ÉcoRide

Documentation technique

# Sommaire – Documentation Technique

I. Introduction

1.1 Présentation du projet  
1.2 Contexte et objectifs (EcoRide)  
1.3 Périmètre fonctionnel et contraintes

II. Analyse des besoins

2.1 Étude des utilisateurs et rôles (Visiteur, Utilisateur, Employé, Administrateur)  
2.2 Analyse des User Stories (US1 à US13)  
2.3 Spécifications fonctionnelles principales  
2.4 Contraintes techniques et de sécurité

III. Architecture de l’application

3.1 Stack technique choisie (Front-end, Back-end, Base de données relationnelle & NoSQL, Déploiement)  
3.2 Architecture logicielle (MVC, API, etc.)  
3.3 Schéma global de l’application

IV. Conception

4.1 Charte graphique (palette de couleurs, typographies, thèmes)  
4.2 Maquettes (wireframes & mockups – desktop & mobile)  
4.3 Diagrammes UML

* Diagramme de cas d’utilisation
* Diagramme de séquence
* Diagramme de classes (si applicable)

## V. Développement de l’application

5.1 Installation et configuration de l’environnement  
5.2 Développement du front-end  
5.3 Développement du back-end et logique métier  
5.4 Gestion de la base de données (relationnelle & NoSQL)  
5.5 API et sécurité

## VI. Déploiement

6.1 Procédure d’installation en local (README.md)  
6.2 Déploiement sur l’hébergement choisi   
6.3 Gestion des environnements

## VII. Gestion de projet

7.1 Méthodologie (Agile/Kanban)  
7.2 Outils utilisés (Trello, Jira, Notion, etc.)  
7.3 Suivi des versions (Git workflow : branches, merges, tests)

## VIII. Manuel d’utilisation

8.1 Présentation de l’application  
8.2 Parcours utilisateur (Visiteur → Utilisateur → Chauffeur/Passager, Employé, Admin)  
8.3 Identifiants de démonstration  
8.4 Fonctionnalités principales (recherche, réservation, gestion des trajets, etc.)

## IX. Conclusion

9.1 Récapitulatif des choix techniques  
9.2 Améliorations possibles et évolutions futures

## X. Annexes

# I. Introduction

## 1.1 Présentation du projet

ÉcoRide est une application web qui met en relation des chauffeurs et des passagers afin de faciliter le covoiturage. Elle a pour ambition de promouvoir une mobilité plus durable, en valorisant notamment les trajets effectués avec des véhicules électriques. Ce projet a été initié par la startup du même nom, engagée pour un transport plus écologique et collaboratif.

## 1.2 Contexte et objectifs

Le projet ÉcoRide s’inscrit dans une démarche de mobilité durable visant à réduire l’impact environnemental des déplacements. Il a pour finalité de limiter les émissions de gaz à effet de serre en favorisant le covoiturage, notamment ceux réalisés à l’aide de véhicules électriques.

L’application web a pour objectif de centraliser les trajets proposés par les chauffeurs et de permettre aux passagers d’y accéder facilement via une interface claire et ergonomique.

Lors de la création d’un compte, l’utilisateur peut choisir un ou plusieurs rôles : chauffeur, passager, ou les deux, selon ses besoins. Cette approche permet de concevoir une plateforme polyvalente et évolutive, facilitant la mise en relation et la gestion des trajets en temps réel.

## 1.3 Périmètre fonctionnel et contraintes

Les principales fonctionnalités prévues dans ce projet sont :

* Système d’inscription et d’authentification des utilisateurs.
* Création, modification et suppression de véhicules (pour les chauffeurs).
* Création, modification et suppression de trajets (pour les chauffeurs).
* Système de mise en relation entre les passagers et les chauffeurs (participation du passager à un covoiturage proposé par un chauffeur).
* Une application dont l’accent est mis sur l’expérience utilisateur (UX) et sur l’optimisation de l’interface utilisateur.
* Une application web responsive.

Certaines fonctionnalités avancées telles que la géolocalisation en temps réel, la messagerie intégrée ou encore la gestion de paiements ne sont pas comprises dans le périmètre de la version actuelle.

Les contraintes se traduisent sous différentes formes :

* Contraintes techniques : technologies web imposées, compatibilité multi-navigateurs, et performance de l’application.
* Contraintes de temps : respect du planning de développement défini avec le client.
* Contraintes de conception : garantir une interface claire, ergonomique et facile à prendre en main.
* Contraintes écologiques : aligner le projet sur la volonté du client de promouvoir une mobilité plus durable.

Afin de rester cohérent avec la volonté du client, ses contraintes doivent être correctement prises en compte afin d’avoir un projet en adéquation avec les contraintes et le périmètre fonctionnel défini.

# II. Analyse des besoins

## 2.1 Étude des utilisateurs et rôles

L’application ÉcoRide est conçue pour répondre à différents besoins de différents types d’utilisateur.

### 2.1.1 Visiteur

Le visiteur correspond à tous utilisateurs visitant le site sans y être inscrit.

Il peut :

* Accéder à la page d’accueil
* Rechercher un covoiturage
* Se rendre dans la page de connexion ou d’inscription.

Il ne peut :

* Accéder aux fonctionnalités réservées aux utilisateurs authentifiés.

### 2.1.2 Utilisateur

L’utilisateur correspond à une personne ayant un compte sur ÉcoRide. Il définit son propre rôle au moment de l’inscription (Passager, Chauffeur, Les deux).

Les deux rôles peuvent :

* Se connecter à l’application.
* Gérer son profil.
* Accéder à son dashboard en fonction de son rôle.

Ils ont des accès restreints selon leur rôle :

* Le chauffeur peut créer, modifier ou supprimer des véhicules et/ou des trajets.
* Le passager peut consulter et participer aux covoiturages proposés par le chauffeur, consulter ses covoiturages prévus, en cours, ou passés le cas échéant.

En tout point, ils ne peuvent :

* Accéder aux fonctionnalités propres aux rôles d’administrations.

### 2.1.3 Employé

L’employé correspond à un utilisateur ayant pour objectifs de modérer les utilisateurs et de valider les avis des utilisateurs. Il peut :

* Valider, rejeter un avis posté par un utilisateur.
* Gérer les litiges en cas d’avis négatif.

Il ne peut :

* Accéder au dashboard d’administration.

### 2.1.4 Administrateur

L’administrateur dispose de tous les droits sur l’application. Il a pour objectif de superviser les données globales et de créer comptes des rôles employés.

## 2.2 Analyse des User Stories

Avec la demande du client, 13 User Stories ont pues être défini. (Cf. Annexe User Stories) Ce tableau récapitulatif a été mis en place :

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Rôle | Objectif / action | Résultat / Bénéfice |
| US1-3 | Visiteur | Rechercher et consulter les covoiturages disponibles | Trouver un trajet |
| US4 | Visiteur | Appliquer des filtres (écologique, prix, durée, note) | Affiner la recherche de trajets |
| US5 | Visiteur / Utilisateur | Consulter les détails d’un covoiturage | Accéder aux informations du trajet |
| US6 | Visiteur / Utilisateur | Participer à un covoiturage | Réserver une place et mettre à jour ses crédits |
| US7 | Visiteur | Créer un compte | Devenir utilisateur (et obtient 20 écopièces) |
| US8 | Utilisateur | Gérer son profil et son rôle (chauffeur / passager) | Accéder aux fonctionnalités liées à son rôle |
| US9 | Chauffeur | Créer un trajet | Proposer un covoiturage |
| US10 | Utilisateur | Consulter ou annuler ses covoiturages | Gérer ses trajets et ses crédits |
| US11 | Chauffeur / Passager | Démarrer et clore un trajet | Suivre et valider un covoiturage terminé |
| US12 | Employé | Valider les avis et traiter les litiges | Garantir la fiabilité et la qualité du service |
| US13 | Administrateur | Gérer les comptes employés et consulter les statistiques | Superviser l’application et son activité |

## 2.3 Spécifications fonctionnelles principales

Les spécifications fonctionnelles décrivent les comportements fonctionnels que le système doit garantir lors des interactions avec les utilisateurs. Elles doivent permettent de répondre aux besoins spécifiques de ce dernier.

Les spécifications fonctionnelles principales sont :

* Le système d’authentification.
* La gestion du profil.
* La gestion des rôles.
* La gestion des covoiturages (pour les chauffeurs).
* La recherche et la consultation des covoiturages proposés ainsi que la possibilité de filtrer ces recherches (pour les passagers et visiteurs).
* Gestion de la participation à un covoiturage (pour les passagers).
* Gestion des différents trajets effectués par le passager et/ou proposé par un chauffeur.
* Le déroulement d’un covoiturage une fois que le chauffeur la réalisé.
* La modération et l’administration du système par les employés et l’administration.

## 2.4 Contraintes techniques et de sécurité

Les contraintes techniques sont définies par les objectifs et l’environnement dans lequel le projet doit être réalisé.

L’application doit être accessible depuis un navigateur web depuis n’importe quelle plateforme. Elle doit être adaptable selon le support utilisé (responsive) en plus d’être intuitive et agréable à utiliser.

Elle est développée avec des technologies web standards, avec l’appuis de framework le cas échéant (cf. III > 3.1). La présence de donnée sensible oblige l’utilisation d’une connexion HTTPS, hébergé sur un serveur fiable permettant un accès continu à l’application.

Les données sont stockées dans deux types de base de données :

* + Relationnelle (SQL)
  + Non relationnelle (No SQL)

L’application doit permettre l’ajout de futures fonctionnalités afin d’améliorer l’expérience utilisateur pour garantir une qualité de service optimal.

La sécurité des données est un enjeu majeur dans chaque projet. Lorsque que l’utilisateur confie des informations confidentielles le concernant, la confiance à l’égard de l’application ne doit jamais être bafouée.

Plusieurs dispositions sont mises en place pour garantir l’intégrité et la sécurité des données :

* Sécurisation de l’authentification
* Protection des données utilisateurs
* Sécurisation des échanges entre l’utilisation et le serveur
* Gestion des erreurs et accès restreints.

# III. Architecture de l’application

## 3.1 Stack technique choisie (Front-end, Back-end, Base de données relationnelle & NoSQL, Déploiement)

Le stack technique doit être en adéquation avec les besoins de l’application. Elle est composée de trois couches principales : Front-End, Back-End et Base de données. Elle sera accessible en ligne via un environnement de déploiement compatible avec les technologies choisies.

### 3.1.1 Front-End

Le Front-End correspond à la partie visible de l’application. Il permet à l’utilisateur d’interagir avec le système. Ce dernier doit rendre les différentes interactions réactives, fluides et intuitives. L’application doit être utilisable sur n’importe quelle plateforme, elle sera donc responsive.

Les technologies utilisées pour gérer le Front-End de l’application sont :

- HTML : Alimenté par le moteur de template Twig.

- CSS : Alimenté par le Framework Bootstrap 5.

- Javascript (ES6) : Qui permettra de rendre l’application agréable à utiliser et réactif.

### 3.1.2 Back-End

Le Back-End correspond à la partie invisible de l’application. Il permet de gérer et interpréter les différentes actions de l’utilisateur en les convertissant en données exploitable par le serveur.

La technologie utilisée pour le Back-End de l’application est :

- PHP : Alimenté par le Framework Symfony et toutes ses dépendances.

### 3.1.3 Base de données

La base de données correspond à la mémoire de l’application. Son rôle est de stocker les différentes informations afin de garder une cohérence des données et permettre le fonctionnement de l’application.

Les technologies utilisées pour gérer les bases de données sont :

MySQL : Pour le Système de Gestion de Base de Données Relationnelle (SGBDR).

MongoDB : Pour le Système de Gestion de Base de Données Non Relationnelle (SGBDNR).

### 3.1.4 Environnement de déploiement et hébergement

Afin de permettre un accès par n’importe quel individu, l’application doit être déployée en utilisant un environnement de déploiement. C’est un outil permettant de mettre en ligne des sites web facilement.

Le déploiement et l’hébergement de l’application ÉcoRide sera effectué sur Heroku. Via l’utilisation d’une fonctionnalité permettant de déployer le site en se connectant à un répertoire GitHub, l’application peut facilement être mise à jour.

## 3.2 Architecture logicielle

L’architecture logicielle correspond à la manière dont l’application est conçue et structurée. Celle utilisée dans le cadre de ce projet est l’architecture Modèle Vue Contrôleur (ou MVC). C’est cette architecture qui permet de répondre au mieux aux besoins de l’application. De plus, le Framework Symfony est adapté à cette dernière.

La *Vue* correspond à l’interface renvoyé à l’utilisateur (Utilisé par le moteur de template TWIG)

Le *Contrôleur* va exploiter les données que le modèle lui fournit, et générer les routes de l’application en interprétant et en renvoyant ces données dans la vue.

Le *Modèle* va gérer les interactions entre la base de données et le contrôleur.

## 3.3 Schéma global de l’application

Une image contenant texte, capture d’écran, diagramme, Police

Le contenu généré par l’IA peut être incorrect.*Schéma global de l’application*

# IV. Conception

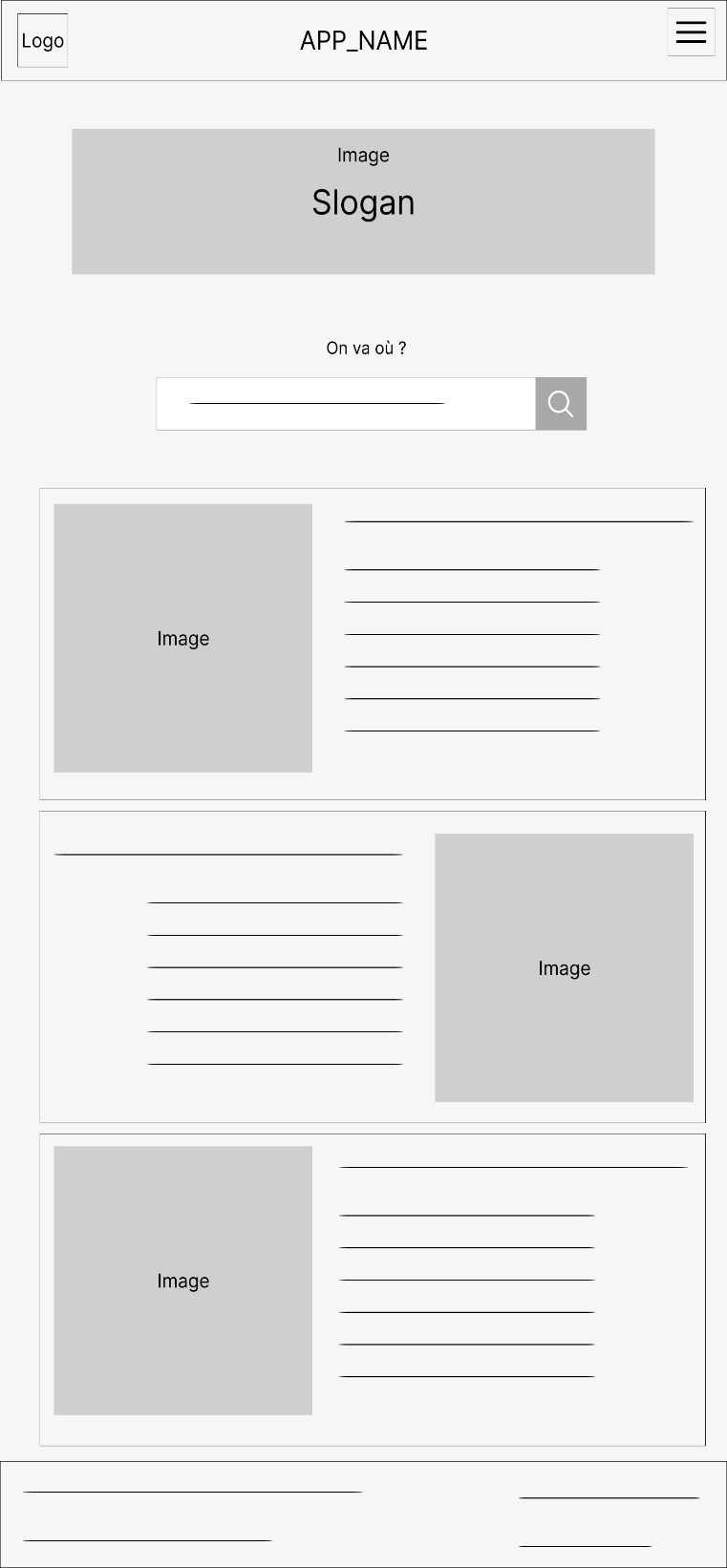
## 4.1 Charte graphique

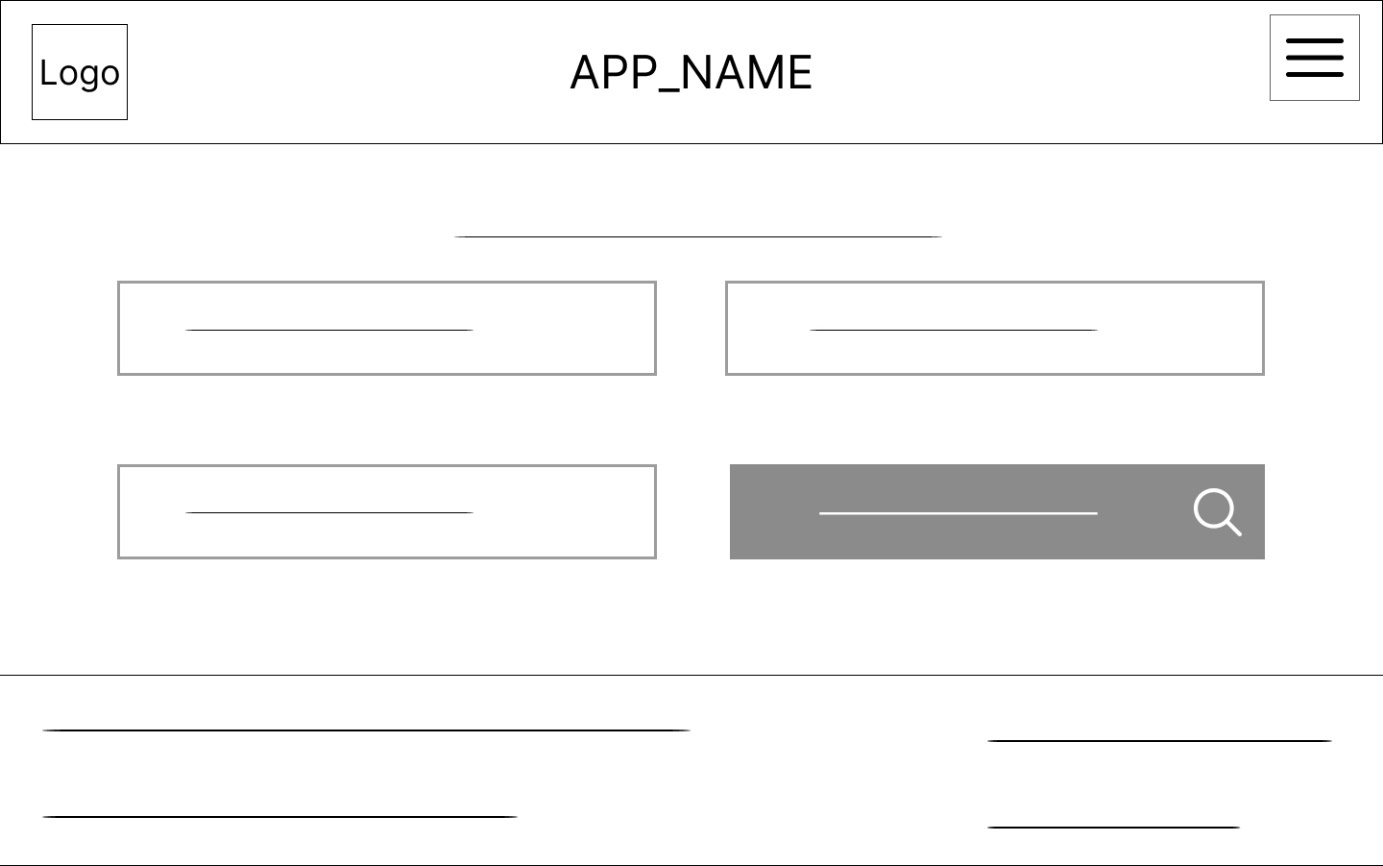
La seule condition requise pour élaborer la chartre graphique est que les couleurs de l’application doivent rappeler la nature et l’écologie.

* Couleur principale : ■ #4E5E40 – C’est la couleur omniprésente au sein de l’application, elle est utilisée comme couleur de typographie.
* Couleur secondaire : ■ #449652 – C’est la couleur qui sera remarqué aussi par l’ensemble des utilisateurs, elle est présente dans certains titres et surtout dans le titre de la barre de navigation « ÉCORIDE ».
* Couleur dégradé : ■■■#BEEBD8 #99E999 #C9E86D – Ce dégradé rappel également le côté nature, avec le vert et le vert jaune, et le côté « électrique » avec la présence du bleu turquoise, elle est présente dans les différents blocs composant cette dernière comme la barre de navigation, ou le pied de page.

## 4.2 Maquettes (wireframes & mockups – desktop & mobile)

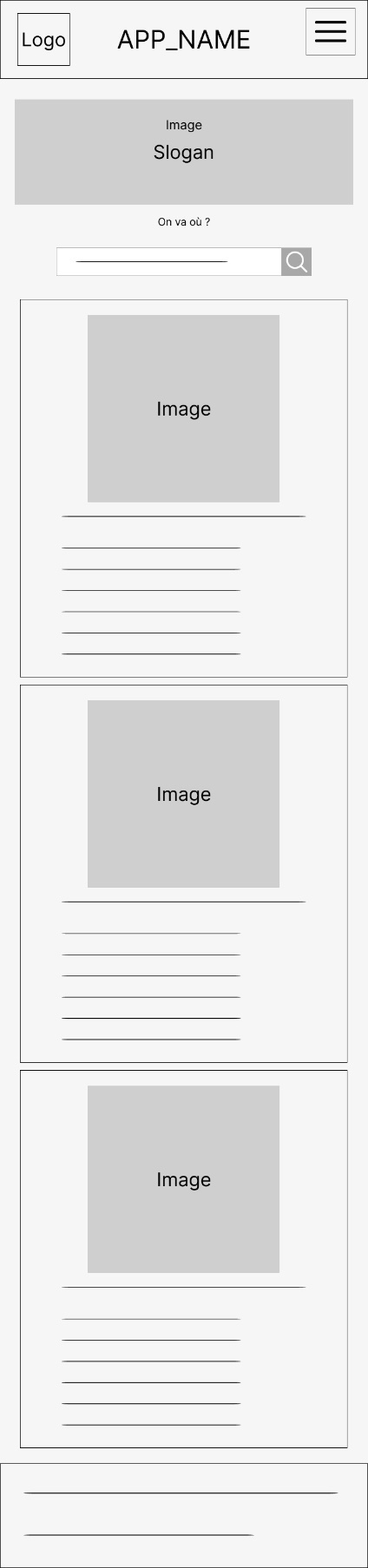
### 4.2.1 Wireframes version desktop

Wireframe de la page d’accueil, version Desktop.

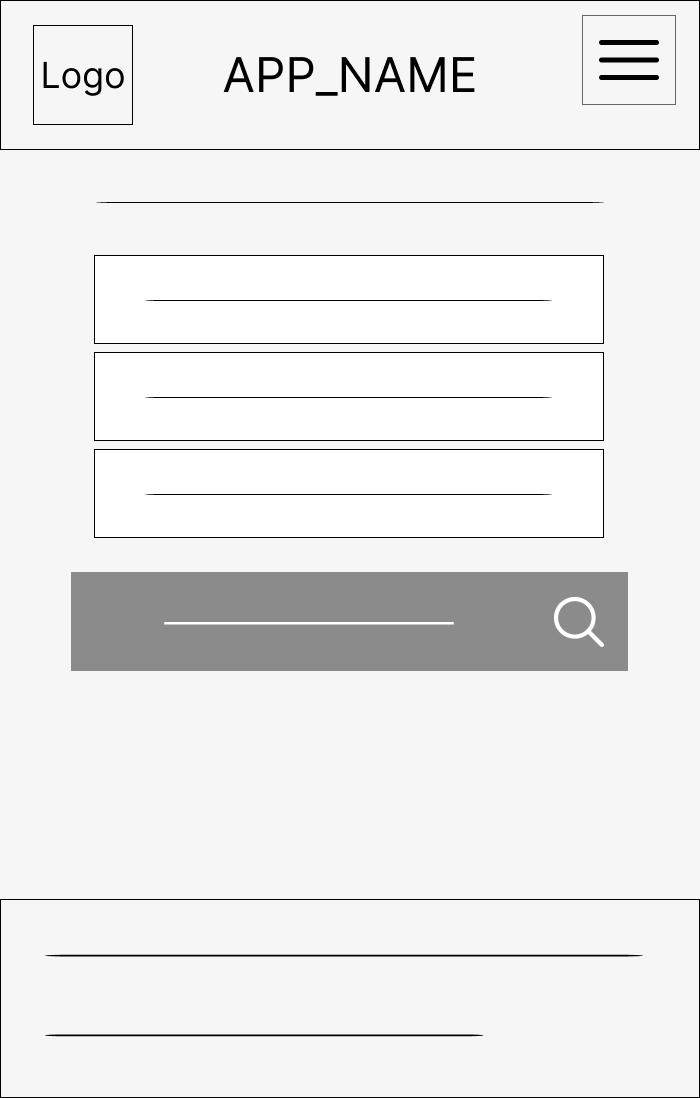
Wireframe de la page de recherche de covoiturage, version Desktop.

Wireframe des résultats de recherche de covoiturage, version Desktop.

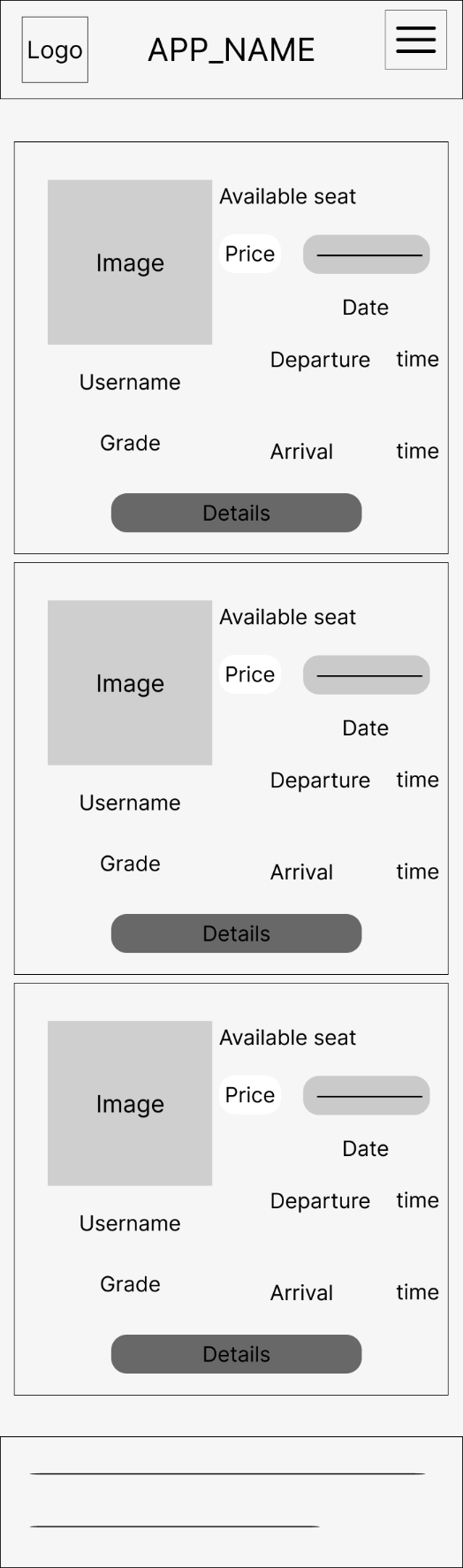
### 4.2.2 Wireframe version mobile

 Wireframe de la page d’accueil, version Mobile.

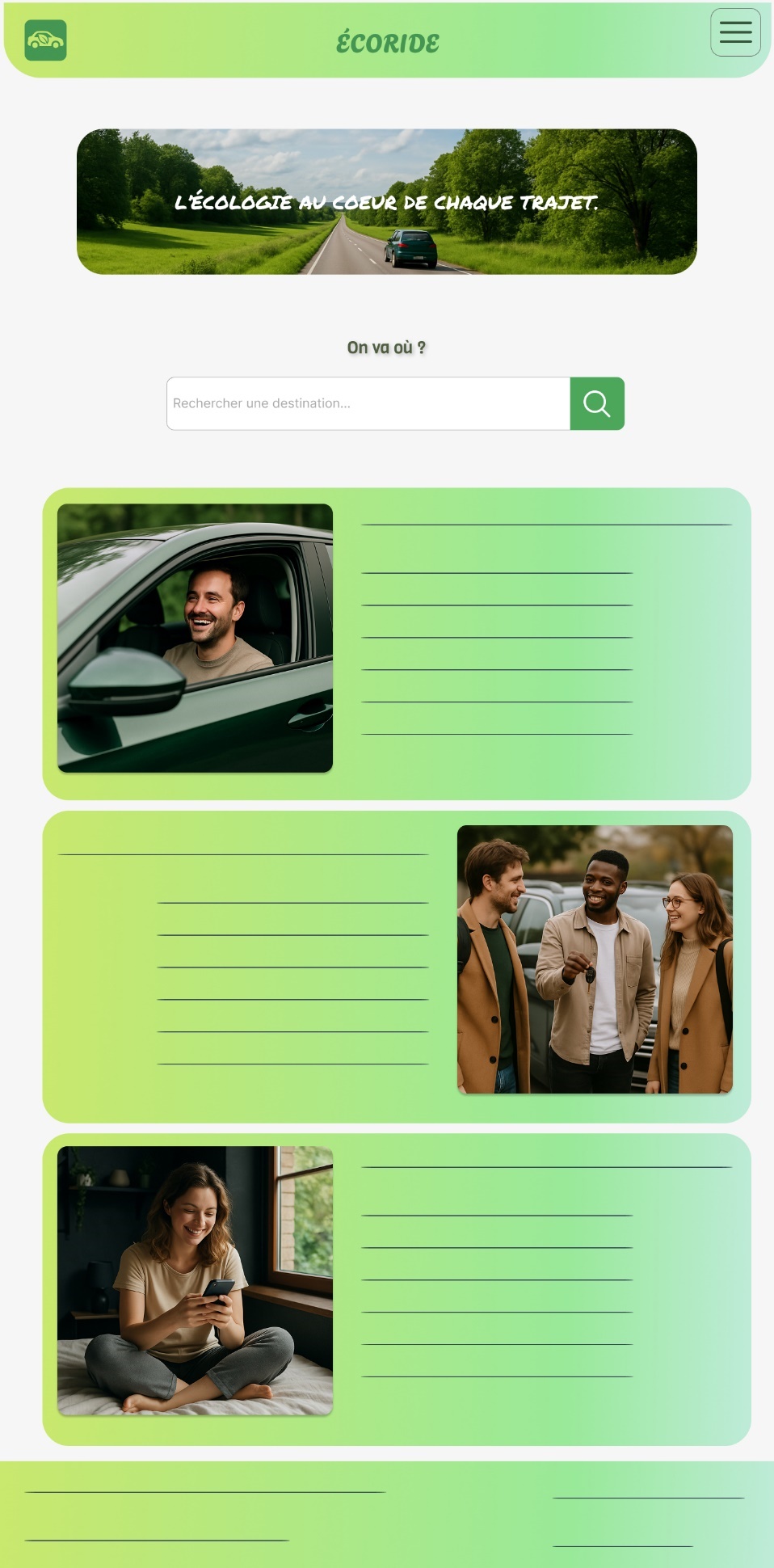
Wireframe de la page de recherche de covoiturage, version Mobile.



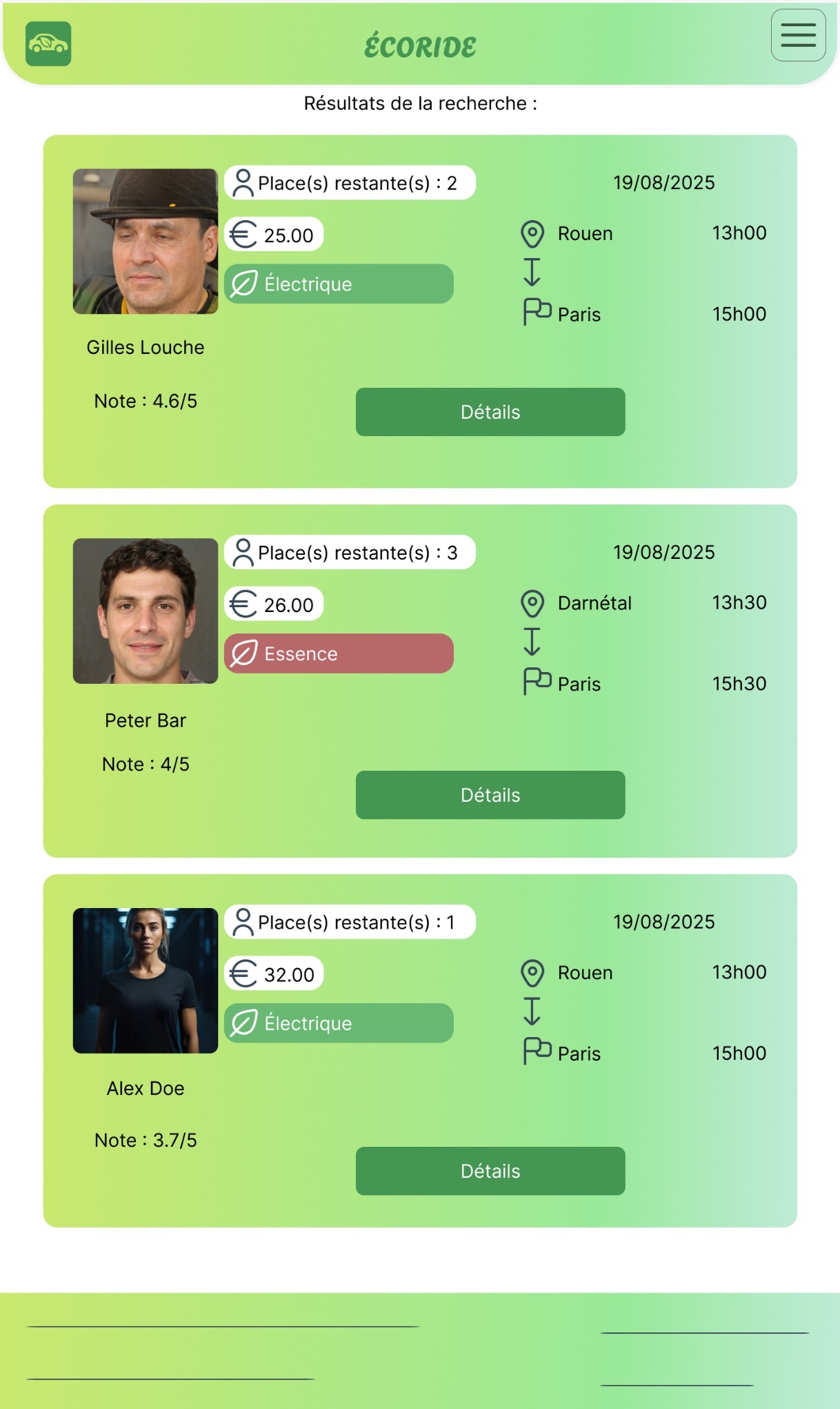
Wireframe des résultats de recherche de covoiturage, version Mobile.



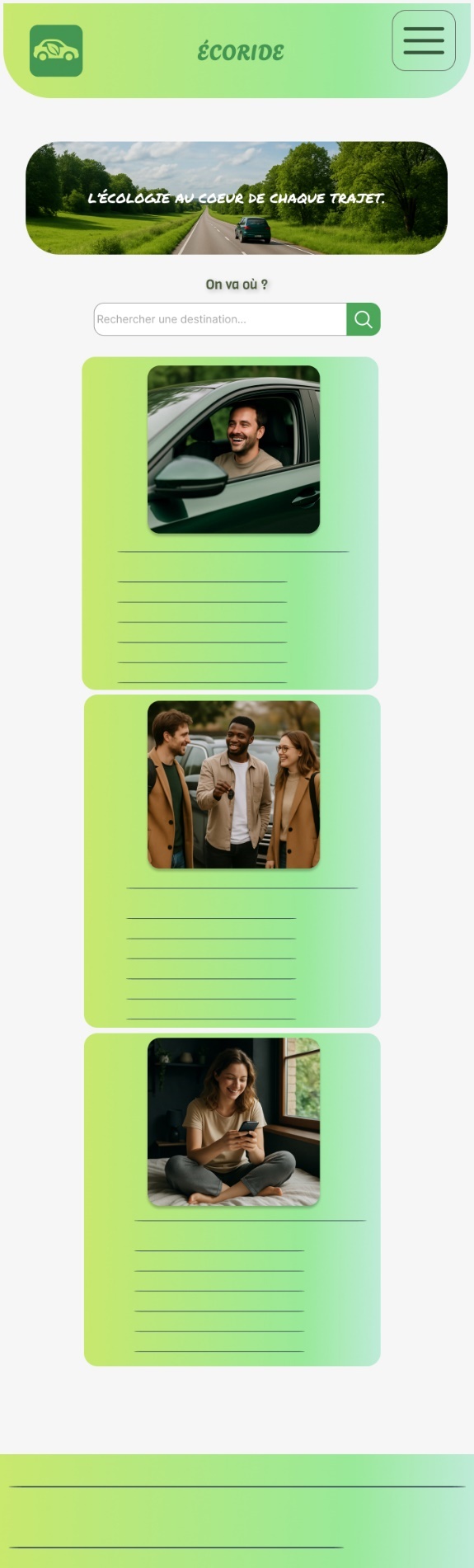
### 4.2.3 Mockup version desktop

Mockup de la page d’accueil, version Desktop.

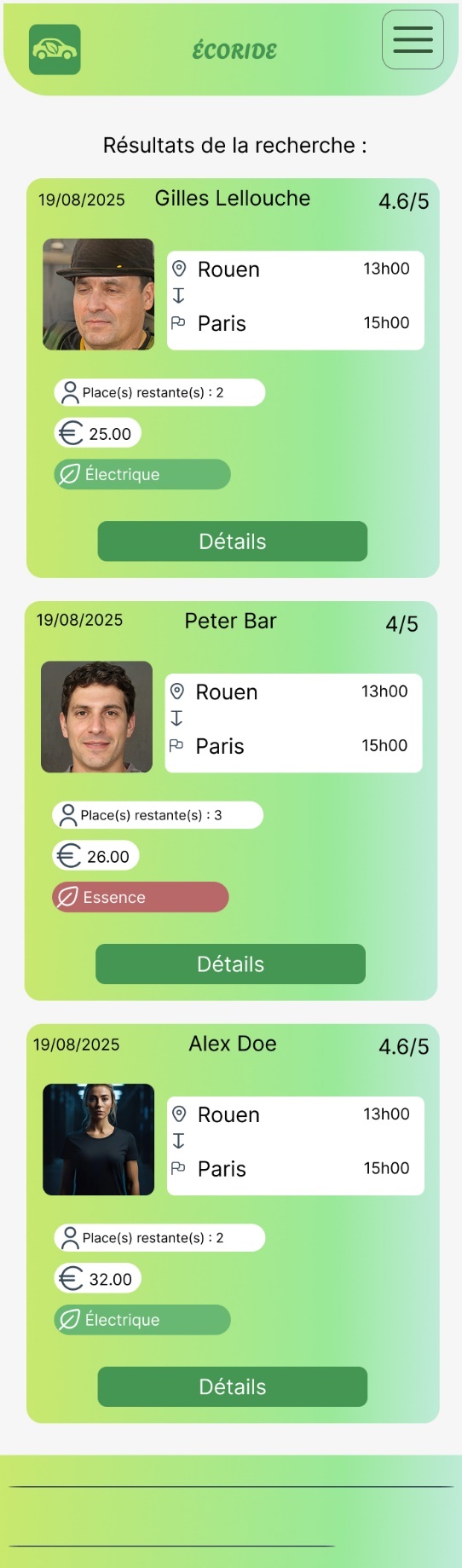
Mockup de la page de recherche de covoiturage, version Desktop.

Mockup des résultats de recherche de covoiturage, version Desktop.

### 4.2.4 Mockup version mobile

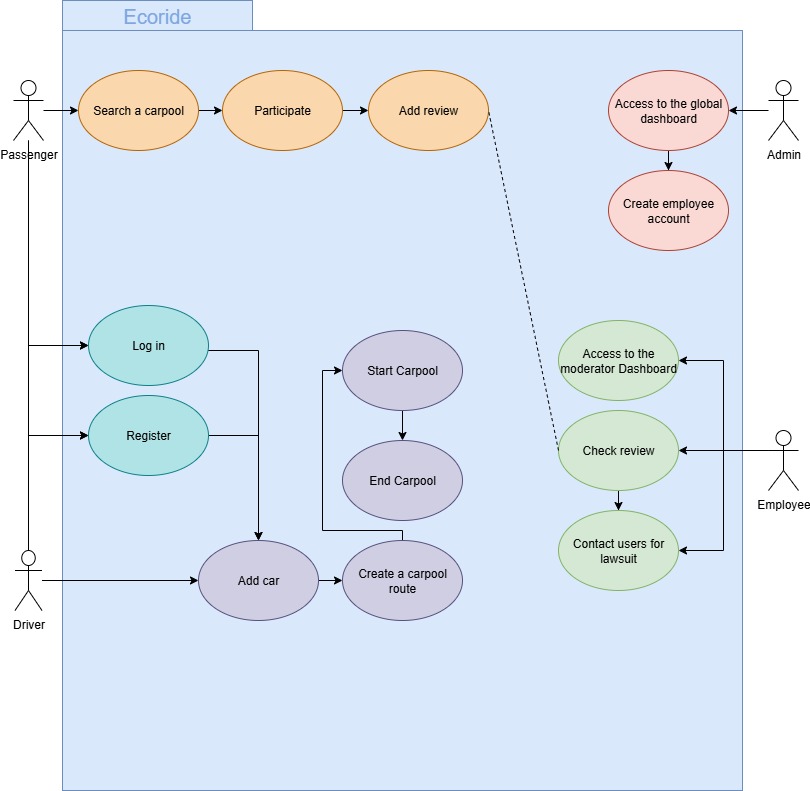
Mockup de la page d’accueil, version Mobile.

Mockup de la page de recherche de covoiturage, version Mobile.

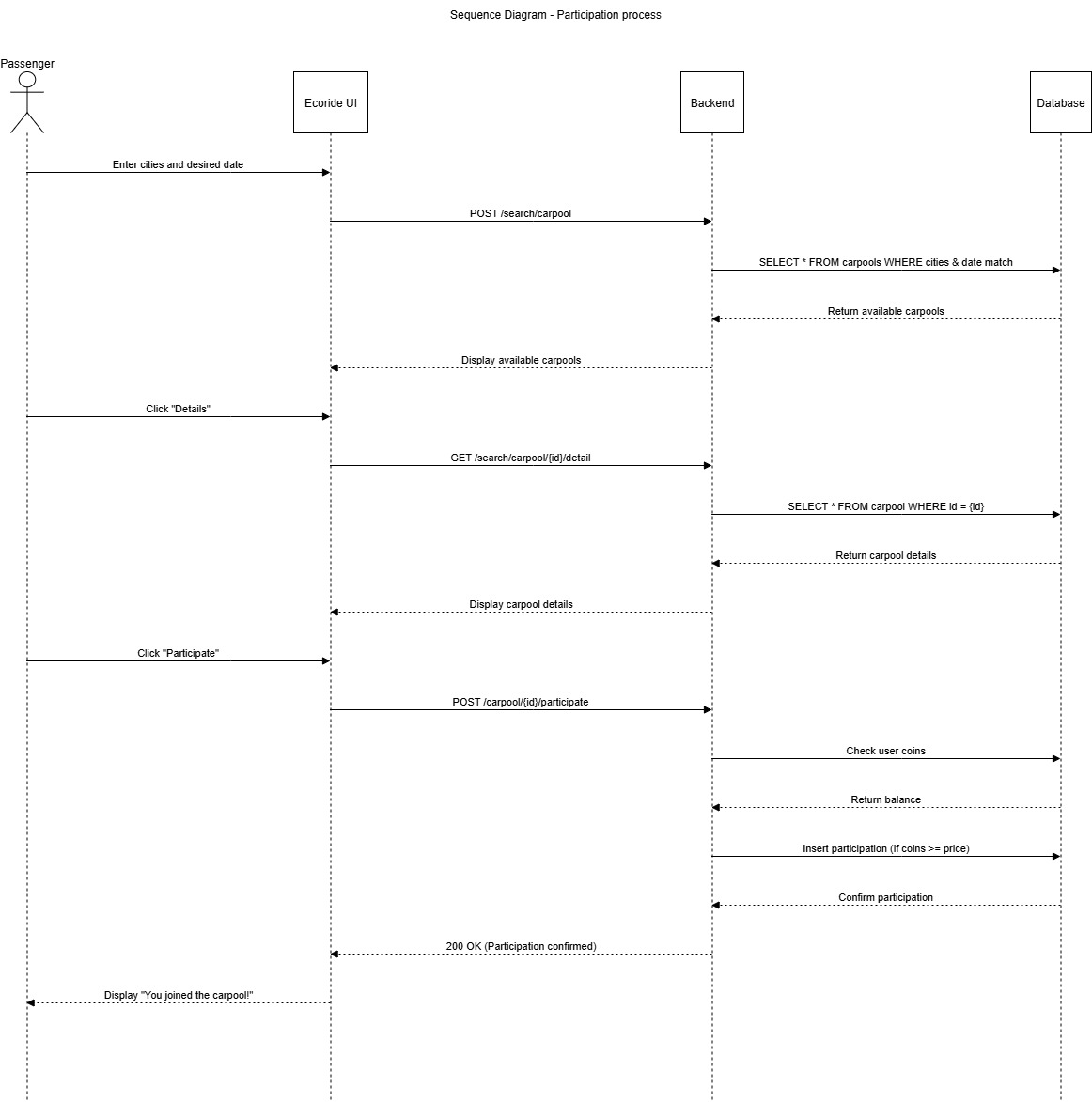
Mockup des résultats de recherche de covoiturage, version Mobile.

## 4.3 Diagrammes UML

### 4.3.1 Diagramme de cas d’utilisation



### 4.3.2 Diagrammes de séquences



### 4.3.3 Diagramme de classes

# V. Développement

## 5.1 Installation et configuration de l’environnement

La mise en place de l’environnement de développement a été effectuée à l’aide de l’IDE Visual Studio Code et de son terminal intégré.

Le framework Symfony a été utilisé pour le développement du projet, avec un serveur local lancé via PHP, les dépendances quant-à-elle ont été installées via « *composer* ».

## 5.2 Développement Front-End

Le développement de la partie Front-End du projet a été réalisé grâce moteur de template Twig, intégré par défaut à Symfony.

Twig permet de structurer les vues de manière claire et de séparer efficacement la logique métier du rendu visuel.

Chaque page est générée à partir d’un fichier .html.twig situé dans le dossier /templates, où sont définis le contenu, la mise en page et les éléments dynamiques.

Afin d’obtenir une interface cohérente et responsive, le framework Bootstrap a été utilisé.

Ce dernier a permis de simplifier la mise en forme des composants visuels (boutons, formulaires, navigation, cartes, etc.) tout en assurant une compatibilité multiplateforme.

Le code CSS personnalisé vient compléter Bootstrap pour ajuster certains détails visuels propres à l’identité graphique du projet.

Extrait de code Twig avec Bootstrap

{% set photoBase64 = participation.user.photo |to\_base64 %}

<div class="row 2 p-lg-3 p-2 bg-light text-dark my-2 rounded shadow align-items-center gap-2">

    <div class="col">

        <div class="d-flex flex-column align-items-center text-center">

            {% if photoBase64 is not null %}

                <img src="data:image/png;base64,{{ photoBase64|e }}" class="profile-img" alt="Photo de profil de {{ participation.user.username }}">

            {% else %}

                <img src="{{ asset('import/blank-profile.png') }}" class="profile-img" alt="Photo de profil par défaut">

            {% endif %}

            <span class="fs-5 fw-bold">{{ participation.user.username }}</span>

        </div>

    </div>

    <div class="col">

        <div class="d-flex flex-column align-items-center text-center">

            <span class="fs-5 fw-bold">{{ participation.user.firstname }}

                {{ participation.user.lastname }}</span>

            <span class="fs-5 mt-3">Note :</span>

            <span>{{ participation.user.grade ?? '--' }}

                / 5</span>

        </div>

    </div>

    {% if carpool.statut == 'ONLINE' and carpool.isCancelable %}

        <div class="col">

            <a href=" {{path('app\_mycarpool\_kick\_user',{'carpool\_id':carpool.id,'user\_id':participation.user.id})}} " class="btn btn-danger">Expulser</a>

        </div>

    {% endif %}

</div>

Concernant la partie JavaScript, elle est utilisée pour ajouter de l’interactivité et améliorer l’expérience utilisateur.

Des scripts simples permettent par exemple d’afficher ou de masquer certaines sections sans rechargement complet de la page.

Des requêtes AJAX seront intégrées progressivement afin de dynamiser le site, notamment pour la mise à jour de certaines données (par exemple lors de la réservation d’un trajet ou du filtrage de résultats).

L’ensemble du code Front-End est organisé de façon à rester lisible et facilement maintenable :

* Les fichiers Twig pour la structure HTML.
* Les fichiers CSS dans /public/styles pour le style global et les ajustements.
* Les scripts JavaScript dans /public/scripts pour la logique côté client.

Cette organisation garantit une séparation claire entre la présentation, le style et la logique interactive, tout en facilitant la maintenance et l’évolution du projet.

5.3 Développement du Back-end et logique métier

Le Back-end est développé en Symfony (PHP). La logique métier est organisée via :

* Entités Doctrine (User, Carpooling, Participation, Car, etc).
* Repositories pour l’accès DB.
* Services pour la logique métier et la coordination entre couches.
* Controllers pour exposer les actions côté serveur et répondre aux requêtes (pages Twig + Endpoint JSON pour AJAX).

Exemples de règles métiers implémentées :

* Une réservation ne peut pas être annulée moins de 2 heures avant l’heure de départ.
* Une participation n’est pas possible si l’heure du trajet est dans moins de 2 heures.
* Vérifications de capacité (places restantes), cohérence des horaires, et contrôle des droits selon le rôle de l’utilisateur.

Exemple d’un service métier (src/Services/CarManagerService)

<?php

namespace App\Services;

use App\Entity\Car;

use App\Entity\User;

use Doctrine\ORM\EntityManagerInterface;

final class CarManagerService

{

    public function \_\_construct(private EntityManagerInterface $em) {}

    public function FinalizeCreate(User $user, Car $car): static

    {

        $car->setUserId($user->getId());

        $user->setCurrentCar($car);

        $this->em->persist($car);

        $this->em->persist($user);

        $this->em->flush();

        return $this;

    }

    public function FinalizeSetDefaultCar(User $user, ?Car $car): static

    {

        $user->setCurrentCar($car);

        $this->em->persist($user);

        $this->em->flush();

        return $this;

    }

    public function FinalizeDelation(User $user, Car $car): static

    {

        $this->em->remove($car);

        $user->setCurrentCar(null);

        $this->em->persist($user);

        $this->em->flush();

        return $this;

    }

}

Les controllers restent fins : ils valident la requête, appellent les services, renvoient la vue ou JSON. Les exceptions métiers sont traduites en messages utilisateurs clairs (flash ou JSON error).

5.4 Gestion de la base de données (relationnelle & NoSQL)

Le projet utilise sur une architecture de données mixte combinant MySQL et MongoDB, afin de profiter à la fois de la fiabilité du modèle relationnel et de la flexibilité du NoSQL.

La base MySQL, gérée par Doctrine, contient les entités principales du système : utilisateurs, trajets, réservations et véhicules. L’ORM facilite la gestion des relations et des requêtes tout en assurant la synchronisation du schéma grâce aux migrations.

Exemple d’entité utilisée pour la gestion des trajets :

<?php

namespace App\Entity;

use App\Repository\ParticipationRepository;

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ParticipationRepository::class)]

#[ORM\UniqueConstraint(name: 'uniq\_user\_carpooling', fields: ['user', 'carpooling'])]

class Participation

{

    #[ORM\Id]

    #[ORM\GeneratedValue]

    #[ORM\Column]

    private ?int $id = null;

    #[ORM\ManyToOne(inversedBy: 'participations')]

    #[ORM\JoinColumn(nullable: false, onDelete: 'CASCADE')]

    private ?User $user = null;

    #[ORM\ManyToOne(inversedBy: 'participations')]

    #[ORM\JoinColumn(nullable: false, onDelete: 'CASCADE')]

    private ?Carpooling $carpooling = null;

    public function getId(): ?int

    {

        return $this->id;

    }

    public function getUser(): ?User

    {

        return $this->user;

    }

    public function setUser(?User $User): static

    {

        $this->user = $User;

        return $this;

    }

    public function getCarpooling(): ?Carpooling

    {

        return $this->carpooling;

    }

    public function setCarpooling(?Carpooling $Carpooling): static

    {

        $this->carpooling = $Carpooling;

        return $this;

    }

}

En parallèle, MongoDB est utilisé pour stocker les données moins structurées, comme les statistiques globales (trajets les plus populaires, volume d’activité) et les préférences utilisateurs (thème, filtres, paramètres personnalisés).

Ce modèle plus souple permet d’ajouter ou de modifier facilement des champs sans impacter la base principale.

Par exemple, lorsqu’un utilisateur valide une réservation, un document est ajouté dans MongoDB pour mettre à jour les statistiques globales, cette fonctionnalité appelle ce service :

<?php

namespace App\Services;

use App\Document\GlobalStat;

use Doctrine\ODM\MongoDB\DocumentManager;

final class GlobalStatService

{

    public const CARPOOL\_STAT = "allCarpool";

    public const ACCOUNT\_STAT = "allAccount";

    public const ECOPIECE\_STAT = "allEcopiece";

    function \_\_construct(private DocumentManager $dm) {}

    /\*\*

     \* Increase MongoDB collection GlobalStats Document

     \*/

    public function incGlobalStat(string $param, int $amount = 1): static

    {

        if (!in\_array($param, [

            self::CARPOOL\_STAT,

            self::ACCOUNT\_STAT,

            self::ECOPIECE\_STAT,

        ], true)) {

            throw new \InvalidArgumentException("Invalid stat field: $param");

        }

        $this->dm->createQueryBuilder(GlobalStat::class)

            ->updateOne()

            ->field($param)->inc($amount)

            ->getQuery()

            ->execute();

        return $this;

    }

}

Cette combinaison MySQL / MongoDB offre un équilibre entre stabilité et adaptabilité, garantissant une gestion des données à la fois robuste et évolutive.

5.5 API et sécurité

### 5.5.1 – Les APIs utilisées

Le projet n’expose pas d’API publique complète, mais repose sur plusieurs endpoints internes utilisés par les vues Twig et les requêtes AJAX pour dynamiser certaines pages.

Ces points d’accès permettent par exemple de filtrer les trajets, d’envoyer des données de réservation ou de récupérer des informations sur un utilisateur sans recharger toute la page.

Symfony gère ces échanges via des contrôleurs dédiés, qui renvoient des réponses au format JSON.

Exemple d’un Endpoint interne pour filtrer les trajets :

#[Route('/trajets/filter', name: 'trajet\_filter', methods: ['POST'])]

public function filter(Request $request, ParticipationRepository $repo): JsonResponse

{

    $filters = json\_decode($request->getContent(), true);

    $carpools = $request->findByFilters($filters);

    return $this->json($carpools);

}

Le projet intègre également des API externes, notamment GeoNames pour la suggestion de lieux et MapBox pour le calcul de temps d’itinéraire.

Ces services sont encapsulés dans des classes spécifiques (services Symfony), ce qui facilite leur maintenance et leur réutilisation tout en isolant la logique d’appel externe.

### 5.5.2 – La sécurité

Sur le plan de la sécurité, l’application s’appuie sur le composant Symfony Security.

Le système gère l’authentification des utilisateurs, la gestion des rôles (ROLE\_USER, ROLE\_DRIVER,ROLE\_PASSENGER, ROLE\_EMPLOYEE, ROLE\_ADMIN) et le hachage des mots de passe avec l’algorithme Bcrypt.

L’accès aux différentes pages est contrôlé par les règles définies dans le fichier security.yaml, et des voters assurent une gestion plus fine des autorisations selon le contexte (par exemple : seul l’auteur d’un trajet ou un administrateur peut le modifier ou l’annuler).

Cette architecture garantit un haut niveau de sécurité et de modularité tout en permettant une communication fluide entre le Front-end, les services internes et les API externes.

# VI. Déploiement

## 6.1 Procédure d’installation en local

L’installation du projet en local permet de lancer et tester l’application dans un environnement de développement complet avant son déploiement sur un serveur.

La procédure d’installation est inscrite dans le fichier readme.md présent dans le répertoire GitHub :

**Local Installation Tutorial**

Follow these steps to set up the project on your local machine:

**Prerequisites**

* [Docker](https://www.docker.com/get-started) and [Docker Compose](https://docs.docker.com/compose/install/) installed
* [Composer](https://getcomposer.org/) installed on your host PC (optional, if you want to run Composer outside Docker)
* PHP installed on your host PC (for running Symfony commands outside Docker)

**1. Clone the Repository**

git clone https://github.com/RemiFerment/Ecoride\_Project.git

cd Ecoride\_Project

**2. Start Docker Services**

This will start the database, phpMyAdmin, Composer, and Mailhog containers.

docker compose up -d

**3. Install PHP Dependencies**

Run Composer inside the Docker container:

docker compose run --rm composer install

**4. Install Importmap**

On your host PC, run:

php bin/console importmap:install

**5. Run Database Migrations**

On your host PC, execute:

php bin/console doctrine:migrations:migrate

**6. Build and Start the Application**

If you haven't already, build and start the app with Docker Compose:

docker compose up --build

**7. Access the Application**

* **Web App:** [http://localhost:8000](http://localhost:8000/)
* **phpMyAdmin:** [http://localhost:8080](http://localhost:8080/)
* **Mailhog:** [http://localhost:8025](http://localhost:8025/)

This project should now be running locally!

## 6.2 Déploiement sur l’hébergeur

Le projet EcoRide est hébergé sur la plateforme Heroku, qui permet de déployer des applications Symfony de manière simple et rapide à partir d’un dépôt GitHub.

Le déploiement est entièrement automatisé : à chaque mise à jour du dépôt distant, Heroku détecte le projet PHP et exécute automatiquement le processus de build grâce au buildpack PHP officiel.

Le déploiement s’effectue directement depuis GitHub à l’aide de la fonctionnalité “Deploy from GitHub” disponible sur le tableau de bord Heroku.

Une fois le projet relié, chaque push sur la branche principale déclenche une nouvelle mise en production.

Heroku détecte automatiquement la présence du fichier composer.json et installe les dépendances nécessaires via Composer.

Un fichier Procfile est utilisé pour indiquer à Heroku le point d’entrée de l’application :

« *web: heroku-php-apache2 public/* »

Ce fichier permet à la plateforme de lancer le serveur Apache intégré de PHP et de servir les fichiers depuis le dossier public/, comme sur un environnement Symfony standard.

Le projet utilise l’add-on JawsDB pour la base MySQL et ObjectRocket pour la base MongoDB.

Ces deux services sont reliés directement à l’application Heroku et leurs URL de connexion sont fournies automatiquement sous forme de variables d’environnement.

Après chaque déploiement, les migrations Doctrine sont exécutées pour mettre à jour le schéma de la base relationnelle :

« *php bin/console doctrine:migrations:migrate --no-interaction* »

Cette commande est lancée manuellement depuis le terminal Heroku ou intégrée dans la phase de release pour automatiser la mise à jour de la base. L’ensemble des paramètres sensibles (clés API, secrets, identifiants de connexion) est géré via les Config Vars d’Heroku.

Ces variables remplacent le fichier .env utilisé en local et sont accessibles depuis le tableau de bord de l’application.

Parmi celles configurées pour EcoRide, on retrouve notamment :

* APP\_SECRET et JWT\_SECRET : pour la sécurité et l’authentification
* DATABASE\_URL : lien vers la base MySQL (JawsDB)
* MONGODB\_URI : connexion à la base MongoDB (ObjectRocket)
* API\_TOKEN\_MAPBOX et API\_USERNAME\_GEONAME : pour les services externes de géolocalisation
* MAILER\_DSN : configuration du service d’envoi de mails
* TRUSTED\_HOSTS et TRUSTED\_PROXIES : pour la gestion des requêtes sécurisées en production

L’utilisation de ces variables garantit la sécurité et la portabilité de l’application sans exposer d’informations sensibles dans le code source.

Une fois le déploiement terminé, l’application est accessible à sur ce lien : https://ecoride-project-studi-42ed0dd5e53f.herokuapp.com/

# VII. Gestion de projet

## 7.1 Méthodologie

La méthodologie utilisée pour la bonne réalisation de ce projet est la méthode Kanban. Il s’agit de répartir les taches en fonction de leur importance, dans plusieurs tableaux :

* « Backlog » => Correspond à l’ensemble des tâches à réaliser pour finaliser le projet.
* « To do » => Correspond à l’ensemble des tâches à réaliser dans la semaine.
* « In Progress » => Correspond à l’ensemble des tâches à réaliser dans la journée.
* « Testing/Review » => Correspond à l’ensemble des tâches achevées, mais devant être testé avant la mise en production.
* « Done » => Correspond à l’ensemble des tâches achevées et mis en production.

## 7.2 Outils utilisés

Il existe une multitude d’outil pour la gestion de projet. Celui utilisé pour le suivi du projet est Trello. Il est gratuit et est propice à la méthodologie Kanban.

Le lien du Trello de projet :

<https://trello.com/b/nw7Pv6vR/ecf>

## 7.3 Suivi des versions

La mise en place du Git a été réalisé au début du projet. Il permettra de suivre l’ensemble des versions mis en place.

Le lien du répertoire Git :

https://github.com/RemiFerment/Ecoride\_Project

Les différentes branches sont :

* main
  + develop
    - home\_feature
    - role\_update
    - carpool\_feature

# VIII. Manuel d’utilisation

## 8.1 Présentation de l’application

…

## 8.2 Parcours utilisateur (Visiteur → Utilisateur → Chauffeur/Passager, Employé, Admin)

…

## 8.3 Identifiants de démonstration

…

## 8.4 Fonctionnalités principales (recherche, réservation, gestion des trajets, etc.)

…

# IX. Conclusion

## 9.1 Récapitulatif des choix techniques

…

## 9.2 Améliorations possibles et évolutions futures

…