

TP 4 : Web Service gRPC de Gestion de Locations de Voitures

Sommaire

1. Contexte & Objectifs	1
2. Étape 1 : Lire et comprendre le fichier <code>.proto</code>	2
2.1. Le service <code>CarRentalService</code>	2
2.2. Les messages	2
3. Étape 2 : Écrire le service	3
3.1. Créer un projet Spring Boot <code>serveur</code>	3
3.2. Générer les fichiers à partir du <code>.proto</code>	3
3.3. Implémenter le service	4
4. Étape 3 : Écrire un premier client	5
4.1. Créer un deuxième projet Spring Boot <code>client</code>	5
4.2. Générer les fichiers à partir du <code>.proto</code>	5
4.3. Implémenter le client et le tester	6
5. Étape 4 : Compléter le service	6
6. Étape 5 : Compléter le client pour tester les nouvelles fonctionnalités du service	7

1. Contexte & Objectifs

L'objectif de ce quatrième travail pratique est de développer un **Web Service gRPC de gestion de locations de voitures**, en utilisant **Spring Boot (Java 21+)**.

Une agence souhaite mettre à disposition d'applications clientes un service permettant de consulter son catalogue de véhicules, de rechercher des voitures disponibles, de créer et gérer des réservations, puis de suivre les opérations réalisées pendant une location.

Contrairement aux TP précédents, les fonctionnalités ne seront pas exposées par une API REST. Les communications entre le client et le serveur seront réalisées à l'aide de **gRPC** et de **Protocol Buffers**.

Vous devrez notamment :

- Comprendre un contrat d'API fourni dans un fichier `.proto` et générer les classes Java correspondantes.
- Développer un **serveur Spring Boot** exposant les procédures définies par ce contrat.
- Mettre en œuvre les **quatre types d'appels RPC** : unary, server streaming, client streaming et bidirectional streaming.
- Développer un **second projet Spring Boot**, jouant le rôle de client du service gRPC, et tester

l'ensemble des procédures.

2. Étape 1 : Lire et comprendre le fichier **.proto**

Le fichier `car_rental.proto` vous est fourni avec le sujet. Il constitue le **contrat partagé** entre l'application cliente et le serveur.

2.1. Le service **CarRentalService**

Le service expose **sept procédures** :

Procédure	Type d'appel	Fonctionnalité
<code>GetCar</code>	Unary RPC	Récupérer les informations d'une voiture à partir de son identifiant.
<code>SearchAvailableCars</code>	Server streaming	Rechercher les voitures disponibles pour une ville, une période et éventuellement une catégorie. Les résultats sont envoyés un par un .
<code>CreateReservation</code>	Unary RPC	Créer une réservation pour un véhicule et une période.
<code>GetReservation</code>	Unary RPC	Consulter une réservation existante.
<code>CancelReservation</code>	Unary RPC	Annuler une réservation.
<code>SubmitInspection</code>	Client streaming	Envoyer successivement les observations d'un état des lieux et recevoir un bilan unique à la fin du flux.

Étudiez la définition du service dans le fichier **.proto**.

1. Identifiez, pour chaque procédure, le **type de requête** et le **type de réponse**.
2. Repérez les mots-clés `stream` et déterminez dans quel sens circulent les flux.

2.2. Les messages

Le fichier `.proto` définit également les structures de données utilisées par le service :

- **Données principales** : `Car`, `Reservation`, `InspectionObservation`, `InspectionSummary`, `VehiclePosition`.
- **Requêtes et réponses complémentaires** : `GetCarRequest`, `SearchCarsRequest`, `CreateReservationRequest`, `GetReservationRequest`, `CancelReservationRequest`, `CancelReservationResponse`.
- **Énumérations** : `CarCategory`, `ReservationStatus`, `DamageSeverity`.

IMPORTANT

Dans ce TP, nous ne séparerons pas les classes métier des messages Protocol Buffers.

Vous utiliserez directement les classes Java générées (`Car`, `Reservation`, `InspectionSummary`, etc.) pour représenter les données manipulées par l'application. Vous n'avez donc **pas** à créer de classes métier équivalentes, ni de DTOs ou de mappers supplémentaires.

Les types `*Request` et `*Response` complètent ces structures pour représenter les entrées et sorties des procédures distantes.

Rappel: Les objets Protocol Buffers générés sont **immutables**. Pour créer un message, utilisez le **Builder**. Par exemple, pour modifier une réservation existante, reconstruisez-la avec `toBuilder()` :

```
Car car = Car.newBuilder()
    .setId("CAR-001")
    .setBrand("Renault")
    .setModel("Clio")
    .setCity("Orleans")
    .setCategory(CarCategory.CAR_CATEGORY_ECONOMY)
    .setPricePerDayCents(4500)
    .build();
```

3. Étape 2 : Écrire le service

3.1. Créer un projet Spring Boot `serveur`

Tâche : Créez un premier projet Maven Spring Boot (packaging Jar), nommé `carrental-serveur`, avec Java 21 ou supérieur et les dépendances nécessaires au développement d'un **serveur gRPC** (I/O: `Spring gRPC Server`).

Le serveur gRPC écoutera sur le port `9090`. Ajouter la ligne suivante dans `src/main/resources/application.properties`

```
spring.grpc.server.port=9090
```

3.2. Générer les fichiers à partir du `.proto`

Tâche : Placez le fichier `.proto` fourni à l'emplacement suivant :

```
carrental-serveur/src/main/proto/car_rental.proto
```

puis lancez :

```
./mvnw clean compile
```

Vous devez notamment obtenir :

- Les **classes Java des messages** : `Car`, `Reservation`, `GetCarRequest`, `InspectionSummary`, etc.
- La classe `CarRentalServiceGrpc`, qui contient les stubs clients et la classe de base du service côté serveur.

Les sources sont généralement générées dans `target/generated-sources/protobuf/` et `target/generated-sources/protobuf/`. Marquez le dossier `generated-sources` comme **Generated Sources Root** (clique droit sur le dossier, "Mark Directory as").

WARNING

Les classes générées **ne doivent jamais être modifiées manuellement**. Toute modification du contrat s'effectue dans le fichier `.proto`, puis nécessite une nouvelle génération.

NOTE

Il existe un problème connu entre IntelliJ et certaines versions récentes de Maven, notamment Maven 3.10.x. Concrètement, malgré la création du projet Maven avec les bonnes dépendences vers Spring Boot, l'IDE n'importe pas correctement les classes depuis le serveur Maven. Une tentative de solution:

1. Ouvre File > Settings (ou IntelliJ IDEA > Settings sur MacOS).
2. Va dans Build, Execution, Deployment > Build Tools > Maven.
3. Repère le paramètre Maven home path.
4. Si tu as Use Maven Wrapper, remplace-le par Bundled (Maven 3) dans la liste déroulante.
5. Clique sur Apply, puis OK.
6. Dans l'onglet Maven, clique sur Reload All Maven Projects.

3.3. Implémenter le service

Tâche : Créez la classe `CarRentalGrpcService`. Elle devra étendre la classe de base **générée** pour le service :

```
@GrpcService
public class CarRentalGrpcService
    extends CarRentalServiceGrpc.CarRentalServiceImplBase {

    @Override
    public void getCar(GetCarRequest request,
        StreamObserver<Car> responseObserver) {
        // A implementer
    }
    ...
}
```

L'annotation `@GrpcService` permet à Spring d'enregistrer le service gRPC. Les signatures Java à

redéfinir proviennent du fichier `.proto`.

Tâche : Implémentez dans un premier temps, les **quatre procédures unary** : `GetCar`, `CreateReservation`, `GetReservation` et `CancelReservation`.

Consignes métier : Vous utiliserez une facade `FacadeCarRental`, que vousinstancierez dans une classe `ServiceConfig`. Votre facade doit pouvoir :

1. Conserver les voitures et les réservations dans des collections en mémoire (par exemple des `Map<String, Car>` et `Map<String, Reservation>`). Il est vivement conseillé d'ajouter un ensemble de voitures à la création de votre facade.
2. Pour `GetCar` et `GetReservation`, rechercher l'objet demandé à partir de son identifiant.
3. Pour `CreateReservation`, vérifier que la voiture existe, que la date de fin est postérieure à la date de début et qu'aucune réservation **confirmée** n'occupe déjà cette voiture sur une période qui se chevauche.
4. Générer un identifiant unique pour chaque nouvelle réservation. Calculer le prix total à partir de `price_per_day_cents` et de la durée de location (pour le TP, toute journée entamée est facturée).
5. Pour `CancelReservation`, faire passer le statut à `RESERVATION_STATUS_CANCELLED` ; un second appel d'annulation sur une réservation déjà annulée retourne `cancelled = false`.

Tâche : Une fois les premières procédures implémentées, et votre début de facade créé, démarrez votre service gRPC.

4. Étape 3 : Écrire un premier client

4.1. Créer un deuxième projet Spring Boot **client**

Tâche : Créez un second projet Maven Spring Boot, nommé `carrental-client` (packaging Jar), indépendant du projet `carrental-serveur`, avec Java 21 ou supérieur. Ce projet doit contenir les dépendances nécessaires à un **client gRPC** (I/O: Spring gRPC Client). Il ne doit pas accéder directement aux collections Java du serveur : **toutes les opérations passent par les procédures gRPC**. Le serveur et le client seront démarrés comme **deux applications distinctes**.

4.2. Générer les fichiers à partir du `.proto`

Tâche: Copiez le même fichier `car_rental.proto` dans :

```
carrental-client/src/main/proto/car_rental.proto
```

Configurez la génération et lancez à nouveau :

```
./mvnw clean compile
```

Vérifiez que les classes Java des messages et `CarRentalServiceGrpc` sont disponibles dans le projet client. Le contrat doit être identique de part et d'autre.

4.3. Implémenter le client et le tester

Tâche : Créez une classe exécutée au démarrage du client (par exemple avec `CommandLineRunner` ou `ApplicationRunner`). Elle devra :

1. Créer un **Channel** vers le serveur gRPC ;
2. Construire un **stubs** à partir de `CarRentalServiceGrpc` ;
3. Invoquer les procédures distantes et afficher les résultats dans la console.

Avec Spring gRPC, vous pouvez configurer un channel nommé `rental` :

```
spring.grpc.client.channels.rental.address=localhost:9090
```

Et utiliser `GrpcChannelFactory` pour créer le channel :

```
ManagedChannel channel = channelFactory.createChannel("rental");
CarRentalServiceGrpc.CarRentalServiceBlockingStub blockingStub = CarRentalServiceGrpc
    .newBlockingStub(channel);
```

Le **blocking stub** convient aux RPC unary et permet de lire un server streaming avec un `Iterator`.

L'autre possibilité est d'instancier le channel à partir de l'adresse et le port du service :

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress("localhost", 9090)
    .usePlaintext()
    .build();
CarRentalServiceGrpc.CarRentalServiceBlockingStub blockingStub = CarRentalServiceGrpc
    .newBlockingStub(channel);
```

Tâche : Réalisez au minimum les tests suivants depuis votre application cliente :

1. **Catalogue** : récupérer une voiture existante avec `GetCar`, puis demander un identifiant inexistant et observer l'erreur gRPC.
2. **Réservations** : créer une réservation, la consulter, vérifier qu'une réservation chevauchante est refusée, puis annuler la première réservation.

5. Étape 4 : Compléter le service

Tâche: Implémentez la procédure **server streaming** `SearchAvailableCars`.

Le serveur doit filtrer les véhicules en fonction de la ville, de la catégorie demandée (la valeur

`CAR_CATEGORY_UNSPECIFIED` signifie toutes les catégories) et des dates de location. Les réservations annulées ne bloquent pas la disponibilité. Complétez la facade en conséquence.

Chaque voiture correspondante est envoyée au client au moyen d'un appel à `responseObserver.onNext(...)`. Le serveur termine ensuite le flux avec `responseObserver.onCompleted()`.

Tâche : Implémentez la procédure **client streaming** `SubmitInspection`. Complétez la facade en conséquence.

Le serveur reçoit les messages `InspectionObservation` au fur et à mesure :

1. Il vérifie que les observations concernent toutes la **même réservation existante**.
2. Il compte le nombre d'observations reçues.
3. Il compte comme dégâts les observations de sévérité `DAMAGE_SEVERITY_MINOR` ou `DAMAGE_SEVERITY_MAJOR`.
4. À la fin du flux envoyé par le client, il renvoie **un seul** `InspectionSummary` contenant les compteurs.

NOTE

Les RPC unary et server streaming possèdent une signature Java avec une requête et un `StreamObserver` de réponse.

Les RPC client streaming **retourne** un `StreamObserver` qui permet au serveur de recevoir les messages envoyés par le client.

Tâche : Gérez les erreurs gRPC de manière appropriée, en utilisant notamment :

- `NOT_FOUND` lorsqu'une voiture ou une réservation est introuvable ;
- `INVALID_ARGUMENT` lorsque les paramètres sont incohérents (dates, positions GPS, observations, etc.) ;
- `FAILED_PRECONDITION` lorsqu'une réservation n'est pas possible car le véhicule est indisponible, ou lorsque l'état de la réservation interdit une opération.

Une erreur doit terminer l'appel via `onError(...)`. Ne tentez pas d'envoyer de réponse après `onError(...)`.

6. Étape 5 : Compléter le client pour tester les nouvelles fonctionnalités du service

Tâche : Ajoutez au minimum les tests suivants depuis votre application cliente :

1. Rechercher les voitures disponibles avec `SearchAvailableCars` et afficher les voitures reçues une par une.
2. Transmettre au moins trois observations (dont au moins un dégât) à `SubmitInspection`, terminer le flux, puis afficher le `InspectionSummary` retourné.

IMPORTANT

Pour le dernier test, utilisez un **stub asynchrone** et veillez à laisser le temps aux callbacks de recevoir les réponses avant l'arrêt de l'application (par exemple avec un mécanisme de synchronisation adapté).

Ne confondez pas le `StreamObserver` utilisé par l'API Java et le type streaming défini dans le `.proto`.