Documentation Technique – EcoRide

1. Réflexion Technologique Initiale

Au début du projet, j'ai commencé par analyser les contraintes techniques imposées par le cahier des charges. Il fallait impérativement utiliser une base de données relationnelle (comme PostgreSQL, MySQL ou MariaDB) et une base de données NoSQL (comme MongoDB). Le déploiement de l'application devait aussi se faire sur une plateforme cloud.

L'analyse des différentes user stories m'a permis d'identifier plusieurs défis majeurs : il fallait gérer plusieurs rôles d'utilisateurs (utilisateur, employé, administrateur), mettre en place un système de recherche et de filtrage assez poussé pour les covoiturages, gérer la disponibilité des places en temps réel, envoyer des notifications par email, afficher des statistiques, et enfin sécuriser efficacement l'accès à l'application.

2. Architecture et Technologies Utilisées

J'ai opté pour une architecture en trois couches distinctes : le front-end (interface utilisateur), le back-end (logique métier et API), et les bases de données. Le front-end a été développé avec React et Vite. J'ai utilisé TypeScript pour sécuriser le typage, et Tailwind CSS pour le design. La navigation a été gérée avec React Router, les requêtes avec Axios, et la gestion des états asynchrones via React Query. Côté formulaire, j'ai utilisé React Hook Form couplé avec Zod pour la validation des données.

Le back-end repose sur Node.js avec Express, également en TypeScript. Pour la base de données relationnelle, j'ai utilisé PostgreSQL avec Prisma comme ORM, ce qui m'a permis d'écrire des requêtes plus rapidement et de garder une bonne cohérence des données. MongoDB, manipulé via Mongoose, m'a servi pour stocker des données plus dynamiques comme les logs ou les statistiques quotidiennes. L'authentification se fait via JWT, et les mots de passe sont chiffrés avec bcrypt.

3. Mise en Place de l'Environnement de Développement

L'environnement de travail a été structuré avec deux dossiers principaux : un pour le client et un pour le serveur. Chaque dossier a sa propre configuration avec un fichier package.json. Pour le développement local, j'ai utilisé Docker afin de lancer rapidement PostgreSQL et MongoDB dans des conteneurs isolés. Un fichier docker-compose.yml centralise toute la configuration.

J'ai également configuré des fichiers .env pour gérer les variables d'environnement (clés secrètes, identifiants DB, etc.), tout en veillant à ne jamais les exposer dans le code source. Le front-end communique avec l'API via une URL définie dans un fichier .env.local spécifique.

Le projet est versionné avec Git et hébergé sur GitHub. J'ai suivi une convention stricte avec une branche principale (main), une branche de développement (develop), et des branches dédiées pour chaque fonctionnalité. Les tests sont réalisés avant chaque fusion dans develop, et la version stable est poussée dans main pour le déploiement.

4. Base de Données et Modèles

J’ai conçu un schéma de données relationnel basé sur PostgreSQL pour les entités principales comme les utilisateurs, les trajets, les véhicules, ou encore les participations. J’ai également créé un schéma NoSQL pour MongoDB, destiné à stocker les statistiques quotidiennes de l’application (nombre de trajets, crédits gagnés, etc.) et les logs d’activité utilisateur.

Chaque base est optimisée pour son usage : relationnelle pour les données structurées et critiques, NoSQL pour les données plus libres et analytiques.

5. Diagrammes UML et Séquences

Pour documenter le fonctionnement de l’application, j’ai réalisé des diagrammes de cas d’usage et de séquence. Par exemple, la création d’un trajet par un conducteur suit une série d’étapes : sélection du véhicule, remplissage du formulaire, enregistrement en base, puis log de l’action. La participation à un trajet implique une transaction sécurisée qui vérifie les crédits et les places restantes avant validation.

Ces diagrammes m’ont aidé à mieux anticiper les échanges entre les différentes couches et à éviter les erreurs logiques.

6. API REST et Sécurité

L’API suit une architecture REST, avec des routes bien définies pour l’authentification, la gestion des trajets, ou encore les profils utilisateurs. Toutes les requêtes critiques nécessitent un token JWT. Les statuts HTTP sont respectés (200, 201, 400, 401, 403, 404, 500) pour indiquer clairement les erreurs ou succès.

La sécurité est une priorité : les mots de passe sont hachés avec bcrypt, et les JWT sont signés avec une clé secrète stockée dans .env. Les formulaires sont validés côté client et côté serveur, et toutes les données sont nettoyées pour éviter les attaques XSS ou injections SQL. J’ai aussi ajouté des protections CORS, un middleware de rate limiting, et Helmet pour les headers HTTP.

7. Documentation et Maintenabilité

Enfin, j’ai documenté toutes les étapes dans un fichier README.md pour expliquer comment lancer le projet localement, quelles commandes utiliser, et comment déployer l’application. Ce fichier sert aussi de guide rapide pour toute personne souhaitant comprendre ou reprendre le projet. La documentation technique est organisée pour pouvoir évoluer en même temps que le projet.