

Documentation technique - EcoRide

Projet de covoiturage écoresponsable



III. Cahier des charges

A. Description de la demande

i. Contexte

La startup « **EcoRide** » fraîchement créée en France, a pour objectif de réduire l'impact environnemental des déplacements en encourageant le covoiturage. **EcoRide** prône une approche écologique et souhaite se faire connaître au travers d'un projet porté par José, le directeur technique, d'une application web.

L'ambition « **EcoRide** » est de devenir la principale plateforme de covoiturage pour les voyageurs soucieux de l'environnement et ceux qui recherchent une solution économique pour leurs déplacements. Il est important à souligner que la plateforme de covoiturage doit gérer uniquement les déplacements en voitures.

ii. Objectifs

La vision du client : Réduire l'impact environnemental des déplacements en valorisant le covoiturage écologique via une application web moderne, intuitive et sécurisée.

Objectifs stratégiques de l'application EcoRide

1. Promouvoir la mobilité durable

- Priorité aux trajets effectués en voiture électrique
- Filtrage écologique dans les résultats de recherche

2. Offrir une alternative économique et solidaire

- Mise en relation de conducteurs et passagers sur des trajets planifiés
- Système de crédits pour simplifier les paiements et favoriser l'engagement

3. Construire une communauté de confiance

- Notation des trajets et des conducteurs
- Validation des avis par des employés
- Gestion des réclamations après les trajets

4. Proposer une expérience utilisateur fluide






- Interface claire avec filtres intelligents (prix, durée, écologie, note)
- Espaces personnalisés (conducteur, passager, employé, administrateur)
- Début et fin de trajet encadrés, validation des participations

5. Pilotage et gestion interne

- Suivi des performances via graphiques (crédits, trajets)

- Suspension de comptes par l'administrateur si besoin

Bénéfices attendus

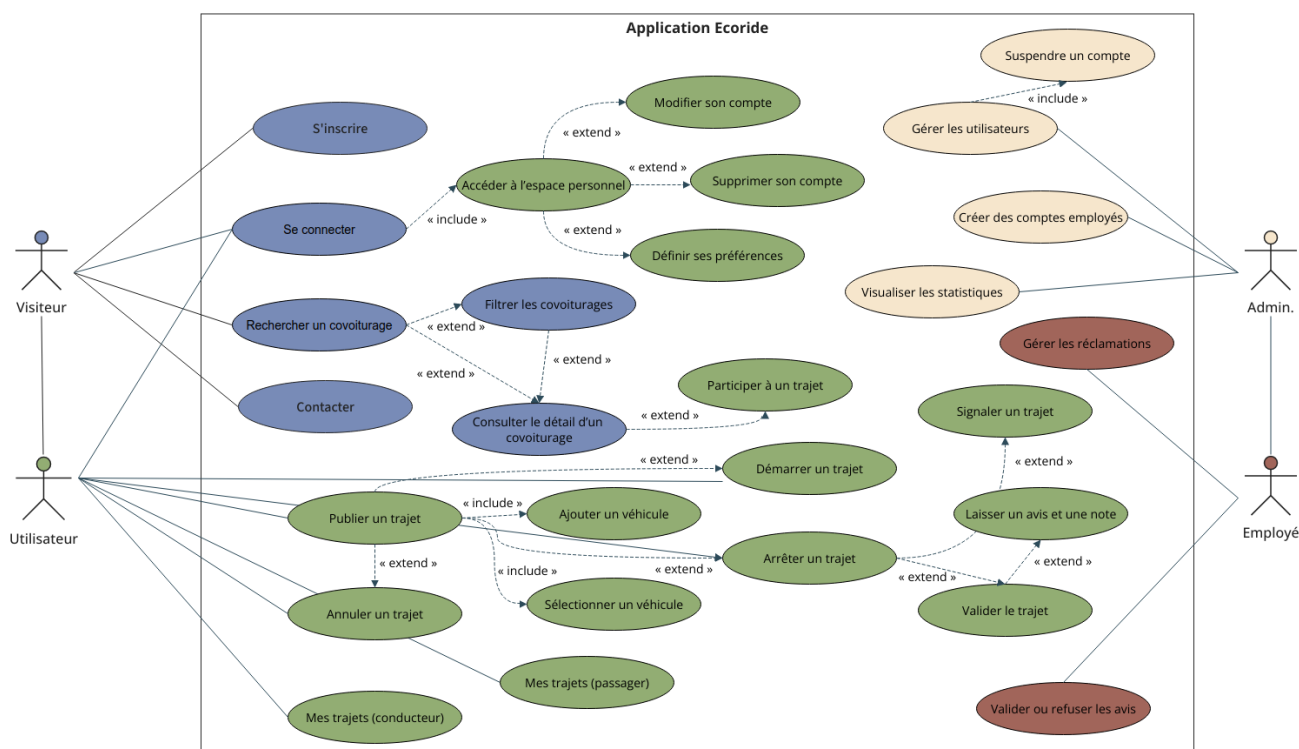
Bénéfice	Détail
 Image écoresponsable	Communication axée sur l'écologie et la durabilité
 Modèle économique maîtrisé	Système de crédits avec commission pour la plateforme
 Engagement communautaire	Avis, notes, préférences des conducteurs, réclamations gérées
 Outils de pilotage	Visualisation en temps réel de l'activité et des gains
 Différenciation claire	Positionnement face aux acteurs généralistes du covoiturage

EcoRide vise à devenir la référence du covoiturage écologique en France, grâce à :

- Une plateforme orientée vers la qualité de service,
- Une modération humaine de la communauté,
- Une approche écoconçue, moderne et responsable.

L'application ne se contente pas de concurrencer les leaders du marché : elle propose une valeur ajoutée claire pour les utilisateurs et pour l'environnement.

iii. Cas d'utilisation



Cas d'utilisation	Type (include/extend)	Explication
S'inscrire		Action principale accessible à tous les visiteurs
Se connecter	include	Nécessaire pour accéder à l'espace personnel
Accéder à l'espace personnel		Point d'entrée à toutes les fonctionnalités "utilisateur"
Modifier son compte	extend	Optionnel depuis l'espace personnel
Supprimer son compte	extend	Optionnel depuis l'espace personnel
Définir ses préférences	extend	Optionnel, après avoir accédé à l'espace personnel
Rechercher un covoiturage		Action libre d'un visiteur ou utilisateur
Filtrer les covoiturages	extend	Optionnel après une recherche
Consulter le détail d'un covoiturage	extend	Optionnel après une recherche
Contacteur		Accès libre (formulaire)
Participer à un trajet	extend	Accessible après consultation du détail d'un trajet
Publier un trajet		Action principale du conducteur
Ajouter un véhicule	include	Est obligatoire lors de la création du trajet
Sélectionner un véhicule	include	Requis lors de la publication d'un trajet
Annuler un trajet	extend	Possible après avoir publié ou rejoint un trajet
Démarrer un trajet	extend	Action du conducteur après publication
Arrêter un trajet	extend	Action après démarrage du trajet
Mes trajets (conducteur)		Historique accessible depuis l'espace conducteur
Mes trajets (passager)		Historique accessible depuis l'espace passager
Valider le trajet	extend	Suite logique de l'arrêt de trajet
Laisser un avis et une note	extend	Optionnel après validation du trajet
Signaler un trajet	extend	Optionnel après validation si problème détecté
Gérer les utilisateurs		Action principale de l'administrateur
Suspendre un compte	include	Action incluse dans la gestion des utilisateurs
Créer des comptes employés		Action directe de l'administrateur
Visualiser les statistiques		Accessible directement par l'administrateur
Gérer les réclamations		Action manuelle de l'employé
Valider ou refuser les avis		Action de modération réservée à l'employé

Quand utiliser « **include** » (trait pointillé avec flèche) :

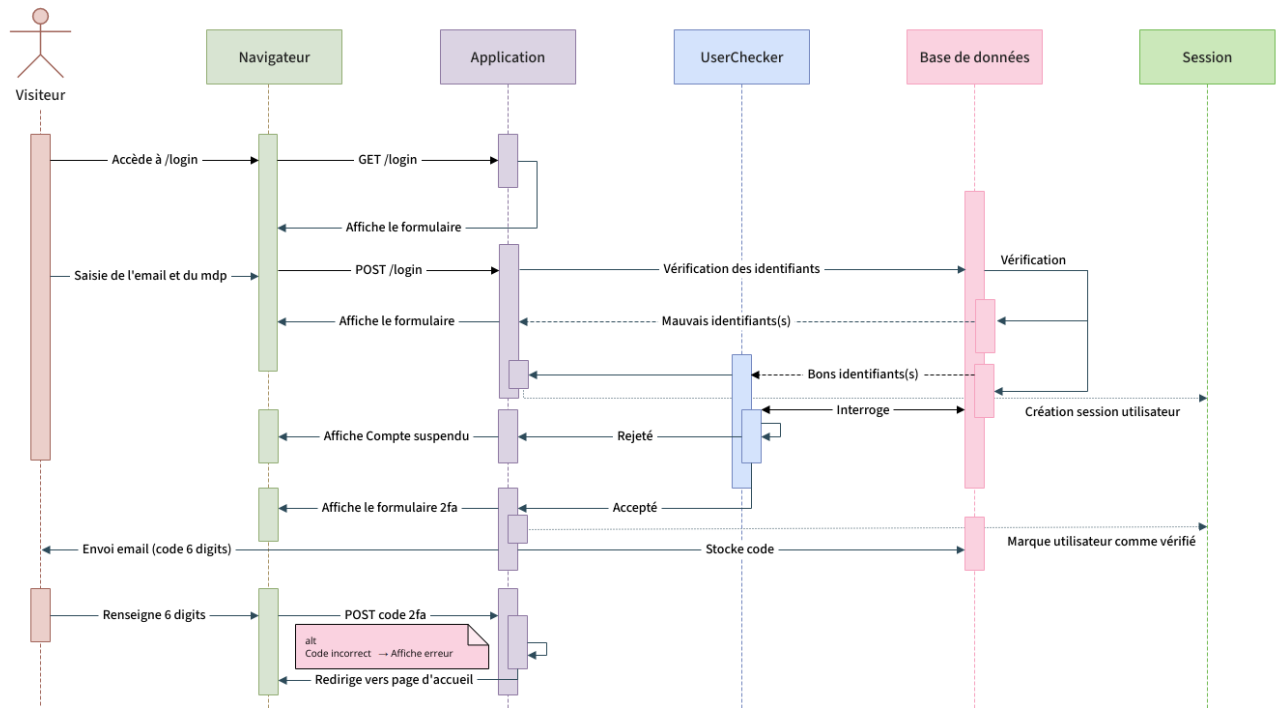
- Quand **plusieurs cas d'utilisation réutilisent une même fonctionnalité commune**. Cela permet de **factoriser un comportement** qui est toujours exécuté dans un autre cas d'utilisation.

Quand utiliser « **extend** » (trait pointillé avec flèche) :

- Quand un comportement **optionnel** ou **conditionnel s'ajoute** à un autre cas.

iv. Scénario

Acteur(s) : visiteur de l'application



Le visiteur

- Accède à la page /login
- Saisit son adresse e-mail et son mot de passe.

Le Navigateur

- Envoie une requête GET pour récupérer le formulaire de connexion.
- Affiche le formulaire à l'utilisateur.
- Transmet les données saisies via une requête POST à l'application.

L'Application

- Reçoit la requête POST.
- Vérifie les identifiants auprès de la base de données.
- Si les identifiants sont incorrects : affiche une erreur.
- Si les identifiants sont corrects :
 - Envoie les informations à **UserChecker** pour vérifier l'état du compte (actif ou suspendu).
 - Si rejeté : affiche « Compte suspendu ».
 - Si accepté :
 - Affiche le formulaire de vérification 2FA.

- Envoie un e-mail contenant un code à usage unique à 6 chiffres.
- Enregistre une session utilisateur dans le système.
- Attend la saisie du code.

Le **UserChecker**

- Reçoit les informations utilisateur.
- Vérifie si le compte est actif.
- Retourne un état (Rejeté ou Accepté).
- Envoie la requête à la base pour création de la session.

La **Base de données**

- Vérifie les identifiants de connexion.
- Répond à Application avec un statut (ok ou erreur).
- Interagit également avec UserChecker et stocke le code 2FA validé.

La **Session**

- Enregistre la session de l'utilisateur après vérification.
- Marque l'utilisateur comme vérifié si le code est correct.

La **Vérification 2FA**

- L'utilisateur saisit le code.
- Si le code est **incorrect** :
 - Un message d'erreur est affiché.
- Si le code est **correct** :
 - L'utilisateur est marqué comme vérifié dans la session.
 - Il est redirigé vers la page d'accueil.

IV. Spécifications techniques

A. Outils de développement

i. IDE

L'IDE utilisé pour ce projet tant côté Front que côté Back, est Visual Studio Code (VsCode), édité par Microsoft. La raison pour laquelle nous nous sommes dirigés vers cet IDE est son confort de développement. Il permet de développer sous différents langages de programmation comme JavaScript, PHP, CSS ou HTML.

ii. JMerise

Une réflexion en amont sous la forme de schémas conceptuels Merise a permis la création des différentes tables de la base de données.

iii. Wampserver

Wampserver 3.3.7, acronyme de Windows, Apache, MySQL et PHP/Perl/Python, est une pile logicielle pour Windows qui permet le développement et le déploiement d'applications Web. Il sert de serveur local sur la machine et permet d'écrire, de déboguer et de tester des applications Web dans un environnement contrôlé qui imite un environnement réel. Ceci est particulièrement utile pour garantir que les applications se comportent comme prévu avant d'être déployées sur un site public.

MySQL est le système de gestion de base de données utilisé dans la pile WAMP. Il stocke, récupère et gère les données dans un format structuré à l'aide de bases de données relationnelles.

PHP 8.3.14 est exécuté via WAMP.

iv. Langages et framework utilisés

- Front : HTML5, CSS3, Javascript - ECMAScript 2022
- Back : PHP 8.3.14, type de serveur de base de données : MySQL 8.x
- Framework : Symfony 7.2

B. Point sécurité

i. Architecture MVC

L'architecture Model-View-Controller (MVC) est un modèle de conception largement utilisé dans le développement de logiciels pour diviser une application en trois parties interconnectées : le modèle, la vue et le contrôleur. Symfony est ainsi construit autour de ce modèle.

Le modèle dans Symfony représente les données de l'application. Il contient la logique métier, les règles métier, les méthodes pour récupérer et stocker l'état persistant des données. Dans Symfony, le modèle est principalement géré par l'ORM Doctrine.

La vue prend les données du modèle et les présente à l'utilisateur sous une forme facilement compréhensible. Dans Symfony, la vue est principalement gérée par le moteur de templates Twig. Twig est un moteur de templates, flexible, rapide et sécurisé. Il permet de séparer la logique de l'application de sa présentation, facilitant ainsi la maintenance et l'amélioration du code. Il intègre des fonctionnalités avancées telles que : l'**héritage** de templates : nous pouvons définir un bloc et le remplacer dans les templates enfants, différents **filtres** et fonctions pour transformer ou afficher les données, un **contrôle de flux** pour exécuter différentes actions en fonction de conditions spécifiées. Twig applique **par défaut un échappement automatique du HTML**, ce qui permet de limiter considérablement les attaques XSS.

Le contrôleur sert d'intermédiaire entre le modèle et la vue. Il traite les requêtes de l'utilisateur, interagit avec le modèle pour récupérer ou stocker des données et met à jour la vue en conséquence.

Dans Symfony, la chaîne de traitement de la requête débute lorsque le HTTPKernel de Symfony reçoit la requête HTTP du client. À partir de là, un certain nombre de processus sont déclenchés, notamment l'interception de la requête, la transformation de la requête en un objet de Symfony Request, puis le resolving vers le contrôleur approprié qui se chargera du traitement ultérieur. Cette résolution se fait via le « Controller Resolver » de Symfony, qui déterminera quel contrôleur et quelle méthode doivent être utilisés pour traiter la requête.

Une fois que le contrôleur a terminé le traitement de la requête, Symfony génère une instance de l'objet Response. Cette instance contient la réponse qui sera finalement renvoyée au client.

ii. Pourquoi l'utilisation de Symfony ?

Symfony offre de nombreux avantages :

1. **Réutilisabilité du code** : Grâce à son système de bundles, le code peut être réutilisé facilement dans différents projets. Cela permet d'économiser du temps et des efforts.
2. **Documentation complète** : Symfony a une très grande communauté et la documentation

est de grande qualité et continuellement mise à jour, permettant l'apprentissage et la résolution de problèmes facilement.

3. **Testabilité** : Symfony est conçu pour faciliter les tests unitaires et fonctionnels, garantissant ainsi la qualité du code.

4. **Performance** : Symfony est connu pour sa vitesse et sa performance, en particulier avec Symfony Flex et Turbo qui optimise le rendu.

5. **Sécurité** : Les composants de sécurité de Symfony sont conçus pour protéger les applications contre les attaques courantes.

6. **Écosystème riche** : Symfony s'intègre facilement avec des outils comme Doctrine ORM, API Platform, Mercure, Webpack/AssetMapper, facilitant le développement moderne.

iii. Pourquoi créer ses propres services Symfony ? Une réponse orientée SOLID

Dans un projet Symfony, la création de services personnalisés (dans le dossier **src/Service/**) est une pratique essentielle pour respecter les **principes SOLID** et maintenir une architecture propre et évolutive.

- **S : Single Responsibility Principle (SRP)**

Chaque service est dédié à une tâche unique : envoi d'e-mails, gestion des créneaux horaires, calcul d'un tarif, etc. Cela évite d'alourdir les contrôleurs ou les entités avec de la logique métier.

- **O : Open/Closed Principle**

Nos services sont **ouverts à l'extension mais fermés à la modification**. On peut injecter de nouvelles dépendances, modifier le comportement via une interface ou un décorateur, sans toucher au cœur de l'application.

- **L : Liskov Substitution Principle**

Lorsque des services implémentent une interface (ex : MailerInterface), ils peuvent être substitués facilement par un mock ou une autre implémentation. Cela améliore les tests et la maintenabilité.

- **I : Interface Segregation Principle**

On préfère des interfaces spécifiques à une responsabilité plutôt qu'une interface géante. Cela garantit que les classes clientes ne dépendent que de ce qu'elles utilisent.

- **D : Dependency Inversion Principle**

Symfony encourage l'**injection de dépendances** via le constructeur. Cela favorise le découplage et la testabilité.

En résumé, créer ses propres services permet de centraliser la logique métier, réutiliser du code, tester facilement et suivre les bonnes pratiques de conception. Dans le projet EcoRide, c'est ce qui a permis de maintenir des contrôleurs fins, des entités sobres et une architecture

modulable.

iv. La sécurité dans Symfony 7.2

Un pilier central du framework ! Symfony 7.2 intègre un système de sécurité robuste, modulaire et extensible, pensé dès l'origine pour protéger les applications web contre les principales attaques connues, tout en restant flexible et personnalisable. Voici un état des lieux complet :

1. XSS – Cross Site Scripting

Objectif de l'attaque : Injecter du JavaScript malveillant dans les pages vues par les utilisateurs (ex : `<script>alert('XSS')</script>` dans un champ commentaire).

Protection dans Symfony :

- **Twig applique un échappement automatique du HTML** : toute variable affichée via `{{ variable }}` est échappée.
- Symfony limite donc les injections de scripts.
- Pour afficher du HTML légitime, il faut utiliser le filtre `|raw` avec précaution.

2. CSRF – Cross Site Request Forgery

Objectif de l'attaque : Forcer un utilisateur connecté à effectuer une action à son insu (ex : suppression de compte).

Protection dans Symfony :

- Le **token CSRF est activé automatiquement** pour les formulaires (`csrf_protection: true`).

```
1 # config/packages/framework.yaml
2 framework:
3     # ...
4     csrf_protection: ~
5     # ...
6     form:
7         csrf_protection:
8             enabled: true
```

- Dans les vues, il est intégré via `{{ form_start(form) }}`.
- En back, Symfony vérifie automatiquement la validité du token.
- Pour les API ou requêtes AJAX, on peut utiliser `csrf_token('token_id')` avec `CsrfTokenManagerInterface`.

Il faut toujours inclure un token dans les formulaires sensibles.

3. SQL Injection

Objectif de l'attaque : Injecter du SQL via un champ utilisateur pour exécuter des requêtes non prévues.

Protection dans Symfony :

- Doctrine ORM **utilise des requêtes préparées** nativement.

- Aucun bindParam ou concaténation manuelle de chaînes SQL n'est nécessaire.
- Les QueryBuilder, DQL, ou Repository::findOneBy() sont tous sécurisés contre l'injection.
- Ne jamais concaténer manuellement des champs utilisateur dans une requête DQL brute.
- Valider les entrées avec le composant Validator.

4. Authentication & Brute Force

Risques : Attaques par force brute sur la page de connexion. Session volée ou détournée.

Protection dans Symfony :

- Système d'authentification basé sur **Security Voter et Firewall**.
- Prise en charge des sessions sécurisées (via cookies HTTP-only, avec expiration).
- **RateLimiter** (limiteur de débit) activable pour la connexion :

```

1 # config/packages/security.yaml
2 security:
3     # ...
4     Firewalls:
5         # ...
6         main:
7             # ...
8             # configure the maximum login attempts
9             login_throttling:
10                 max_attempts: 3             # per minute ...

```

Options avancées :

- Authentification 2FA avec scheb/2fa-bundle (TOTP ou mail).
- Authentification multi-dispositif (sessions multiples, tokens par appareil).
- « Remember Me » chiffré avec cookies sécurisés.
- Restreindre l'accès aux routes par rôle.
- Imposer une complexité minimale sur les mots de passe.
- Activer les logs de sécurité (security.yaml > access_control + Monolog).

5. Clickjacking

Objectif : Afficher votre site dans un <iframe> malveillant pour voler des clics.

Protection dans Symfony :

- Installer le package nelmio/security-bundle dans votre composer.json et mettre à jour les dépendances :

```
$ composer require nelmio/security-bundle
```

```

1 # config/packages/nelmio_security.yaml
2 nelmio_security:
3     # ...
4     clickjacking:
5         paths:
6             '^/.*': DENY
7         hosts:
8             - '^foo\.com$'
9             - '\.example\.org$'

```

6. Gestion des droits et des accès (ACL)

Composant utilisé :

- `is_granted()` ou attribut **#[IsGranted]** dans les contrôleurs, templates.
- Voters qui permettent une granularité fine d'accès par entité ou ressource.
- Utiliser des rôles (ROLE_USER, ROLE_ADMIN, etc.) combinés aux voters pour les objets sensibles.
- Séparer la logique métier de la logique de permission (voter = bon endroit).

7. Validation des données (serveur)

Symfony sécurise les entrées utilisateur grâce au composant Validator :

- Contraintes comme `#[Assert\NotBlank]`, `#[Assert\Email]`...
- Protection contre des données invalides ou malformées.
- Toujours valider les classes de modèles de formulaires côté serveur.
- Ne pas se fier uniquement à la validation JavaScript côté client.

8. Hashage sécurisé des mots de passe

Utilisation de :

- **UserPasswordHasherInterface**
- Algorithmes modernes : `bcrypt`, `argon2id` (automatique avec `auto` dans `security.yaml`)
- Ne jamais stocker un mot de passe en clair.
- Appliquer un sel fort (géré automatiquement par Symfony).

Symfony propose une architecture orientée sécurité par défaut :

- Sécurisation des entrées/sorties,
- Contrôle d'accès granulaire,
- Intégration native des standards modernes (limiteurs, authentification, hashage).

L'utilisation de ses composants (Security, Validator, Form, HttpKernel, CSRF, Twig...) permet de construire une application robuste et résiliente, même en cas de mauvaise manipulation utilisateur ou tentative d'attaque ciblée.

C. Adaptation pour le web mobile

Le cahier des charges d'ECORIDE stipule de développer l'interface au format web mobile. L'application a donc été conçue en *mobile-first*, ce dernier répondant parfaitement à tous les supports d'écrans.

Ainsi, le framework Bootstrap a été utilisé et plusieurs éléments CSS ont été ajoutés.

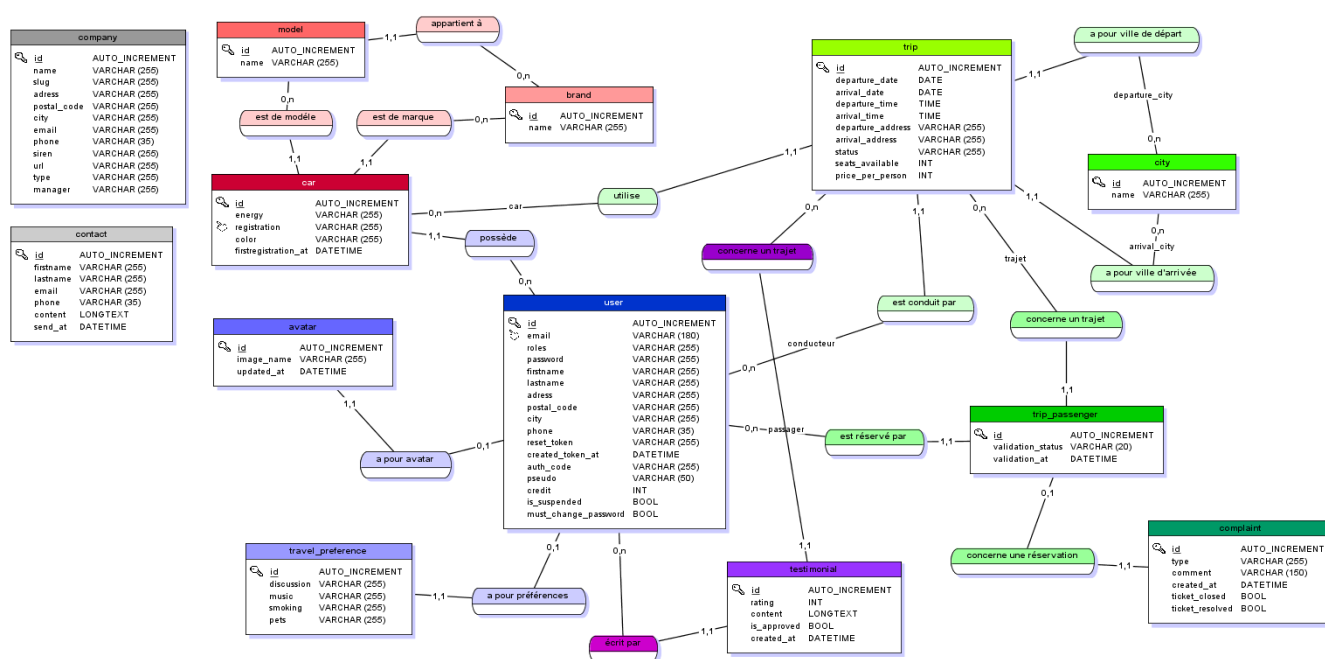
V. Réalisation

A. Conception

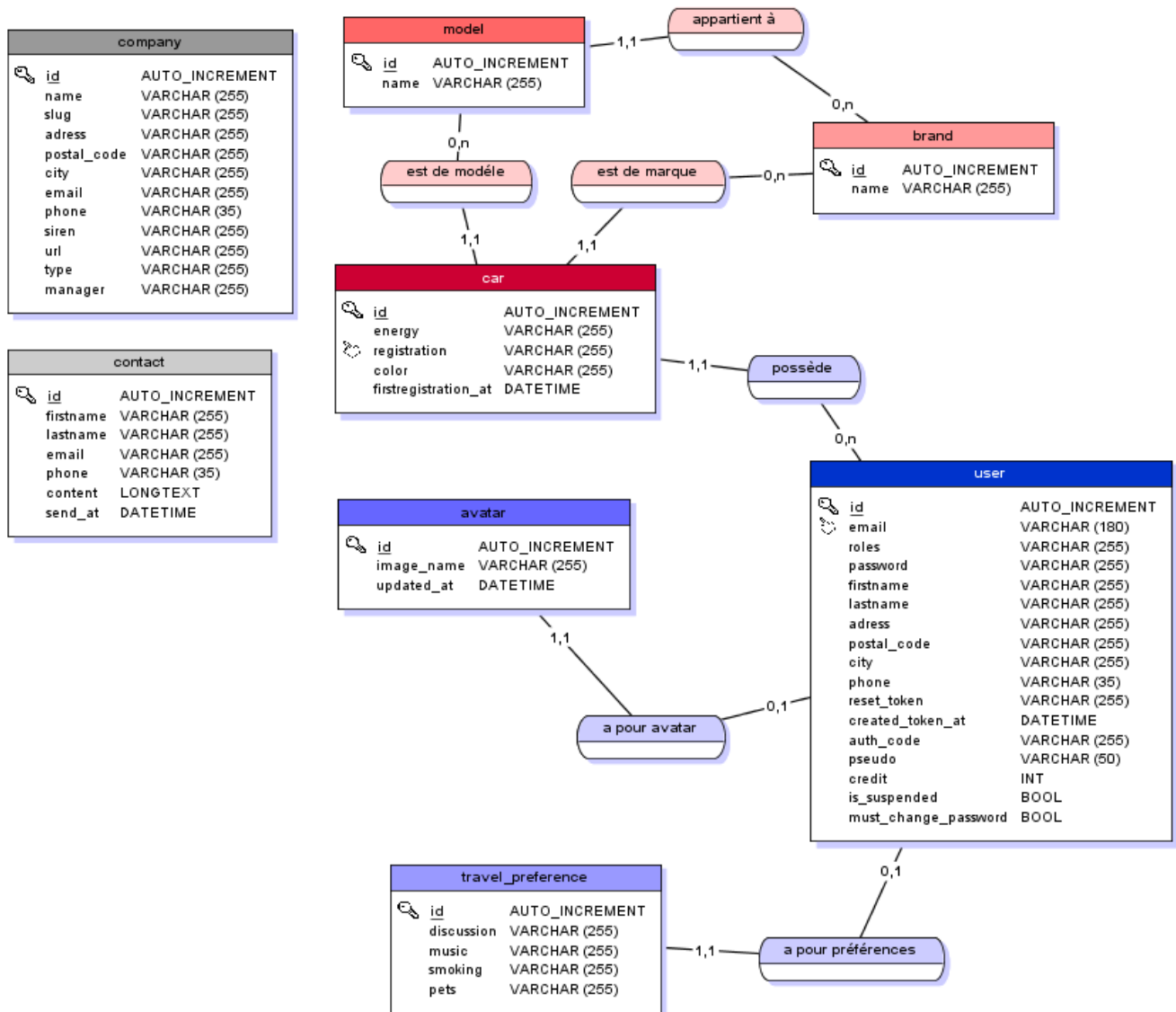
i. Structure de données

Une réflexion en amont sous la forme de schémas conceptuels Merise : MCD (Modèle Conceptuel des données), MLD (Modèle Logique des données), a permis la création, sur phpMyAdmin, des différentes tables permettant de gérer les données recueillies.

MCD :



MCD ECORIDE - général



MCD ECORIDE – partie 1

Explications des relations :

1. avatar ↔ user (a pour avatar)

La table avatar est associée à la table user par une cardinalité **(1,1)** car chaque avatar est obligatoirement lié à un utilisateur unique. Inversement, la table user est associée à avatar par une cardinalité **(0,1)** car un utilisateur peut ne pas avoir d'avatar ou n'en avoir qu'un seul.

2. travel_preference ↔ user (a pour préférences)

La table travel_preference est liée à user par une cardinalité **(1,1)** car chaque préférence appartient à un utilisateur. Côté user, la cardinalité est **(0,1)** car un utilisateur peut avoir 0 ou 1 préférence de voyage.

3. car ↔ user (possède)

Chaque voiture (car) est associée à la table user par une cardinalité **(1,1)**, car elle appartient toujours à un utilisateur. Côté user, la cardinalité est **(0,n)**, car un utilisateur peut posséder plusieurs voitures.

4. car ↔ model (est de modèle)

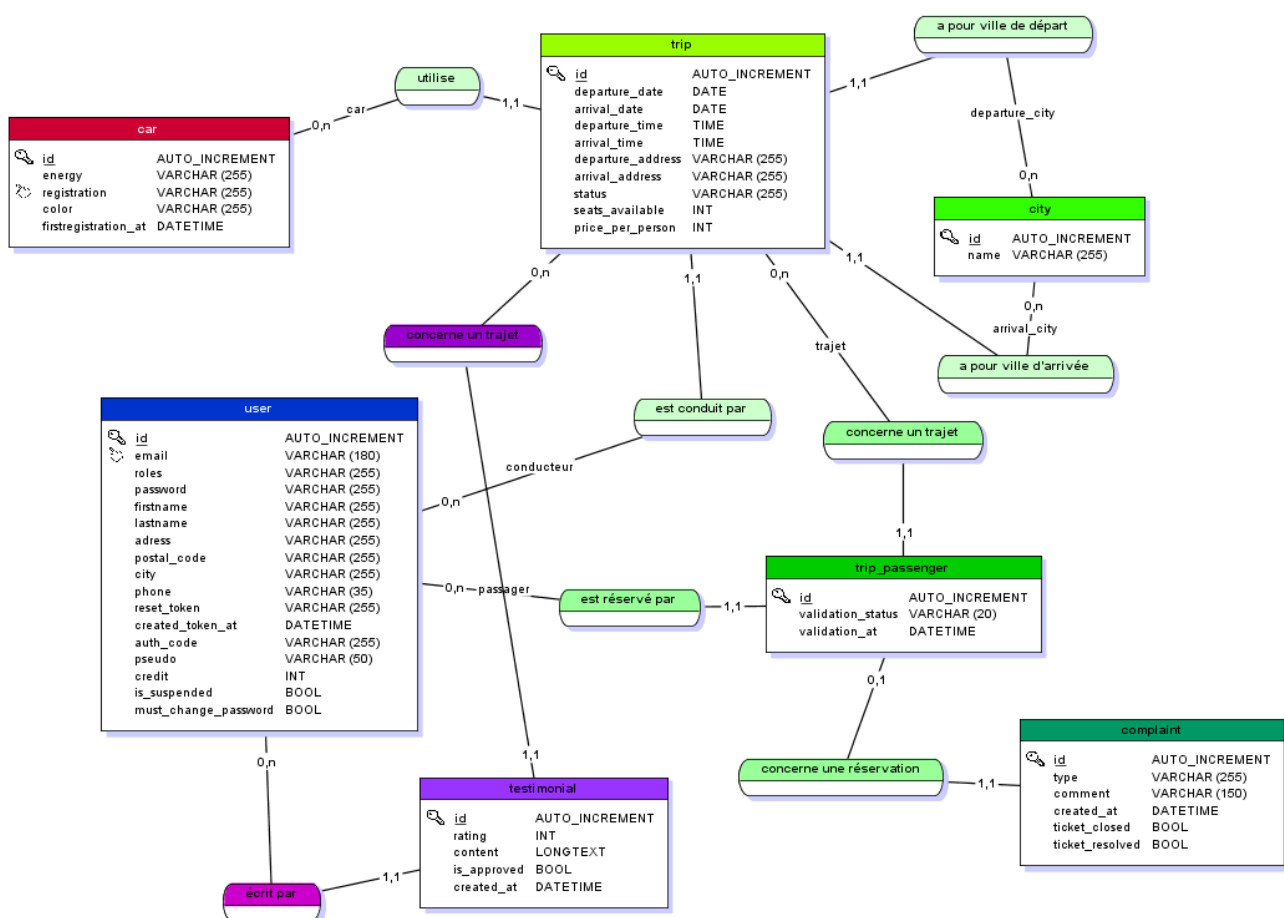
La table car est liée à model par une cardinalité **(1,1)** car chaque voiture est d'un modèle unique. Un modèle peut être partagé par plusieurs voitures : la cardinalité côté model est donc **(0,n)**.

5. car ↔ brand (est de marque)

Chaque voiture est liée à une marque (brand) par une cardinalité **(1,1)**. Une marque peut être utilisée par plusieurs voitures, d'où la cardinalité **(0,n)** côté brand.

6. model ↔ brand (appartient)

Un modèle appartient à une seule marque, soit une cardinalité **(1,1)** côté brand. Une marque peut proposer plusieurs modèles, soit une cardinalité **(0,n)** côté model.



MCD ECORIDE - partie 2

7. trip ↔ user (conducteur) (est conduit par)

Chaque trajet est conduit par un utilisateur, d'où une cardinalité **(1,1)** côté user. Un utilisateur peut être conducteur de plusieurs trajets, ce qui implique une cardinalité **(0,n)** côté user.

8. trip ↔ car (utilise)

Chaque trajet utilise une voiture **(1,1)**. Une voiture peut être utilisée dans plusieurs trajets

(0,n).

9. trip ↔ city (*a pour ville d...*)

Un trajet est lié à deux villes : une de départ et une d'arrivée. Dans les deux cas, la ville a une cardinalité **(1,1)** côté trip, et **(0,n)** côté city, car une ville peut être point de départ ou d'arrivée de plusieurs trajets.

10. tripPassenger ↔ trip (*concerne un trajet*)

Chaque passager est lié à un seul trajet **(1,1)**, tandis qu'un trajet peut avoir plusieurs passagers **(0,n)**.

11. tripPassenger ↔ user (*est réservé par*)

Chaque réservation de passager (trip_passenger) est liée à un seul utilisateur **(1,1)**, et un utilisateur peut être passager sur plusieurs trajets **(0,n)**.

12. complaint ↔ trip_passenger (*concerne une réservation*)

Une réclamation est liée à une réservation unique **(1,1)**. Une réservation peut ne pas faire l'objet d'une réclamation, ou au plus d'une seule **(0,1)**.

13. testimonial ↔ trip (*concerne un trajet*)

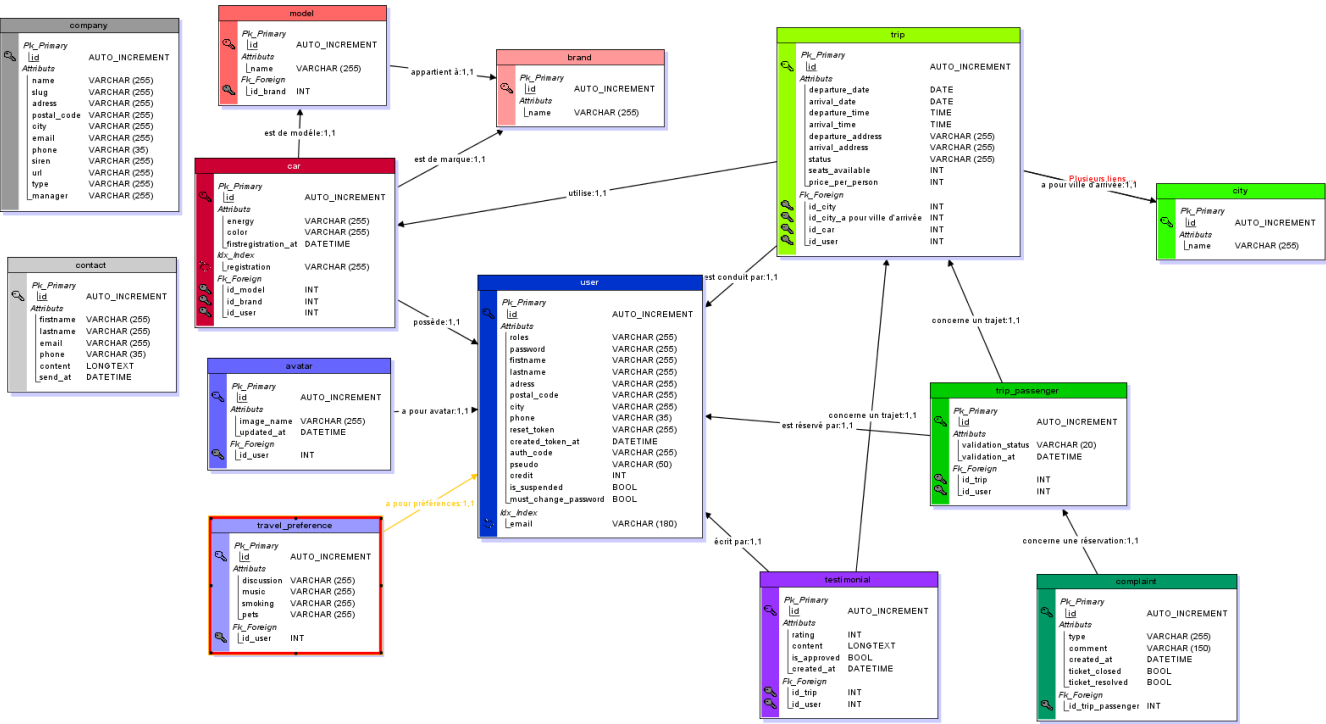
Chaque témoignage porte sur un trajet précis **(1,1)**. Un trajet peut faire l'objet de plusieurs témoignages **(0,n)**.

14. testimonial ↔ user (*écrit par*)

Chaque témoignage est rédigé par un utilisateur **(1,1)**. Un utilisateur peut rédiger plusieurs témoignages **(0,n)**.

MLD :

La table avatar qui a une Cardinalité 1,1 avec la table user aura alors une clé secondaire pour lier les deux tables avec l'id du user correspondant.

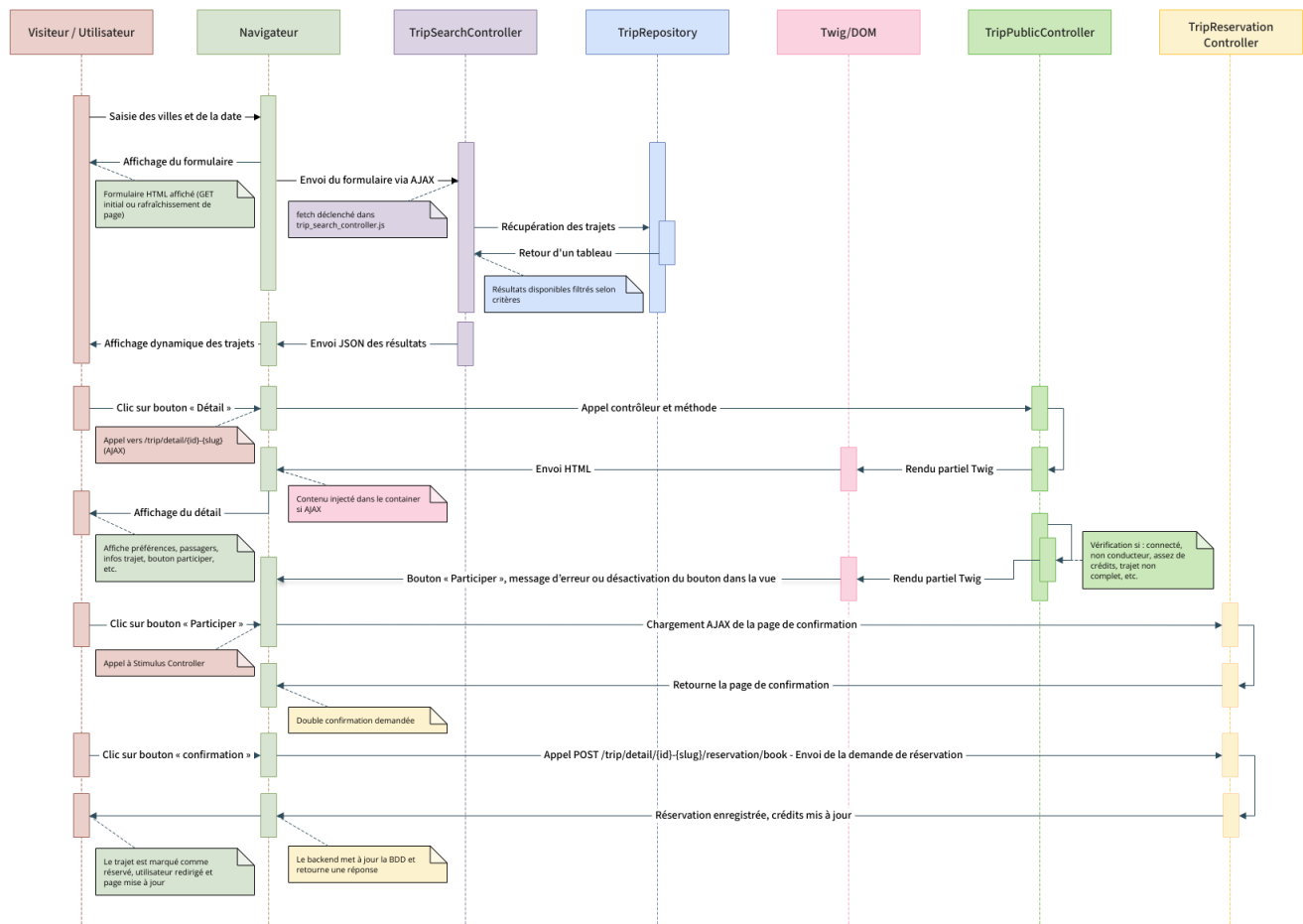


MPD (Modèle Physique de Données) (cf. annexes) :

- Chaque relation devient une table.
- Chaque attribut de la relation devient une colonne de la table correspondante.
- Chaque clé primaire devient une PRIMARY KEY.
- Chaque clé étrangère devient une FOREIGN KEY.

ii. Organisation des tâches du programme

Séquences modélisées de la recherche d'un trajet avec l'utilisation des filtres jusqu'à la réservation :



Séquence 1 – Recherche d'un trajet

Objectif : Permettre à un visiteur ou utilisateur de rechercher un trajet à partir d'une ville de départ, d'une ville d'arrivée et d'une date.

- L'utilisateur saisit les informations → formulaire AJAX est envoyé.
- Le contrôleur (TripSearchController) traite la demande.
- Le repository récupère les trajets filtrés.
- Les résultats sont retournés en JSON.
- Le navigateur affiche dynamiquement les cartes des trajets disponibles.

Séquence 2 – Consultation du détail d'un trajet

Objectif : Permettre de consulter les informations détaillées d'un trajet sélectionné.

- L'utilisateur clique sur « Détail » → appel du contrôleur TripPublicController.
- Le contrôleur rend une vue partielle Twig.
- Cette vue est injectée dans le DOM.
- Le détail du trajet s'affiche : conducteur, passagers, préférences, voiture, avis, etc.

Séquence 3 – Réservation d'un trajet

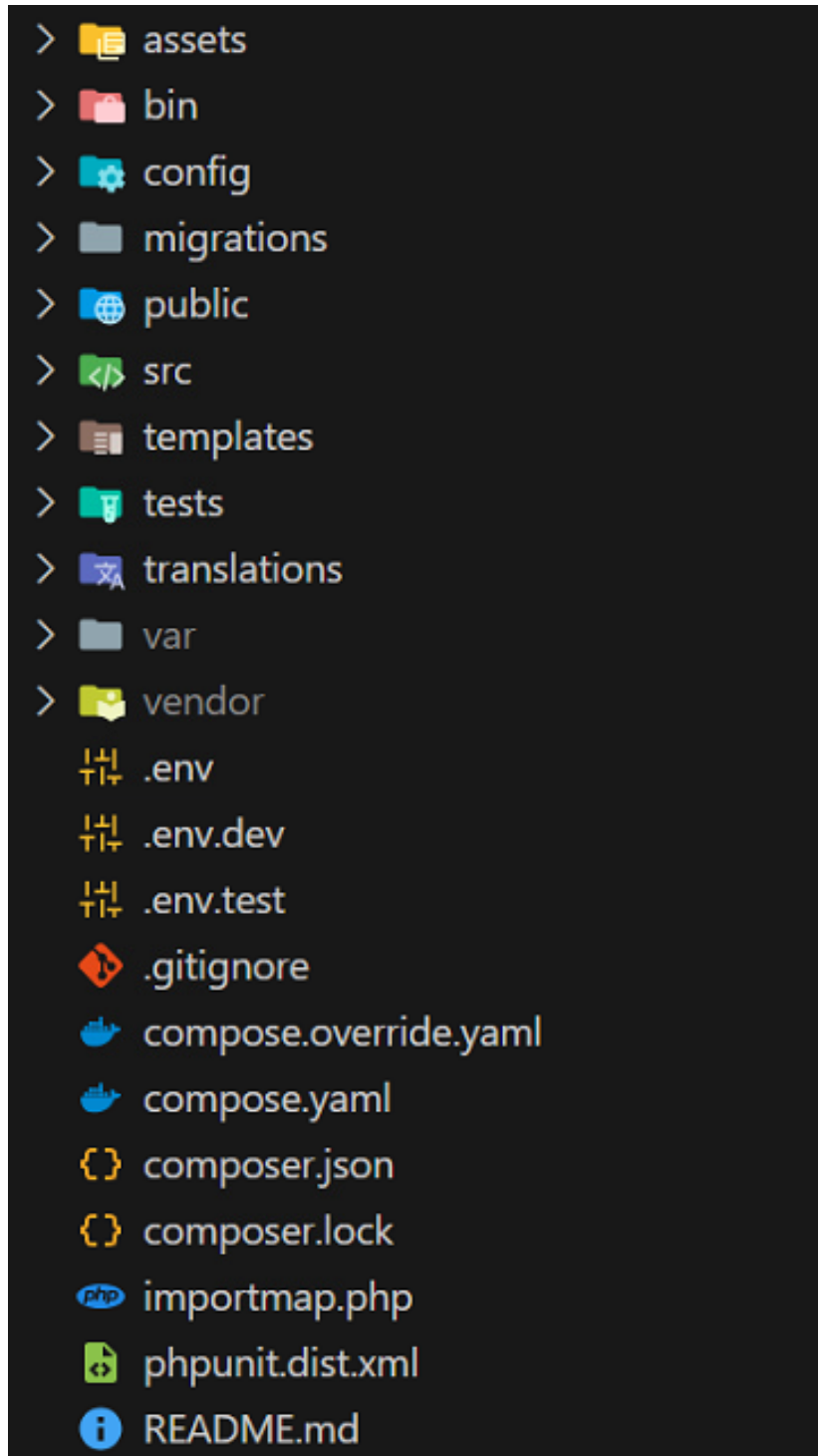
Objectif : Effectuer une demande de participation à un trajet, sous condition.

- L'utilisateur clique sur « Participer » :
 - S'il est non connecté / sans crédits / conducteur / déjà passager ou trajet complet → bouton désactivé.
 - Sinon, la page de **confirmation** s'affiche (AJAX).
- Un second clic confirme la réservation :
 - Appel POST /trip/detail/{id}-{slug}/reservation/book.
 - Le contrôleur vérifie toutes les conditions métier.
 - Si tout est ok : enregistrement en BDD, mise à jour des crédits, et rafraîchissement de la vue.

B. Développement

i. Architecture

L'application est construite selon une architecture MVC, « Model-View-Controller ». Elle est donc organisée de la façon suivante :



- Le dossier **assets** : ce dossier contient les fichiers sources utilisés pour les feuilles de style, les scripts JavaScript et les éventuelles ressources front-end comme les images ou les polices. Contrairement au dossier **public**, les fichiers présents dans **assets** ne sont

pas directement accessibles par le navigateur. Ils sont compilés (généralement via Webpack Encore ou AssetMapper) puis placés dans **public**. Ce processus permet notamment la minification, la transpilation ou encore l'ajout automatique de versions pour le cache.

- Le dossier **bin** contenant les exécutables pour effectuer des opérations. C'est notamment ici que viennent les commandes de la console Symfony.
- Le dossier **config** contenant tous les fichiers de configuration.
- Le dossier **migrations** contenant les différentes versions des schémas de l'application. On y retrouve les scripts SQL permettant de générer une structure de données en correspondance avec nos entités (les objets stockés en base de données).
- Le dossier **public** : le « document root » du projet. Il contient le contrôleur frontal index.php. Il s'agit du point d'entrée du site. Ce dossier contient également tous les fichiers destinés à nos visiteurs : images, fichiers CSS et JavaScript, etc. C'est le seul répertoire accessible par le serveur web.
- Le dossier **src** : toute la logique métier de l'application est écrite ici. Il contient le code source de l'application et notamment les contrôleurs, les entités, les formulaires, les services etc.
- Le dossier **templates** : ce dossier contient les templates Twig.
- Le dossier **tests** qui contient tous les tests unitaires et fonctionnels.
- Le dossier **translations** contenant les fichiers de traduction dans le cas de sites multilingues.
- Le dossier **var** : Il contient les données générées au moment de l'exécution comme le cache et les fichiers journaux.
- Le dossier **vendor** : Il contient toutes les bibliothèques tierces. Il est géré par Composer.
- Les fichiers **.env** (comme .env, .env.local, etc.) sont utilisés pour stocker les variables d'environnement. Ils permettent de configurer l'application Symfony sans modifier le code source, en fonction du contexte (développement, production, test...). On y définit, par exemple, les paramètres de connexion à la base de données, le mode de debug, ou encore les clés d'API. Le fichier .env.local est ignoré par Git afin de garder les données sensibles confidentielles.