
Développé par **Jallal En Naour**

Titre professionnel visé : **Développeur Web et Web Mobile**

Novembre 2025

# Sommaire


1. **Introduction**
2. **Spécifications techniques**
3. **Liste des compétences**
4. **Écoconception**
5. **Gestion de projet**
6. **Base de données**
   - 6.1. Modélisation initiale
   - 6.2. Normes de conception
   - 6.3. Améliorations itératives
   - 6.4. Choix de l'ORM Doctrine
7. **Gestion des contrôleurs et des réponses**
8. **Sécurité**
9. **Design & Maquette**
   - 9.1. Design
   - 9.2. Maquette
10. **Front-end**
11. **Contenu et Interactivité**
12. **Sécurité**
13. **Road-map**
    - 13.1. API
    - 13.2. Front-end
14. **Veille Technologique**
15. **Conclusion**
16. **Annexe**

# 1. Introduction

J'aime l'automobile ! Mon entourage proche est également souvent confronté au marché de l'occasion. C'est là que j'ai trouvé l'inspiration qui me permet de vous présenter aujourd'hui le projet **La Conciergerie Auto**. Ce projet m'a permis de mettre en application les connaissances et les compétences acquises lors de ma formation CCI de Développeur Web et Web Mobile. Il s'agit d'une plateforme web de service qui agit comme un tiers de confiance pour l'achat et la vente de véhicules.

Il existe aujourd'hui un engouement certain pour le marché de l'occasion, motivé par des raisons économiques et écologiques. Cependant, les solutions actuelles entre particuliers manquent souvent de sécurité et de sérénité. **La Conciergerie Auto** répond à un besoin de confiance, de gain de temps et de simplification administrative. Elle permet de vendre son véhicule sans stress grâce à un expert dédié, et d'acheter en toute sécurité un véhicule inspecté et garanti.

La vente ou l'achat d'une voiture est une étape importante qui devrait être un plaisir, pas une source d'inquiétude. C'est un changement qui nous permet d'évoluer, de gagner en liberté ou de répondre à de nouveaux besoins. Mais il s'agit avant tout d'une transaction qui nécessite de la rigueur. Libre à nous de choisir la tranquillité d'esprit en déléguant cette tâche complexe.

**Bonne route et belles transactions avec La Conciergerie Auto !**

# 2. Spécifications techniques

## Vue.js

J'ai choisi Vue.js comme framework JavaScript pour le frontend en raison de son approche orientée utilisateur et de sa réactivité. Vue.js est reconnu pour sa simplicité et sa facilité d'intégration, ce qui me permet de développer rapidement des interfaces intuitives adaptées aux besoins spécifiques de La Conciergerie Auto.

En plus de son approche orientée utilisateur, Vue.js facilite la création d'applications single-page (SPA), offrant une expérience utilisateur fluide lors de la navigation entre le catalogue de véhicules, les demandes d'estimation et la gestion des rendez-vous. Son architecture basée sur des composants favorise la modularité, ce qui facilite la maintenance et l'évolution de l'application à long terme. Par exemple, les composants comme `VehicleCard` ou `EstimationTimeline` peuvent être réutilisés dans différentes parties de l'application.

Grâce à son écosystème riche, j'ai également la possibilité d'intégrer facilement des bibliothèques tierces comme Pinia pour la gestion d'état et Vue Router pour la navigation, augmentant ainsi les fonctionnalités de l'application sans complexifier le code. Cette approche modulaire est particulièrement adaptée à un projet comportant plusieurs rôles utilisateurs (Client, Concierge, Admin) avec des interfaces spécifiques pour chacun.

## TypeScript

L'intégration de TypeScript dans mon projet Vue.js me permet de bénéficier d'un typage statique qui renforce la robustesse et la maintenabilité du code frontend. TypeScript détecte les erreurs potentielles dès la phase de développement, ce qui réduit considérablement les bugs en production. Pour un projet comme La Conciergerie Auto qui gère des données sensibles (transactions, documents d'identité, estimations), cette sécurité supplémentaire est essentielle.

Les interfaces TypeScript me permettent de définir clairement la structure des données échangées avec l'API Symfony (véhicules, utilisateurs, estimations), facilitant ainsi la communication entre le frontend et le backend. Cette approche améliore également l'expérience de développement grâce à l'autocomplétion et à la documentation intégrée dans l'éditeur de code.

## CSS personnalisé

Pour le styling de l'application, j'ai opté pour du CSS personnalisé plutôt qu'un framework CSS lourd. Cette approche me permet de créer une identité visuelle unique pour La Conciergerie Auto avec un thème dark mode premium qui se démarque des plateformes automobiles traditionnelles.

L'utilisation de variables CSS (custom properties) me permet de maintenir une cohérence visuelle à travers toute l'application tout en facilitant les modifications futures du design. J'ai également mis en place un système de classes utilitaires réutilisables pour accélérer le développement des composants. Cette approche légère garantit des performances optimales, un critère important pour une application qui affiche des galeries photos de véhicules et doit offrir une expérience fluide sur mobile comme sur desktop.

## Outils de qualité de code

Pour garantir la qualité et la cohérence de mon code frontend, j'ai intégré ESLint et Prettier dans mon environnement de développement. ESLint me permet de détecter rapidement les erreurs et d'appliquer des règles de style, ce qui facilite la maintenance et réduit les bugs, particulièrement important dans un projet comportant de nombreux composants interconnectés. Prettier assure une mise en forme uniforme, qui renforce la lisibilité et l'homogénéité du code.

# Symfony

Dans le cadre de mon projet La Conciergerie Auto, j'ai choisi de développer une API RESTful en utilisant Symfony 6.4. Cette approche me permet de créer une interface de communication standardisée entre le frontend Vue.js et le backend, ce qui facilite l'échange de données pour toutes les fonctionnalités de la plateforme : gestion des véhicules, estimations, transactions et rendez-vous.

Symfony, avec sa robustesse et sa flexibilité, s'avère être un excellent choix pour développer une API complexe. J'utilise les composants de Symfony pour gérer les requêtes HTTP, les routes et la sérialisation des données via Doctrine ORM, assurant ainsi une structure claire et maintenable. L'architecture REST que j'ai mise en place me permet de créer des endpoints pour toutes les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) liées aux véhicules, estimations, transactions et utilisateurs, qui constituent le cœur de l'application.

La sécurité de mon API est primordiale, particulièrement dans un contexte d'intermédiation financière et de gestion de documents d'identité. Symfony offre des fonctionnalités intégrées pour gérer l'authentification et l'autorisation, ce qui me permet de protéger les données sensibles des utilisateurs et de contrôler l'accès à certaines ressources de l'application. En utilisant des tokens JWT (JSON Web Tokens) via le bundle LexikJWTAuthenticationBundle, je peux garantir que seules les requêtes authentifiées peuvent interagir avec les endpoints sécurisés. Le système de rôles hiérarchiques (ROLE_CLIENT, ROLE_CONCIERGE, ROLE_ADMIN) est géré nativement par le composant Security de Symfony.

Cette API RESTful est également conçue pour être évolutive, ce qui me permet d'ajouter facilement de nouvelles fonctionnalités (comme un système de notation ou de messagerie) sans perturber l'expérience utilisateur. Grâce à cette architecture, je suis en mesure d'offrir une expérience fluide et réactive tant sur le frontend que sur le backend.

# MySQL

J'ai retenu MySQL comme système de gestion de base de données relationnelle (SGBDR) en raison de sa fiabilité et de sa performance. Étant l'un des SGBDR les plus utilisés au monde, MySQL offre une scalabilité et une gestion des transactions solides, essentielles pour une application qui doit traiter des informations sensibles (transactions financières, documents d'identité) et des interactions utilisateur fréquentes.

La nature relationnelle de MySQL est particulièrement adaptée au modèle de données de La Conciergerie Auto, qui comporte de nombreuses relations entre entités : un véhicule est lié à un modèle, une marque, un vendeur, un concierge et potentiellement plusieurs photos et transactions.

MySQL me permet de gérer efficacement ces relations complexes tout en maintenant l'intégrité des données grâce aux contraintes de clés étrangères.

L'intégration avec Doctrine ORM facilite grandement la manipulation de ces données en PHP, me permettant de travailler avec des objets plutôt qu'avec des requêtes SQL brutes, tout en conservant la possibilité d'optimiser les requêtes critiques si nécessaire.

## AWS S3

Pour la gestion du stockage des fichiers, j'ai intégré Amazon S3 (Simple Storage Service). Ce choix s'impose pour un projet comme La Conciergerie Auto qui doit gérer un grand volume de photos de véhicules et de documents administratifs (cartes d'identité, documents d'assurance).

AWS S3 offre une solution de stockage cloud scalable, sécurisée et performante. J'ai mis en place un service S3Uploader personnalisé qui gère automatiquement la conversion des images au format WebP pour optimiser les performances de chargement, un aspect crucial pour l'expérience utilisateur sur mobile. Le système distingue les dossiers publics (photos de véhicules) et privés (documents d'identité), avec un accès aux fichiers privés via des URLs pré-signées temporaires (10 minutes) pour garantir la sécurité des données sensibles.

Cette architecture de stockage externe présente également l'avantage de ne pas surcharger le serveur web et de permettre une distribution géographique optimisée des contenus via le réseau CDN d'Amazon.

```php
public function upload(UploadedFile $file, string $directory = 'vehicles'): ?string
    {
        $originalFilename = pathinfo($file->getClientOriginalName(), PATHINFO_FILENAME);
        $safeFilename = $this->slugger->slug($originalFilename)->lower();
        $originalExtension = $file->guessExtension();

        $sourcePath = $file->getRealPath();
        $finalExtension = $originalExtension;
        $finalMimeType = $file->getMimeType();
        $isTempFile = false;

        if (in_array($originalExtension, ['jpg', 'jpeg', 'png'])) {
            $convertedWebPPath = $this->convertToWebP($sourcePath, $originalExtension);

            if ($convertedWebPPath) {
                $sourcePath = $convertedWebPPath;
                $finalExtension = 'webp';
                $finalMimeType = 'image/webp';
                $isTempFile = true;
            }
        }

        $newFilename = $safeFilename.'-'.uniqid().'.'.$finalExtension;
        $key = $directory . '/' . $newFilename;

        try {
            error_log("Tentative d'upload vers S3 - Bucket: {$this->bucketName}, Key: {$key}");

            $result = $this->s3Client->putObject([
                'Bucket' => $this->bucketName,
                'Key'    => $key,
                'SourceFile' => $sourcePath,
                'ContentType' => $finalMimeType,
            ]);

            error_log("Upload S3 réussi pour Key: {$key}");

            if ($isTempFile) {
                @unlink($sourcePath);
            }

            return $key;
```

```
    } catch (AwsException $e) {
        error_log("!! AWS S3 Upload Error: " . $e->getMessage() . " !! AWS Error Code: [" .
        if ($isTempFile) { @unlink($sourcePath); }
        return null;
    } catch (\Exception $e) {
         error_log("!! General Upload Error during S3 process: " . $e->getMessage());
         if ($isTempFile) { @unlink($sourcePath); }
         return null;
    }
}
```

## Git et GitHub

Pour la gestion de version de mon projet La Conciergerie Auto, j'ai choisi d'utiliser Git en conjonction avec GitHub. Git est un système de contrôle de version décentralisé qui me permet de suivre les modifications apportées à mon code de manière précise et organisée. Grâce à sa fonctionnalité de branches, je peux travailler sur de nouvelles fonctionnalités (comme l'ajout d'un système de messagerie entre clients et concierges) ou corriger des bugs sans impacter la version stable de l'application.

En utilisant GitHub, j'ai l'avantage de stocker mon code en ligne dans un repository privé, ce qui me garantit la sécurité et la confidentialité de mon travail. Même en travaillant seul, GitHub me permet de conserver un historique complet des modifications et de revenir facilement à des versions antérieures si nécessaire. Je veille également à effectuer des commits réguliers et descriptifs pour garder une traçabilité claire des modifications, une pratique essentielle dans le cadre d'un projet professionnel.

Ce système de versioning m'assure une gestion efficace et sécurisée du développement de La Conciergerie Auto, tout en me permettant de garder un œil sur l'évolution du projet et de documenter mon processus de développement pour le dossier professionnel.

## Visual Studio Code

Pour le développement de l'application La Conciergerie Auto, j'ai choisi Visual Studio Code (VS Code) comme environnement de développement intégré (IDE). Ce choix repose sur ses nombreuses fonctionnalités et sa flexibilité qui répondent parfaitement aux besoins de ce projet fullstack.

VS Code offre un excellent support pour le développement Vue.js avec TypeScript grâce à l'extension Volar, ainsi qu'un support natif pour PHP et Symfony. L'intégration Git, le terminal intégré et le système d'extensions me permettent de centraliser tous mes outils de développement dans une seule interface. Les fonctionnalités d'IntelliSense et de débogage facilitent grandement le développement et la résolution de problèmes.

## Déploiement - o2switch

Pour l'hébergement et le déploiement de La Conciergerie Auto, j'ai choisi o2switch, un hébergeur français qui propose une solution complète adaptée aux besoins de mon projet. O2switch offre un support PHP et MySQL nécessaire pour Symfony, ainsi qu'un certificat SSL pour sécuriser les échanges HTTPS, indispensable pour une application manipulant des données sensibles.

L'hébergement inclut également un service de mailing SMTP que j'utilise via symfony/mailer pour envoyer les emails de vérification de compte, de notification d'estimation et de confirmation de rendez-vous. Cette infrastructure me permet de déployer une application professionnelle sécurisée tout en respectant le budget d'un projet de formation.

# Conclusion

Ces choix technologiques résultent d'une réflexion approfondie sur les besoins spécifiques de La Conciergerie Auto. Ils visent à garantir une expérience utilisateur optimale pour les trois types d'utilisateurs (Clients, Concierges, Administrateurs) tout en facilitant le développement et la maintenance. En combinant ces technologies, je m'assure que le projet sera à la fois performant, évolutif et maintenable dans le temps.

De plus, l'intégration de Git et GitHub pour le versioning renforce cette approche en me permettant de gérer mon code de manière structurée et sécurisée. Même en travaillant seul sur un repository privé, je peux suivre l'historique des modifications et revenir facilement à des versions antérieures si nécessaire. Cela me garantit une gestion efficace du développement de La Conciergerie Auto, tout en me permettant de rester concentré sur l'innovation et l'amélioration continue de l'application.

J'insiste également sur l'importance de l'évolutivité et de la maintenance continue dans le choix de ces technologies. En optant pour des outils largement adoptés et bien soutenus par la communauté (Symfony, Vue.js, MySQL), je me positionne pour gérer avec succès les futures évolutions de La Conciergerie Auto. Cette plateforme d'intermédiation automobile pourra ainsi s'adapter aux besoins changeants du marché et aux retours des utilisateurs, tout en maintenant un haut niveau de qualité et de sécurité.

La séparation claire entre le backend API (Symfony) et le frontend SPA (Vue.js) permet également une flexibilité future, comme le développement potentiel d'une application mobile native qui pourrait consommer la même API. Cette architecture moderne et modulaire constitue une base solide pour un projet professionnel de qualité.

# 3. Liste des compétences

Ce projet couvre les compétences du référentiel suivant :

• Configurer son environnement de travail
• Maquetter une application
• Réaliser une interface utilisateur web statique et responsive
• Réaliser la partie dynamique
• Base de données
• Développer les composants métier
• Création d'une documentation

## Configurer son Environnement de Travail

Pour garantir un développement fluide et productif, j'ai configuré mon environnement de travail avec les outils nécessaires au développement front-end et back-end de La Conciergerie Auto. Utilisant VS Code comme IDE, j'ai installé et configuré divers plugins pour optimiser le linting et le formatage du code, notamment avec ESLint et Prettier pour le frontend Vue.js, ainsi que des extensions pour PHP et Symfony (Intelephense, Symfony Support).

La gestion de l'API a été facilitée par Bruno (ou Postman), un client API qui m'a permis de tester les endpoints RESTful et de maintenir une collection de requêtes organisée par fonctionnalité (authentification, véhicules, estimations, transactions, rendez-vous). Cette organisation a permis de valider rapidement le bon fonctionnement de l'API backend avant l'intégration frontend.

J'ai également configuré l'environnement local avec XAMPP pour le serveur MySQL et PHP, ainsi que Composer pour la gestion des dépendances Symfony. L'intégration de Git via VS Code a facilité le versioning du projet. Ces choix ont structuré un environnement de travail cohérent et propice à un développement efficace d'une application fullstack professionnelle.

## Maquetter une Application

Pour poser les fondations de l'expérience utilisateur sur La Conciergerie Auto, j'ai élaboré les maquettes des principales pages en adoptant une approche mobile-first. Cette décision s'avère particulièrement pertinente dans le secteur automobile, où les utilisateurs consultent fréquemment les annonces de véhicules depuis leur smartphone.

Les pages principales incluent la page d'accueil avec présentation des services, le catalogue de véhicules avec système de filtres avancés, la page de détail d'un véhicule avec galerie photos, le

formulaire de demande d'estimation pour les clients, ainsi que les interfaces spécifiques pour les concierges (pool d'estimations, gestion des véhicules) et les administrateurs (dashboard desktop-only avec gestion complète).

Chaque maquette a été pensée pour offrir une navigation intuitive adaptée au rôle de l'utilisateur (Client, Concierge, Admin), avec une attention particulière portée à l'accessibilité et à la hiérarchie visuelle de l'information. Le thème dark mode premium avec les couleurs signature (vert néon #9dff80 sur fond noir) a été défini dès cette étape pour créer une identité visuelle distinctive dans le secteur automobile. Cette approche a permis de créer une vue d'ensemble complète de l'application, offrant une direction claire pour la mise en œuvre des fonctionnalités.

## Réaliser une Interface Utilisateur Web Statique et Responsive

Une fois les maquettes finalisées, j'ai développé la version statique et responsive de chaque page de La Conciergerie Auto. Utilisant du CSS personnalisé avec un système de variables CSS (custom properties) pour une gestion cohérente des couleurs, espacements et rayons de bordure, j'ai pu garantir une adaptabilité et une ergonomie optimales sur divers formats d'écran.

J'ai mis en place un système de breakpoints clairs (mobile < 768px, tablet 768-1024px, desktop ≥ 1024px) et créé des classes utilitaires réutilisables ( `.btn` , `.card` , `.form-group` , `.badge` , etc.) qui accélèrent le développement tout en maintenant une cohérence visuelle. L'approche mobile-first m'a permis de construire des interfaces qui s'adaptent naturellement aux écrans plus larges grâce aux media queries.

Une attention particulière a été portée à l'optimisation des performances avec des images responsive et un chargement progressif des galeries de photos de véhicules. Grâce à cette base statique solide et ce système de design cohérent, j'ai structuré un modèle visuel stable dans lequel les fonctionnalités dynamiques ont pu être intégrées sans compromettre l'expérience utilisateur.

## Réaliser la Partie Dynamique

Pour rendre La Conciergerie Auto interactif, j'ai développé la partie dynamique en utilisant Vue.js 3 avec TypeScript et la Composition API. L'intégration de composants dynamiques tels que le système de filtres de véhicules, les galeries photos interactives, les formulaires de demande d'estimation avec validation en temps réel, et les cartes de véhicules cliquables a permis de créer une expérience utilisateur fluide et réactive.

J'ai utilisé Vue Router pour gérer la navigation entre les différentes pages et implémenter les guards de navigation qui contrôlent l'accès aux routes selon le rôle de l'utilisateur (Client, Concierge, Admin).

Par exemple, les routes administratives sont protégées et nécessitent le rôle ROLE_ADMIN, avec une redirection automatique pour les utilisateurs non autorisés.

Pinia a été choisi comme store de gestion d'état pour centraliser les données utilisateur (authentification, profil), les véhicules, et les filtres actifs. Le store `authStore` gère notamment l'authentification JWT, la récupération automatique du profil utilisateur au chargement de l'application, et la persistance du token dans le localStorage.

J'ai développé un service API centralisé avec Axios ( `apiClient` ) qui ajoute automatiquement le token JWT dans les headers et gère les erreurs globales (déconnexion automatique en cas de token expiré). Les composables comme `useResponsive` permettent de gérer facilement le comportement responsive (affichage du menu mobile vs desktop, adaptation des filtres).

Ces choix technologiques permettent de gérer efficacement les interactions utilisateurs complexes (upload de photos, validation de formulaires multi-étapes, gestion des états de transaction) tout en maintenant une structure claire, modulaire et évolutive.

# Base de Données

La conception de la base de données de La Conciergerie Auto a été pensée de manière rigoureuse en s'appuyant sur la méthode Merise pour modéliser et structurer les données de manière optimale. La première étape a consisté à élaborer un Modèle Conceptuel de Données (MCD) afin d'identifier les entités essentielles telles que les utilisateurs (User), les profils utilisateurs (ProfileUser), les marques (Brand), les modèles (Model), les véhicules (Vehicle), les photos de véhicules (VehiclePhoto), les demandes d'estimation (EstimationRequest), les transactions (Transaction) et les rendez-vous (Appointment). À partir de ce MCD, j'ai pu établir les relations entre ces entités et définir leurs attributs clés.

Les relations identifiées incluent notamment :

- Un utilisateur possède un profil (relation 1:1)
- Une marque possède plusieurs modèles (relation 1:N)
- Un véhicule appartient à un modèle, un vendeur et un concierge (relations N:1)
- Un véhicule possède plusieurs photos (relation 1:N)
- Une estimation concerne un client et peut être traitée par un concierge (relations N:1)
- Une transaction lie un acheteur et un véhicule (relations N:1)
- Un rendez-vous implique un client, un concierge et éventuellement un véhicule (relations N:1)

Ensuite, le Modèle Logique de Données (MLD) a été conçu pour formaliser et organiser les tables en tenant compte des règles d'intégrité et des dépendances fonctionnelles. Ce modèle logique a permis d'assurer une structure cohérente et une manipulation efficace des données, facilitant le stockage et la récupération d'informations complexes comme l'historique complet d'un véhicule (vendeur, estimation d'origine, photos, transactions, rendez-vous).

La méthode que j’ai conçue s’est révélée particulièrement utile pour anticiper et résoudre les problèmes d’intégrité référentielle, un enjeu crucial dans un contexte d’intermédiation où chaque véhicule, transaction et document doit rester traçable et sécurisé. En modélisant les relations entre les entités et en utilisant les clés étrangères de manière stratégique avec Doctrine ORM, j'ai garanti l'intégrité des données et minimisé les risques de redondance ou d'incohérence.

J'ai également défini des énumérations pour les statuts (statut de véhicule : LISTED/RESERVED/SOLD, statut d'estimation : PENDING_POOL/CLAIMED/ESTIMATED/ACCEPTED/REJECTED, statut de transaction : PAYMENT_PENDING/COMPLETED) qui structurent les workflows métier de l'application. Ce processus de conception a permis de mettre en place une base solide, normalisée et évolutive, indispensable pour supporter les futures évolutions de La Conciergerie Auto.

# Développer les Composants Métier

Le développement des composants métier a permis de structurer les fonctionnalités essentielles de La Conciergerie Auto autour des véhicules, utilisateurs, estimations, transactions et rendez-vous. Ces composants, intégrés dans Symfony avec une architecture MVC, assurent une séparation claire des responsabilités et facilitent l'ajout de nouvelles fonctionnalités.

Les principaux contrôleurs métier développés incluent :

**AuthController** : Gère l'inscription des utilisateurs avec envoi d'email de vérification, la vérification du compte, et l'authentification JWT via le endpoint `/login_check`.

**VehicleController** : Permet la consultation du catalogue public avec filtres (marque, modèle, prix, kilométrage, année), la consultation du détail d'un véhicule, ainsi que les opérations CRUD complètes pour les concierges (ajout avec upload de photos vers S3, modification, suppression de photos).

**EstimationRequestController** : Gère le workflow complet des estimations : création par un client, pool d'estimations disponibles pour les concierges, attribution (claim) d'une estimation à un concierge, soumission d'un prix estimé, acceptation ou rejet par le client.

**TransactionController** : Supervise le processus d'achat avec création de transaction, confirmation de paiement, upload du document d'assurance, et consultation de l'historique des transactions.

**AppointmentController** : Coordonne la prise de rendez-vous avec vérification des disponibilités du concierge, création de rendez-vous (visite ou récupération), et gestion des statuts (REQUESTED/CONFIRMED/CANCELLED).

**AdminController** : Fournit des interfaces CRUD complètes pour la gestion des utilisateurs, véhicules, estimations et transactions, avec génération d'URLs S3 pré-signées pour l'accès sécurisé aux documents privés.

J'ai également développé des services métier réutilisables comme **S3Uploader** pour la gestion unifiée des uploads vers AWS S3 avec conversion WebP automatique, et **EmailService** (via symfony/mailer) pour l'envoi des notifications tout au long des workflows.

En structurant l'application en modules métiers cohérents avec des repositories Doctrine dédiés pour les requêtes complexes, j'ai pu construire une architecture évolutive et extensible qui respecte les principes SOLID. Cette organisation permet de répondre aux besoins futurs (ajout d'un système de notation, de messagerie, ou d'historique de maintenance) sans compromettre la cohésion du code existant.

# Création d'une Documentation

La documentation de l'API, cruciale pour assurer une continuité dans le développement de La Conciergerie Auto et faciliter une éventuelle collaboration future, a été réalisée de manière exhaustive. J'ai créé un document technique complet

( `readme.md` ) qui décrit :

- L'architecture globale du projet et les technologies utilisées
- Le schéma détaillé de la base de données avec toutes les entités et leurs relations
- La liste complète des endpoints de l'API organisés par fonctionnalité
- Les mécanismes de sécurité (JWT, rôles, accès aux fichiers privés)
- Les services externes intégrés (AWS S3, SMTP)

L'utilisation de Bruno (ou Postman) comme client API a permis de créer une collection de requêtes documentées pour chaque endpoint, avec des exemples de payloads JSON et les réponses attendues. Cette collection peut être exportée et partagée, offrant un accès rapide et clair aux informations essentielles pour tester l'API.

Pour le frontend, j'ai documenté la structure des stores Pinia, les interfaces TypeScript définissant les contrats de données, et les composables réutilisables. Les composants Vue complexes sont commentés pour expliquer leur logique métier.

Cette documentation complète pose des bases solides pour la maintenance de l'application, l'onboarding de futurs développeurs, et constitue un élément important du dossier professionnel présenté dans le cadre du titre professionnel de Développeur Web niveau Bac+2.

# 4. Écoconception

Dans la conception de La Conciergerie Auto, j'ai pris en compte les principes de l'écoconception afin de minimiser son empreinte écologique tant au niveau du frontend que du backend. Le choix des technologies comme Vue.js pour une interface réactive et Symfony pour une gestion efficace des requêtes a été motivé par leur performance et leur impact environnemental mesuré.

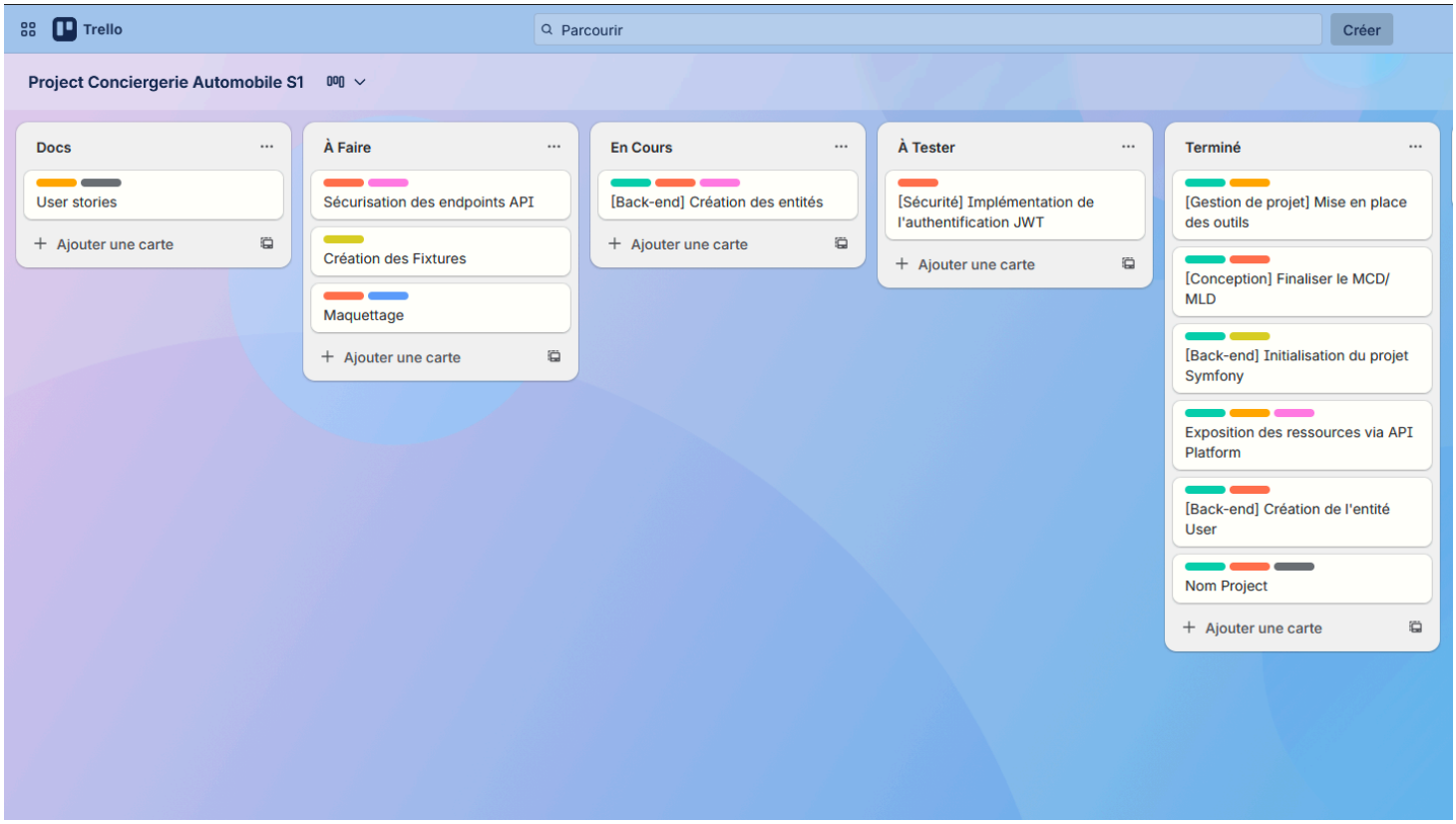
Par exemple, l'usage de pratiques telles que le chargement paresseux des images (lazy loading) pour les galeries de véhicules et l'optimisation automatique au format WebP via le service S3Uploader ont permis de réduire significativement le poids des pages. La conversion WebP réduit la taille des images de 25 à 35% sans perte de qualité, minimisant ainsi la consommation de bande passante.

J'ai également optimisé les requêtes API en utilisant des filtres côté serveur plutôt que de charger l'ensemble du catalogue côté client. J'ai aussi limité le nombre de dépendances en choisissant des solutions légères et indispensables. Par exemple, l'utilisation de CSS personnalisé plutôt qu'un framework lourd réduit le poids initial de l'application de plusieurs centaines de kilooctets, améliorant le temps de chargement et limitant la consommation énergétique globale de la plateforme.

# 5. Gestion de projet

Dans le cadre du développement de La Conciergerie Auto, j'ai mis en place une gestion de projet rigoureuse en adoptant une approche MVP (Minimum Viable Product) afin de concentrer mes efforts sur les fonctionnalités essentielles qui répondent directement aux besoins des utilisateurs. Cela m'a permis de prioriser les éléments les plus importants comme le catalogue de véhicules, le système d'estimation et l'authentification JWT, et de garantir une première version stable et fonctionnelle, tout en prévoyant des améliorations futures (système de messagerie, notation des concierges).

Pour gérer efficacement les tâches, j'ai utilisé Trello comme outil de gestion de projet. Grâce aux tickets organisés en colonnes (Backlog, En cours, Tests, Terminé), j'ai pu planifier les tâches par fonctionnalité (authentification, véhicules, estimations, transactions), définir des priorités et suivre l'avancement en temps réel. Cette méthode agile a facilité la visualisation des progrès du projet et m'a permis de m'adapter rapidement aux nouvelles contraintes techniques ou aux ajustements à intégrer suite aux tests utilisateurs.



# 6.Base de données

## 6.1 Modélisation initiale

Dès le début du projet La Conciergerie Auto, j'ai accordé une attention particulière à la structure de la base de données pour m'assurer qu'elle réponde aux besoins fonctionnels de l'application tout en restant évolutive. En appliquant la méthode Merise, j'ai réalisé une analyse approfondie des entités et des relations nécessaires pour structurer les informations de manière cohérente et optimale. Cette méthode m'a permis de concevoir la base de données en trois étapes clés : le modèle conceptuel des données (MCD), le modèle logique des données (MLD) et enfin le modèle physique des données (MPD).

Le modèle conceptuel des données (MCD) représente les entités principales comme les utilisateurs, les véhicules, les marques, les modèles, les estimations, les transactions, les rendez-vous et leurs relations. J'ai ainsi identifié les éléments essentiels qui allaient structurer le cœur de l'application et leurs interactions. Par exemple, dans la table des utilisateurs (User), j'ai non seulement défini des attributs de base comme l'email et le mot de passe hashé, mais j'ai aussi ajouté des champs comme les rôles (sous forme de tableau JSON pour gérer ROLE_CLIENT, ROLE_CONCIERGE, ROLE_ADMIN), le statut de vérification du compte (isVerified) et les dates de création et de mise à jour pour permettre une meilleure gestion des utilisateurs sur le long terme. J'ai également créé une entité ProfileUser séparée en relation OneToOne avec User pour stocker les informations complémentaires (adresse, téléphone, date de naissance, chemins des documents d'identité), respectant ainsi le principe de séparation des préoccupations.

Dans la table des véhicules (Vehicle), j'ai inclus des informations spécifiques telles que le prix, l'année, le kilométrage, la description et le statut (LISTED, RESERVED, SOLD) afin de répondre aux besoins de l'application pour un affichage détaillé de chaque annonce. J'ai structuré les relations de manière à ce qu'un véhicule soit toujours lié à un modèle (qui lui-même appartient à une marque), un vendeur (User) et un concierge (User) responsable de la gestion. Cette hiérarchisation Brand → Model → Vehicle permet une organisation cohérente et facilite les filtres de recherche par marque ou modèle.

Ensuite, j'ai élaboré le modèle logique des données (MLD) pour décrire les relations entre les entités en passant d'une représentation conceptuelle des informations à une structuration plus concrète. En utilisant la méthode Merise, j'ai pu identifier précisément les cardinalités, comme par exemple les relations ManyToOne entre les véhicules et leurs modèles, les OneToMany entre un véhicule et ses photos (VehiclePhoto), ou encore les relations entre les demandes d'estimation (EstimationRequest) et les clients/concierges qui les traitent. Ce modèle logique a permis d'optimiser la structuration de la

base de données pour la phase de développement, assurant ainsi une correspondance entre les besoins fonctionnels identifiés (workflows d'estimation, processus de transaction, gestion des rendez-vous) et la représentation technique de la base de données.

Enfin, avec le modèle physique des données (MPD), j'ai transposé la conception en tables concrètes dans le système de gestion de base de données MySQL en créant les tables avec leurs attributs et contraintes. Le passage au MPD m'a permis de m'assurer que la structure soit adaptée aux exigences de performance en tenant compte de l'indexation et de l'intégrité référentielle. J'ai notamment créé des index sur les colonnes fréquemment utilisées dans les recherches comme le statut des véhicules, l'ID du modèle, ou le statut des estimations. J'ai également défini des champs de type JSON pour des informations flexibles comme les détails du véhicule dans les estimations (vehicleDetails) et les rôles des utilisateurs, afin de rendre la base de données plus flexible et de permettre des évolutions futures sans restructuration lourde.

## 6.2 Normes de conception

Dans la conception de cette base de données, j'ai appliqué les principes de normalisation des données pour optimiser la structure et réduire la redondance. En utilisant la méthode Merise, j'ai pu analyser chaque entité et ses relations en respectant les principes de la première, deuxième et troisième formes normales. Par exemple, les informations des marques (Brand) et des modèles (Model) sont isolées dans des tables dédiées plutôt que d'être répétées pour chaque véhicule, ce qui évite la duplication des données et assure une meilleure intégrité. De même, les photos de véhicules sont stockées dans une table séparée (VehiclePhoto) avec une relation OneToMany, permettant une gestion flexible du nombre de photos par véhicule. Le principe DRY (Don't Repeat Yourself) est également un pilier de la conception, car il assure que chaque information ne soit stockée qu'une seule fois, facilitant ainsi la gestion des données et réduisant le risque d'incohérences.

J'ai également respecté les bonnes pratiques de sécurité et d'intégrité des données. En définissant des clés primaires auto-incrémentées et des contraintes de clés étrangères avec des règles de cascade appropriées, j'ai renforcé la cohérence des données dans la base. Pour chaque table, j'ai également mis en place des contraintes NOT NULL pour des champs critiques, garantissant ainsi que certaines informations essentielles, comme l'email pour les utilisateurs, le prix pour les véhicules, ou le transactionId pour les transactions, soient toujours présentes dans la base de données. Les mots de passe sont systématiquement hashés avec l'algorithme bcrypt via le composant Security de Symfony, assurant la protection des informations sensibles.

Enfin, en utilisant des index sur les colonnes fréquemment requêtées comme les identifiants des véhicules, les statuts, et les clés étrangères (modelId, sellerId, conciergeId), j'ai pu optimiser les

performances de la base de données, particulièrement pour les requêtes de recherche et de filtrage du catalogue qui sont susceptibles d'être appelées fréquemment par les utilisateurs.

Grâce à l'utilisation de ces normes, la base de données de La Conciergerie Auto est non seulement performante mais également adaptable aux évolutions futures de l'application, comme l'ajout potentiel d'un système de notation des concierges ou d'un historique de maintenance des véhicules.

## 6.3 Améliorations itératives

Lors des phases de tests, j'ai pu recueillir des retours qui m'ont permis de faire évoluer la base de données. Ces retours ont révélé des besoins d'ajustement pour mieux refléter les cas d'utilisation réels et optimiser la performance. Par exemple, la gestion initiale des photos de véhicules s'est avérée trop basique. Après analyse, j'ai ajouté un champ `isPrimary` dans la table VehiclePhoto pour identifier la photo principale d'un véhicule, facilitant ainsi l'affichage des vignettes dans le catalogue sans avoir à charger toutes les photos à chaque fois. J'ai également ajouté un ordre de tri pour organiser les photos dans la galerie.

Les tests ont également révélé la nécessité d'améliorer la traçabilité des transactions. J'ai donc enrichi l'entité Transaction en ajoutant un champ `transactionId` unique généré automatiquement, ainsi qu'un champ pour stocker le chemin du document d'assurance (insuranceDocPath) uploadé sur S3. Cette évolution permet de sécuriser le processus d'achat et de conserver une preuve documentaire de chaque transaction.

De plus, en analysant les workflows des concierges, j'ai identifié le besoin d'un système de statuts plus granulaire pour les estimations. J'ai donc défini une énumération complète (PENDING_POOL, CLAIMED, ESTIMATED, ACCEPTED, REJECTED) qui représente fidèlement le cycle de vie d'une demande d'estimation, depuis sa création par un client jusqu'à son acceptation ou son rejet. Cette fonctionnalité a permis aux concierges de mieux suivre leur charge de travail et aux clients de comprendre l'avancement de leur demande.

J'ai également ajouté la table Appointment suite aux retours utilisateurs qui exprimaient le besoin de planifier des visites de véhicules et des récupérations. Cette table permet de gérer les rendez-vous avec un système de types (VIEWING pour une visite, PICKUP pour une récupération) et de statuts (REQUESTED, CONFIRMED, CANCELLED). L'itération de la conception a donc été un processus clé pour garantir que la structure de la base de données soit non seulement performante, mais aussi en parfaite adéquation avec les attentes des utilisateurs et les réalités opérationnelles du métier d'intermédiation automobile.

# 6.4 Choix de l'ORM Doctrine

Pour gérer les relations complexes entre les tables, j'ai choisi d'utiliser Doctrine comme ORM (Object-Relational Mapping). Doctrine m'a offert une solution puissante et flexible pour gérer efficacement la persistance des données dans la base de données en facilitant les opérations de création, de mise à jour et de suppression des enregistrements, tout en travaillant avec des objets PHP plutôt qu'avec du SQL brut.

Doctrine a particulièrement facilité la gestion des relations complexes de La Conciergerie Auto. Par exemple, la relation entre un véhicule et ses multiples photos (OneToMany), les relations ManyToOne entre un véhicule et son modèle, son vendeur et son concierge, ou encore les relations bidirectionnelles entre les utilisateurs et leurs estimations ont été configurées simplement grâce aux annotations (ou attributs PHP 8) de Doctrine. Le système de lazy loading de Doctrine permet également d'optimiser les performances en ne chargeant les entités liées que lorsque cela est nécessaire.

De plus, Doctrine propose un système de repositories personnalisables qui m'a permis d'encapsuler la logique de requêtes complexes. Par exemple, j'ai créé une méthode dédiée `findUserAppointments()` dans AppointmentRepository pour récupérer tous les rendez-vous d'un utilisateur, qu'il soit le client ou le concierge concerné. Cette méthode utilise le QueryBuilder de Doctrine pour construire une requête SQL optimisée avec deux conditions `WHERE` liées par un `OR`, permettant de récupérer les rendez-vous dans les deux cas de figure, puis les trie par date décroissante pour afficher les plus récents en premier.

Cette approche respecte le principe de séparation des responsabilités et facilite la maintenance du code en centralisant les requêtes métier dans des classes Repository dédiées plutôt que de les disperser dans les contrôleurs. Les contrôleurs peuvent ainsi simplement appeler `$appointmentRepository->findUserAppointments($user)` pour obtenir les données nécessaires, rendant le code plus lisible, testable et maintenable.

```php
/**
     * Trouve les rendez-vous où l'utilisateur est soit client, soit concierge.
     * @return Appointment[]
     */
    public function findUserAppointments(User $user): array
    {
        return $this->createQueryBuilder('a')
            ->where('a.client = :user')
            ->orWhere('a.concierge = :user')
            ->setParameter('user', $user)
            ->orderBy('a.scheduleAt', 'DESC')
            ->getQuery()
            ->getResult();
    }
```

# 7 Gestion des contrôleurs et des réponses

Gestion des réponses : mes contrôleurs ne se contentent pas de traiter les requêtes, ils sont également responsables de la gestion des réponses envoyées au client. Par exemple, après avoir créé un nouveau véhicule, la méthode `add()` du VehicleController renvoie une réponse HTTP appropriée, incluant un code de statut 201 (Created) et les détails du véhicule créé au format JSON. Cette approche standardisée garantit une communication claire entre le frontend Vue.js et le backend Symfony.

## Création d'un véhicule

La méthode `add()` du VehicleController est fondamentale pour permettre l'ajout de nouveaux véhicules à l'API par les concierges. Accessible via une requête HTTP de type POST sur l'endpoint `/api/vehicles/add` , cette méthode reçoit les données JSON envoyées dans le corps de la requête, incluant les informations du véhicule (modelId, sellerId, price, year, mileage, description) ainsi que les fichiers photos à uploader.

Une fois les données reçues, la méthode procède à une validation des informations à l'aide du composant de validation de Symfony. Cette étape est cruciale pour garantir que toutes les informations nécessaires telles que le modèle, le prix, l'année et le kilométrage sont fournies et respectent les contraintes définies (par exemple, le prix doit être positif, l'année doit être cohérente). Si des erreurs de validation sont détectées, une réponse JSON est renvoyée avec un statut 422 (Unprocessable Entity), accompagnée d'une liste détaillée des erreurs. Cela permet à l'utilisateur de corriger les informations avant de soumettre à nouveau la requête.

Si toutes les validations passent avec succès, la méthode traite l'upload des photos vers AWS S3 via le service S3Uploader. Ce service gère automatiquement la conversion des images au format WebP pour optimiser les performances et organise les fichiers dans le bucket S3 dans un dossier public. Pour chaque photo uploadée, une entité VehiclePhoto est créée avec le chemin de l'image stocké.

La méthode appelle ensuite `persist()` pour préparer l'entité Vehicle à être sauvegardée dans la base de données, suivie de `flush()` qui applique réellement les modifications. Le véhicule est automatiquement créé avec le statut 'LISTED' et associé au concierge connecté. À la fin du processus, une réponse JSON est renvoyée avec un statut 201 (Created), incluant les détails du véhicule nouvellement créé avec ses photos, son modèle (incluant la marque), le vendeur et le

concierge. Cette structure de réponse complète permet au frontend d'afficher immédiatement le véhicule sans avoir à effectuer de requête supplémentaire.

## Création d'une demande d'estimation

La méthode `add()` du EstimationRequestController permet aux clients de soumettre une demande d'estimation pour leur véhicule. Accessible via une requête POST sur l'endpoint `/api/estimation-requests/add`, cette méthode reçoit les détails du véhicule sous forme JSON (marque, modèle, année, kilométrage, type de carburant, transmission, couleur) ainsi qu'un message optionnel du client.

Dès réception de la requête, la méthode vérifie que l'utilisateur authentifié possède bien le rôle ROLE_CLIENT. Les données sont ensuite validées pour s'assurer que les informations essentielles sont présentes. Si la validation échoue, une réponse 422 est renvoyée avec les détails des erreurs.

Une fois validée, l'entité EstimationRequest est créée avec le statut 'PENDING_POOL', la rendant immédiatement disponible dans le pool d'estimations accessibles aux concierges. Un email de confirmation est envoyé au client via le service EmailService pour l'informer que sa demande a bien été prise en compte. La méthode renvoie ensuite une réponse 201 avec les détails de l'estimation créée, permettant au frontend de rediriger l'utilisateur vers une page de suivi.

## Mise à jour d'un véhicule

La méthode `update()` du VehicleController est essentielle pour permettre la modification des informations d'un véhicule existant dans l'API. Elle est accessible via une requête HTTP de type PUT sur l'endpoint `/api/vehicles/{id}`, où {id} correspond à l'identifiant du véhicule à mettre à jour. Cette fonctionnalité est réservée aux concierges et administrateurs.

Dès qu'une requête est reçue, la méthode commence par vérifier si le véhicule correspondant à l'ID fourni existe. Si le véhicule n'est pas trouvé, une réponse JSON est renvoyée avec un statut 404 (Not Found) et un message d'erreur explicite, indiquant que le véhicule n'existe pas. Cela garantit une communication claire avec l'utilisateur de l'API et évite toute confusion.

Si le véhicule est trouvé, la méthode vérifie ensuite que l'utilisateur connecté a bien les droits nécessaires pour modifier ce véhicule. Un concierge ne peut modifier que les véhicules dont il est responsable, tandis qu'un administrateur peut modifier n'importe quel véhicule. Si l'utilisateur n'a pas les droits, une réponse 403 (Forbidden) est renvoyée.

Une fois les vérifications effectuées, la méthode met à jour les champs modifiables du véhicule (prix, kilométrage, description, statut) avec les nouvelles données fournies dans la requête. Une validation

est effectuée à l'aide du composant de validation de Symfony. Si des erreurs sont détectées (par exemple, si le prix est négatif ou si le statut n'est pas valide), une réponse 422 (Unprocessable Entity) est renvoyée avec les détails des erreurs. Cela permet à l'utilisateur de comprendre rapidement ce qui doit être corrigé.

Si toutes les validations réussissent, la méthode appelle `persist()` pour préparer l'entité à être sauvegardée, suivie de `flush()` pour enregistrer les modifications dans la base de données. Si de nouvelles photos sont fournies, elles sont uploadées sur S3 via le service S3Uploader et ajoutées au véhicule. Enfin, une réponse JSON est renvoyée avec le statut 200 (OK), incluant les détails du véhicule mis à jour avec toutes ses relations (modèle, marque, photos).

## Gestion des transactions

Le TransactionController gère le processus complet d'achat d'un véhicule. La méthode `create()` permet à un utilisateur d'initier une transaction pour l'achat d'un véhicule disponible. Elle vérifie d'abord que le véhicule existe et qu'il a le statut 'LISTED' (disponible). Si le véhicule est déjà réservé ou vendu, une erreur 400 (Bad Request) est renvoyée.

Une nouvelle transaction est ensuite créée avec un identifiant unique (transactionId) généré automatiquement, le statut 'PAYMENT_PENDING', et les relations vers l'acheteur et le véhicule. Le statut du véhicule est automatiquement mis à jour à 'RESERVED' pour empêcher d'autres achats simultanés. Une réponse 201 est renvoyée avec les détails de la transaction, permettant au frontend de guider l'utilisateur vers les étapes suivantes (paiement, upload du document d'assurance).

La méthode `uploadInsurance()` permet ensuite à l'acheteur d'uploader son document d'assurance. Le fichier PDF est stocké dans un dossier privé sur S3, et le chemin est sauvegardé dans la transaction. Cette étape est essentielle avant la finalisation de la vente.

En résumé, les contrôleurs de mon API REST sont conçus pour être modulaires, intuitifs et robustes. Grâce à une structure claire et à des méthodes bien définies, ils permettent de gérer efficacement les ressources (véhicules, estimations, transactions, rendez-vous), tout en garantissant une expérience utilisateur fluide et réactive. Cette architecture facilite également les évolutions futures du projet, en permettant d'ajouter de nouvelles fonctionnalités comme un système de messagerie ou de notation sans perturber l'ensemble de l'application.

# 8.Sécurité

La sécurité est un aspect essentiel de l'application La Conciergerie Auto, particulièrement dans la gestion des accès aux différentes fonctionnalités selon les rôles utilisateurs et la protection des données sensibles (documents d'identité, informations de transaction). J'ai utilisé les fonctionnalités de sécurité robustes de Symfony pour garantir une protection complète de l'application, en m'appuyant sur les composants suivants : le système de rôles hiérarchiques pour le contrôle d'accès, l'Access Control de Symfony pour restreindre les routes, le bundle JWT pour une authentification sécurisée, et des contrôles manuels dans les contrôleurs pour les permissions fines.

## Système de rôles hiérarchiques

Dans La Conciergerie Auto, j'ai mis en place une hiérarchie de rôles claire qui reflète les différents niveaux d'accès nécessaires à la plateforme :

**ROLE_CLIENT** : Les clients peuvent consulter le catalogue de véhicules, créer des demandes d'estimation, visualiser leurs estimations et rendez-vous, et initier des transactions d'achat.

**ROLE_CONCIERGE** : Les concierges héritent des permissions CLIENT et peuvent en plus accéder au pool d'estimations, réclamer et traiter des estimations, créer et gérer des véhicules, uploader des photos, gérer les transactions et leur agenda de rendez-vous.

**ROLE_ADMIN** : Les administrateurs héritent des permissions CONCIERGE et disposent d'un accès complet à toutes les fonctionnalités, incluant la gestion des utilisateurs, l'accès à tous les véhicules et estimations, la génération d'URLs S3 pré-signées pour les documents privés, et l'accès au dashboard d'administration.

**Principe de fonctionnement** : Symfony évalue automatiquement la hiérarchie des rôles définie dans le fichier de configuration `security.yaml` . Par exemple, un utilisateur avec ROLE_ADMIN possède automatiquement les permissions de ROLE_CONCIERGE et ROLE_CLIENT sans avoir besoin de les déclarer explicitement dans son tableau de rôles. Cette approche simplifie la gestion des permissions tout en maintenant une flexibilité maximale.

**Utilisation dans La Conciergerie Auto** : Pour des opérations sensibles comme la modification d'un véhicule, les contrôleurs vérifient non seulement le rôle de l'utilisateur mais aussi sa relation avec la ressource. Par exemple, un concierge peut uniquement modifier les véhicules dont il est responsable (où `conciergeId` correspond à son ID), tandis qu'un administrateur peut modifier n'importe quel

véhicule. Cette logique de contrôle d'accès granulaire est implémentée directement dans les méthodes des contrôleurs.

## Access Control de Symfony

L'Access Control de Symfony est utilisé pour appliquer des règles de sécurité globales aux routes de l'application. J'ai défini dans le fichier `security.yaml` des restrictions d'accès pour chaque catégorie de routes, en spécifiant les rôles nécessaires pour y accéder.

**Routes publiques** : Les routes comme `/api/register`, `/api/login_check`, `/api/vehicles` (consultation du catalogue) et `/api/vehicles/{id}` (détail d'un véhicule) sont accessibles sans authentification (access_control : `PUBLIC_ACCESS`).

**Routes CLIENT** : Les routes préfixées par `/api/estimation-requests/add`, `/api/estimation-requests/mine`, `/api/profile` nécessitent le rôle ROLE_CLIENT minimum.

**Routes CONCIERGE** : Les routes comme `/api/estimation-requests/pool`, `/api/estimation-requests/{id}/claim`, `/api/vehicles/add`, `/api/transactions/managed` nécessitent ROLE_CONCIERGE.

**Routes ADMIN** : Toutes les routes préfixées par `/api/admin` nécessitent ROLE_ADMIN et sont exclusivement réservées aux administrateurs pour la gestion complète de la plateforme.

En utilisant cette fonctionnalité, je m'assure qu'aucune route sensible n'est accessible publiquement, réduisant ainsi les risques d'accès non autorisé aux données personnelles des utilisateurs, aux documents d'identité ou aux fonctionnalités d'administration. Le système renvoie automatiquement une erreur 401 (Unauthorized) pour les utilisateurs non authentifiés tentant d'accéder à une route protégée, et 403 (Forbidden) pour les utilisateurs authentifiés mais sans le rôle requis.

## Bundle JWT (LexikJWTAuthenticationBundle)

Pour l'authentification, j'ai intégré le bundle LexikJWTAuthenticationBundle, un système qui permet une authentification sécurisée basée sur des jetons JWT (JSON Web Token). Cette approche présente plusieurs avantages pour un projet comme La Conciergerie Auto :

**Sécurisation des sessions utilisateur** : Lorsqu'un utilisateur se connecte via l'endpoint `/api/login_check`, un jeton JWT est généré et envoyé au client. Ce jeton, signé avec une clé secrète asymétrique (paire de clés privée/publique), est ensuite utilisé pour toutes les requêtes suivantes. Le frontend Vue.js stocke ce token dans le localStorage et l'ajoute automatiquement dans le header `Authorization: Bearer {token}` de chaque requête via l'intercepteur Axios configuré dans `api.ts`.

**Confidentialité et sécurité** : Le JWT contient les informations essentielles concernant l'utilisateur (ID, email, rôles), encodées et signées. Cette signature garantit qu'aucune information ne peut être modifiée sans être détectée. Si quelqu'un tente de modifier le contenu du token (par exemple pour s'attribuer le rôle ROLE_ADMIN), la signature ne correspondra plus et Symfony rejettera automatiquement le token. Ce système réduit considérablement le risque de session hijacking et offre une sécurité renforcée pour les opérations sensibles comme la création de transactions ou l'upload de documents d'identité.

**Gestion de l'expiration** : Les tokens JWT ont une durée de vie limitée. Lorsqu'un token expire, l'intercepteur Axios côté frontend détecte l'erreur 401 et déconnecte automatiquement l'utilisateur, le redirigeant vers la page de login. Cette approche garantit qu'un token volé ne peut être utilisé indéfiniment.

**Vérification du compte** : J'ai implémenté un UserChecker personnalisé qui vérifie, avant d'émettre un token JWT, que le compte utilisateur est bien vérifié ( `isVerified: true` ). Cela empêche les utilisateurs qui n'ont pas cliqué sur le lien de vérification dans leur email de se connecter, ajoutant une couche supplémentaire de sécurité contre les inscriptions frauduleuses.

**Expérience utilisateur et performance** : Le JWT permet une expérience de navigation fluide pour les utilisateurs connectés tout en maintenant un niveau de sécurité élevé. En stockant le jeton côté client, le serveur n'a pas besoin de gérer des sessions utilisateur côté serveur, ce qui réduit la charge et améliore les performances, particulièrement important pour une API qui peut gérer de nombreux utilisateurs simultanés.

# Protection des documents privés

Un aspect crucial de la sécurité de La Conciergerie Auto concerne les documents sensibles (cartes d'identité, documents d'assurance) stockés sur AWS S3. J'ai implémenté un système de dossiers séparés :

**Dossier public** ( `public/vehicles/` ) : Contient les photos de véhicules accessibles publiquement via des URLs directes.

**Dossier privé** ( `private/identity/` , `private/insurance/` ) : Contient les documents sensibles qui ne doivent être accessibles qu'aux utilisateurs autorisés.

Pour accéder aux documents privés, j'ai créé un endpoint admin ( `/api/admin/documents/view` ) qui génère des URLs S3 pré-signées avec une validité de 10 minutes. Ces URLs temporaires permettent de visualiser un document sans exposer de façon permanente des informations sensibles. Seuls les

administrateurs peuvent générer ces URLs, garantissant un contrôle strict sur l'accès aux documents d'identité.

# Conclusion

Grâce au système de rôles hiérarchiques de Symfony, au contrôle d'accès granulaire, au bundle LexikJWTAuthenticationBundle pour l'authentification, et à la gestion sécurisée des fichiers sur AWS S3, l'application La Conciergerie Auto bénéficie d'une sécurité solide et multi-couches. Ces éléments assurent que les accès aux données sensibles (informations personnelles, documents d'identité, détails de transactions) et aux fonctionnalités critiques de l'application sont strictement contrôlés et limités aux utilisateurs authentifiés et autorisés selon leur rôle, offrant ainsi une expérience sécurisée et fiable tant pour les clients que pour les concierges et administrateurs.

# 9. Design & Maquette

## 9.1 Design

Le design de La Conciergerie Auto a été conçu pour offrir une expérience utilisateur moderne et distinctive dans le secteur automobile, où l'interface premium renforce la confiance et la crédibilité de la plateforme. Ce choix esthétique se reflète dans les couleurs et la typographie soigneusement sélectionnées, qui contribuent à une interface élégante et professionnelle pour les utilisateurs.

Pour La Conciergerie Auto, j'ai opté pour un thème dark mode premium avec une palette de couleurs audacieuse. Le noir profond (#000000) sert de couleur de fond principale, créant une toile élégante qui met en valeur le contenu. Le vert néon (#9dff80) est utilisé comme couleur primaire pour les éléments interactifs (boutons, liens, prix), créant un contraste fort et moderne qui attire l'œil sans agresser. Cette combinaison noir/vert néon évoque à la fois la technologie moderne et l'écologie, des valeurs importantes dans le secteur automobile contemporain. Des nuances de gris (#0a0a0a, #121212, #1e1e1e) sont utilisées pour créer de la profondeur et hiérarchiser les informations, tandis qu'un vert secondaire plus doux (#cfffea) apporte de la variation dans les accents visuels.

Pour la typographie, j'ai choisi d'utiliser la police système par défaut ( `-apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Helvetica', 'Arial', sans-serif` ). Ce choix offre plusieurs avantages : une lisibilité optimale sur tous les appareils, des performances de chargement excellentes (pas de téléchargement de police externe), et une intégration native qui respecte les préférences système de l'utilisateur. Cette approche s'inscrit également dans une démarche d'écoconception en réduisant le poids de l'application.

L'harmonie entre les choix de couleur et de typographie crée un environnement accueillant et professionnel pour les utilisateurs. Ce design dark mode premium, axé sur des contrastes forts et des accents de couleur stratégiques, favorise une navigation fluide et réduit la fatigue visuelle, particulièrement important pour une plateforme où les utilisateurs peuvent passer du temps à consulter de nombreuses annonces de véhicules. Le thème sombre est également apprécié pour la consultation en soirée, une période où de nombreux utilisateurs recherchent des véhicules.

# 9.2 Maquette

Avant de concevoir l'interface de La Conciergerie Auto, j'ai commencé par définir des user stories (**Voir annexe**)détaillant les besoins et attentes spécifiques des trois types d'utilisateurs : les clients cherchant à vendre ou acheter un véhicule, les concierges gérant les estimations et véhicules, et les administrateurs supervisant la plateforme. Ces histoires ont permis de prioriser les fonctionnalités essentielles et de comprendre comment chaque interaction devait se dérouler selon le rôle de l'utilisateur.

Pour garantir un design responsive et adapté aux usages réels, j'ai adopté une approche mobile-first en concevant des maquettes pour deux formats principaux : mobile (< 768px) et desktop (≥ 1024px). Cette méthodologie reflète la réalité d'usage du secteur automobile, où plus de 60% des consultations d'annonces se font depuis un smartphone. Cela m'a permis de m'assurer que chaque élément reste accessible et intuitif, peu importe la taille de l'écran, avec des adaptations spécifiques comme le menu hamburger sur mobile et une navigation horizontale sur desktop.

L'organisation visuelle de La Conciergerie Auto vise à assurer une navigation fluide et adaptée au rôle de chaque utilisateur. La cohérence visuelle est maintenue à travers toutes les pages grâce à un header fixe contenant le logo, la navigation principale et les actions utilisateur, une structure de page familière avec sidebar sur desktop pour les filtres, et des composants réutilisables (cards de véhicules, badges de statut, formulaires) qui assurent une harmonie graphique.

Sur mobile, le header se simplifie pour afficher uniquement le logo et un bouton menu hamburger qui déploie un menu latéral (slide-in) donnant accès à toutes les fonctionnalités. Les filtres de recherche, affichés horizontalement sur desktop, sont regroupés dans un drawer (tiroir) qui s'ouvre depuis le bas de l'écran sur mobile, optimisant ainsi l'espace disponible. Ce design responsive réduit le nombre de clics nécessaires pour l'utilisateur et adapte intelligemment la densité d'information selon la taille d'écran, facilitant ainsi une navigation intuitive et agréable sur tous les appareils.

# Page de détail d'un véhicule

La page de détail d'un véhicule est l'une des pages les plus importantes de La Conciergerie Auto. Elle regroupe toutes les informations pertinentes concernant un véhicule spécifique : galerie photos haute résolution, informations principales (marque, modèle, année, kilométrage, prix), caractéristiques techniques (motorisation, carburant, transmission, options), description détaillée, et informations sur le vendeur et le concierge responsable.

Cette page a été conçue avec une mise en page en deux colonnes sur desktop : la galerie photo occupe les deux tiers de l'écran à gauche avec une image principale et des thumbnails cliquables, tandis que les informations essentielles et le call-to-action (contacter, prendre rendez-vous) sont regroupés dans un panneau fixe à droite. Sur mobile, ces éléments sont empilés verticalement avec la galerie en premier pour maximiser l'impact visuel immédiat.

L'intégration de fonctionnalités interactives comme la galerie photo avec zoom, les badges de statut dynamiques (Disponible/Réservé/Vendu), et les boutons d'action contextuels selon le statut du véhicule, ajoute un niveau d'interaction qui enrichit l'expérience utilisateur et renforce l'engagement sur la plateforme. Les caractéristiques techniques sont présentées dans des sections dépliables pour ne pas surcharger la page tout en rendant l'information accessible. Ce design a été pensé pour s'adapter naturellement aux mobiles tout en garantissant une structure élégante et fonctionnelle sur les écrans d'ordinateur, avec une attention particulière portée à la hiérarchie visuelle qui guide naturellement l'œil vers les informations les plus importantes.

# 10. Front-end

## Technologie

Pour développer le frontend de La Conciergerie Auto, j'ai sélectionné un ensemble de technologies modernes, alliant Vue.js, TypeScript, et CSS personnalisé, complétées par des bibliothèques spécifiques comme Pinia, Vue Router et Axios. Ces choix ont permis de créer une application performante, maintenable et agréable à utiliser, adaptée aux besoins spécifiques d'une plateforme d'intermédiation automobile.

## Vue.js

Vue.js 3 avec la Composition API forme le cœur de l'interface de La Conciergerie Auto, grâce à sa flexibilité et à sa capacité à organiser les composants de manière modulaire et réactive. Ce choix a permis une structuration claire de l'interface, chaque fonctionnalité clé étant développée sous forme de composants indépendants et réutilisables : `VehicleCard` pour l'affichage des véhicules dans le catalogue, `VehiclePhotoGallery` pour les galeries d'images interactives, `VehicleFilters` pour le système de filtrage et `MobileMenu` pour la navigation mobile.

Pour renforcer cette modularité, j'ai également intégré Vue Router pour gérer les transitions entre les différentes pages de l'application et implémenter un système de navigation guards (gardes de navigation) qui contrôlent l'accès aux routes selon le rôle de l'utilisateur. Vue Router permet d'offrir une expérience de navigation fluide et intuitive, facilitant le passage de l'accueil au catalogue de véhicules, de la création d'estimation aux détails d'un véhicule, tout en gérant intelligemment les redirections selon le statut d'authentification (utilisateurs non connectés redirigés vers la page de login, utilisateurs connectés depuis login/register redirigés vers leur dashboard).

Les routes sont organisées de manière hiérarchique avec des routes publiques (home, catalogue, détail véhicule, login/register), des routes client (création d'estimation, mes estimations, profil, rendez-vous), des routes concierge (pool d'estimations, gestion véhicules, transactions, agenda), et des routes admin (gestion utilisateurs, véhicules, estimations, transactions) accessibles uniquement depuis desktop.

# TypeScript

Pour accroître la robustesse et la lisibilité du code, j'ai choisi TypeScript comme langage principal pour le frontend. TypeScript permet un typage statique qui aide à détecter les erreurs en amont, réduisant les bugs et améliorant considérablement la qualité du code. Avec un projet comme La Conciergerie Auto, où la gestion de données complexes (véhicules avec leurs relations modèle/marque/photos, estimations avec leurs statuts, transactions avec documents) est cruciale, TypeScript garantit que les composants respectent des structures précises.

J'ai créé un fichier d'interfaces centralisé ( `src/shared/interfaces/index.ts` ) qui définit tous les types de données de l'application : `User` , `Vehicle` , `ProfilzUser` , `Transaction` , `Appointment` , ainsi que leurs types associés ( `VehicleStatus` , `EstimationStatus` , `TransactionStatus` ). Ces interfaces servent de contrat entre le frontend et le backend, assurant que les données échangées via l'API respectent toujours la structure attendue. TypeScript détecte automatiquement si un composant tente d'accéder à une propriété inexistante ou de type incorrect, évitant ainsi de nombreuses erreurs runtime.

```ts
export interface User {
  id: number;
  email: string;
  roles: string[];
  firstName: string;
  lastName: string;
  isVerified: boolean;
  verificationToken?: string;
  profile?: ProfileUser;
  createdAt: string;
  updatedAt: string;
}

export interface ProfileUser {
  id: number;
  userId: number;
  address?: string;
  postCode?: string;
  city?: string;
  phone?: string;
  bithdate?: string;
  identityRectroPath?: string;
  identityVersoPath?: string;
  updatedAt?: string;
}
```

# CSS personnalisé

Pour gérer les styles de La Conciergerie Auto, j'ai opté pour du CSS personnalisé plutôt qu'un framework CSS comme Bootstrap ou Tailwind. Ce choix s'inscrit dans une démarche d'écoconception et de personnalisation totale du design. J'ai créé un fichier `style.css` centralisé qui définit l'ensemble du système de design via des variables CSS (custom properties) :

```
:root {
  --color-bg: #000000;
  --color-bg-secondary: #0a0a0a;
  --color-primary: #9dff80;
  --radius-md: 12px;
  --shadow: 0 4px 16px rgba(0, 0, 0, 0.4);
}
```

Cette approche me permet de centraliser les variables de couleurs, les espacements, les rayons de bordure et les ombres, facilitant ainsi la personnalisation et assurant une cohérence visuelle sur toutes les pages de La Conciergerie Auto. Le noir profond et le vert néon, choisis pour créer une identité visuelle distinctive dans le secteur automobile, sont gérés directement dans ces variables, permettant des ajustements globaux rapides si nécessaire.

J'ai également développé un système de classes utilitaires réutilisables ( `.btn` , `.btn-primary` , `.card` , `.form-group` , `.badge` , `.grid` ) qui accélèrent le développement tout en maintenant la cohérence. Ces classes sont utilisées dans tous les composants Vue, évitant la duplication de code CSS. L'utilisation de media queries permet une adaptation responsive fluide avec des breakpoints clairement définis (mobile < 768px, et desktop ≥ 1024px).

Cette approche CSS personnalisée permet également de réduire considérablement le poids de l'application par rapport à l'utilisation d'un framework complet, ne chargeant que les styles réellement utilisés. Le fichier CSS final est minifié en production pour des performances optimales.

# Axios

Pour la communication avec l'API backend Symfony, j'ai intégré Axios comme client HTTP. J'ai créé une instance Axios configurée ( `apiClient` dans `api.ts` ) qui centralise toute la logique de communication avec le backend. Cette instance ajoute automatiquement le token JWT dans le header `Authorization` de chaque requête via un intercepteur de requête, simplifiant ainsi considérablement le code dans les composants et stores qui n'ont plus à gérer manuellement l'authentification.

Un intercepteur de réponse gère également les erreurs globales, notamment la détection des erreurs 401 (token expiré) qui déclenchent automatiquement une déconnexion et une redirection vers la page de login. Cette centralisation de la gestion des erreurs améliore la robustesse de l'application et offre une expérience utilisateur cohérente en cas de problème de connexion ou d'authentification.

```
// Configuration de l'API
export const API_BASE_URL =  'http://127.0.0.1:8000/api';

// Instance Axios configurée
import axios from 'axios';

export const apiClient = axios.create({
  baseURL: API_BASE_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});

// Intercepteur pour ajouter le token JWT à chaque requête
apiClient.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('jwt_token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return (error);
  }
);

// Intercepteur pour gérer les erreurs globales
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      // Token invalide ou expiré
      localStorage.removeItem('jwt_token');
      window.location.href = '/login';
    }
    return (error);
  }
);
```

# Icônes

Pour les icônes de La Conciergerie Auto, j'ai fait le choix délibéré d'utiliser des emojis natifs plutôt qu'une bibliothèque d'icônes externe comme Lucide ou Font Awesome. Ce choix s'inscrit dans une démarche d'écoconception et de simplicité technique, offrant plusieurs avantages significatifs.

Les emojis natifs ne nécessitent aucun téléchargement de bibliothèque externe, réduisant ainsi le poids du bundle JavaScript et améliorant les performances de chargement de l'application. Cette approche est cohérente avec l'objectif de sobriété numérique du projet. De plus, les emojis sont universellement reconnus et compris, facilitant la navigation intuitive pour tous les utilisateurs sans nécessiter d'apprentissage.

Les emojis sont utilisés de manière stratégique dans toute l'application : icônes de navigation dans le menu mobile (🏠 Accueil, 🚗 Véhicules, 📊 Estimations, 📅 Rendez-vous, 👤 Profil), icônes d'action dans les boutons et call-to-action (➕ Ajouter, 🔍 Filtres, 📧 Contact), et emojis descriptifs dans les sections de la page d'accueil pour illustrer les services (🤝 Assistance, 🚚 Convoyage, ⚙️ Gestion, 🛡️ Partenariats).

Cette approche minimaliste présente l'avantage d'être native au système d'exploitation, garantissant un affichage cohérent et optimisé sur tous les appareils sans nécessiter de chargement supplémentaire. Les emojis s'adaptent automatiquement au thème système et à la taille de police définie, assurant une accessibilité optimale.

# Gestion des Formulaires et des États

L'expérience utilisateur étant au cœur de La Conciergerie Auto, j'ai accordé une attention particulière à la gestion des formulaires, particulièrement critiques pour l'inscription, la connexion, la création d'estimations et l'ajout de véhicules avec upload de photos. J'ai implémenté une validation côté client avec des messages d'erreur clairs et contextuels, tout en conservant une validation serveur pour la sécurité. Les formulaires utilisent les directives Vue natives ( `v-model` , `@submit.prevent` ) et affichent les erreurs de l'API de manière compréhensible pour l'utilisateur.

Pour gérer les données d'état centralisées de l'application, Pinia s'est avéré être un choix optimal. Ce store moderne et léger, successeur officiel de Vuex pour Vue 3, gère de manière réactive et efficace les informations critiques de l'application. J'ai créé plusieurs stores spécialisés :

**authStore** : Gère l'authentification complète (login, register, logout, vérification email), stocke les informations de l'utilisateur connecté et son token JWT, et expose des getters pour vérifier rapidement les rôles (isClient, isConcierge, isAdmin). Ce store initialise automatiquement l'utilisateur au chargement de l'application si un token valide existe dans le localStorage.

**vehicleStore** (potentiel) : Pourrait gérer le cache des véhicules consultés, les filtres actifs, et les véhicules favoris pour améliorer les performances et réduire les appels API redondants.

En centralisant ces données dans des stores Pinia, je permets une synchronisation fluide des informations entre les différents composants de l'application sans avoir besoin de prop drilling (passage de props à travers de nombreux niveaux de composants). Par exemple, le composant `Header` peut accéder directement aux informations utilisateur via `authStore` pour afficher le menu approprié selon le rôle, et le composant `MobileMenu` peut utiliser les mêmes données sans duplication.

# Composables

J'ai également créé des composables personnalisés qui encapsulent de la logique réutilisable. Le plus notable est `useResponsive` qui détecte en temps réel la taille de l'écran et expose des variables réactives ( `isMobile` , `isTablet` , `isDesktop` ) utilisées dans les composants pour adapter le comportement et l'affichage. Ce composable est par exemple utilisé dans `VehicleListView` pour décider d'afficher les filtres horizontalement (desktop) ou dans un drawer (mobile), et dans `AppLayout` pour gérer l'affichage du menu hamburger.

# Conclusion

Ce choix technologique centré autour de Vue.js 3 avec Composition API, TypeScript, CSS personnalisé, et des bibliothèques ciblées comme Pinia, Vue Router, Axios et Lucide, a permis de construire une interface front-end robuste, performante, évolutive et intuitive pour La Conciergerie Auto. Ces outils modernes assurent une organisation du code efficace et maintenable, une expérience de développement agréable grâce au typage TypeScript, et une expérience utilisateur de haute qualité avec des performances optimisées.

L'architecture modulaire basée sur des composants réutilisables, combinée à une gestion d'état centralisée via Pinia et une communication API simplifiée avec Axios, facilite grandement les évolutions futures de la plateforme. L'approche responsive mobile-first garantit que La Conciergerie Auto offre une expérience optimale sur tous les appareils, un critère essentiel dans le secteur automobile où la majorité des consultations se font depuis un smartphone.

# 11. Contenu et Interactivité

Dans mon projet La Conciergerie Auto, j'ai veillé à appliquer les bonnes pratiques pour garantir une accessibilité optimale, un référencement efficace et une structure de code maintenable. J'ai pris soin de créer une hiérarchie HTML cohérente et bien organisée en utilisant des balises sémantiques qui facilitent la compréhension de la structure du contenu pour les moteurs de recherche et les technologies d'assistance. De plus, en optimisant le contraste des couleurs du thème dark mode (vert néon #9dff80 sur fond noir #000000 avec un ratio de contraste élevé) et en intégrant des attributs alt descriptifs pour toutes les images de véhicules, j'ai amélioré l'accessibilité pour les utilisateurs de lecteurs d'écran. J'ai également respecté les standards de développement web modernes pour assurer une expérience utilisateur fluide et accessible sur tous les appareils, du smartphone à l'écran desktop.

Pour illustrer la mise en pratique de ces bonnes pratiques, je vais maintenant me concentrer sur la page de liste des véhicules ( `VehicleListView.vue` ), qui est l'un des éléments centraux de l'application La Conciergerie Auto. Cette page représente bien l'attention que j'ai portée à l'organisation, l'accessibilité, la structuration du contenu et la gestion des états dans le projet. En intégrant des fonctionnalités avancées de recherche et de filtrage, une gestion responsive intelligente avec drawer mobile, et un code structuré avec Vue.js et TypeScript, elle constitue un excellent exemple de la manière dont j'ai appliqué ces principes au sein de mon application.

# Structure et hiérarchie du contenu

En haut de la page, un titre clair et descriptif "Nos véhicules disponibles" est défini avec une balise `<h2>` . Cela permet de structurer le contenu de manière cohérente pour les moteurs de recherche et les utilisateurs de lecteurs d'écran, qui comprennent immédiatement l'objet de la page. Juste en dessous, un sous-titre avec la classe `.text-muted` précise "Découvrez notre sélection de véhicules inspectés et garantis", renforçant le message de confiance et de qualité qui est au cœur de la proposition de valeur de La Conciergerie Auto.

# Système de filtres adaptatif

L'un des aspects les plus sophistiqués de cette page est son système de filtres qui s'adapte intelligemment selon la taille de l'écran. Sur desktop, les filtres sont affichés horizontalement dans une barre dédiée ( `.filters-horizontal` ) permettant une sélection rapide par marque, modèle, prix

maximum, kilométrage et année minimum. Chaque select est accompagné d'un label descriptif qui améliore l'accessibilité et guide l'utilisateur dans ses choix.

Sur mobile, pour optimiser l'espace limité de l'écran, j'ai implémenté un système de drawer (tiroir) qui s'ouvre depuis le bas de l'écran. Un bouton "🔍 Filtres" permet d'afficher ce drawer via un état réactif ( `showFilters` ). Le composable `useResponsive` détecte automatiquement la taille de l'écran et adapte l'affichage en conséquence grâce à la variable réactive `isMobile` . Cette approche responsive offre une expérience utilisateur optimale sur chaque type d'appareil, en rendant les filtres facilement accessibles sans encombrer l'interface mobile.

Le drawer mobile utilise une transition Vue.js ( `<Transition name="slide-up">` ) qui anime l'apparition depuis le bas de l'écran, créant une expérience fluide et moderne. Un overlay semi-transparent permet de fermer le drawer en cliquant en dehors, suivant les conventions d'interface utilisateur actuelles. Les filtres sélectionnés sont appliqués en temps réel grâce au système de réactivité de Vue.js, sans nécessiter de rechargement de page.

## Gestion des états et retours visuels

La page implémente une gestion d'états complète pour offrir un retour visuel approprié dans toutes les situations. Trois états principaux sont gérés via des variables réactives :

**État de chargement** : Lorsque les véhicules sont en cours de récupération depuis l'API ( `loading.value = true` ), un spinner animé s'affiche accompagné du message "Chargement des véhicules...". Cet indicateur visuel informe l'utilisateur que l'application traite sa requête et améliore la perception de la performance.

**État d'erreur** : En cas de problème de connexion à l'API ou d'erreur serveur, un message d'erreur s'affiche dynamiquement dans une section dédiée grâce à une condition `v-if="error"` . Le message explique clairement le problème et propose un bouton "Réessayer" qui relance la requête API. Cette communication claire avec l'utilisateur améliore la fiabilité perçue de l'application et offre une solution immédiate plutôt que de laisser l'utilisateur face à une page blanche.

**État vide** : Lorsque la recherche ou les filtres appliqués ne correspondent à aucun véhicule ( `filteredVehicles.length === 0` ), un état vide personnalisé s'affiche avec une grande icône de véhicule emoji (🚗), un titre "Aucun véhicule trouvé", un message explicatif "Essayez de modifier vos filtres", et un bouton pour réinitialiser tous les filtres. Ce retour visuel évite la confusion et guide l'utilisateur vers une action concrète pour résoudre le problème.

## Grille de véhicules et composants

La grille des véhicules constitue le cœur visuel de cette page. La structure HTML comprend une `div` avec la classe `.vehicles-grid` qui dispose chaque véhicule sous forme de cartes individuelles générées dynamiquement avec `v-for` . L'organisation en grille CSS (grid-template-columns) permet de maintenir une présentation nette et ordonnée, avec 3 colonnes sur desktop, 2 sur tablette, et 1 seule colonne sur mobile grâce aux media queries définies dans le CSS.

Chaque véhicule est rendu via le composant réutilisable `VehicleCard` qui reçoit les données du véhicule en props. Ce composant affiche la photo principale (ou un placeholder si aucune photo), le titre (marque + modèle), les informations essentielles (année, kilométrage formaté), le prix formaté en euros, une description tronquée, et un badge de statut si le véhicule est réservé ou vendu. L'image est accompagnée d'un attribut `alt` descriptif généré dynamiquement (par exemple : "Tesla Model 3") pour offrir une description textuelle essentielle pour l'accessibilité et le référencement SEO.

En termes de style, j'ai organisé le CSS de manière modulaire avec des classes spécifiques pour chaque élément : `.vehicle-title` pour le titre, `.vehicle-price` pour le prix mis en évidence en vert

néon, `.vehicle-info` pour les métadonnées, et `.vehicle-features` pour les caractéristiques. Cette structure favorise une hiérarchie visuelle intuitive pour l'utilisateur et facilite la maintenance du code.

# Approche mobile-first

J'ai adopté une approche mobile-first afin de garantir une expérience utilisateur fluide et optimisée sur tous les appareils, en commençant par les écrans de petite taille. Dans ce contexte, le design et la structure de la grille de véhicules ont d'abord été pensés pour offrir une navigation simple et claire sur mobile, avec une mise en page compacte et des informations essentielles hiérarchisées. Les utilisateurs peuvent facilement parcourir les véhicules en scrollant, sans être surchargés d'éléments inutiles.

Cette conception initiale permet d'ajouter des styles et des éléments supplémentaires progressivement à mesure que la taille de l'écran augmente, garantissant ainsi une expérience enrichie sur tablettes et ordinateurs. Par exemple, sur desktop, la grille passe à 3 colonnes et les filtres sont affichés horizontalement pour une sélection plus rapide. Grâce à cette approche, la page reste rapide à charger et ergonomique, optimisant la lisibilité et l'accessibilité sur chaque appareil.

# Navigation et interactivité

Chaque carte de véhicule est enveloppée dans un `router-link` qui redirige vers la page de détail du véhicule au clic. Cette approche native de Vue Router offre plusieurs avantages : navigation instantanée sans rechargement de page (SPA), gestion automatique de l'état actif pour le style, et amélioration du référencement grâce aux vraies balises `<a>` avec attributs `href` . Le bouton "Voir le détail" guide les utilisateurs de manière naturelle et leur offre une navigation fluide vers plus d'informations.

# Filtrage réactif côté client

Un aspect important de cette page est le système de filtrage réactif implémenté via une propriété calculée Vue.js ( `filteredVehicles` ). Cette propriété observe les changements dans l'objet `filters` et applique automatiquement les critères sélectionnés (marque, modèle, prix, kilométrage, année) sur le tableau de véhicules chargés depuis l'API. Cette approche offre un filtrage instantané sans appel API supplémentaire, améliorant considérablement la réactivité de l'interface.

Lorsque l'utilisateur sélectionne un filtre de marque, la liste des modèles disponibles est automatiquement mise à jour via une autre propriété calculée ( `availableModels` ) qui ne montre que les modèles correspondant à la marque sélectionnée. Cette logique en cascade améliore l'expérience utilisateur en évitant les sélections impossibles.

## Section Call-to-Action

En bas de page, une section CTA (Call-to-Action) encourage les utilisateurs qui n'ont pas trouvé leur véhicule idéal à contacter l'équipe. Cette section, stylisée comme une card mise en évidence, contient un titre accrocheur "Vous n'avez pas trouvé le véhicule idéal ?", un message rassurant "Contactez-nous, nous vous aiderons à le trouver !", et un bouton d'action. Cette approche proactive transforme une potentielle frustration (ne pas trouver le véhicule souhaité) en opportunité de contact et de service personnalisé, renforçant la proposition de valeur du "concierge" qui accompagne l'utilisateur.

# Conclusion sur l'interactivité

Chaque élément de cette page a donc été pensé pour optimiser l'accessibilité, le référencement, et la fluidité d'interaction. Cette page de liste des véhicules est un exemple clé de la manière dont j'ai structuré l'application La Conciergerie Auto pour fournir une expérience utilisateur intuitive, inclusive et performante, en tenant compte des spécificités de chaque taille d'écran et en offrant des retours visuels appropriés pour chaque état de l'application.

# 12. Sécurité

Dans le cadre de La Conciergerie Auto, j'ai mis en place une stratégie de sécurité rigoureuse côté frontend pour protéger l'accès aux routes sensibles de l'application, en particulier celles destinées aux concierges et aux administrateurs. Pour cela, j'ai intégré des navigation guards dans Vue Router qui contrôlent à la fois l'authentification et les autorisations de rôle des utilisateurs lors de chaque changement de route. Ce système de guards, implémenté dans le fichier `router/index.ts`, repose sur plusieurs vérifications essentielles : l'état d'authentification, les rôles requis, et des règles spécifiques comme l'accès desktop-only pour l'administration.

## Navigation Guards et contrôle d'accès

Le guard principal utilise la méthode `router.beforeEach()` qui s'exécute avant chaque navigation. Il commence par initialiser le store d'authentification si nécessaire en appelant `authStore.initialize()`, qui tente de récupérer les informations de l'utilisateur depuis le token JWT stocké dans le localStorage. Cette initialisation garantit que l'état d'authentification est toujours à jour, même après un rafraîchissement de page.

**Vérification de l'authentification** : Si un utilisateur tente d'accéder à une route nécessitant une authentification (spécifiée par `to.meta.requiresAuth`) sans être connecté, il est automatiquement redirigé vers la page de connexion. Le système conserve l'URL de destination dans les paramètres de requête (`query: { redirect: to.fullPath }`), permettant de rediriger l'utilisateur vers sa destination initiale après une connexion réussie. Cette fonctionnalité améliore considérablement l'expérience utilisateur en évitant la frustration de devoir retrouver la page souhaitée après connexion.

**Vérification des rôles** : Une fois authentifié, le guard vérifie si la route nécessite un rôle spécifique via `to.meta.roles`. Par exemple, les routes du concierge nécessitent `ROLE_CONCIERGE`, et les routes admin nécessitent `ROLE_ADMIN`. Le système vérifie que l'utilisateur dispose d'au moins un des rôles requis via `authStore.user?.roles.includes(role)`. Si l'utilisateur ne dispose pas du rôle approprié, il est redirigé vers sa page d'accueil par défaut selon son rôle actuel, déterminée par la fonction `getDefaultRoute()`. Cette fonction retourne 'admin-dashboard' pour les administrateurs, 'concierge-dashboard' pour les concierges, 'client-dashboard' pour les clients, et 'home' pour les visiteurs non authentifiés.

```
// Navigation Guards
router.beforeEach(async (to, from, next) => {
  const authStore = useAuthStore();

  // Initialiser le store si ce n'est pas déjà fait
  if (!authStore.user && authStore.token) {
    try {
      await authStore.initialize();
    } catch (error) {
      console.error('Erreur lors de l\'initialisation:', error);
    }
  }

  // Vérifier si la route nécessite une authentification
  const requiresAuth = to.matched.some((record) => record.meta.requiresAuth);

  // Rediriger les utilisateurs authentifiés depuis login/register
  if (to.meta.hideForAuth && authStore.isAuthenticated) {
    return next({ name: getDefaultRoute(authStore) });
  }

  // Vérifier l'authentification
  if (requiresAuth && !authStore.isAuthenticated) {
    return next({ name: 'login', query: { redirect: to.fullPath } });
  }

  // Vérifier les rôles requis
  const requiredRoles = to.meta.roles as string[] | undefined;
  if (requiredRoles && authStore.user) {
    const hasRole = requiredRoles.some((role) =>
      authStore.user?.roles.includes(role)
    );
    if (!hasRole) {
      return next({ name: getDefaultRoute(authStore) });
    }
  }

  next();
});
```

**Redirection des utilisateurs authentifiés** : Le système implémente également une logique de redirection pour éviter que les utilisateurs déjà connectés n'accèdent aux pages de login ou register. Si la route a la métadonnée `hideForAuth` (comme login et register) et que l'utilisateur est authentifié, il est redirigé vers sa page d'accueil par défaut. Cette approche améliore la cohérence de l'expérience utilisateur et évite les confusions.

## Exemple de configuration des routes

Chaque route de l'application définit ses exigences de sécurité via l'objet `meta` :

```
// Route publique
{ path: '/vehicles', name: 'vehicles', meta: { requiresAuth: false } }

// Route client
{ path: '/client/estimations/new', meta: { requiresAuth: true, roles: ['ROLE_CLIENT'] } }

// Route concierge
{ path: '/concierge/vehicles/new', meta: { requiresAuth: true, roles: ['ROLE_CONCIERGE'] } }

// Route admin (desktop uniquement)
{ path: '/admin/users', meta: { requiresAuth: true, roles: ['ROLE_ADMIN'], desktopOnly: true } }
```

Cette configuration déclarative rend le système de sécurité transparent et facile à maintenir. Chaque développeur peut comprendre immédiatement les exigences d'accès d'une route en consultant sa définition.

## Gestion centralisée de l'authentification avec Pinia

Le store `authStore` (Pinia) centralise toute la logique d'authentification et expose des getters pour vérifier facilement les rôles :

- `isAuthenticated` : Vérifie si un token et un utilisateur sont présents
- `isAdmin` : Vérifie si l'utilisateur a le rôle ROLE_ADMIN
- `isConcierge` : Vérifie si l'utilisateur a le rôle ROLE_CONCIERGE
- `isClient` : Vérifie si l'utilisateur a le rôle ROLE_CLIENT

Ces getters sont utilisés dans les guards de navigation et dans les composants pour afficher conditionnellement certains éléments d'interface. Par exemple, le composant `MobileMenu` utilise ces getters pour afficher le menu approprié selon le rôle de l'utilisateur connecté.

# Validation des formulaires

La sécurité des données transmises par l'utilisateur est également primordiale côté frontend, bien que la validation définitive soit toujours effectuée côté serveur. J'ai implémenté une validation en deux temps :

**Validation HTML5 native** : Les champs de formulaire utilisent les attributs HTML5 standard ( `required` , `type="email"` , `min` , `max` , `pattern` ) pour une première couche de validation immédiate et accessible. Par exemple, le formulaire d'inscription utilise `type="email"` pour valider automatiquement le format de l'email, et `required` pour empêcher la soumission de champs vides.

**Validation personnalisée côté client** : Pour les formulaires complexes comme la création d'estimation ou l'ajout de véhicule, j'ai implémenté une validation personnalisée en JavaScript qui vérifie la cohérence des données avant l'envoi. Par exemple, lors de l'upload de photos de véhicule, le système vérifie le type MIME (image/jpeg, image/png), la taille maximale du fichier, et le nombre maximum de photos autorisées. Les messages d'erreur sont affichés de manière claire et contextuelle dans l'interface.

**Gestion des erreurs API** : Lorsque le backend Symfony retourne des erreurs de validation (statut 422 avec un objet d'erreurs), le frontend les extrait et les affiche à côté des champs concernés. Cette approche garantit que l'utilisateur comprend précisément ce qui doit être corrigé sans avoir à deviner. L'objet d'erreur retourné par l'API suit un format standardisé qui facilite cette intégration.

# Protection contre les actions non autorisées

Au-delà des guards de navigation, certains composants implémentent des vérifications supplémentaires. Par exemple, dans la liste des véhicules du concierge, les boutons d'édition et de suppression ne s'affichent que pour les véhicules dont le concierge est responsable (vérification de `vehicle.conciergeId === authStore.user.id` ). Cette double vérification (frontend + backend) améliore l'expérience utilisateur en masquant les options non disponibles tout en maintenant la sécurité grâce à la validation serveur.

## Gestion sécurisée du token JWT

Le token JWT est stocké dans le localStorage du navigateur pour persister entre les sessions. L'intercepteur Axios configuré dans `api.ts` ajoute automatiquement ce token dans le header `Authorization: Bearer {token}` de chaque requête API. En cas d'erreur 401 (token expiré ou invalide), l'intercepteur de réponse déclenche automatiquement la déconnexion en appelant

`authStore.logout()` , nettoyant le localStorage et redirigeant vers la page de login. Cette gestion automatique évite que l'utilisateur ne reste bloqué dans un état incohérent avec un token invalide.

## Conclusion sur la sécurité frontend

Grâce aux navigation guards de Vue Router, à la gestion centralisée de l'authentification via Pinia, et à la validation multi-niveaux des formulaires, l'application La Conciergerie Auto bénéficie d'un niveau de sécurité frontend élevé. Ces mécanismes renforcent la protection des routes sensibles de manière proactive, améliorent l'expérience utilisateur en guidant clairement les accès autorisés, et minimisent les risques d'actions non autorisées. Cette sécurité côté client, combinée avec les contrôles rigoureux côté backend Symfony (JWT, rôles hiérarchiques, validation des données, contrôle d'accès aux ressources), offre une défense en profondeur qui protège efficacement les données sensibles des utilisateurs et l'intégrité de la plateforme.

# 13. Road-map

## 13.1 API

L'API de La Conciergerie Auto va évoluer pour offrir des fonctionnalités enrichies, répondant aux attentes des clients, concierges et de la plateforme dans son ensemble. Dans les prochaines itérations, l'API intégrera un système de messagerie interne permettant aux clients et concierges de communiquer directement au sein de la plateforme. Cela nécessitera de créer des endpoints dédiés pour gérer l'envoi, la réception, la lecture et l'archivage des messages, avec un système de notifications en temps réel pour alerter les utilisateurs des nouveaux messages.

Un autre objectif majeur sera d'implémenter un système de notation et d'avis sur les concierges. Les clients ayant finalisé une transaction pourront évaluer leur expérience avec le concierge qui les a accompagnés, en attribuant une note sur 5 étoiles et en laissant un commentaire détaillé. Ces évaluations seront visibles publiquement sur les profils des concierges, renforçant la transparence et la confiance au sein de la plateforme. L'API devra gérer la création, la modération et l'affichage de ces avis, tout en calculant automatiquement la note moyenne de chaque concierge.

En complément, l'API sera enrichie d'un système d'historique de maintenance des véhicules. Cette fonctionnalité permettra aux vendeurs et concierges de documenter l'historique d'entretien d'un véhicule (révisions, changements de pièces, contrôles techniques), ajoutant une valeur significative aux annonces et renforçant la confiance des acheteurs potentiels. Les documents justificatifs (factures, rapports) pourront être uploadés sur S3 et liés à chaque entrée d'historique.

Une autre amélioration prévue concerne la géolocalisation et la recherche géographique. L'ajout d'une fonctionnalité permettant de filtrer les véhicules par proximité géographique facilitera la recherche pour les clients souhaitant visiter physiquement les véhicules. Cela nécessitera d'enrichir l'entité Vehicle avec des coordonnées géographiques et d'implémenter des requêtes de recherche par rayon.

Enfin, l'API inclura un système de notifications push pour informer en temps réel les utilisateurs des événements importants : nouvelle estimation traitée, rendez-vous confirmé, transaction finalisée, nouveau message reçu. Ces notifications utiliseront une architecture event-driven avec Symfony Messenger pour garantir la scalabilité. Ces modifications ouvriront de nouvelles routes et nécessiteront l'ajout d'une table Notification dans la base de données pour stocker l'historique des notifications.

# 13.2 Front-end

Du côté front-end, la feuille de route met l'accent sur la création d'interfaces modernes et intuitives pour améliorer l'expérience utilisateur sur tous les rôles. Le système de messagerie interne sera intégré avec une interface de chat en temps réel développée en Vue.js, potentiellement avec WebSockets ou Server-Sent Events pour les mises à jour instantanées. Une icône de messagerie dans le header affichera le nombre de messages non lus, et une page dédiée permettra de gérer les conversations avec un design inspiré des applications de messagerie modernes.

L'interface de notation des concierges sera conçue de manière intuitive, avec un système d'étoiles cliquables et un champ de texte pour le commentaire. Les notes et avis seront affichés sur les profils des concierges avec des statistiques visuelles (graphiques, distribution des notes) et des filtres pour trier les avis par date ou pertinence. Cette transparence renforcera la crédibilité de la plateforme et encouragera les concierges à maintenir un service de qualité.

L'ajout du système d'historique de maintenance nécessitera la création d'un composant dédié affichant une timeline visuelle de l'entretien du véhicule sur la page de détail. Les utilisateurs pourront consulter les interventions passées, visualiser les documents justificatifs, et avoir une vision claire de l'état de maintenance du véhicule, un critère décisif dans leur décision d'achat.

Pour la recherche géographique, une carte interactive (via Leaflet ou Google Maps API) sera intégrée à la page de catalogue, permettant aux utilisateurs de visualiser géographiquement les véhicules disponibles et de filtrer par rayon autour d'une localisation. Cette visualisation cartographique offrira une alternative intuitive aux filtres traditionnels et facilitera la découverte de véhicules à proximité.

Enfin, un centre de notifications sera développé pour afficher l'historique complet des notifications de l'utilisateur. Une icône dans le header avec un badge numérique indiquera les notifications non lues, et un panneau déroulant permettra de consulter rapidement les dernières alertes. Les notifications seront catégorisées par type (estimations, rendez-vous, transactions) avec des icônes et couleurs distinctives pour faciliter la lecture.

Un tableau de bord amélioré pour les concierges et administrateurs intégrera des statistiques visuelles (graphiques de performance, évolution des estimations, taux de conversion), des raccourcis vers les actions fréquentes, et une vue d'ensemble de l'activité récente. Ce dashboard centralisé optimisera le workflow quotidien des concierges et facilitera la supervision pour les administrateurs.

Ces améliorations permettront de faire évoluer La Conciergerie Auto vers une plateforme plus complète et professionnelle, offrant des fonctionnalités de communication avancées et renforçant la transparence et la confiance entre tous les acteurs de la plateforme.

# 14. Veille Technologique

Dans le cadre de mon projet La Conciergerie Auto, je me suis appuyé sur une veille technologique active pour rester à jour avec les évolutions du développement web moderne et optimiser continuellement l'application. GitHub a été central pour suivre l'évolution des dépendances de mon projet, tant côté backend (Symfony, Doctrine, LexikJWTAuthenticationBundle, AWS SDK) que frontend (Vue.js, Pinia, Vue Router, Axios). La fonctionnalité Dependabot de GitHub m'a permis de recevoir des alertes automatiques sur les mises à jour de sécurité et les nouvelles versions, garantissant que l'application reste sécurisée et bénéficie des dernières améliorations.

La documentation officielle des outils utilisés a constitué une ressource fondamentale pour approfondir ma compréhension de chaque technologie. J'ai consulté régulièrement la documentation de Symfony (symfony.com/doc), particulièrement les sections sur Security, Doctrine ORM, et les bundles utilisés. Pour Vue.js, la documentation officielle (vuejs.org) et le guide sur la Composition API m'ont permis de maîtriser les patterns modernes de développement Vue 3. La documentation AWS S3 a été essentielle pour implémenter correctement l'upload et la gestion sécurisée des fichiers.

En complément, j'ai utilisé SymfonyCasts pour approfondir mon expertise en Symfony, notamment sur les sujets de sécurité avancée (JWT, voters, firewall), l'optimisation des performances avec Doctrine, et les best practices pour les API RESTful. Pour Vue.js, j'ai suivi des cours sur Vue School et consulté des articles sur Vue Mastery, deux plateformes éducatives réputées qui m'ont permis de maîtriser des techniques avancées comme la gestion d'état avec Pinia, les patterns de composition, et l'optimisation des performances des SPA. Ces ressources m'ont permis d'appliquer les meilleures pratiques dans mon code et d'éviter les pièges courants.

J'ai également suivi activement les blogs techniques et newsletters spécialisés dans le développement fullstack. Pour le backend, les articles de SymfonyCasts Blog et Symfony Station m'ont tenu informé des nouvelles fonctionnalités de Symfony et des patterns émergents. Pour le frontend, Vue.js News et des développeurs influents de la communauté Vue m'ont aidé à découvrir de nouveaux outils et techniques. La newsletter de JavaScript Weekly m'a permis de rester au courant des évolutions de l'écosystème JavaScript en général.

Pour le domaine spécifique de l'automobile et des plateformes d'intermédiation, j'ai étudié les interfaces et fonctionnalités de plateformes établies (LeBonCoin, AutoScout24, La Centrale) pour comprendre les standards du secteur, les attentes des utilisateurs, et les fonctionnalités considérées comme essentielles. Cette analyse concurrentielle m'a permis d'identifier des opportunités de différenciation et d'amélioration.

Enfin, pour anticiper les évolutions technologiques, J’assure également une veille technologique régulière en suivant quotidiennement des contenus sur LinkedIn, Daily.dev et X (Twitter), afin de rester à jour sur les dernières tendances du développement web. Ces événements m'ont permis de découvrir des fonctionnalités émergentes comme les nouveaux composants Symfony (Symfony UX, Live Components), les évolutions de Vue.js 3 et l'écosystème Vite, ainsi que les tendances en matière d'architecture API (GraphQL, API Platform). J'ai également consulté des ressources sur l'écoconception web et l'accessibilité pour garantir que La Conciergerie Auto demeure alignée avec les standards les plus récents en matière de développement durable et inclusif.

Cette veille active et diversifiée m'a permis de prendre des décisions technologiques éclairées, d'éviter les solutions obsolètes, et de garantir que La Conciergerie Auto bénéficie des meilleures pratiques actuelles tout en conservant une flexibilité pour de futures améliorations et évolutions technologiques.

# 15. Conclusion

Le développement de La Conciergerie Auto a représenté un défi technique enrichissant et formateur, permettant de concevoir une plateforme complète d'intermédiation automobile avec des fonctionnalités visant à créer un écosystème de confiance entre clients, concierges et la plateforme. En choisissant des technologies robustes et modernes comme Symfony 6.4 pour le back-end avec une architecture API RESTful, et Vue.js 3 avec TypeScript pour le front-end sous forme de SPA, j'ai pu élaborer une architecture moderne, modulable et performante, qui assure une expérience utilisateur à la fois réactive, intuitive et sécurisée sur tous les types d'appareils.

La conception d’une base de données normalisée, capable de gérer les relations complexes entre utilisateurs, véhicules, estimations, transactions et rendez-vous, a permis d’établir une fondation solide et évolutive. L’utilisation de Doctrine ORM a simplifié la manipulation de ces données tout en offrant une flexibilité suffisante pour accompagner les évolutions futures. L’intégration d’AWS S3 pour le stockage optimisé des photos et documents — incluant la conversion automatique en WebP et la gestion des accès publics/privés via des URLs pré-signées — reflète une approche professionnelle et sécurisée de la gestion des ressources.

La réflexion autour de la sécurité a également été une composante clé de ce projet, que ce soit via le système de rôles hiérarchiques de Symfony (CLIENT, CONCIERGE, ADMIN), l'authentification JWT via LexikJWTAuthenticationBundle pour sécuriser les sessions utilisateur, les navigation guards de Vue Router pour contrôler les accès aux routes frontend, ou encore la validation multi-niveaux des données (HTML5, JavaScript, Symfony Validator). Ces choix garantissent la protection des données sensibles (informations personnelles, documents d'identité, détails de transactions) et limitent les risques liés aux accès non autorisés, assurant une plateforme de confiance pour tous les utilisateurs.

La partie documentation, matérialisée par un document technique complet et une collection de requêtes API organisée dans Postman, offre une base claire pour les futurs développements, facilitant la prise en main du projet par de nouveaux développeurs ou pour une présentation professionnelle. Cette documentation exhaustive couvre l'architecture, le schéma de base de données, tous les endpoints API, les mécanismes de sécurité et l'intégration des services externes, constituant un référentiel technique complet.

En termes de design, les choix d'un thème dark mode premium avec la palette noir/vert néon, l'utilisation de polices système pour les performances, et l'ergonomie soignée répondent aux besoins d'une application moderne du secteur automobile, et l'approche mobile-first garantit une expérience fluide sur tous les appareils. Le système de design cohérent basé sur des variables CSS et des

composants réutilisables facilite la maintenance et assure une identité visuelle distinctive qui différencie La Conciergerie Auto des plateformes concurrentes.

Les fonctionnalités prévues dans la roadmap, notamment le système de messagerie interne, la notation des concierges, l'historique de maintenance des véhicules, la recherche géographique avec carte interactive, et le système de notifications push, viendront renforcer l'aspect professionnel et complet de La Conciergerie Auto en le rendant plus interactif, transparent et efficace pour tous ses utilisateurs.

Ce projet a également été l'occasion de faire preuve d'une veille technologique active et ciblée. En m'appuyant sur des ressources éducatives comme SymfonyCasts pour le backend, Vue School et Vue Mastery pour le frontend, ainsi que sur la documentation officielle de toutes les technologies, j'ai pu intégrer les meilleures pratiques actuelles et m'assurer que La Conciergerie Auto reste un projet moderne, performant et adaptable aux évolutions futures du web.

L'attention portée à l'écoconception, avec l'optimisation des images en WebP, le lazy loading, le CSS personnalisé léger plutôt qu'un framework lourd, et la limitation des dépendances JavaScript, démontre une prise de conscience des enjeux environnementaux du numérique et une volonté de créer une application responsable et performante.

En somme, La Conciergerie Auto est le fruit d'une démarche technique rigoureuse, créative et professionnelle, qui a su combiner des choix technologiques pertinents et modernes (Symfony, Vue.js, TypeScript, AWS S3, MySQL, JWT), une architecture sécurisée et évolutive respectant les principes MVC et REST, une base de données normalisée et optimisée, un design réfléchi et responsive avec une identité visuelle distinctive, et une approche d'écoconception pour aboutir à une application performante, sécurisée et engageante pour tous les acteurs de l'intermédiation automobile : clients cherchant à vendre ou acheter un véhicule, concierges gérant les workflows, et administrateurs supervisant la plateforme.

Ce projet de titre professionnel Développeur Web niveau Bac+2 démontre ma capacité à concevoir et développer une application web fullstack complète, de la modélisation de la base de données au déploiement en production, en passant par la création d'une API REST sécurisée et d'une interface utilisateur moderne et responsive. Les compétences acquises et mises en pratique couvrent l'ensemble du référentiel du titre professionnel : configuration d'environnement, maquettage, développement frontend et backend, gestion de base de données, sécurité, documentation, et déploiement.

# 16. ANNEXE

# Légende des Priorités

- **P0 — Critique / MVP**
  Indispensable pour que l'application fonctionne.
- **P1 — Important**
  Fonctionnalité majeure, mais le site peut fonctionner sans.
- **P2 — Confort / Bonus**
  Améliorations futures.

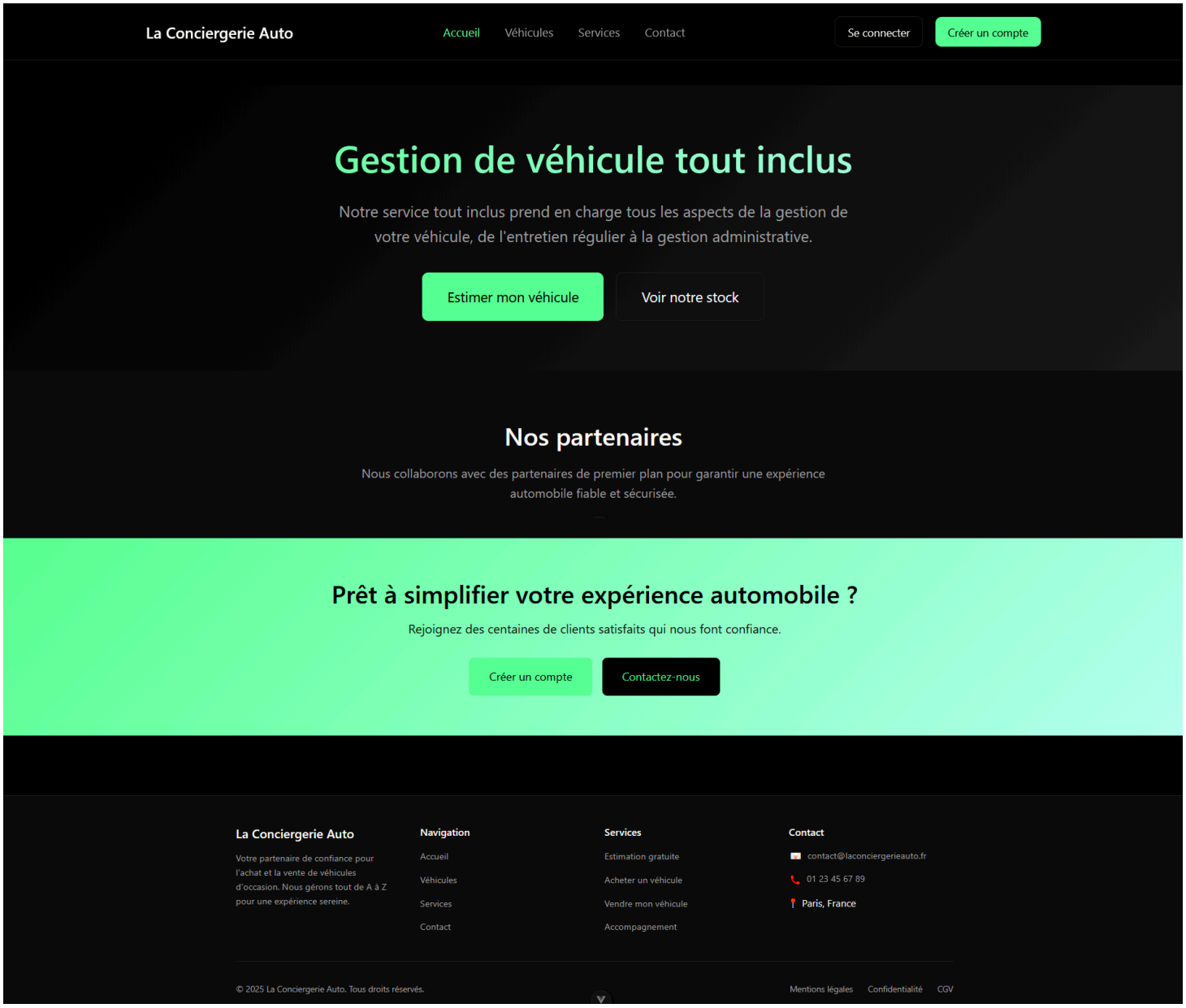
## USER STORIES

| Rôle | Je veux… | Afin de… | Priorité |
|---|---|---|---|
| **Visiteur** | M'inscrire en créant un compte | Accéder aux services de vente et d'achat | P0 |
| Visiteur | Recevoir un e-mail de confirmation | Valider que mon adresse e-mail est réelle | P0 |
| Visiteur | Consulter la liste des véhicules | Voir les voitures disponibles à l'achat | P0 |
| Visiteur | Voir le détail d’un véhicule | Connaître ses caractéristiques techniques | P0 |
| Visiteur | Filtrer les véhicules (Marque, Prix...) | Trouver rapidement ce que je cherche | P2 |
| **Utilisateur** | Me connecter avec email/mot de passe | Accéder à mon espace personnel sécurisé | P0 |
| Utilisateur | Réinitialiser mon mot de passe oublié | Récupérer l'accès à mon compte | P2 |
| **Client** | Uploader mes documents d'identité (CNI) | Valider mon profil pour vendre/acheter | P0 |

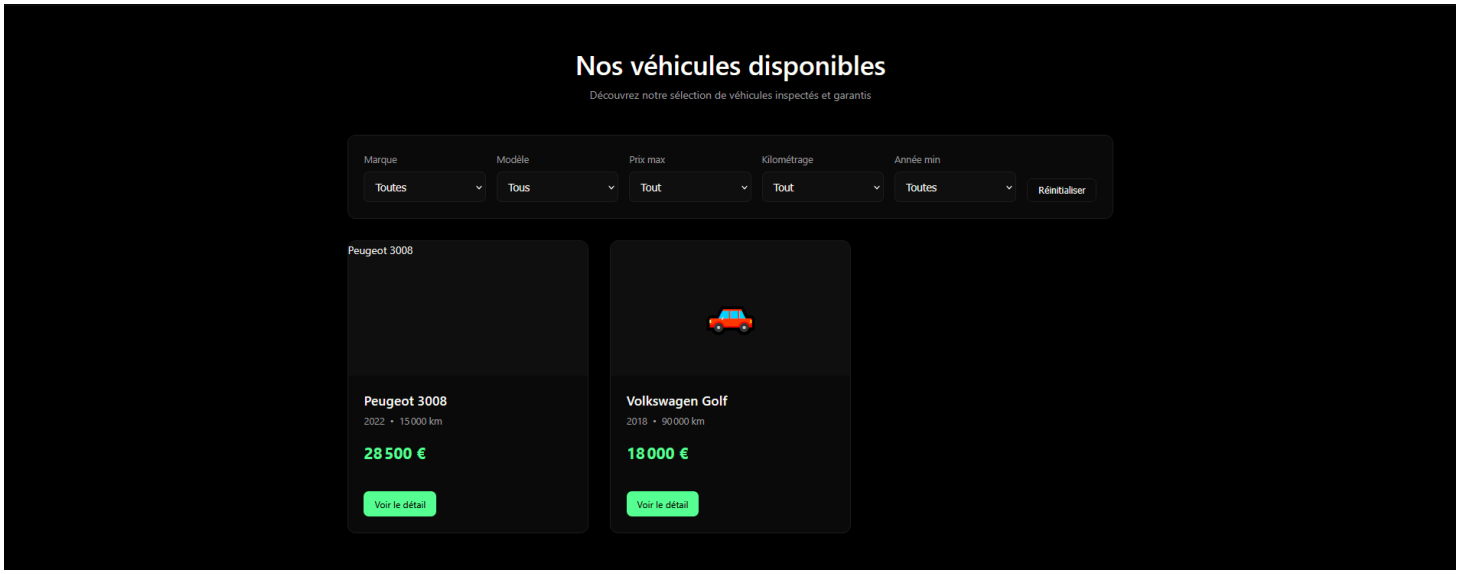
| Rôle | Je veux... | Afin de... | Priorité |
|---|---|---|---|
| Client | Modifier mes informations personnelles | Mettre à jour mon adresse ou mon téléphone | P1 |
| **Client Vendeur** | Remplir un formulaire d'estimation | Proposer mon véhicule à la vente | P0 |
| Client Vendeur | Voir la liste de mes demandes | Suivre l'état d'avancement | P1 |
| Client Vendeur | Recevoir une notification e-mail | Savoir que mon estimation est prête | P1 |
| Client Vendeur | Accepter ou refuser l'estimation | Valider la mise en vente ou stopper le processus | P0 |
| **Client Acheteur** | Voir mes transactions en cours | Suivre l'avancement de mon achat | P1 |
| Client Acheteur | Uploader mon attestation d’assurance | Prouver que je suis assuré pour le retrait | P0 |
| **Client (RDV)** | Voir les disponibilités d’un concierge | Choisir un créneau | P0 |
| Client (RDV) | Demander un RDV de visite | Voir et essayer le véhicule | P0 |
| Client (RDV) | Demander un RDV de retrait | Récupérer mon véhicule acheté | P0 |
| **Concierge** | Voir le pool des demandes en attente | Choisir les véhicules à traiter | P0 |
| Concierge | M'attribuer une demande | Gérer un dossier spécifique | P0 |
| Concierge | Soumettre une estimation de prix | Proposer une offre de rachat | P0 |
| Concierge | Créer une fiche véhicule | Transformer une demande acceptée en annonce | P0 |
| Concierge | Uploader des photos du véhicule | Rendre l’annonce attrayante | P0 |

| Rôle | Je veux… | Afin de… | Priorité |
|---|---|---|---|
| Concierge | Modifier les informations d’un véhicule | Corriger ou ajuster le prix | P1 |
| Concierge | Supprimer une photo spécifique | Nettoyer l’annonce | P1 |
| Concierge | Créer une transaction | Réserver le véhicule pour un acheteur | P0 |
| Concierge | Confirmer la réception du paiement | Valider la vente (statut « VENDU ») | P0 |
| Concierge | Voir mon agenda | Organiser ma journée | P0 |
| Concierge | Confirmer ou annuler un RDV | Gérer ma disponibilité | P1 |
| **Admin** | Créer un compte Concierge | Ajouter des employés | P0 |
| Admin | Modifier le rôle d’un utilisateur | Promouvoir un client en Concierge | P0 |
| Admin | Voir toutes les transactions | Suivre l’activité financière | P1 |
| Admin | Voir la liste de tous les utilisateurs | Gérer la base d’utilisateurs | P1 |
| Admin | Visionner les documents privés (CNI, Assurance) | Vérifier la conformité des dossiers | P0 |
| Admin | Gérer les marques et modèles | Maintenir la base référentielle | P2 |
| Admin | Voir les statistiques (KPI) | Suivre la performance de l’entreprise | P2 |
| **Système (API)** | Bloquer un RDV Retrait si absence d’assurance | Garantir la sécurité juridique | P0 |

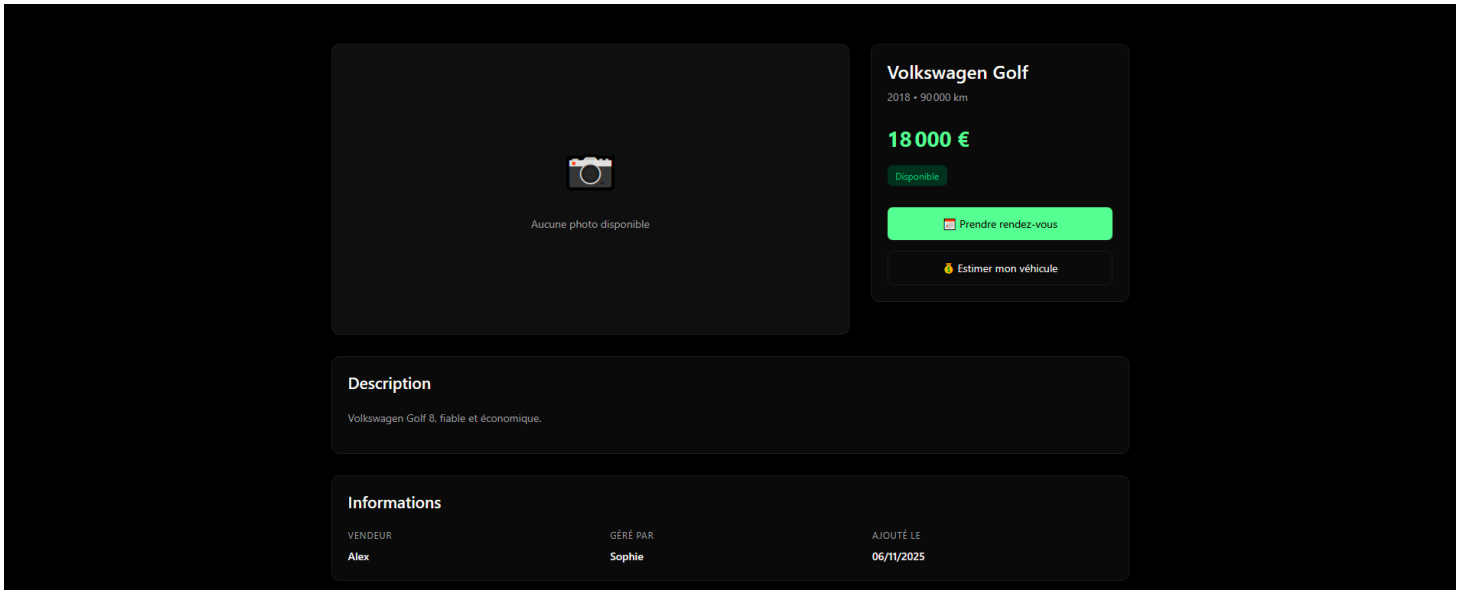
# ANNEXE


La Conciergerie Auto
Accueil
Véhicules
Services
Contact
Se connecter
Créer un compte
Gestion de véhicule tout inclus
Notre service tout inclus prend en charge tous les aspects de la gestion de votre véhicule, de l'entretien régulier à la gestion administrative.
Estimer mon véhicule
Voir notre stock
Nos partenaires
Nous collaborons avec des partenaires de premier plan pour garantir une expérience automobile fiable et sécurisée.
Prêt à simplifier votre expérience automobile ?
Rejoignez des centaines de clients satisfaits qui nous font confiance.
Créer un compte
Contactez-nous
La Conciergerie Auto
Votre partenaire de confiance pour l'achat et la vente de véhicules d'occasion. Nous gérons tout de A à Z pour une expérience sereine.
Navigation
Accueil
Véhicules
Services
Contact
Services
Estimation gratuite
Acheter un véhicule
Vendre mon véhicule
Accompagnement
Contact
contact@laconciergerieauto.fr
01 23 45 67 89
Paris, France
© 2025 La Conciergerie Auto. Tous droits réservés.
Mentions légales
Confidentialité
CGV

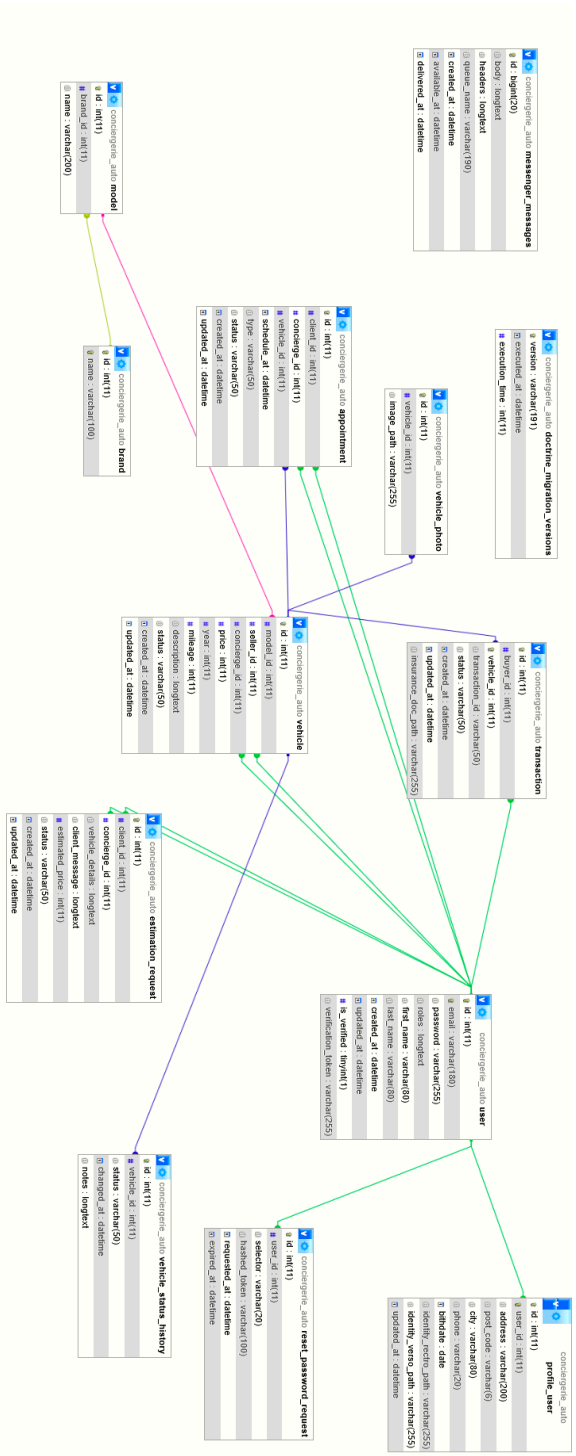
# ANNEXE


Nos véhicules disponibles
Découvrez notre sélection de véhicules inspectés et garantis
Marque
Toutes
Modèle
Tous
Prix max
Tout
Kilométrage
Tout
Année min
Toutes
Réinitialiser
Peugeot 3008
Peugeot 3008
2022 • 15000 km
28500 €
Voir le détail
Volkswagen Golf
2018 • 90000 km
18000 €
Voir le détail

# ANNEXE


Aucune photo disponible
Volkswagen Golf
2018 • 90 000 km
18 000 €
Disponible
Prendre rendez-vous
Estimer mon véhicule
Description
Volkswagen Golf 8, fiable et économique.
Informations
VENDEUR
Alex
GÉRÉ PAR
Sophie
AJOUTÉ LE
06/11/2025

# ANNEXE


concierge_auto messenger_messages
id : bigint(20)
body : longtext
headers : longtext
queue_name : varchar(190)
created_at : datetime
available_at : datetime
delivered_at : datetime
concierge_auto doctrine_migration_versions
version : varchar(191)
executed_at : datetime
execution_time : int(11)
concierge_auto model
id : int(11)
brand_id : int(11)
name : varchar(200)
concierge_auto appointment
id : int(11)
client_id : int(11)
concierge_id : int(11)
vehicle_id : int(11)
schedule_at : datetime
type : varchar(50)
status : varchar(50)
created_at : datetime
updated_at : datetime
concierge_auto vehicle_photo
id : int(11)
vehicle_id : int(11)
image_path : varchar(255)
concierge_auto brand
id : int(11)
name : varchar(100)
concierge_auto transaction
id : int(11)
buyer_id : int(11)
vehicle_id : int(11)
transaction_id : varchar(50)
status : varchar(50)
created_at : datetime
updated_at : datetime
insurance_doc_path : varchar(255)
concierge_auto vehicle
id : int(11)
model_id : int(11)
seller_id : int(11)
concierge_id : int(11)
price : int(11)
year : int(11)
mileage : int(11)
description : longtext
status : varchar(50)
created_at : datetime
updated_at : datetime
concierge_auto estimation_request
id : int(11)
client_id : int(11)
concierge_id : int(11)
vehicle_details : longtext
client_message : longtext
estimated_price : int(11)
status : varchar(50)
created_at : datetime
updated_at : datetime
concierge_auto user
id : int(11)
email : varchar(180)
password : varchar(255)
roles : longtext
first_name : varchar(80)
last_name : varchar(80)
created_at : datetime
updated_at : datetime
is_verified : tinyint(1)
verification_token : varchar(255)
concierge_auto vehicle_status_history
id : int(11)
vehicle_id : int(11)
status : varchar(50)
changed_at : datetime
notes : longtext
concierge_auto reset_password_request
id : int(11)
user_id : int(11)
selector : varchar(20)
hashed_token : varchar(100)
requested_at : datetime
expired_at : datetime
concierge_auto profile_user
id : int(11)
user_id : int(11)
address : varchar(200)
post_code : varchar(6)
city : varchar(80)
phone : varchar(20)
birthdate : date
identity_recto_path : varchar(255)
identity_verso_path : varchar(255)
updated_at : datetime