Pour ce projet de gestion d'un parc automobile, voici un plan détaillé pour le développement en utilisant une architecture microservices, avec une durée de travail limitée à un mois. Chaque point aborde les étapes et exigences nécessaires :

1. Cahier des Charges et Modèle UML

Objectif :

Concevoir un système pour gérer un parc automobile, incluant la gestion des utilisateurs, des véhicules, des réservations, des coûts de maintenance, et des notifications.

Fonctionnalités Principales :

1. Gestion des Utilisateurs :

- Créer, modifier, supprimer et consulter des utilisateurs.

- Authentification et autorisation des utilisateurs.

2. Gestion des Véhicules:

- Ajouter, modifier, supprimer et consulter les véhicules disponibles dans le parc.

- Suivi de l’état des véhicules (disponible, en maintenance, réservé).

3. Réservation de Véhicules :

- Réserver des véhicules pour une période donnée.

- Annuler ou modifier des réservations.

4. Gestion des Coûts de Maintenance :

- Enregistrer et consulter les coûts de maintenance associés à chaque véhicule.

- Générer des rapports sur les coûts d'entretien.

5. Notifications\*:

- Envoyer des notifications par email/sms pour des rappels de maintenance, des confirmations de réservation, etc.

Modele UML :

- \*Diagramme de Cas d'Utilisation\* : Présenter les fonctionnalités principales.

- \*Diagramme de Classe\* : Détailler les entités comme Utilisateur, Véhicule, Réservation, Coût, Notification.

- \*Diagramme de Séquence\* : Pour montrer le flux des interactions entre les microservices lors des processus critiques comme la réservation d’un véhicule.

- \*Diagramme de Déploiement\* : Présenter l’architecture des microservices et leurs interactions.

2. Domain-Driven Design (DDD)

#### \*Bounded Contexts et Microservices Requis :\*

1. Service Utilisateur (Java - Spring Boot)

- \*Contexte\* : Gestion des utilisateurs, authentification, autorisation.

- \*Domaine\* : Utilisateur.

- \*Entités\* : Utilisateur, Rôle.

- \*Base de données\* : MySQL ou PostgreSQL.

2. Service Véhicule (Python - Flask)

- \*Contexte\* : Gestion des véhicules, état, maintenance.

- \*Domaine\* : Véhicule.

- \*Entités\* : Véhicule, État, Type.

- \*Base de données\* : MySQL ou PostgreSQL

Si la gestion des véhicules doit inclure des données non structurées ou semi-structurées (ex : photos, logs de maintenance), une base NoSQL comme MongoDB pourrait également être utile pour une plus grande flexibilité.

3. Service Réservation (Node.js - Express.js)

- \*Contexte\* : Gestion des réservations de véhicules.

- \*Domaine\* : Réservation.

- \*Entités\* : Réservation, Utilisateur, Véhicule.

- \*Base de données\* : MySQL (pour les transactions ACID).

4. Service Maintenance et Coûts (Java - Spring Boot)

- \*Contexte\* : Gestion des coûts et des opérations de maintenance.

- \*Domaine\* : Maintenance.

- \*Entités\* : Opération, Coût, Véhicule.

- \*Base de données\* : PostgreSQL (pour la gestion structurée des coûts).

Service Maintenance :

Enregistrement de la maintenance : Lorsque de la maintenance est effectuée sur un véhicule, un enregistrement est ajouté dans la table Maintenance avec les détails de l'entretien.

Gestion des coûts : Lors de l'enregistrement d'une maintenance, les coûts associés doivent être ajoutés à l'entité Maintenance.

Calcul du prochain entretien : Après chaque maintenance, vous pouvez calculer la prochaine date d'entretien ou le prochain kilométrage où une maintenance est nécessaire (par exemple, tous les 10 000 km ou tous les 6 mois).

5. Service Notification (Node.js - Express.js)

- \*Contexte\* : Gestion des notifications et alertes aux utilisateurs.

- \*Domaine\* : Notification.

- \*Entités\* : Notification, Type, Utilisateur.

- \*Base de données\* : MongoDB - Cassandra (Les notifications peuvent avoir un format flexible (par exemple, notifications texte, notifications avec des objets JSON, etc.) et peuvent être créées et lues rapidement. MongoDB est idéal pour le stockage de documents avec un schéma flexible, tandis que Cassandra peut être utilisé pour gérer un grand nombre de notifications à grande échelle.)

Ce service est avec base de données (notifications persistées)

Pour garder une trace des notifications envoyées (par exemple, pour des raisons de suivi, d'historique, ou pour permettre à un utilisateur de consulter ses anciennes notifications), une base de données peut être utile.

Dans ce cas, on peut stocker :

• L'utilisateur ciblé (à qui la notification est envoyée).

• Le type de notification (par exemple : alerte de maintenance, réservation confirmée).

• Le statut de la notification (lue, non lue, etc.).

• Le contenu de la notification.

Une base de données NoSQL comme MongoDB peut être particulièrement adaptée si vous avez besoin de flexibilité pour stocker des notifications avec des formats variés, tandis qu'une base relationnelle (comme PostgreSQL ou MySQL) pourrait être utilisée pour un stockage plus structuré avec des tables dédiées aux notifications.

Cas d'utilisation :

Historique des notifications pour chaque utilisateur.

Rapports et statistiques sur les notifications envoyées.

Gestion des notifications lues/non lues.

3. Développement des Microservices

#### \*Langages Utilisés :\*

- \*Service Utilisateur\* : Java avec Spring Boot.

- \*Service Véhicule\* : Python avec Flask.

- \*Service Réservation\* : Node.js avec Express.js.

- \*Service Maintenance et Coûts\* : Java avec Spring Boot.

- \*Service Notification\* : Node.js avec Express.js.

Consignes Techniques :

- \*Interfaces de Communication\* :

- \*REST\* : Pour la majorité des échanges standard.

- \*GraphQL\* : Pour des requêtes de données complexes.

- \*gRPC\* : Pour des communications performantes entre certains microservices.

- \*WebSocket\* : Pour les notifications en temps réel.

- Documentation des APIs : Utilisation de \*Swagger\* pour chaque microservice.

- \*Tolérance aux pannes et résilience\* : Utiliser des bibliothèques comme \*Hystrix\* (Java) ou \*Circuit Breaker\* pour chaque service.

- \*Surveillance et journalisation\* : Intégrer \*Prometheus\* pour la surveillance et \*ELK (Elasticsearch, Logstash, Kibana)\* pour la journalisation.

4. Bases de Données

5. Microservices Techniques

- \*API Gateway\* : Utilisation de \*Spring Cloud Gateway\* ou \*Kong\* pour gérer le routage des requêtes et la sécurité (OAuth2, JWT).

- \*Service de Découverte\* : Utilisation de \*Eureka\* pour la découverte des services.

- \*Gestion de la Configuration\* : Utilisation de \*Spring Cloud Config Server\* pour centraliser la configuration des microservices.

6. Sécurité

- \*Authentification et Autorisation\* : Implémentation avec \*OAuth2\* et \*JWT\* pour sécuriser les communications entre le frontend et les microservices.

- \*Contrôle d'accès\* : Gestion des rôles et permissions pour restreindre l'accès à certaines fonctionnalités.

7. Déploiement avec Kubernetes

- \*Cluster Kubernetes\* : Utiliser \*Minikube\* pour le développement local et \*GKE (Google Kubernetes Engine)\* ou \*AKS (Azure Kubernetes Service)\* pour le déploiement en production.

- \*Déploiement des microservices\* avec des fichiers \*YAML\* pour la configuration des déploiements, services, et Ingress.

- \*Équilibrage de charge\* : Utiliser \*Kubernetes Ingress\* pour la gestion des entrées HTTP.

8. Développement Frontend

- \*Choix de Technologie\* : \*Angular\* pour sa robustesse et son intégration avec REST/GraphQL.

- \*Fonctionnalités Frontend\* :

- Authentification des utilisateurs.

- Visualisation des véhicules et gestion des réservations.

- Affichage des notifications en temps réel.

- Gestion des coûts et des statistiques de maintenance.

9. Rapport et Présentation

- \*Rapport Structuré\* :

- \*Introduction\* : Objectifs et portée du projet.

- \*Analyse Fonctionnelle\* : Détails des fonctionnalités et des cas d'utilisation.

- \*Conception\* : Diagrammes UML, choix technologiques.

- \*Implémentation\* : Langages, frameworks, microservices et leurs responsabilités.

- \*Déploiement\* : Stratégie de déploiement avec Kubernetes.

- \*Sécurité\* : Mise en œuvre de OAuth2 et JWT.

- \*Résilience et Surveillance\* : Tolérance aux pannes, journaux et surveillance.

- \*Frontend\* : Fonctionnalités et interfaces utilisateur.

- \*Présentation\* : Slide PowerPoint avec les points clés, démonstrations, et résultats.

Conclusion

Ce projet de gestion d'un parc automobile avec 5 microservices est ambitieux pour une période d'un mois, mais réalisable avec une approche agile et des priorités claires pour chaque phase. Priorisez d'abord les fonctionnalités essentielles et la communication entre les microservices pour assurer un prototype fonctionnel à la fin du mois.

• Utilisation de GraphQL :

1. Accès aux Données Complexes : Si vos microservices doivent souvent retourner des données combinées (par exemple, les informations d'un utilisateur et ses réservations de véhicules), GraphQL peut regrouper ces données dans une seule requête.

2. Réduction du Nombre de Requêtes API : Avec une seule requête GraphQL, vous pouvez obtenir des informations provenant de plusieurs microservices, ce qui simplifie l'architecture de communication.

- Architecture avec GraphQL:

• Service Gateway avec GraphQL

Utilisez un API Gateway qui intègre GraphQL pour centraliser l'accès aux microservices. Cela permet de faire des requêtes GraphQL qui récupèrent des données de plusieurs microservices en une seule fois.

- API Gateway avec GraphQL pourrait être développé en utilisant Node.js avec Apollo Server ou Express.js.

- L'API Gateway peut aussi gérer l'authentification et l'autorisation avec OAuth2 et JWT.

• Microservices Individuels

Chaque microservice continue de fonctionner indépendamment avec une API REST traditionnelle, mais le Gateway GraphQL peut les interroger pour rassembler les données :

1- Service Utilisateur (Java - Spring Boot) :

• Peut exposer une API REST classique.

• Le Gateway GraphQL interroge le service utilisateur pour obtenir des informations sur les utilisateurs.

2- Service Véhicule (Python - Flask) :

• Peut être interrogé via des endpoints REST pour les données des véhicules.

• Les requêtes GraphQL provenant du Gateway peuvent combiner des données des véhicules avec des informations de réservation, si nécessaire.

3- Service Réservation (.Net) :

• Expose ses données via une API REST.

• Le Gateway peut récupérer des informations sur les réservations et les combiner avec les informations des utilisateurs ou des véhicules.

4- Service Maintenance et Coût (Java - Spring Boot) :

• Les données de coûts et de maintenance peuvent être requêtées via REST.

• Le Gateway GraphQL peut assembler les coûts associés à un véhicule ou les détails de maintenance spécifiques.

5- Service Notification (Python) :

• Les notifications peuvent être exposées via REST.

• Utilisation de WebSocket pour les notifications en temps réel.

Les utilisateurs de l'application :

1. Administrateur

L'administrateur est le rôle clé dans une application, ayant un contrôle total sur tous les aspects du système. Ce rôle est essentiel pour la gestion de l'application.

Permissions principales :

Gérer les utilisateurs et leurs rôles.

Gérer les véhicules et leurs informations.

Superviser les maintenances, les rapports et les statistiques.

Gérer la configuration du système et des paramètres généraux.

2. Gestionnaire de parc

Ce rôle est essentiel pour la gestion de la flotte de véhicules et de leur maintenance. Il s'assure que les véhicules sont en bon état de fonctionnement et que la maintenance est correctement planifiée et suivie.

Permissions principales :

Ajouter, modifier, supprimer des véhicules.

Planifier et suivre les maintenances des véhicules.

Visualiser l'historique des maintenances des véhicules.

Gérer les informations relatives aux véhicules (kilométrage, état, etc.).

3. Technicien de maintenance /Mecaniciens

Le technicien de maintenance est responsable de l'exécution des tâches de maintenance sur les véhicules. Ce rôle est crucial pour garantir que les véhicules sont entretenus de manière appropriée.

Permissions principales :

Effectuer des maintenances sur les véhicules.

Mettre à jour l'état des véhicules après chaque intervention.

Accéder aux rapports et historiques de maintenance des véhicules.

4. Utilisateur client(Conducteurs)

Ce rôle permet aux utilisateurs ayant un accès limité de visualiser les informations relatives aux véhicules, de suivre les maintenances et d'effectuer certaines actions de base sans affecter l'administration du système.

Permissions principales :

Consulter les informations sur les véhicules.

Soumettre des demandes de maintenance pour les véhicules.

Mettre à jour certaines informations de son profil.

Gestion des entretiens pour un parc automobile

Dans ce contexte, la gestion des entretiens concerne spécifiquement la maintenance des véhicules du parc automobile. Les fonctionnalités typiques incluent :

Suivi des véhicules : Gestion des informations de chaque véhicule.

Planification des entretiens : Programmation des entretiens (préventifs et correctifs) pour chaque véhicule.

Historique des entretiens : Enregistrement des détails des interventions passées.

Gestion des pièces détachées : Suivi des pièces utilisées pour la maintenance.

Alertes et notifications : Rappels pour les entretiens à venir basés sur les kilomètres parcourus ou la date.

Fonctionnalités du Service Véhicule :

Création d'un Véhicule : Ajouter un nouveau véhicule au système.

Mise à jour des informations d'un Véhicule : Modifier les détails d'un véhicule (comme le kilométrage ou l'état).

Suppression d'un Véhicule : Retirer un véhicule du parc automobile.

Consultation des Détails d'un Véhicule : Afficher les informations détaillées d'un véhicule spécifique.

Liste de Tous les Véhicules : Voir l'ensemble des véhicules enregistrés dans le parc.

Filtrage et Recherche de Véhicules : Recherche par marque, modèle, année, statut, etc.

Suivi du Kilométrage : Mise à jour des kilomètres parcourus pour planifier la maintenance préventive.

Gestion du Statut des Véhicules : Indication si un véhicule est disponible, en maintenance, ou hors service.

Entités pour le Service Véhicule

Pour gérer ces fonctionnalités, les entités suivantes sont pertinentes :

Vehicle (Véhicule) :

id: Identifiant unique du véhicule.

registrationNumber: Numéro d'immatriculation.

brand: Marque du véhicule (ex : Toyota, BMW).

model: Modèle du véhicule (ex : Corolla, X5).

year: Année de fabrication.

mileage: Kilométrage actuel.

fuelType: Type de carburant (essence, diesel, hybride, électrique).

status: Statut du véhicule (Disponible, En maintenance, Hors service).

lastMaintenanceDate: Date du dernier entretien.

VehicleServiceRecord (Enregistrement de Service du Véhicule) :

id: Identifiant unique du service.

vehicleId: Référence au véhicule concerné.

serviceDate: Date du service.

serviceType: Type de service (Révision, Réparation, Remplacement de pièces).

description: Description des travaux effectués.

cost: Coût de la maintenance.

• Implementation de Sécurité et Conformité :

Description : Garantir la sécurité des véhicules est une priorité absolue. Le système doit veiller à ce que tous les véhicules respectent les normes de sécurité en vigueur et que les entretiens soient réalisés dans les délais impartis.

• Fonctionnalité de sécurité des véhicules :

1. Microservice Vehicule-Service :

Ce microservice est dédié à la gestion des informations de base de chaque véhicule, y compris leur état général et conformité.

Fonctionnalités :

- Gestion des informations sur chaque véhicule (marque, modèle, immatriculation, etc.).

- Suivi de la conformité de chaque véhicule aux normes de sécurité.

- Mise à jour du statut de conformité après chaque maintenance.

- Accès aux informations sur la dernière maintenance pour chaque véhicule.

Entités :

• Vehicle : Gère toutes les informations de base sur les véhicules.

• Champs supplémentaires : isSafetyCompliant pour indiquer si un véhicule est conforme aux normes de sécurité.

Interactions entre Microservices :

• Mise à jour de la conformité :

Lorsqu'une maintenance est effectuée via le Maintenance-Service, une notification est envoyée au Vehicule-Service pour mettre à jour le statut de conformité du véhicule (isSafetyCompliant).

• Implementation de Realisation des entretiens dans les delais imperatifs :

• Microservice Entretien

Ce microservice est responsable de la gestion des entretiens des véhicules : planification, suivi, gestion des statuts, etc.

Il interagira avec le microservice Véhicule pour obtenir des informations sur les véhicules et mettre à jour les entretiens dans la base de données.

• Microservice Notification

Ce microservice est responsable de l'envoi des notifications aux utilisateurs (par exemple, gestionnaires de parc, responsables de la maintenance) en cas de rappel ou de retard pour un entretien.

Il sera déclenché par le microservice Entretien lorsqu'un entretien approche de sa date limite ou est en retard.

Étapes pour l'implémentation

1. Microservice Entretien

Le microservice Entretien sera responsable des tâches suivantes :

- Gérer les informations d'entretien des véhicules.

- Planifier les entretiens périodiques.

- Calculer la date de l'entretien suivant.

- Mettre à jour le statut des entretiens (en cours, terminé, en retard).

- Déclencher une notification au microservice Notification si l'entretien est en retard ou doit être effectué.

• Modèle Entretien

Voici un modèle simplifié pour représenter l'entité Entretien dans le microservice Entretien.

@Entity

public class Entretien {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private Long vehiculeId; // Lien avec le véhicule

private Date dateEntretien; // Date de l'entretien planifié

private Date prochainEntretien; // Date de l'entretien suivant calculée

private String typeEntretien; // Type d'entretien (changement d'huile, inspection, etc.)

private String statut; // Statut (Planifié, Terminé, En retard)

// Getters et Setters

}

• Planification des entretiens

Le microservice Entretien doit calculer et planifier les prochains entretiens en fonction d'un intervalle fixe ou basé sur les kilomètres.

public Date calculerProchainEntretien(Date dernierEntretien, int intervalleMois) {

Calendar calendar = Calendar.getInstance();

calendar.setTime(dernierEntretien);

calendar.add(Calendar.MONTH, intervalleMois); // Intervalle en mois

return calendar.getTime();

}

• Vérification et mise à jour du statut

Chaque jour ou à intervalles réguliers, une tâche planifiée dans Entretien peut vérifier si les entretiens doivent être réalisés, et mettre à jour le statut des entretiens si nécessaire.

@Scheduled(fixedRate = 86400000) // Chaque jour

public void verifierEntretiens() {

List<Entretien> entretiens = entretienRepository.findByStatut("Planifié");

for (Entretien entretien : entretiens) {

if (entretien.getProchainEntretien().before(new Date())) {

entretien.setStatut("En retard");

entretienRepository.save(entretien);

// Envoi de la notification au service Notification

notificationService.envoyerNotification(entretien);

}

}

}

2. Microservice Notification

Le microservice Notification est responsable de l'envoi de notifications par différents moyens (email, SMS, ou notifications dans l'application) pour alerter les utilisateurs concernant les entretiens en retard ou à venir.

• Modèle de Notification

Voici un modèle pour l'entité Notification.

@Entity

public class Notification {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private Long entretienId; // Lien vers l'entretien concerné

private String message; // Message de la notification

private String type; // Email, SMS, etc.

private String statut; // Envoi réussi, échoué

// Getters et Setters

}

• Envoi de Notifications

Lorsqu'un entretien devient en retard ou qu'il approche, le microservice Entretien appelle le microservice Notification pour envoyer un rappel ou une alerte.

@Service

public class NotificationService {

@Autowired

private RestTemplate restTemplate;

public void envoyerNotification(Entretien entretien) {

Notification notification = new Notification();

notification.setEntretienId(entretien.getId());

notification.setMessage("L'entretien du véhicule " + entretien.getVehiculeId() + " est en retard !");

notification.setType("Email"); // Exemple d'email

notification.setStatut("Non envoyé");

// Appel du microservice Notification via RestTemplate

restTemplate.postForObject("http://notification-service/send", notification, Notification.class);

}

}

• Microservice Notification : API d'envoi

Le microservice Notification expose une API REST pour recevoir les notifications à envoyer.

@RestController

@RequestMapping("/send")

public class NotificationController {

@PostMapping

public ResponseEntity<String> envoyerNotification(@RequestBody Notification notification) {

// Logique d'envoi de notification (email, SMS, etc.)

emailService.envoyerEmail(notification.getMessage());

return ResponseEntity.ok("Notification envoyée");

}

}

3. Communication entre Microservices

Le microservice Entretien peut envoyer des messages au microservice Notification via REST API ou un mécanisme de message queue (par exemple, RabbitMQ ou Kafka) pour une meilleure scalabilité.

Dans l'exemple précédent, nous avons utilisé RestTemplate pour envoyer une notification de manière synchrone











