

# Dossier de Projet Professionnel



Jules Jean-Louis

# SOMMAIRE

<b>Introduction</b>	<b>3</b>
Compétences couvertes par le projet	4
<b>1. Contexte et Besoins</b>	<b>4</b>
Contexte de réalisation du projet	4
Problématique métier	4
Analyse des solutions existantes	5
Analyse de la cible : TPE et PME	5
Solutions Proposer : Proxifix	5
Analyse des Besoins Métier	6
Interface Client	6
Interface Entreprise (Techniciens et Administrateurs)	6
Analyse des Besoins Non Fonctionnels	6
Choix Techniques et Justifications	7
Stack Frontend Mobile	8
Stack Backend	8
Autres Outils	9
Gestion de Projet et Méthodologie	9
Stratégie Git	10
<b>3. Conception et Architecture</b>	<b>11</b>
Architecture SaaS Multi-Tenant	11
Architecture par couche	11
Architecture Globale du Système	12
Conception de la Base de Données	13
Modèle conceptuel de données	13
Modèle Logique de Données (MLD)	15
Modèle Physique de Données (MPD)	15
Maquettes et Prototypage	16
Maquettage sur Figma	18
<b>4. Développement et Implémentation</b>	<b>22</b>
Environnement de Développement - Docker et Containerisation	22
Développement du backend de l'application	23
Arborescence et Organisation	23
Fonctionnement de l'API et Routage	24
Contrôleurs	25
Services	25
ORM Doctrine	25
Entités	26
Repositories	27
Sécurité de l'API	27
Chiffrement des mots de passe et Authentification JWT	27
Configuration de sécurité (security.yaml)	29

Protection CORS	30
Problématiques Rencontrées : Gestion des créneaux disponibles	30
Solution Implémentée : Refactoring vers une Architecture Orientée Services	31
Documentation automatique avec API Platform	32
Développement Frontend - React Native	33
Navigation avec Expo Router	34
Principe DRY avec les composants	35
Gestion de l'Authentification	36
Gestion des appels API	37
Validation de Formulaires et Sécurité Client	37
<b>6. Qualité, Tests et Déploiement</b>	<b>39</b>
Plan de tests fonctionnels	39
Tests Unitaires	39
Tests d'Intégration API	40
Cahier de Recette	41
Intégration Continue (CI) et Qualité du Code	42
Préparation au Déploiement et Déploiement Continu (CD)	43
Veille Technologique et Sécurité	44
<b>Conclusion</b>	<b>45</b>
<b>ANNEXE</b>	<b>46</b>

# Introduction

Je m'appelle Jules Jean-Louis, j'ai 29 ans et je suis en reconversion professionnelle vers les métiers du développement et de la conception d'applications.

Je suis actuellement en alternance chez Natural-solutions, en vue de la préparation au diplôme de **Concepteur Développeur d'Applications**, je souhaiterais pouvoir continuer mon apprentissage après cette formation.

Dans le cadre de ma formation, j'ai eu l'opportunité de concevoir et de développer Proxifix, une application mobile dédiée à la gestion des interventions informatiques. Ce projet est né d'un constat simple : l'absence de solutions mobiles complètes pour les techniciens sur le terrain, la plupart des outils existants étant conçus pour une utilisation sur ordinateur.

Proxifix répond à ce besoin en offrant une plateforme intuitive et performante. Elle permet aux professionnels de planifier, suivre et gérer efficacement leurs interventions directement depuis leurs appareils mobiles.

Ce dossier présente mon projet, en détaillant l'analyse des besoins, les choix techniques, les étapes de développement et les solutions apportées aux problématiques rencontrées. Il a pour objectif de démontrer mes compétences techniques et méthodologiques acquises durant ma formation.

# Compétences couvertes par le projet

Ce projet couvre les compétences du titre suivantes :

- Installer et configurer son environnement de travail en fonction du projet
- Développer des interfaces utilisateur
- Développer des composants métier
- Contribuer à la gestion d'un projet informatique
- Analyser les besoins et maquetter une application
- Définir l'architecture logicielle d'une application
- Concevoir et mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL
- Préparer et exécuter les plans de tests d'une application
- Préparer et documenter le déploiement d'une application
- Contribuer à la mise en production dans une démarche DevOps

# 1. Contexte et Besoins

## Contexte de réalisation du projet

Ce projet a été réalisé dans le cadre de ma formation de Concepteur Développeur d'Application, sur un rythme d'une semaine tous les mois lorsque nous étions en centre. Le projet a été réalisé sur la première partie en groupe de 3, la seconde en développement solo.

## Problématique métier

Un membre de notre équipe de développement travaillait déjà avec un logiciel de gestion d'interventions informatiques dans le cadre de son alternance. Cet outil, cependant, n'est qu'une solution destinée à des ordinateurs de bureau, sans possibilité d'utilisation mobile. Très rapidement, nous nous sommes mis d'accord sur la réalisation d'une application mobile, avec comme point de départ le même concept.

## Analyse des solutions existantes

Notre analyse du marché a révélé l'existence de plusieurs solutions spécialisées dans la réparation et maintenance de matériel informatique, mais la grande majorité d'entre elles sont aussi destinées à desktop/web. Avec des applications mobiles souvent limitées ou inexistantes ou nécessitant des frais supplémentaires. Cette lacune représente une opportunité, notamment pour les techniciens qui interviennent directement chez les clients.

## Analyse de la cible : TPE et PME

Notre analyse s'est focalisée sur les difficultés rencontrées par les TPE et PME qui constituent la cible principale. Les TPE/PME du secteur font face à plusieurs défis majeurs impactant directement leur productivité et satisfaction client. La gestion manuelle des interventions via papier ou tableaux génère une perte de temps considérable, les techniciens devant ressaisir les mêmes informations multiples fois (fiche intervention, rapport, facturation), transformant une partie significative de leur temps en tâches administratives non facturables. Le manque de traçabilité des équipements complique le suivi technique, particulièrement problématique pour les matériels récidivants où l'absence d'historique peut conduire à répéter des diagnostics ou réparations déjà effectués. La communication défaillante entre clients et techniciens provoque des malentendus fréquents : les clients ignorent l'heure d'arrivée du technicien et l'état d'avancement de leur demande, générant de multiples appels de suivi qui surchargent le service client. Cette absence de transparence et de suivi temps réel nuit considérablement à l'image de professionnalisme et à la satisfaction clientèle.

## Solutions Proposer : Proxifix

Notre solution consiste à développer une application mobile qui permet de digitaliser complètement la gestion des interventions informatiques. Cette solution couvre tous les types d'interventions dans le domaine informatique, qu'il s'agisse de maintenance préventive ou curative, de réparation de matériel défaillant, ou d'installation de nouveaux équipements.

L'application fonctionne selon le modèle SaaS qui signifie "Software as a Service" (Logiciel en tant que Service). Il s'agit d'un système unifié qui accompagne les informations depuis la demande initiale du client jusqu'à la finalisation de l'intervention et sa facturation. Cette approche garantit qu'aucune information ne se perd en cours de processus et que chaque étape est tracée et documentée.

## Analyse des Besoins Métier

Une analyse fonctionnelle détaillée a permis d'identifier quatre rôles utilisateurs distincts, chacun avec des besoins et des niveaux d'accès spécifiques. L'application est donc divisée en deux grandes interfaces :

### Interface Client

Les clients ont besoin d'être autonomes pour gérer leurs demandes d'intervention :

- **Consulter l'état de leurs interventions** : Suivre l'avancement de leur demande, voir les tâches réalisées et estimer les délais sans avoir à contacter le service client.
- **Gérer leur parc d'équipements** : Ajouter de nouveaux matériels ou modifier les détails des équipements existants.
- **Prendre rendez-vous directement** : Une interface automatisée permet de prendre rendez-vous 24h/24, réduisant les délais et les erreurs.
- **Accéder à l'historique complet de leurs interventions** : Une traçabilité essentielle pour les garanties, le suivi technique et l'analyse des pannes récurrentes.

### Interface Entreprise (Techniciens et Administrateurs)

Cette partie est dédiée au personnel de l'entreprise :

#### Technicien :

- **Visualiser leurs plannings** avec les interventions assignées.
- **Accéder aux données techniques** des équipements à réparer.
- **Documenter les tâches** effectuées pendant l'intervention.
- **Modifier le statut** des interventions en fonction de leur avancement.

#### Administrateurs d'Entreprise :

- Avoir une **vue d'ensemble** de toutes les interventions en cours pour leur entreprise.
- **Assigner et réassigner** les techniciens en fonction des priorités.
- **Gérer les catalogues de services**, les tâches et la planification.
- **Créer et gérer** les comptes clients et techniciens.

### Super Administrateur :

- **Valider et intégrer** de nouvelles entreprises sur la plateforme.
- Assurer une **supervision technique globale**.
- **Gérer les droits et les permissions** au niveau de la plateforme.

### Analyse des Besoins Non Fonctionnels

Pour répondre à ces exigences, la solution technique a été choisie pour :

- Être utilisable sur **iOS et Android** (cross-plateforme).
- Offrir des **coûts réduits** grâce à une instance unique plutôt qu'un déploiement par entreprise.
- Garantir une **sécurité accrue** et une **herméticité des données** entre les différentes entreprises.



## 2. Contraintes et Organisation

### Choix Techniques et Justifications

Nos choix technologiques ont été principalement dictés par l'**expérience préalable de l'équipe** et les **délais serrés** imposés par le planning de formation. Cette approche nous a permis de nous concentrer pleinement sur la logique métier.

#### Stack Frontend Mobile



Nous avons opté pour **React Native** pour le développement frontend, car tous les membres du groupe possédaient déjà des compétences sur cette technologie. React Native est un framework de développement d'applications mobiles qui permet de produire du code natif iOS et Android à partir d'une **unique base de code JavaScript**

**compilée**. Sa vaste communauté active et la richesse de ses librairies additionnelles ont également été des atouts majeurs.



Pour faciliter le développement, nous avons intégré **Expo**. Cet outil nous a permis de nous affranchir des SDK dédiés à iOS ou Android Studio, simplifiant ainsi grandement le processus. Expo facilite également la mise en production grâce aux mises à jour "Over-the-Air" (OTA), permettant de déployer des modifications sans repasser par les stores.

#### Stack Backend



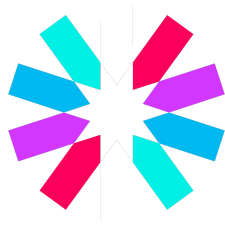
Symfony

Notre choix s'est porté sur **Symfony 6 avec PHP 8.3** pour sa simplicité d'utilisation et ses excellentes capacités de développement d'API. Couplé à l'**ORM Doctrine**, il offre une couche d'abstraction essentielle entre le backend et la base de données. Cela nous a aussi donné la flexibilité de modifier le schéma via les migrations et de gérer aisément les montées de versions.



**Docker** a été un élément clé pour créer un **environnement de développement reproductible et cohérent**, très similaire à celui qui serait utilisé en production. Grâce à la conteneurisation, nous n'avons besoin que de Docker et Docker Compose, ce qui nous a permis d'isoler

les services comme PHP et la base de données, sans nous soucier de la compatibilité des dépendances sur les différentes machines.



La **sécurité** est assurée par l'utilisation de **JSON Web Tokens (JWT)**, garantissant une authentification "stateless" (sans sauvegarde de données de session sur le serveur). Le JWT est un standard qui permet de transmettre des données de manière sécurisée sous forme de JSON.

## Autres Outils

**Insomnia** : Pour tester les requêtes API.

**Visual Studio Code** : Notre environnement de développement intégré (IDE).

**Git et GitHub** : Pour le versioning du code et l'hébergement du dépôt.

**Figma** : Utilisé pour élaborer la charte graphique, le design system et les maquettes.

## Gestion de Projet et Méthodologie

Pour gérer les délais, le flux de travail et la qualité de notre projet, nous avons adopté la méthodologie **Kanban**, une approche issue du "Manifeste Agile" qui s'est avérée parfaitement adaptée à notre rythme de travail. Plutôt que de suivre des cycles de développement rigides, Kanban nous a permis de maintenir un flux de travail continu, ce qui nous a offert la flexibilité nécessaire pour réajuster nos priorités en cas de bugs ou de nouvelles fonctionnalités urgentes.

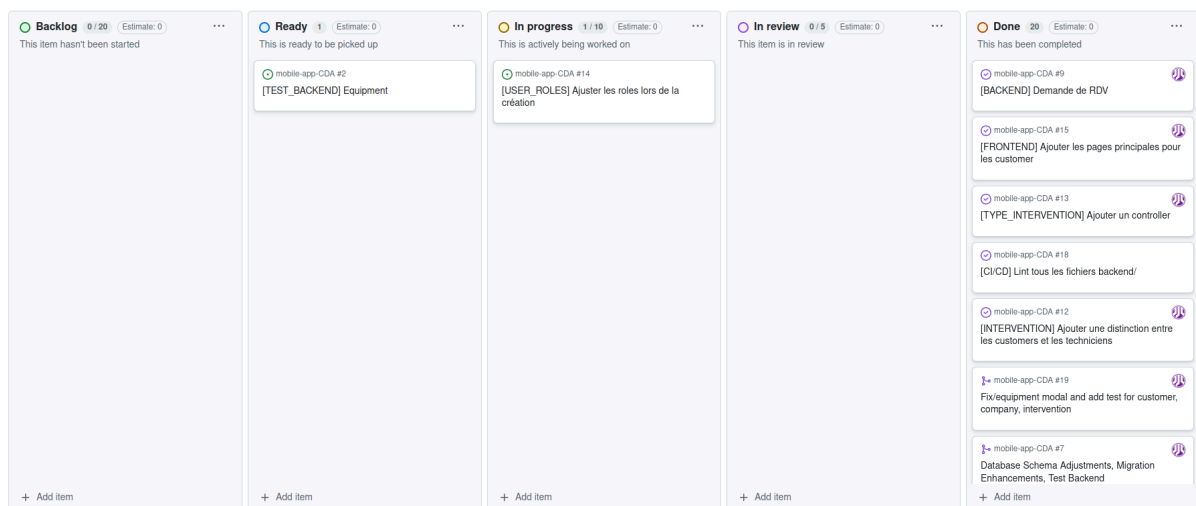
Pour matérialiser cette méthode, nous avons mis en place un tableau de gestion des tâches sur GitHub. Ce tableau était organisé en plusieurs colonnes distinctes, chacune représentant une étape clé de notre processus de développement. L'objectif était de découper les fonctionnalités en tâches plus petites et réalisables rapidement, garantissant ainsi le respect de nos délais.

Le processus commençait dans le **Backlog**, notre réservoir d'idées où nous stockions toutes les tâches potentielles pour le futur. Ces tâches étaient priorisées, mais non formellement engagées, ce qui nous laissait une grande marge de manœuvre.

Une fois que les tâches étaient jugées pertinentes, elles rejoignaient la colonne **À faire (Ready)**. Cette étape marquait le moment où une tâche était prête à être commencée. C'est ici que nous menions l'analyse, la conception, la réflexion sur les solutions techniques, fonctionnelles ou encore les spécifications. Chaque membre pouvait alors s'auto-attribuer une tâche et la déplacer dans la colonne **En cours (In Progress)**. Cette colonne était soumise à une **limite de travail en cours**, une règle fondamentale de Kanban pour éviter la dispersion, gagner en concentration et en efficacité.

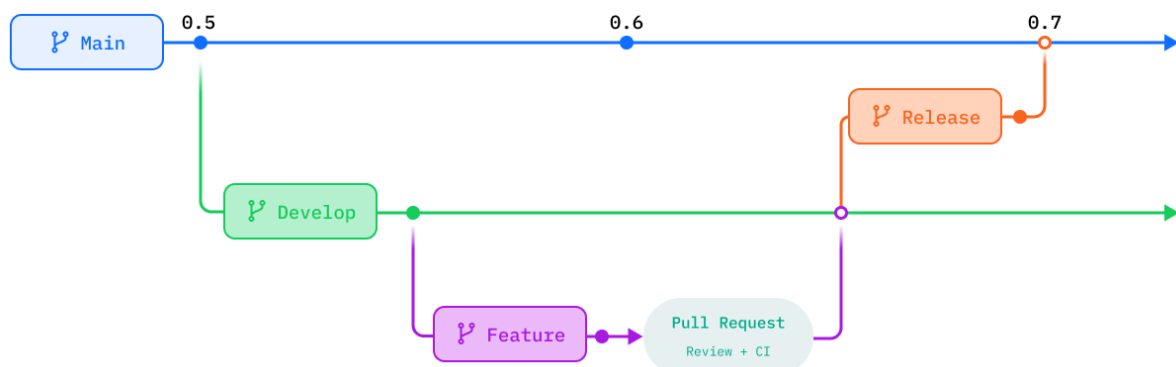
Lorsqu'une tâche était terminée, elle passait en **Révision de code (In Review)**, une étape essentielle pour maintenir la qualité. C'est dans cette colonne que les membres de l'équipe examinaient le code pour s'assurer de sa robustesse, de l'absence d'erreurs, et effectuaient tous les tests nécessaires (unitaires, fonctionnels, etc.) pour valider que la tâche répondait aux attentes.

Enfin, une fois que toutes les validations étaient passées, la tâche était déplacée dans la colonne **Terminé (Done)**, marquant sa finalisation complète. Ce flux de travail transparent et la collaboration autour d'issues GitHub détaillées nous ont permis de maintenir une communication efficace et de respecter nos objectifs de projet.



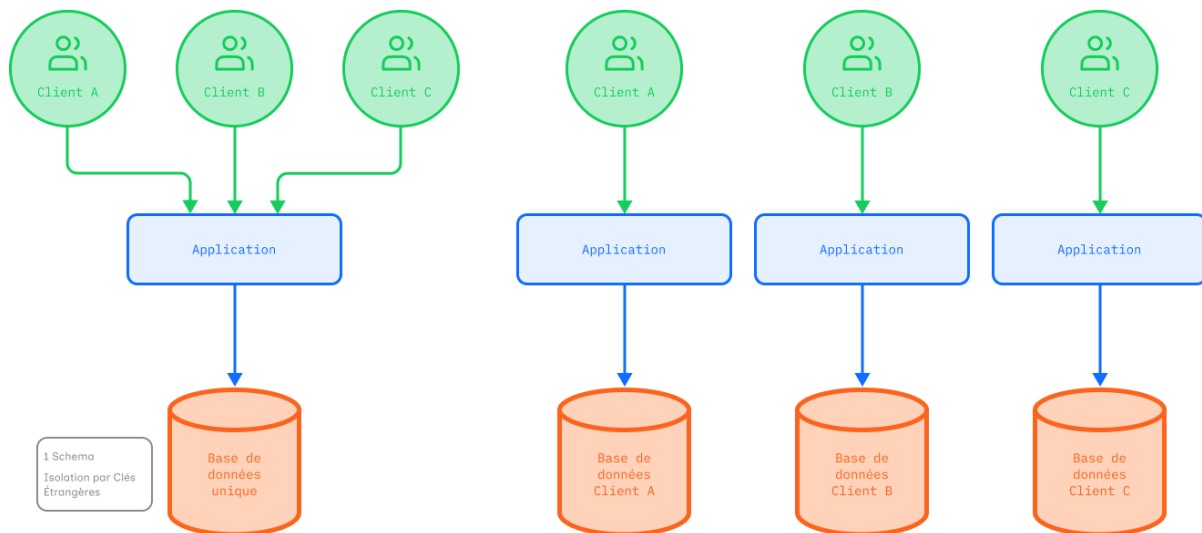
## Stratégie Git

On a mis en place une stratégie de versionnement basée sur Git Flow. On utilise deux branches principales protégées : **main** et **develop**. Toutes les nouvelles fonctionnalités et les correctifs sont développés sur des branches dédiées avant d'être fusionnés sur la branche **develop**. Une fois que les développements sur **develop** justifient une nouvelle version stable de l'application, les modifications sont alors fusionnées sur la branche **main**. Cela nous permet de bien séparer le travail en cours des versions stables.



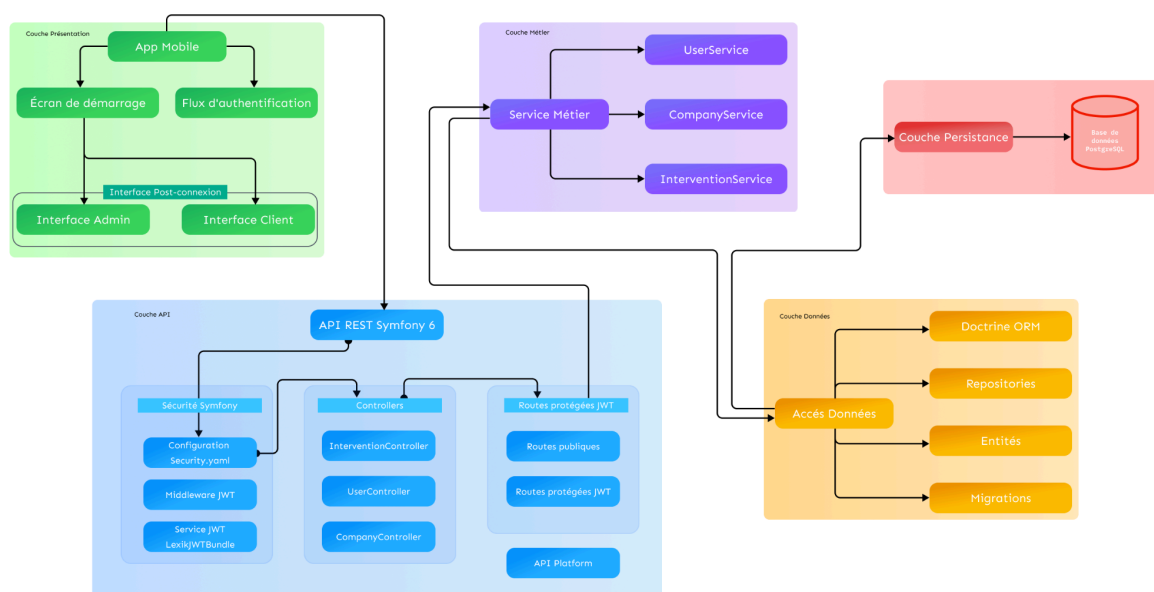
# 3. Conception et Architecture

## Architecture SaaS Multi-Tenant



On a fait le choix d'une **architecture multi-tenant**, plutôt qu'une solution avec une instance par entreprise. Ça veut dire que toutes les entreprises partagent la **même base de données (shared-database)**, avec un seul schéma. Mais attention, les données de chaque entreprise sont bien **séparées et sécurisées** grâce à des clés étrangères dans les tables. Le gros avantage, c'est que ça **optimise les coûts** en mutualisant les ressources, et ça nous permet de garder la plateforme **toujours à jour** et de proposer rapidement de **nouvelles fonctionnalités** à tout le monde.

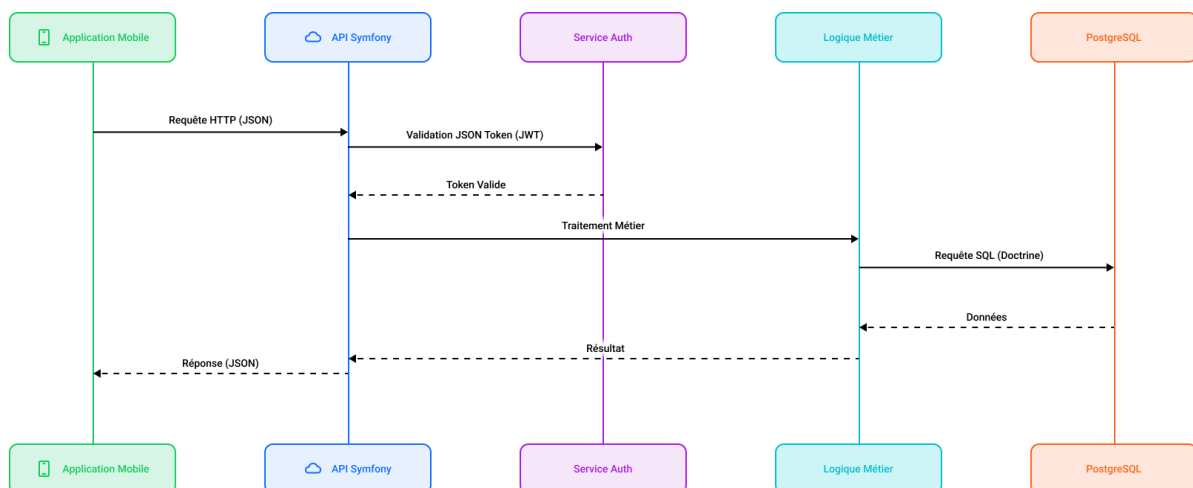
## Architecture par couche



L'architecture de notre application est divisée en plusieurs couches distinctes, chacune ayant une responsabilité claire et bien définie. Cette approche garantit la modularité, la maintenabilité et la bonne organisation de notre code.

La **Couche Présentation** représente l'interface utilisateur, accessible via une application mobile. C'est elle qui gère l'écran de démarrage, le flux d'authentification et les différentes interfaces utilisateur (administrateur, client, et point-concessionnaire), permettant l'interaction directe avec l'utilisateur. En dessous se trouve la **Couche API (API REST Symfony 6)**, qui sert de pont vers notre backend. Elle reçoit les requêtes de la couche de présentation et est responsable de la sécurité (avec la configuration `security.yaml` et un middleware JWT), du routage et de la logique métier de haut niveau. C'est à ce niveau qu'**API Platform** est intégré pour faciliter la création des endpoints REST. La **Couche Métier**, quant à elle, contient la logique métier complexe de l'application, organisée en services comme le `UserService` pour les utilisateurs, le `CompanyService` pour les entreprises ou l'`InterventionService` pour les interventions. Les contrôleurs de l'API délèguent les tâches les plus complexes à ces services pour maintenir la séparation des responsabilités. Ces services interagissent avec la **Couche Données**, qui est responsable de l'accès et de la gestion de la base de données. Elle inclut **Doctrine ORM**, qui sert d'interface entre l'application et la base de données, ainsi que les **Repositories** (pour les requêtes complexes), les **Entités** (qui représentent nos tables) et les **Migrations** (pour gérer l'évolution du schéma). Enfin, la **Couche Persistance** représente la base de données elle-même, une base de données SQL dans notre cas.

## Architecture Globale du Système



L'architecture globale de notre système s'articule autour d'un flux de communication précis et sécurisé entre les différentes couches. Le processus débute par l'**Application Mobile** qui envoie une requête HTTP (au format JSON) à notre **API Symfony**. Cette requête est d'abord traitée par un service d'authentification qui se charge de valider le **JSON Web Token (JWT)**. Si le token est valide, la requête est transmise pour son **Traitement Métier**. Ce traitement interagit avec notre couche de **Logique Métier** qui, à son tour, exécute une requête SQL (via Doctrine) vers la base de données **PostgreSQL**. Une fois que les données sont récupérées et la logique métier appliquée, le résultat est renvoyé à l'API Symfony, qui convertit en une réponse JSON. C'est cette réponse qui est finalement transmise à l'Application Mobile pour être affichée à l'utilisateur. Ce circuit garantit que chaque requête

est authentifiée, que la logique est gérée de manière centralisée et que l'accès aux données est sécurisé.

## Conception de la Base de Données

La conception de la base de données, nous avons suivi la **méthode Merise**, une approche structurée et rigoureuse qui aide à éviter les redondances et à garantir la cohérence des données. Merise, une méthode française, se décompose en plusieurs étapes progressives, chacune aboutissant à des modèles de plus en plus détaillés.

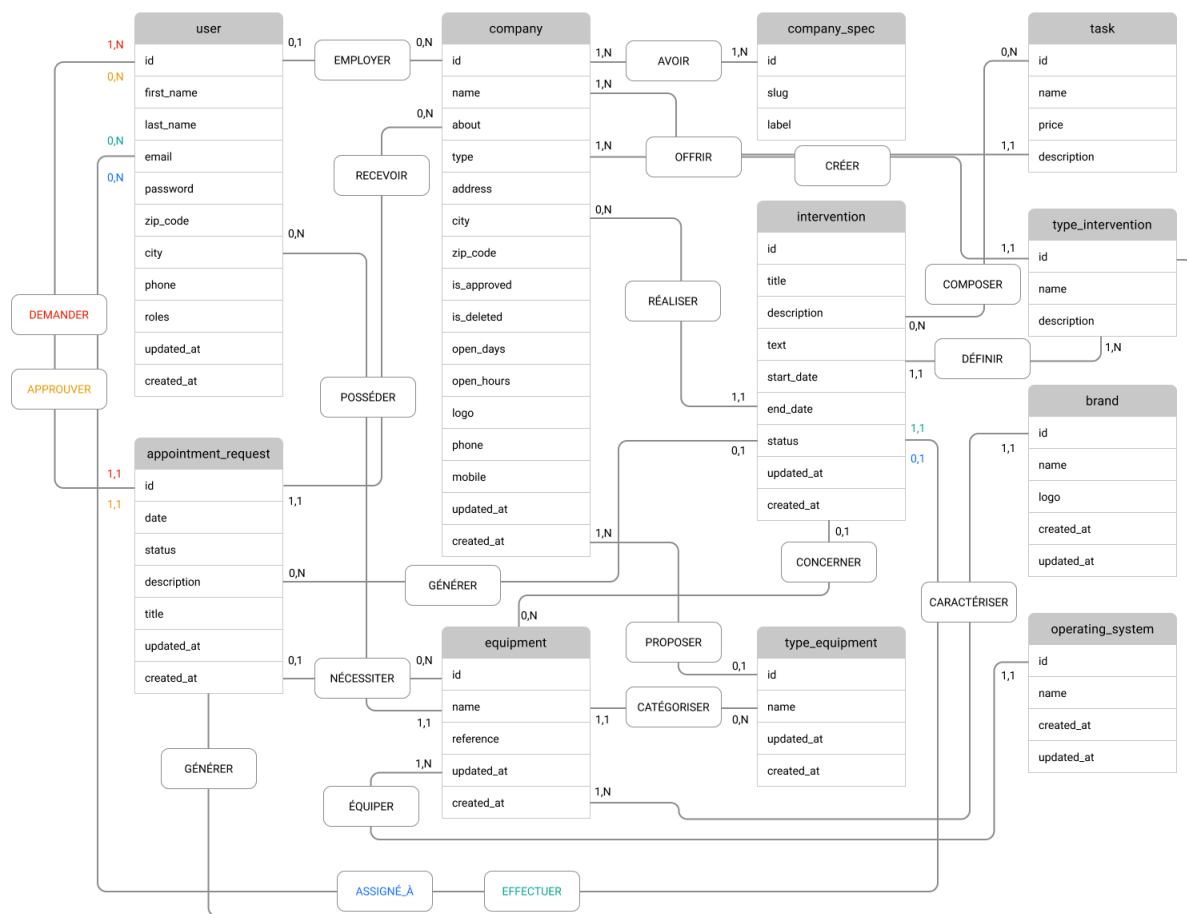
La base de données s'organise autour de l'entité **Company**, qui représente chaque entreprise de réparation présente sur la plateforme. Un champ **is\_approved** permet une validation des entreprises par les super administrateurs. Pour s'adapter à notre architecture multi-tenant, chaque entité métier intègre une **clé étrangère company\_id**, ce qui permet d'isoler automatiquement et efficacement les données de chaque entreprise au sein du schéma unique.

### Modèle conceptuel de données

Nous avons commencé par le **MCD (Modèle Conceptuel de Données)**. À cette étape, j'ai défini toutes les entités de l'application, leurs attributs, et mis en évidence les cardinalités (les relations entre les entités).

Pour les utilisateurs, j'ai choisi d'utiliser une seule table (**user**) pour gérer à la fois les clients et les administrateurs/techniciens, en les distinguant par leurs rôles. Pour simplifier, les utilisateurs ayant les rôles de technicien ou d'administrateur sont rattachés à une seule entreprise. C'est pourquoi l'entité **user** est liée à l'entité **intervention** de différentes manières selon le rôle : un client peut avoir de 0 à plusieurs interventions, tandis qu'un technicien ou un administrateur peut être assigné à 0 à plusieurs interventions. Une intervention doit être liée à un client dès sa création, mais l'assignation à un technicien peut

se faire après coup, d'où une cardinalité de 0 à 1 pour le technicien.



Chaque **equipment** (équipement) est clairement lié à :

- Un **Type d'équipement** (par exemple, ordinateur, smartphone, tablette).
- Une **Brand** (comme Apple, Samsung, HP).
- Un **Operating System** (comme Windows, macOS, Android).

Pour les **interventions**, nous avons défini **plusieurs états** pour suivre leur progression :

- **PENDING** : L'intervention est créée mais pas encore assignée.
- **ASSIGNED** : Elle a été assignée à un technicien.
- **AWAITING\_PICKUP** : En attente de récupération par le client.
- **IN\_PROGRESS** : L'intervention est en cours de traitement.
- **COMPLETED** : L'intervention est terminée.
- **CANCELLED** : L'intervention a été annulée.

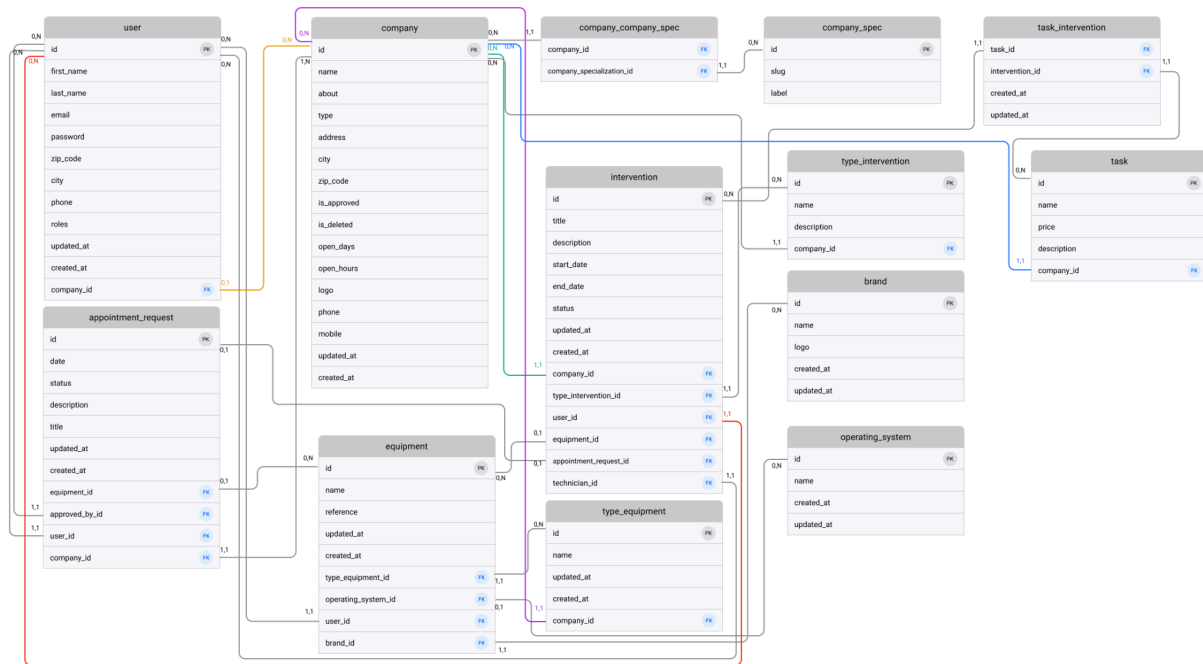
Un système de prise de rendez-vous est également intégré via l'entité **AppointmentRequest** et suit un workflow précis :

1. Le client crée une demande avec le statut **PENDING**.
2. Le technicien peut la confirmer, elle passe alors au statut **CONFIRMED**.

3. Une fois planifiée, la demande passe en **SCHEDULED** et génère automatiquement une **intervention**.

## Modèle Logique de Données (MLD)

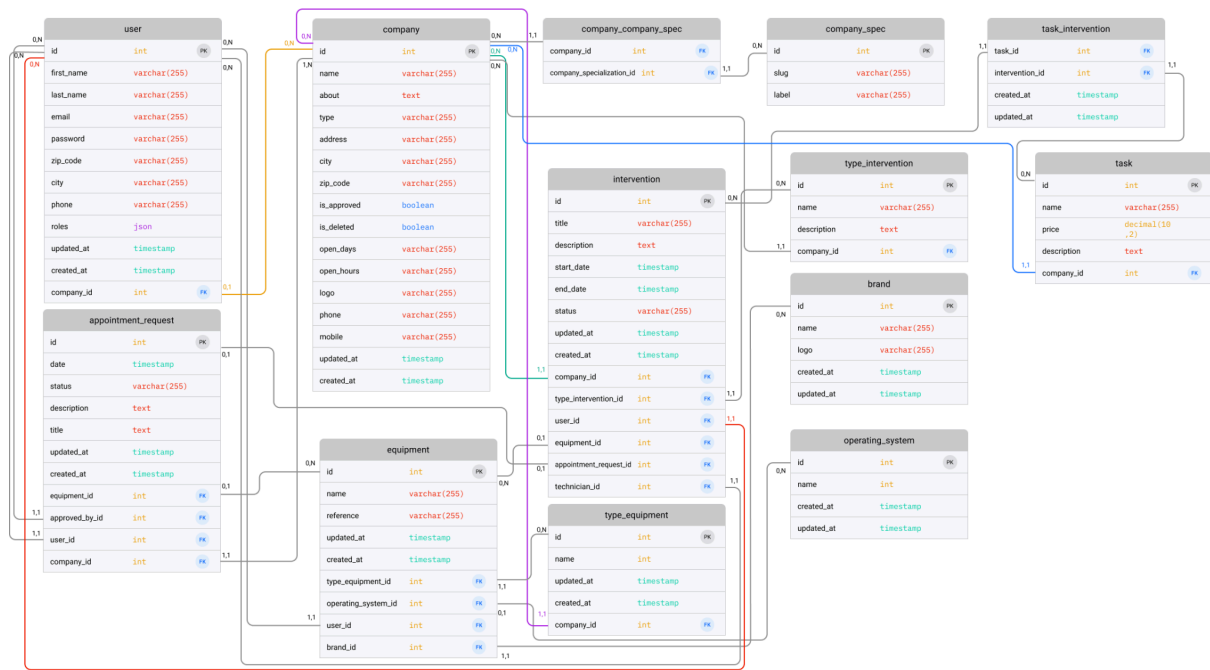
Dans cette étape, j'ai traduit le MCD en MLD. J'ai supprimé les verbes entre les entités et fait apparaître les **tables de liaison** pour les relations de plusieurs à plusieurs (comme entre **intervention** et **task**), ainsi que toutes les **clés étrangères**. Les cardinalités ont également été adaptées pour refléter cette structure logique.



## Modèle Physique de Données (MPD)

La dernière étape fut le **MPD**, qui est la représentation la plus proche du schéma réel implémenté dans ma base de données PostgreSQL. Ici, j'ai spécifié les **types de données** précis pour chaque attribut de chaque table.



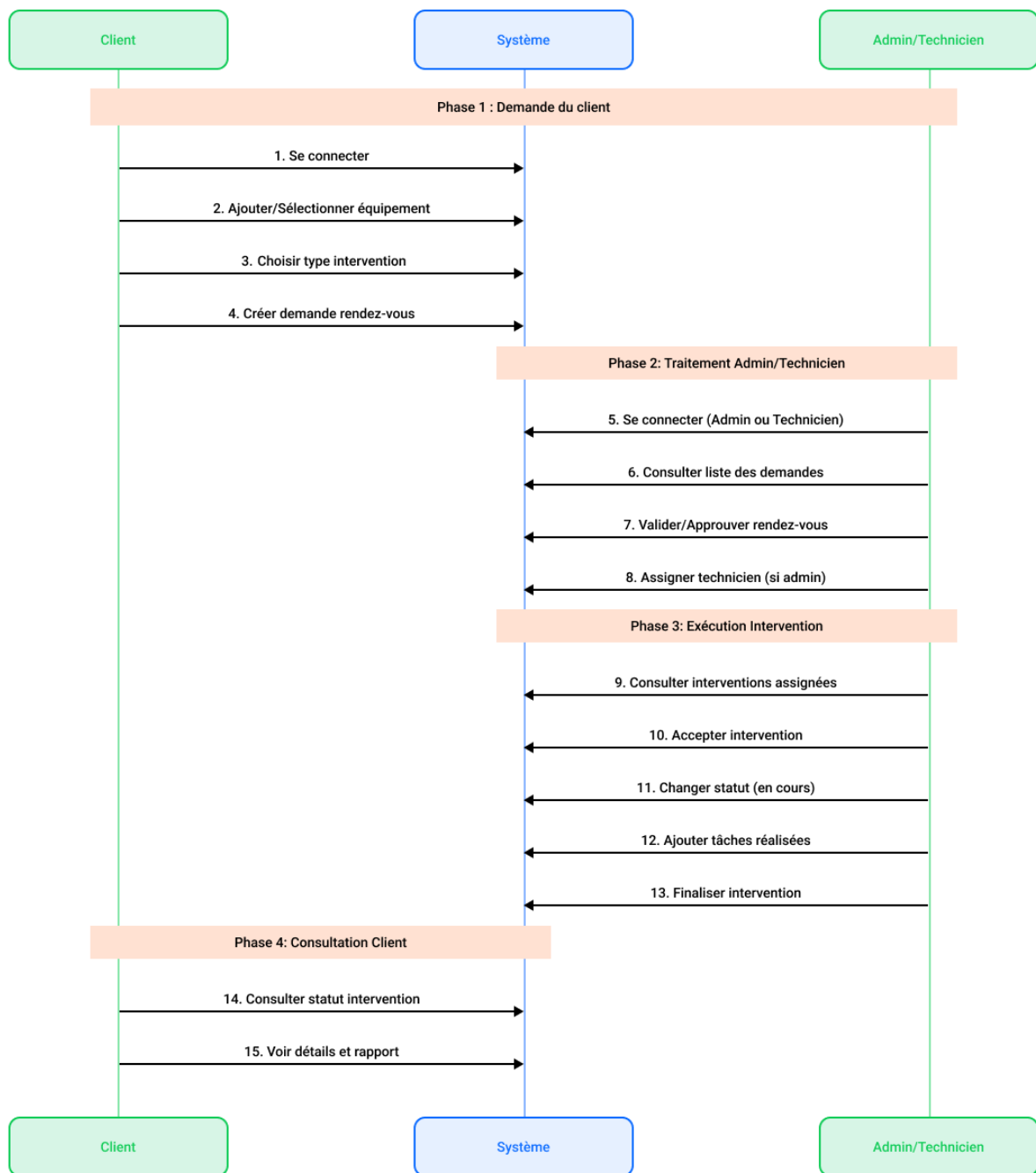


## Maquettes et Prototypage

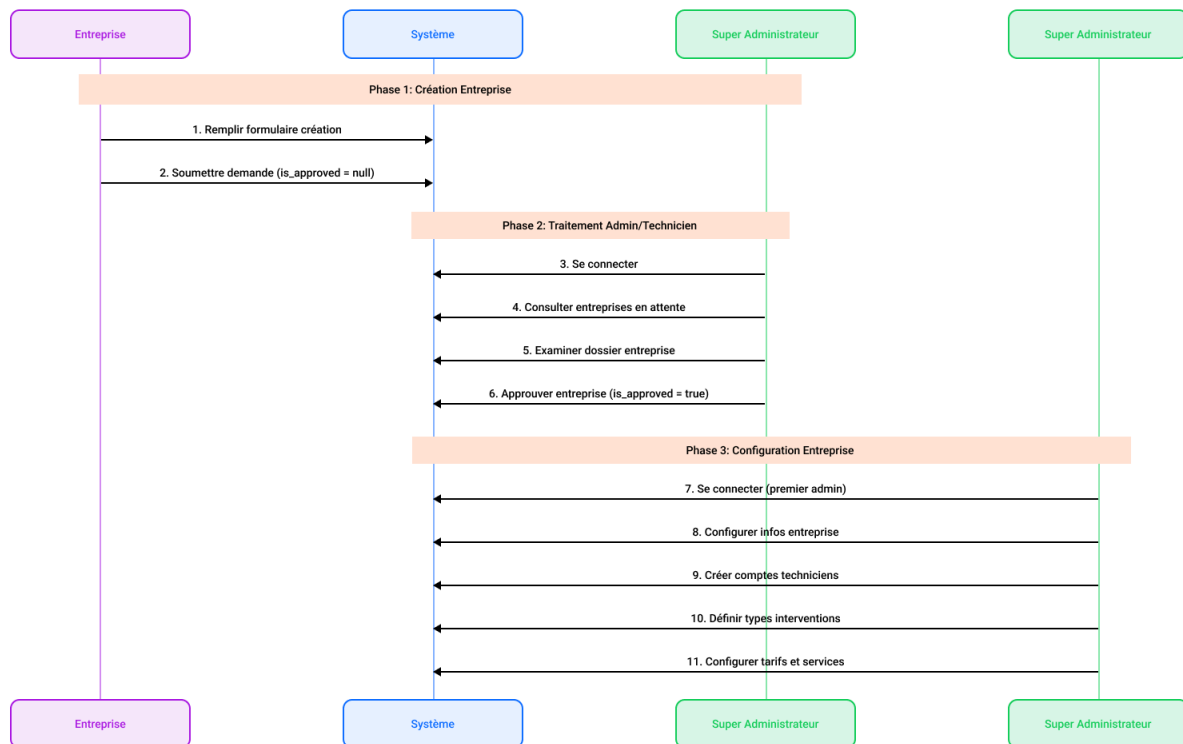
Je me suis ensuite attelé à la création de la maquette, constitue une étape fondamentale dans le processus de développement, car elle permet de valider les parcours utilisateur et l'ergonomie avant d'écrire la première ligne de code. Cette validation précoce réduit considérablement les risques de refonte tardive qui pourraient compromettre les délais.

Je me suis concentré sur deux scénarios clés : d'une part, la création et le processus d'une intervention par un client, et d'autre part, l'ajout et la validation de nouvelles entreprises sur la plateforme. Cette approche, basée sur des cas d'usage concrets, m'a permis de comprendre en amont le flux de données et les interactions essentielles à implémenter, bien avant de concevoir les maquettes. C'était une étape cruciale pour assurer la pertinence et l'efficacité des interfaces.

### Scénario 1 : Demande d'Intervention Complète

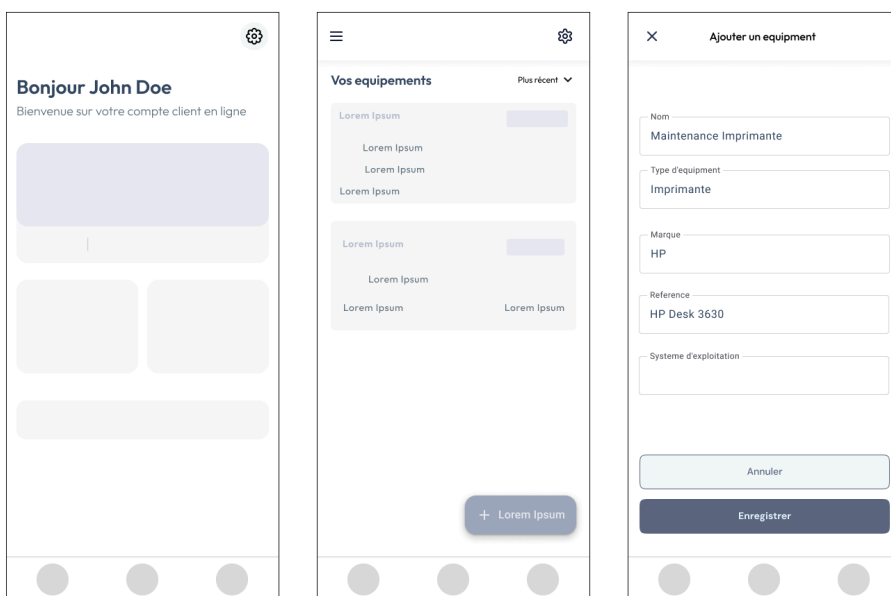


## Scénario 2 : Gestion d'Entreprise (validation manuelle)



## Maquettage sur Figma

Le processus que j'ai suivi pour le maquettage s'est articulé en plusieurs étapes successives, chacune apportant un niveau de précision supplémentaire. Nous avons commencé par créer des **wireframes basse-fidélité** pour définir rapidement la structure générale et l'organisation des informations sur l'écran.

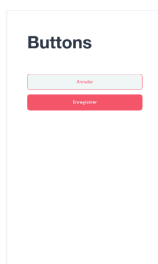


J'ai ensuite puis établir la **charte graphique**, qui définit l'identité visuelle du produit. Avec une couleur primaire et secondaire distinct, afin d'avoir un contraste assurant une bonne visibilité. La typographie définit les polices de caractères utilisées pour les titres, le texte courant et les éléments d'interface.

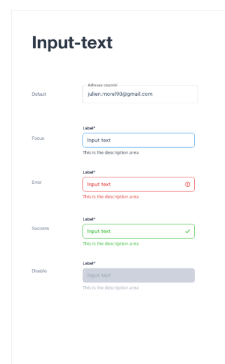


Grâce à la charte graphique, j'ai pu définir le **design system**, regroupant l'ensemble des composants d'interface réutilisables avec leurs variations et états. Il inclut tous les éléments récurrents : boutons, champs de saisie, cartes, modales, barres de navigation, sous forme de composant réutilisable.

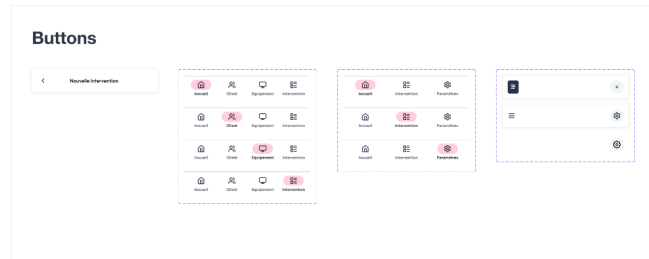
## Buttons



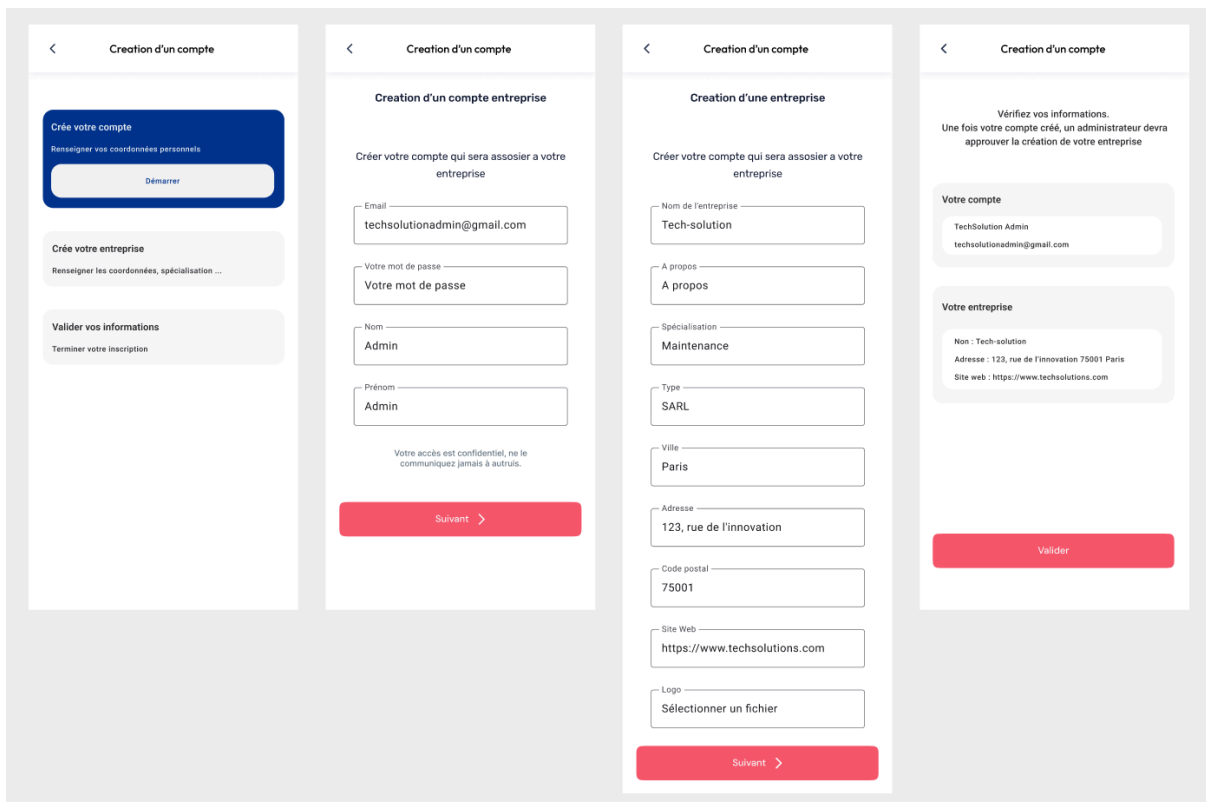
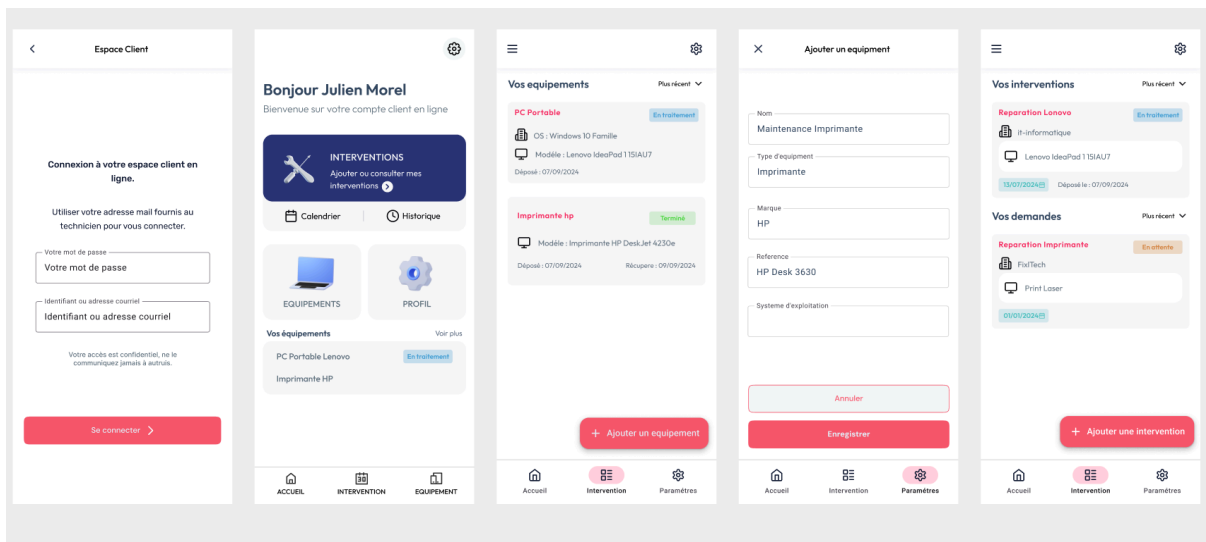
## Inputs



## Navbar components



La dernière étape fut la réalisation des **maquettes haute-fidélité**, intégrant tous les éléments du design system. Ces maquettes représentent l'interface exactement telle qu'elle apparaîtra dans l'application finale, avec les bonnes couleurs, les polices, les icônes et les textes. Elles montrent tous les écrans principaux de l'application, du tableau de bord à la création d'une intervention.



L'**accessibilité** a été un enjeu essentiel pour garantir que l'application soit utilisable par tous. Pour respecter ces critères, nous nous sommes appuyés sur le Référentiel Général d'Amélioration de l'Accessibilité (**RGAA**). Le choix des couleurs a joué un rôle essentiel dans l'accessibilité visuelle, en garantissant un contraste élevé entre les éléments textuels et les arrière-plans, un critère clé du RGAA.

Ces maquettes nous ont permis de formaliser les besoins utilisateurs et de valider les parcours fonctionnels avant le développement, tout en nous assurant de la conformité de l'application aux standards d'accessibilité.

# 4. Développement et Implémentation

## Environnement de Développement – Docker et Containerisation

Pour garantir un environnement de travail cohérent et isolé, nous avons choisi la conteneurisation avec Docker. L'orchestration des services est gérée par un fichier `docker-compose.yaml` où chaque service, comme le web-server Nginx et PHP-FPM, fonctionne dans un conteneur isolé au sein d'un même réseau Docker. L'architecture inclut également deux services de base de données PostgreSQL distincts, l'un pour le développement et l'autre pour les tests. Cette séparation des environnements garantit que les tests n'interfèrent jamais avec les données de développement. Un `Dockerfile` a été conçu pour construire une image Docker sur mesure, incluant toutes les dépendances et extensions nécessaires au bon fonctionnement de l'application Symfony et de Doctrine. Le Dockerfile suivant s'appuie sur l'image officielle `php:8.3-fpm`, installe les extensions requises pour PostgreSQL et configure Composer pour la gestion des dépendances. Un script de démarrage personnalisé permet de créer la base de données via les commandes CLI de Symfony lorsque celle-ci est prête.

```
FROM php:8.3-fpm

RUN apt-get update && apt-get install -y \
    # Install
    && echo 'alias sf="php bin/console"' >> ~/.bashrc

RUN docker-php-ext-configure gd --with-jpeg --with-freetype

RUN docker-php-ext-install \
    pdo pdo_pgsql pgsql zip xsl gd intl opcache exif mbstring

RUN curl -sS https://getcomposer.org/installer | php --
    --install-dir=/usr/local/bin --filename=composer

WORKDIR /var/www/symfony

# Copy startup script
COPY docker-entrypoint.sh /usr/local/bin/
RUN chmod +x /usr/local/bin/docker-entrypoint.sh
```

```
ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["php-fpm"]
```

Ce Dockerfile s'appuie sur l'image officielle `php:8.3-fpm`, installe les extensions requises pour PostgreSQL (`pdo_pgsql`) et configure Composer. Un script de démarrage personnalisé lance les commandes CLI de Symfony pour la création de la base de données une fois celle-ci disponible.

## Développement du backend de l'application

Pour le développement de notre API, nous avons fait le choix de nous appuyer sur les principes de la **Programmation Orientée Objet (POO)** afin de construire une base solide, modulaire et facile à maintenir. Le framework **Symfony**, idéal pour cette approche, a été adopté. Nous avons mis en place une architecture **MVC** (Modèle-Vue-Contrôleur) adaptée pour le pattern API-Only, où le **Modèle** est géré par les entités Doctrine qui intègrent la logique de l'application, la **Vue** est exclusivement générée en JSON grâce au Serializer de Symfony, et le **Contrôleur** agit comme l'orchestrateur, gérant les requêtes et renvoyant les réponses JSON.

L'intégration d'**API Platform** a été une étape clé, nous permettant de créer et d'exposer de manière élégante et rapide des opérations CRUD (Create, Read, Update, Delete) sur nos entités. Cette approche a non seulement accéléré le développement, mais nous a aussi laissé la flexibilité de personnaliser les points d'accès selon nos besoins métier. Pour respecter les bonnes pratiques d'une API REST, nous avons privilégié les requêtes efficaces avec des paramètres plutôt que des routes complexes, et nous respectons scrupuleusement les différents verbes HTTP.

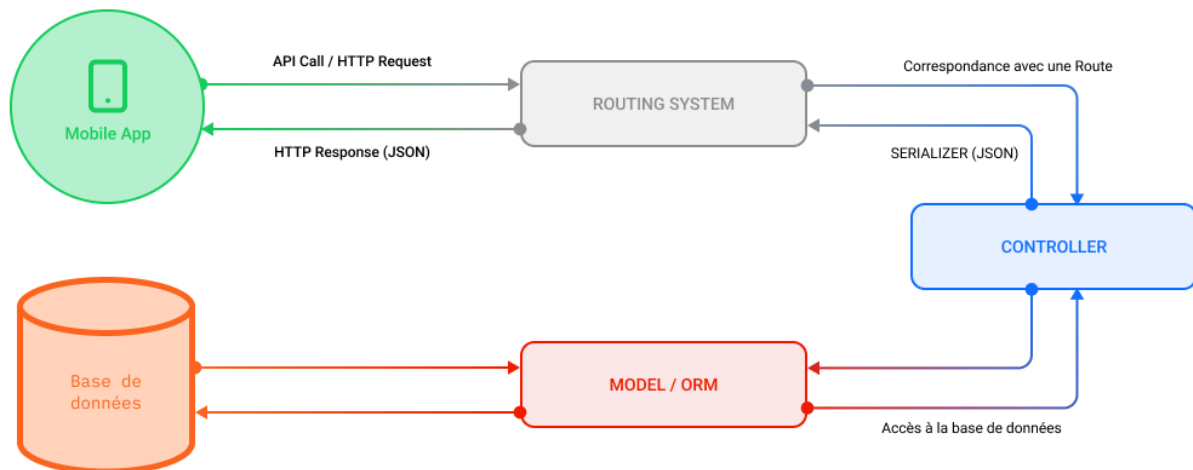
## Arborescence et Organisation

L'architecture de notre backend respecte une séparation rigoureuse des préoccupations, ce qui se reflète dans notre arborescence. Chaque dossier contient une couche de l'application avec un rôle bien défini, ce qui facilite la navigation et la collaboration. Les noms des classes et des méthodes sont conformes aux standards PSR-1/PSR-12 pour une cohérence optimale. Notre backend est donc composé des dossiers suivants :

- **Controller/** : Regroupe les contrôleurs, généralement un par ressource de l'API.
- **Service/** : Regroupe la logique métier complexe.
- **Entity/** : Contient les entités Doctrine qui représentent nos modèles de données, soit l'équivalent de nos tables de base de données.
- **Repository/** : Contient les repositories Doctrine, où sont implémentées les requêtes personnalisées d'accès aux données.
- **config/** : Rassemble les fichiers de configuration, notamment pour les routes, la sécurité, et API Platform.
- **migrations/** : Stocke les fichiers de migration de base de données, permettant d'appliquer et de versionner les changements de schéma.

- **tests/** : Destiné aux tests unitaires et fonctionnels, réalisés avec PHPUnit pour simuler les interactions via des requêtes HTTP.
- 

## Fonctionnement de l'API et Routage



Lorsqu'un client envoie une requête, le routeur de Symfony analyse l'URL et la méthode HTTP pour diriger la demande vers le contrôleur approprié. Ce contrôleur, via l'EntityManager de Doctrine, interagit avec la base de données à travers nos entités et repositories. Une fois la logique métier exécutée, une réponse est renvoyée au format JSON avec un code de statut approprié. Les statuts HTTP utilisés sont standardisés pour indiquer clairement le résultat de chaque requête :

- **200 OK** : La requête a réussi.
- **201 CREATED** : La ressource a été créée avec succès.
- **400 BAD REQUEST** : Erreur de syntaxe dans la requête.
- **401 UNAUTHORIZED** : Les informations d'authentification sont manquantes.
- **403 FORBIDDEN** : L'utilisateur n'a pas les droits nécessaires.
- **404 NOT FOUND** : La ressource demandée n'existe pas.
- **500 INTERNAL SERVER ERROR** : Une erreur interne est survenue sur le serveur.

Les méthodes HTTP sont utilisées de manière sémantique : **GET** pour la récupération, **POST** pour l'enregistrement, **PUT** et **PATCH** pour la mise à jour (totale et partielle), et **DELETE** pour la suppression de ressources. Pour déclarer les routes, j'ai tiré parti du système de routage puissant de Symfony via les attributs PHP. Cette approche rend le routage à la fois simple et facile à gérer. L'attribut `#[IsGranted('ROLE_ADMIN')]` agit comme un middleware qui vérifie les droits d'accès avant l'exécution du contrôleur.

```
<?php
#[Route('/api')]
```



```
final class CompanyController extends AbstractController
{
    #[IsGranted('ROLE_ADMIN')]
    #[Route('/company', name: 'app_company_post', methods: ['POST'])]
    public function create(Request $request): JsonResponse
    {
        // Logique de création d'entreprise
    }
}
```

## Contrôleurs

Dans les contrôleurs, on trouve les tâches de validation des données et d'orchestration des appels aux services. La logique métier complexe est déléguée aux services pour maintenir la séparation des responsabilités.

## Services

Afin de maintenir une bonne séparation des préoccupations, la logique métier complexe est encapsulée dans des classes dédiées appelées "**Services**". L'utilisation de cette approche est une excellente pratique de conception pour plusieurs raisons. Premièrement, elle applique le **Principe de Responsabilité Unique** : chaque service a une seule et unique responsabilité. Par exemple, un **CompanyManager** gère toutes les opérations liées à la création, à la modification et à la suppression des entreprises. Deuxièmement, cette architecture favorise la **réutilisabilité**. La logique métier d'un service peut être invoquée par différents contrôleurs ou par d'autres parties de l'application, évitant ainsi de dupliquer du code. Enfin, les services sont essentiels pour la **testabilité**. En isolant la logique métier de l'infrastructure (comme la gestion des requêtes HTTP), ils sont beaucoup plus faciles à tester unitairement. En conséquence, le code de nos contrôleurs reste concis. Il se concentre uniquement sur la gestion de la requête et de la réponse HTTP, déléguant au service le soin d'exécuter le cœur de la logique de l'application.

## ORM Doctrine

**Doctrine ORM** agit comme une couche d'abstraction essentielle entre l'application et la base de données. Il permet de manipuler les données sous forme d'objets PHP plutôt qu'en écrivant directement des requêtes SQL. L'utilisation de Doctrine nous apporte plusieurs avantages cruciaux : il renforce la sécurité en protégeant contre les injections SQL via des requêtes préparées, il facilite le développement en générant automatiquement les requêtes SQL à partir de nos manipulations d'objets, et il améliore la maintenabilité en gérant les changements de schéma via des migrations versionnées, ce qui rend le code indépendant du type de base de données. Nous utilisons une base relationnelle SQL, plus précisément le SGBD **PostgreSQL**. Ce choix a été fait pour sa stabilité, sa robustesse et sa scalabilité, ainsi que pour son excellente intégration native avec Symfony.

Doctrine est un **ORM** (Object-Relational Mapping), une couche plus haut niveau qui fait le lien entre le code objet, en PHP, et la base de données relationnelle. Les tables SQL sont traduites en objets PHP (appelés entités), ce qui nous permet de travailler uniquement avec ces objets sans avoir à écrire directement des requêtes SQL. L'entité PHP est automatiquement liée à une table et les opérations CRUD (Create, Read, Update, Delete) sont gérées par les méthodes de l'**EntityManager**. Cette abstraction procure un gain de temps significatif, sécurise le code via des requêtes paramétrées, et assure une meilleure maintenabilité. Les différentes requêtes SQL sont disponibles dans les migrations, qui sont des fichiers PHP générés à l'aide de la commande **symfony console doctrine:make:migration**. Ces fichiers contiennent le code SQL nécessaire pour mettre à jour la structure de la base de données afin qu'elle corresponde aux entités définies dans le code. Chaque fichier de migration est une classe avec trois méthodes principales : **up()**, qui applique les modifications, **down()** qui les annule, et **getDescription()** qui décrit la migration. Ce mécanisme nous offre un processus versionné, automatisé et sécurisé pour gérer les évolutions du schéma, évitant ainsi les erreurs liées aux manipulations manuelles. Cela garantit une parfaite cohérence entre la structure du code et la base de données, tout en facilitant le travail en équipe et le déploiement sur différents environnements.

## Entités

Les entités représentent la pierre angulaire de notre couche de persistance. Elles sont mappées directement à nos tables de base de données et suivent les bonnes pratiques d'encapsulation. Toutes les propriétés sont privées afin de protéger les données internes de l'objet. L'accès à ces propriétés se fait uniquement via des méthodes publiques appelées **getters** et **setters**, garantissant un meilleur contrôle sur la lecture et la modification des données. Pour garantir la cohérence et la fiabilité des données, des annotations de validation peuvent être ajoutées directement sur les propriétés. Par exemple, une annotation **#[UniqueEntity('email')]** garantit l'unicité de l'email dans la base de données avant même que Doctrine ne tente de le persister. L'exemple ci-dessous illustre la structure de l'entité Company.

```
<?php
#[ApiResponse]
#[ORM\Entity(repositoryClass: CompanyRepository::class)]
class Company
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $name = null;
```

```
#[ORM\ManyToMany(targetEntity: CompanySpecialization::class, inversedBy:
'companies')]
private Collection $specializations;

public function __construct()
{
    $this->specializations = new ArrayCollection();
}
}
```

## Repositories

Les repositories sont des classes dédiées à la persistance et à la récupération des données. Ils agissent comme un pont entre l'application et la base de données. Chaque entité a son propre repository, qui étend généralement `ServiceEntityRepository` de Doctrine. C'est dans ces classes que nous implémentons des méthodes pour des requêtes personnalisées plus complexes que les simples `find()` ou `findAll()`. Un repository nous permet par exemple de chercher une entreprise par son nom ou de récupérer une liste filtrée, en utilisant le **DQL** (Doctrine Query Language) ou le **Query Builder** de Doctrine. Le QueryBuilder génère automatiquement la requête SQL sécurisée correspondante. Cette séparation nous permet de maintenir la logique de requête dans une couche dédiée, loin des entités et des contrôleurs, pour une meilleure organisation du code.

## Sécurité de l'API

Les API sont un moyen d'accéder aux informations de la base de données, ainsi, il existe de nombreux risques :

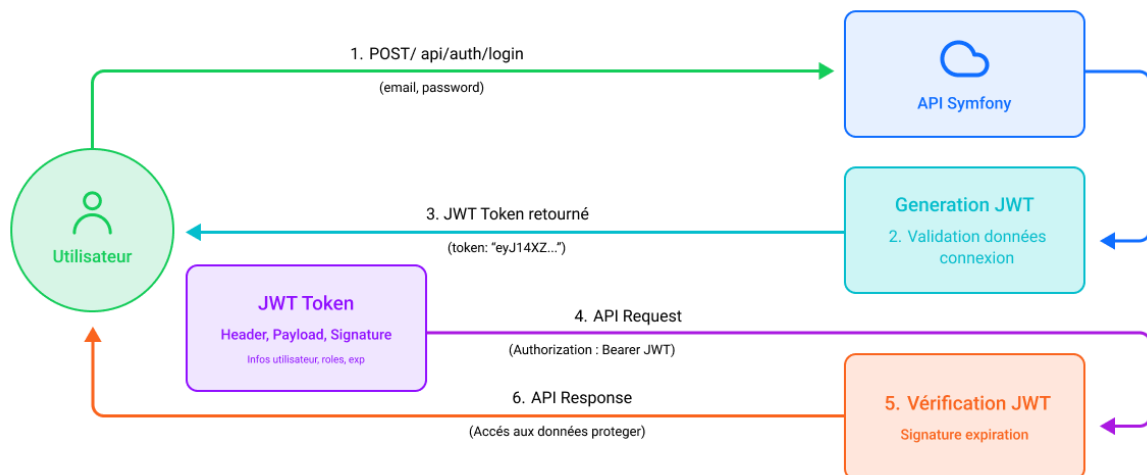
- **Injections SQL** : tentatives d'insertion de code malveillant dans les requêtes
- **Attaques par force brute** : tentatives répétées de connexion
- **Vol de tokens** : interception des jetons d'authentification
- **Attaques DDOS** : surcharge du serveur par un trafic massif
- **Man-in-the-middle** : interception des communications

Pour sécuriser mon API, j'ai mis en place plusieurs mesures de protection.

## Chiffrement des mots de passe et Authentification JWT

Les mots de passe des utilisateurs ne sont jamais stockés en clair. Pour les hacher de manière sécurisée, nous utilisons le composant `PasswordHasher` de Symfony Security Bundle, qui s'appuie par défaut sur l'algorithme `bcrypt` avec un coût élevé.

Afin d'effectuer une authentification sécurisée et **stateless**, nous avons intégré le bundle **LexikJWTAuthenticationBundle**.



Le **JWT** est un jeton qui permet l'échange des informations sur l'utilisateur de manière sécurisée. C'est une méthode de communication entre deux parties.

Ce jeton est composé de trois parties :

- Un **header** : identifie quel algorithme a été utilisé pour générer la signature
- Un **payload** : contient les informations de l'utilisateur (id, email, rôles) encodées en base64
- La **signature** : créée à partir du header, du payload et d'une clé secrète

```

{
  "iat": 1642678901,      // Issued at timestamp
  "exp": 1642682501,      // Expiration timestamp
  "roles": ["ROLE_ADMIN"], // Rôles utilisateur
  "username": "admin@company.com",
  "company": {
    "id": 42,
    "name": "TechService Pro"
  },
  "user": {
    "id": 15,
    "first_name": "Jean",
    "last_name": "Dupont"
  }
}

```

`AuthController` permettant de générer le token lors de la connexion :

```
<?php
#[Route('/api/auth/login', name: 'app_login', methods: ['POST'])]
public function login(Request $request, UserPasswordHasherInterface
$passwordHasher): JsonResponse
{
    $data = json_decode($request->getContent(), true);
    $email = $data['email'] ?? '';
    $password = $data['password'] ?? '';

    // Recherche de l'utilisateur
    $user =
$this->entityManager->getRepository(User::class)->findOneBy(['email' =>
$email]);

    if (!$user || !$passwordHasher->isPasswordValid($user, $password)) {
        return new JsonResponse(['error' => 'Invalid credentials'], 401);
    }

    // Le JWT sera généré automatiquement par LexikJWTAuthenticationBundle
    return new JsonResponse(['user' => $user->getEmail()]);
}
```

La clé secrète pour signer les JWT est stockée dans les variables d'environnement et ne doit jamais être partagée. Cette clé garantit l'intégrité du token.

## Configuration de sécurité (**security.yaml**)

Le fichier `config/packages/security.yaml` définit comment Symfony doit gérer l'authentification et l'autorisation pour chaque route :

```
security:
    password_hashers:
        App\Entity\User:
            algorithm: auto

    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email

    firewalls:
        api:
            pattern: ^/api
```

```

stateless: true
jwt: ~

access_control:
- { path: ^/api/auth, roles: PUBLIC_ACCESS }
- { path: ^/api/admin, roles: ROLE_ADMIN }
- { path: ^/api, roles: ROLE_USER }

```

Cette configuration :

- Définit le système de hachage des mots de passe.
- Configure le provider d'utilisateurs.
- Active l'authentification JWT pour toutes les routes `/api`.
- Définit les règles d'accès par rôle.

## Protection CORS

La configuration CORS est gérée via **NelmioCorsBundle** pour sécuriser les requêtes cross-origin tout en permettant l'accès depuis le frontend :

```

nelmio_cors:
  defaults:
    origin_regex: true
    allow_origin: ['%env(CORS_ALLOW_ORIGIN)%']
    allow_methods: ['GET', 'OPTIONS', 'POST', 'PUT', 'PATCH', 'DELETE']
    allow_headers: ['Content-Type', 'Authorization']
    expose_headers: ['Link']
    max_age: 3600

```

Cette configuration protège contre les attaques CSRF tout en autorisant les requêtes légitimes depuis le domaine du frontend.

## Problématiques Rencontrées : Gestion des créneaux disponibles

Initialement, pour la gestion des créneaux de disponibilité, j'avais opté pour une fonction PostgreSQL (`get_free_slots()`) directement intégrée à la base de données. Ce choix visait à optimiser les performances par l'exécution des calculs complexes au plus près des données, à garantir la cohérence logique et à simplifier l'appel depuis le backend.

Cependant, cette approche a rapidement révélé des limitations significatives lors du développement et des évolutions :

- **Complexité des tests** : La mise en place de tests unitaires pour une fonction de

base de données s'est avérée ardue, compromettant l'isolation et la fiabilité des tests automatisés.

- **Difficulté de débogage** : Les erreurs au sein de la fonction PostgreSQL étaient souvent cryptiques, rendant le débogage complexe et me sortant de l'environnement de développement PHP habituel.
- **Violation des principes architecturaux** : Plus fondamentalement, cette conception enfreignait le principe de **Séparation des Préoccupations (SoC)**, la logique métier résidant dans la couche de persistance. Le **Single Responsibility Principle (SRP)** était également compromis, la base de données assumant un double rôle.
- **Couplage fort et évolutivité limitée** : L'ajout de nouvelles fonctionnalités (comme le comptage des techniciens par rôle) nécessitait des modifications de la fonction PostgreSQL et donc des migrations de base de données. Ce couplage fort entre la logique applicative et la structure de données rendait les évolutions itératives complexes et limitait la scalabilité. Une "base de données intelligente" devenait un goulot d'étranglement.

## Solution Implémentée : Refactoring vers une Architecture Orientée Services

Face à ces défis, une décision de refactoring a été prise pour migrer l'intégralité de la logique de calcul des créneaux vers un service PHP dédié. Cette démarche visait à créer une architecture plus maintenable, testable et évolutive, en adhérant aux bonnes pratiques de développement logiciel.

Le refactoring a consisté à déplacer la logique de calcul vers un service PHP spécialisé, l'**AvailabilityService**. Cette nouvelle architecture respecte désormais une séparation claire des responsabilités :

- Le service orchestre les appels aux repositories pour récupérer les données.
- La logique métier est entièrement implémentée en PHP pur, exploitant la **Programmation Orientée Objet (POO)**.
- La base de données retrouve son rôle primaire de stockage et de récupération efficace des données, sans traitement métier complexe.

Cette approche a permis d'appliquer concrètement les principes **SOLID** (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion), notamment par l'utilisation de l'**Injection de Dépendances** pour une meilleure testabilité et modularité. Chaque méthode de l'**AvailabilityService** a une responsabilité précise (validation, récupération, génération, filtrage, formatage), facilitant la compréhension, la maintenance et l'évolution future du code.

```
<?php
class AvailabilityService
{
    public function getFreeSlots(
        \DateTime $date,
        ?int $companyId = null,
```

```

        int $intervalMinutes = 30,
        string $startTime = '09:00:00',
        string $endTime = '18:00:00',
        ?string $roleSearch = null
    ): array {
        // Logique métier entièrement en PHP
        $this->validateParameters($intervalMinutes, $startTime,
$endTime);
        $existingInterventions =
$this->getInterventionsForDate($date, $companyId);
        $availableTechnicians =
$this->getAvailableTechnicians($companyId, $roleSearch);
        $allSlots = $this->generateAllSlots($date, $startTime,
$endTime, $intervalMinutes);
        $freeSlots = $this->filterFreeSlots($allSlots,
$existingInterventions, $availableTechnicians);

        return $this->formatFreeSlots($freeSlots);
    }
}

```

## Documentation automatique avec API Platform

Étant donné que l'objectif principal d'une API est d'être consommée par d'autres développeurs, il est primordial de fournir une documentation technique complète et toujours à jour. Une documentation claire, avec des instructions précises sur le fonctionnement de l'API, facilite l'intégration et l'utilisation, et doit impérativement évoluer avec le code pour rester pertinente.

Pour répondre à ce besoin, nous avons utilisé **API Platform**, un framework qui permet de créer rapidement des APIs REST modernes et de générer automatiquement leur documentation au format **Swagger/OpenAPI**. Cette approche nous a permis d'obtenir un gain de temps considérable, car nous n'avons pas eu besoin de maintenir une documentation séparée. Chaque modification du code, qu'il s'agisse de l'ajout de nouvelles fonctionnalités ou d'ajustements des schémas de données, met à jour la documentation de manière automatique.

Pour parvenir à ce résultat, API Platform se base sur les annotations présentes dans nos entités. Les annotations telles que `#[ApiResponse]` et `#[Groups]` permettent non seulement d'exposer de manière élégante les endpoints CRUD, mais aussi de générer la documentation des schémas de données, de définir les champs qui sont lisibles ou modifiables, et de documenter les codes de réponse HTTP.

Le résultat de cette intégration est une interface graphique interactive, **Swagger UI**,



accessible via le chemin [/api/docs](#). Grâce à cette interface, les développeurs peuvent visualiser l'ensemble des endpoints de l'API et même les tester directement depuis leur navigateur, ce qui accélère considérablement le processus d'intégration. Cette approche garantit la cohérence entre le code et la documentation, le respect des standards OpenAPI 3.0 et une grande productivité.

# Développement Frontend – React Native

J'ai développé l'application mobile avec React Native et Expo Router. L'architecture suit une approche modulaire et organisée pour faciliter la maintenance et l'évolutivité.

```

  ▾ MOBILE
    > .expo
    ▾ app
      ▾ (auth)
        ✎ loginAdmin.tsx
        ✎ loginCustomer.tsx
        ✎ registerCustomer.tsx
      ▾ (main)
        ▾ admin
          > interventions
          ✎ index.tsx
          ✎ registerTech.jsx
        ▾ customer
          > equipment
          > equipments
          > intervention
          > interventions
          > settings
          ✎ _layout.tsx
          ✎ index.tsx
          ✎ _layout.tsx
        > assets
        ▾ components
          > appointment
          > auth
          > buttons
          > equipment
          > inputs
          > intervention
          > navigation
        > context
        > hooks
        > types
        > utils
        ✎ _layout.tsx
        ✎ index.tsx
        > node_modules
        > scripts
        > styles
    ✎ .env
```

(auth) : Écrans d'authentification (login admin, login customer, register)

(main) : Écrans principaux protégés par authentification, organisés par rôle (admin/customer)

components : Composants réutilisables triés par fonctionnalité (appointment, auth, buttons, equipment, etc.)

context : Gestion de l'état global (authentification, session)

hooks : Hooks personnalisés réutilisables

types : Définitions TypeScript pour le typage

utils : Fonctions utilitaires et configuration API

styles : Styles globaux du projet

## Navigation avec Expo Router

Pour la gestion de la navigation, nous avons opté pour **Expo Router**. Ce choix a été motivé par son approche intuitive de navigation basée sur le système de fichiers, directement inspirée de **Next.js**. Les fichiers `_layout.tsx` jouent un rôle crucial, agissant comme des points d'entrée qui définissent la structure commune pour un groupe de routes. Ils permettent de partager le code, de gérer de manière centralisée les logiques d'authentification et d'organiser la navigation de l'application, qu'il s'agisse de **Stack**, de **Tabs** ou d'un **Drawer**. Cette approche simplifie grandement la mise en place d'une navigation complexe et conditionnelle.

```
export default function RootLayout() {
  return (
    <SessionProvider>
      <Stack>
        <Stack.Screen name="index" options={{ headerShown: false }} />
        <Stack.Screen name="(auth)" options={{ headerShown: false }} />
        <Stack.Screen name="(main)" options={{ headerShown: false }} />
      </Stack>
    </SessionProvider>
  );
}
```

## Principe DRY avec les composants

Afin d'éviter la répétition de code et d'assurer une cohérence visuelle, nous avons appliqué le principe **DRY (Don't Repeat Yourself)** en développant des composants réutilisables.

Un exemple parfait de cette approche est notre composant **AppButton**, qui centralise la logique et le style de tous les boutons de l'application. En définissant une interface **AppButtonProps** qui étend les propriétés de **ButtonProps** de **react-native-paper**, nous avons pu créer différentes variations de boutons (primaire, secondaire, tertiaire) tout en garantissant un design uniforme.

```
import React from "react";
import {Button, ButtonProps} from "react-native-paper";
import {StyleSheet, TextStyle, ViewStyle} from "react-native";
interface AppButtonProps extends ButtonProps {
  type: "primary" | "secondary" | "tertiary";
}
export const AppButton: React.FC<AppButtonProps> = ({children, type, ...props}) => {
  const buttonStyle: ViewStyle = styles[type];
```

```

const textStyle: TextStyle = styles[`${type}Text`];

return (
  <Button style={buttonStyle} labelStyle={textStyle} {...props}>
    {children}
  </Button>
);
};

```

Cette méthode offre de multiples avantages, notamment une maintenance simplifiée (toute modification est centralisée), une cohérence visuelle grâce à un design system unifié, et une productivité accrue.

## Gestion de l'Authentification

L'authentification de l'utilisateur est gérée de manière centrale grâce à un `<Context Provider>`, qui encapsule toute la logique de session. Ce contexte permet de partager l'état d'authentification à travers toute l'application, en offrant des fonctions pour la connexion (`signIn`), la déconnexion (`signOut`) et la gestion du token de session.

Le `SessionProvider` interagit avec notre API pour vérifier les identifiants de l'utilisateur et stocker le JWT (token et refresh token) de manière sécurisée.

```

export function SessionProvider({ children }: PropsWithChildren) {
  const [[isLoading, session], setSession] = useStorageState("session");

  return (
    <AuthContext.Provider
      value={{
        signIn: async (email: string, password: string) => {
          // Appel API de connexion
          const resp = await fetch(`${apiUrl}/auth/login`, {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ email, password }),
          });

          const data = await resp.json();
          const sessionData = JSON.stringify({
            token: data.token,
            refreshToken: data.refresh_token,
          });

          setSession(sessionData);
          return true;
        },
        signOut: () => {

```

```

        setSession(null);
        router.replace("/");
      },
      session,
      isLoading,
    )}
  >
  {children}
</AuthContext.Provider>
);
}

```

La protection des routes s'effectue de manière automatique via le layout, qui vérifie la présence d'une session valide avant d'autoriser l'accès aux écrans principaux.

## Gestion des appels API

L'ensemble des appels à l'API est centralisé par un hook personnalisé `useApi()`. Cet outil se charge de la configuration des requêtes avec des fonctionnalités automatiques :

- L'ajout automatique du JWT token à chaque requête.
- Le refresh automatique du token en cas d'expiration.
- La gestion d'erreur centralisée.

Cette approche simplifie considérablement le développement et garantit que chaque requête envoyée est correctement authentifiée.

```

api.interceptors.request.use(async (config) => {
  const sessionStr = await getStoredSession();
  if (sessionStr) {
    const session = JSON.parse(sessionStr);
    if (session.token) {
      config.headers.Authorization = `Bearer ${session.token}`;
    }
  }
  return config;
});

```

## Validation de Formulaires et Sécurité Client

Pour renforcer la sécurité de l'application, nous avons mis en place une couche de validation des formulaires côté client. J'ai utilisé la bibliothèque **react-hook-form** qui offre une gestion avancée des erreurs et une validation des données en amont de l'envoi au serveur. L'outil **Controller** permet de définir les règles de validation (champs requis, format d'email, etc.). Si une des règles n'est pas respectée, un message d'erreur est affiché à l'utilisateur et le bouton de soumission du formulaire est désactivé, empêchant l'envoi de données invalides à l'API.

```

import {useForm, Controller} from 'react-hook-form';

const LoginForm = () => {
  const {control, handleSubmit, formState: {errors}} = useForm();

  const onSubmit = (data) => {
    // Envoi des données validées
  };

  return (
    <View>
      <Controller
        control={control}
        name="email"
        rules={{
          required: 'Email requis',
          pattern: {
            value: /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i,
            message: 'Format email invalide'
          }
        }}
        render={({field}) => (
          <TextInput
            placeholder="Email"
            value={field.value}
            onChangeText={field.onChange}
            onBlur={field.onBlur}
          />
        )}
      />
      {errors.email && <Text
style={styles.error}>{errors.email.message}</Text>}

      <AppButton type="primary" onPress={handleSubmit(onSubmit)}>
        Se connecter
      </AppButton>
    </View>
  );
};

```

Cette approche garantit une meilleure expérience utilisateur et réduit la charge de travail côté serveur.

# 6. Qualité, Tests et Déploiement

## Plan de tests fonctionnels

J'ai mis en place un plan de tests complet pour valider le bon fonctionnement de l'API. Ce plan couvre les différents aspects de l'application : fonctionnalités principales, gestion des erreurs, sécurité et performance, à l'aide du framework de test PHPUnit, qui offre un environnement complet pour l'écriture et l'exécution de tests unitaires et d'intégration.

## Tests Unitaires

Les tests unitaires vérifient le comportement de chaque composant en isolation. Ils testent les entités, les services métier, et les utilities sans dépendance externe. Exemple avec l'entité Company :

```
<?php
namespace App\Tests\Unit\Entity;

use App\Entity\Company;
use PHPUnit\Framework\TestCase;

class CompanyTest extends TestCase
{
    public function testCompanyCreation(): void
    {
        $company = new Company();
        $company->setName('TechService Pro');
        $company->setType('Informatique');

        $this->assertEquals('TechService Pro', $company->getName());
        $this->assertEquals('Informatique', $company->getType());
        $this->assertInstanceOf(\DateTimeImmutable::class,
            $company->getCreatedAt());
    }

    public function testCompanyNameValidation(): void
    {
        $company = new Company();

        $this->expectException(\InvalidArgumentException::class);
        $company->setName(''); // Nom vide non autorisé
    }
}
```

## Tests d'Intégration API

Les tests d'intégration vérifient le comportement de l'API complètent en simulant des requêtes HTTP réelles. Ils utilisent un client de test Symfony et une base de données de test

isolée :

```
<?php
namespace App\Tests\Controller;

use App\Tests\ApiTestCase;
use Symfony\Component\HttpFoundation\Response;

class CompanyControllerTest extends ApiTestCase
{
    public function testCreateCompany(): void
    {
        $this->loginAsAdmin();

        $client = static::createClient();
        $client->request('POST', '/api/company', [], [], [
            'CONTENT_TYPE' => 'application/json',
        ], json_encode([
            'name' => 'Nouvelle Entreprise',
            'type' => 'Services'
        ]));

        $this->assertEquals(Response::HTTP_CREATED,
        $client->getResponse()->getStatusCode());

        $responseData = json_decode($client->getResponse()->getContent(), true);
        $this->assertEquals('Nouvelle Entreprise', $responseData['name']);
    }

    public function testCreateCompanyUnauthorized(): void
    {
        $client = static::createClient();
        $client->request('POST', '/api/company', [], [], [
            'CONTENT_TYPE' => 'application/json',
        ], json_encode(['name' => 'Test']));

        $this->assertEquals(Response::HTTP_UNAUTHORIZED,
        $client->getResponse()->getStatusCode());
    }
}
```

La classe `ApiTestCase` fournit des méthodes utilitaires pour l'authentification et la préparation des données de test :

```
<?php
abstract class ApiTestCase extends WebTestCase
{
    protected EntityManagerInterface $entityManager;
    protected JWTTokenManagerInterface $jwtManager;

    protected function setUp(): void
    {
```



```

        $kernel = self::bootKernel();
        $this->entityManager =
$kernel->getContainer()->get('doctrine')->getManager();
        $this->jwtManager =
$kernel->getContainer()->get('lexik_jwt_authentication.jwt_manager');
    }

    protected function loginAsAdmin(): string
    {
        $user = $this->entityManager->getRepository(User::class)->findOneBy(['email'
⇒ 'admin@test.com']);
        if (!$user) {
            $user = new User();
            $user->setEmail('admin@test.com');
            $user->setRoles(['ROLE_ADMIN']);
            $this->entityManager->persist($user);
            $this->entityManager->flush();
        }

        $token = $this->jwtManager->create($user);
        static::createClient()->setServerParameter('HTTP_AUTHORIZATION',
sprintf('Bearer %s', $token));

        return $token;
    }
}

```

## Cahier de Recette

Le cahier de recette définit les scénarios de test fonctionnels validant les exigences métier. Il structure les tests selon les rôles utilisateur et les fonctionnalités principales :

Fonctionnalité	Rôle	Scénario	Résultat Attendu
Authentification	Tous	Connexion avec identifiants valides	Token JWT retourné, code 200
Authentification	Tous	Connexion avec identifiants invalides	Erreur retournée, code 401
Gestion Entreprises	Admin	Création nouvelle entreprise	Entreprise créée, code 201
Gestion Entreprises	Technicien	Tentative création entreprise	Accès refusé, code 403
Gestion Interventions	Technicien	Création intervention	Intervention créée, code 201

Gestion Interventions	Client	Consultation ses interventions	Liste des interventions, code 200
Gestion Interventions	Client	Consultation interventions autres	Accès refusé, code 403

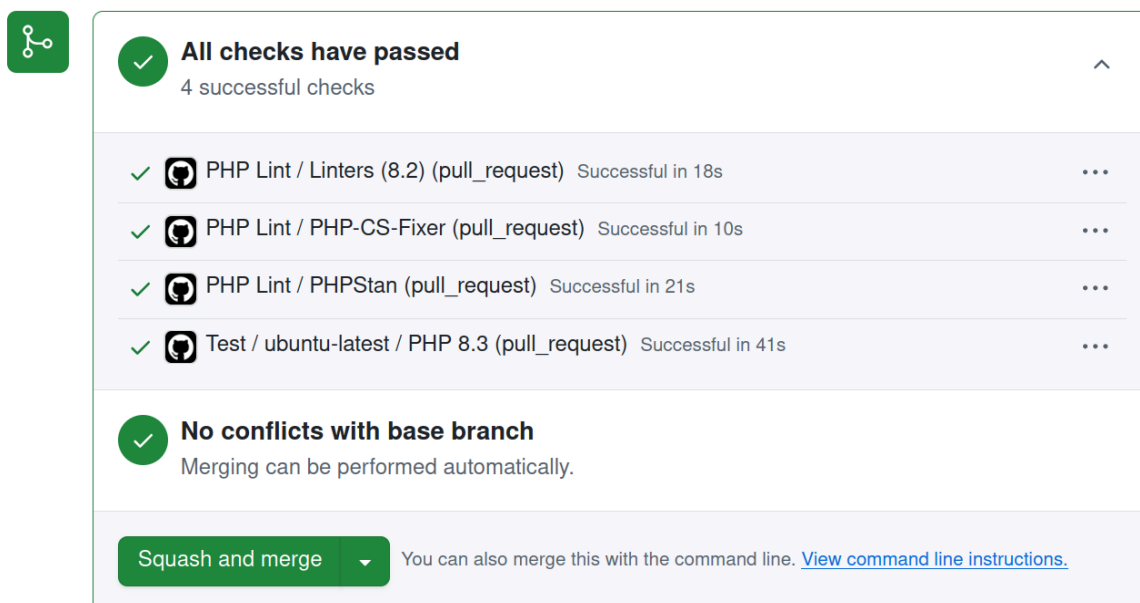
```

1  Run tests
2  Run vendor/bin/phpunit
3  PHPUnit 9.6.22 by Sebastian Bergmann and contributors.
4
5  Testing
6  ..... 49 / 49 (100%)
7
8  Time: 00:01.296, Memory: 48.50 MB
9
10 OK (49 tests, 143 assertions)
11
12 Remaining indirect deprecation notices (259)
13
14 258x: Relying on non-optimal defaults for ID generation is deprecated, and IDENTITY
15 results in SERIAL, which is not recommended.
16 Instead, configure identifier generation strategies explicitly through
17 configuration.
18 We currently recommend "SEQUENCE" for "Doctrine\DBAL\Platforms\PostgreSQLPlatform", when using DBAL 3,
19 and "IDENTITY" when using DBAL 4,
20 so you should probably use the following configuration before upgrading to DBAL 4,
21 and remove it after deploying that upgrade:
22
23 $configuration->setIdentityGenerationPreferences([
24     "Doctrine\DBAL\Platforms\PostgreSQLPlatform" => ClassMetadata::GENERATOR_TYPE_SEQUENCE,
25 ]);
26 (ClassMetadataFactory.php:635 called by ClassMetadataFactory.php:546, https://github.com/doctrine/orm/issues/8893, package doctrine/orm)
27 24x in CompanyControllerTest::testCreateCompany from App\Tests\Controller
28 21x in InterventionControllerTest::testCreateIntervention from App\Tests\Controller
29 20x in CustomerControllerTest::testCustomerCannotJoinCompany from App\Tests\Controller
30 19x in InterventionControllerTest::testAddTaskToIntervention from App\Tests\Controller
31 19x in InterventionControllerTest::testRemoveTaskFromIntervention from App\Tests\Controller
32 ...
33
34 1x: Since gesdinet/jwt-refresh-token-bundle 1.0: The "Gesdinet\JWTRefreshTokenBundle\Entity\AbstractRefreshToken" class is deprecated, use
35 "Gesdinet\JWTRefreshTokenBundle\Model\AbstractRefreshToken" instead.
36 1x in CompanyControllerTest::testCreateCompany from App\Tests\Controller
37
41

```

## Intégration Continue (CI) et Qualité du Code

L'intégration continue est gérée par **GitHub Actions** via deux pipelines distincts. Le fichier **lint.yaml** est dédié à la qualité du code, en imposant le respect du standard **PSR-12** avec l'outil **PHP-CS-Fixer** et en réalisant une analyse statique approfondie avec **PHPStan** au niveau 8, le plus strict. Le second pipeline, **test.yaml**, exécute l'intégralité des tests PHPUnit dans un conteneur Docker à chaque **push** ou **pull request**, garantissant ainsi qu'aucune régression n'est introduite. Ces pipelines fournissent un retour immédiat aux développeurs, empêchant la fusion de code non conforme aux standards



The image shows a GitHub Actions workflow status interface. At the top, a green checkmark icon is followed by the text "All checks have passed" and "4 successful checks". Below this, a list of four checks is shown, each with a green checkmark, a GitHub Actions icon, and details: "PHP Lint / Linters (8.2) (pull\_request) Successful in 18s", "PHP Lint / PHP-CS-Fixer (pull\_request) Successful in 10s", "PHP Lint / PHPStan (pull\_request) Successful in 21s", and "Test / ubuntu-latest / PHP 8.3 (pull\_request) Successful in 41s". Each check has a three-dot menu icon to its right. Below the list, a green checkmark icon is followed by the text "No conflicts with base branch" and "Merging can be performed automatically.". At the bottom, there is a green button labeled "Squash and merge" with a dropdown arrow, and a link that says "You can also merge this with the command line. [View command line instructions.](#)".

Pour aller plus loin dans l'analyse de la qualité du code, une prochaine étape pourrait être l'intégration de **SonarQube**. Cet outil permet d'analyser le code source pour détecter les bugs, les "code smells" et les vulnérabilités de sécurité. Il se complémente avec PHPUnit et Xdebug pour générer un rapport de couverture de code via la commande `XDEBUG_MODE=coverage ./vendor/bin/phpunit --coverage-text`. Ce rapport serait ensuite lu par SonarQube lors d'un scan pour afficher des métriques précises et donner des conseils sur la manière de résoudre les problèmes. L'objectif serait d'atteindre un taux de couverture de 80 %, une norme de qualité élevée en entreprise.

## Préparation au Déploiement et Déploiement Continu (CD)

Pour une éventuelle mise en production, une procédure de déploiement a été documentée. Les étapes de base consistent à transférer le code source sur le serveur, installer les dépendances de production avec `composer install --no-dev --optimize-autoloader`, lancer les migrations pour mettre à jour la base de données, et enfin vider le cache avec la commande `APP_ENV=prod APP_DEBUG=0 php bin/console cache:clear`. D'autres tâches spécifiques peuvent être nécessaires, comme l'ajout de tâches CRON. Les stratégies de déploiement peuvent varier, du transfert manuel via FTP à l'utilisation d'outils de gestion de version comme Git ou d'un PaaS tel que Platform.sh, recommandé pour les applications Symfony.

La mise en place d'une démarche de **Déploiement Continu (CD)** prolonge naturellement le processus d'intégration continue. Une fois que le pipeline de CI a validé la qualité et le bon fonctionnement du code, un job de CD se déclenche automatiquement. Ce pipeline pourrait inclure des étapes telles que la création d'une image Docker de l'application, son envoi vers un registre d'images, puis une connexion sécurisée au serveur de production (via SSH) pour y extraire la nouvelle image et relancer l'application. Cette automatisation permet un déploiement rapide et fiable, réduisant le risque d'erreurs humaines.

## Veille Technologique et Sécurité

Une veille technologique continue est essentielle pour maintenir le projet à jour et sécurisé. Pour cela, je consulte régulièrement les blogs et les documentations officielles de Symfony, ainsi que des sources d'information fiables sur PHP et son écosystème, comme [php.net](#), Stack Overflow, Reddit (r/PHP, r/Symfony) et SymfonyCasts.

Concernant la sécurité, une surveillance des **CVE** (Common Vulnerabilities and Exposures) pour les dépendances de l'application (Symfony, Doctrine, etc.) est primordiale. L'outil [composer audit](#) permet d'automatiser cette vérification. De plus, j'ai suivi les recommandations de l'**OWASP**, en particulier les points d'attention de leur Top 10, pour garantir que l'application respecte les meilleures pratiques en matière de sécurité.

# Conclusion

Pour conclure, la réalisation du projet Proxifix a été une expérience d'apprentissage particulièrement enrichissante et formatrice. Ce projet, mené sur **6 mois**, m'a permis d'approfondir mes compétences techniques en développement d'applications mobiles (React Native, JavaScript) et backend (Symfony, PHP, PostgreSQL). L'intégration d'outils comme PHPUnit et GitHub Actions a été clé pour découvrir et mettre en pratique des méthodologies de développement modernes et robustes.

Ce projet m'a appris à structurer et organiser mon travail de manière autonome, et à gérer les défis techniques et architecturaux, comme le refactoring de la gestion des créneaux. J'ai pu appliquer concrètement des principes fondamentaux du génie logiciel tels que **SOLID**, la **Séparation des Préoccupations** et le **Single Responsibility Principle**.

Finalement, ce projet m'a démontré ma capacité à m'adapter aux différentes situations, à résoudre des problématiques complexes et à mener à bien un projet de bout en bout.

# Annexe

docker-compose.yml

services:

database:

image: postgres:\${POSTGRES\_VERSION:-15}-alpine

environment:

POSTGRES\_DB: \${DB\_DEV\_NAME:-cda\_app}

POSTGRES\_PASSWORD: \${DB\_DEV\_PASSWORD:-password}

POSTGRES\_USER: \${DB\_DEV\_USER:-cda\_app}

networks:

- symfony

volumes:

- database\_data:/var/lib/postgresql/data:rw

ports:

- "\${DB\_DEV\_PORT:-5434}:5432"

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U \${DB\_DEV\_USER:-cda\_app}"]

interval: 10s

timeout: 5s

retries: 5

database\_test:

image: postgres:\${POSTGRES\_VERSION:-15}-alpine

environment:

POSTGRES\_DB: \${DB\_TEST\_NAME:-cda\_app\_test}

POSTGRES\_PASSWORD: \${DB\_TEST\_PASSWORD:-password}

POSTGRES\_USER: \${DB\_TEST\_USER:-cda\_app}

networks:

- symfony

volumes:

- database\_test\_data:/var/lib/postgresql/data:rw

ports:

- "\${DB\_TEST\_PORT:-5433}:5432"

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U \${DB\_TEST\_USER:-cda\_app}"]

interval: 10s

timeout: 5s

retries: 5

php:

build: ./php

volumes:

```

    - ../backend:/var/www/symfony
networks:
  - symfony
environment:
  DATABASE_URL:
    "postgresql://${DB_DEV_USER:-cda_app}:${DB_DEV_PASSWORD:-password}@database
:5432/${DB_DEV_NAME:-cda_app}?serverVersion=15&charset=utf8"
  DATABASE_TEST_URL:
    "postgresql://${DB_TEST_USER:-cda_app}:${DB_TEST_PASSWORD:-password}@databa
se_test:5432/${DB_TEST_NAME:-cda_app_test}?serverVersion=15&charset=utf8"
  APP_ENV: ${APP_ENV:-dev}
depends_on:
  database:
    condition: service_healthy
  database_test:
    condition: service_healthy

```

```

nginx:
  build: ./nginx
  ports:
    - "${NGINX_PORT:-80}:80"
  volumes:
    - ../backend:/var/www/symfony
  networks:
    - symfony
  depends_on:
    - php

```

```

networks:
  symfony:

```

```

volumes:
  database_data:
  database_test_data:

```

test.yaml

```

name: Test

```

```

on:
  push:
    branches: ["main", "dev"]
  pull_request:
    branches: ["main", "dev"]

```

```

permissions:

```

```

contents: read

jobs:
  test:
    name: "${{ matrix.operating-system }} / PHP ${{
matrix.php-version }}"
    runs-on: "${{ matrix.operating-system }}"
    continue-on-error: false
    defaults:
      run:
        working-directory: ./backend

services:
  postgres:
    image: postgres:15
    env:
      POSTGRES_PASSWORD: password
      POSTGRES_USER: postgres
      POSTGRES_DB: proaxive_test
    ports:
      - 5432:5432
    options: >-
      --health-cmd pg_isready
      --health-interval 10s
      --health-timeout 5s
      --health-retries 3

strategy:
  matrix:
    operating-system: ["ubuntu-latest"]
    php-version: ["8.3"]

steps:
  - name: "Checkout code"
    uses: actions/checkout@v4

  - name: "Install PHP with extensions"
    uses: shivammathur/setup-php@v2
    with:
      coverage: "none"
      extensions: "intl, mbstring, pdo_pgsql, zip"
      php-version: "${{ matrix.php-version }}"
      tools: composer:v2

  - name: "Add PHPUnit matcher"
    run: echo "::add-matcher::${{ runner.tool_cache
}}/phpunit.json"

```



```

- name: "Install dependencies"
  run: composer install --ansi --no-interaction --no-progress
--no-scripts

- name: "Generate JWT keys"
  run: |
    mkdir -p config/jwt
    openssl genpkey -out config/jwt/private.pem -algorithm
rsa -aes256 -pass pass:proaxive_jwt_passphrase
    openssl pkey -in config/jwt/private.pem -passin
pass:proaxive_jwt_passphrase -out config/jwt/public.pem -pubout

- name: "Prepare test database"
  run: |
    mkdir -p var
    cp .env.test.ci .env.test.local
    php bin/console doctrine:database:create --env=test
--if-not-exists
    php bin/console doctrine:migrations:migrate --env=test
--no-interaction
    php bin/console doctrine:fixtures:load --env=test
--no-interaction

- name: "Run tests"
  run: vendor/bin/phpunit

```