

Réflexion préliminaire technologique et choix de stack

1) Démarrage : d'abord vérifier que “ça marche”

Au début, je n'ai pas cherché à choisir “la meilleure stack”. Mon objectif était beaucoup plus basique : réussir à faire tourner quelque chose en local.

Je voulais valider rapidement :

- que le projet est bien accessible via localhost,
- que PHP s'exécute vraiment,
- et que je peux construire une arborescence sans me perdre.

Donc j'ai commencé simple, avec un environnement local qui me permettait de tester tout de suite la chaîne “fichier → serveur → navigateur”. L'idée était d'éviter de passer du temps sur une architecture compliquée alors que je n'avais même pas encore confirmé que l'exécution de base était juste.

2) Clarifier les parcours avant de coder trop vite

Dans l'application Ecoride il y a des rôles (visiteur, utilisateur connecté, employé, administrateur) et des règles métier (statuts de trajets, réservations, crédits, avis, modération, incidents).

Assez vite, je me suis rendu compte que le vrai danger, c'était de faire des écrans, puis de découvrir après coup que les parcours ne sont pas cohérents.

Du coup, j'ai pris beaucoup de temps sur la gestion de projet :

- écrire les user stories pour cadrer ce qui est attendu,
- utiliser un kanban pour prioriser et suivre l'avancement,
- faire un user story mapping pour voir clairement les parcours selon les rôles,
- utiliser des diagrammes (cas d'utilisation, séquences, classes) pour formaliser la logique.

Cela m'a été très utile et surtout évité de partir dans tous les sens et ça m'a servi de référence pour la suite.

3) Base relationnelle : une base qui empêche les incohérences

Le cœur d'EcoRide repose sur des données qui doivent rester cohérentes : utilisateurs, véhicules, trajets, participations, avis, crédits, commissions.

Pour ça, une base relationnelle était le choix le plus logique, parce qu'elle permet :

- des liens clairs entre les données (clés étrangères),
- des règles directement dans la base (contraintes sur les statuts, les places etc.),
- des transactions (éviter les mises à jour à moitié).

J'ai donc modélisé le schéma et ajouté un jeu de données de démo rejouable, avec :

- un ordre d'insertion logique (parents puis enfants),
- un script qui remet la base à zéro proprement,
- des requêtes de vérification pour montrer que les règles tiennent.

L'objectif était que la base protège la cohérence, même si l'application fait une erreur.

4) Remettre de l'ordre dans l'environnement pour éviter les faux problèmes

Pendant la mise en place, j'ai eu des problèmes assez classiques : confusion entre plusieurs environnements, connexions qui marchent "parfois", difficulté à savoir ce qui tourne réellement et où.

À ce moment-là, je me suis rendu compte qu'un environnement trop complexe trop tôt me faisait perdre du temps.

J'ai donc cherché à réduire les ambiguïtés :

- éviter les installations en double qui se contredisent,
- clarifier quel service tourne où,
- privilégier un environnement plus stable et plus facile à diagnostiquer.

C'était surtout une démarche "anti-perte de temps".

5) Passer à une stack reproductible : Docker + séparation claire des rôles

Une fois que j'avais validé le fonctionnement global et clarifié les besoins, je suis passée sur une stack plus "propre" et reproductible :

- Symfony pour structurer le projet (routes, contrôleurs, services, templates),
- PostgreSQL pour la base métier (la source de vérité),
- MongoDB pour un journal d'événements (traçabilité),

- Docker Compose pour lancer le tout ensemble,
- Nginx + PHP-FPM pour une exécution web standard.

Pourquoi Docker Compose ici ? Parce que ça me donne :

- un seul point de lancement pour tous les services,
- moins d'écart entre machines / configurations,
- un réseau interne clair (les services se parlent par nom),
- des volumes persistants pour les bases.

Et surtout : quand j'ai un problème (port occupé, service non joignable), je peux le corriger proprement sans tout casser.

6) Accès à PostgreSQL : PDO centralisé plutôt qu'un ORM

Pour la base relationnelle, j'ai choisi d'utiliser PDO avec une connexion centralisée, injectée dans mes services de persistance.

Ça me convient parce que :

- j'ai un seul point de configuration,
- les options PDO sont homogènes,
- je garde la main sur les requêtes SQL,
- et il y a moins "d'automatisme caché".

C'est aussi plus simple à expliquer et défendre dans un projet de certification : je sais exactement ce qui est exécuté.

7) MongoDB : journal technique

MongoDB n'est pas là pour remplacer PostgreSQL. Je l'utilise comme journal d'événements : une suite d'événements horodatés qui sert à garder une trace (debug, audit, statistiques futures).

Je garde donc une séparation très claire :

- PostgreSQL = données métier, contraintes, transactions,
- MongoDB = traçabilité et historique technique.

Ça évite de mélanger deux usages différents.

8) Interfaces : valider les parcours, puis appliquer la charte

Pour le front, j'ai fait pareil : d'abord fonctionnel, puis j'ai peaufiné.

Donc :

- templates simples pour vérifier routes, contrôleurs, formulaires et navigation,
- ensuite application progressive de la charte (CSS global, polices, cohérence visuelle, intégration des éléments de maquettes).

L'idée était de ne pas passer du temps sur le design tant que je n'étais pas sûre que les parcours fonctionnent.

Conclusion

Le choix de stack EcoRide s'est fait par étapes, plutôt que d'un coup :

- d'abord valider l'exécution locale,
- ensuite stabiliser les parcours et règles métier,
- consolider une base relationnelle cohérente,
- rendre l'environnement reproductible avec Docker,
- séparer données métier (PostgreSQL) et traçabilité (MongoDB),
- structurer Symfony avec un accès base maîtrisé (PDO),
- finir les interfaces dans le respect de la charte.

Le but était d'avoir une stack cohérente avec les besoins exprimés.