

133 - Réaliser des applications web en session-handling

Richoz Matteo

Rapport personnel

Version 1 du 06.05.2024 20:41:00

Module du : 21.03.2024

Au : 03.05.2024

Date de création : 21.03.2024 14:56:00

Table des matières

1	Introduction et contexte du projet	4
2	Tests technologiques selon les exercices	5
2.1	Installation et Hello World.....	5
2.1.1	Observez la console pour comprendre comment le projet est lancé et comment il tourne ?5	
2.1.2	C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ?	5
2.1.3	Y a-t-il un serveur web ?.....	5
2.1.4	Quelle version de java est utilisée ?.....	5
2.1.5	S'il y a un serveur web, quelle version utilise-t-il ?.....	5
2.2	Conteneurisation	5
2.2.1	Pourquoi faire un container pour une application Java ?	5
2.2.2	Y a-t-il un serveur web ? Ou se trouve-t-il ?	5
2.2.3	A quoi faut-il faire attention (pensez aux versions !) ?	6
2.3	Création d'un projet Spring Boot	6
2.3.1	Quelles sont les annotations utilisée (commencent par @) dans votre controller ?	6
2.4	Connexion à la DB JDBC.....	6
2.4.1	Créer une base de données dans un container	6
2.5	Connexion à la DB JPA.....	7
2.5.1	À quoi sert l'annotation @Autowired dans vos controlleur pour les Repository ?	7
2.5.2	A quoi sert l'annotation @ManyToOne dans l'entité skieur ?	7
2.5.3	Sur la même ligne, quel FetchType est utilisé et pourquoi, réessayer avec le FetchType LAZY et faites un getSkieur.	7
2.6	Connexion à la DB JPA avec DTO	7
2.6.1	Pourquoi dans ce cas, on retrouve un SkierDTO et pas de PaysDTO ?	7
2.6.2	Expliquez dans votre rapport à quoi servent les model, les repository, les dto, les services et les controleurs en vous basant sur le code donné.	7
2.7	Gestion des sessions	8
2.8	Documentation API avec Swagger	10
2.9	Hébergement.....	Error! Bookmark not defined.
3	Analyse à faire complètement avec EA -> à rendre uniquement le fichier EA.....	12
3.1	Use case client et use case Rest.....	12
3.1.1	Use case client 1	12
3.1.2	Use case client 2	13
3.1.3	Use case API Gateway	14

3.1.4	Use case API Rest 1	15
3.1.5	Use case API Rest 2	16
3.2	Sequence System global entre les applications.....	17
4	Conception à faire complètement avec EA -> à rendre uniquement le fichier EA	18
4.1	Diagramme class Rest 1.....	18
4.2	Diagramme class Rest 2.....	19
4.3	Diagramme class Rest 2 après implémentation	19
4.4	Diagramme class API Gateway.....	20
5	Bases de données.....	21
5.1	Modèles WorkBench MySQL	21
6	Implémentation des applications <Le client Ap1> et <Le client Ap2>	22
6.1	Une descente de code client	22
7	Implémentation de l'application <API Gateway>	24
7.1	Une descente de code APIGateway	24
8	Implémentation des applications <API élève1> et <API élève2>.....	25
8.1	Une descente de code de l'API REST	25
9	Hébergement.....	26
10	Installation du projet complet avec les 5 applications.....	27
11	Tests de fonctionnement du projet	29
12	Auto-évaluations et conclusions.....	33
12.1	Auto-évaluation	33
12.2	Conclusion	33

1 Introduction et contexte du projet

Le projet se nomme **YouQuiz**. Il permettra à des utilisateurs connectés de remplir des quiz ou d'en créer pour les administrateurs.

Premièrement il y aura la partie qui permettra de créer des quiz pour les utilisateurs connectés sur celui-ci. Ensuite il y aura la deuxième partie qui permet à un utilisateur connecté de remplir des quiz. Un utilisateur peut aussi liker les quiz, ceux-ci seront ainsi triés par leur nombre de likes.

Les quiz consistent en plusieurs questions qui ont plusieurs réponses possibles. Le choix de réponse peut être unique, multiples ou vrai ou faux.

David va s'occuper de la partie permettant la création de quiz et moi de la partie permettant de remplir les sites.

2 Tests technologiques selon les exercices

2.1 Installation et Hello World

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

```
richozm@STEF-A39-17:~/gs-rest-service$ /usr/bin/env /usr/lib/jvm/java-17-openjdk-amd64/bin/java @/tmp/cp_6rzc01w5nsxk87jyiet8vjyxq.argfile com.example.restservice.RestServiceApplication

:: Spring Boot ::
(v3.2.0)

2024-03-21T15:58:29.493+01:00 INFO 12425 --- [main] c.e.restservice.RestServiceApplication : Starting RestServiceApplication using Java 17.0.10 with PID 12425 (/home/richozm/gs-rest-service/complete/target/classes started by richozm in /home/richozm/gs-rest-service)
2024-03-21T15:58:29.495+01:00 INFO 12425 --- [main] c.e.restservice.RestServiceApplication : No active profile set, falling back to 1 default profile: "default"
2024-03-21T15:58:30.075+01:00 INFO 12425 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-03-21T15:58:30.084+01:00 INFO 12425 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-03-21T15:58:30.084+01:00 INFO 12425 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.16]
2024-03-21T15:58:30.128+01:00 INFO 12425 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-03-21T15:58:30.129+01:00 INFO 12425 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 597 ms
2024-03-21T15:58:30.349+01:00 INFO 12425 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-03-21T15:58:30.353+01:00 INFO 12425 --- [main] c.e.restservice.RestServiceApplication : Started RestServiceApplication in 1.094 seconds (process running for 1.326)
2024-03-21T15:58:42.028+01:00 INFO 12425 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-03-21T15:58:42.029+01:00 INFO 12425 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-03-21T15:58:42.029+01:00 INFO 12425 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
```

Pour lancer le projet, on utilise Spring Boot par le port 8080

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

On utilise Maven pour build notre projet

2.1.3 Y a-t-il un serveur web ?

Oui il y a un serveur Apache Tomcat

2.1.4 Quelle version de java est utilisée ?

On utilise la version 17 de java

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

Apache Tomcat version 10.1.16

2.2 Conteneurisation

2.2.1 Pourquoi faire un container pour une application Java ?

- Portabilité
- Facilité de déploiement
- Gestion des dépendances
- Isolation et sécurité

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

Oui, Tomcat, compilé dans le jar.

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

Changé la version de java dans le pom.xml et dans le Dockerfile.

2.3 Création d'un projet Spring Boot

2.3.1 Quelles sont les annotations utilisées (commencent par @) dans votre controller ?

- **@RestController** : Cette annotation indique que la classe est un contrôleur REST, ce qui signifie qu'elle est responsable de la gestion des requêtes HTTP et des réponses en utilisant les principes RESTful.
- **@GetMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP GET aux méthodes de gestion appropriées dans le contrôleur. Elle spécifie que la méthode annotée répondra uniquement aux requêtes HTTP GET.
- **@PostMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP POST aux méthodes de gestion appropriées dans le contrôleur. Elle spécifie que la méthode annotée répondra uniquement aux requêtes HTTP POST.
- **@PutMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP PUT aux méthodes de gestion appropriées dans le contrôleur. Elle spécifie que la méthode annotée répondra uniquement aux requêtes HTTP PUT.
- **@RequestParam** : Cette annotation est utilisée pour extraire les paramètres de requête de l'URL et les associer aux paramètres de la méthode dans le contrôleur. Elle permet de récupérer les valeurs des paramètres de requête HTTP dans le contrôleur.
- **@RequestBody** : Cette annotation est utilisée pour mapper le corps de la requête HTTP à un objet Java dans le contrôleur. Elle permet de désérialiser automatiquement le corps de la requête JSON (ou XML) en un objet Java correspondant.

2.4 Connexion à la DB JDBC

2.4.1 Créer une base de données dans un container

Pour créer une nouvelle db dans un container, il faut faire ces commandes :

```
#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
```

2.5 Connexion à la DB JPA

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

L'annotation @Autowired dans Spring est utilisée pour l'injection automatique des dépendances. Cela signifie que Spring va chercher et instancier automatiquement le bean qui correspond à la dépendance déclarée.

2.5.2 À quoi sert l'annotation @ManyToOne dans l'entité skieur ?

L'annotation @ManyToOne est utilisée pour établir une relation de plusieurs à un entre deux entités dans votre modèle JPA.

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

Le FetchType est EAGER, pour aller chercher le pays directement. Si j'utilise le FetchType Lazy j'obtiens une erreur en transformant en JSON les skieurs car le pays n'est pas défini.

2.6 Connexion à la DB JPA avec DTO

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

Car Pays n'a pas de fk et donc pas besoin de fetch d'autres objets.

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

- Model : Les modèles représentent les entités de votre domaine d'application. Dans ce cas, Skieur et Pays sont des modèles qui représentent les skieurs et les pays dans votre application.
- Repository : Les repositories sont des interfaces qui permettent d'interagir avec la base de données. Ils fournissent des méthodes pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur les entités. Par exemple, SkieurRepository est une interface qui fournit des méthodes pour interagir avec les données de Skieur dans la base de données.
- DTO (Data Transfer Object) : Les DTO sont des objets qui encapsulent les données qui doivent être transférées entre les différentes couches de l'application. Par exemple, SkieurDTO est un objet qui contient les données d'un Skieur qui doivent être transférées de la couche de service à la couche de contrôleur.

- **Service** : Les services contiennent la logique métier de l'application. Ils coordonnent les opérations entre les différentes parties de l'application, comme l'interaction avec les repositories pour récupérer ou enregistrer des données.
- **Controller** : Les contrôleurs gèrent les interactions avec l'utilisateur. Ils reçoivent les requêtes de l'utilisateur, appellent les services appropriés pour traiter ces requêtes, et renvoient les réponses appropriées. Par exemple, dans votre code, Controller est une classe qui gère les requêtes HTTP et renvoie les réponses appropriées.

2.7 Gestion des sessions

Pour effectuer cet exercice, j'ai dû créer un nouveau controller avec 3 méthodes à l'intérieur. Une pour se login, une pour se logout et une pour compter le nombre de visites. Voici le contenu du controller :

```
@RestController
@RequestMapping("/user")
public class UserController {

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestParam String username,
    @RequestParam String password, HttpSession httpSession) {

        httpSession.setAttribute("username", username);
        httpSession.setAttribute("visites", 0);
        return ResponseEntity.ok("Login successful");
    }

    @PostMapping("/logout")
    public ResponseEntity<String> logout(HttpSession httpSession) {
        httpSession.invalidate();
        return ResponseEntity.ok("Logout successful");
    }

    @GetMapping("/visites")
    public ResponseEntity<String> getVisites(HttpSession httpSession) {
        Integer visites = (Integer) httpSession.getAttribute("visites");
        if (visites == null) {
            return ResponseEntity.status(HttpStatus.I_AM_A_TEAPOT).body("NON !!!");
        }
        visites++;
    }
}
```



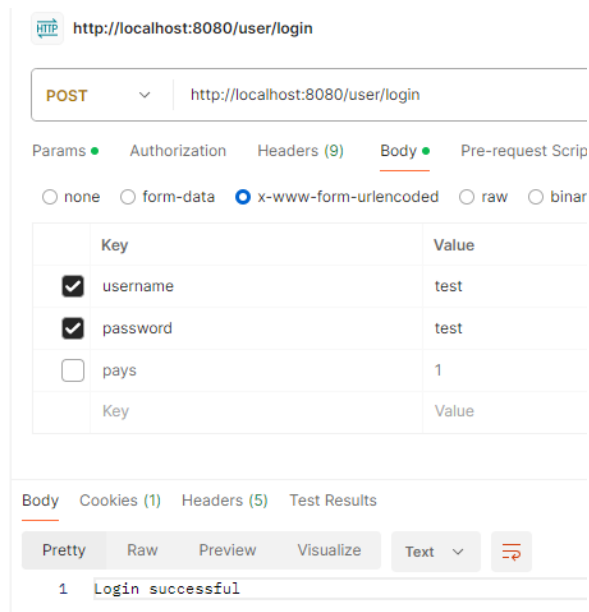
```
    httpSession.setAttribute("visites", visites);  
    return ResponseEntity.ok("Visites : " + visites);  
}  
}
```

Il ne faut pas oublier d'ajouter l'import pour HttpSession

```
import jakarta.servlet.http.HttpSession;
```

Ensuite pour tester je me suis rendu sur Postman et j'ai tester mes méthodes comme ceci :

Login :



Pour le comptage de visites :

HTTP **http://localhost:8080/user/visites**


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

Params • Authorization Headers (9) **Body** • Pre-request Sc

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

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

Body Cookies (1) Headers (5) Test Results

Pretty Raw Preview Visualize Text 

1 **Visites : 1**

Pour le logout :


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

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

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

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

Body Cookies (1) Headers (5) Test Results

Pretty Raw Preview Visualize Text 

1 **Logout successful**

2.8 Documentation API avec Swagger

Pour faire notre documentation avec Swagger, j'ai dû ajouter une dépendance dans mon pom.xml :

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

Une fois que c'est ajouté j'ai relancé mon projet et je suis allé sur ce lien pour voir ma doc : <http://localhost:8080/swagger-ui/index.html>

user-controller

POST /user/logout

POST /user/login

Parameters

Try it out

Name	Description
username * required string (query)	<input type="text" value="username"/>
password * required string (query)	<input type="text" value="password"/>

Responses

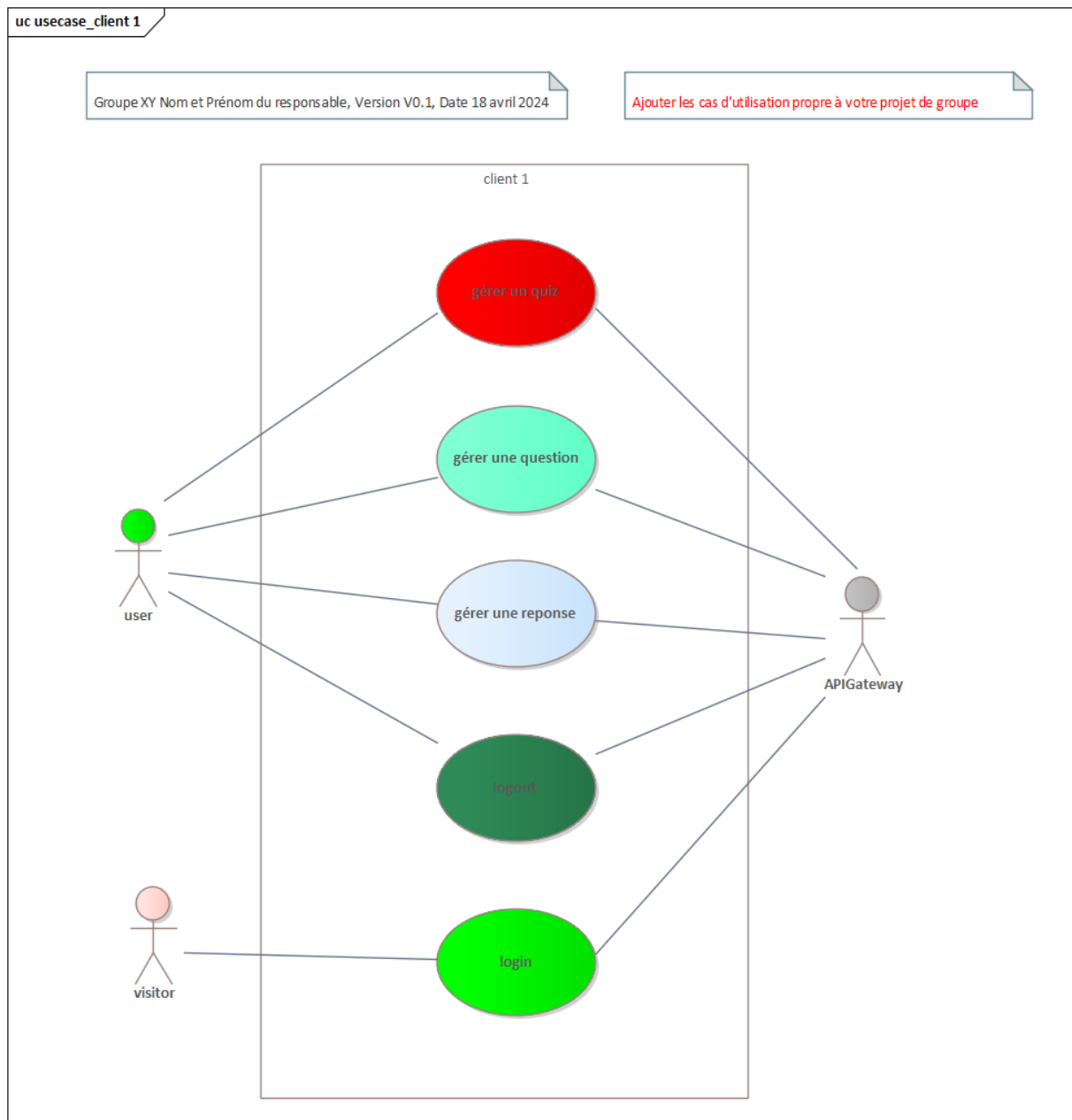
Code	Description	Links
200	OK <div>Media type <input type="text" value="*/"/> <small>Controls Accept header.</small> <div>Example Value Schema</div><div>string</div></div>	No links

GET /user/visites

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

3.1 Use case client et use case Rest

3.1.1 Use case client 1



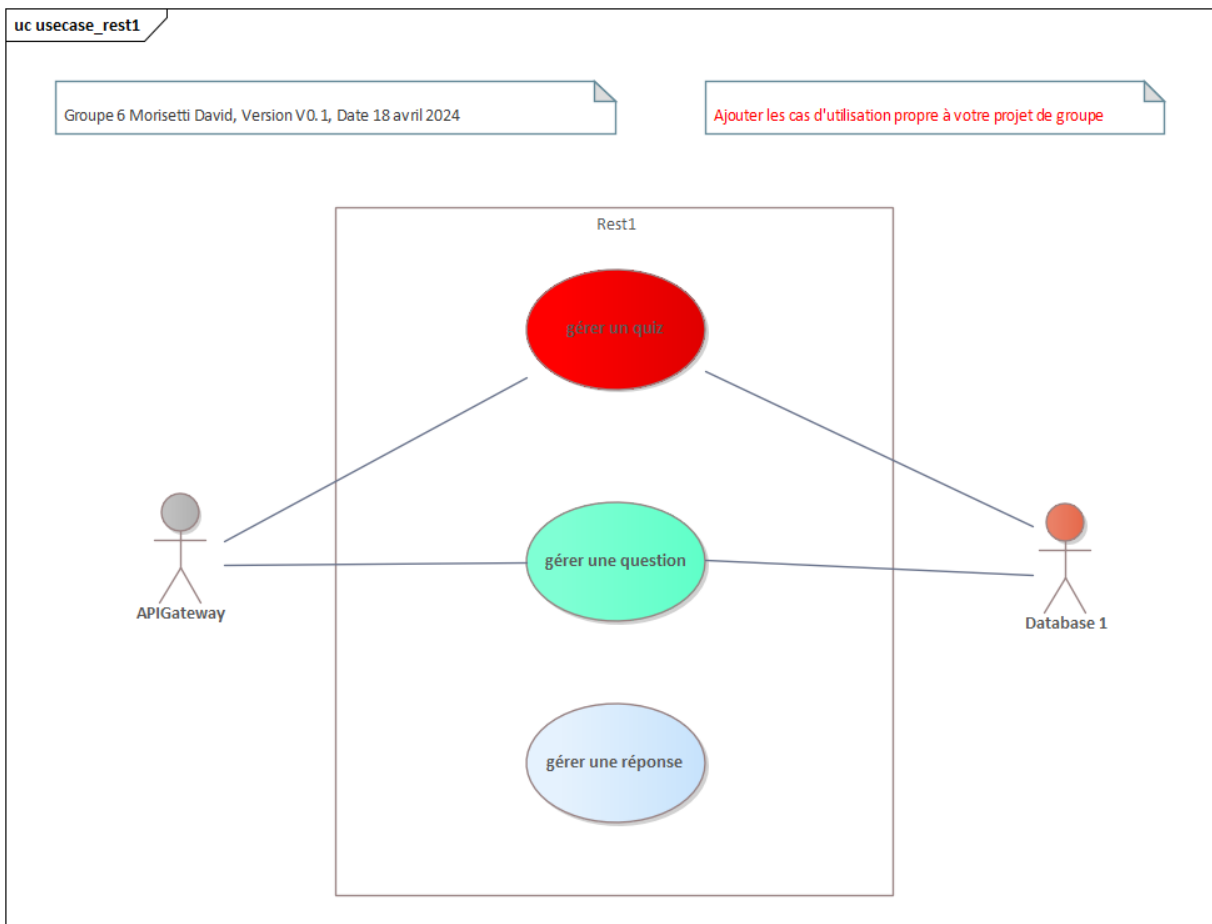
3.1.2 Use case client 2



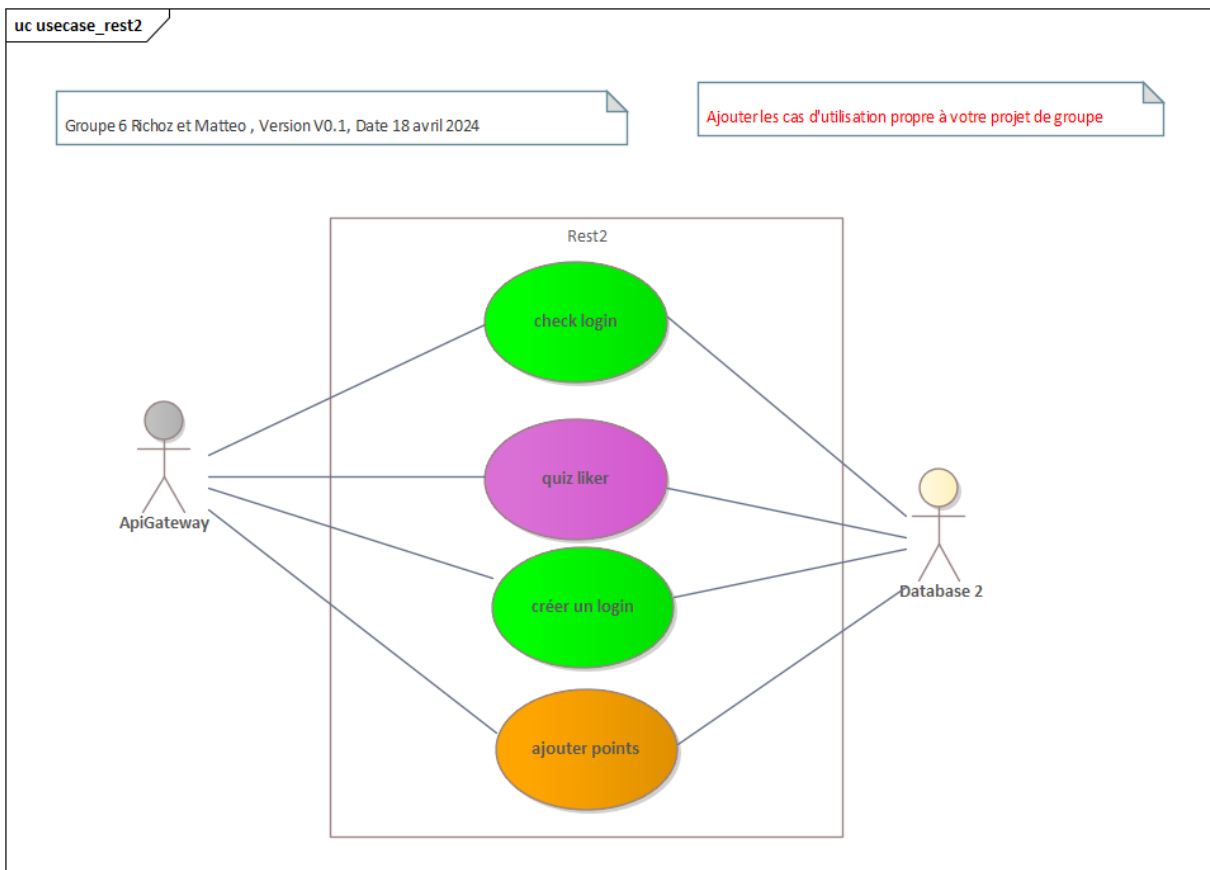
3.1.3 Use case API Gateway



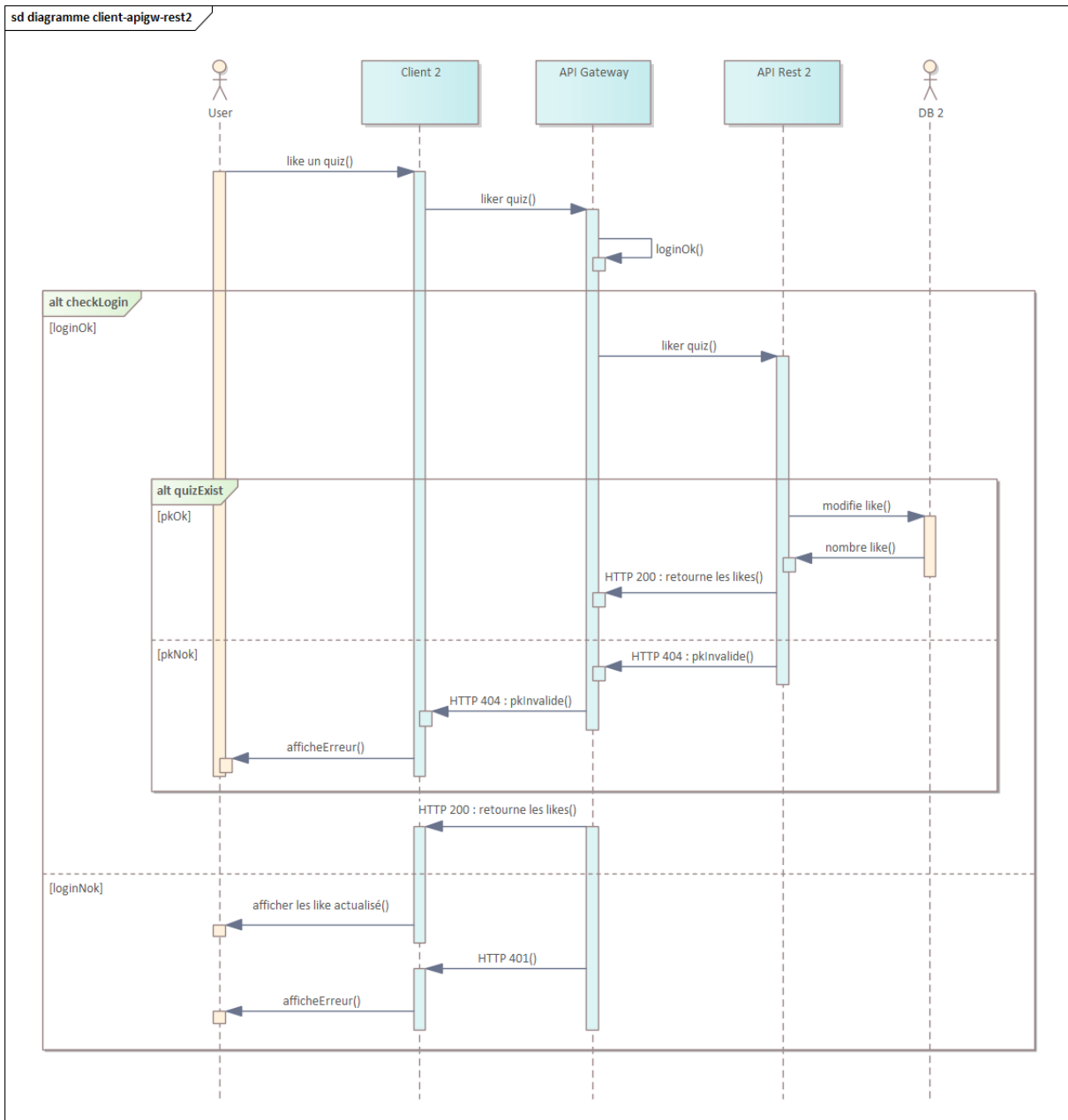
3.1.4 Use case API Rest 1



3.1.5 Use case API Rest 2

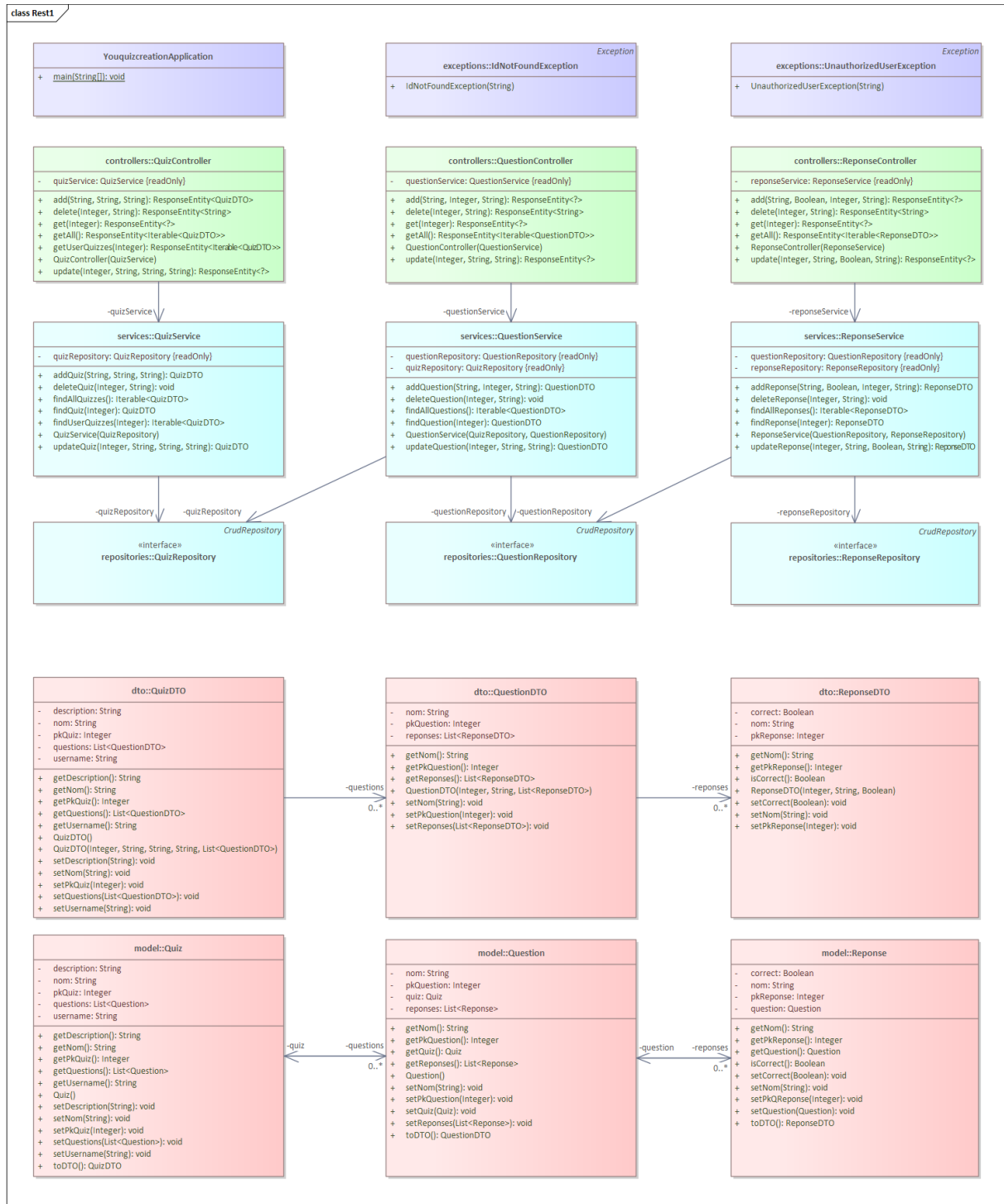


3.2 Sequence System global entre les applications

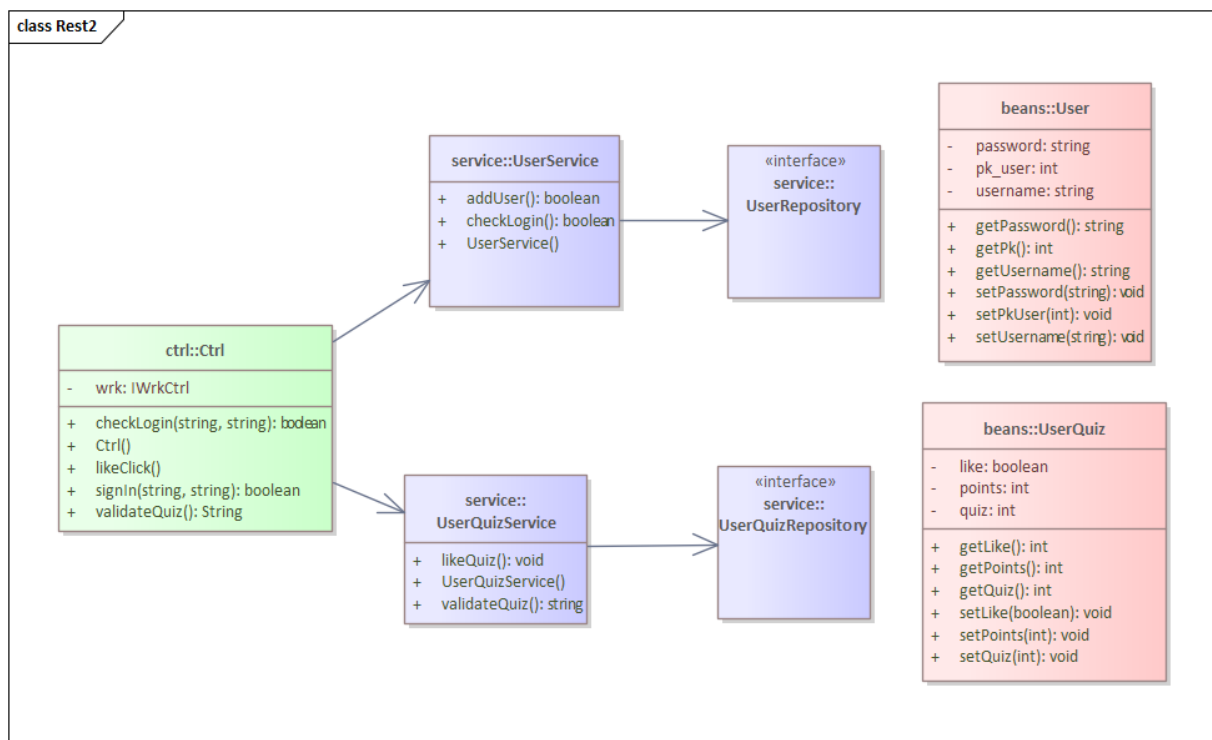


4 Conception à faire complètement avec EA -> à rendre uniquement le fichier EA

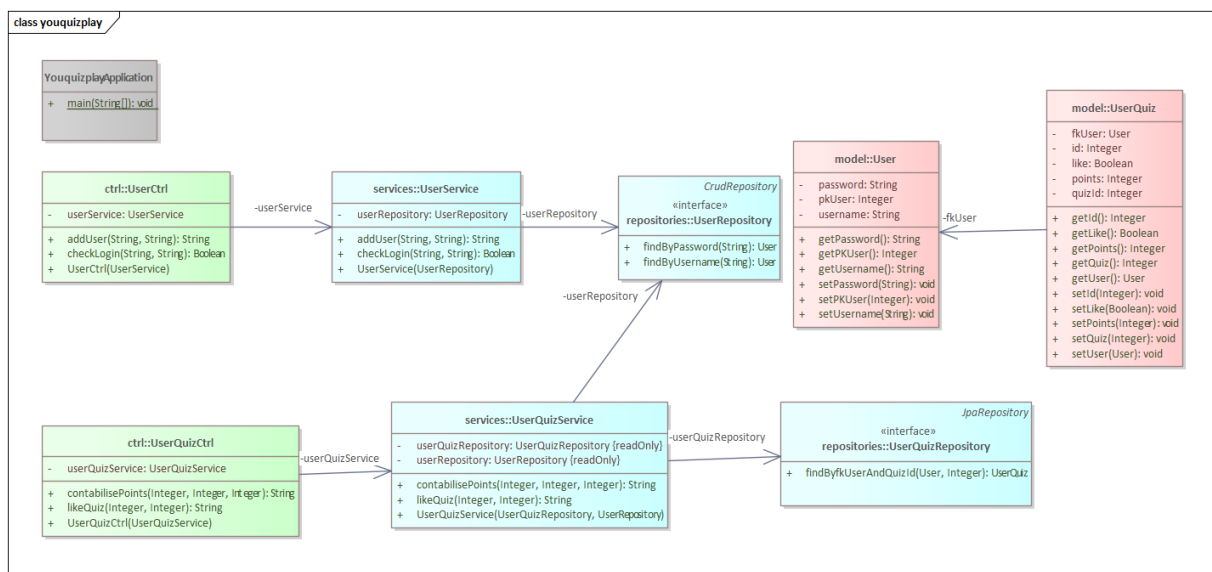
4.1 Diagramme class Rest 1



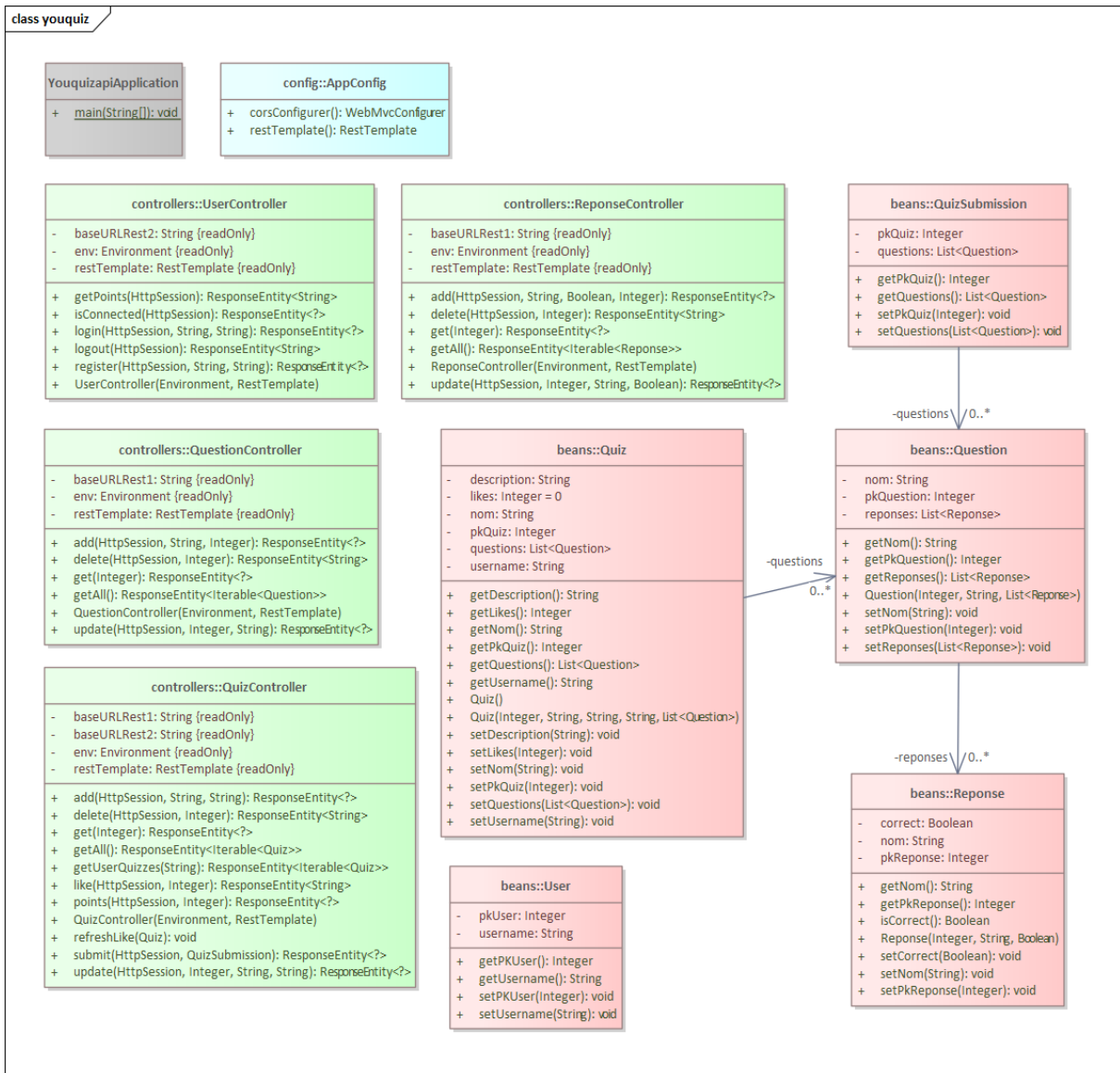
4.2 Diagramme class Rest 2



4.3 Diagramme class Rest 2 après implémentation

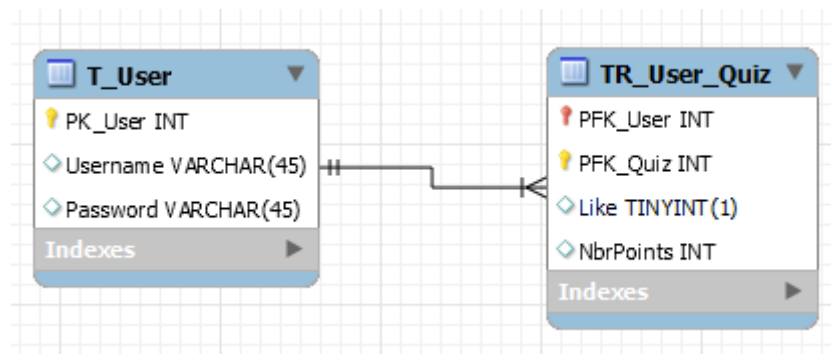


4.4 Diagramme class API Gateway



5 Bases de données

5.1 Modèles WorkBench MySQL



Dans ma base de données, je vais stocker toutes les données liées aux utilisateurs comme leur nom d'utilisateur, le mot de passe, le nombre de points qu'il ont faits aux quiz et s'ils ont liké le quiz ou pas.

6 Implémentation des applications <Le client Ap2>

6.1 Une descente de code client

Pour ma descente de code, je vais prendre comme exemple de liker un quiz. Tout commence dans le client lorsque l'utilisateur clique sur les likes en bas à droite d'un quiz dans le homeCtrl.js :

```
$(".like").click((event) => {  
    event.stopPropagation();  
    const $target = $(event.currentTarget);  
    let pkQuiz = $target.parent().parent().attr("id");  
    liker(pkQuiz, () => { }, (jqXHR) => {  
        if (jqXHR.status === 403) {  
            this.vueService.afficherErreur("Connectez vous pour pouvoir liker", ()  
=> { });  
        } else if (jqXHR.status === 200) {  
            getQuiz(pkQuiz, (data) => {  
                $target.find(".likes").text(data.likes);  
            }, (jqXHR) => {  
                console.error("Quiz pas trouvé");  
            });  
        }  
    });  
});
```

Voici le code lorsqu'un utilisateur like un quiz. On commence par créer une variable pkQuiz qui contient l'id du quiz sur lequel l'utilisateur a cliqué. Ensuite on appelle la méthode liker dans le serviceHttp.js qui prend pkQuiz en paramètre et qui permet de liker le quiz. Ensuite, on a un test qui permet de voir si l'utilisateur à le droit ou non de like le quiz (si login). Si c'est bon, on affiche le nombre de like.

Code dans serviceHttp.js :

```
function liker(pkQuiz, successCallback, errorCallback) {  
    $.ajax({  
        type: "POST",  
        dataType: "json",  
        xhrFields: {  
            withCredentials: true  
        },  
        url: BASE_URL + "quiz/like/" + pkQuiz,  
        success: successCallback,
```

```
    error: errorCallback,  
  });  
}
```

Cette méthode va transmettre à l'api Gateway la pk du quiz sur lequel l'utilisateur a cliqué avec l'url : <https://backend-6.emf4you.ch/quiz/like/pkquiz>

7 Implémentation de l'application <API Gateway>

7.1 Une descente de code APIGateway

Une fois dans l'api Gateway, on arrive dans cette méthode :

```
@PostMapping(path = "/like/{id}")

    public ResponseEntity<String> like(HttpSession session, @PathVariable("id")
Integer pkQuiz) {

    // Vérifie si l'utilisateur est connecté

    User user = (User) session.getAttribute("user");

    if (user != null) {

        try {

            restTemplate.getForEntity(baseUrlRest1 + "/get/" + pkQuiz,
Quiz.class);

        } catch (HttpClientErrorException e) {

            return
ResponseEntity.status(e.getStatusCode()).body(e.getResponseBodyAsString());

        }

        LinkedMultiValueMap<String, String> params = new
LinkedMultiValueMap<>();

        params.add("userId", String.valueOf(user.getPKUser()));

        HttpHeaders headers = new HttpHeaders();

        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

        HttpEntity<MultiValueMap<String, String>> requestEntity = new
HttpEntity<>(params, headers);

        ResponseEntity<String> response =
restTemplate.postForEntity(baseUrlRest2 + "/userquiz/like/" +
String.valueOf(pkQuiz), requestEntity, String.class);

        return ResponseEntity.ok(response.getBody());

    } else {

        return new ResponseEntity<>("Connexion nécessaire pour le like d'un
quiz.", HttpStatus.FORBIDDEN);

    }

}
```

Dans cette méthode, on commence par vérifier que l'utilisateur est connecté grâce à la session. Ensuite, on va faire une requête vers l'API REST pour mettre à jour le nombre de like et enfin on retourne ou le nombre de like ou qu'il faut se connecter.

8 Implémentation des applications <API REST2>

8.1 Une descente de code de l'API REST

Dans l'API REST, on arrive dans UserQuizCtrl.java dans la méthode like Quiz qui return un String que lui donne la méthode likeQuiz dans UserQuizService :

```
@Transactional

    public String likeQuiz(Integer userId, Integer quizId) {

        User user = userRepository.findById(userId).orElse(null);

        if (user == null) {

            return "pk invalide !";

        }

        UserQuiz userQuiz = userQuizRepository.findByfkUserAndQuizId(user, quizId);

        if (userQuiz != null) {

            userQuiz.setLike(!userQuiz.getLike());

        } else {

            userQuiz = new UserQuiz();

            userQuiz.setUser(user);

            userQuiz.setQuiz(quizId);

            userQuiz.setLike(true);

        }

        userQuizRepository.save(userQuiz);

        return "Like sauvegardé !";

    }
```

Dans cette méthode, on commence par créer un utilisateur avec l'id reçu en paramètre. Ensuite on crée une nouvelle variable userQuiz qui permet de contrôler si l'utilisateur reçu en paramètre a déjà liké ou pas le quiz reçu lui aussi en paramètre. Ensuite de quoi si oui on va set le like à 0 pour dire qu'il ne le like plus et sinon on set le like à 1 pour dire qu'il le like. Enfin on return un string qui dit que le like est sauvegardé.

9 Hébergement

Pour ce qui est de l'hébergement, on trouve :

Le client 1 sur : <https://morissettid.emf-informatique.ch/youquizcreation>

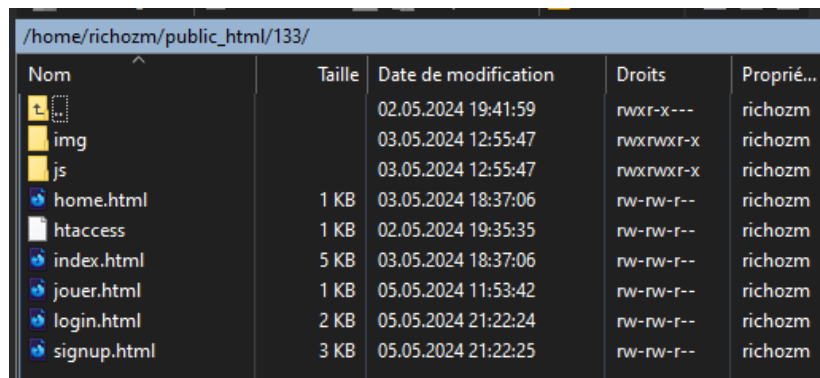
Le client 2 sur : <https://richozm.emf-informatique.ch/133>

L'API Gateway sur : <https://backend-6.emf4you.ch>

La documentation swagger sur : <https://backend-6.emf4you.ch/swagger-ui/index.html>

10 Installation du projet complet avec les 5 applications

Pour ce qui est des clients, nous avons tous mis sur cpanel :



Nom	Taille	Date de modification	Droits	Proprié...
t		02.05.2024 19:41:59	rw-r-x---	richozm
img		03.05.2024 12:55:47	rw-rwxr-x	richozm
js		03.05.2024 12:55:47	rw-rwxr-x	richozm
home.html	1 KB	03.05.2024 18:37:06	rw-rw-r--	richozm
htaccess	1 KB	02.05.2024 19:35:35	rw-rw-r--	richozm
index.html	5 KB	03.05.2024 18:37:06	rw-rw-r--	richozm
jouer.html	1 KB	05.05.2024 11:53:42	rw-rw-r--	richozm
login.html	2 KB	05.05.2024 21:22:24	rw-rw-r--	richozm
signup.html	3 KB	05.05.2024 21:22:25	rw-rw-r--	richozm

Pour les API Gateway ainsi que les API REST 1 et 2, tout a été mis en place avec docker compose sur un serveur docker ou tout est redirigé par un proxy mis en place par l'autre classe. L'unique point d'entrée vers le backend est de passer par l'api gateway qui elle a un port ouvert. Voici le contenu du docker compose :

```
version: '3.9'

services:
  apigateway:
    image: dadamomo/youquiz-apigateway:latest
    container_name: youquiz-apigateway
    environment:
      - REST1_URL=http://youquiz-rest1:8080
      - REST2_URL=http://youquiz-rest2:8080
    ports:
      - "8080:8080"
    depends_on:
      - youquiz-rest1
      - youquiz-rest2
    restart: unless-stopped
  youquiz-rest1:
    image: dadamomo/youquiz-rest1
    container_name: youquiz-rest1
    depends_on:
      - db
    environment:
      - DATABASE_URL=jdbc:mariadb://db:3306/youquizcreation
```

```
restart: unless-stopped

youquiz-rest2:

  image: ricooz/youquiz-rest2

  container_name: youquiz-rest2

  depends_on:

    - db

  environment:

    - DATABASE_URL=jdbc:mariadb://db:3306/youquizplay

  restart: unless-stopped

db:

  image: mariadb:latest

  environment:

    - MARIADB_ROOT_PASSWORD=emf123

  volumes:

    - ./data:/var/lib/mysql

    - ./docker-entrypoint-initdb.d:/docker-entrypoint-initdb.d

  restart: unless-stopped
```

11 Tests de fonctionnement du projet

11.1 Login

Lorsque je me connecte avec un utilisateur qui existe déjà, dans mon cas, le nom d'utilisateur est richozm et le mot de passe emf123, en haut à droite doit s'afficher le nom d'utilisateur ainsi que le bouton de déconnexion :



11.2 Log out

Lorsque l'utilisateur clique sur le bouton de déconnexion, en haut à droite doit s'afficher le bouton d'enregistrement et de login :



11.3 Enregistrement

L'utilisateur peut créer un nouvel utilisateur. Si le nom d'utilisateur existe déjà, la création est refusée :

Création invalide car nom utilisateur invalide :

The image shows a registration form titled 'Enregistrer un compte'. It has three input fields: 'Nom d'utilisateur' containing 'richozm', 'Mot de passe' with masked characters, and 'Confirmation du mot de passe' also with masked characters. A blue 'S'enregistrer' button is at the bottom. The form is set against a dark background.



Création invalide car mot de passe et confirmation invalide :

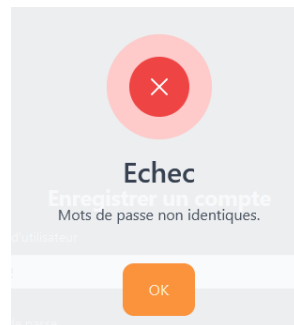
Enregistrer un compte

Nom d'utilisateur

Mot de passe

Confirmation du mot de passe

[S'enregistrer](#)



Création valide :

Enregistrer un compte

Nom d'utilisateur

Mot de passe

Confirmation du mot de passe

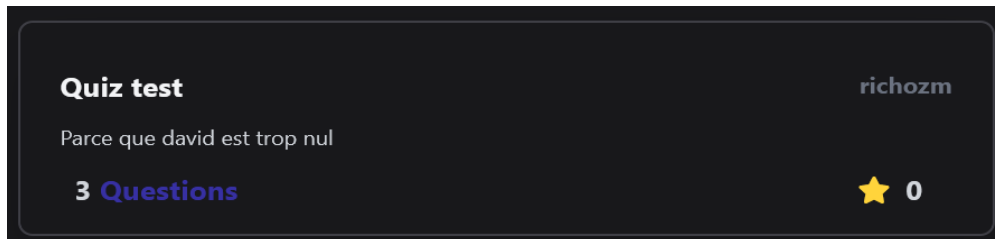
[S'enregistrer](#)



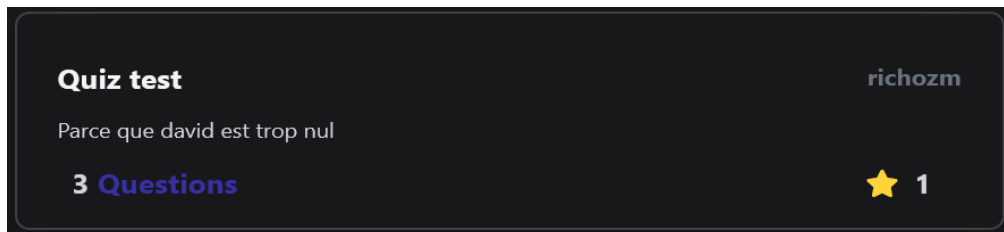
On voit que l'utilisateur a pu être créer.

11.4 Liker un quiz

Voila l'état du quiz avant de le like :



Ensuite, lorsque l'utilisateur clique en bas à droite pour like le quiz le nombre de like change :



11.5 Soumettre un questionnaire

Lorsque l'utilisateur clique sur un quiz pour le jouer, une nouvelle page s'affiche avec le titre du quiz ainsi que toutes les réponses et questions de celui-ci. Pour répondre aux questions, l'utilisateur doit cocher ou décocher la box sur la droite de la question. Enfin, l'utilisateur peut annuler sa tentative en cliquant sur le bouton rouge en bas du questionnaire ou valider les réponses. Dans mon exemple, je vais jouer le quiz nommé : Quiz test :

Quiz test

Est ce que david est nul ?

☒ oui

☐ non

Est ce que matteo est nul ?

☐ non

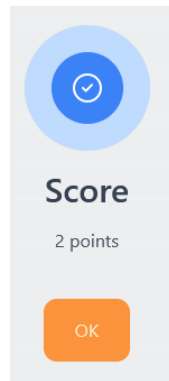
☒ oui

lkdjfshlhd

☒ demain

☐ aujourd'hui

Lorsque l'utilisateur clique sur le bouton pour soumettre, un popup s'affiche avec le nombre de points obtenus :



12 Auto-évaluations et conclusions

12.1 Auto-évaluation

Je trouve que j'ai assez bien réalisé ce module même si certaines parties étaient assez compliquées dans le projet. J'ai tout de même réussi à produire avec David un projet qui ressemble très fortement à ce que nous avions prévu de faire lors de la conception de notre projet.

12.2 Conclusion

Nous avons terminé notre projet avec toutes les fonctionnalités que nous souhaitons implémenter. En règle générale, le module était vraiment cool et j'ai vraiment aimé la manière dont nous l'avons fait avec toutes les nouvelles fonctionnalités et pour ce qui est de l'hébergement. Le petit point que j'ai moins aimé c'est que j'aurais aimé avoir une semaine de plus pour le projet même si les exercices que nous avons fait en classe restent intéressants et importants.