

Las Dunas Rent A Car

Application Web dédiée à la location de voiture

Projet réaliser pour la prestation de l'examen du titre
professionnel de

Concepteur Développeur d' Applications

Presente par
Esteban BARE

Introduction

Après ma première année en DWWM, j'ai souhaité approfondir mes compétences et en acquérir de nouvelles. La formation professionnelle CDA m'a permis d'aller plus loin, notamment en travaillant sur les bases de données NoSQL, le back-end en Java, la POO et le framework Spring Boot, qui apporte sécurité et fonctionnalités avancées aux applications.

J'ai également beaucoup progressé sur les API et les microservices, ce qui m'a donné une meilleure compréhension des architectures modernes. De plus, j'ai surmonté mes difficultés avec Docker, et je suis maintenant capable de l'utiliser efficacement pour gérer et déployer mes projets.

Cette expérience m'a aussi permis de renforcer mes soft skills grâce au travail en équipe et aux échanges avec différents formateurs, ce qui m'a donné de nouvelles perspectives de développement.

Enfin, tout ce que j'ai appris au cours de cette année m'a permis de réaliser mon projet Las Dunas Rent a Car, que je présenterai pour l'examen de CDA.

Liste de compétences du référentiel couvertes par le projet

Compétence 1 — Installer et configurer son environnement

Pour démarrer le projet, j'ai installé la JDK, Maven, Node.js et Angular CLI afin de lancer les microservices Spring Boot et le front Angular. Après avoir cloné le dépôt GitHub, j'ai configuré les fichiers application.properties et vérifié le bon fonctionnement des services via SpringCloudConfig et SpringCloudGateway.

Compétence 2 — Développer des interfaces utilisateur

Dans le dossier Front-Dunas, j'ai développé l'interface utilisateur en Angular. J'ai créé des composants, connecté les services aux APIs des microservices et géré la validation des formulaires. J'ai également ajouté la gestion des erreurs côté client et préparé des tests unitaires avec Jasmine/Karma pour sécuriser l'application.

Compétence 3 — Développer des composants métier

J'ai implémenté la logique métier dans différents microservices (Ms-Rental, Ms-Pricing, Ms-Promo, Ms-Comments). Chaque service a ses entités et traitements spécifiques, et la sécurité est gérée par Ms-Security. J'ai rédigé des tests unitaires avec JUnit pour valider les règles métier et documenté le fonctionnement des APIs.

Compétence 4 — Contribuer à la gestion d'un projet informatique

J'ai utilisé GitHub pour organiser et suivre mon travail (issues, branches fonctionnelles et commits réguliers). J'ai respecté des conventions de code et documenté l'avancement par des notes de suivi, ce qui m'a permis de gérer efficacement les différentes étapes du projet et d'assurer la traçabilité.

Compétence 5 — Analyser les besoins et réaliser des wireframes

J'ai étudié les besoins liés à la réservation et la gestion de véhicules puis créé des wireframes pour représenter l'enchaînement des écrans (recherche, réservation, gestion client). Ces wireframes m'ont permis d'anticiper la navigation et la structure fonctionnelle de l'application avant de passer au développement.

Compétence 6 — Définir l'architecture logicielle

J'ai conçu une architecture multicouche reposant sur une séparation claire : Angular pour le front, Spring Cloud Gateway pour le routage, les microservices Spring Boot pour la logique métier et Spring Cloud Config pour la configuration centralisée. Cette organisation permet une meilleure évolutivité et une gestion centralisée de la sécurité.

Compétence 7 — Concevoir et mettre en place une base relationnelle

J'ai défini le schéma relationnel de l'application (clients, véhicules, réservations, tarification) et préparé les scripts SQL nécessaires à l'initialisation des bases de données des microservices. J'ai également constitué un jeu de données de test et mis en place les règles d'accès et de droits utilisateurs.

Compétence 8 — Développer des composants d'accès aux données

J'ai créé les repositories et composants d'accès aux données pour chaque microservice en appliquant des requêtes sécurisées et en gérant les transactions. J'ai validé ces accès par des tests unitaires et d'intégration, afin de garantir la fiabilité et la sécurité du stockage des informations.

Compétence 9 — Préparer et exécuter les plans de tests

J'ai élaboré un plan de tests comprenant des tests unitaires, d'intégration et de non-régression. J'ai testé la communication entre les microservices via l'API Gateway et vérifié les contrôles de sécurité des endpoints. Enfin, j'ai exécuté des scénarios utilisateurs pour valider les fonctionnalités principales de l'application.

Compétence 10 — Préparer et documenter le déploiement

J'ai rédigé une procédure de déploiement détaillant les étapes de build (Maven pour les microservices, Angular pour le front), la configuration des environnements et les vérifications après installation. J'ai défini les environnements SIT/UAT/PROD et décrit les étapes de mise en ligne pour assurer une installation maîtrisée.

Compétence 11 — Contribuer à la mise en production (DevOps)

J'ai préparé l'intégration continue en organisant les services pour être conteneurisés et déployés plus facilement. J'ai documenté la structure pour mettre en place un pipeline CI/CD avec build, tests et packaging automatisés. L'architecture microservices est prête à être intégrée dans une chaîne DevOps complète avec Docker et GitHub Actions.

Résumé du Projet

Pour ce projet, je voulais m'inspirer d'expériences que tout le monde connaît déjà : les plateformes de location en ligne. On a tous déjà réservé une voiture pour un voyage ou un déplacement, mais souvent les sites existants manquent de clarté ou de modernité. C'est là que naît **Las Dunas Rent A Car** : une application pensée pour rendre la location de véhicules plus simple, plus intuitive et plus agréable.

L'idée est d'avoir une plateforme où l'utilisateur peut chercher une voiture, voir les tarifs en temps réel, profiter de promotions et même laisser des commentaires sur son expérience. Le but est de rendre le parcours utilisateur fluide : de la recherche jusqu'à la réservation, en passant par la gestion de son profil.

Le projet est construit sur une architecture moderne, avec un front-end en Angular pour offrir une navigation rapide et agréable, et des microservices en Java/Spring Boot pour gérer la logique métier (locations, tarifs, promos, avis, sécurité). Cette organisation permet de séparer clairement chaque fonction et de rendre l'ensemble évolutif et sécurisé.

En résumé, **Las Dunas Rent A Car** se veut être bien plus qu'un simple site de location : c'est une solution moderne et accessible, pensée pour que chacun puisse trouver et réserver une voiture facilement, tout en bénéficiant d'une expérience claire, conviviale et transparente.

Cahier des charges

Objectif

L'objectif principal de mon projet est de permettre aux utilisateurs de **réserver facilement une voiture en ligne**. Le site doit offrir une navigation simple et intuitive pour rechercher un véhicule, consulter les prix et finaliser une réservation en quelques clics.

Je souhaite également intégrer des **promotions et des réductions** afin d'attirer et fidéliser les clients. Les utilisateurs pourront laisser des **commentaires et avis** sur leur expérience, ce qui apportera plus de transparence et aidera les futurs clients à faire leur choix.

Pour améliorer l'expérience, il sera possible de **trier et filtrer les véhicules** (par prix, catégorie, disponibilité) et d'afficher des suggestions en fonction des besoins de l'utilisateur.

Enfin, il est indispensable de mettre en place un **système sécurisé d'inscription et de connexion** pour protéger les données personnelles. Un **back-office** sera prévu afin de gérer les utilisateurs, les réservations, les véhicules et les promotions.

User Stories

ID	En tant que...	Je veux...	Afin de...	Priorité
US1	Utilisateur	Créer un compte et me connecter	Accéder à mes réservations et mes infos	Haute
US2	Utilisateur	Rechercher un véhicule par date, type, prix	Trouver une voiture adaptée rapidement	Haute
US3	Utilisateur	Voir le détail d'un véhicule	Choisir le véhicule le plus adapté	Haute
US4	Utilisateur	Réserver un véhicule en ligne	Gagner du temps et éviter de me déplacer	Haute
US5	Utilisateur	Payer ma réservation de manière sécurisée	Finaliser mon achat en toute confiance	Haute
US6	Utilisateur	Bénéficier de promotions et réductions	Louer une voiture moins cher	Moyenne
US7	Utilisateur	Laisser un commentaire	Partager mon avis et aider les autres	Moyenne
US8	Utilisateur	Consulter les avis des autres	Faire un choix éclairé	Moyenne
US9	Utilisateur	Modifier ou annuler une réservation	Avoir de la flexibilité	Moyenne
US10	Utilisateur	Gérer mes infos personnelles	Garder mes données à jour	Moyenne
US11	Administrateur	Gérer les véhicules	Maintenir un catalogue toujours à jour	Haute
US12	Administrateur	Gérer les utilisateurs	Assurer la sécurité de la plateforme	Haute
US13	Administrateur	Gérer les réservations	Aider les utilisateurs en cas de problème	Haute
US14	Administrateur	Créer et gérer des promotions	Attirer et fidéliser des clients	Moyenne
US15	Administrateur	Consulter des statistiques	Suivre l'activité et améliorer le service	Basse

Fonctionnalités

En suivant les user stories, je peux lister les fonctionnalités principales du projet **Las Dunas Rent A Car** :

1. **Accueil** : Présentation des véhicules disponibles et mise en avant des promotions en cours.
2. **Recherche de véhicules** : Filtrer par date, type de véhicule, prix ou catégorie.
3. **Détail d'un véhicule** : Voir la description complète, photos, tarifs, promotions et avis clients.
4. **Réservation en ligne** : Choisir un véhicule, sélectionner les dates, confirmer et payer de façon sécurisée.
5. **Gestion du profil utilisateur** : Modifier ses informations personnelles, mot de passe et consulter l'historique de ses réservations.
6. **Commentaires et avis** : Publier un avis sur une location et consulter les retours d'autres utilisateurs.
7. **Promotions et réductions** : Affichage des offres spéciales pour fidéliser les clients.
8. **Administration** : Tableau de bord pour gérer les véhicules, les utilisateurs, les réservations, les promotions et consulter des statistiques.

Rôles et permissions

1. Utilisateur non authentifié :

- Consulter la liste des véhicules.
- Accéder aux détails d'un véhicule et lire les avis.
- Rechercher et filtrer les véhicules.

2. Utilisateur authentifié :

- Réserver un véhicule en ligne et effectuer un paiement.
- Gérer son profil et consulter son historique de réservations.
- Publier et consulter des commentaires.

3. Administrateur :

- Gérer les véhicules (ajouter, modifier, supprimer).
- Gérer les utilisateurs (activer, suspendre).
- Gérer les réservations (valider, annuler).
- Créer et gérer des promotions.
- Accéder à un tableau de bord avec statistiques et rapports.

Wireframes

Voici quelques wireframes que j'ai créés afin d'anticiper la structure de mon projet et de planifier le développement de mon application.

- Page d'accueil

PAGE D'ACCUEIL

LOGO LAS DUNAS

MENU | CONNEXION | PANIER

HERO SECTION

Titre principal + slogan

Image de fond véhicule

BARRE DE RECHERCHE

DATE DÉBUT

DATE FIN

TYPE VÉHICULE

RECHERCHER

SECTION PROMOTIONS

Bannière offres spéciales

TITRE : VÉHICULES DISPONIBLES

IMAGE VÉHICULE 1

NOM VÉHICULE
TYPE - PLACES
PRIX/JOUR
[BOUTON DÉTAILS]

IMAGE VÉHICULE 2

NOM VÉHICULE
TYPE - PLACES
PRIX/JOUR
[BOUTON DÉTAILS]

IMAGE VÉHICULE 3

NOM VÉHICULE
TYPE - PLACES
PRIX/JOUR
[BOUTON DÉTAILS]

IMAGE VÉHICULE 4

NOM VÉHICULE
TYPE - PLACES
PRIX/JOUR
[BOUTON DÉTAILS]

FOOTER

Contact | À propos | Conditions | FAQ

Page 11

- Page de connexion

PAGE CONNEXION

LOGO LAS DUNAS

← RETOUR ACCUEIL

TITRE : CONNEXION

EMAIL

[CHAMP EMAIL]

MOT DE PASSE

[CHAMP PASSWORD]

☐ SE SOUVENIR

[MOT DE PASSE OUBLIÉ ?](#)

SE CONNECTER

OU

GOOGLE

FACEBOOK

[PAS DE COMPTE ? S'INSCRIRE](#)

FOOTER

- Détails d'un véhicule

DÉTAIL VÉHICULE

LOGO LAS DUNAS

← RETOUR | COMPTE | PANIER (1)

ACCUEIL > VÉHICULES > BMW X3

IMAGE PRINCIPALE
Photo principale du véhicule

IMG 1

IMG 2

IMG 3

IMG 4

NOM DU VÉHICULE
★★★★★ (4,8/5 - 24 avis)

TARIFICATION

Prix/jour : X0€
Promotion active : -15%
Prix final : X0€

CARACTÉRISTIQUES

- Type véhicule
- Nombre de places
- Transmission
- Climatisation
- GPS, Bluetooth
- Assurance incluse

FAVORIS

RÉSERVER

AVIS CLIENTS (24)

CLIENT 1 ★★★★★
Commentaire client lorem ipsum...
Il y a 2 jours

CLIENT 2 ★★★★★
Commentaire client lorem ipsum...
Il y a 1 semaine

[BOUTON VOIR TOUS LES AVIS]

FOOTER

Page 13

Stack et technologies utilisées

Front-End

Pour le front-end, j'ai utilisé **Angular** avec **TypeScript**, **HTML** et **CSS**. Angular me permet de créer des interfaces dynamiques et modulaires, tout en profitant de son écosystème (Angular CLI, npm) pour gérer les dépendances et lancer le projet facilement.

Back-End

Pour le back-end, j'ai choisi **Java** avec le framework **Spring Boot**. C'est une solution fiable et largement utilisée pour développer des applications robustes et sécurisées. Spring Boot simplifie la création d'APIs REST et l'intégration avec d'autres outils de l'écosystème Java.

Base de données

J'utilise principalement **MySQL** avec **Spring Data JPA** pour les opérations CRUD, et **MongoDB (NoSQL)** pour certains microservices, grâce à une dépendance qui permet de l'utiliser de manière similaire à JPA.



Architecture de Las Dunas Rent A Car

1. Microservices Spring Boot

Le projet adopte une **architecture en microservices**, où chaque composant métier est encapsulé dans un service dédié. On retrouve notamment :

- **Ms-Rental** (gestion des locations)
- **Ms-Pricing** (gestion des tarifs)
- **Ms-Promo** (gestion des promotions)
- **Ms-Comments** (gestion des avis clients)
- **Ms-Security** (authentification et autorisation)

Chacun de ces services est développé en **Java** avec **Spring Boot**, ce qui permet une modularité, une évolutivité et une maintenance facilitées.

2. API Gateway avec Spring Cloud Gateway

L'**API Gateway** (via Spring Cloud Gateway) agit comme point d'entrée unique pour toutes les requêtes externes. Il oriente les appels vers les microservices appropriés, gère la sécurité des endpoints, les questions de CORS, et peut appliquer des filtres (auth, headers, etc.).

3. Configuration centralisée avec Spring Cloud Config

La configuration des microservices est externalisée et centralisée via **Spring Cloud Config**, ce qui permet de gérer de manière fluide et cohérente les paramètres (URL de DB, ports, clés, etc.) selon les environnements (dev, test, prod).

4. Communication inter-services via API REST

Les microservices communiquent entre eux via des **APIs REST**. Chaque service est autonome, avec son propre domaine fonctionnel et pouvant évoluer indépendamment des autres.

5. Base de données distribuée (une par service)

Chaque microservice gère sa propre base de données relationnelle via **Spring Data JPA**, ce qui permet de découpler la gestion des données, d'évoluer indépendamment et de garantir la résilience et la cohérence du système. Par ailleurs, deux microservices utilisent une base **NoSQL MongoDB**, et grâce à une dépendance spécifique, je peux interagir avec MongoDB de manière similaire à JPA, ce qui simplifie le développement et la maintenance.

Base de données

MCD / MLD :

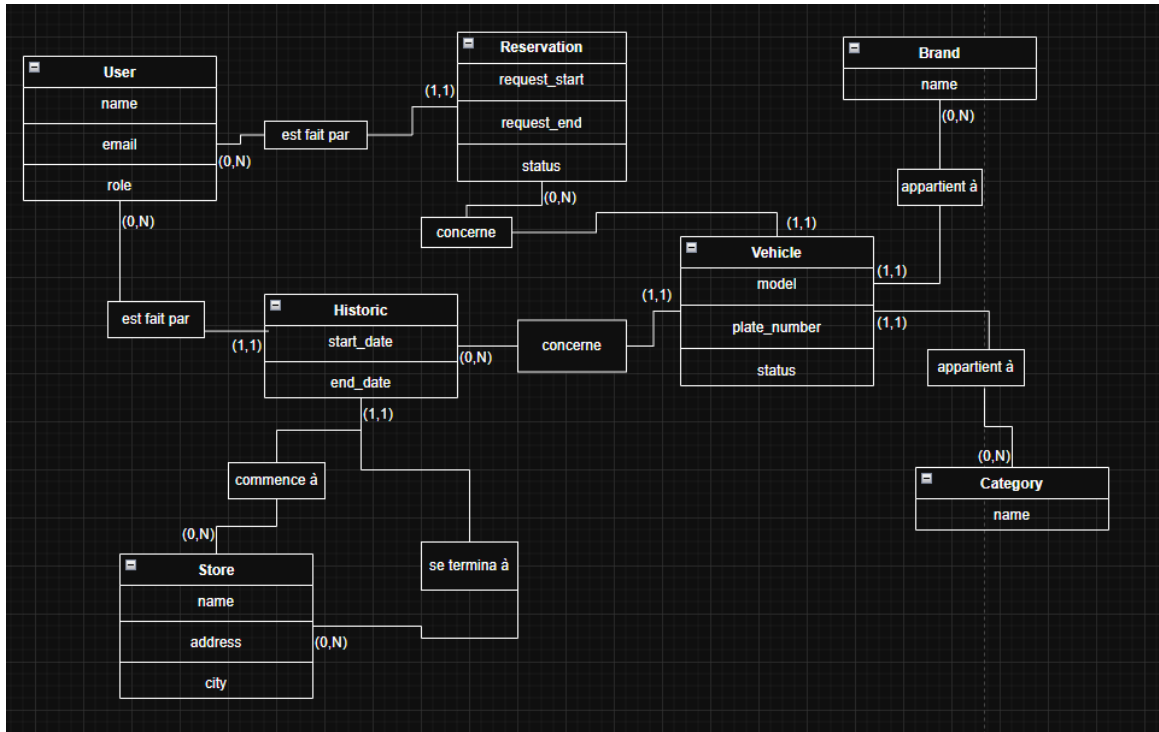
Pour la modélisation de la base de données du projet **Las Dunas Rent A Car**, j'ai utilisé **Mermaid** afin de créer le schéma conceptuel (MCD) et le schéma logique (MLD).

Après la création du MCD, j'ai pu analyser les relations entre les différentes entités :

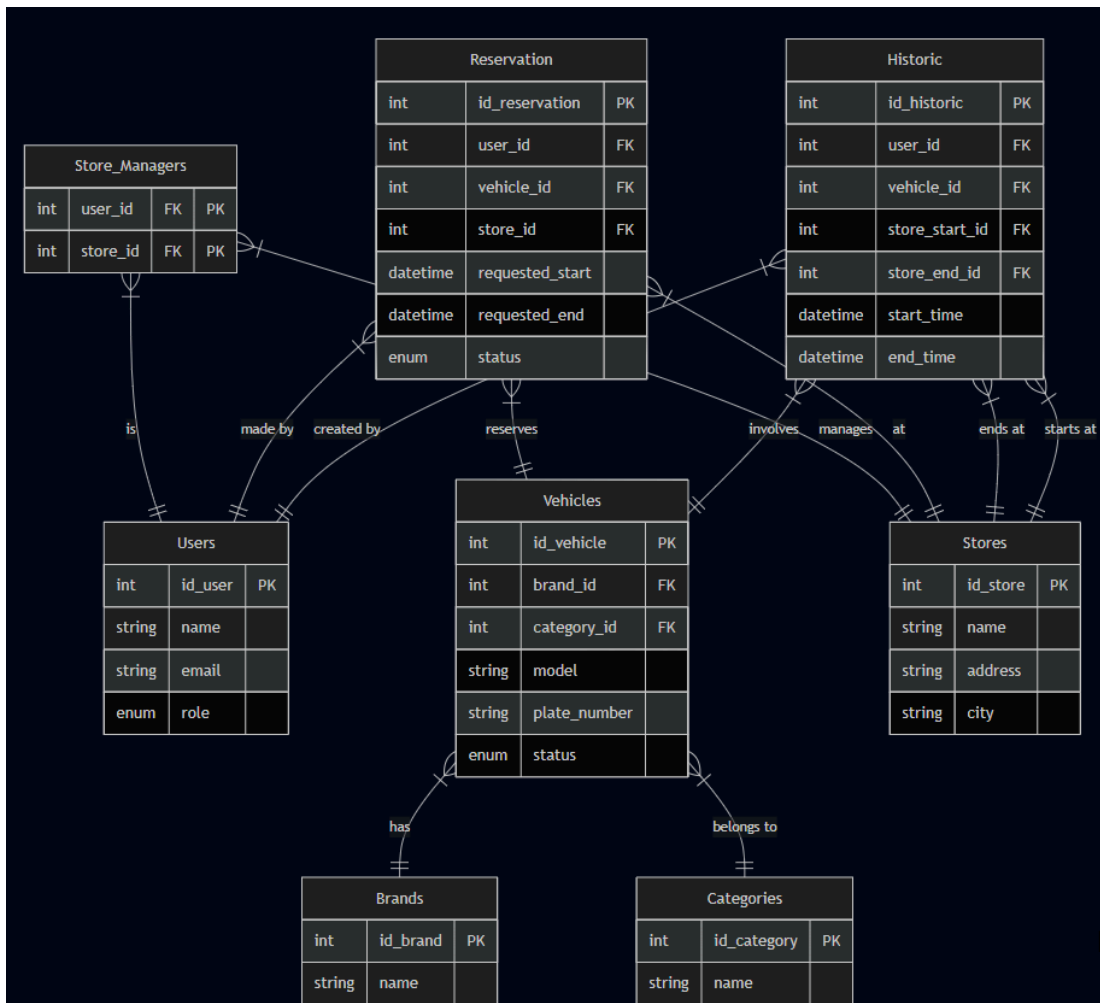
- Les **voitures** et les **catégories de véhicules** sont en relation Many-to-Many, car une voiture peut appartenir à plusieurs catégories et chaque catégorie peut regrouper plusieurs voitures. J'ai donc créé une table de pivot `vehicle_category` pour gérer cette relation.
- Un **utilisateur** peut effectuer plusieurs **réservations**, mais une réservation n'appartient qu'à un seul utilisateur (relation One-to-Many).
- De même, un utilisateur peut laisser plusieurs **avis** sur les voitures, mais chaque avis est associé à un seul utilisateur et une seule voiture (relation One-to-Many pour chaque côté).
- Les relations plus simples, comme les liens entre les voitures et leurs caractéristiques, ont été modélisées directement dans le MCD avec des clés étrangères dans le MLD.

Étant donné que le projet est structuré autour d'une architecture **microservices**, je n'ai pas généré de MPD complet, car chaque microservice gère sa propre base de données et il serait trop complexe de représenter l'ensemble des données dans un seul MPD.

MCD



MLD



Créer la base de données

Pour créer la base de données, j'utilise **JPA (Java Persistence API)** avec Spring Boot.

1. Configuration de la source de données

Avant tout, je configure le fichier **application.properties** (ou **application.yml**) pour définir la connexion à la base de données. Par exemple, pour une base H2 en mémoire :

- URL, utilisateur et mot de passe
- Dialecte Hibernate
- Option **ddl-auto=create** pour créer automatiquement les tables au démarrage de l'application

```
spring.datasource.url=jdbc:mysql://localhost:3308/las_rental
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

2. Définition des entités

Chaque table de la base est représentée par une classe Java annotée avec **@Entity**. Les attributs de la classe correspondent aux colonnes de la table. Par exemple, une entité **Vehicle** possède des champs comme **id**, **brand** ou **model**.

```

@Entity  ⤴ Esteban
@Setter
@Getter
@AllArgsConstructor
@NoArgsConstructor
public class Vehicle {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String model;

    @Column(name = "plateNumber", nullable = false)
    private String plateNumber;

    @Column(name = "status", nullable = false)
    @Enumerated(EnumType.STRING)
    private StatusVehicle status;

    @Column(name = "price_per_day", precision = 10, scale = 2)
    private BigDecimal pricePerDay;

    @ManyToOne
    @JoinColumn(name = "brand_id", nullable = false)
    @JsonManagedReference
    private Brand brand;
}

```

3. Création des repositories

Pour chaque entité, je crée un repository en étendant **JpaRepository**. Cela permet de gérer facilement les opérations CRUD sur la table correspondante.

```

@Repository  6 usages  ⤴ Esteban
public interface VehicleRepository extends JpaRepository<Vehicle, Long> {
}

```

4. Initialisation des données fictives

Pour tester l'application, je crée des données fictives avec **JavaFaker**.

- Une classe annotée **@Component** implémente **CommandLineRunner** pour s'exécuter au démarrage.
- À l'aide de **Faker**, je génère des valeurs aléatoires et les insère dans la base avec les repositories.

```
@Bean
@Component
public class CommandLineRunner implements CommandLineRunner {

    private final BrandRepository brandRepository;
    private final CategoryRepository categoryRepository;
    private final VehicleRepository vehicleRepository;
    private final StoreRepository storeRepository;

    public CommandLineRunner(BrandRepository brandRepository, CategoryRepository categoryRepository, VehicleRepository vehicleRepository, StoreRepository storeRepository) {
        this.brandRepository = brandRepository;
        this.categoryRepository = categoryRepository;
        this.vehicleRepository = vehicleRepository;
        this.storeRepository = storeRepository;
    }

    @Override
    public void run(String[] args) throws Exception {
        // Initialize the database with some data
        brandRepository.save(new Brand("Toyota"));
        brandRepository.save(new Brand("Honda"));
        brandRepository.save(new Brand("Ford"));

        categoryRepository.save(new Category("SUV"));
        categoryRepository.save(new Category("Sedan"));
        categoryRepository.save(new Category("Truck"));
        categoryRepository.save(new Category("Compact"));

        storeRepository.save(new Store("Toulon Centre", "1 Place de la Liberté 83000", "0123456789", "toulon"));
        storeRepository.save(new Store("Toulon Port", "2 Rue de la République 83000", "0987654321", "toulon"));
        storeRepository.save(new Store("Marseille Gare Saint Charles", "Sq. Narvik, 13232 Marseille", "0147852369", "marseille"));

        vehicleRepository.save(new Vehicle("RAV4", "ABC123", new BigDecimal("50.00"), StatusVehicle.AVAILABLE, brandRepository.findByName("Toyota").orElse(null)));
        vehicleRepository.save(new Vehicle("Civic", "XYZ789", new BigDecimal("40.00"), StatusVehicle.AVAILABLE, brandRepository.findByName("Honda").orElse(null)));
        vehicleRepository.save(new Vehicle("F-150", "LMN456", new BigDecimal("60.00"), StatusVehicle.AVAILABLE, brandRepository.findByName("Ford").orElse(null)));
        vehicleRepository.save(new Vehicle("Ka", "ABC456", new BigDecimal("30.00"), StatusVehicle.AVAILABLE, brandRepository.findByName("Ford").orElse(null)));
    }
}
```

5. Utilisation de MongoDB

Pour utiliser **MongoDB**, j'ai utilisé la dépendance **Spring Boot MongoDB**, qui me permet d'exploiter du NoSQL de la même façon que JPA.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Document(collection = "comment")
public class Comment {

    @MongoId
    private String id;

    @Field(name = "vehicle_id")
    private String vehicleId;

    @Field(name = "user_id")
    private String userId;

    private Integer rating;

    private String comment;

    private String timestamp;
}
```

```
@Repository
public interface CommentRepository extends MongoRepository<Comment, Long> {

    List<Comment> findCommentsByVehicleId(String vehicleId);

    Comment findById(String commentId);

    void deleteById(String commentId);
}
```

```
spring.data.mongodb.authentication-database=admin
spring.data.mongodb.database=comments
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27018
spring.data.mongodb.username=root
spring.data.mongodb.password=rootpassword
```

Routes

Microservice	Endpoint	Méthode	Fonctionnalité / Description	Contrôleur / Fichier
Ms-Pricing	/pricing	POST	Calculer / retourner le prix d'un véhicule (VehiclePriceDto)	PricingController (Ms-Pricing/.../PricingController.java)
Ms-Security	/api/auth/login	POST	Authentifier un utilisateur (création cookie JWT)	LoginController
Ms-Security	/api/auth/logout	POST	Déconnexion (invalidation du cookie JWT)	LoginController
Ms-Security	/api/auth/check-header-bearer	POST	Vérifier présence du header Authorization et renvoyer le token brut	LoginController
Ms-Security	/api/auth/register	POST	Inscription utilisateur	LoginController
Ms-Security	/api/user/all	GET	Lister tous les utilisateurs	UserController
Ms-Security	/api/user/{id}	GET	Récupérer un utilisateur par id (requiert X-User-Role=ADMIN)	UserController
Ms-Rental	/api/rental/stores/stores?city={ville}	GET	Lister les agences par ville	StoreController (Ms-Rental/src/main/java/dev/esteban/msrental/controller/StoreController.java)
Ms-Rental	/api/rental/vehicles/all	GET	Lister tous les véhicules	VehicleController (Ms-Rental/.../VehicleController.java)
Ms-Rental	/api/rental/vehicles/{id}	GET	Obtenir un véhicule par id	VehicleController
Ms-Rental	/api/rental/vehicles/available	POST	Rechercher véhicules disponibles (VehicleSearchDto)	VehicleController
Ms-Rental	/api/rental/reservations/create	POST	Créer une réservation (NewReservationDto)	ReservationController

Ms-Rental	/api/rental/reservations/{id}	GET	Obtenir une réservation par id	ReservationController
Ms-Rental	/api/rental/reservations/user/{userId}	GET	Lister les réservations d'un utilisateur	ReservationController
Ms-Rental	/api/rental/reservations/{id}/cancel	PUT	Annuler une réservation	ReservationController
Ms-Rental	/api/rental/rentals/create/{reservationId}	POST	Créer une location depuis une réservation	RentalController (Ms-Rental/.../RentalController.java)
Ms-Rental	/api/rental/rentals/{id}	GET	Obtenir une location par id	RentalController
Ms-Rental	/api/rental/rentals/user/{userId}	GET	Lister les locations d'un utilisateur	RentalController
Ms-Rental	/api/rental/rentals/all	GET	Lister toutes les locations	RentalController
Ms-Rental	/api/rental/rentals/status/{status}	GET	Lister les locations par statut	RentalController
Ms-Rental	/api/rental/rentals/active	GET	Lister les locations actives (IN_PROGRESS)	RentalController
Ms-Rental	/api/rental/rentals/{id}/complete	PUT	Marquer une location comme complétée	RentalController
Ms-Rental	/api/rental/rentals/{id}/return	PUT	Enregistrer le retour du véhicule	RentalController
Ms-Rental	/api/rental/rentals/{id}/cancel	PUT	Annuler une location	RentalController
Ms-Rental	/api/rental/rentals/check-overdue	POST	Vérifier et marquer les locations en retard	RentalController
Ms-Comments	/api/comments/add	POST	Ajouter un commentaire (CommentDto)	CommentController (Ms-Comments/.../CommentController.java)
Ms-Comments	/api/comments/all	GET	Lister tous les commentaires	CommentController

Dans le tableau, on peut voir à quel microservice appartient chaque endpoint, la méthode utilisée, sa fonctionnalité ainsi que le contrôleur associé. On y remarque également l'utilisation de plusieurs méthodes HTTP et une bonne séparation des endpoints pour une meilleure organisation.

Configuration centralisée avec Spring Cloud Config

Pour gérer la configuration de chacun de mes microservices, qui sont des API indépendantes, j'ai utilisé **Spring Cloud Config**. Cela m'a permis d'assurer une bonne cohérence de mon environnement à toutes les étapes du projet (développement, test et production) et de mettre à jour facilement la configuration, comme les URL de la base de données, les mots de passe, les ports du serveur ou encore certaines valeurs spécifiques. Les configurations peuvent être stockées soit en **local**, soit dans un **dépôt GitHub**, ce qui permet une meilleure gestion et un meilleur suivi des versions.

Exemple:

Durant le développement du microservice **Ms-Rental**, chargé de gérer les réservations et les locations, j'ai utilisé une base de données **en mémoire**. Plus tard, pour la production, j'ai utilisé une **vraie base de données MySQL**. À ce moment-là, j'ai dû changer la configuration, et c'est pour cela que j'ai utilisé **Spring Cloud Config**. Voici quelques images pour illustrer le fonctionnement.

```
spring.application.name=config-server
server.port=3030
spring.cloud.config.server.native.search-locations=classpath:/configuration
spring.profiles.active=native
logging.level.org.springframework.cloud.config.server.environment=TRACE

management.endpoints.web.exposure.include=*
management.health.readinessstate.enabled=true
management.health.livenessstate.enabled=true
management.endpoint.health.probes.enabled=true

spring.cloud.config.server.overrides.JWT_KEY=${JWT_KEY}
spring.cloud.config.server.overrides.DB_PASSWORD=${DB_PASSWORD}
```


Ceci est la configuration de mon **Spring Cloud Config**. On peut voir qu'il me permet également de gérer des valeurs sensibles depuis un fichier .env, comme la clé JWT ou le mot de passe de la base de données.

rental.properties

```
spring.application.name=rental
server.port=8081
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
eureka.instance.prefer-ip-address=true

spring.datasource.url=jdbc:mysql://localhost:3308/las_rental
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

management.endpoints.web.exposure.include=*
management.health.readinessstate.enabled=true
management.health.livenessstate.enabled=true
management.endpoint.health.probes.enabled=true
```

rental-dev.properties

```
spring.application.name=rental
server.port=8081
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
eureka.instance.prefer-ip-address=true

spring.datasource.url=jdbc:h2:mem:rentaldb;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true

management.endpoints.web.exposure.include=*
management.health.readinessstate.enabled=true
management.health.livenessstate.enabled=true
management.endpoint.health.probes.enabled=true
```


```
spring.config.import=optional:configserver:http://localhost:3030
spring.profiles.active=dev
spring.cloud.config.name=rental
```

Comme on peut le voir sur ces images, dans la configuration je déclare le port du serveur, la base de données, etc., à partir d'un fichier. Dans la troisième image, j'indique simplement le **endpoint** où le microservice doit chercher la configuration, le **profil** (dev ou default/production) et le **nom du fichier**, ici rental.

Eureka

Eureka me sert pour la **déclaration et la découverte des services**. Chaque microservice s'y enregistre, ce qui permet à **Feign** de communiquer facilement entre eux sans configurer manuellement les adresses. La **Gateway** utilise également Eureka pour faire du **load balancing** et diriger chaque requête vers le bon microservice disponible.

Images:

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2025-09-10T10:20:48 +0200
Data center	default	Uptime	00:03
		Lease expiration enabled	false
		Renews threshold	6
		Renews (last min)	4

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GATEWAY	n/a (1)	(1)	UP (1) - Esteban-PC.mshome.net.gateway:8077
RENTAL	n/a (1)	(1)	UP (1) - Esteban-PC.mshome.net.rental:8081
SECURITY	n/a (1)	(1)	UP (1) - Esteban-PC.mshome.net.security:8082

General Info

Ici, on peut voir comment mes microservices sont enregistrés dans **Eureka** sous leur nom, grâce aux propriétés définies dans la configuration.

```
spring.application.name=rental
server.port=8081
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
eureka.instance.prefer-ip-address=true
```

Feign

Avec les microservices déclarés dans **Eureka**, je peux maintenant communiquer beaucoup plus facilement en faisant des requêtes HTTP, simplement en utilisant le nom du service.

```
@FeignClient("rental") 2 usages  Esteban
public interface MsRentalFeignClient {

    @RequestMapping(method = RequestMethod.POST, value = "/api/rental/vehicle/{id}")  Esteban
    Object getVehicleById(@PathVariable Long id);
}
```

Dans cette image, on peut voir que j'utilise uniquement le nom **rental** et non pas l'adresse du serveur. Cela permet, par exemple, au service des commentaires d'envoyer une requête directement vers le microservice qui gère les locations.

Gateway

Ma **Gateway**, que j'expliquerai plus tard, utilise aussi **Eureka** avec le **load balancer** (un système qui choisit l'instance la moins sollicitée) pour traiter les requêtes. Elle s'appuie sur les microservices déclarés dans Eureka et les redirige vers la bonne instance.

```
Prefixed URI with: /api -> http://localhost:8077/api/auth/login
RouteToRequestUrlFilter start
ReactiveLoadBalancerClientFilter url before: lb://security/api/auth/login
LoadBalancerClientFilter url chosen: http://172.21.48.1:8082/api/auth/login
Will instrument the HTTP request headers [Content-Type:"application/json", User-Agent:"PostmanRuntime/7.46.0", Accept:"/*/*", Cache-Control:"no-cache"]
Client observation {name=http.client.requests(null), error=null, context=name='http.client.requests', contextualName='null', error='null', lowCardinality=true}
[41bd6080-1, L:/172.21.48.1:4045 - R:/172.21.48.1:8082] Handler is being applied: {uri=http://172.21.48.1:8082/api/auth/login, method=POST}
outbound route: 41bd6080, inbound: [d066a0d4-1]
[41bd6080-1, L:/172.21.48.1:4045 - R:/172.21.48.1:8082] Received response (auto-read:false) : RESPONSE(decodeResult: success, version: HTTP/1.1)
```

```

@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(id: "rental-service", PredicateSpec r -> r.path("/rental/**")) BooleanSpec
        .filters( GatewayFilterSpec f -> f
            .prefixPath("/api")
            .addResponseHeader( headerName: "x-Powered-By", headerValue: "Esteban's API")
        ) UriSpec
        .uri("lb://rental")
    )
        .route(id: "security-auth-service", PredicateSpec r -> r.path("/auth/**")) BooleanSpec
        .filters( GatewayFilterSpec f -> f
            .prefixPath("/api")
            .addResponseHeader( headerName: "x-Powered-By", headerValue: "Esteban's API")
        ) UriSpec
        .uri("lb://security")
    )
        .route(id: "security-user-service", PredicateSpec r -> r.path("/user/**")) BooleanSpec
        .filters( GatewayFilterSpec f -> f
            .prefixPath("/api")
            .addResponseHeader( headerName: "x-Powered-By", headerValue: "Esteban's API")
        ) UriSpec
        .uri("lb://security")
    )
        .build();
}

```

Dans cette image, on peut voir que dans l'URI il est écrit lb://security, ce qui correspond au microservice **security** déclaré dans **Eureka**.

Un point que je n'ai pas exploité est **la scalabilité**. Avec **Docker** ou **Kubernetes**, il serait possible de déployer plusieurs instances d'un service très sollicité afin de répartir la charge. C'est une piste d'amélioration que je n'ai pas encore mise en place.

Sécurité et Gateway

Dans ce projet, la sécurité repose sur les dépendances **Spring Security** et **OAuth2**, ainsi qu'un service de génération de **JWT tokens** que j'ai développé. La logique est la suivante : un utilisateur non connecté doit soit créer un compte, soit se connecter s'il en possède déjà un, afin de recevoir un **JWT token**. Une fois le token obtenu, il peut accéder aux fonctionnalités principales réservées aux utilisateurs. Selon le rôle associé à son compte (**user**, **manager**, **admin**), il obtient un niveau d'accès différent : un utilisateur classique accède aux fonctionnalités de base, tandis qu'un manager ou un administrateur accède au **back-office** avec plus ou moins de privilèges.

```
public String createJwtToken(UserLogDto info) { 1 usage  ▲ Esteban
    User user = userRepository.findByEmail(info.getEmail()).orElseThrow(() -> new RuntimeException("Email not found"));
    if (!isValidUser(info)) {
        throw new RuntimeException("Password is incorrect");
    }
    Instant now = Instant.now();
    JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuer("self")
        .issuedAt(now)
        .expiresAt(now.plus(amountToAdd: 1, ChronoUnit.DAYS))
        .subject(info.getEmail())
        .claim(name: "role", user.getRole())
        .claim(name: "userId", user.getId())
        .build();
    JwtEncoderParameters params = JwtEncoderParameters.from(JwsHeader.with(MacAlgorithm.HS256).build(), claims);
    return this.jwtEncoder.encode(params).getTokenValue();
}
```

```

public ResponseEntity<?> loginUser(UserLogDto info, HttpServletResponse response) { 1 usage  Esteban
    try {
        String token = jwtService.createJwtToken(info);

        User user = userRepository.findByEmail(info.getEmail()).orElseThrow(() -> new RuntimeException("User not found"));

        Cookie cookie = new Cookie("JWT", token);
        cookie.setHttpOnly(true);
        cookie.setSecure(true);
        cookie.setPath("/");
        cookie.setMaxAge(86400);

        response.addCookie(cookie);

        Map<String, Object> userInfo = new HashMap<>();
        userInfo.put("email", user.getEmail());
        userInfo.put("role", user.getRole());

        return ResponseEntity.ok(userInfo);
    } catch (RuntimeException e) {
        return ResponseEntity.status(401).body(Map.of("error", e.getMessage()));
    }
    catch (Exception e) {
        return ResponseEntity.status(401).body(Map.of("error", "Problem during login"));
    }
}

```

```

@PostMapping("/login")  Esteban
public ResponseEntity<?> login(@RequestBody UserLogDto info, HttpServletResponse response) {
    ResponseEntity<?> responseEntity = userService.loginUser(info, response);
    if (responseEntity.getStatusCode().is2xxSuccessful()) {
        return ResponseEntity.ok(responseEntity.getBody());
    } else {
        System.out.println(responseEntity.getBody());
        return ResponseEntity.status(responseEntity.getStatusCode()).body(responseEntity.getBody());
    }
}

```

Ici, on peut voir qu'au moment de la connexion, un **cookie sécurisé** (*Secure* et *HttpOnly*) est envoyé à l'utilisateur. Ce cookie contient le **token JWT**, généré à partir des informations de l'utilisateur récupérées dans la base de données du service de sécurité.

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
        .csrf(CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
        .sessionManagement(SessionManagementConfigurer<HttpSecurity> session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(AuthorizationManagerRequestMat... auth -> auth
            .requestMatchers(
                "/api/auth/login",
                "/api/auth/register",
                "/api/auth/validate",
                "/api/auth/logout",
                "/actuator/**"
            )
            .permitAll()
            .requestMatchers(HttpServletRequest request ->
                "internal-service".equals(request.getHeader("X-Internal-Service")))
            .permitAll()
            .anyRequest().authenticated())
        .httpBasic(Customizer.withDefaults())
        .oauth2ResourceServer(OAuth2ResourceServerConfigurer<HttpSecurity> oauth2 -> oauth2
            .jwt(Customizer.withDefaults()))
        .build();
}

```

Grâce à **Spring Security**, certains endpoints ne nécessitent pas de token JWT, comme register, login, logout et validate (qui sert à vérifier la validité d'un token).

La **Gateway** filtre les requêtes grâce au **token JWT**, en s'appuyant sur un endpoint du microservice de sécurité qui valide le token.


```

@PostMapping("/validate")
public ResponseEntity<> validateToken(@RequestHeader("Authorization") String token) {
    System.out.println("Validating token: " + token);
    try {
        String jwt = token.replace(target: "Bearer ", replacement: "");
        Jwt decodedJwt = jwtDecoder.decode(jwt);

        Map<String, Object> response = new HashMap<>();
        response.put("valid", true);
        response.put("subject", decodedJwt.getSubject());
        response.put("role", decodedJwt.getClaim("role"));

        return ResponseEntity.ok(response);
    } catch (Exception e) {
        return ResponseEntity.ok(Map.of(k1: "valid", v1: false, k2: "error", e.getMessage()));
    }
}

```

```

private Mono<Boolean> isValidToken(String token, ServerWebExchange exchange) {
    System.out.println("Token: " + token.substring(0, 10) + "...");

    return WebClientBuilder.build()
        .post()
        .uri(uri: "http://security/api/auth/validate") // Utilisation du schéma lb:// pour Load Balancing
        .header(HttpHeaders.AUTHORIZATION, ...headerValues: "Bearer " + token)
        .retrieve()
        .bodyToMono(Map.class)
        .flatMap(Map r -> {
            System.out.println("Response: " + r);
            Boolean isValid = (Boolean) r.get("valid");
            if (isValid != null && isValid) {
                exchange.getAttributes().put("USER_ROLE", r.get("role"));
                exchange.getAttributes().put("USER_EMAIL", r.get("subject"));
                return Mono.just(data: true);
            }
            System.out.println("Token is invalid");
            return Mono.just(data: false);
        })
        .onErrorResume(Throwable e -> {
            System.err.println("Error: " + e);
            return Mono.just(data: false);
        });
}

```

```

@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
    ServerHttpRequest request = exchange.getRequest();
    if (isSecured(request)) {
        String token = extractTokenFromCookies(request);
        if (token == null) {
            return onError(exchange, HttpStatus.UNAUTHORIZED);
        }
        return isValidToken(token, exchange)
            .flatMap( Boolean isValid -> {
                if (!isValid) {
                    return onError(exchange, HttpStatus.UNAUTHORIZED);
                }

                // Propagation des informations utilisateur aux services en aval
                ServerHttpRequest modifiedRequest = exchange.getRequest().mutate()
                    .header(headerName: "X-User-Role",
                        exchange.getAttribute(name: "USER_ROLE").toString())
                    .header(headerName: "X-User-Email",
                        exchange.getAttribute(name: "USER_EMAIL").toString())
                    .header(headerName: "x-User-Id",
                        exchange.getAttribute(name: "USER_ID") != null ? exchange.getAttribute(name: "USER_ID").toString() : "")
                    .header(HttpHeaders.AUTHORIZATION, headerValues: "Bearer " + token)
                    .build();

                return chain.filter(exchange.mutate().request(modifiedRequest).build());
            });
    }

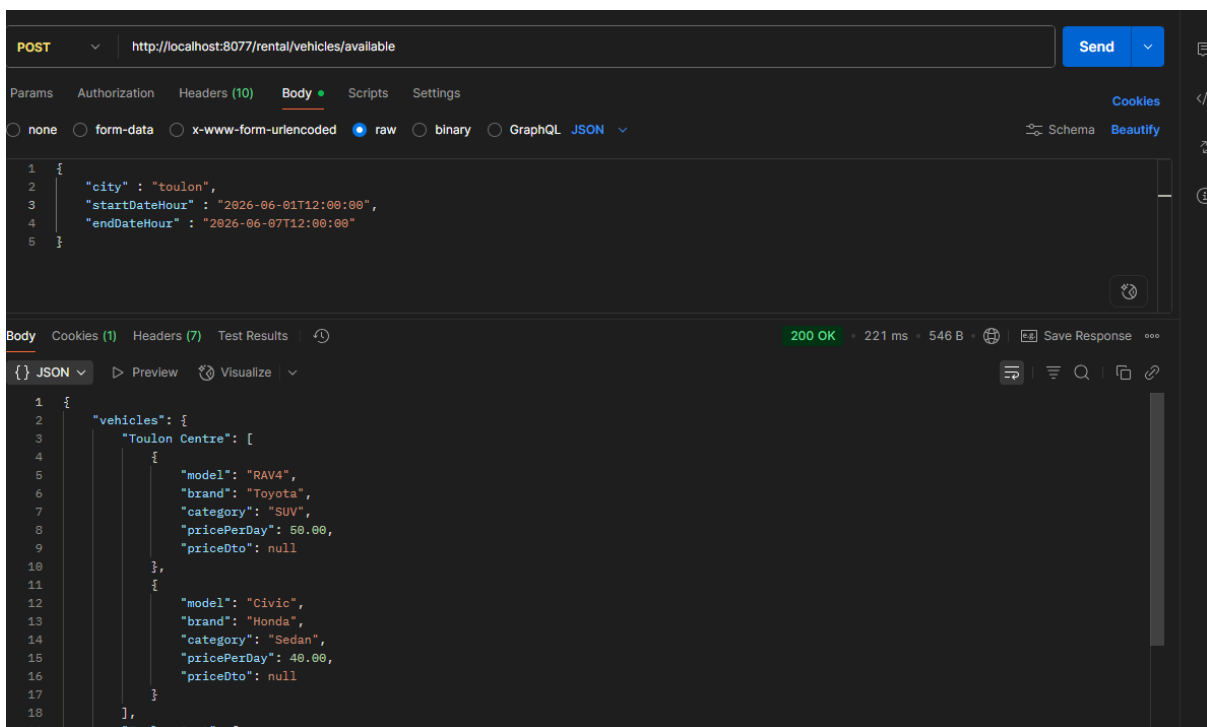
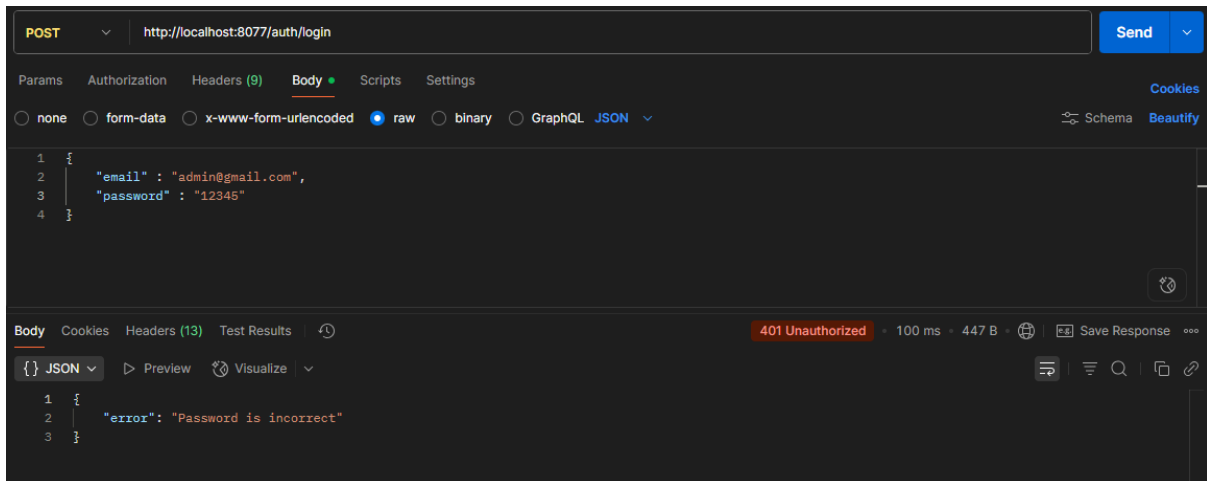
    return chain.filter(exchange);
}

```

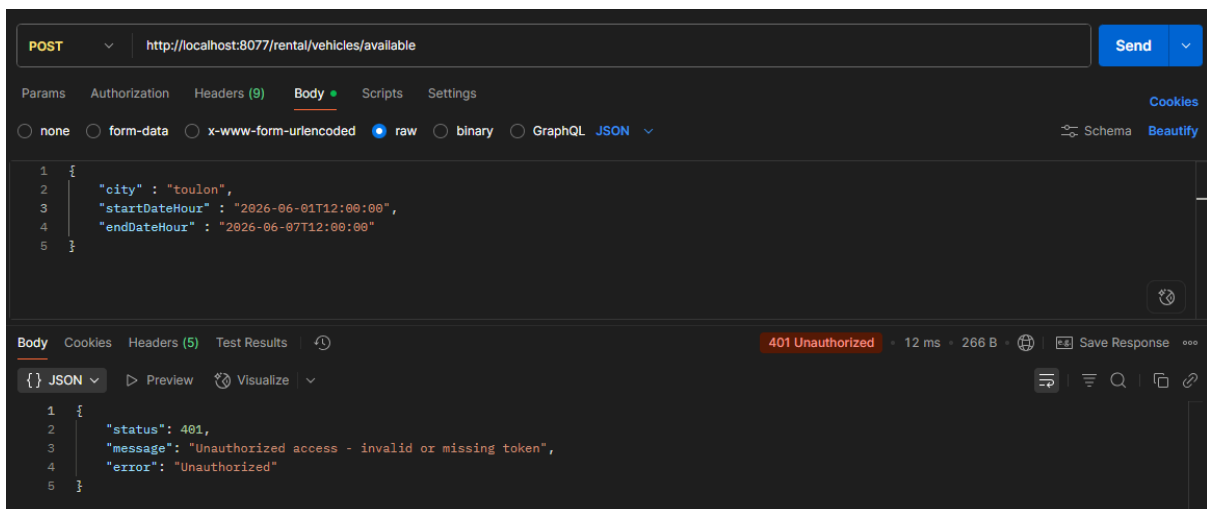
Sur les images, on peut voir que, via l'endpoint validateToken du service de sécurité intégré à ma Gateway, la méthode isValidToken renvoie **true** ou **false** selon que le token est valide ou non. Cette méthode est utilisée dans un **filtre** qui contrôle chaque requête avant qu'elle n'atteigne les microservices nécessitant un token. Si le token n'est pas valide, la requête reçoit un **message 401 Unauthorized**. Dans le cas contraire, la requête atteint le serveur avec des **headers** contenant les informations présentes dans le token (id, rôle et email). Voici des image des test avec postman:

The screenshot shows a Postman test of a login endpoint. The request is a POST to `http://localhost:8077/auth/login` with a JSON body containing `"email": "admin@gmail.com"` and `"password": "123456"`. The response is a `200 OK` with a status of `847 ms` and a body size of `740 B`. A JWT token is stored in the cookies.

Name	Value	Domain	Path	Expires	HttpOnly	Secure
JWT	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1b290aWZzWmliwlc3VlIjoYWRtaW5A2Z1haWwY29tliwlc9sZS16lkFETUIOlwiZXBwIjoxNzU3NTk2MzA3LCJpYXQiOiE3NTc1MDk5MDRlbnVzZjU3ZjZC16MX0uJ2mBy46UKKW3hqXM06u6ifu1R4kKumYbAN5jijHTbhc	localhost	/	Thu, 11 Sep 2025 13:11:...	true	true



Sans cookie



CRUD et fonctionnalités principales

Le cœur de l'application réside dans la création de **réservations utilisateurs** et le suivi des **locations**, ce qui constitue la logique métier. Pour cela, j'ai créé des **services** et des **controllers**, en m'appuyant sur les **repositories JPA** pour effectuer des requêtes SQL. Cela me permet de développer une application capable de gérer toutes les fonctionnalités attendues, comme exemple:

- Pour récupérer les véhicules disponibles à une date

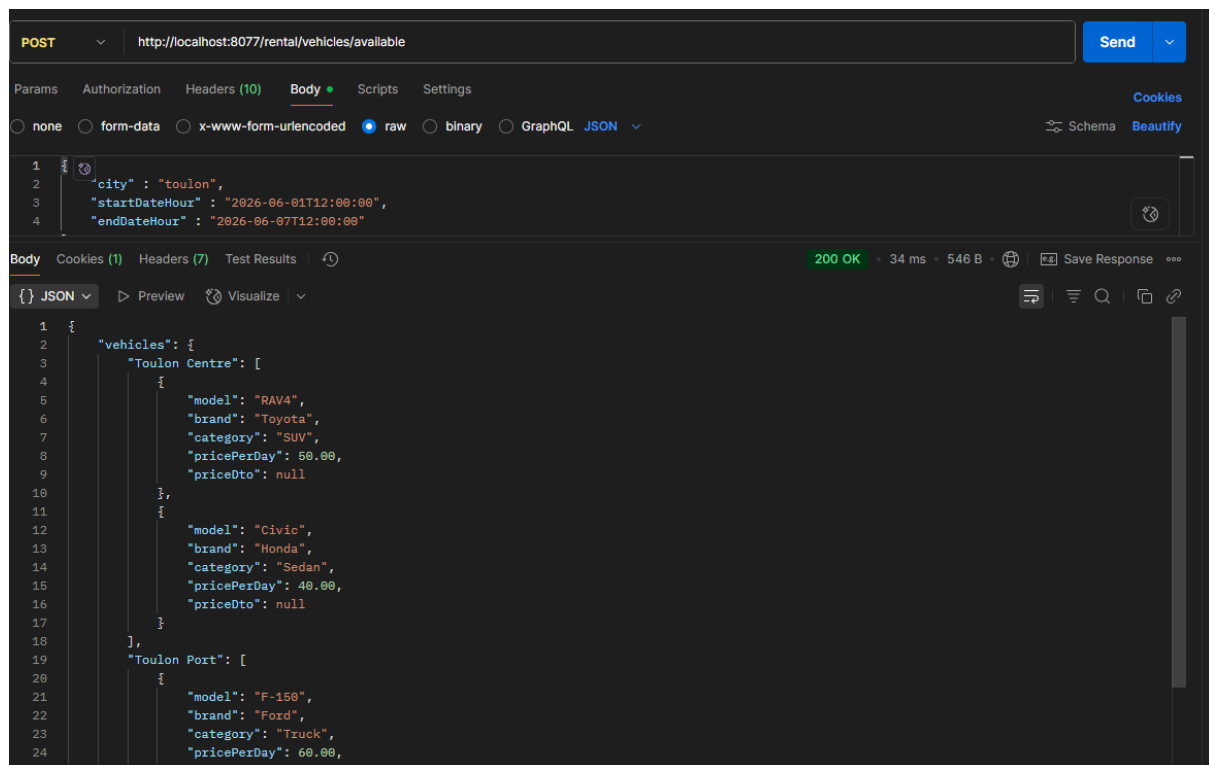
```
@PostMapping("/available")
public ResponseEntity<?> getAvailableCars(@RequestBody VehicleSearchDto vehicleSearchDto) {
    return vehicleService.getAvailableCars(vehicleSearchDto);
}
```

Pour la majorité des requêtes, j'utilise des **DTO (Data Transfer Object)**, qui permettent de gérer le corps JSON transmis dans une requête et de le convertir en **objets Java**.

```
public ResponseEntity<?> getAvailableCars(VehicleSearchDto vehicleSearchDto) {
    List<Store> stores = storeService.getStoresByCity(vehicleSearchDto.getCity());
    if (stores.isEmpty()) {
        return ResponseEntity.badRequest().body("No stores available in this city");
    }
    Map<String, List<VehicleDto>> storeVehicles = new HashMap<>();
    stores.forEach(store -> {
        List<VehicleDto> vehicles = store.getVehicles().stream().filter(vehicle -> {
            boolean isAvailable = vehicleIsAvailable(vehicle);
            boolean isReserved = vehicleIsReservedBetweenTwoDates(vehicle, vehicleSearchDto.getStartDateTime(), vehicleSearchDto.getEndDateTime());
            return isAvailable && !isReserved;
        }).map(vehicle -> {
            try {
                ResponseEntity<PriceDto> response = msPricingFeignClient
                    .getPricesByCar(new VehiclePriceDto(vehicle.getId(),
                        vehicle.getCategory().getName(), vehicle.getPricePerDay(), vehicleSearchDto.getStartDateTime(), vehicleSearchDto.getEndDateTime()));
                if (response.getStatusCode().is2xxSuccessful() && response.getBody() != null) {
                    PriceDto pricesDto = response.getBody();
                    return new VehicleDto(vehicle.getModel(), vehicle.getBrand().getName(), vehicle.getCategory().getName(), vehicle.getPricePerDay(), pricesDto);
                } else {
                    return new VehicleDto(vehicle.getModel(), vehicle.getBrand().getName(), vehicle.getCategory().getName(), vehicle.getPricePerDay(), priceDto: null);
                }
            } catch (Exception e) {
                return new VehicleDto(vehicle.getModel(), vehicle.getBrand().getName(), vehicle.getCategory().getName(), vehicle.getPricePerDay(), priceDto: null);
            }
        }).collect(Collectors.toList());
        storeVehicles.put(store.getName(), vehicles);
    });
    if (storeVehicles.isEmpty()) {
        return ResponseEntity.badRequest().body("No vehicles available in this city");
    }
    AvailableVehicleDto availableVehicleDto = new AvailableVehicleDto(storeVehicles);
    return ResponseEntity.ok(availableVehicleDto);
}
```

Dans cette méthode, on voit que je filtre les voitures disponibles en fonction des données transmises (boutique, date de début et date de fin). J'utilise

également le **service Ms-Pricing** via **Feign Client** pour récupérer les prix. Si le service n'est pas disponible, je peux quand même retourner un résultat, mais sans les prix. Voici le résultat testé avec **Postman**.



- Pour créer une réservation

```
@PostMapping("/create") new *
public ResponseEntity<?> createReservation(@RequestBody NewReservationDto newReservationDto) {
    return reservationService.createReservation(newReservationDto);
}
```

```
@Transactional
public ResponseEntity<?> createReservation(NewReservationDto newReservationDto) {
    try {
        Optional<Vehicle> vehicle = vehicleRepository.findById(newReservationDto.getVehicleId());
        if (vehicle.isEmpty()) {
            return ResponseEntity.badRequest().body("Vehicle not found");
        }
        Optional<Store> store = storeRepository.findById(newReservationDto.getStoreId());
        if (store.isEmpty()) {
            return ResponseEntity.badRequest().body("Store not found");
        }
        boolean isAvailable = isVehicleAvailable(
            newReservationDto.getVehicleId(),
            newReservationDto.getRequestedStartDate(),
            newReservationDto.getRequestedEndDate()
        );
        if (!isAvailable) {
            return ResponseEntity.badRequest().body("Vehicle is not available for the requested dates");
        }
        Reservation reservation = new Reservation(newReservationDto.getUserId(), vehicle.get(), store.get(), newReservationDto.getRequestedStartDate(),
            newReservationDto.getRequestedEndDate(), newReservationDto.getReservationPrice(), newReservationDto.getInsurancePrice(), ReservationStatus.COMPLETED
        );
        Reservation savedReservation = reservationRepository.save(reservation);
        paymentService.createPaymentForReservation(savedReservation, PaymentType.RESERVATION, newReservationDto.getReservationPrice());
        if (newReservationDto.getInsurancePrice().compareTo(BigDecimal.ZERO) > 0) {
            paymentService.createPaymentForReservation(savedReservation, PaymentType.INSURANCE, newReservationDto.getInsurancePrice());
        }
        return ResponseEntity.ok(savedReservation);
    } catch (Exception e) {
        return ResponseEntity.internalServerError().body("An error occurred while creating the reservation: " + e.getMessage());
    }
}
```

```
private boolean isVehicleAvailable(Long vehicleId, LocalDateTime requestedStartDate, LocalDateTime requestedEndDate) { 1 usage new *
    List<Reservation> overlappingReservations = reservationRepository.findOverlappingReservations(vehicleId, requestedStartDate, requestedEndDate);
    return overlappingReservations.isEmpty();
}
```

```
@Query("SELECT r FROM Reservation r WHERE r.vehicle.id = :vehicleId AND " + 1 usage new *
        "r.status = 'COMPLETED' AND " +
        "((r.requested_start_date <= :endDate AND r.requested_end_date >= :startDate))")
List<Reservation> findOverlappingReservations(@Param("vehicleId") Long vehicleId,
                                             @Param("startDate") LocalDateTime startDate,
                                             @Param("endDate") LocalDateTime endDate);
```

Pour créer une réservation, j'utilise également un **DTO**. Je vérifie d'abord que les informations sont correctes, puis j'appelle la méthode `isVehicleAvailable`, qui utilise une requête SQL personnalisée dans le repository `findOverlappingReservations`, pour m'assurer qu'aucune autre réservation n'existe pour ce véhicule. Ensuite, je crée le **paiement correspondant**, puis la réservation elle-même.

- Pour créer un commentaire

Pour créer des commentaires, j'ai un **service dédié** qui gère les commentaires par rapport à un véhicule. Ce service utilise une **base de données NoSQL**, adaptée pour gérer de grandes quantités de données, permettre une lecture/écriture rapide, et faciliter la scalabilité.

```
@PostMapping("/{add}") 1 usage new *
public ResponseEntity<?> addComment(@RequestBody CommentDto comment) {
    try {
        String response = commentService.saveComment(comment);
        return ResponseEntity.ok().body(response);
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(e.getMessage());
    } catch (Exception e) {
        return ResponseEntity.status(500).body("Internal server error");
    }
}

@GetMapping("/{all}") 1 usage new *
public ResponseEntity<?> getAllComments() {
    try {
        return ResponseEntity.ok().body(commentService.getAllComments());
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body(e.getMessage());
    } catch (Exception e) {
        return ResponseEntity.status(500).body("Internal server error");
    }
}
```

```

public String saveComment(CommentDto comment) { 1 usage ▴ Esteban *
    try {
        Comment newComment = new Comment();

        if (comment.getVehicleId() == null) throw new RuntimeException("Vehicle ID cannot be null.");
        if (msRentalFeignClient.getVehicleById(Long.valueOf(comment.getVehicleId())) == null)
            throw new RuntimeException("Vehicle with ID " + comment.getVehicleId() + " does not exist.");
        newComment.setVehicleId(comment.getVehicleId());

        if (comment.getUserId() == null) throw new RuntimeException("User ID cannot be null.");
        try {
            msSecurityFeignClient.getUserById(Long.valueOf(comment.getUserId()));
        } catch (Exception e) {
            throw new RuntimeException("User with ID " + comment.getUserId() + " does not exist.");
        }
        newComment.setUserId(comment.getUserId());

        if (comment.getRating() < 1 || comment.getRating() > 5)
            throw new RuntimeException("Rating must be between 1 and 5.");
        newComment.setRating(comment.getRating());

        if (comment.getComment() == null || comment.getComment().isEmpty())
            throw new RuntimeException("Comment cannot be empty.");
        newComment.setComment(comment.getComment());

        newComment.setTimestamp(LocalDate.now().toString());
        System.out.println(newComment);
        commentRepository.save(newComment);
        return "Comment saved successfully";
    } catch (Exception e) {
        throw new RuntimeException("Error saving comment: " + e.getMessage());
    }
}

public List<Comment> getAllComments() { 1 usage ▴ Esteban
    return commentRepository.findAll();
}

```

Ici, on peut voir une méthode pour créer un commentaire qui vérifie que chaque donnée existe, en effectuant également des requêtes vers les autres services via **Feign Client**, et que les informations sont valides. J'ai également une requête `getAllComments` qui, grâce au **repository MongoDB**, permet de récupérer tous les commentaires.