

# Dossier de Projet professionnel



## SafeBase

Lambert Julie

<b>Introduction.....</b>	<b>3</b>
Contexte de la formation.....	3
Présentation personnelle.....	3
Présentation du projet – SafeBase.....	4
Objectifs du projet.....	5
Organisation du dossier.....	6
<b>Partie 1 – Conception.....</b>	<b>7</b>
1. Analyse des besoins et cahier des charges.....	7
2. Modélisation de la base de données (MCD, MLD, MPD).....	8
3. Conception des interfaces utilisateur avec Figma.....	11
Wireframes (basse fidélité).....	11
Maquettes finales (haute fidélité).....	12
4. Design system et design pattern.....	12
Design Pattern MVC.....	14
5. Choix techniques.....	15
Laravel 12.....	15
Docker.....	16
MySQL & PostgreSQL.....	17
Autres outils.....	17
7. Fonctionnalités principales.....	17
<b>Partie 2 – Développement.....</b>	<b>18</b>
1. Développement Front-End.....	18
a. Authentification et rôles utilisateurs.....	18
b. Interface de connexion et tableau de bord.....	18
c. Gestion des connexions aux bases de données.....	18
d. Lancement et restauration des sauvegardes.....	19

---

e. Utilisation de Blade et Tailwind CSS.....	19
f. Sécurité du front-end.....	19
g. Tests front-end avec PHPUnit.....	21
h. Perspectives d'amélioration.....	23
b. Développement Back-End.....	23
ORM Eloquent et logique métier.....	23
Routage Laravel et middlewares.....	25
Sauvegarde manuelle et restauration.....	25
Sauvegarde automatique avec CRON.....	26
Sécurité back-end renforcée.....	26
Sécurité des données sensibles.....	27
a. Hachage des mots de passe utilisateurs.....	27
b. Chiffrement des mots de passe de connexion aux BDD externes.....	27
c. Clé d'application et variables sensibles.....	27
Tests back-end.....	28
Compétences couvertes (Back-End).....	28
Partie 3 – Intégration continue et déploiement continu (CI/CD).....	28
1. Présentation générale du pipeline CI/CD.....	29
2. Étapes du pipeline d'intégration continue (CI).....	31
a. Récupération du code.....	31
b. Configuration de l'environnement PHP.....	31
c. Installation des dépendances PHP.....	31
d. Configuration de Node.js et installation des dépendances front-end.....	32
e. Compilation des assets avec Vite.....	32
f. Préparation d'un environnement de test.....	32
g. Génération de la clé de l'application.....	33

---

h. Lancement des tests automatisés.....	33
3. Étapes du pipeline de déploiement continu (CD).....	34
a. Connexion au registre GitHub Container Registry (GHCR).....	34
b. Construction de l'image Docker.....	34
c. Publication de l'image.....	34
4. Intérêt de cette stratégie CI/CD pour SafeBase.....	35
5. Améliorations envisagées.....	35
Partie 4 – Mise en production, sécurité et documentation technique.....	36
1. Mise en production de l'application.....	36
a. Création d'une image Docker prête à l'emploi.....	36
b. Déploiement avec Docker Compose.....	36
c. Accès au service.....	37
2. Sécurité de l'application.....	37
b. Données utilisateurs et protection des mots de passe.....	37
c. Protection des routes sensibles.....	37
d. Vulnérabilités connues et mises à jour.....	37
3. Documentation technique.....	38
a. Documentation utilisateur.....	38
b. Arborescence du projet.....	38
c. Bonnes pratiques.....	38
Conclusion et retour d'expérience.....	39

---

## Introduction

### Présentation personnelle

Je m'appelle **Julie Lambert**, j'ai **34 ans** et je suis actuellement en **3e année de Bachelor** à l'école **La Plateforme** à Marseille. Ce parcours s'inscrit dans le cadre d'une **reconversion professionnelle**, guidée par l'envie d'exercer un métier stimulant, concret et en phase avec mes centres d'intérêt.

Depuis **2 ans**, j'effectue mon **alternance au sein de la Carsat Sud-Est**, en tant que **développeur web**. Cette expérience en entreprise m'a permis de travailler sur des projets concrets, d'interagir avec des équipes pluridisciplinaires (MOA, utilisateurs finaux...), et de développer mon autonomie ainsi que ma rigueur professionnelle.

Au fil de ma formation, j'ai consolidé mes compétences techniques autour de **PHP**, **JavaScript**, **SQL**, des frameworks comme **Laravel**, et des outils comme **Docker**, **GitHub Actions**, ou encore **Figma**. J'ai appris à structurer mes projets selon une **architecture logicielle claire**, à **adopter les bonnes pratiques de sécurité**, à suivre une **démarche agile**, et à **produire une documentation de qualité**.

Le projet **SafeBase**, que j'ai imaginé et développé intégralement, constitue une **illustration concrète de cette montée en compétence**. Il m'a permis d'aborder toutes les dimensions du métier de conceptrice développeuse : de la conception UML au développement backend, en passant par les tests, l'automatisation CI/CD, et le déploiement.

Aujourd'hui, je suis pleinement investie dans ce métier. Mon objectif est de **continuer à évoluer au sein d'équipes de développement**, sur des projets exigeants, où je pourrai mettre mes compétences au service de **solutions fiables, innovantes et sécurisées**.

## Contexte de la formation

Dans le cadre du diplôme **Concepteur Développeur d'Applications**, il nous a été demandé de concevoir, développer et documenter une application répondant à un besoin fonctionnel précis.

Ce projet doit permettre de mettre en œuvre l'ensemble des compétences techniques, méthodologiques et organisationnelles acquises durant la formation. Il constitue une étape essentielle de l'évaluation, en lien direct avec les référentiels Front, Back, DevOps et CI/CD.

## Compétences visées par le projet

Activité-type	Compétences couvertes	Réalisation dans SafeBase
AT1 – Développer une application sécurisée	♦ Installer l'environnement ♦ Développer des interfaces ♦ Développer des composants métier	Laravel 12, UI Blade, routes, middleware, authentification
AT2 – Concevoir et développer une application en couches	♦ Maquettage ♦ Conception base de données ♦ Architecture logicielle (MVC) ♦ Accès aux données SQL	MCD/MLD/MPD, design en couches, Repository, Eloquent ORM
AT3 – Préparer le déploiement d'une application sécurisée	♦ Tests unitaires ♦ CI/CD ♦ Dockerisation & publication	PHPUnit, GitHub Actions, Dockerfile, GHCR
AT4 – Travailler en mode projet	♦ Gestion du projet ♦ Documentation ♦ Collaboration	Kanban Trello, README.md, dossier professionnel

---

## Présentation du projet – *SafeBase*

Le projet que j'ai choisi de développer s'intitule **SafeBase**. Il s'agit d'une **plateforme web sécurisée** permettant aux utilisateurs de **sauvegarder et restaurer des bases de données** (MySQL et PostgreSQL) de manière simple, automatisée et centralisée. L'application propose une interface utilisateur développée avec le framework Laravel 12, un **système d'authentification avec rôles (utilisateur/admin)**, la **gestion des connexions à plusieurs bases de données**, un système de **versionnement des sauvegardes**, ainsi que des **notifications d'alerte** en cas d'erreur. L'application est conteneurisée avec **Docker**, et intègre un pipeline **CI/CD automatisé** avec **GitHub Actions** et **GitHub Container Registry**.

### Objectifs du projet

Ce projet avait pour but de mettre en œuvre des compétences avancées dans les domaines suivants :

- **Développement back-end** avec Laravel : architecture MVC, routes, contrôleurs, services, tests automatisés.
- **Gestion de bases de données** : connexions multiples, commandes artisan personnalisées pour les sauvegardes/restaurations, sécurité des accès.
- **Déploiement et conteneurisation** : création de conteneurs Docker pour l'application, la base de données et phpMyAdmin.
- **Automatisation et CI/CD** : exécution de tests automatisés et déploiement d'images Docker dans GitHub Container Registry via GitHub Actions.
- **Approche DevOps** : gestion des environnements, configuration de **.env** spécifiques pour la production ou la CI, journalisation, et documentation.

Ce projet m'a également permis de développer mon autonomie, ma capacité à **documenter le code et les procédures**, et à **travailler avec des outils de versioning** (Git/GitHub) dans un cadre structuré.

---

## Organisation du dossier

Ce dossier est organisé par blocs de compétences, en lien direct avec les référentiels du titre professionnel. Il est structuré comme suit :

- Partie 1 : Conception
- Partie 2 : Développement
- Partie 3 : Intégration continue et déploiement continu (CI/CD)
- Partie 4 : Mise en production, sécurité et documentation technique



---

## Partie 1 – Conception

Le développement front-end sécurisé de l'application SafeBase repose sur une démarche structurée et rigoureuse, allant de la conception à l'implémentation. Cette section décrit en détail les étapes de conception de l'interface utilisateur, les choix techniques et les outils utilisés pour assurer à la fois la qualité visuelle et la sécurité du front-end de l'application.

### 1. Analyse des besoins et cahier des charges

Avant de débiter le développement, un **cahier des charges fonctionnel** a été rédigé. Il a permis de définir précisément les besoins de l'application en termes de fonctionnalités, de sécurité, et d'ergonomie. Il comporte notamment :

- **Contexte et justification du projet** : expliquer pourquoi l'application est nécessaire dans un environnement professionnel.
- **Objectifs fonctionnels** : sauvegarder et restaurer des bases MySQL et PostgreSQL, journaliser les opérations, permettre une gestion multi-utilisateur avec rôles.
- **Contraintes techniques** : application développée avec Laravel, interface responsive, sécurité par authentification et rôles, Docker pour l'environnement.
- **Délais de réalisation** : le projet est planifié sur plusieurs semaines avec un jalonnement des livrables (maquettes, base de données, fonctionnalités).
- **Public cible** : utilisateurs internes à l'entreprise (services IT, développeurs, managers).
- **Ergonomie attendue** : navigation fluide, messages d'erreur clairs, interface minimaliste et lisible.

Le cahier des charges constitue la **colonne vertébrale** de la phase de conception. Il a servi de point d'ancrage pour toutes les décisions prises ultérieurement en matière de développement et de tests.

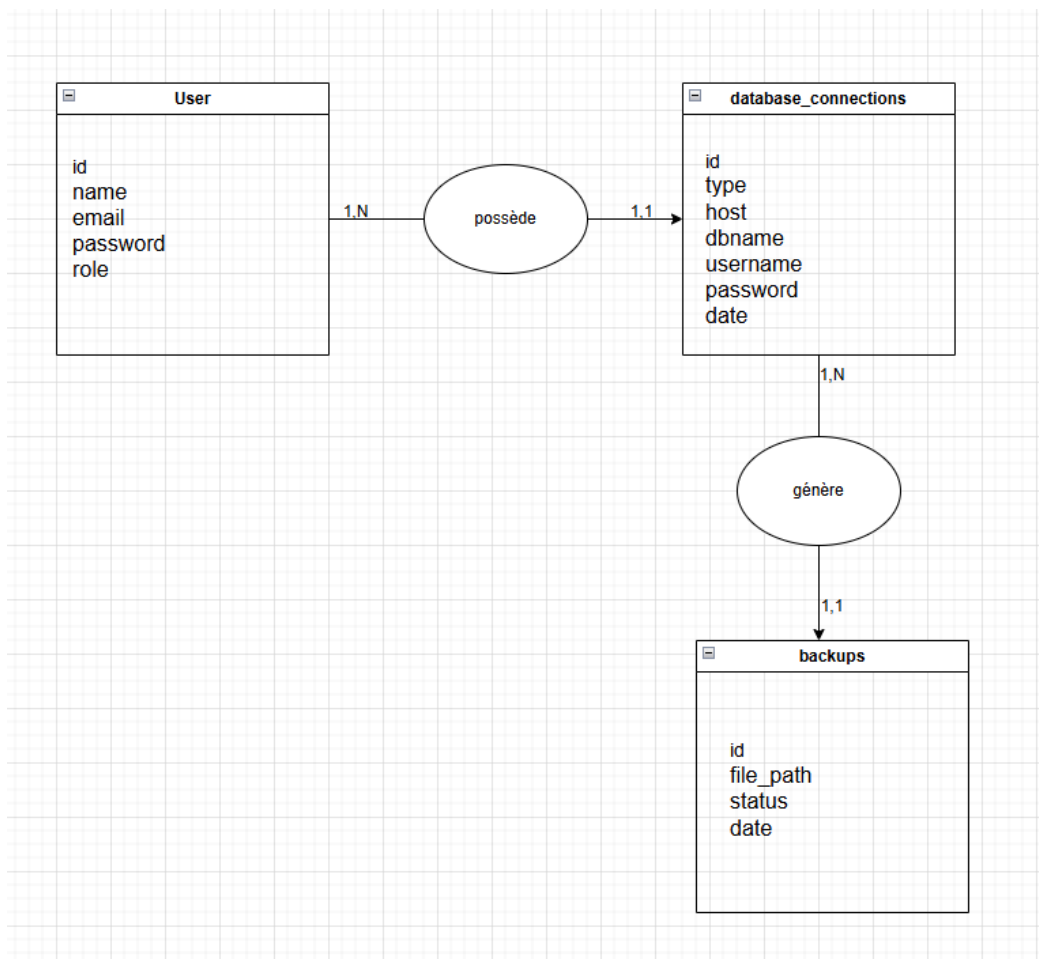
## 2. Modélisation de la base de données (MCD, MLD, MPD)

Pour structurer les données, une modélisation Merise a été réalisée :

- **MCD (Modèle Conceptuel de Données)** : C'est une représentation graphique de haut niveau qui permet facilement et simplement de comprendre comment les différents éléments sont liés entre eux.

Il fait apparaitre :

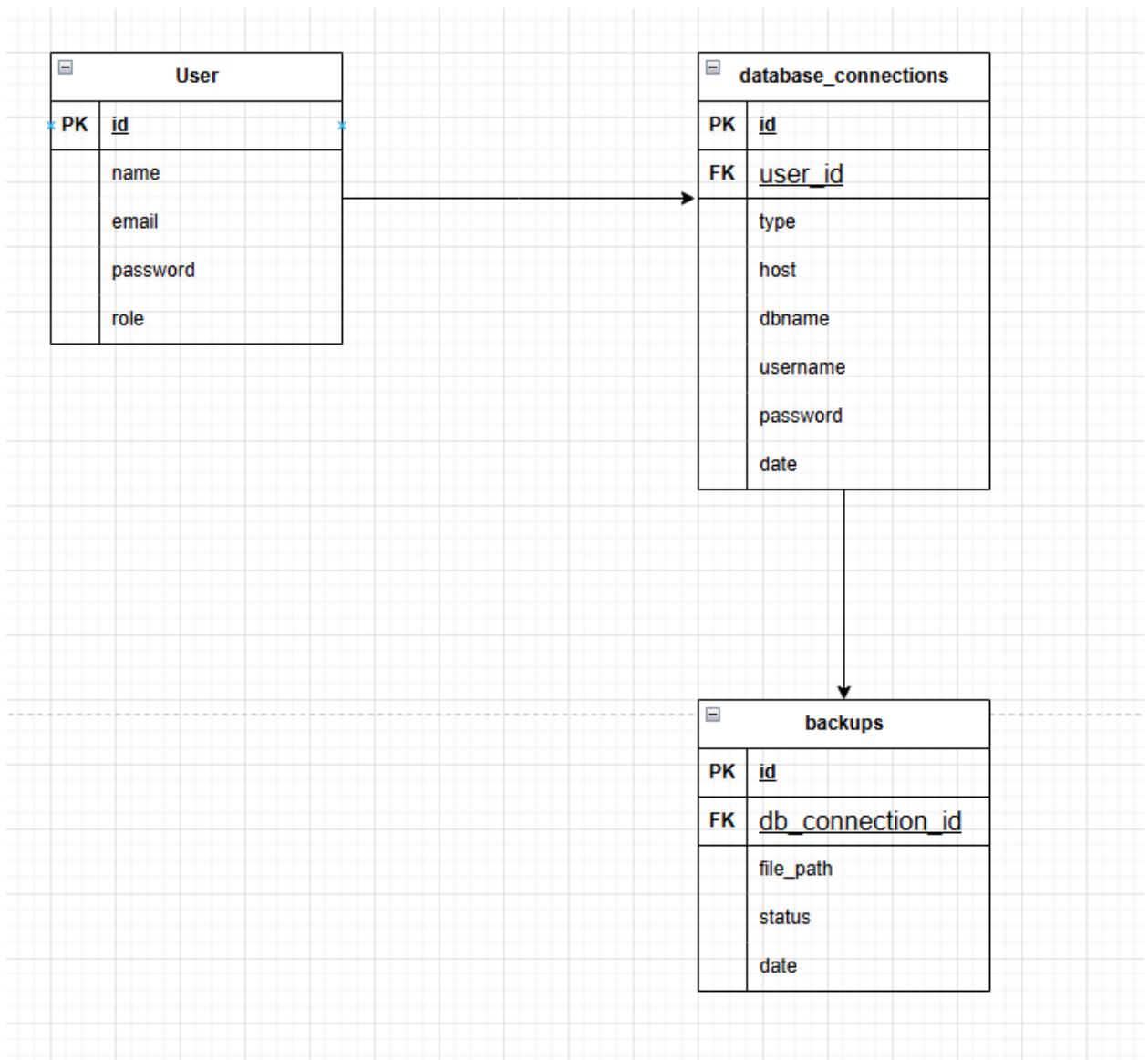
- Les entités
- Les propriétés
- Les relations qui expliquent et précisent comment les entités sont reliées entre elles
- Les cardinalités



- **MLD (Modèle Logique de Données)** : C'est la représentation textuelle du MPD. Il représente la structure de la base de données.

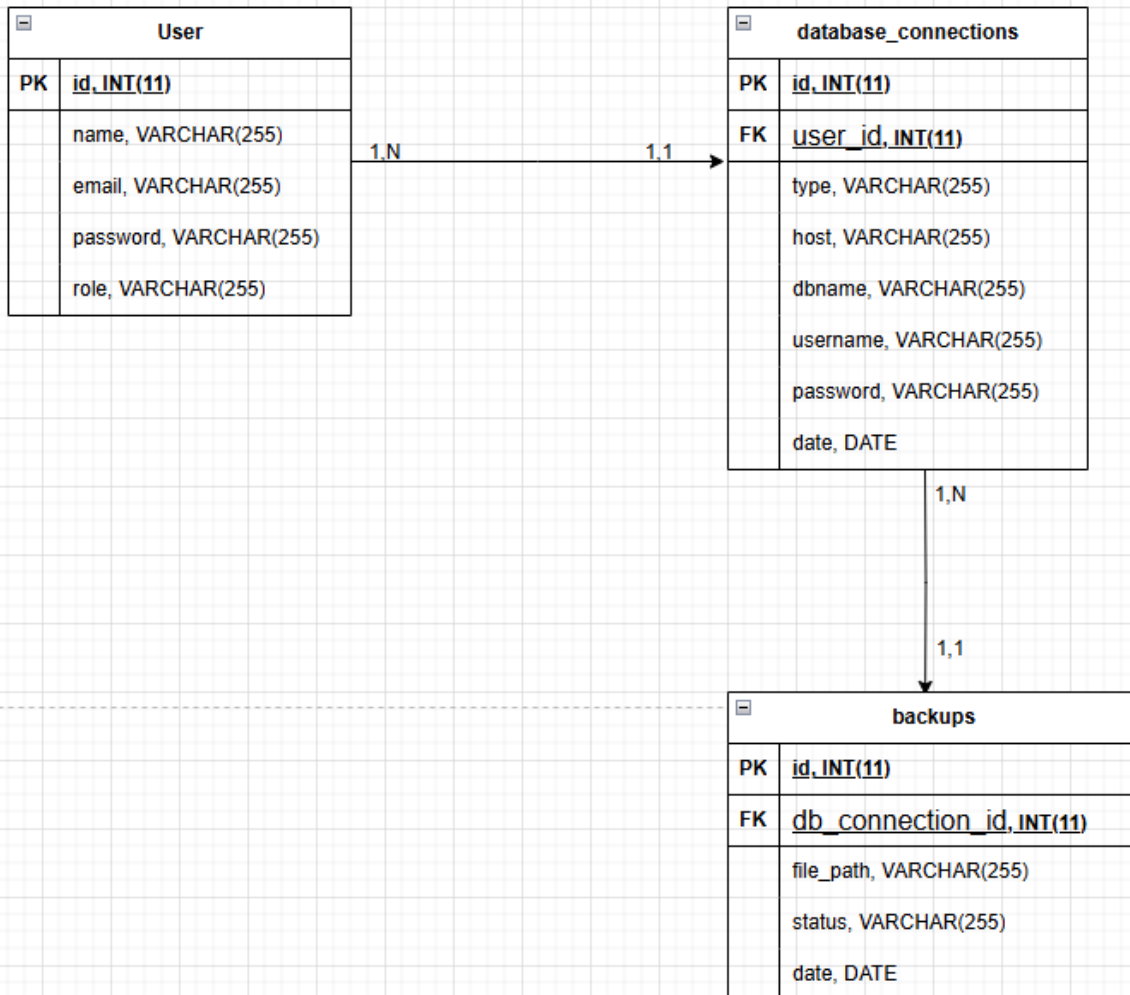
Il fait apparaitre:

- Les tables
- Les tables de relations
- Les clés primaires
- Les clés étrangères



- **MPD (Modèle Physique de Données)** : il traduit le MLD dans le langage du SGBD utilisé. Pour SafeBase, cela se traduit par des migrations Laravel pour générer les tables dans MySQL ou PostgreSQL.

Ce travail de modélisation permet une cohérence entre la structure de la base, la logique applicative, et l'interface utilisateur.



### 3. Conception des interfaces utilisateur avec Figma

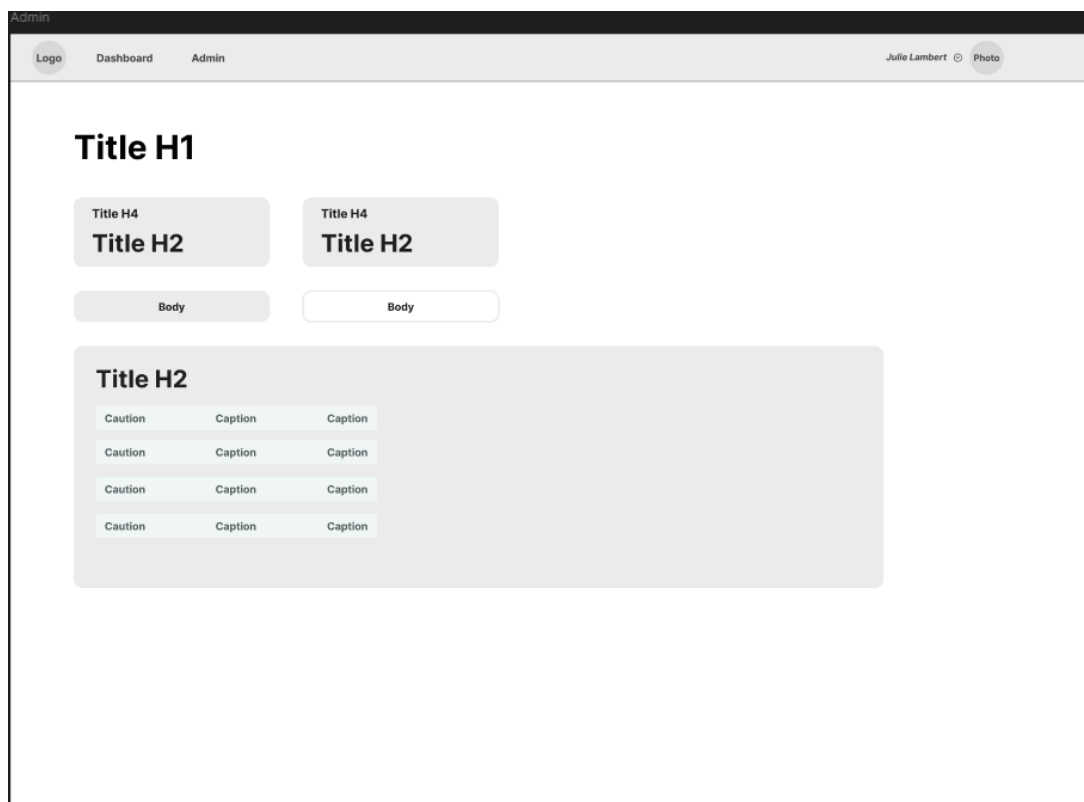
La phase de conception UI a été réalisée sur Figma, un outil de prototypage visuel collaboratif. Deux types de maquettes ont été développées :

#### Wireframes (basse fidélité)

Ils ont permis de poser l'ergonomie générale des pages, comme :

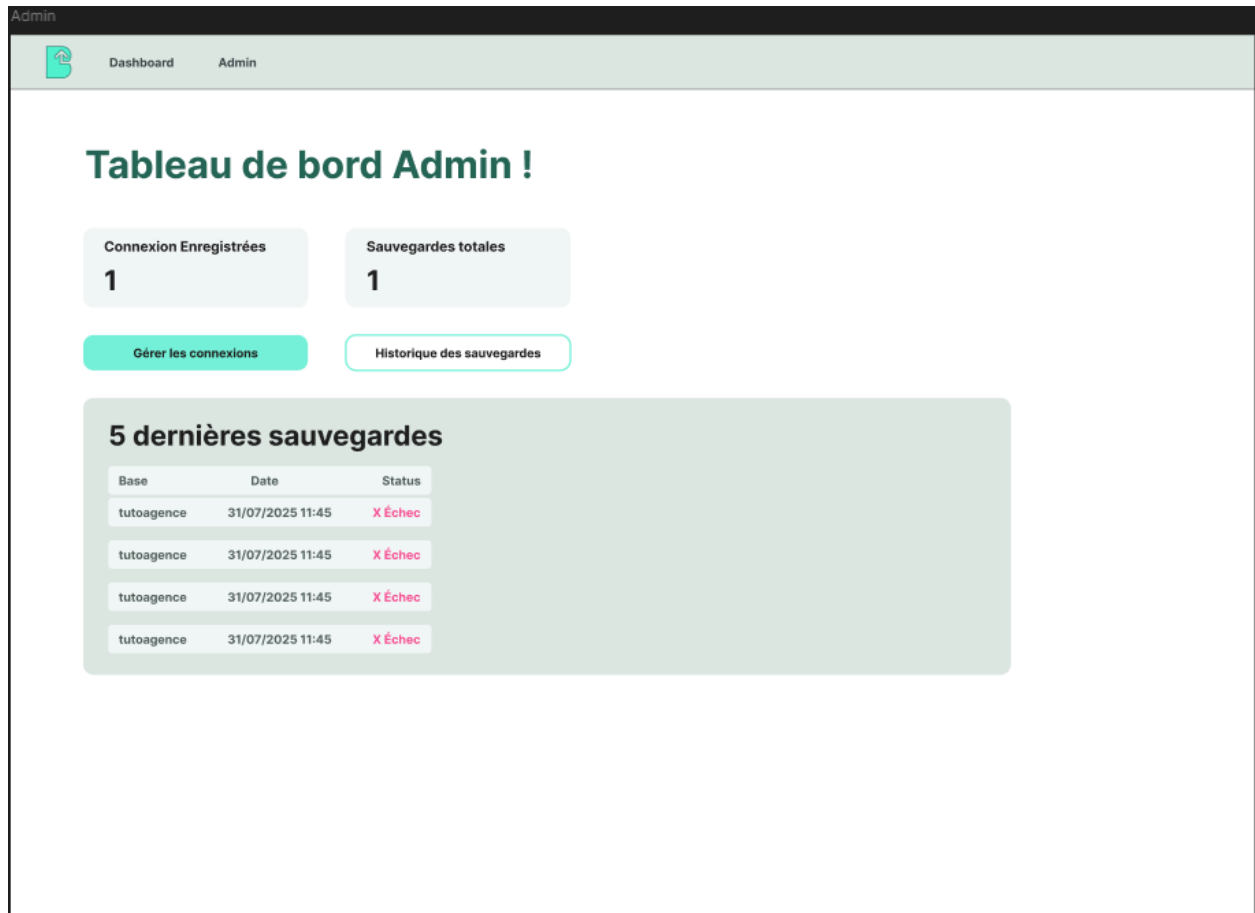
- Page d'accueil / login
- Tableau de bord avec les connexions et les backups
- Page de configuration d'une nouvelle base

Ces wireframes ont servi à structurer les parcours utilisateur et à recueillir des retours précoces sur l'agencement des éléments.



## Maquettes finales (haute fidélité)

Les maquettes haute fidélité ont intégré un design system complet (voir ci-dessous) et simulent précisément le rendu visuel et les interactions. Une approche mobile-first a été retenue afin d'assurer une compatibilité multi-supports. La navigation y est fluide et intuitive, adaptée aussi bien aux écrans de bureau qu'aux tablettes.

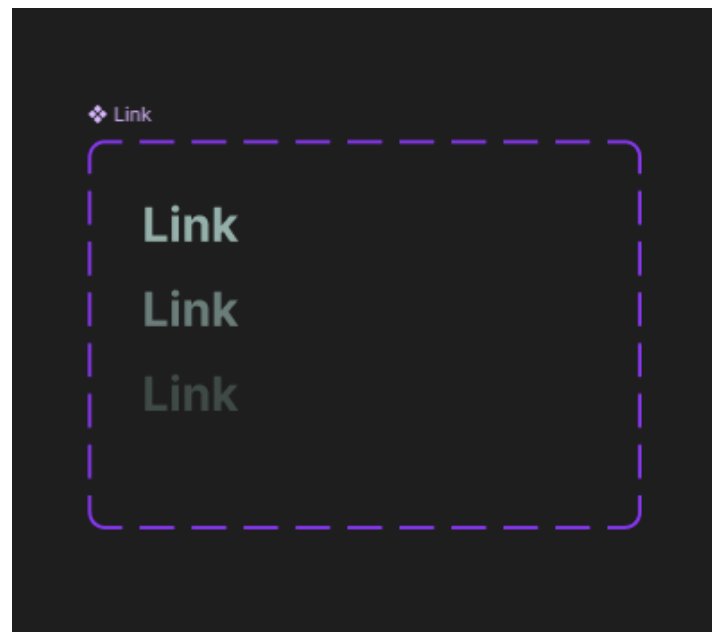
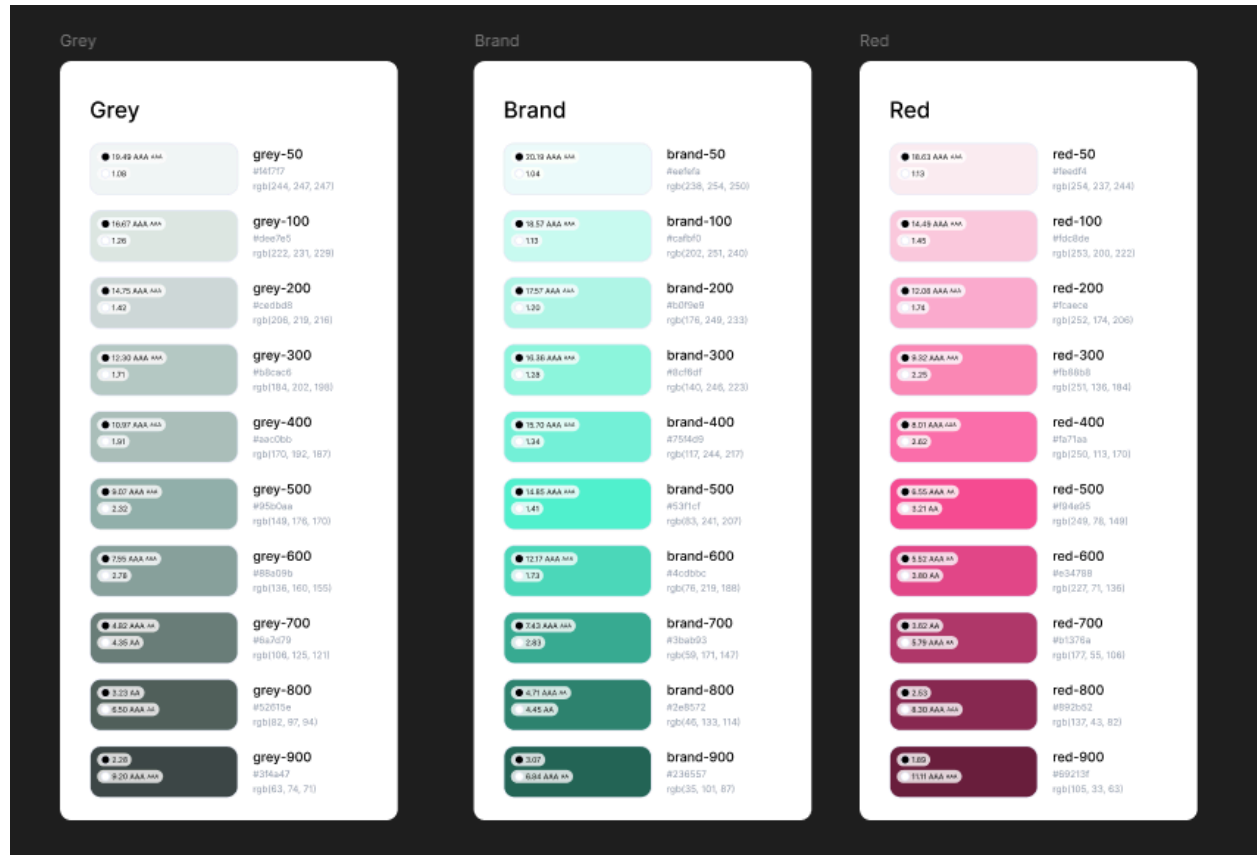


## 4. Design system et design pattern

Le **design system** regroupe tous les éléments réutilisables de l'interface :

- **Palette de couleurs**
- **Typographie**
- **Composants UI** : boutons, cartes, tableaux, formulaires cohérents visuellement

Cela permet une cohérence graphique et facilite la maintenance de l'interface.



---

## Design Pattern MVC

L'application suit une **architecture MVC (Modèle – Vue – Contrôleur)** native à Laravel.

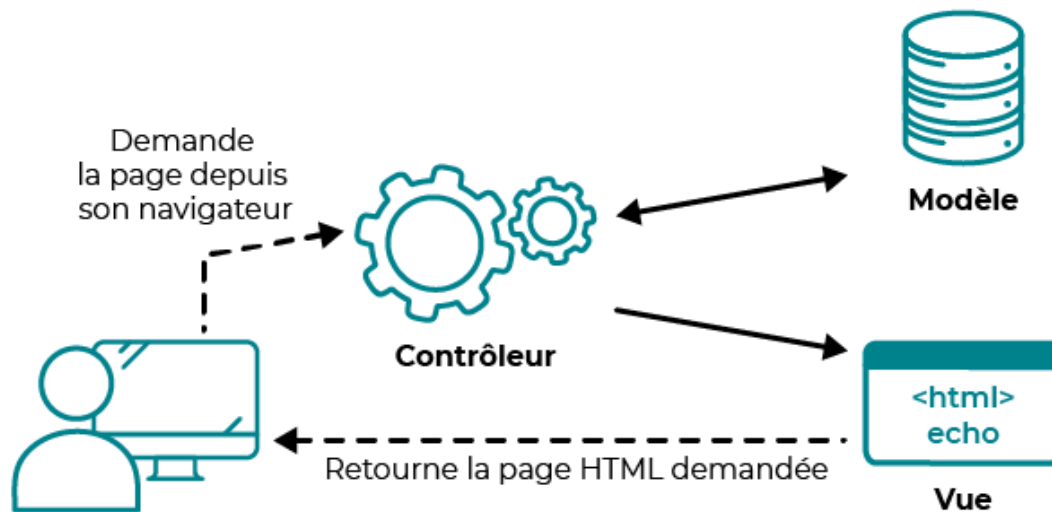
Le design pattern MVC est l'un des patterns les plus utilisés. Le but du MVC est de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts:

- **Modèles** : cette partie gère ce qu'on appelle la logique métier du site. Elle comprend notamment la gestion des données qui sont stockées, mais aussi tout le code qui prend des décisions autour de ces données. Son objectif est de fournir une interface d'action la plus simple possible au contrôleur. On y retrouve donc entre autres des algorithmes complexes et des requêtes SQL. (ex. **Backup, User, DatabaseConnection**).
- **Vues** : Cette partie se concentre sur l'affichage. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions PHP très simples, pour afficher par exemple une liste de messages. Avec Laravel ce sont les fichiers Blade (**.blade.php**) qui définissent l'interface utilisateur HTML rendue dans le navigateur.
- **Contrôleurs** : ils reçoivent les requêtes HTTP, appellent les modèles et renvoient les vues avec les données nécessaires.

Cette partie gère les échanges avec l'utilisateur. C'est en quelque sorte l'intermédiaire entre l'utilisateur, le modèle et la vue. Le contrôleur va recevoir des requêtes de l'utilisateur. Pour chacune, il va demander au modèle d'effectuer certaines actions et de lui renvoyer les résultats. Puis il va adapter ce résultat et le donner à la vue. Enfin, il va renvoyer la nouvelle page HTML, générée par la vue, à l'utilisateur.

Cette séparation des responsabilités améliore la maintenabilité, la testabilité et la sécurité du code.





## 5. Choix techniques

### Laravel 12

Laravel est un framework PHP moderne, reconnu pour :

- Sa syntaxe élégante et expressive
- Son système d'ORM (Eloquent) facilitant la gestion des données
- Ses outils CLI puissants (artisan) pour les migrations, les tests, la gestion des tâches, etc.
- Sa sécurité native (CSRF, hashing des mots de passe, etc.)
- Sa communauté active et sa documentation très complète

La version 12 a été choisie pour bénéficier des dernières améliorations de performances et de sécurité.

---

## Laravel Breeze

Laravel Breeze est un package minimaliste conçu pour gérer l'authentification dans les applications Laravel.

Le choix de **Laravel Breeze** pour l'authentification s'explique par :

- sa simplicité d'intégration,
- sa légèreté (pas de JavaScript imposé),
- sa compatibilité avec Blade et Tailwind CSS,
- son support natif pour l'enregistrement, la connexion, la vérification d'email, la réinitialisation de mot de passe, etc.

Cela a permis de sécuriser rapidement les accès aux routes protégées de l'application et de mettre en place un tableau de bord distinct selon le rôle de l'utilisateur (admin / utilisateur standard).

Breeze permet donc de rester simple tout en ajoutant rapidement une authentification robuste.

## Docker

Docker est une plateforme logicielle permettant de créer, déployer et gérer des applications dans des conteneurs virtuels. Les conteneurs Docker encapsulent tout ce dont une application a besoin pour s'exécuter, comme le code, les bibliothèques et les dépendances. Cela permet d'assurer que l'application fonctionne de manière cohérente dans différents environnements informatiques.

L'environnement de développement est conteneurisé avec Docker pour plusieurs raisons :

- Garantir la portabilité entre environnements (local, CI/CD, production)
- Éviter les problèmes de dépendances (versions de PHP, MySQL...)
- Faciliter le déploiement automatique via des fichiers Dockerfile et docker-compose.yml

Chaque service (Laravel, MySQL, phpMyAdmin) tourne dans un conteneur séparé, ce qui permet une isolation optimale.

---

## MySQL & PostgreSQL

SafeBase permet de gérer deux moteurs de base de données :

- MySQL, utilisé par défaut pour stocker les informations de l'application
- PostgreSQL, pour permettre aux utilisateurs de sauvegarder aussi ce type de base

Cela couvre la majorité des cas d'usage rencontrés en entreprise.

### Autres outils

- Composer : gestionnaire de dépendances PHP
- NPM + Vite : pour compiler les assets front-end (CSS, JS)
- GitHub : hébergement du code source, versionning, intégration avec GitHub Actions pour le CI/CD

## 7. Fonctionnalités principales

Voici les principales fonctionnalités mises en œuvre dans SafeBase :

- Authentification sécurisée (avec rôles : utilisateur / administrateur)
- Gestion des connexions aux bases de données : ajout, modification, suppression
- Sauvegarde : création de dumps via mysqldump ou pg\_dump côté serveur
- Restauration : envoi d'un fichier dump et restauration automatique sur la base cible
- Interface utilisateur moderne, responsive et intuitive
- Journalisation des sauvegardes (date, statut, utilisateur initiateur)

---

## Partie 2 – Développement

### 1. Développement Front-End

Le développement front-end de SafeBase s'inscrit dans une architecture MVC (Modèle-Vue-Contrôleur) conforme aux standards Laravel. Cette organisation permet une séparation claire des responsabilités, une évolutivité facilitée, ainsi qu'une meilleure

#### a. Authentification et rôles utilisateurs

L'authentification a été mise en place à l'aide de Laravel Breeze. Ce starter kit permet une gestion complète des comptes utilisateurs : inscription, connexion, déconnexion, vérification d'adresse email, et réinitialisation de mot de passe. Les routes sont automatiquement protégées par les middlewares **auth et verified**. Un système de rôles a été implémenté (admin / utilisateur) afin d'adapter dynamiquement l'affichage des interfaces et les autorisations disponibles pour chaque utilisateur.

#### b. Interface de connexion et tableau de bord

La page de connexion affiche un formulaire simple, sécurisé par un token CSRF, qui valide les entrées utilisateur côté serveur. Une fois connecté, l'utilisateur est redirigé vers un tableau de bord personnalisé. Ce tableau de bord permet de consulter les connexions à des bases de données existantes, d'en ajouter de nouvelles, et d'accéder à l'historique des sauvegardes.

#### c. Gestion des connexions aux bases de données

L'interface de gestion des connexions permet à l'utilisateur d'enregistrer différentes configurations (type de SGBD, hôte, nom de la base, identifiants...). Chaque enregistrement est validé côté serveur afin de prévenir les erreurs ou les configurations invalides. L'affichage de ces connexions dans le tableau utilise les composants Blade pour afficher les données sous forme de tableau avec boutons d'action (modifier / supprimer).

---

#### d. Lancement et restauration des sauvegardes

L'utilisateur peut, via l'interface front, lancer manuellement une sauvegarde de l'une des bases de données enregistrées. Il peut également restaurer une base à partir d'un fichier de dump précédemment généré. Les boutons d'action déclenchent des routes POST protégées par authentification, et les messages de confirmation ou d'erreur sont affichés dans des composants Blade personnalisés (bannières vertes ou rouges).

#### e. Utilisation de Blade et Tailwind CSS

Le moteur de vues Blade est utilisé pour la création de l'interface. Il permet de structurer les vues de manière modulaire grâce aux directives **@extends**, **@section**, **@yield**, et **@include**. Cette approche facilite la réutilisation des composants et garantit une cohérence visuelle sur l'ensemble de l'application.

Le design est assuré à l'aide de Tailwind CSS, un framework de classes utilitaires qui permet de concevoir rapidement des interfaces réactives, claires et esthétiques. Ce choix offre une grande souplesse dans l'implémentation du responsive design, tout en restant conforme aux normes d'accessibilité.

#### f. Sécurité du front-end

Les bonnes pratiques de sécurité ont été rigoureusement respectées :

- Tous les formulaires sont protégés par un token CSRF: Le jeton CSRF (Cross-Site Request Forgery) est une mesure de sécurité utilisée dans les applications web pour prévenir les attaques où un utilisateur non autorisé pourrait soumettre des requêtes en se faisant passer pour un utilisateur légitime.

Ce jeton est généré côté serveur et envoyé au client, généralement sous forme de cookie ou de champ caché dans un formulaire. Lorsque l'utilisateur soumet une requête, le serveur vérifie que le jeton CSRF correspond à celui attendu pour valider la requête. Cela aide à garantir que les actions soumises proviennent bien de l'utilisateur authentifié et non d'une source malveillante.

En pratique, dans Laravel, par exemple, le jeton CSRF est automatiquement inclus dans les formulaires générés par Blade à l'aide de la directive `@csrf`. Cela permet de sécuriser les actions POST, PUT, DELETE, etc., en vérifiant que le jeton CSRF envoyé correspond à celui stocké sur le serveur.

- Les routes sensibles sont protégées par des middlewares : fonctions ou des scripts intermédiaires qui traitent les requêtes HTTP avant qu'elles n'atteignent la route finale de l'application. Ils permettent d'effectuer des tâches telles que l'authentification des utilisateurs, la gestion des sessions, la validation des données, ou encore la gestion des erreurs.

```
1 reference | 0 implementations
class AdminOnly
{
    0 references | 0 overrides
    public function handle(Request $request, Closure $next): mixed
    {
        // Vérifie si l'utilisateur est connecté
        $user = $request->user();

        if (!$user || $user->role !== 'admin') {
            abort(code: 403, message: 'Accès interdit');
        }

        return $next($request);
    }
}
```

```
->withMiddleware(function (Middleware $middleware) {

    $middleware->alias([

        'admin' => \App\Http\Middleware\AdminOnly::class,

    ]);

})
```

- Les données affichées sont échappées automatiquement par Blade : Laravel empêche par défaut l'injection de code HTML ou JavaScript dangereux dans tes vues. C'est une mesure de sécurité contre les attaques XSS (Cross-Site Scripting). Pour échapper les données en Laravel, utilisez les doubles accolades {{ }}. Cette syntaxe échappe automatiquement les caractères spéciaux, transformant par exemple les balises HTML en entités HTML sécurisées.

```
<h1 class="mb-6 text-3xl font-bold">  
  🙋 Bonjour {{ Auth::user()->name }}  
</h1>
```

- Aucune donnée sensible n'est stockée ou exposée côté client.

#### g. Tests front-end avec PHPUnit

Pour garantir la stabilité de l'interface, des tests fonctionnels ont été mis en place à l'aide de PHPUnit. Ils valident par exemple l'accessibilité de la page de connexion ou le bon fonctionnement de la redirection après authentification.

Les routes protégées sont également testées pour s'assurer qu'un utilisateur non authentifié est bien redirigé vers la page de connexion.

```
● PS C:\wamp64\www\safeBase> php artisan test  
  
PASS Tests\Unit\ExampleTest  
✓ that true is true  
  
PASS Tests\Feature\Auth\AuthenticationTest  
✓ login screen can be rendered  
✓ users can authenticate using the login screen  
✓ users can not authenticate with invalid password  
✓ users can logout
```

```
0 references | 0 overrides
public function guest_cannot_access_backups_route(): void
{
    $response = $this->get(uri: '/backups');
    $response->assertRedirect(uri: '/login');
}

0 references | 0 overrides
public function authenticated_user_can_access_backups(): void
{
    $user = User::factory()->create();

    $response = $this->actingAs(user: $user)->get(uri: '/backups');
    $response->assertStatus(status: 200);
}
```

Le développement front-end répond aux compétences suivantes du référentiel :

- **Développer des interfaces utilisateur sécurisées** : interface conforme aux maquettes, responsive, validation des saisies.
- **Utiliser un framework front-end (Laravel Blade)** : composants dynamiques, layouts factorisés.
- **Mettre en œuvre l'authentification et la gestion des rôles** : Breeze, middleware, vue conditionnelle.
- **Tester les composants front-end** : tests PHPUnit sur vues et redirections.
- **Respecter les bonnes pratiques de sécurité** : CSRF, middleware, échappement des données.

Ainsi, le front-end de SafeBase constitue une interface moderne, claire et sécurisée, pleinement alignée avec les attentes des utilisateurs et les standards du développement professionnel Laravel.



## **h. Perspectives d'amélioration**

Le front-end de SafeBase pourra évoluer vers une structure plus dynamique à l'aide d'Inertia.js ou Vue.js, ce qui permettrait une expérience utilisateur plus fluide. Des tests end-to-end pourraient également être ajoutés via Laravel Dusk pour tester l'enchaînement des interactions utilisateur.

En résumé, le développement front-end de SafeBase couvre l'ensemble des compétences attendues : mise en œuvre d'une interface responsive et sécurisée, gestion des rôles, interaction avec les fonctionnalités back-end, tests de vérification, et application d'un design system cohérent basé sur Tailwind.

## **b. Développement Back-End**

Le développement back-end de l'application SafeBase repose sur une architecture robuste et modulaire, mettant en œuvre les principes de la programmation orientée objet (POO), l'ORM Eloquent, les middlewares Laravel, le routage avancé et une gestion centralisée des accès et des autorisations. Cette section détaille les choix techniques, les composants développés, leur fonctionnement et les mécanismes de sauvegarde et de restauration.

### **ORM Eloquent et logique métier**

Laravel Eloquent est l'ORM utilisé pour interagir avec la base de données. Chaque table est représentée par une classe modèle (Model).

Un ORM est un logiciel qui facilite la gestion des enregistrements de bases de données en représentant les données sous forme d'objets, fonctionnant comme une couche d'abstraction au-dessus du moteur de base de données utilisé pour stocker les données d'une application.

```

*/
public function up(): void
{
    Schema::create(table: 'backups', callback: function (Blueprint $table): void {
        $table->id();
        $table->foreignId(column: 'db_connection_id')->constrained(table: 'database_connections')->onDelete(action: 'cascade');
        $table->string(column: 'file_path');
        $table->enum(column: 'status', allowed: ['success', 'fail'])->default(value: 'success');
        $table->timestamps();
    });
}

```

```

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

11 references | 0 implementations
class Backup extends Model
{
    0 references
    protected $fillable = [
        'db_connection_id', // ✅ on autorise la clé étrangère
        'file_path',
        'status',
    ];

    0 references | 0 overrides
    public function database(): BelongsTo
    {
        return $this->belongsTo(related: DatabaseConnection::class, foreignKey: 'db_connection_id');
    }
}

```

Les relations entre entités sont utilisées (hasMany, belongsTo) pour relier utilisateurs, connexions et sauvegardes. Cela permet une navigation fluide entre les tables et une encapsulation de la logique métier.

## Routeage Laravel et middlewares

Le fichier **web.php** contient les routes organisées selon les rôles et niveaux d'authentification. Exemple :

```
// ✅ Routes réservées aux ADMIN
Route::middleware(middleware: ['auth', 'admin'])->prefix(prefix: 'admin')->group(callback: function (): void {

    // Tableau de bord admin
    Route::get(uri: '/dashboard', action: [AdminDashboardController::class, 'index'])->name(name: 'admin.dashboard');

    // Gestion des connexions BDD
    Route::resource(name: '/databases', controller: DatabaseConnectionController::class)->except(methods: ['show']);

    // Sauvegarde manuelle d'une base
    Route::post(uri: '/databases/{database}/backup', action: [BackupController::class, 'run'])->name(name: 'databases.backup');

    // Historique complet (admin)
    Route::get(uri: '/backups', action: [BackupController::class, 'index'])->name(name: 'backups.index');
    Route::get(uri: '/backups/download/{backup}', action: [BackupController::class, 'download'])->name(name: 'backups.download');
    Route::post(uri: '/backups/restore/{backup}', action: [BackupController::class, 'restore'])->name(name: 'backups.restore');
});
```

Les middlewares personnalisés tels que **admin** restreignent l'accès aux actions critiques.

## Sauvegarde manuelle et restauration

Lorsqu'un utilisateur clique sur « Sauvegarder », le contrôleur **BackupController** construit dynamiquement une commande **mysqldump** ou **pg\_dump** en fonction du type de base sélectionnée (MySQL ou PostgreSQL). Cette commande est ensuite exécutée à l'aide de **exec()**, et le fichier **.sql** généré est enregistré dans le dossier **storage/app/backups**, accompagné d'un enregistrement dans la base via le modèle **Backup**.

Extrait de code :

```
$cmd = "mysqldump -h {$database->host} -u {$database->username} {$passwordOption} {$database->dbname} > \"{$backupPath}\" 2> \"{$errorLogPath}\"";

exec(command: $cmd, output: &$output, result_code: &$result);
```

Cette commande permet d'exécuter un **backup MySQL** en appelant **mysqldump** depuis PHP (en ligne de commande), et de **sauvegarder le résultat dans un fichier .sql**, tout en **loggant les erreurs** dans un fichier texte.

Pour la restauration, le processus inverse est effectué, également via **exec()**, en injectant le fichier de dump dans la base d'origine. Aucune commande Artisan spécifique n'est utilisée pour ces opérations.

### Sauvegarde automatique avec CRON

Un CRON est configuré dans le planificateur Laravel pour exécuter régulièrement les sauvegardes :

La commande **backup:run** parcourt toutes les connexions actives et crée un dump pour chacune. Cela garantit une sauvegarde quotidienne sans intervention manuelle.

```
// Planifier une sauvegarde automatique tous les jours à 2h du matin
Schedule::call(callback: function (): void {
    $connections = DatabaseConnection::all();

    foreach ($connections as $db) {
        // Appeler la commande Artisan pour chaque connexion
        Artisan::call(command: 'backup:run', parameters: [
            'db_id' => $db->id
        ]);
    }

    info(message: '✅ Sauvegarde automatique terminée à ' . now());
})->dailyAt(time: '02:00');
```

### Sécurité back-end renforcée

Les validations sont faites avec des classes **FormRequest** : Les form requests dans Laravel sont des classes spéciales conçues pour la validation des données avant leur traitement par le contrôleur. Elles permettent de centraliser la logique de validation des requêtes HTTP entrantes, ce qui simplifie et sécurise le processus de validation des données.

```
public function rules(): array
{
    return [
        'email' => ['required', 'string', 'email'],
        'password' => ['required', 'string'],
    ];
}
```

Les dumps sont stockés hors du dossier public

## Sécurité des données sensibles

Certaines informations manipulées par SafeBase sont sensibles (mots de passe de connexion à une BDD par exemple). Pour cela, plusieurs protections ont été mises en place :

### a. Hachage des mots de passe utilisateurs

Les mots de passe sont automatiquement hachés à l'inscription ou la modification via **Hash::make(\$password)**. Cela signifie qu'ils ne peuvent pas être récupérés même en cas d'accès à la base.

### b. Chiffrement des mots de passe de connexion aux BDD externes

Les mots de passe utilisés pour se connecter aux bases à sauvegarder sont chiffrés avec l'aide de **Crypt::encryptString()** de Laravel. Ils ne sont donc lisibles qu'à l'intérieur de l'application, et sont déchiffrés uniquement au moment du dump.

### c. Clé d'application et variables sensibles

Laravel utilise une clé d'application (**APP\_KEY**) pour chiffrer les données sensibles comme les cookies et les tokens. Cette clé est générée automatiquement (à l'aide de **php artisan key:generate**) et est stockée dans le fichier **.env**, qui n'est **jamais** versionné.

---

## Tests back-end

Des tests automatisés permettent de valider :

- la création de dumps,
- la restauration fonctionnelle,
- la protection des routes sensibles selon le rôle utilisateur,
- les erreurs en cas de connexion invalide.

## Compétences couvertes (Back-End)

- Utilisation de l'ORM Eloquent pour gérer les modèles et relations,
- Mise en place de middlewares et politiques d'accès sécurisées,
- Automatisation des processus via Artisan et CRON,
- Architecture MVC respectée dans tous les composants,
- Tests robustes sur les opérations critiques du back-end.

## Partie 3 – Intégration continue et déploiement continu (CI/CD)

Le système CI/CD (Continuous Integration / Continuous Deployment) mis en place pour SafeBase permet d'automatiser l'ensemble du cycle de test et de déploiement de l'application. Cela garantit une qualité constante du code, une livraison plus rapide des fonctionnalités et une meilleure collaboration au sein d'une équipe de développement.

J'utilise **GitHub Actions**, un outil d'automatisation intégré à GitHub, pour définir et exécuter nos pipelines. Cela me permet de lancer automatiquement une suite d'actions à chaque fois qu'un développeur pousse du code sur la branche principale du projet.

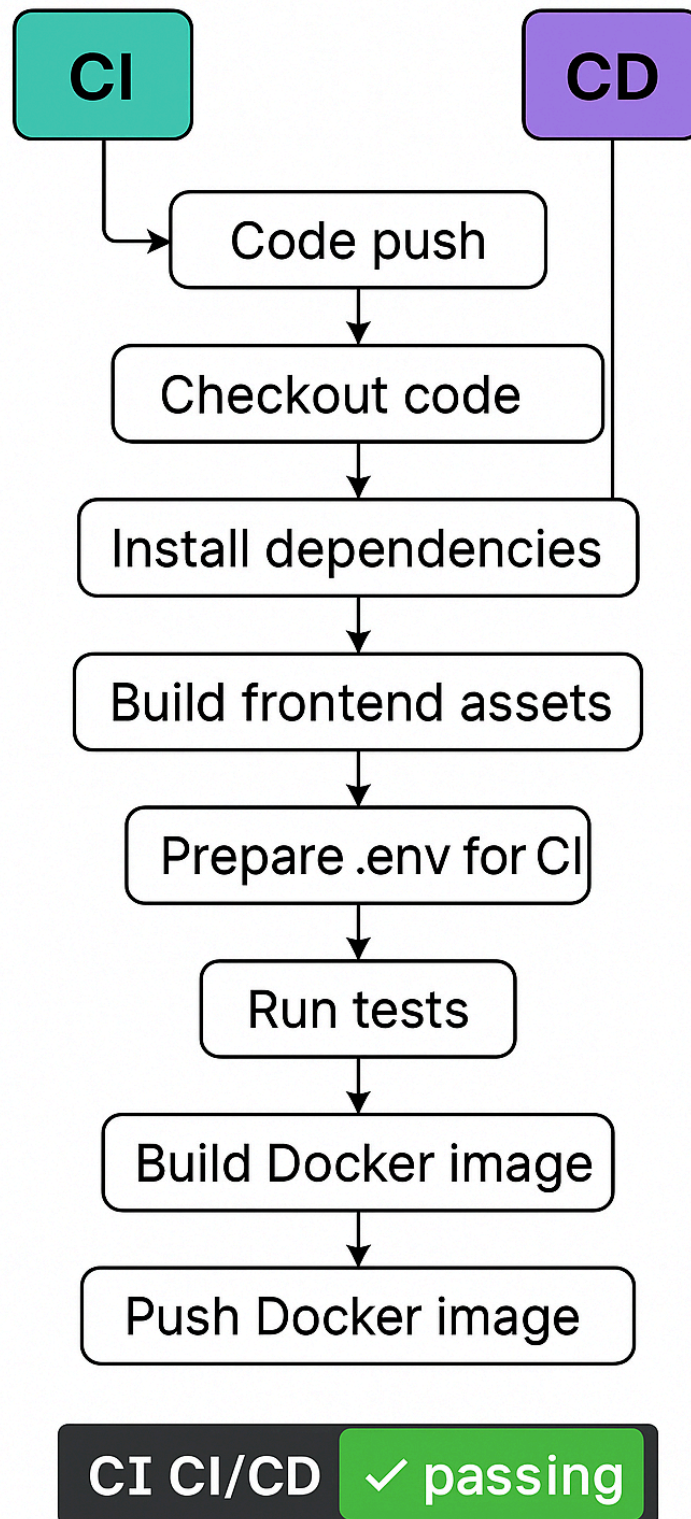
---

## 1. Présentation générale du pipeline CI/CD

Le pipeline CI/CD de SafeBase est défini dans un fichier YAML situé à l'adresse [.github/workflows/ci-cd.yml](#). Ce fichier décrit étape par étape les instructions que GitHub doit exécuter automatiquement lorsqu'un développeur pousse du code ou publie une nouvelle version.

Le pipeline est structuré en deux parties principales :

- **L'intégration continue (CI)**, qui vérifie automatiquement que le code fonctionne correctement,
- **Le déploiement continu (CD)**, qui construit une image Docker du projet et la publie dans un registre de conteneurs pour faciliter le déploiement.





## 2. Étapes du pipeline d'intégration continue (CI)

Le job CI s'intitule **build-and-test**. Il est exécuté à chaque push sur la branche **main**. Voici le détail de chacune de ses étapes :

### a. Récupération du code

```
# 1. Récupérer le code
- name: Checkout code
  uses: actions/checkout@v4
```

Cette étape permet de cloner le code du dépôt GitHub pour l'utiliser dans les étapes suivantes. C'est la base de tout pipeline, car sans le code, rien ne peut être testé ni déployé.

### b. Configuration de l'environnement PHP

```
# 2. Installer PHP + Composer
- name: Setup PHP
  uses: shivammathur/setup-php@v2
  with:
    php-version: '8.2'
```

Nous installons ici PHP 8.2, car c'est la version utilisée par Laravel 12. Cette étape garantit que toutes les commandes PHP et Composer s'exécuteront correctement.

### c. Installation des dépendances PHP

```
- name: Install Composer dependencies
  run: composer install --no-interaction --prefer-dist
      --optimize-autoloader
```

Composer est le gestionnaire de dépendances pour PHP. Il permet d'installer toutes les bibliothèques dont le projet a besoin (Laravel, Breeze, etc.). Les options utilisées évitent les interactions manuelles et optimisent l'autoloading.

#### d. Configuration de Node.js et installation des dépendances front-end

```
# 3. Installer Node.js (pour builder Vite)
- name: Setup Node.js
  uses: actions/setup-node@v4
  with:
    node-version: '20'

# 4. Installer les dépendances JS
- name: Install NPM dependencies
  run: npm ci
```

Node.js est nécessaire pour compiler les assets front-end. `npm ci` est utilisé ici car il est plus rapide et plus fiable que `npm install` dans les contextes CI.

#### e. Compilation des assets avec Vite

```
# 5. Builder les assets Vite
- name: Build frontend assets
  run: npm run build
```

Cette commande utilise Vite pour compiler les fichiers CSS et JavaScript en version optimisée, que Laravel pourra ensuite servir aux utilisateurs finaux.

#### f. Préparation d'un environnement de test

```
# 6. Préparer un .env minimal pour CI
- name: Prepare Laravel .env for CI
  run: |
    cp .env.example .env
    echo "DB_CONNECTION=sqlite" >> .env
    echo "DB_DATABASE=:memory:" >> .env
```

Dans ce contexte, on utilise une base SQLite stockée en mémoire (volatile). Cela évite d'avoir à configurer une vraie base de données et accélère les tests.

### g. Génération de la clé de l'application

```
# 7. Générer la clé Laravel
- name: Generate app key
  run: php artisan key:generate --force
```

Cette commande est indispensable pour que Laravel fonctionne : elle permet de sécuriser les sessions, les tokens, et d'autres données sensibles.



### h. Lancement des tests automatisés

```
# 8. Lancer les tests Laravel
- name: Run Laravel tests
  run: php artisan test
```

Tous les tests définis dans le répertoire **tests/** sont lancés. Ces tests permettent de vérifier que les fonctionnalités critiques (authentification, sauvegarde, restauration...) fonctionnent toujours correctement après chaque modification du code.

Si l'un des tests échoue, le job s'arrête immédiatement, empêchant ainsi le déploiement d'une version instable.

Triggered via push 33 minutes ago

 julie-lambert pushed  14fa36e main

Status

Success

Total duration


1m 19s


Artifacts

—

ci-cd.yml

on: push

 build-and-test 23s

 docker-build-and-push 51s

### 3. Étapes du pipeline de déploiement continu (CD)

Une fois que les tests sont passés, le second job **docker-build-and-push** est déclenché. Ce job permet de construire une image Docker du projet et de l'envoyer vers un registre sécurisé.

#### a. Connexion au registre GitHub Container Registry (GHCR)

```
- name: Log in to GitHub Container Registry
  uses: docker/login-action@v2
  with:
    registry: ghcr.io
    username: ${ github.actor }
    password: ${ secrets.GHCR_PAT }
```

Pour publier une image Docker, il faut d'abord se connecter à un registre. Ici, on utilise le registre GHCR (fourni par GitHub). Le mot de passe est un **token personnel** enregistré dans les secrets du dépôt GitHub.

#### b. Construction de l'image Docker

```
- name: Build Docker image
  run: |
    docker build -t ghcr.io/${ github.repository_owner }/
    $IMAGE_NAME:latest .
```

L'image est construite à partir du fichier **Dockerfile** situé à la racine du projet. Elle contient tout le code PHP, les dépendances Laravel, les fichiers compilés, etc. Elle est ensuite étiquetée avec un tag **latest**.

#### c. Publication de l'image

```
- name: Push Docker image
  run: docker push ghcr.io/${ github.repository_owner }/
    $IMAGE_NAME:latest
```

Cette étape pousse l'image construite vers GHCR. Elle peut ensuite être déployée sur un serveur distant, dans un environnement cloud ou dans un cluster Kubernetes.



Install from the command line

[Learn more about packages](#)

```
$ docker pull ghcr.io/julie-lambert/safebase:latest
```



## 4. Intérêt de cette stratégie CI/CD pour SafeBase

Ce pipeline offre de nombreux avantages :

- **Sécurité renforcée** : seules les versions du projet qui passent les tests sont construites et envoyées,
- **Automatisation complète** : plus besoin de lancer manuellement les tests ou de construire les images,
- **Rapidité** : chaque modification est immédiatement testée et prête à être déployée,
- **Traçabilité** : chaque exécution du pipeline est archivée, avec un retour détaillé en cas d'erreur.

## 5. Améliorations envisagées

Voici quelques pistes pour aller plus loin dans l'automatisation DevOps :

- Ajouter des **tests d'interface utilisateur** avec Laravel Dusk,
- Générer un **rapport de couverture de test** (ex : avec PHPUnit + Xdebug),
- Ajouter un **déploiement automatique** vers un serveur distant après le push Docker,
- Ajouter un **badge d'état des tests** dans le README du dépôt GitHub.

Ce système CI/CD constitue une base solide pour maintenir la qualité, la sécurité et la fiabilité du projet SafeBase dans le temps. Il montre également une bonne compréhension des pratiques DevOps modernes, indispensables dans tout projet professionnel.

---

## Partie 4 – Mise en production, sécurité et documentation technique

La mise en production d'une application comme SafeBase ne se limite pas à copier des fichiers sur un serveur : elle implique une série d'étapes critiques pour garantir que l'application soit déployée dans un environnement stable, sécurisé et conforme aux bonnes pratiques. Cette partie détaille l'approche mise en place pour la publication, la protection et la documentation de SafeBase.

---

### 1. Mise en production de l'application

#### a. Création d'une image Docker prête à l'emploi

Grâce à l'image Docker générée dans le pipeline CI/CD, la mise en production de SafeBase devient une opération simple et fiable. Cette image contient tout le nécessaire pour exécuter l'application :

- Le code source Laravel compilé
- Les dépendances PHP installées
- Les assets front-end générés avec Vite

#### b. Déploiement avec Docker Compose

Pour lancer l'application sur un serveur distant ou une machine locale, il suffit d'utiliser **docker-compose up -d** avec le fichier **docker-compose.yml**. Ce fichier définit trois services :

- **app** : le conteneur Laravel
- **mysql** : la base de données MySQL
- **phpmyadmin** : une interface graphique pour manipuler les bases

Ce choix garantit une cohérence entre les environnements de développement, de test et de production.

#### c. Accès au service

Une fois les conteneurs lancés, l'application est accessible sur **http://localhost:8010** et phpMyAdmin sur **http://localhost:8081**. Ces ports sont configurés dans le **docker-compose.yml**.

---

## 2. Sécurité de l'application

### b. Données utilisateurs et protection des mots de passe

Les mots de passe sont **hachés** automatiquement à l'enregistrement grâce à la classe **Hash** de Laravel. Cela signifie qu'ils ne peuvent pas être décryptés même par un administrateur. En cas de fuite de données, les mots de passe restent protégés.

### c. Protection des routes sensibles

SafeBase implémente un système de rôles via middleware Laravel :

- Les routes administrateur (ajout d'une base, restauration) sont réservées aux comptes ayant le rôle **admin**.
- Les utilisateurs standards n'ont accès qu'à la visualisation des sauvegardes.

### d. Vulnérabilités connues et mises à jour

L'utilisation de Composer permet de gérer les mises à jour de sécurité via **composer update**. De plus, l'intégration de GitHub Actions permettrait de recevoir des alertes sur les vulnérabilités connues.

## 3. Documentation technique

### a. Documentation utilisateur

SafeBase inclut un README.md dans le dépôt GitHub qui explique :

- Comment installer le projet
- Comment exécuter l'application avec Docker
- Comment effectuer une sauvegarde ou une restauration

Ce document est utile pour tout nouvel utilisateur ou développeur souhaitant reprendre le projet.

## b. Arborescence du projet

Voici un extrait de la structure du projet :

```
├─ app/
├─ database/
├─ public/
├─ routes/
├─ tests/
├─ Dockerfile
├─ docker-compose.yml
└─ .github/workflows/ci-cd.yml
```

## c. Bonnes pratiques

- **Variables d'environnement** : centralisées dans `.env`
- **Pas de mot de passe en dur** : tous les accès sont configurés via Docker ou `.env`
- **Commentaires dans le code** : chaque méthode principale est documentée
- **Versionnement** : Git est utilisé tout au long du projet avec des commits atomiques et clairs

Cette partie montre que SafeBase ne se limite pas à du code : c'est une application déployable, sécurisée, et bien documentée, prête pour une utilisation professionnelle ou une extension future.



---

## Conclusion et retour d'expérience

Le développement de l'application **SafeBase** m'a permis de mettre en œuvre l'ensemble des compétences attendues d'un Concepteur Développeur d'Application, dans un contexte concret, structuré et complet. Ce projet m'a confrontée à toutes les étapes du cycle de vie logiciel, de la conception initiale (MCD, maquettes, cahier des charges) jusqu'au déploiement et la mise en production avec Docker et GitHub Actions.

Sur le plan **technique**, j'ai pu approfondir mon usage de Laravel 12, notamment en ce qui concerne l'authentification sécurisée, la structuration MVC, l'utilisation de l'ORM Eloquent et la mise en place de tests automatisés. Le système de sauvegarde/restauration des bases de données m'a permis de manipuler des commandes système avec **exec()** de façon sécurisée, tout en assurant le suivi des opérations dans l'application via des enregistrements en base.

Le volet **CI/CD** m'a donné l'occasion de créer un pipeline complet avec GitHub Actions, garantissant que seules les versions validées passent en production. Cela m'a sensibilisée à l'importance de l'automatisation, de la qualité de code et du travail collaboratif reproductible.

Sur le plan **organisationnel**, j'ai appliqué une démarche agile en structurant mon travail en tâches planifiées, en suivant des jalons clairs, et en m'adaptant aux imprévus, notamment les incidents matériels rencontrés. Cette expérience a renforcé ma capacité à être autonome, à prendre des décisions techniques raisonnées. Enfin, ce projet m'a permis de mieux appréhender les **enjeux de sécurité** (protection des accès, validation des entrées, cloisonnement des utilisateurs), d'accessibilité (interfaces claires et responsives), et de durabilité (sauvegarde).

**SafeBase** constitue donc une expérience formatrice complète, qui m'a permis de consolider mes acquis, de monter en compétence sur des aspects avancés du développement web et DevOps, et de produire une application utile, sécurisée, et maintenable.