TP GRPC

Douhane Nadim, Geoffrey Pierre

Décembre 2024

Architecture logicielle distribuée

1 Conception

L'objectif de ce tp était de réaliser une application de réservation d'hotels en utilisant une architecture GRPC.

1.1 Base de données

La base de donnée de ce projet est similaire à celles des 2 TPs précédents, l'application place la table **Hotel** au centre du projet. Un hotel possède des **Chambres** dont chacunes peut être réservée. Chaque hotel peut être mis en partenariat avec une **Agence** qui est représenté ici par l'entité **Partenaire**. Chaque agence propose des offres concernant des chambres d'hotels auquels elles est ratachée.

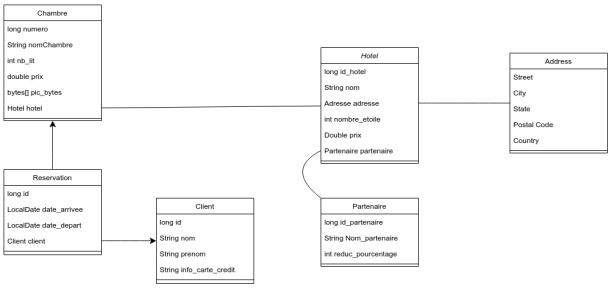


Figure 1: Schéma de la base de données

1.2 Projet commons

Pour mettre en place ce service, il fallait créer un projet Common qui contiendra un fichier ".proto" utilisé par Protocol Buffers, un format développé par Google pour sérialiser des données structurées. Il permet de spécifier le format des données que le client et le serveur vont échanger. C'est-à-dire que le client et le serveur auront accès au projet Common. Le serveur pourra définir le comportement des méthodes déclarées dans le Protobuf, et le client pourra appeler ces méthodes car il connaît leurs noms

---

```
syntax = "proto3";

option java_multiple_files = true;

package com.example.grpc;

message Empty {}

message Hotel {
        int64 id = 1;
        string name = 2;
        string role = 3;
        string email = 4;
```

```
}

message HotelId {
    int64 id = 1;
}

message HotelList {
    repeated Hotel hotels = 1;
}

message HotelCount {
    int32 count = 1;
}

service HotelService {
    rpc getAllHotels (Empty) returns (HotelList);
    rpc getHotelCount (Empty) returns (HotelCount);
    rpc getHotelById (HotelId) returns (Hotel);
    rpc addHotel (Hotel) returns (Hotel);
    rpc deleteHotelById (HotelId) returns (Empty);
    rpc updateHotel (Hotel) returns (Hotel);
}

// Message representant une adresse
message Adresse {
    string pays = 1;
    string ville = 2;
    string rue = 3;
    string lieuDit = 4;  // Correspond a lieuDit de votre modele
    int32 numero = 5;    // Correspond a numero
    int32 positionGPS = 6; // Correspond a positionGPS
}
```

Dans ce fichier, je définis les messages qui spécifient la structure des données échangées entre le client et le serveur. Chaque attribut des messages est associé à un entier unique représentant son ID. Le service HotelGrpcService décrit les méthodes qui seront implémentées côté serveur et invoquées par le client.Dans ce projet, le fichier pom.xml joue un rôle central dans la gestion et la configuration du module Common.

```
<modelVersion>4.0.0</modelVersion>
<groupId>org.anonbnr.web_services.grpc</groupId>
<artifactId>org.anonbnr.web_services.grpc.employees.commons</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>org.anonbnr.web_services.grpc.employees.commons</name>
<description>A commons project that will host the .proto files and generated stubs for the gRPC server and cli

<!-- Protobuf Compiler -->
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>${protobuf.version}</version>
</dependency>
<!-- gRPC -->
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty</artifactId>
    <version>${grpc.version}</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>${grpc.version}</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>${grpc.version}</version>
</dependency>
```

Concernant gRPC, il est utilisé pour configurer le plugin protobuf-maven-plugin, ce qui permet de

générer automatiquement les classes Java à partir des fichiers .proto. Ces classes incluent la définition des messages et des services gRPC, qui seront exploitées aussi bien par le client que par le serveur via les commande suivante :

```
mvn clean
mvn install
```

1.3 Projet serveur

Une fois cela en place, nous avons créé un projet serveur dans lequel nous avons ajouté les dépendances nécessaires dans le pom.xml, indispensables au fonctionnement du serveur. Pour chacunes des méthodes du projet commons, il fallait les récupérer dans le projet serveur et ajouter l'annotation adéquate.

```
@GrpcService
public class ChambreServiceImpl extends ChambreServiceImplBase {

    private final ChambreRepository chambreRepository;

    public ChambreServiceImpl(ChambreRepository chambreRepository) {
        this.chambreRepository = chambreRepository;
    }

    @Override
    public void getAllChambres(Empty request, StreamObserver<ChambreList> responseObserver) {
        ChambreList.Builder chambreListBuilder = ChambreList.newBuilder();
        chambreRepository.findAll().forEach(chambre -> {
            // Recuperer l'objet Hotel de votre modele Java (org.anonbnr.web_services.grpc.employees.server.mo
            org.anonbnr.web_services.grpc.employees.server.model.Hotel hotel = chambre.getHotelId();

            // Construire un objet Hotel de type com.example.grpc.Hotel pour le message gRPC
            Hotel grpcHotel = Hotel.newBuilder()
                .setNom(hotel.getNom()) // Utilisez les getters de votre modele Java pour remplir les champs
                .setAdresse(Adresse.newBuilder()
                        .setRue(hotel.getAdresse().getRue())
                        .setVille(hotel.getAdresse().getVille())
                        .setPays(hotel.getAdresse().getPays())
                        .build()) // Si applicable, sinon retirez cette ligne
                .setNombreEtoiles(hotel.getNombreEtoiles()) // Si applicable
                .setPartenaire(Partenaire.newBuilder()
                        .setNomPartenaire(hotel.getPartenaire().getNom_partenaire())
                        .setReducPourcentage(hotel.getPartenaire().getReduc_pourcentage())
                        .build())
                .build();

            // Construire l'objet Chambre gRPC
            Chambre grpcChambre = Chambre.newBuilder()
                .setNumero(chambre.getNumero())
                .setNomChambre(chambre.getnomChambre()) // Nom de la chambre
                .setNbLit(chambre.getNb_lit()) // Nombre de lits
                .setPrix(chambre.getPrixParNuit()) // Prix par nuit
                .setHotel(grpcHotel) // Utiliser l'objet Hotel gRPC
                .build();

            chambreListBuilder.addChambres(grpcChambre);
        });

        responseObserver.onNext(chambreListBuilder.build());
        responseObserver.onCompleted();
    }
```

1.4 Projet client

Le client devait pouvoir récupérer les méthodes conçues précédemment, ce qui a été réalisé via le service suivant :

```
@Service
public class HotelServiceClient {

        // Automatically inject the gRPC client stub using the @GrpcClient annotation
        @GrpcClient("hotel-service")
        private HotelServiceBlockingStub blockingStub;

        public HotelList getAllHotels(Empty request) {
                return blockingStub.getAllHotels(request);
        }

        public Hotel getHotelById(HotelId request) {
                return blockingStub.getHotelById(request);
        }

        public HotelCount getHotelCount(Empty request) {
                return blockingStub.getHotelCount(request);
        }

        public Hotel addHotel(Hotel request) {
                return blockingStub.addHotel(request);
        }

        public Empty deleteHotelById(HotelId request) {
                return blockingStub.deleteHotelById(request);
        }

        public Hotel updateHotel(Hotel request) {
                return blockingStub.updateHotel(request);
        }
}
```

La variable blockingStub reçoit l'injection du client gRPC à l'aide de l'annotation @GrpcClient("hotel-service"). Elle sert à appeler les différentes méthodes du service gRPC. Dans le service AgenceService, Nous avons créé les objets de type request et les transmettre aux fonctions mentionnées précédemment

```
private void handleGetHotelCount() {
    // TODO Auto-generated method stub
    try {
        HotelCount count = hotelServiceClient.getHotelCount(Empty.newBuilder().build());
        System.out.println("Total number of hotels: " + count.getCount());
    } catch (StatusRuntimeException e) {
        //System.err.println("Error: " + e.fillInStackTrace());
        e.printStackTrace();
    }
}

private void handleGetHotelById(Scanner scanner) {
    // TODO Auto-generated method stub
    long id = 0;
    try {
        System.out.print("Enter hotel ID: ");
        id = Long.parseLong(scanner.nextLine().trim());
        Hotel hotel = hotelServiceClient.getHotelById(
                HotelId.newBuilder().setIdHotel(id).build());

        System.out.println("Hotel [idHotel=" + hotel.getIdHotel() + ", nom=" + hotel.getNom() + ", prix=" + ho
    } catch (NumberFormatException e) {
        System.err.println("Error: Invalid Hotel ID " + e.getMessage());
    } catch (StatusRuntimeException e) {
```

```
        System.err.println("Error: No Hotel exists with ID " + id);
    }
}
```

1.5 Images

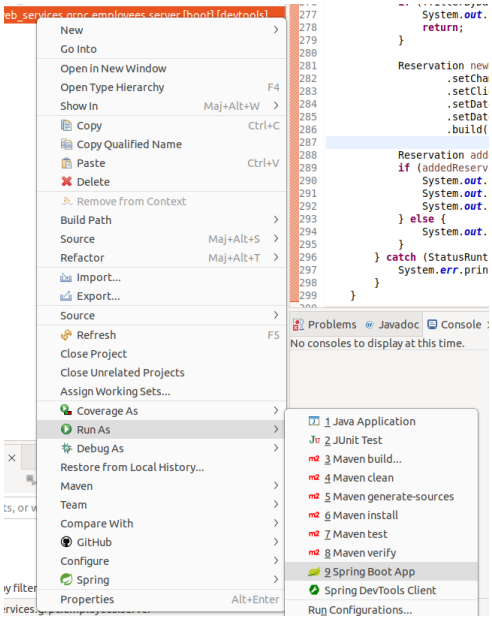
La gestion des images a été effectuée de façon semblable à celles des TP précédents. Une chaîne de caractère est enregistrée depuis eclipse, et convertie ensuite en base 64 dans le serveur puis envoyée au client lors d'une demande de consultation de chambre. L'image se retrouve alors dans le dossier img du projet client.

```
 // Decodage de l'image Base64 en byte[].
String imageString = chambre.getImagePath();
if (imageString == null || imageString.isEmpty()) {
    System.out.println("Aucune image associee a cette chambre.");
    return;
}

byte[] byteArray = Base64.getDecoder().decode(imageString);

// Lecture des donnees d'image.
ByteArrayInputStream bis = new ByteArrayInputStream(byteArray);
BufferedImage bImage2 = ImageIO.read(bis);
if (bImage2 == null) {
    System.out.println("Les donnees d'image ne sont pas valides.");
    return;
}

// Chemin de sauvegarde dans le dossier img du projet client.
String img = System.getProperty("user.dir") + "/img";
File dossier = new File(img);
if (!dossier.exists()) {
    boolean created = dossier.mkdirs();
    if (!created) {
        System.out.println("Pas de dossier img.");
        return;
    }
}

// Creation et ecriture du fichier image.
String nomFichier = "chambre" + chambre.getNumero() + ".jpg";
File fichierImage = new File(dossier, nomFichier);
ImageIO.write(bImage2, "jpg", fichierImage);

System.out.println("Image creee et enregistree dans : " + fichierImage.getAbsolutePath());

} catch (NumberFormatException e) {
System.err.println("Error: Invalid Chambre ID " + e.getMessage());
} catch (StatusRuntimeException e) {
System.err.println("Error: No Chambre exists with ID " + id);
} catch (IOException e) {
System.err.println("Error: Problem handling the image - " + e.getMessage());
}
```

2 Lancement du projet

- 1 - Télécharger le ZIP et extraire les 3 projets
- 2 - Ouvrir les projets sous Eclipse

  (Si des problèmes de packages apparaissent, clore les projets et les rouvrir. Se rendre dans le projet **commons** et exécuter les commandes **mvn clean** puis **mvn install**)
- 3 - Lancer le fichier `Application.java` depuis le projet **server** et attendre quelques secondes le chargement des données. (click droit sur le projet puis **Run as** et **Spring Boot App**



- 4 - Lancer le fichier `Application.java` depuis le projet **client** (même technique)
- 5 - Testez !