Documentation technique

de l’application EcoRide

Leila El ouaazizi

Graduate developpeur full stack

Promo Janvier-Février 2026

1 Table des matières

2 Réflexion initiale technologique sur le sujet

2.1 Liste des technologies

Pour développer l'application, plusieurs technologies ont été utilisées :

- IDE : Visual Studio Code
- Framework back-end: Symfony 6.4
- Language : PHP 8.2
- Bases de données :
    - MySQL
    - MongoDB
    - Utilisation de Doctrine ORM et de Doctrine ODM
- Front-end :
    - Node.js
    - Bootstrap
- Authentication : Symfony Security
- Gestion des dépendances :
    - Composer (PHP)
    - npm
- Outils de développement :
    - Git (gestion de version)
    - GitHub (hébergement)
    - Symfony CLI
    - Extensions de VS Code comme Live Sass Compiler et PHP Server

2.2 Justification et conditions d'utilisation

Visual Studio Code

Condition d'utilisation : installé depuis le site officiel, utilisé sur Windows/Linux.

Justification : Visual Studio Code a été choisi comme environnement de développement intégré (IDE) pour les raisons suivantes :

- Logiciel libre et gratuit accessible sur Windows, macOS et Linux
- Compatibilité avec les langages du projet : HTML, CSS, Javascript, SCSS et Boostrap
- Dispose d'extensions pour faciliter le développement web
- Auto-complétion et suggestions de code
- Intégration Git pour la gestion de version
- Thèmes personnalisables

Symfony

Condition d'utilisation : nécessité d'un framework pour structurer une application web complète intégrant sécurité, routage, gestion des utilisateurs.

Justification : Symfony a été choisi en raison de :

- Ses fonctionnalités de sécurité intégrées (authentification, gestion des rôles tels que ROLE_USER, ROLE_ADMIN), protection contre les attaques courantes
- Son architecture MVC (Modèle-Vue-Contrôleur) qui facilite l'organisation du code
- Son écosystème riche incluant Doctrine ORM/ODM pour la gestion des bases de données, Mailer pour l'envoi d'e-mails, **Nelmio** pour la sécurité et la gestion des CORS
- Ses outils en ligne de commande (CLI) permettant de générer rapidement des entités, contrôleurs, migrations, et de lancer un serveur local.
- Sa compatibilité avec des bibliothèques externes, ce qui accélère le développement.

MySQL

Condition d'utilisation : gestion des relations entre entités.

Justification : MySQL a été choisi comme système de gestion de base de données relationnelles car :

- Il offre une structure relationnelle robuste avec des tables reliées par des clés primaires et étrangères.
- Les contraintes SQL assurent l'intégrité des données et la validation automatique des relations.

MongoDB

Condition d'utilisation : gestion des requêtes géospatiales.

Justification : MongoDB a été choisi pour :

- Sa capacité à stocker les noms de villes et données géospatiales
- Ses index 2dsphere qui permettent d'effectuer des recherches rapides par proximité

Node.js

Condition d'utilisation : besoin d'un environnement permettant d'exécuter du Javascript côté serveur et de gérer les dépendances du projet front-end.

Justification : Node.js a été utilisé car :

- Il offre un gestionnaire de paquets (npm) qui permet d'installer facilement des librairies comme Bootstrap ou Sass.
- Il facilite l'automatisation de tâches.

Bootstrap

Condition d'utilisation : besoin d'un front-end responsive permettant de gagner du temps.

Justification : Bootstrap a été utilisé car :

- Il propose de nombreux composants prêts à l'emploi (boutons, formulaires, etc.).
- Son système de grille responsive facilite la création d'interfaces adaptatives.
- Il réduit le temps de développement tout en assurant une qualité dans la conception front-end.
- Il garantit une compatibilité multi-appareils et multi-navigateurs.

Composer

Condition d'utilisation : gestion centralisée des dépendances PHP.

Justification : Composer a été choisi car :

- Il automatise l'installation et la mise à jour des bibliothèques.
- Il gère efficacement les dépendances indispensables au framework Symfony.

Symfony CLI

Condition d'utilisation : automatiser le développement.

Justification : Symfony CLI a été choisi car :

- Il facilite la création de fichiers (contrôleurs, entités, etc.).
- Il simplifie le lancement d'un serveur local pour tester l'application.

Git & GitHub

Condition d'utilisation : suivi de version et hébergement du code source.

Justification : Git et GitHub ont été choisis car :

- Git permet un suivi précis des modifications du code et facilite les retours en arrière.
- GitHub offre une plateforme collaborative pour héberger le projet et gérer les contributions.
- Les branches permettent de travailler en parallèle sans impacter la version principale du projet.
- L'intégration avec Visual Studio Code simplifie le processus.

Extensions

Condition d'utilisation : besoin d'extensions permettant d'automatiser certaines tâches front-end et de tester rapidement le rendu des pages.

Justification : ces extensions ont utilisées car :

- Live Sass Compiler permet de compiler automatiquement les fichiers Sass en CSS, facilitant la personnalisation du style et l'intégration de Bootrstrap.
- PHP Server permet de lancer un serveur local pour tester rapidement les pages HTML/CSS et vérifier leur rendu

3 Installation et configuration de l'environnement de travail front-end

3.1 Installation

- IDE (ex VS code) (https://code.visualstudio.com/)
- Extension VS Code PHP server (permet de lancer un serveur local rapidement)
- Node.js (https://nodejs.org/en/download)
- Bootstrap (https://getbootstrap.com/docs/5.3/getting-started/introduction/)
- Extension VS Code Live Sass Compiler

3.2 Configuration de l'environnement de travail front-end

Le front end est écrit en HTML, CSS, JavaScript.

Pour gérer la navigation du site, j'ai mis en place un système de routage organisé en trois fichiers principaux :

- Le fichier **Route.js** qui définit une classe *Route* représentant une route de l'application, caractérisée par son URL, son titre, le chemin vers un fichier HTML et éventuellement un chemin vers un fichier JavaScript associé.
- Le fichier **allRoutes.js** qui centralise l'ensemble des routes de l'application dans un tableau nommé *allRoutes* où chaque route est créée à partir de la classe *Route* avec les paramètres appropriés. Ce fichier définit également la variable *websiteName* qui représente le nom du site web.
- Enfin, le fichier **router.js** qui importe la classe Route et les variables *allRoutes* et *websiteName* du fichier allRoutes.js afin de mettre en œuvre la logique de routage, c'est-à-dire le chargement dynamique des vues en fonction de la navigation de l'utilisateur.

Pour la mise en forme, j'ai choisi CSS combiné au framework CSS Bootstrap afin de bénéficier de composants prêts à l'emploi et d'un système de grille réactif. Bootstrap et Bootstrap Icons ont été installés avec npm via les commandes suivantes :

```
npm install bootstrap
```

```
npm install bootstrap icons
```

Afin de personnaliser le style, j'ai utilisé Sass, un préprocesseur CSS qui facilite la gestion des variables et la réutilisation du code. J'ai créé un dossier **scss** contenant deux fichiers : **_custom.scss**, qui sert à surcharger et personnaliser les variables de Bootstrap et qui importe les fichiers de Bootstrap et **main.scss** qui importe à la fois **_custom.scss** et les fichiers de Bootstrap Icons. J'ai installé l'extension Live Sass Compiler qui permet de compiler **main.scss** en **main.css**, lequel est ensuite inclus dans le fichier **index.html** avec les liens de la police sélectionnée.

4 Installation et configuration de l'environnement de travail back-end

4.1 Installation

Les éléments cités dans la section 1.1 ont été téléchargés depuis les source suivantes:

- IDE (ex VS code) (https://code.visualstudio.com/)
- XAMPP (https://www.apachefriends.org/fr/download.html)
- Symfony CLI (https://symfony.com/download)
- Composer (https://getcomposer.org/download/)
- Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
- MongoDB et MongoDB Compass (https://www.mongodb.com/try/download/community)
- Extension MongoDB (https://pecl.php.net/package/mongodb)

Pour vérifier les exigences minimales pour le projet, j'ai utilisé la commande suivante:

```
symfony check:requirements
```

4.2 Configuration de l'environnement de travail back-end

Après avoir installé tous les composants cités dans la section 1, le projet Symfony EcoRide est créé avec la commande suivante :

```
composer create-project symfony/skeleton:"^6.4" Ecoride
```

4.2.1 Mise en place d'une base de données relationnelle

Pour simplifier l'accès aux données, j'utilise Doctrine ORM qui permet de créer un mappage entre les classes PHP appelées « Entités » et les tables de la base de données via des attributs. Cela permet de travailler avec des objets plutôt qu'avec des tables tout en réduisant les risques d'injections SQL. Pour installer doctrine, j'utilise la commande suivante :

```
composer require symfony/orm-pack
```

Pour m'aider à créer les entités Doctrine, j'utilise la commande suivante permettant d'obtenir MakerBundle qui est un composant aidant à générer du code :

```
composer require --dev symfony/maker-bundle
```

Pour s'assurer que le projet est à jour, j'utilise la commande :

```
composer update
```

La connexion entre Doctrine et la base de données MySQL se fait avec le fichier «*.env.local*» qui est une copie du fichier « *.env* », fichier de configuration global où sont stockées les identifiants SQL et permettant de stocker les variables d'environnement. Je crée un compte via phpMyAdmin et récupère les identifiants de celui-ci pour les mettre dans le fichier «*.env.local*» dans la variable DATABASE_URL :

```
DATABASE_URL="mysql://username:password@127.0.0.1:3306/EcoRide1?serverVersion=10.4.32-MariaDB&charset=utf8mb4"
```

Je crée ensuite la base de données via la commande :

```
php bin/console doctrine:database:create
```

4.2.2 Mise en place d'une base de données non relationnelle

Pour gérer les données géospatiales, notamment les noms de villes et leur coordonnées, j'utilise MongoDB, une base NoSQL, et Doctrine ODM que j'installe via la commande suivante :

```
composer require doctrine/mongodb-odm-bundle
```

La connexion à MongoDB se fait via le fichier .env.local en y insérant les informations suivantes :

MONGODB_URL=mongodb://localhost:27017

MONGODB_DB=villes

Doctrine permet de mapper les documents MongoDB sur des classes PHP.

Pour créer un document MongoDB :

```
Php bin/console make:document Ville
```

Les documents sont générés dans s**rc/Document** et les repositories dans **src/Repository**. Chaque document peut contenir des types géospatiaux comme les coordonnées GPS, permettant des requêtes rapides par proximité grâce aux index 2sphere.

Je télécharge l'extension MongoDB citée en section 1 que je place dans le dossier **ext** de **php** puis j'ajoute dans **php.ini** :

```
Extension=php_mongodb.dll.
```

Je lance MongoDB Compass et je me connecte au serveur local puis je crée une base de données et j'importe le fichier villes.json qui contient le nom des villes et leur coordonnées. Je crée enfin un index géospatial 2dshpere sur le champ location pour permettre des recherches par proximité. Le projet EcoRide peut maintenant stocker et interroger des données géospatiales, ce qui permet une recherche rapide des villes et coordonnées GPS.

4.2.3 Développement des composants d'accès aux données SQL et NoSQL

- **Accès aux données relationnelles (MySQL)**

J'utilise Doctrine ORM pour gérer les entités et les relations. Je crée les entités PHP correspondant aux tables du diagramme de classes avec MakerBundle et sa commande « *make:entity* » :

```
php bin/console make:entity Covoiturage
```

Je réponds aux questions demandées puis le code correspondant aux entités est généré dans le répertoire **src/Entity** et celui à leur classe de repository dans le dossier **src/Repository**. Les repositories sont des classes permettant d'interagir avec la base de données en effectuant des requêtes SQL via des méthodes PHP comme **findBy()** et **findOneBy()** sans écrire de SQL brut.

Avec la commande suivante, je génère une classe de migration. Celle-ci permet à Symfony de transformer une entité en un ensemble de requêtes SQL destinées à créer la table correspondante en base de données :

```
php bin/console make:migration
```

J'exécute les requêtes SQL de la classe migration permettant la création des tables dans la base de données via la commande suivante :

```
php bin/console doctrine:migrations:migrate
```

Les entités peuvent être sérialisées en JSON pour être envoyées au front-end via les API.

- **Accès aux données non relationnelles (MongoDB)**

J'utilise Doctrine ODM pour mapper les documents MongoDB sur des classes PHP. Je crée le document avec la commande :

```
Php bin/console make:document Ville
```

Les documents sont générés dans le répertoire **src/Document** et les repositories dans le dossier **src/Repository**. Les repositories permettent de faire des requêtes géospatiales, par exemple trouver des villes proches d'un point GPS via un index 2dsphere.

Les documents peuvent aussi être sérialisés en JSON pour les API.

- **Sérialisation et API**

La requête est souvent communiquée en JSON alors que le projet est écrit en PHP. Il est donc nécessaire de pouvoir transformer ces objets pour pouvoir interagir avec un client web. Pour cela, j'installe le composant **symfony/serializer** qui permet de transformer un objet PHP en un format spécifique (sérialisation) et vice versa (désérialisation) :

```
composer require symfony/serializer-pack
```

Pour que les routes exposant les données soient bien lues par Symfony, il faut s'assurer qu'elles soit configurées via des attributs dans les contrôleurs et que le fichier de configuration **config/routes.annotations.yaml** existe bien avec la configuration suivante :

```
resource: ../../src/Controller/

type: attribute

kernel:

resource: ../../src/Kernel.php

type: attribute
```

- **Gestion des requêtes cross-orgin (CORS)**

Pour autoriser les requêtes HTTP provenant d'origines différentes que celle où tourne mon serveur, j'utilise le bundle **NelmioCorsBundle** de **Nelmio** (la même organisation qui me permettra de documenter mon API) :

```
composer require nelmio/cors-bundle
```

Symfony Flex met automatiquement en place la configuration nécessaire et crée un fichier **config/packages/nelmio_cors.yaml** qui doit être configuré ainsi:

```
nelmio_cors:

      defaults:

            origin_regex: true

            allow_origin: ['%env(CORS_ALLOW_ORIGIN)%']

            allow_methods: ['GET', 'OPTIONS', 'POST', 'PUT', 'PATCH',
'DELETE']

            allow_headers: ['Content-Type', 'Authorization', 'X-AUTH-
TOKEN']

            expose_headers: ['Link']

            max_age: 3600
```

```
paths:
  '^/':
      allow_credentials: true
      origin_regex: true
      allow_origin: ['%env(CORS_ALLOW_ORIGIN)%']
      allow_methods: ['GET', 'OPTIONS', 'POST', 'PUT', 'PATCH',
'DELETE']
      allow_headers: ['Content-Type', 'Authorization','X-AUTH-TOKEN',
'X-CSRF-TOKEN']
      expose_headers: ['Link']
      max_age: 360
```

- **Composant Mailer**

Pour envoyer et recevoir des e-mails avec l'application j'utilise le composant **Mailer** de Symfony. C'est un outil performant et flexible qui permet de gérer et d'envoyer des e-mails de manière simple. Je l'installe avec la commande suivante :

```
composer require  symfony/mailer
```

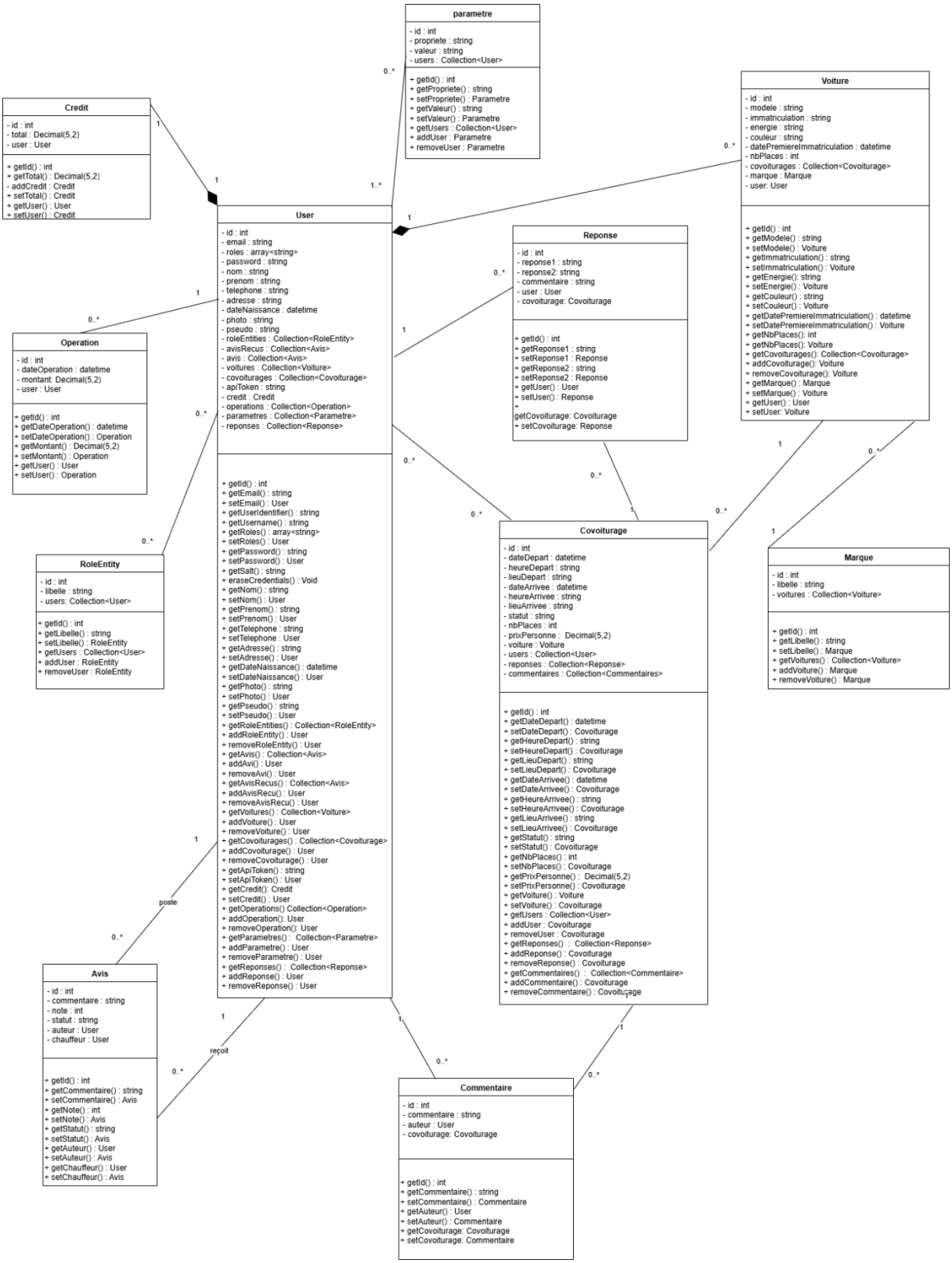
Ensuite, je configure le serveur SMTP, qui est un serveur informatique gérant la réception et l'envoi des e-mails, dans le fichier **.env.local** :

```
MAILER_DSN=smtp://user:pass@smtp.example.com:port
```

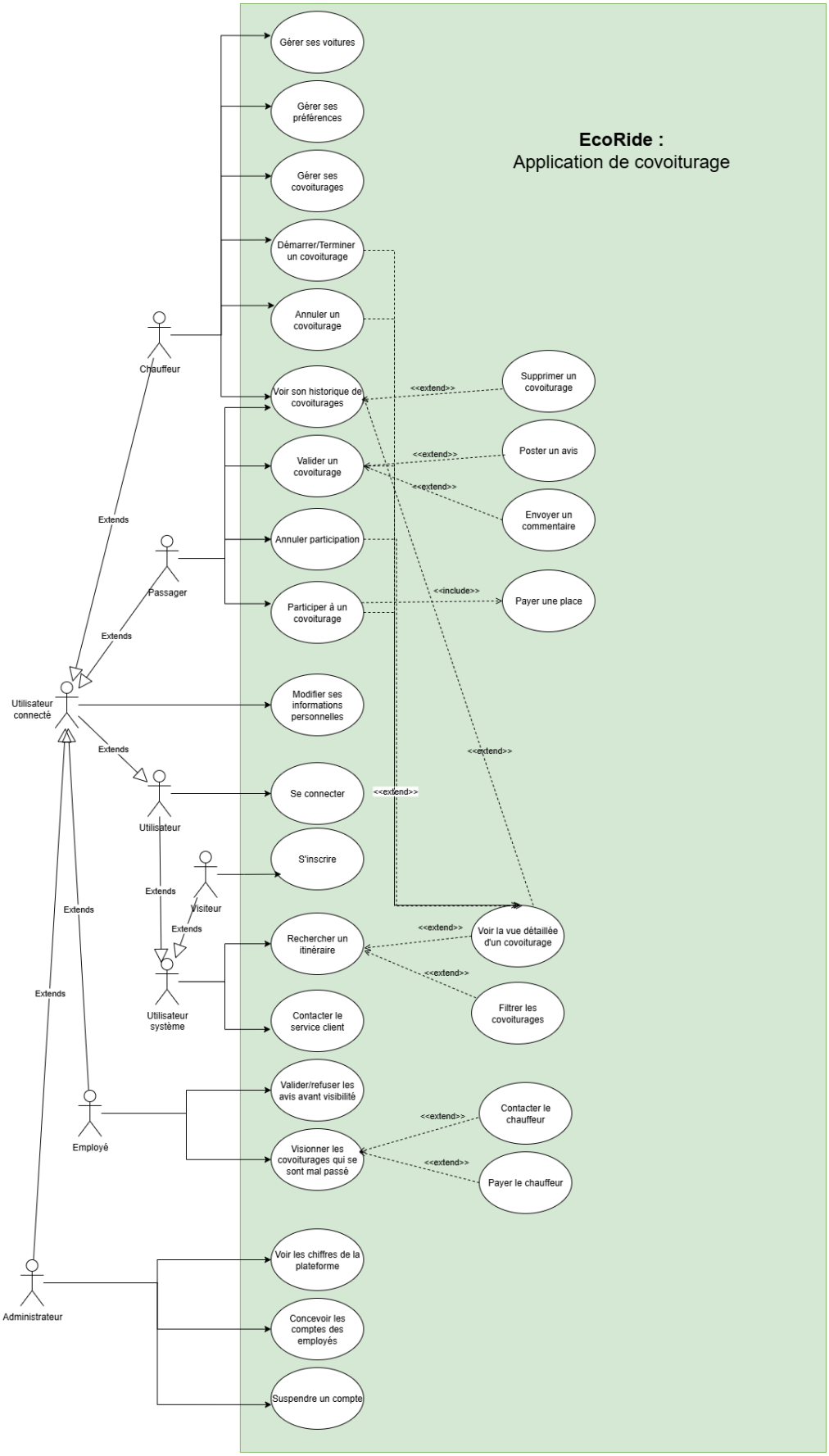
Enfin, pour les tests, je me suis inscris sur Mailtrap et j'ai récupéré l'identifiant et le mot de passe de la boîte qui m'a été attribuée.

5 Diagrammes

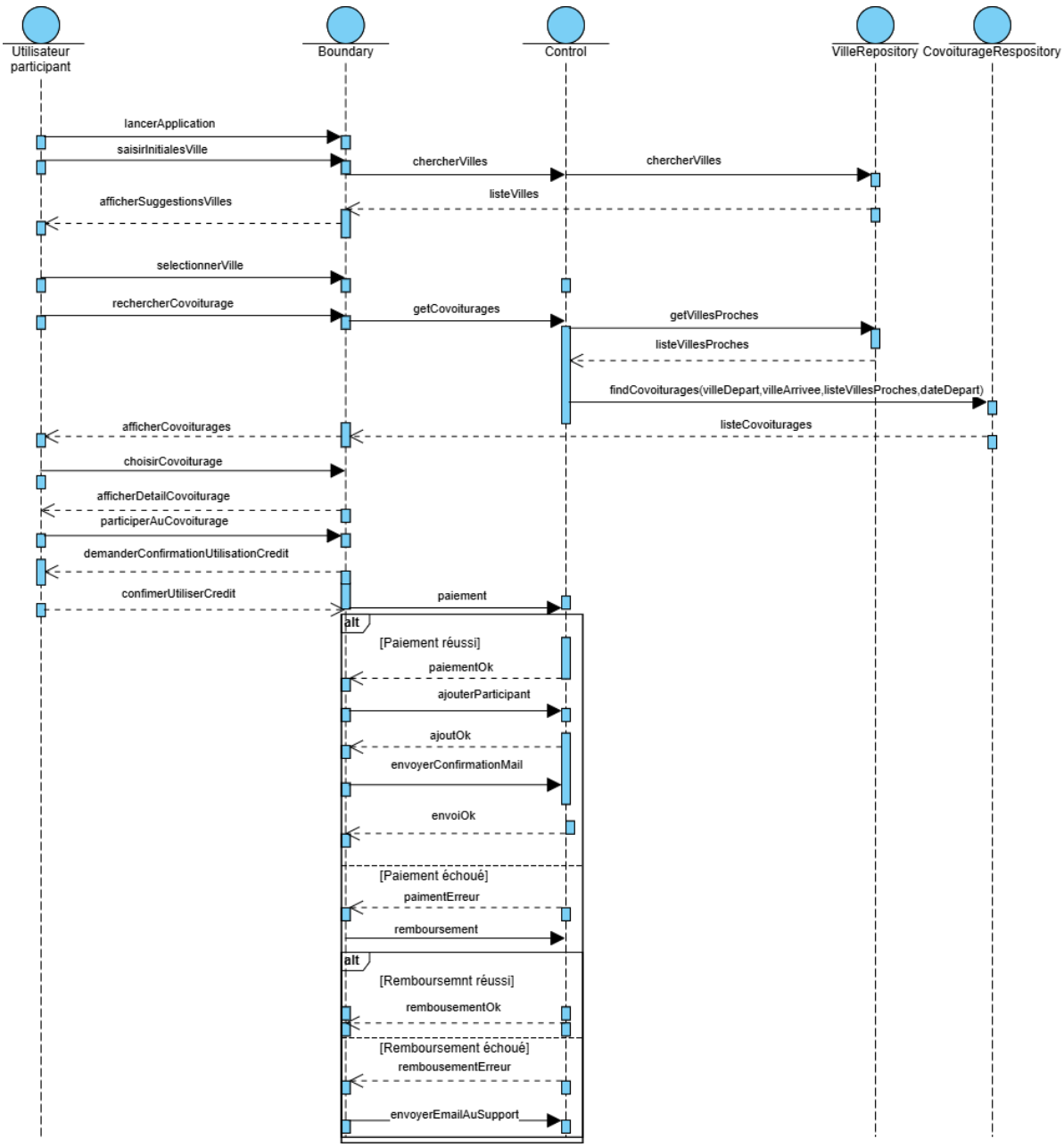
5.1 Diagramme de classes



5.2 Diagramme d’utilisation



5.3 Diagramme de séquence



6 Déploiement

L'objectif de cette partie est de décrire en détail le processus de déploiement de l'application EcoRide sur un hébergement mutualisé Hostinger. Le déploiement a été effectué en utilisant une connexion SSH afin de garantir une meilleure flexibilité dans l'exécution des commandes et la gestion de l'environnement serveur.

6.1 Prérequis

Avant de procéder au déploiement, j'ai vérifié que plusieurs conditions étaient remplies :

- Un compte Hostinger actif avec l'option SSH activée depuis le hPanel
- Un sous-domaine pointant vers l'hébergement et associé à un dossier nommé ecoride, situé dans le dossier public_html, destiné à accueillir le front-end
- Un second sous-domaine pointant vers l'hébergement et associé à dossier nommé api_symfony, situé dans le dossier public_html, destiné à accueillir le dossier public de Symfony
- Un dossier nommé symfony_app destiné à accueillir le reste du projet Symfony
- La base de données MySQL créée et configurée dans le hPanel
- Un compte MongoDB Atlas ainsi qu'une base de donnée et une collection contenant un fichier de villes au format JSON
- Les outils installés sur la machine locale à savoir : ssh et composer

6.2 Connexion au serveur via SSH

La première étape a constitué à se connecter au serveur distant fourni par Hostinger. Pour cela j'ai utilisé les informations disponibles dans le hPanel, à savoir l'hôte, le port et le nom d'utilisateur. La connexion a été établie à l'aide de la commande suivante :

```
ssh -p 65002 utilisateur@hostname
```

Une fois la connexion réussie, j'ai accédé au dossier nommé ecoride pointé par le sous-domaine.

6.3 Transfert des fichiers

Le code source a été transféré manuellement sur le serveur :

- Dans le dossier ecoride, j'ai transféré l'intégralité du dossier front-end.
- Dans le dossier api_symfony, j'ai transféré le contenu du dossier public de Symfony.
- Dans le dossier symfony_app j'ai transféré le reste du projet symfony, à l'exception du dossier vendor.

6.4 Installation des dépendances

Une fois les fichiers présents sur le serveur, j'ai procédé à l'installation des dépendances nécessaires au bon fonctionnement de l'application. Comme l'application a été développée en Symfony, j'ai utilisé Composer et la commande suivante :

```
composer install --no-dev --optimize-autoloader
```

6.5 Configuration de l'environnement

L'étape suivante a consisté à configurer l'environnement de production.

J'ai transféré mon fichier .env.local en remplaçant mes informations de connexion locales par celles correspondant :

- à la base de données MySQL créée sur Hostinger,
- et à la base MongoDB Atlas.

J'ai également modifié les variables suivantes :

APP_ENV=prod

APP_DEBUG=0

Ensuite, j'ai vidé le cache de Symfony et généré les fichiers optimisés pour la production :

```
php bin/console cache:clear –env=prod.
```

Enfin, j'ai exécuté les migrations afin de mettre à jour le schéma de la base de données :

```
php bin/console doctrine:migrations:migrate
```

6.6 Configuration du CORS

L'étape suivante a consisté à configurer les règles CORS afin de permettre les échanges entre l'API Symfony et le front-end. Pour cela j'ai modifié la variable suivante dans le fichier de configuration NelmioCorsBundle :

```
allow_origin: ['%env(CORS_ALLOW_ORIGIN)%']
```

Cette valeur a été remplacée par l'URL du sous-domaine correspondant au front-end, afin que celui-ci puisse communiquer correctement avec l'API.

6.7 Modification du fichier ApiTokenAuthenticator

Lors de la récupération d'un header personnalisé en PHP, certains serveurs transforment automatiquement son nom en le préfixant par HTTP_ et en le mettant en majuscules.

Par exemple : X-AUTH-TOKEN devient disponible dans $_SERVER['HTTP_X_AUTH_TOKEN '].

Pour assurer, la compatibilité quelle que soit la configuration PHP/serveur, la ligne suivante

```
$apiToken = $request->headers->get('X-AUTH-TOKEN');
```

a été modifiée en

```
$apiToken = $request->headers->get('X-AUTH-TOKEN') ??
$_SERVER['HTTP_X_AUTH_TOKEN'];
```