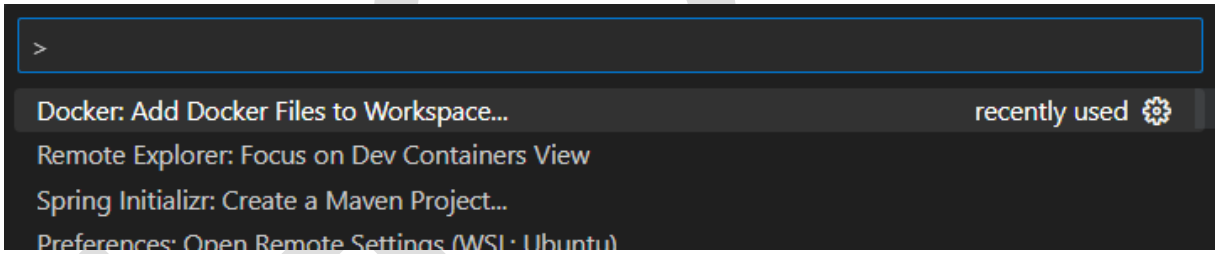
- Java en version 17.0.6

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

- Tomcat en version 10.0.16

2.2 Conteneurisation

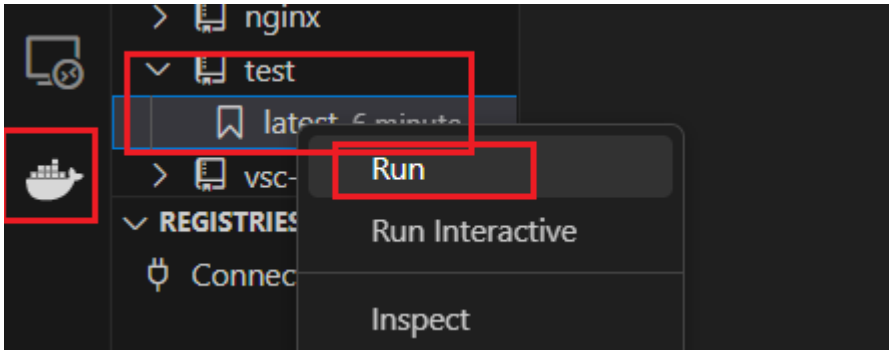
J'ai repris l'exercice précédent dans VSCode : ouvert le dossier gs-rest-service/complete qui contient l'application d'exemple.



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

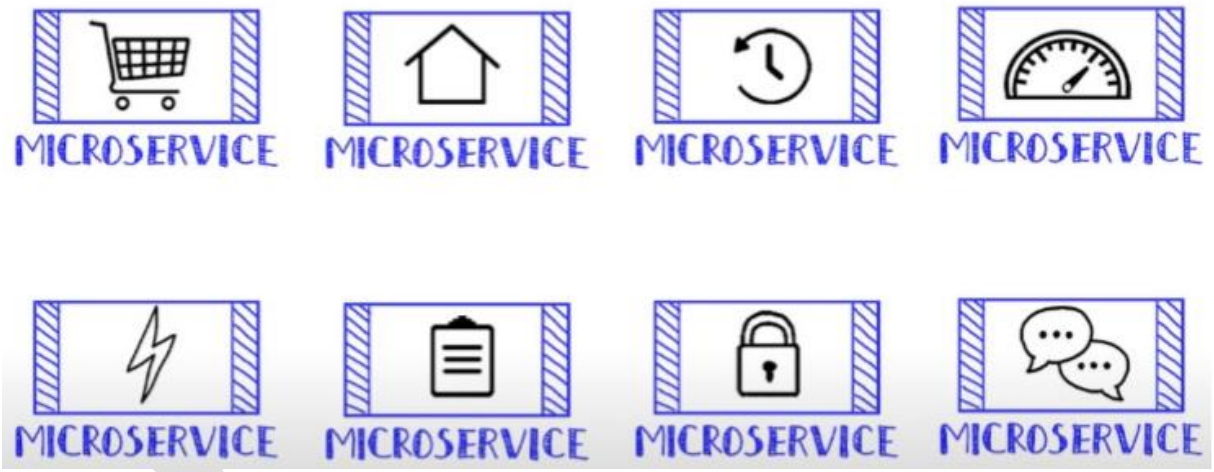
```
FROM openjdk:17-jdk-alpine
VOLUME /tmp
ARG JAVA_OPTS
ENV JAVA_OPTS=$JAVA_OPTS
COPY target/rest-service-complete-0.0.1-SNAPSHOT.jar
gsrestservice.jar
EXPOSE 8080
ENTRYPOINT exec java $JAVA_OPTS -jar gsrestservice.jar
# For Spring-Boot project, use the entrypoint below to reduce Tomcat
startup time.
#ENTRYPOINT exec java $JAVA_OPTS -
Djava.security.egd=file:/dev/./urandom -jar gsrestservice.jar
```

2.2.1 Création de l'image docker



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.



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

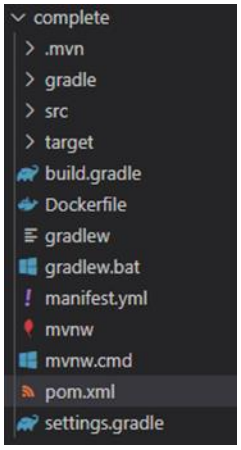
```
FROM openjdk:17-jdk-alpine
```

- Faire attention à la version du pom.xml

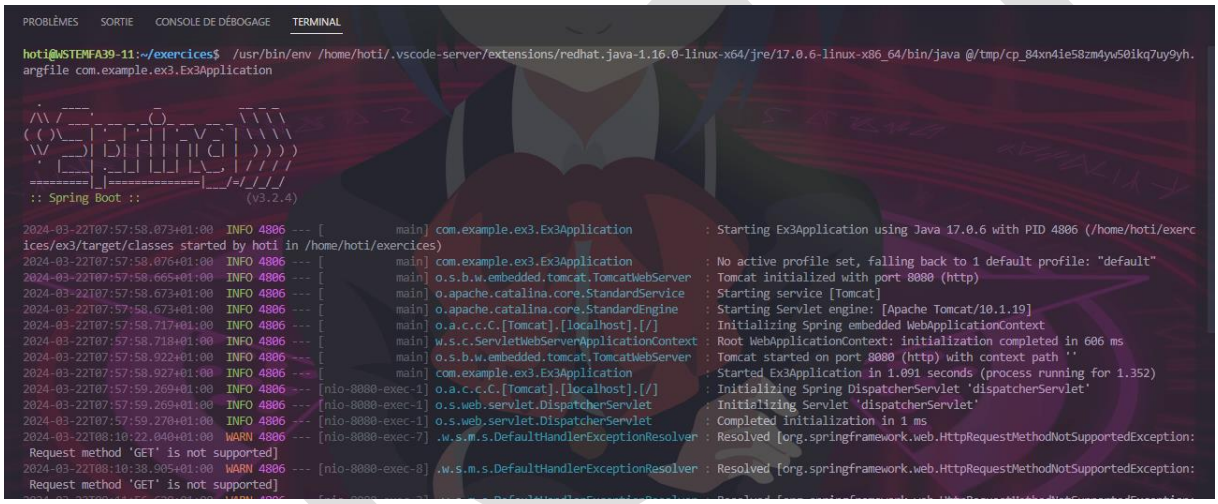
```
<properties>
  <java.version>17</java.version>
```

```
</properties>
```

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



2.3 Création d'un projet Spring Boot

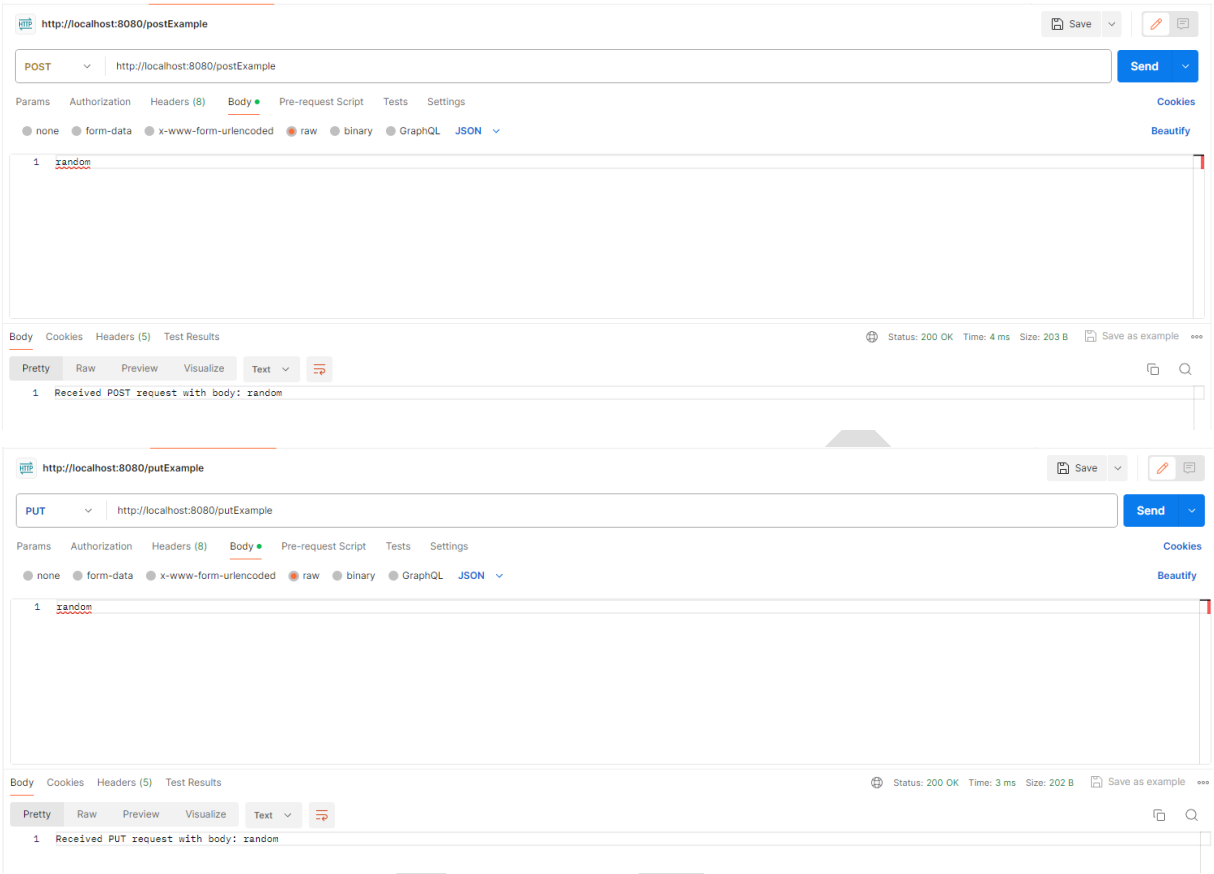


@RestController : Cette annotation indique que la classe est un contrôleur REST. Cela signifie que les méthodes de cette classe sont prêtes à traiter les requêtes HTTP entrantes.

@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.



2.4 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>
```

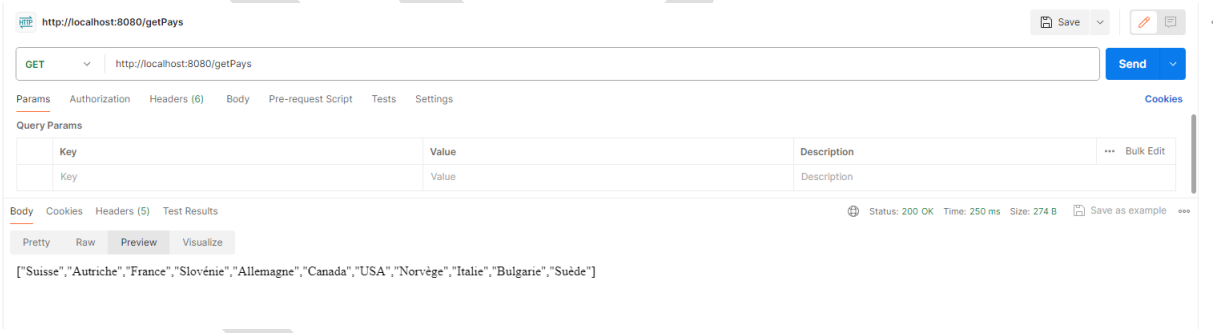
Pour se connecter à la base de données on a une classe dédiée :

```
  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;
    }
```



2.5 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.

2.6 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.

2.7 Gestion des sessions

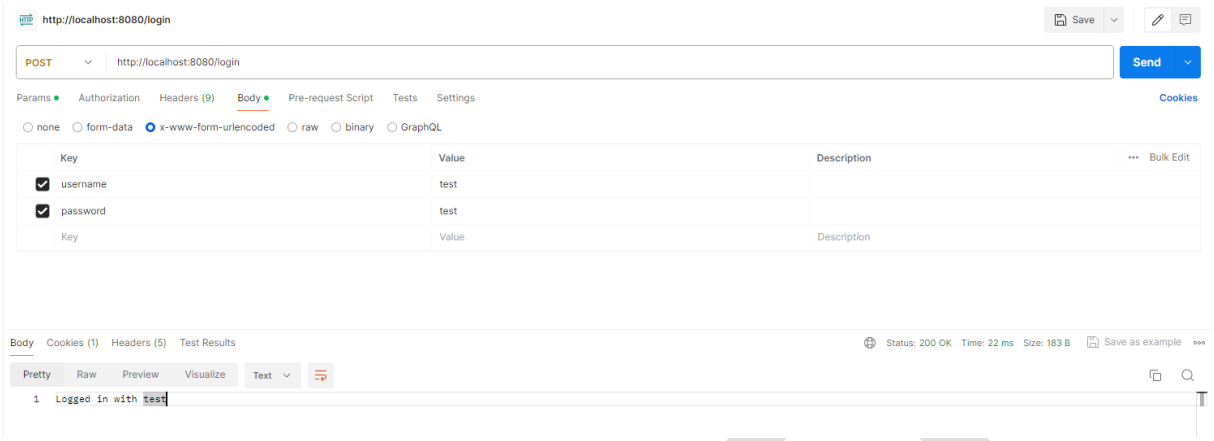
HttpSession est une interface dans Servlet qui fournit un moyen d'identifier un utilisateur à travers plusieurs requêtes HTTP. Lorsqu'un utilisateur accède à votre application web pour la première fois, le serveur crée une nouvelle session HTTP pour cet utilisateur. Pour chaque session, le serveur génère 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 :

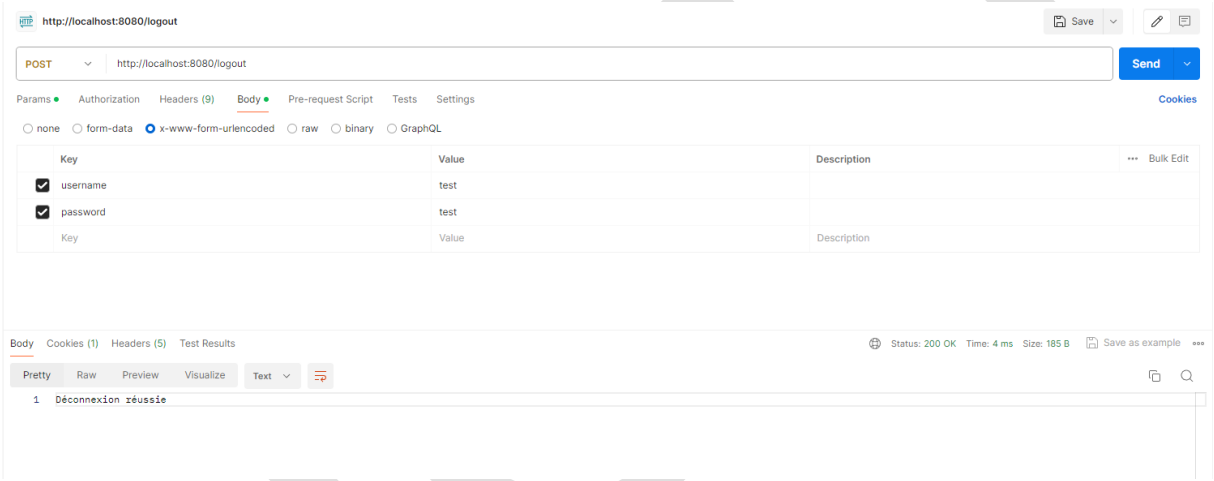
```
@PostMapping(path = "/login")
    public ResponseEntity<String> login(@RequestParam String
username, @RequestParam String password,
            HttpSession session) {

        // 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);
    }
```

2.7.1 Test

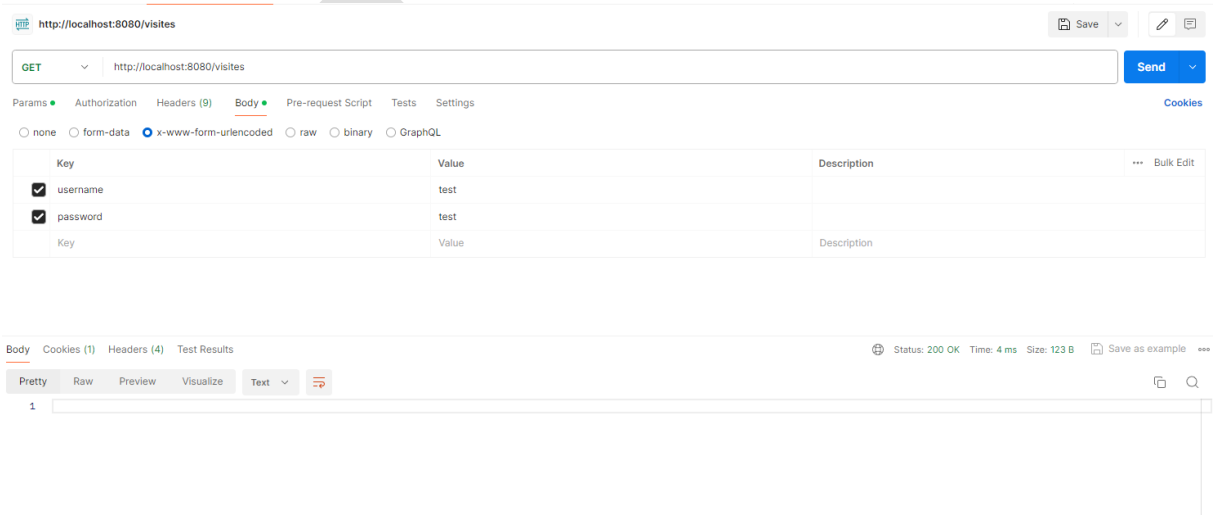
LOGIN :



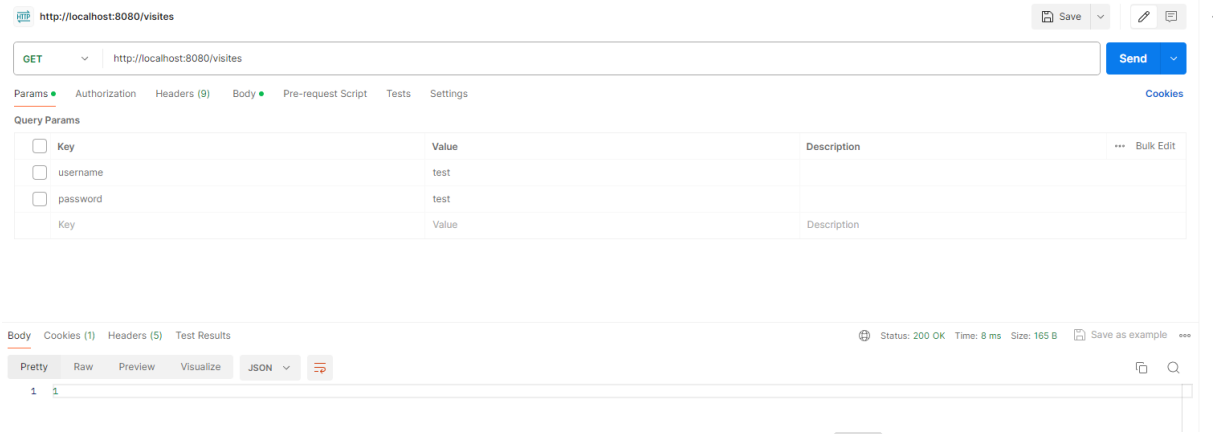
LOGOUT :



VISITES : avant de se loger



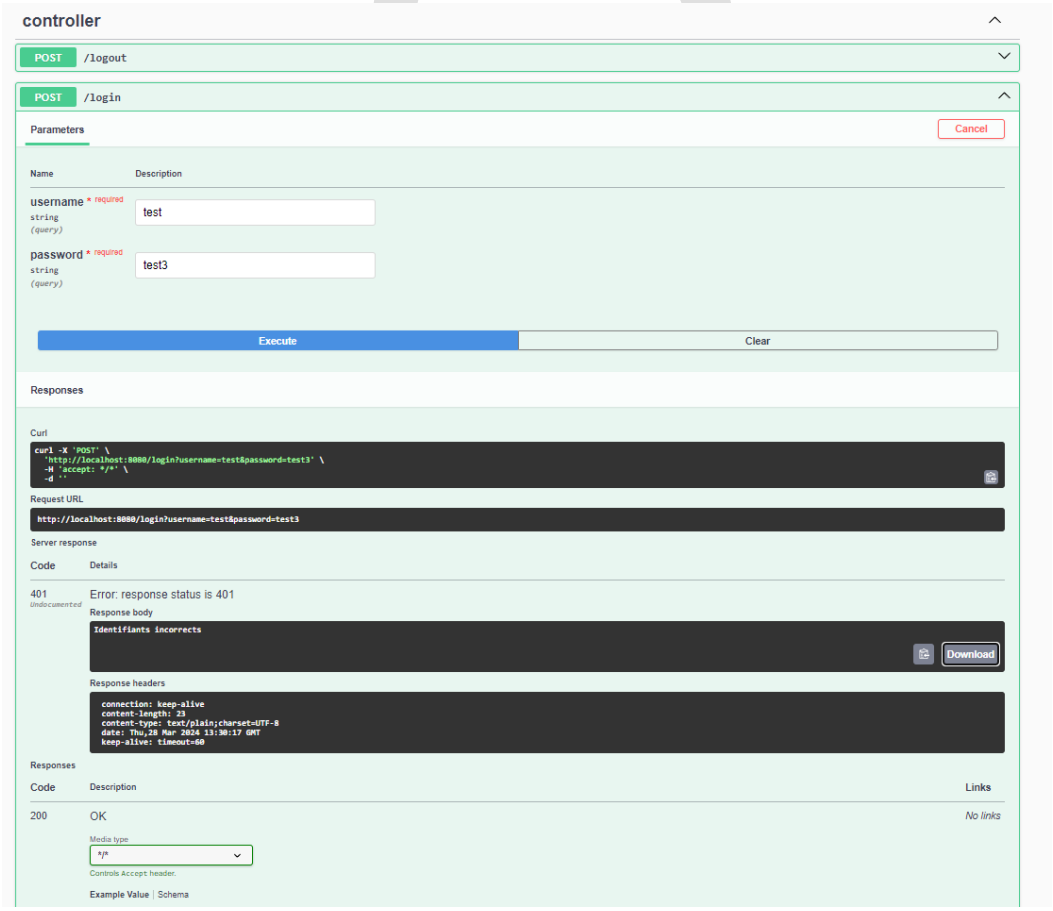
Après qu'on s'est connecté une fois :



2.8 Documentation API avec Swagger

Swagger UI

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.



2.9 Hébergement

Envoyer son image docker sur dockerhub :

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

Depuis VSCode, dans le terminal, faites la commande suivante et entrez le token :

- docker login -u username

Faites ensuite les commandes suivantes pour tag et push l'image sur dockerhub :

- docker tag nomImageLocale:latest username/nomRepo:latest
- docker push username/nomRepo:latest

```
Docker images
docker tag ex6:latest 133projetg03:latest
docker tag 133projetg03:latest imhoti/133groupe03:latest
docker push imhoti/133groupe03:latest
```

Ubuntu

```
docker run --name skibiditoilet -p 8080:8080 -d imhoti/133groupe03
```

HOT

3 Projet – DH Destinations

Dans mon projet le client se retrouve dans une page de voyages, dans laquelle il aura un choix entre plusieurs voyages.

S'il veut il a la possibilité d'en réserver un ou plusieurs s'il est connecté, il pourrait voir la liste de ses réservations.

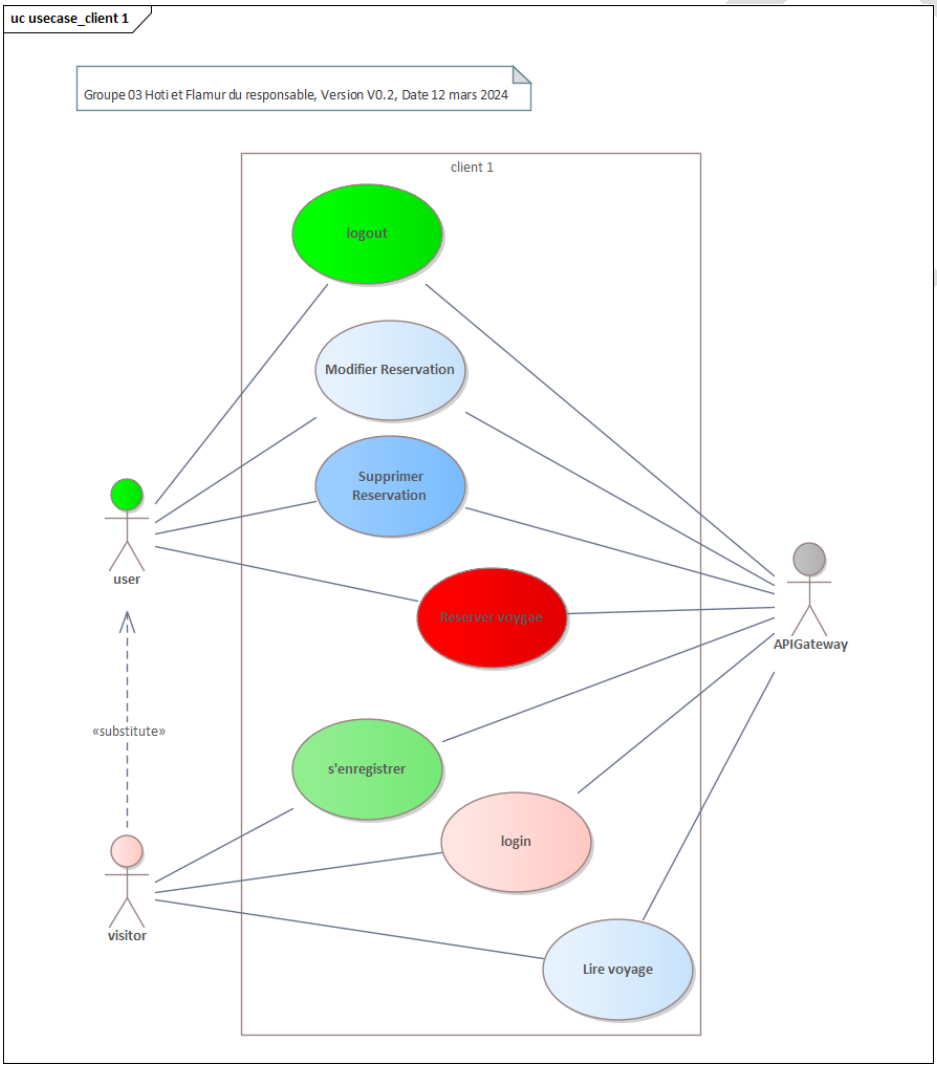
Il pourra supprimer cette réservation ou même la modifier en tente qu'utilisateur connecté.

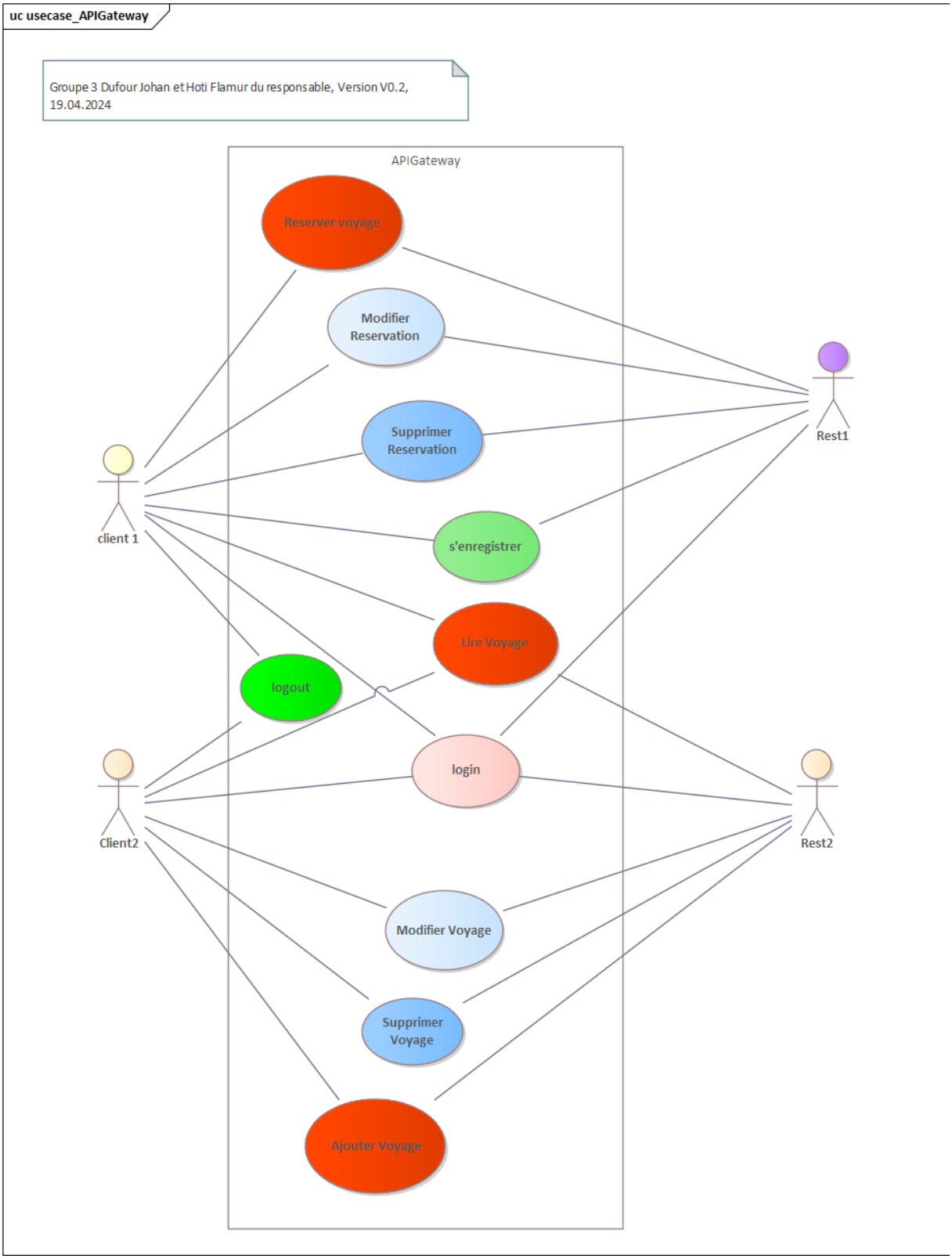
3.1 Analyse à faire complètement avec EA -> à rendre uniquement le fichier EA

3.1.1 Use case client et use case Rest.

Le client pourra visualiser les voyages puis s'il se connecte peux en réserver, modifier et supprimer.

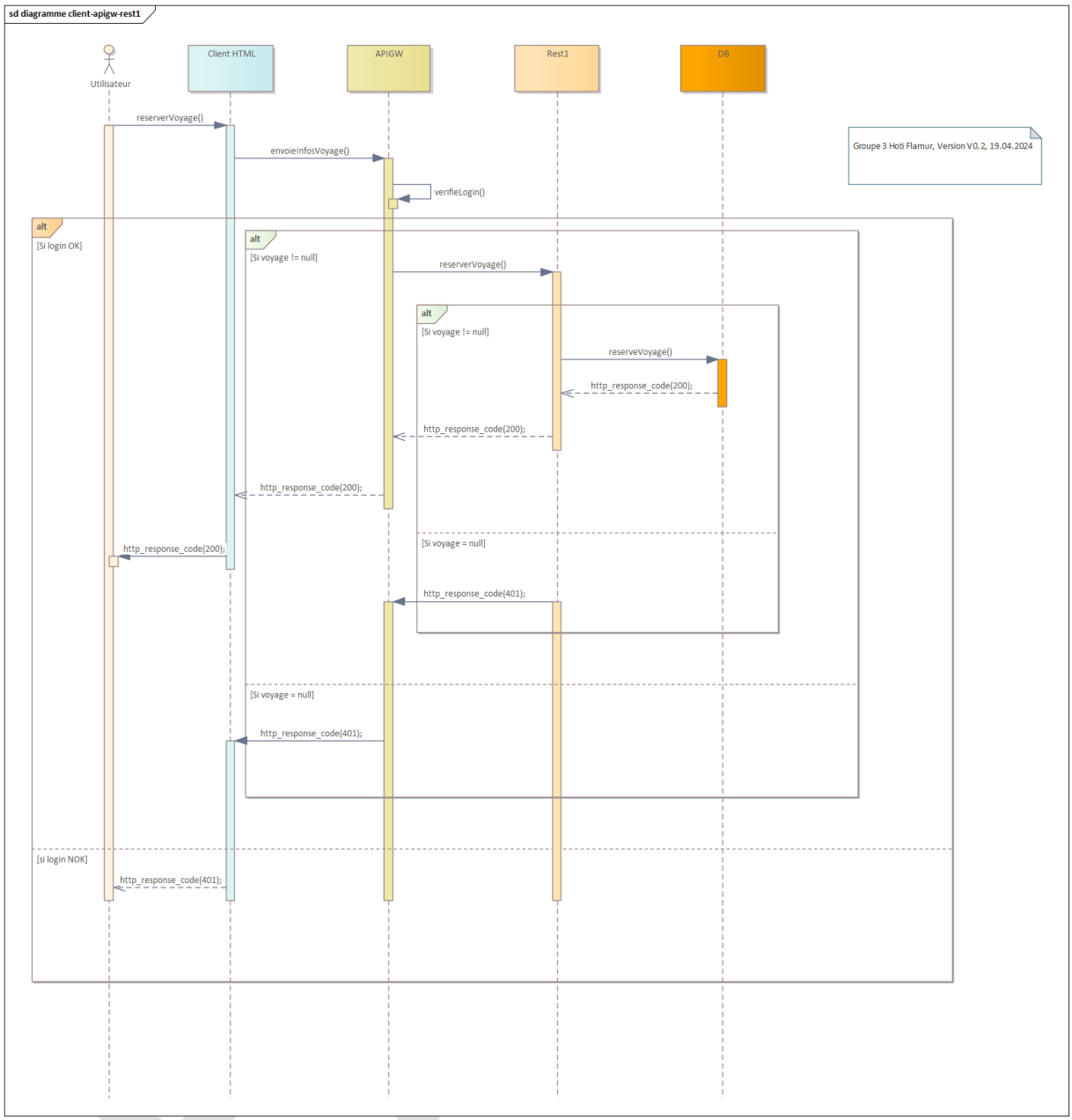
S'il n'a pas de compte il peut également s'enregistrer.





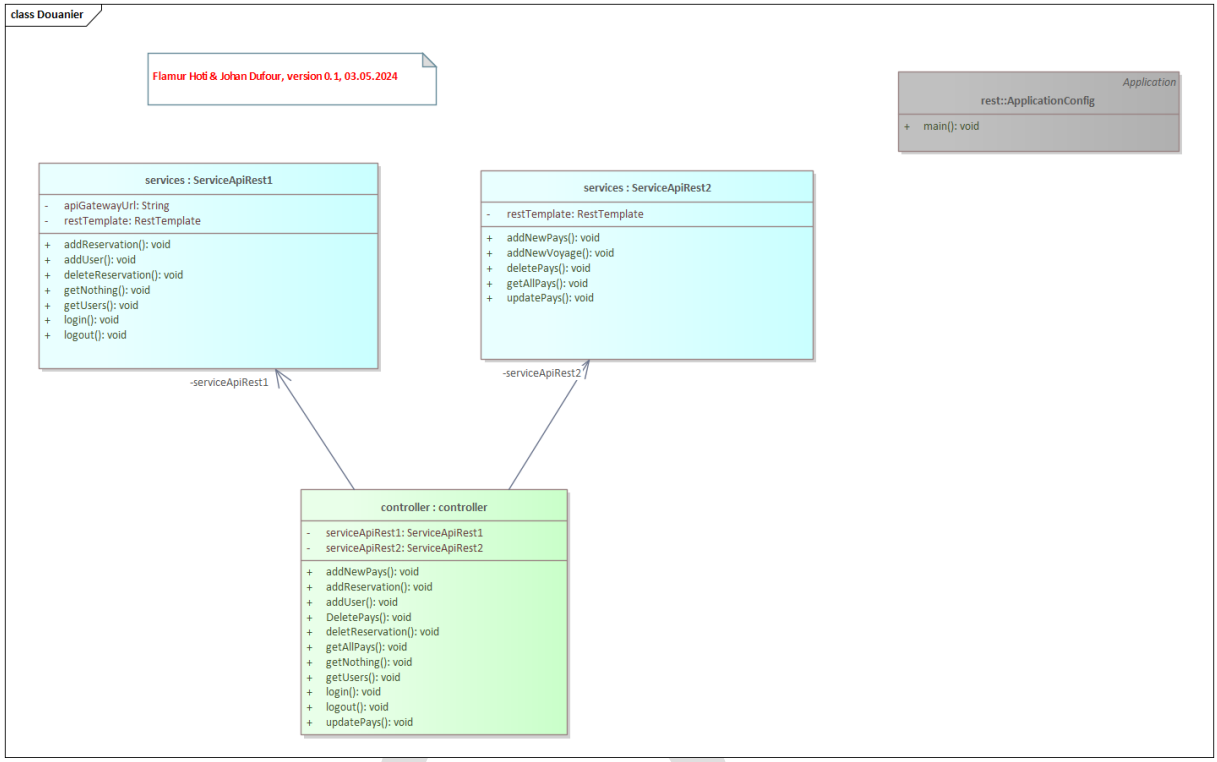
3.1.2 Séquence System global entre les applications

Dans ce diagramme de séquence j'ai décidé de représenter l'action réserver un voyage.

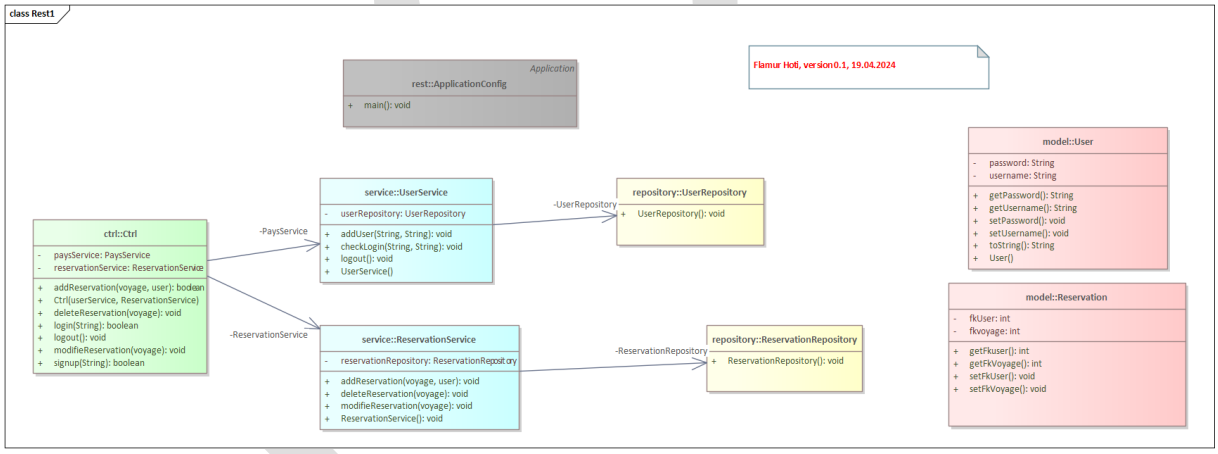

sd diagramme client-apigw-rest1
Utilisateur
Client HTML
APIGW
Rest1
DB
Groupe 3 Hoti Flamur, Version V0.2, 19.04.2024
reserverVoyage()
envoieInfosVoyage()
verifieLogin()
alt
[Si login OK]
alt
[Si voyage != null]
reserverVoyage()
alt
[Si voyage != null]
reserveVoyage()
http_response_code(200);
http_response_code(200);
http_response_code(200);
http_response_code(200);
[Si voyage = null]
http_response_code(401);
[Si voyage = null]
http_response_code(401);
[si login NOK]
http_response_code(401);

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

3.2.1 Class Diagram ApiGateway



3.2.2 Class Diagram complet avec les explications de chaque application



Model : User :

Contient des informations sur l'utilisateur.

- Attributs: password, username.
- Méthodes : getUserPassword(), getUserName(), setUserPassword(), setUserName().

Model : Réservation :

Gère les détails de la réservation.

- Attributs : date, commentaire, id_user, prix, place, id_reservation.

- Méthodes : getCommentaire(), getDate(), getId_reservation(), setId_reservation(), setDate(), setPrix().

Rest : ApplicationConfig :

- Configuration pour une application RESTful.

ctrl : Ctrl :

- Contrôleur pour les requêtes HTTP.

service::UserService :

- Logique métier liée aux utilisateurs.
- Interagit avec repository::UserRepository.

service::ReservationService :

- Logique métier pour les réservations.
- Méthodes : pour créer, supprimer, modifier les réservations.

repository::UserRepository :

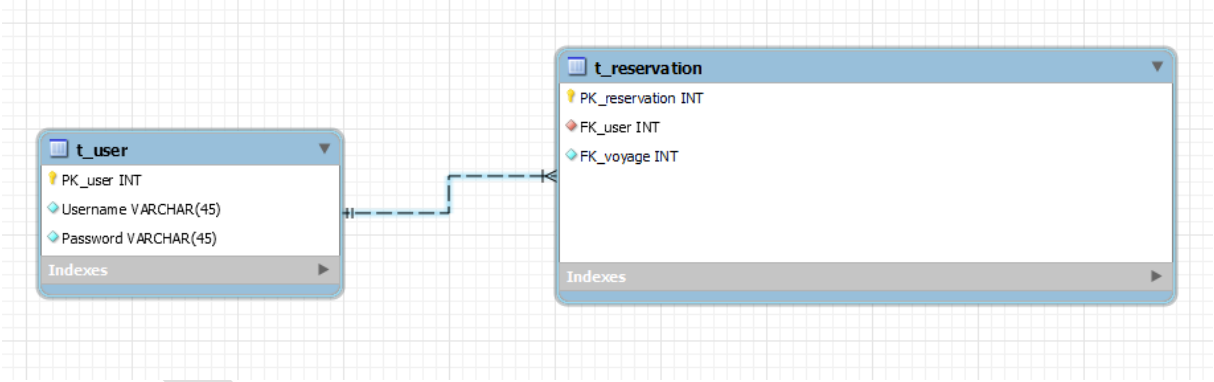
- Interface pour les opérations CRUD sur les données utilisateur.

repository::ReservationRepository :

- Interface pour les opérations CRUD sur les données de réservation.

3.3 Bases de données

3.3.1 Modèles WorkBench MySQL



3.4 Implémentation de l'application <Le client Ap1>

3.4.1 Une descente de code client

Depuis le contrôleur de la page login pour ajouter un utilisateur :

```
butSignUp.click(function (event) {
        var username = document.getElementById("txtLogin").value;
        var password = document.getElementById("password").value;

        if (username != "" && password != "") {
            signUp(username, password, signUpSuccess,
CallbackError);
        } else {
            alert("Erreur lors de l'enregistrement");
        }
```

```
    });
```

Cette méthode est lancée dès qu'on clique sur le buton s'enregistrer, donc les services http sont chargé et la requête est envoyé :

```
/**
 * Fonction permettant d'ajouter un utilisateur.
 * @param {string} username Le nom d'utilisateur à ajouter.
 * @param {string} password Le mot de passe associé à l'utilisateur.
 * @param {function} successCallback Fonction de callback lors du
retour avec succès de l'appel.
 * @param {function} errorCallback Fonction de callback en cas
d'erreur.
 */
function addUser(username, password, successCallback, errorCallback)
{
    $.ajax({
        type: "POST",
        dataType: "json",
        data: {
            username: username,
            password: password
        },
        url: BASE_URL + "addUser",
        xhrFields: {
            withCredentials: true
        },
        success: successCallback,
        error: errorCallback,
    });
}
```

L'url sera :

```
var BASE_URL = "https://backend-3.emf4you.ch/";
```

3.5 Implémentation de l'application <API Gateway>

3.5.1 Une descente de code APIGateway

Depuis le contrôleur j'appelle la méthode « /addUser » :

```
@PostMapping("/addUser")
    public ResponseEntity<String> addUser(@RequestParam String
username,
            @RequestParam String password) {
        try {
            // Ajoute l'utilisateur en utilisant le service
approprié
            serviceApiRest1.addUser(username, password);

            // Crée un objet JSON pour la réponse de succès
            String successMessage = "Utilisateur ajouté avec
succès";
            return ResponseEntity.ok().body("{\"message\": \"" +
successMessage + "\"}");
        } catch (Exception e) {
```

```
            // Crée un objet JSON pour la réponse d'erreur
            String errorMessage = "Erreur lors de l'ajout de
l'utilisateur : " + e.getMessage();
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("{\"error\": \""
+ errorMessage + "\"}");
        }
    }
```

Et depuis mes services je vais faire une requête vers l'apiREST1 pour ajouter un utilisateur :

```
    public ResponseEntity<String> addUser(@RequestParam String
username, @RequestParam String password) {
        String url = apiGatewayUrl + "/addUser";

        // User userDTO = new User(username, password);

        LinkedMultiValueMap<String, String> params = new
LinkedMultiValueMap<>();
        params.add("username", username);
        params.add("password", password);

        HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
        HttpEntity<MultiValueMap<String, String>> requestEntity =
new HttpEntity<>(params, headers);

        try {
            ResponseEntity<String> response =
restTemplate.postForEntity(url, requestEntity, String.class);
            // Vérifier la réponse
            if (response.getStatusCode().is2xxSuccessful()) {
                // Traitement réussi
                return ResponseEntity.ok(response.getBody());
            } else {
                // Gérer les erreurs si nécessaire
                return
ResponseEntity.badRequest().body(response.getBody());
            }
        } catch (HttpClientErrorException.BadRequest ex) {
            return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("L'utilisateur "
+ username + " existe déjà !");
        }
    }
```

3.6 Implémentation de l'application <API élève1>

3.6.1 Une descente de code de l'API REST

Cette méthode du contrôleur permet l'inscription de nouveaux utilisateurs.

```
@PostMapping(path = "addUser")
```

```
    public ResponseEntity<String> addUser(@RequestParam String
username, @RequestParam String password) {
        try {
            // Hacher le mot de passe
            String hashedPassword =
passwordEncoder.hashPassword(password);

            // Ajouter l'utilisateur en utilisant le service
utilisateur avec le mot de
            // passe haché et le statut d'administrateur
            userService.addUser(username, hashedPassword);

            return ResponseEntity.ok("Utilisateur ajouté avec
succès");

        } catch (IllegalArgumentException e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }
```

Le hachage du mot de passe avant le stockage garantit la sécurité des informations des utilisateurs.

```
public String hashPassword(String password) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-
256");
            byte[] hashBytes = digest.digest(password.getBytes());
            String hashedPassword =
Base64.getEncoder().encodeToString(hashBytes);
            return hashedPassword;
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException("Algorithme de hachage
non disponible", e);
        }
    }
```

Puis elle appelle les services des users :

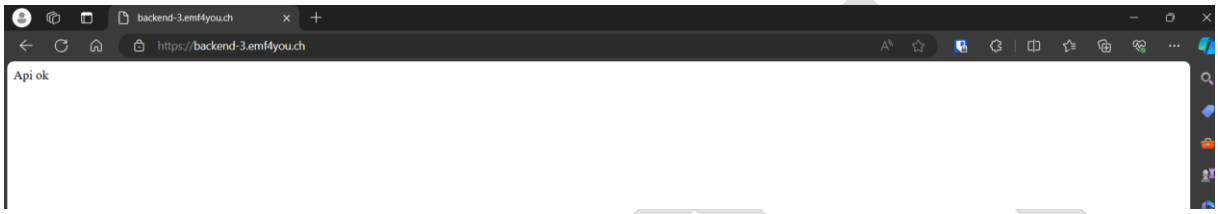
```
  @Transactional
    public User addUser(String username, String password) {
        // Vérifier si l'utilisateur existe déjà
        if (userRepository.findByUsername(username) != null) {
            throw new IllegalArgumentException("L'utilisateur existe
déjà");
        }

        // Créer un nouvel utilisateur avec le nom d'utilisateur et
le mot de passe
        // fournis
        User newUser = new User();
        newUser.setUsername(username);
        newUser.setPassword(password);

        // Enregistrer le nouvel utilisateur dans la base de données
        return userRepository.save(newUser);
    }
```

Qui ajoute l'user dans la base de donnée grâce au repository :

```
public interface UserRepository extends CrudRepository<User, Long> {
    User findByUsername(String username);

    User save(User user);

    User findByid(Integer id);
}
```

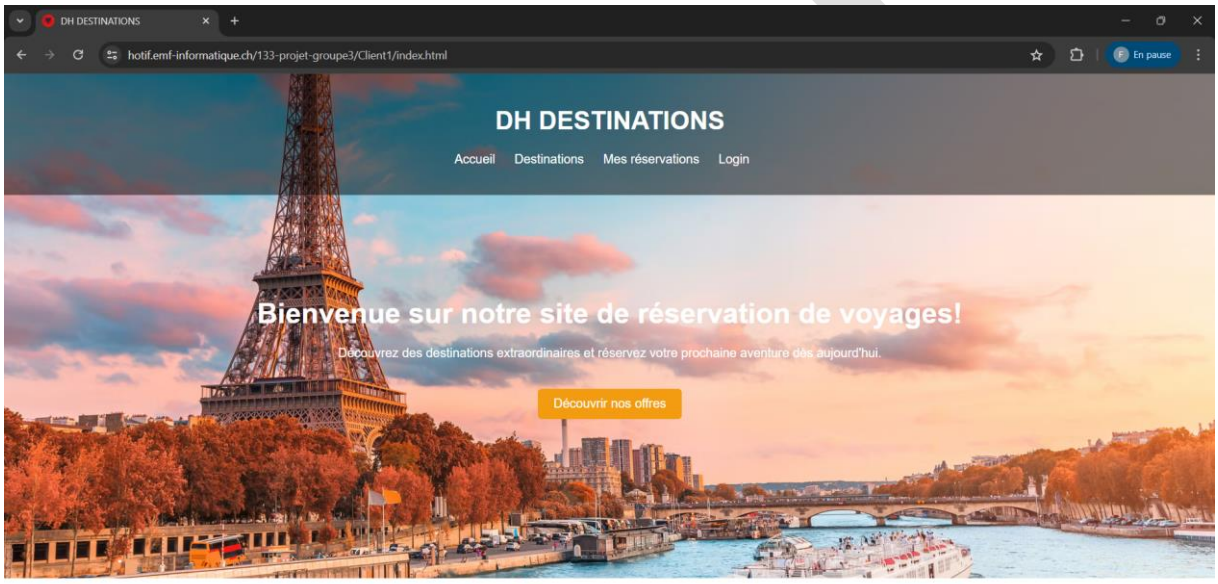
3.7 Hébergement

L'apiREST1 et l'apiGateway sont hébergé sur le serveur, on peut le voir :



3.8 Installation du projet complet avec les 5 applications

https://hotif.emf-informatique.ch/133-projet-groupe3/Client1/index.html

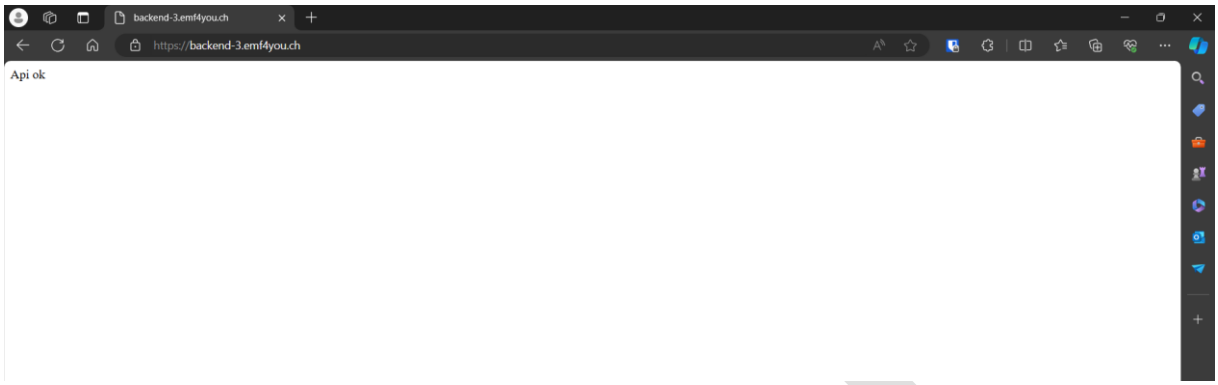


3.9 Tests de fonctionnement du projet

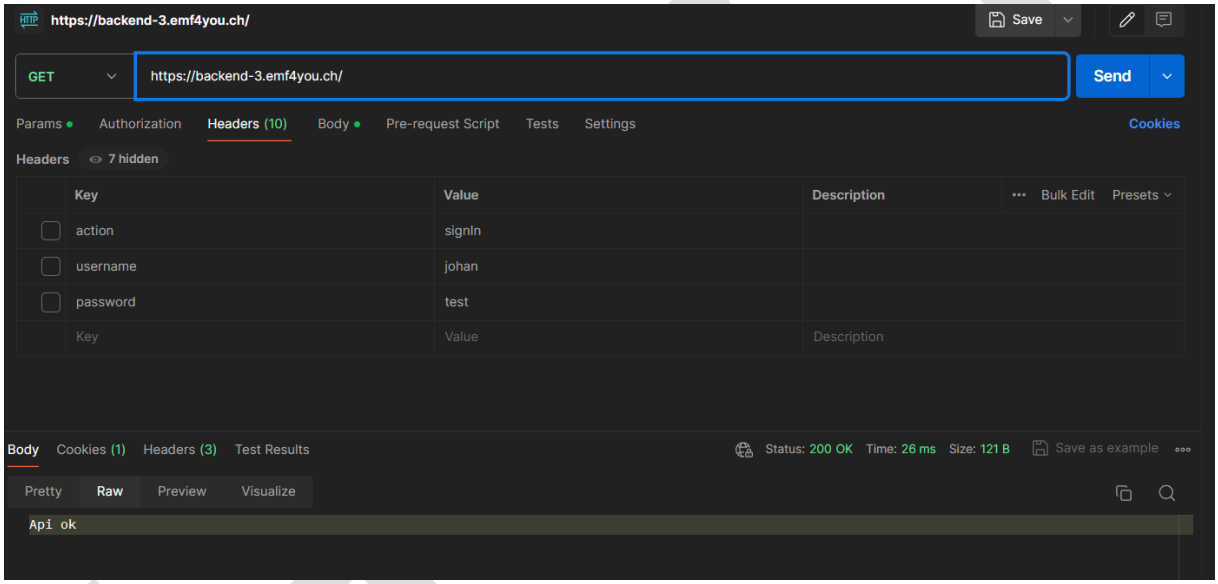
Test en local depuis Postman sur l'apiGateway ou apiREST1 :

Protocole de test				
Utilisateur	**Cas**	**Tests**	**Attendu**	**Resultat**
Visiteur	Connexion	Avec login existant et mot de passe correct.	On est connecté en tant qu'utilisateur	OK
	Voir les voyages dans la page destinations	En accedant simplement au site et en cliquant sur destinations.	Une liste de voitures s'affiches.	OK
	Connexion	Avec login et mot de passe vide ou faux	Pas possible.	OK
	S'enregistrer	Il peut s'enregistrer avec un password et username s'il n'existe pas	Il est enregistrer et il pourra se connecter.	OK
Client connecté	Ajouter une reservation	L'utilisateur ajoute une reservation.	L'operation c'est bien passée.	OK
	Suppression reservation	L'utilisateur supprime une reservation existant.	L'operation c'est bien passée.	OK
	Modification reservation	L'utilisateur modifie une reservation.	L'operation c'est bien passée.	NOK
	Déconnexion	L'utilisateur se déconnecte.	La page d'accueil s'affiche.	OK

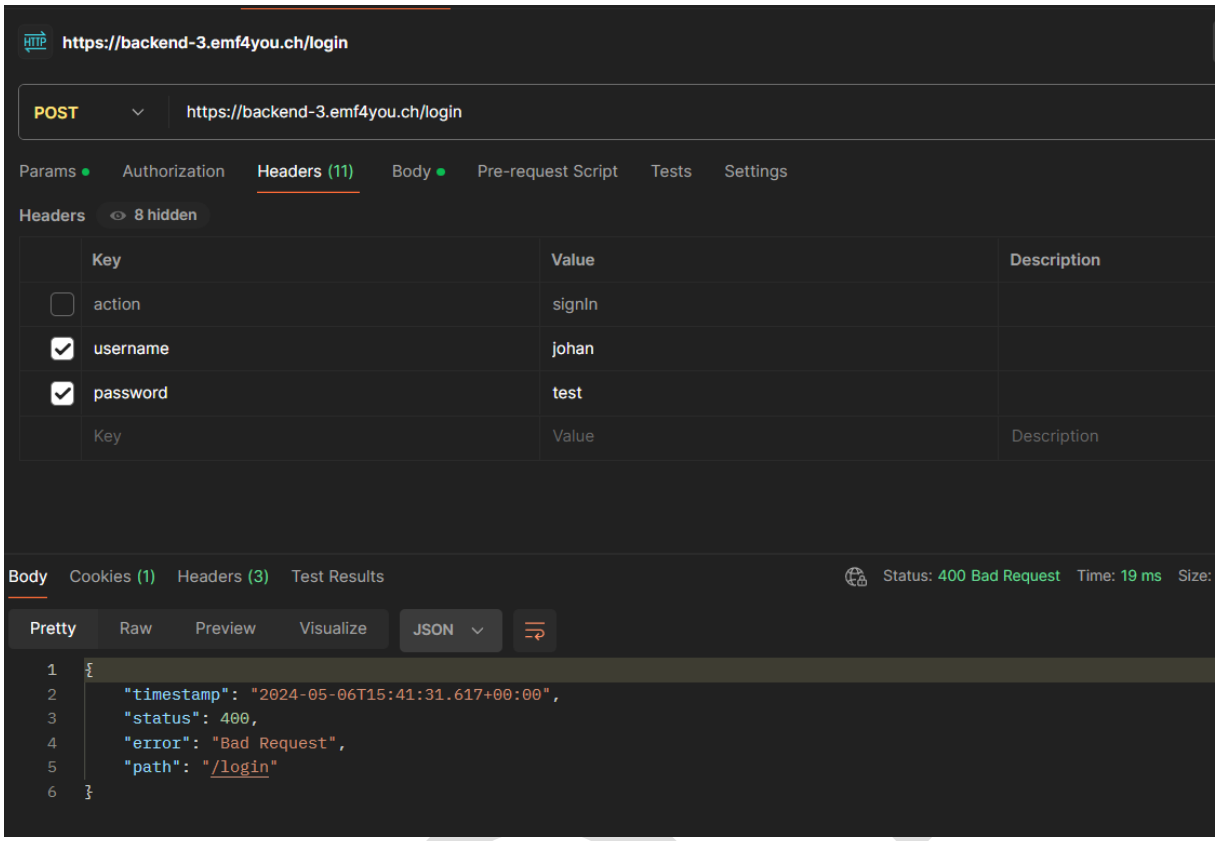
Cependant je n'ai pas pu tester depuis la partie du client, on peut voir que l'apiGateway et l'apiREST1 sont hébergés :



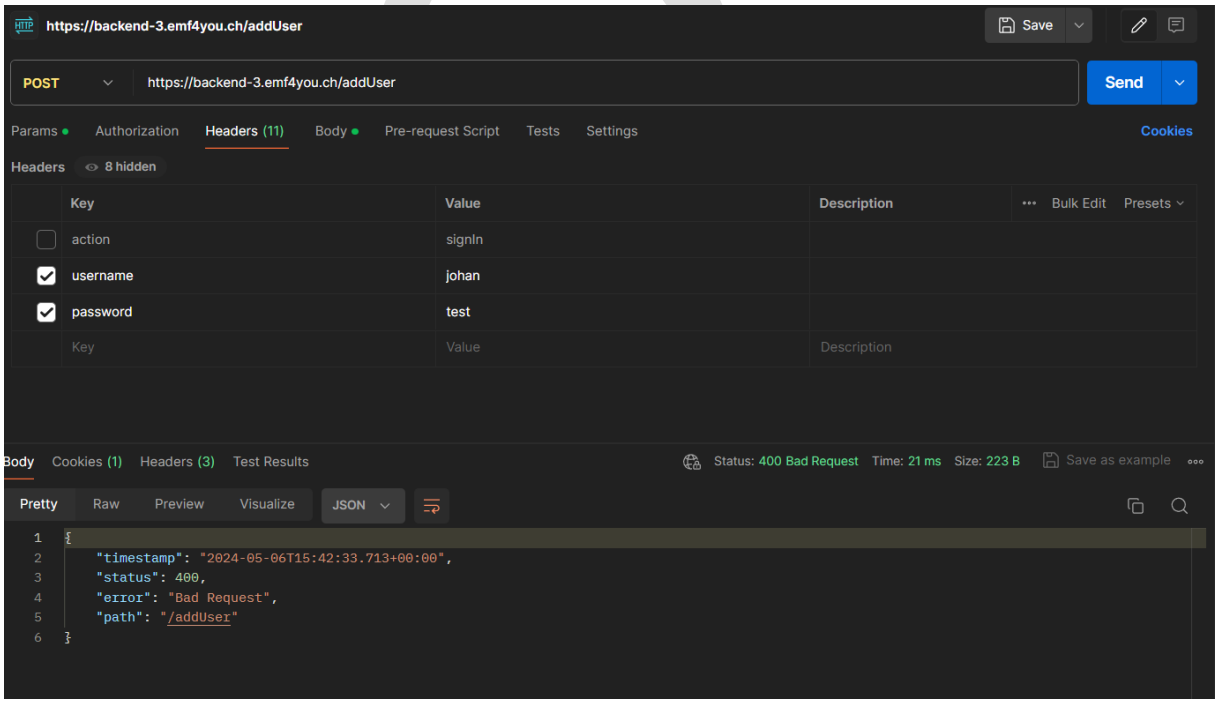
Mais si depuis Postman j'essaie de faire un login ou ajouter un user ça ne fonctionne pas et je ne comprends pas la raison car en classe avant de faire l'hébergement sur le serveur j'ai tout testé et tout fonctionné à la perfection.



Login :


https://backend-3.emf4you.ch/login
POST
https://backend-3.emf4you.ch/login
Params
Authorization
Headers (11)
Body
Pre-request Script
Tests
Settings
Headers
8 hidden
Key
Value
Description
action
signIn
username
johan
password
test
Key
Value
Description
Body
Cookies (1)
Headers (3)
Test Results
Status: 400 Bad Request
Time: 19 ms
Size:
Pretty
Raw
Preview
Visualize
JSON
{
"timestamp": "2024-05-06T15:41:31.617+00:00",
"status": 400,
"error": "Bad Request",
"path": "/login"
}

addUser :


https://backend-3.emf4you.ch/addUser
Save
POST
https://backend-3.emf4you.ch/addUser
Send
Params
Authorization
Headers (11)
Body
Pre-request Script
Tests
Settings
Cookies
Headers
8 hidden
Key
Value
Description
Bulk Edit
Presets
action
signIn
username
johan
password
test
Key
Value
Description
Body
Cookies (1)
Headers (3)
Test Results
Status: 400 Bad Request
Time: 21 ms
Size: 223 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"timestamp": "2024-05-06T15:42:33.713+00:00",
"status": 400,
"error": "Bad Request",
"path": "/addUser"
}

4 Auto-évaluations et conclusions

4.1 Auto-évaluation et conclusion

Le module je l'ai bien aimé, j'ai appris beaucoup de nouveaux outils pour le développement, par contre je trouve qu'on n'a pas eu assez du temps pour le projet vue qu'on a perdu pas mal d'heures durant les 5 semaines mais ça a été un bon défit pour se mettre en jeux

HOTI