

# Dossier de projet - TPI

---

OUTIL DE MONITORING DE SITES / PROJETS DE NOS CLIENTS

Florent Scheibler

FIRSTPOINT SÀRL | GALERIE SAINT-FRANÇOIS B 1003 LAUSANNE

## TABLE DES MATIERES

Analyse préliminaire .....	4
Introduction .....	4
Objectifs.....	4
Analyse /conception .....	5
Phase 1 - Concept.....	6
Architecture .....	6
Base de données.....	8
Modèle conceptuel.....	8
Modèle relationnel.....	8
Interface utilisateur .....	9
Page de connexion (login) .....	9
Page vue d'ensemble .....	10
Page détails .....	10
Risques techniques.....	11
Planification.....	12
Phase 2 - Concept.....	13
Phase 2.1 - Installation et configuration initiale de Laravel .....	13
Phase 2.2 - Implémentation de la base de données .....	14
Phase 2.3 - Intégration de l'authentification.....	14
Phase 2.4 - Développement de l'interface utilisateur – layout de base.....	15
Phase 2.5 - Intégration du services Oh Dear .....	16
Phase 2.6 - Intégration du service Flare.....	17
Phase 2.7 - Mise en place des tâches cron et fonctionnalité de rafraichissement .....	18
Phase 2.8 - Intégration de LiVewire.....	18
Phase 2.9 – Finalisation de l'interface utilisateur – Intégration services tiers .....	19
Phase 2.10 - Corrections et refactorisation .....	20
Phase 2.11 - Documentation utilisateur.....	21
Phase 2.12 – Déploiement de la documentation.....	21
Phase 2.13 - Corrections et refactorisation .....	22
Phase 2.14 - Evaluation et clôture du projet.....	22
Réalisation.....	24
Installation et configuration de l'environnement de développement .....	24
Installation de WAMP .....	24
Configuration d'Apache 2.4.58.....	24
Configuration de MySql 8.2.0.....	25
Configuration de PHP 8.2.13.....	26
Mise en place de laravel.....	27
Installation de composer 2.7.1 .....	27

Création du projet laravel.....	28
Mise en place dans l'IDE .....	30
Variables d'environnement (.env).....	31
Configuration de Tailwind CSS.....	31
Implémentation de la base de données .....	34
Créations des migrations de notre base de données .....	34
Table projects .....	35
Table sources .....	36
Table results .....	36
Exécuter les migrations .....	37
Modèles.....	39
Factories.....	40
Seeders .....	41
Authentification avec laravel Breeze .....	42
Installation.....	42
Création de la page dashboard.....	43
Routes et middlewares.....	43
Création d'un utilisateur avec les seeders .....	44
Création d'un utilisateur avec php artisan tinker : .....	45
Implémentation UI – layout de base .....	46
Changement du logo.....	46
Suppression menu nav .....	46
Footer .....	46
Page .....	48
Intégration services tiers : .....	49
Configuration et Variable d'environnement.....	49
Service container Oh Dear .....	50
Service container FLARE .....	53
Implémentation CRON.....	54
Création d'un service de gestion des résultats .....	55
Implémentation de la commande CRON .....	55
Planificateur de tâches .....	56
Configuration du Cron sur le Serveur .....	56
Laravel Telescope.....	57
Index.....	62
Table des illustrations .....	64
Sources & bibliographies.....	65

## ANALYSE PRÉLIMINAIRE

### INTRODUCTION

Afin d'assurer un suivi proactif des projets de nos clients, nous avons besoin d'un outil en ligne permettant de monitorer les sites et d'avoir une vue d'ensemble des projets que nous gérons.

Le projet est conçu pour répondre aux besoins croissants de surveillance et de gestion des performances des sites web dans un contexte professionnel.

Réalisé dans le cadre de la formation d'informaticien CFC, ce projet vise à développer une application web utilisant la stack TALL (TailwindCSS, AlpineJS, Laravel, Livewire) pour intégrer des services de monitoring tels que Oh Dear et Flare.

Ce choix s'appuie sur la nécessité d'offrir une solution complète pour le suivi des performances web, la surveillance des certificats SSL, et la détection des liens brisés, tout en proposant une interface utilisateur intuitive et sécurisée.

### OBJECTIFS

#### Phase d'analyse :

- Recueillir et analyser les exigences détaillées du projet.
- Concevoir l'expérience utilisateur (UX) et l'interface utilisateur (UI).
- Élaborer les schémas de la base de données pour supporter efficacement les fonctionnalités prévues.
- Identifier les intégrations nécessaires avec les services tiers (Oh Dear, Flare) et les défis techniques associés.
- Identifier les différentes phases d'implémentation et les stratégies de tests liées.

#### Phase d'implémentation :

- Concevoir et implémenter les schémas de base de données, en tenant compte des besoins d'évolutivité et de performance.
- Intégrer un système d'authentification sécurisé et flexible pour les utilisateurs de l'application (breeze).
- Créer des composants UI réactifs et accessibles en utilisant Tailwind CSS et Alpine.js, conformément aux maquettes UX/UI.
- Utiliser Livewire pour ajouter des fonctionnalités interactives sans compromettre la performance ou l'accessibilité.
- Intégrer efficacement les APIs d'Oh Dear et de Flare pour le monitoring et la gestion des incidents.
- S'assurer de la modularité et de la maintenabilité du code.
- Configurer des tâches automatisées pour la mise à jour des données.
- Allouer du temps pour l'ajustement basé sur les retours initiaux et pour intégrer des fonctionnalités supplémentaires non prévues.
- Effectuer des tests exhaustifs pour s'assurer de la stabilité, et de la performance de l'application.
- Rédiger une documentation technique complète et des manuels d'utilisation pour faciliter la prise en main de l'application.
- Analyser le déroulement du projet, recueillir les retours, et documenter les leçons apprises pour les projets futurs.

## ANALYSE /CONCEPTION

Selon les directives du cahier des charges, le projet sera structuré en plusieurs phases distinctes, chacune soumise à des tests de validation préalables avant de procéder à l'étape suivante. La planification initiale divise le projet en deux grandes sections :

**Analyse et Conception :** Cette étape initiale se consacre à l'élaboration des objectifs et à la conception globale du projet, y compris l'architecture de l'application et l'interface utilisateur.

**Implémentation et Tests :** Divisée en plusieurs sous-phases, cette section couvre l'exécution concrète du projet.

Chaque sous-phase comprend l'implémentation, les tests spécifiques à cette étape, et la documentation associée. Une stratégie de test dédiée est déployée pour chaque sous-phase afin de garantir la conformité aux objectifs fixés dans le cahier des charges et de valider le passage à l'étape suivante.

Chaque phase vise à accomplir intégralement ou partiellement les objectifs définis dans le cahier des charges, avec une validation basée sur les résultats des tests de phase.

Aucun budget ne sera défini dans ce projet, cependant, tout coût accessoire engagé (appel API, etc..) sera mentionné.

En complément à cette structure, toute la documentation du projet sera rédigée en français afin de maintenir la cohérence avec la langue du cahier des charges et de faciliter l'accessibilité pour les parties prenantes francophones. Cependant, pour promouvoir les bonnes pratiques de développement, toutes les variables, commentaires, fonctions et noms de code seront en anglais.

## PHASE 1 - CONCEPT

### ARCHITECTURE

#### Matériel :

- **Modèle** : Lenovo ThinkPad P16s Gen 2
- **Mémoire** : 64 Go
- **Stockage** : 1 To
- **Processeur** 13th Gen Inter Core I7-11370P, 1900 MHz

#### Environnement de travail :

- **Système d'exploitation** : Microsoft Windows 11 Professionnel
- **Version**: 10.0.22631 build 22631

#### Navigateur :

- Google Chrome
- Microsoft Edge

#### Base de données :

- **Système de gestion de base de données (SGBD)** : MySQL 8.2.0 en raison de sa compatibilité étendue avec Laravel et sa facilité d'intégration dans l'environnement WAMP.
- **Modélisation** : Utilisation de migrations Laravel pour définir la structure de la base de données. Les relations entre les tables seront définies à l'aide des Eloquent ORM pour faciliter les opérations CRUD.

#### Environnement de Développement Local :

- **Serveur WAMP** : configuration standard avec Apache, MySQL 8.2.13, et PHP 8.2. Cela assure une compatibilité optimale avec Laravel 10 et les autres technologies du stack TALL

#### Stack TALL :

- **Tailwind CSS** pour le design rapide et réactif de l'interface utilisateur.
- **Alpine.js** pour ajouter des comportements interactifs côté client avec une empreinte légère.
- **Laravel** 10.45.1 comme framework principal pour le back-end.
- **Livewire** pour créer une interface utilisateur dynamique avec la simplicité du développement front-end traditionnel.

#### Authentification et Sécurité :

- Mise en place d'une authentification par email et mot de passe, avec Laravel Breeze pour la gestion sécurisée des fonctionnalités d'authentification.
- Utilisation de politiques de sécurité strictes pour protéger les routes et les actions de l'application contre les accès non autorisés.
- Sécurisation des variables d'environnement sensibles grâce au fichier .env de Laravel, assurant que ces informations restent invisibles dans le code source et hors du versionning.

#### Services Tiers :

- Intégration des API **d'Oh Dear** et **Flare**, en utilisant des clients HTTP Laravel pour les requêtes et la gestion des réponses.

#### Développement et Maintenance :

- **IDE** : PhpStorm configuré avec des plugins spécifiques pour Laravel et Tailwind CSS, afin d'améliorer la productivité du développement.

- **Gestionnaire de Version** : GitHub, avec une utilisation simplifiée des branches pour ce projet local. Tous les commits peuvent être réalisés directement sur la branche main.
- **Outils de Conception** : Création de maquettes avec Wireframe et modélisation des bases de données avec draw.io pour visualiser l'architecture de l'application et la structure des données.

#### Documentation et Tests :

- **Documentation** : Utilisation de Markdown pour créer un fichier README.md clair, détaillant les étapes d'installation et de configuration, ainsi que la documentation du code et des API.
- **Tests** : Mise en place d'un plan de test pour valider chaque étape du projet, ce plan de test est voué à évoluer lors des phases d'implémentation pour avoir une vue d'ensemble des tests et résultats effectués tout au long de la réalisation du projet

#### Gestionnaire de paquets (node npm)

- Node.js 20.11.1

## BASE DE DONNÉES

Dans le cadre du projet, le modèle relationnel établit une base de données structurée comprenant trois tables :

- projects (projets / sites)
- results (résultats)
- sources

Les relations entre ces tables sont clairement définies. La table résultats est au cœur de l'architecture, enregistrant des entités issues des projets, qui sont ensuite alimentées par des données provenant de la table sources. Ce schéma reflète la nécessité d'une surveillance proactive et d'un suivi des projets clients, ce schéma permet l'ajout de sources et projets sans nécessiter de modification, augmentant l'évolutivité du projet.

Chaque résultat est directement lié à un projet et à une source, permettant une récupération et une gestion efficaces des données. Cette structuration des données est conforme aux bonnes pratiques et aux exigences techniques spécifiques du projet, assurant évolutivité et intégrité des données.

Les tables supplémentaires nécessaires au fonctionnement de Laravel et Breeze, automatiquement générées pour le framework et l'authentification, ne sont pas explicitement détaillées ici.

### MODÈLE CONCEPTUEL

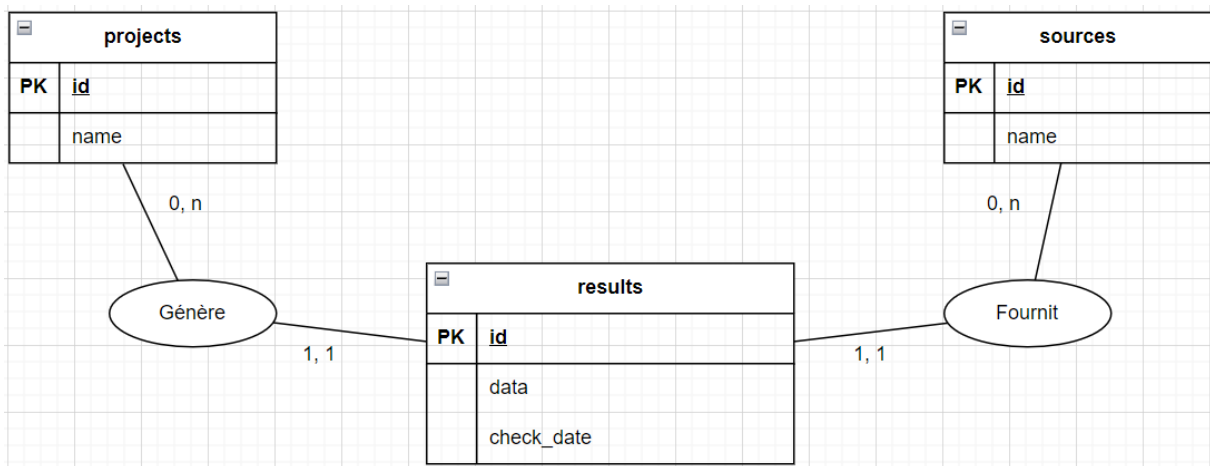


Figure 1 Schéma BDD modèle conceptuel

### MODÈLE RELATIONNEL

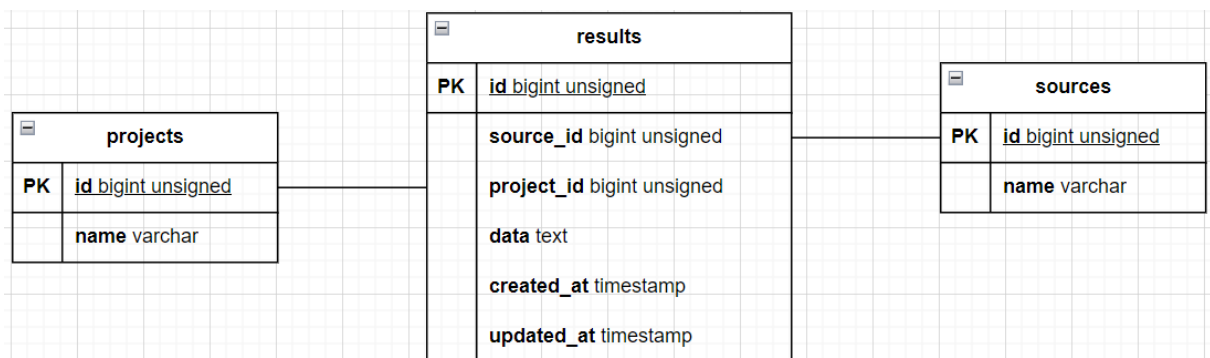


Figure 2 Schéma BDD modèle relationnel



## INTERFACE UTILISATEUR

Les prototypes de l'interface utilisateur sont conçus spécifiquement pour le mode desktop, avec une adaptation au responsive design et aux formats tablette/mobile prévus durant la phase d'implémentation.

Cette approche permet de se concentrer d'abord sur la conception desktop tout en intégrant la flexibilité nécessaire pour les ajustements responsive ultérieurs.

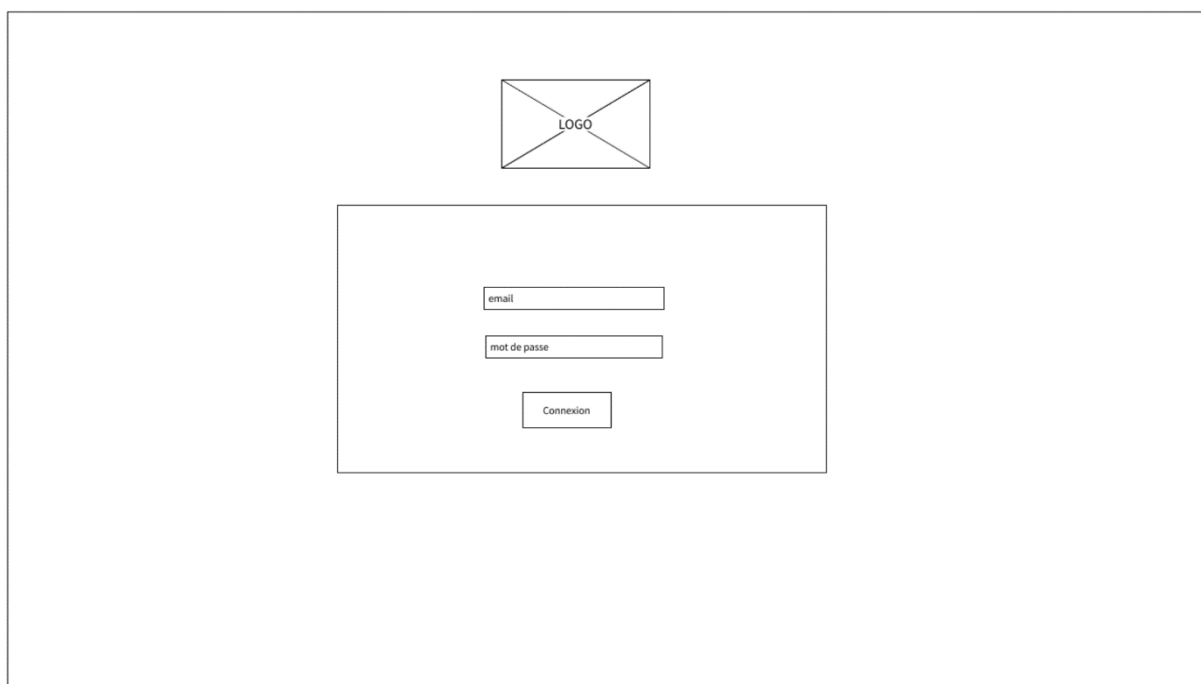
Des tests de validation dédiés au responsive design seront réalisés tout au long du projet, garantissant ainsi que l'application offre une expérience utilisateur optimale sur tous les appareils.

L'ajout de composants Blade réutilisables facilitera le développement en permettant la réutilisation de sections d'interface courantes, assurant ainsi une uniformité à travers le projet et une maintenance simplifiée

---

### PAGE DE CONNEXION (LOGIN)

Porte d'entrée de l'application, cette page présente un formulaire de connexion qui permet aux utilisateurs de s'identifier avec leur email et mot de passe pour accéder à l'ensemble des fonctionnalités.



The image shows a wireframe for a login page. It features a central rectangular area containing a logo placeholder (a rectangle with an 'X' and the word 'LOGO') at the top. Below the logo are two input fields: the first is labeled 'email' and the second is labeled 'mot de passe'. At the bottom of this central area is a button labeled 'Connexion'.

Figure 3Maquette authentification

## PAGE VUE D'ENSEMBLE

Cette page offre un tableau de bord synthétique qui présente les projets et indique leur état général grâce à des indicateurs visuels, permettant ainsi une évaluation rapide de la "santé" de chaque projet.

LOGO	Select	Soumettre	Menu utilisateur
Site	Performance OhDear	Code erreur flare	type d'erreur
FirstPoint.ch	USA	Apple Inc , Microsoft	Egregium esse egregium amicitia amicitia parem colebat numquam is anteibat virum per numquam se inferioris.
IKEA	Sweden	IKEA Furnitures , Spotify	
Spotify	Finland	Nokia Communications	Egregium esse egregium amicitia amicitia parem colebat numquam is anteibat virum per numquam se inferioris.
FirstPoint.ch	USA	Apple Inc , Microsoft	
IKEA	Sweden	IKEA Furnitures , Spotify	Egregium esse egregium amicitia amicitia parem colebat numquam is anteibat virum per numquam se inferioris.
IKEA	Sweden	IKEA Furnitures , Spotify	
Spotify	Finland	Nokia Communications	Egregium esse egregium amicitia amicitia parem colebat numquam is anteibat virum per numquam se inferioris.
Spotify	Finland	Nokia Communications	
FirstPoint.ch	USA	Apple Inc , Microsoft	Egregium esse egregium amicitia amicitia parem colebat numquam is anteibat virum per numquam se inferioris.
IKEA	Sweden	IKEA Furnitures , Spotify	
Documentation			

Figure 4 Maquette vue d'ensemble

## PAGE DÉTAILS

Elle fournit une analyse approfondie pour un projet ou site sélectionné, en détaillant des informations spécifiques telles que les messages d'erreur, les performances et autres données critiques pour le suivi et la maintenance

LOGO

Select

Soumettre

Menu utilisateur

< Vue d'ensemble

Uptime

Performance

Certificate health

Broken links

Mixed content

Lighthouse

Cron

Application health

DNS

Domain

Certificate transparency

Recharger les données

Message d'erreur	Date première occurrence	Date dernière occurrence	Fichier et ligne	Source url
ERROR MESSAGE	12.02.2018	12.02.2020	error.php ligne 12	url source message error
ERROR MESSAGE	12.02.2018	12.02.2020	error.php ligne 12	url source message error
ERROR MESSAGE	12.02.2018	12.02.2020	error.php ligne 12	url source message error

Documentation

Figure 5 Maquette détails

#### **Animations / effets :**

- Boutons : au survol changement de couleur et de curseur
- Boutons : au clic changement de couleur
- Effet visible de chargement lorsqu'il y a une recharge des données

#### **Composants Livewire :**

- Sélecteur de sites/projets et liste déroulante
- Affichage des données Oh Dear et Flare (sur toutes les pages)

### **RISQUES TECHNIQUES**

Les principaux risques techniques du projet incluent la complexité inhérente à la conception et au développement de certains tâches, un manque de certaines compétences, ainsi que l'incertitude quant au comportement de l'application en conditions réelles, puisqu'il n'est pas prévu de la déployer dans un environnement de production.

De plus, l'absence de tests unitaires, due à un manque d'expérience et au temps limité, augmente le risque d'erreurs non détectées lors du développement.

Le nombre de tâches inhérente au projet étant élevée, toutes ne pourront pas être réalisées selon les bonnes pratiques du développement.

**Priorisation des tâches** : Concentration sur les fonctionnalités clés et les composants critiques de l'application pour optimiser l'utilisation des ressources disponibles et minimiser les impacts des risques.

**Actions préventives** : Mise en place d'une revue régulière du code et d'analyses pour compenser l'absence de tests unitaires. Cela permettra d'identifier et de corriger les vulnérabilités potentielles ou les erreurs de conception dès les premiers stades du développement.

**Gestion par phases avec tests** : La structuration du projet en phases distinctes, incluant des tests spécifiques avant la transition, contribue positivement à la qualité.

**Simulation d'environnement de production** : Utilisation d'outils de simulation pour évaluer le comportement de l'application dans un environnement qui se rapproche le plus possible de la production, afin d'identifier les problèmes de performance ou de compatibilité en utilisant un serveur PHP local et intégré à Laravel.

## PLANIFICATION

La planification de ce projet est conçue pour garantir une progression méthodique et structurée à travers les différentes étapes de développement. Chaque phase est allouée avec une durée précise et liée à la complexité de la tâche, permettant une gestion efficace du temps et des ressources. Voici un résumé des grandes lignes de cette planification :

### Analyse Préliminaire (19.02.2024 - 19.02.2024) :

Cette phase initiale consiste à déterminer les objectifs globaux du projet et à préparer le terrain pour le développement. Elle est concise mais cruciale pour le succès global du projet.

### Phase 1 (20.02.2024 - 21.02.2024) :

Elle comprend le concept de base de données, l'interface utilisateur et l'architecture, et établit les fondations de l'application. C'est une phase intensive de planification et de conception qui dure deux jours.

### Phase 2 (22.02.2024 - 04.03.2024) :

Découpée en sous-phases d'implémentation, cette période est consacrée au développement concret des diverses fonctionnalités de l'application, allant de l'installation de Laravel à l'intégration des services tiers. Chaque sous-phase d'implémentation est méthodiquement allouée sur une journée pour assurer une attention dédiée à chaque aspect.

### Responsive (22.02.2024 - 04.03.2024) :

En parallèle avec d'autres tâches, l'adaptation responsive est un processus continu qui s'assure que l'application reste fonctionnelle et esthétique sur tous les types d'appareils.

### Documentation (20.02.2024 - 05.03.2024) :

La documentation est un processus continu et chaque phase est documentée en temps réel pour assurer l'exactitude des informations tout au long du projet. Cette pratique permet d'intégrer immédiatement les modifications et les ajouts apportés durant chaque étape. Une période est spécifiquement allouée en fin de projet, juste avant la date limite, pour peaufiner et finaliser tous les documents, s'assurant ainsi que toute la documentation est complète, précise et prête pour la révision finale.

### Tests (22.02.2024 - 04.03.2024) :

Les phases de tests se concentrent exclusivement sur la Phase 2 du projet, où chaque fonctionnalité et composant sont méticuleusement testés selon le plan de test préétabli. Ces périodes de test garantissent que chaque élément de l'application fonctionne correctement et répond aux critères de qualité avant de passer à l'étape suivante.

Chaque bloc de travail dans le planning est représenté par des barres orangées, et leur longueur indique la durée de l'effort prévu. Les points rouges signalent les échéances importantes. Ce modèle visuel aide à suivre l'avancement du projet et à identifier rapidement les sections qui requièrent une attention immédiate.

## PHASE 2 - CONCEPT

### PHASE 2.1 - INSTALLATION ET CONFIGURATION INITIALE DE LARAVEL

#### Implémentation :

- Configurer l'environnement avec PHP 8.2
- Utiliser Composer pour installer Laravel 10.x (installer composer si besoin)
- Configurer l'environnement (.env) pour la base de données, etc.
- Installer Tailwind CSS via npm et configurer selon les besoins du projet.

#### Documentation :

- Documenter étape par étape sur l'installation de Laravel 10.x, incluant la configuration du fichier .env pour la base de données.

#### Tests :

Configuration de l'environnement avec PHP 8.2

- **Procédure** : Exécuter **php -v** dans le terminal et vérifier que la version retournée est 8.2.
- **Critère de réussite** : Le système retourne PHP 8.2

Installation de Laravel 10.x avec Composer

- **Procédure** : Exécuter **composer show laravel/framework** et vérifier la version installée.
- **Critère de réussite** : Le système affiche une version commençant par 10.x.y

Configuration de l'environnement (.env)

- **Procédure** : Vérifier la présence et la configuration du fichier .env pour la connexion à la base de données et autres variables environnementales.
- **Critère de réussite** : Les variables de connexion à la base de données (DB\_CONNECTION, DB\_DATABASE, etc.) sont correctement définies et personnalisées pour le projet.

Installation et configuration de Tailwind CSS via npm

- **Procédure** : compiler les assets (**npm run dev**) et vérifier la présence de Tailwind dans le fichier CSS compilé.
- **Critère de réussite** : Le fichier CSS final inclut les styles de Tailwind, et les changements sont visibles sur l'interface utilisateur lorsque le serveur est lancé avec **php artisan serve**.

## PHASE 2.2 - IMPLÉMENTATION DE LA BASE DE DONNÉES

### Implémentation :

- Utiliser les migrations de Laravel pour reproduire le schéma défini dans la phase de conception
- Utiliser un typage fort

### Documentation :

- Documenter le schéma de base de données, en mettant en évidence l'utilisation des types de données de PHP 8.2

### Tests :

Vérification des migrations sans erreurs

- **Procédure** : Exécuter **php artisan migrate** pour appliquer toutes les migrations.
- **Critère de réussite** : Le terminal ne doit retourner aucune erreur et afficher un message confirmant l'exécution réussie de chaque migration.

Insertion et lecture de données tests

- **Procédure** : Utiliser les seeders Laravel pour insérer des données de test dans chaque table, puis effectuer des requêtes de lecture pour valider l'intégrité des données.
- **Critère de réussite** : Les données insérées doivent correspondre exactement aux données lues sans aucune perte ou modification, confirmant l'intégrité de la base de données.

## PHASE 2.3 - INTÉGRATION DE L'AUTHENTIFICATION

### Implémentation :

- Ajouter un système d'authentification (Breeze)
- Modifier les routes Laravel pour n'afficher les données qu'aux utilisateurs authentifiés, qui doivent être redirigé sur la page « vue d'ensemble »

### Documentation :

- Documenter le système d'authentification (manuel d'utilisation), notamment pour la création d'un utilisateur via php thinker

### Tests :

Création d'utilisateurs via seeders

- **Procédure** : Utiliser les seeders de Laravel pour créer des utilisateurs en base de données.
- **Critère de réussite** : Vérifier en base de données que les utilisateurs créés existent et que leurs informations correspondent à ce qui a été défini dans les seeders.

Création d'utilisateurs via Tinker

- **Procédure** : Utiliser Laravel Tinker pour créer des utilisateurs manuellement.
- **Critère de réussite** : Confirmer que l'utilisateur créé via Tinker est présent en base de données et peut se connecter à l'application.

Accès à la page "vue d'ensemble" sans Authentification

- **Procédure** : Créer la page « vue d'ensemble », configurer la route et y accéder sans être authentifié.
- **Critère de réussite** : L'utilisateur non authentifié est redirigé vers la page de connexion sans pouvoir accéder au dashboard.

## PHASE 2.4 - DÉVELOPPEMENT DE L'INTERFACE UTILISATEUR – LAYOUT DE BASE

### Implémentation :

- Implémenter le layout de base selon les maquettes établies lors de la phase d'analyse
- Utiliser Tailwind CSS pour le style et le responsive

### Documentation :

- Documenter les composants développés.

### Tests :

#### Test desktop

- **Procédure** : Afficher l'application sur différents écrans de bureau avec des résolutions variées.
- **Critère de réussite** : Le layout doit rester cohérent et conforme aux maquettes sur toutes les résolutions testées.

#### Test tablettes

- **Procédure** : Tester l'application sur des tablettes en orientations portrait et paysage.
- **Critère de réussite** : L'interface doit s'ajuster correctement aux deux orientations, en conservant une navigation et une interaction fluides.

#### Test smartphones

- **Procédure** : Vérifier le responsive sur différents modèles de smartphones, en portant une attention particulière aux tailles d'écran les plus populaires.
- **Critère de réussite** : Le design doit s'adapter à la taille de chaque écran, assurant une expérience utilisateur optimale sans besoin de zoomer ou de déplacer horizontalement la vue.

## PHASE 2.5 - INTÉGRATION DU SERVICES OH DEAR

### Implémentation :

- Utiliser le service container de laravel pour définir un service client personnalisé pour l'API.
- Utiliser des bloc try/catch pour gérer les exceptions
- Intégrer et configurer quelques sites pour le monitoring dans le fichier config

### Documentation :

- Documenter le processus d'intégration de l'API d'Oh Dear, y compris la configuration et la gestion des erreurs.

### Tests :

Tests d'Intégration spécifiques à Oh Dear

- **Procédure** : Exécuter des requêtes vers l'API Oh Dear pour tester l'intégration.
- **Critère de réussite** : Les réponses de l'API correspondent aux attentes, indiquant une intégration réussie.

Vérification des données retournées

- **Procédure** : Analyser les données retournées par l'API pour s'assurer qu'elles sont correctes et complètes.
- **Critère de réussite** : Les données reçues sont exactes et peuvent être utilisées pour le monitoring comme prévu.

Gestion des erreurs

- **Procédure** : Tester le système avec des scénarios d'erreur simulés pour évaluer la gestion des exceptions.
- **Critère de réussite** : Les erreurs sont correctement gérées, et l'application continue de fonctionner sans interruption.



## PHASE 2.6 - INTÉGRATION DU SERVICE FLARE

### Implémentation :

- Utiliser le service container de laravel pour définir un service client personnalisé pour l'API.
- Utiliser des bloc try/catch pour gérer les exceptions
- Intégrer et configurer quelques sites pour le monitoring dans le fichier config

### Documentation :

- Documenter le processus d'intégration de l'API de Flare, y compris la configuration et la gestion des erreurs.

### Tests :

#### Tests d'intégration spécifiques à Flare

- **Procédure** : Réaliser des appels API vers Flare pour vérifier l'intégration.
- **Critère de réussite** : L'application reçoit les réponses attendues, confirmant l'intégration réussie.

#### Tests d'intégration spécifiques à Flare

- **Procédure** : Exécuter des requêtes vers l'API Flare pour tester l'intégration.
- **Critère de réussite** : Les réponses de l'API correspondent aux attentes, indiquant une intégration réussie.

#### Vérification des données retournées

- **Procédure** : Analyser les données retournées par l'API pour s'assurer qu'elles sont correctes et complètes.
- **Critère de réussite** : Les données reçues sont exactes et peuvent être utilisées pour le monitoring comme prévu.

#### Gestion des erreurs

- **Procédure** : Tester le système avec des scénarios d'erreur simulés pour évaluer la gestion des exceptions.
- **Critère de réussite** : Les erreurs sont correctement gérées, et l'application continue de fonctionner sans interruption.

## PHASE 2.7 - MISE EN PLACE DES TÂCHES CRON ET FONCTIONNALITÉ DE RAFFRAICHISSEMENT

### Implémentation :

- Utiliser la planification de tâches de Laravel (**kernel.php**) pour définir des tâches cron
- Programmer des tâches pour l'exécution régulière de la mise à jour des données depuis les API
- S'assurer que les tâches sont réparties de manière à éviter une surcharge du serveur.

### Documentation :

- Description des tâches et intervalle de temps recommandé.

### Tests :

#### Configuration des tâches cron dans Laravel

- **Procédure** : Utiliser **kernel.php** pour programmer des tâches cron pour la mise à jour des données depuis les API.
- **Critère de réussite** : Les tâches sont correctement programmées et exécutées à intervalles réguliers sans provoquer de surcharge du serveur.

#### Tests manuels et surveillance des performances

- **Procédure** : Déclencher manuellement les tâches via **php artisan schedule:run** et utiliser Laravel Telescope pour surveiller l'impact sur les performances.
- **Critère de réussite** : Les tâches s'exécutent comme prévu sans impact négatif significatif sur les performances de l'application.

## PHASE 2.8 - INTÉGRATION DE LIVEWIRE

### Implémentation :

- Installer Livewire via Composer
- Création de composants et intégration.

### Documentation :

- Documenter l'utilisation de Livewire dans le projet, y compris les composants créés, pour faciliter leur réutilisation et maintenance.

### Tests :

#### Installation de Livewire via composer

- **Procédure** : Confirmer l'installation en vérifiant si Livewire est accessible dans le projet.
- **Critère de réussite** : Livewire est listé parmi les packages installés le de l'exécution de la commande **composer show**.

#### Création et intégration de composants Livewire

- **Procédure** : Tester les interactions dynamiques des composants Livewire dans différentes parties de l'application.
- **Critère de réussite** : Les interactions et mises à jour des composants se font sans rechargement complet de la page, démontrant leur bon fonctionnement.

## PHASE 2.9 – FINALISATION DE L'INTERFACE UTILISATEUR – INTÉGRATION SERVICES TIERS

### Implémentation :

- Finaliser le développement de l'interface utilisateur en se basant sur les designs préalablement établis, en accordant une attention particulière aux sections affichant les données provenant des services tiers et des composants Livewire

### Documentation :

- Mettre en évidence les principes d'UX/UI appliqués pour faciliter la navigation intuitive et présenter efficacement les informations critiques aux utilisateurs.

### Tests :

#### Conformité aux designs et intégration des données des services tiers

- **Procédure** : Assurer que l'interface utilisateur respecte les designs établis et intègre correctement les données provenant des services tiers.
- **Critère de réussite** : L'interface correspond aux maquettes, et les données de services tiers (comme Oh Dear et Flare) sont correctement affichées et mises à jour.

#### Tests de responsive sur différents appareils

- **Procédure** : Vérifier l'affichage et la fonctionnalité de l'interface sur des appareils de bureau, tablettes, et smartphones, dans diverses résolutions et orientations.
- **Critère de réussite** : Le design est fluide et responsive sur tous les appareils

#### Accessibilité et interactivité

- **Procédure** : Tester la taille et l'accessibilité des éléments interactifs sur des appareils mobiles pour s'assurer de leur facilité d'utilisation.
- **Critère de réussite** : Les boutons, liens et autres éléments interactifs ont une taille adéquate, permettant une interaction facile et précise.

#### Lisibilité et conformité UX

- **Procédure** : Évaluer la lisibilité des informations provenant des services tiers et la conformité de l'interface aux standards UX.
- **Critère de réussite** : Les informations critiques sont présentées clairement et de manière intuitive, favorisant une expérience utilisateur optimale.

## PHASE 2.10 - CORRECTIONS ET REFACTORISATION

### Implémentation :

- Réviser le code existant pour identifier les éventuelles optimisations / améliorations
- Voir les erreurs qui remontent via l'IDE

### Documentation :

- Mettre à jour la documentation pour refléter les changements réalisés pendant cette phase, en se concentrant sur les améliorations apportées. Partager les leçons apprises lors du processus, ce qui a fonctionné ou pas.

### Tests :

#### Révision et optimisation du code

- **Procédure** : Examiner le code existant pour identifier les opportunités d'optimisation et d'amélioration, et corriger les erreurs signalées par l'IDE.
- **Critère de réussite** : Le code est optimisé pour une meilleure performance et maintenabilité, sans introduire de nouveaux bugs.

#### Tests d'Intégration et fonctionnalités

- **Procédure** : Effectuer des tests d'intégration pour vérifier que tous les composants et fonctionnalités du site fonctionnent correctement après les modifications.
- **Critère de réussite** : Toutes les fonctionnalités et liens sur le site sont opérationnels et les tests d'intégration passent sans erreurs.

## PHASE 2.11 - DOCUMENTATION UTILISATEUR

### Implémentation :

- Finaliser toute la documentation utilisateur nécessaire pour le projet en vue du déploiement
- Inclure des captures d'écran lorsque c'est utile.

### Documentation :

- S'assurer que toute la documentation est bien organisée, facile à naviguer, et accessible aux développeurs et aux utilisateurs finaux.

### Tests :

Finalisation de la documentation utilisateur

- **Procédure** : Compléter la documentation utilisateur avec toutes les informations nécessaires, incluant des captures d'écran pour illustrer les fonctionnalités.
- **Critère de réussite** : La documentation est complète, claire, et fournit une aide pratique aux utilisateurs finaux pour naviguer et utiliser l'application efficacement.

Tests de clarté et de navigation

- **Procédure** : Passer en revue la documentation pour évaluer sa clarté, sa structure, et l'orthographe. Tester également tous les liens pour s'assurer qu'ils fonctionnent correctement.
- **Critère de réussite** : La documentation est bien structurée, sans fautes d'orthographe, et tous les liens sont fonctionnels et mènent aux bonnes sections ou ressources externes.

## PHASE 2.12 – DÉPLOIEMENT DE LA DOCUMENTATION

### Implémentation :

- Préparer la documentation pour le déploiement, en s'assurant que l'environnement de production est correctement configuré.

### Documentation :

- Documenter le processus de déploiement, y compris les étapes spécifiques

### Tests :

Test d'intégration dans l'application

- **Procédure** : Contrôler que le lien application - documentation fonctionne
- **Critère de réussite** : La documentation est bien affichée dans le navigateur

Résolution des problèmes identifiés

- **Procédure** : Analyser les retours d'utilisation pour identifier et corriger les bugs, réaliser des ajustements, ou modifier la mise en page si nécessaire.
- **Critère de réussite** : Les problèmes identifiés sont résolus.

### PHASE 2.13 - CORRECTIONS ET REFACTORISATION

#### Implémentation :

- Répondre aux problèmes découverts pendant l'utilisation.
- Ajustements, corrections de bugs, ou ajout de petite fonctionnalité afin de couvrir les objectifs du cahier des charges.

#### Documentation :

- Mettre à jour la documentation pour inclure les changements et les nouvelles fonctionnalités développées durant cette phase.

#### Tests :

Tests des ajustements et nouvelles fonctionnalités

- **Procédure** : Tester les modifications pour s'assurer qu'elles répondent aux exigences et n'introduisent pas de nouveaux problèmes.
- **Critère de réussite** : Les ajustements et nouvelles fonctionnalités fonctionnent comme prévu, sans effets secondaires indésirables.

### PHASE 2.14 - EVALUATION ET CLÔTURE DU PROJET

#### Évaluation :

- Réaliser une évaluation complète du projet pour s'assurer que tous les objectifs ont été atteints et que le produit final répond aux exigences de qualité, de performance, et d'usabilité.

#### Documentation :

- Compléter un rapport final qui résume les réalisations du projet, les leçons apprises, et les recommandations pour les projets futurs. S'assurer que toute la documentation est finalisée et bien organisée.



## RÉALISATION

//TODO texte intro

## INSTALLATION ET CONFIGURATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

## INSTALLATION DE WAMP

Version installée : **WampServer 64 BITS (x64) - 3.3.2**

Chemin d'installation : C:/wamp/

Cette version comprend :

- **Wampserver 3.3.2 installateur complet 64 bit**
- **Apache 2.4.58**
- **PHP 7.4.33/8.0.30/8.1.26/8.2.13/8.3.0**
- **MySQL 8.2.0**
- MariaDB 11.2.2
- PhpMyAdmin 5.2.1
- Adminer 4.8.1
- PhpSysInfo 3.4.3.

En gras, les outils nécessaires au fonctionnement de notre projet, les autres outils fournis avec la version ne seront pas utiles pour la réalisation de notre projet.

Une fois l'installation terminée, lancer **Wampserver** et attendre que les services aient fini de démarrer



Figure 6 WampServer

## CONFIGURATION D'APACHE 2.4.58

Chemin des fichiers de configuration d'Apache : C:\wamp\bin\apache\apache2.4.58\conf\

**Fichier httpd.conf :**

Le fichier **httpd.conf** est le fichier de configuration principal d'Apache. Il contient les directives de configuration qui dictent le fonctionnement du serveur web.

Ce fichier permet de configurer les modules chargés par Apache, les chemins des fichiers de log, les directives de sécurité, ainsi que les paramètres de performance comme le nombre maximal de connexions simultanées. Nous allons laisser la configuration de base et nous reviendrons sur celle-ci si des changements sont nécessaires.

**Fichier httpd-vhosts.conf**

Apache permet également de configurer des virtual hosts dans le fichier **httpd-vhosts.conf**. Les vhosts permettent d'héberger plusieurs sites web sur un seul serveur physique en configurant des domaines ou sous-domaines spécifiques pour pointer vers des répertoires différents sur le serveur.



Pour notre projet, bien que la configuration des vhosts soit possible pour simuler un environnement de production sur notre machine locale, nous avons choisi d'utiliser le serveur de développement intégré de Laravel en exécutant la commande **php artisan serve**.

Cette approche simplifie le processus de développement en évitant la nécessité de configurer des vhosts pour le projet. Le serveur de développement de Laravel est idéal pour un développement local et des tests, offrant une solution pratique sans configuration supplémentaire d'Apache pour les besoins immédiats de notre projet.

## CONFIGURATION DE MYSQL 8.2.0

Afin de rendre accessible la gestion de la base de données depuis le terminal, nous allons ajouter une nouvelle variable d'environnement (propriétés système, paramètres systèmes avancés, variables d'environnement.)

Dans variables système, ouvrir les variable Path et ajouter le chemin de mysql

`C:\wamp\bin\mysql\mysql8.2.0\bin`

Puis valider, nous pouvons maintenant accéder directement à mysql depuis le terminal :

Ouvrir le terminal (PowerShell), exécuter la commande :

```
mysql -u root -p
```

Valider, aucun mot de passe n'est défini pour l'utilisateur root.

Nous sommes bien dans l'environnement mysql avec la version 8.2.0

```
PS C:\Users\schei> mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 42
Server version: 8.2.0 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

Figure 7 MySQL config

## Création d'une base de données pour notre projet.

Nous allons directement créer ici la base de données nécessaire à notre application.

Effectuer la commande

```
CREATE DATABASE laravel ;
```

Puis vérifier que la base est bien créée : avec la commande la commande suivante pour lister toutes les tables

```
SHOW DATABASES ;
```

```
mysql> CREATE DATABASE laravel;
Query OK, 1 row affected (0.01 sec)

mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| blanclabo |
| information_schema |
| laravel |
| mysql |
| performance_schema |
| pspcrawl |
| sys |
+-----+
7 rows in set (0.01 sec)
```

Figure 8 MySQL databases

Nous pouvons voir que les 2 commandes (création et affichage des bases de données) sont bien exécutées et notre base de données **Laravel** est bien créée.

### CONFIGURATION DE PHP 8.2.13

Comme pour la configuration initiale de MySQL nous allons ajouter une nouvelle variable d'environnement Path afin de pouvoir accéder à PHP via le terminal. Ajouter le chemin de PHP :

```
C:\wamp\bin\php\php8.2.13
```

Dans le terminal effectuer la commande suivante pour afficher la version de PHP et vérifier que la configuration est effective :

```
php -v
```

```
PS C:\Users\schei> php -v
PHP 8.2.13 (cli) (built: Nov 21 2023 16:09:25) (ZTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.2.13, Copyright (c) Zend Technologies
    with Zend OPcache v8.2.13, Copyright (c), by Zend Technologies
    with Xdebug v3.2.2, Copyright (c) 2002-2023, by Derick Rethans
```

Figure 9 PHP version

Nous pouvons voir que PHP est bien configuré pour la version 8.2.13.

Il n'est pas nécessaire d'aller plus loin dans la configuration de PHP pour l'instant, si nécessaire, les différents fichiers de configuration se trouvent dans le chemin d'installation de PHP (C:\wamp\bin\php\php8.2.13\)

### Fichier php.ini

Le fichier principal de configuration PHP. Il contrôle plusieurs aspects du comportement de PHP, tels que les limites de mémoire, les extensions chargées, et les paramètres de date/heure (memory\_limit, upload\_max\_filesize, et date.timezone etc.. ). Nous reviendrons sur ces différents aspects selon les besoins de notre projet, nous laissons pour l'instant la configuration initiale.

## MISE EN PLACE DE LARAVEL

## INSTALLATION DE COMPOSER 2.7.1

L'installation de composer est nécessaire pour pouvoir créer un projet Laravel, de plus, PHP doit déjà être installé

Pour installer composer :

- Se rendre sur le site Composer : <https://getcomposer.org/download/>
- Télécharger et exécuter Composer-Setup.exe

Lors de l'installation, sélectionne le chemin de l'exécutable PHP correspondant à la version souhaitée :

Choose the command-line PHP you want to use:

C:\wamp\bin\php\php8.2.13\php.exe [Browse...](#)

This is the PHP in your path. Click Next to use it.

Une fois l'installation terminée, pas besoin d'ajouter les variables d'environnement Path, l'installateur ajoute automatiquement une nouvelle variable, simplement ouvrir le terminal et effectuer la commande suivante :

```
composer -v
```

```
PS C:\Users\schei> composer -v
```



```
Composer version 2.7.1 2024-02-09 15:26:28
```

### Figure 10 Composer version

Composer 2.7.1 est bien installé, nous allons maintenant pouvoir créer notre projet Laravel

## CRÉATION DU PROJET LARAVEL

Pour procéder à la création du projet Laravel, ouvrir un terminal et se rendre dans le dossier qui doit accueillir le projet :

```
composer create-laravel laravel TPI
```

TPI étant le nom du projet, composer va maintenant exécuter une série de commandes et créer tous les fichiers nécessaires au projet :

```

- Installing phpunit/php-invoker (4.0.0): Extracting archive
- Installing phpunit/php-file-iterator (4.1.0): Extracting archive
- Installing theseer/tokenizer (1.2.2): Extracting archive
- Installing sebastian/lines-of-code (2.0.2): Extracting archive
- Installing sebastian/complexity (3.2.0): Extracting archive
- Installing sebastian/code-unit-reverse-lookup (3.0.0): Extracting archive
- Installing phpunit/php-code-coverage (10.1.11): Extracting archive
- Installing phar-io/version (3.2.1): Extracting archive
- Installing phar-io/manifest (2.0.3): Extracting archive
- Installing myclabs/deep-copy (1.11.1): Extracting archive
- Installing phpunit/phpunit (10.5.10): Extracting archive
- Installing spatie/backtrace (1.5.3): Extracting archive
- Installing spatie/flare-client-php (1.4.4): Extracting archive
- Installing spatie/ignition (1.12.0): Extracting archive
- Installing spatie/laravel-ignition (2.4.2): Extracting archive
45/111 [=====>-----] 40%
```

Figure 11 Laravel install

Une fois l'installation terminée, nous pouvons vérifier la version des différents composants dont la version Laravel à l'aide de la commande :

```
composer show laravel/framework
```

```

PS C:\sites\TPI> composer show laravel/framework
name      : laravel/framework
descrip.  : The Laravel Framework.
keywords : framework, laravel
versions : * v10.45.1
```

Figure 12 Laravel version

Nous avons bien une version 10, ici la version 10.45.1. et nous pouvons lancer le serveur de développement intégré à Laravel à l'aide de la commande :

```
php artisan serve
```

Puis en se rendant à l'adresse <http://127.0.0.1:8000/> sur notre navigateur nous voyons la page de notre projet :

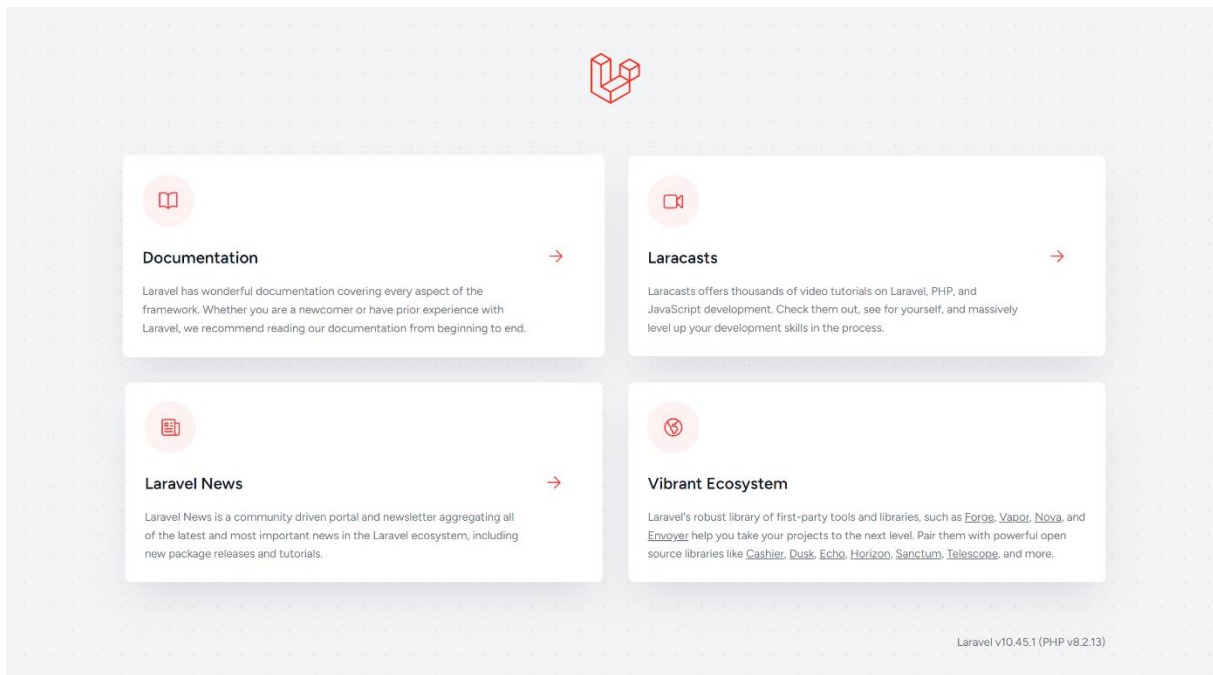


Figure 13 Laravel page principale

Nous avons désormais terminé la création de notre projet Laravel

La structure d'un projet Laravel est bien organisée, chaque dossier et fichier joue un rôle spécifique. Voici une liste des principaux dossiers et fichiers dans un projet Laravel standard, ainsi que leur contenu et leur rôle :

### Dossier /app

Ce dossier contient le code source de l'application. Il est structuré en plusieurs sous-dossiers pour les modèles, les contrôleurs, les notifications, etc.

- **/Models** : Contient les modèles Eloquent de notre application. Les modèles représentent les tables de notre base de données et permettent d'interagir avec ces données.
- **/Http** : Contient les contrôleurs et les middlewares. Les contrôleurs gèrent la logique des requêtes HTTP.
  - o **/Controllers** : Chaque contrôleur traite les entrées et les retours de vue ou de données au client.
  - o **/Middleware** : Les middlewares filtrent les requêtes HTTP entrantes dans l'application.

### Dossier /config

Contient tous les fichiers de configuration du projet. Ces fichiers permettent de configurer les aspects de l'application comme la base de données, le système de fichier, les services, etc..

### Dossier /database

- **/migrations** : Contient les fichiers de migration qui permettent de définir et de modifier la structure de la base de données au fil du temps.
- **/seeds** : Les fichiers seeders permettent de remplir les tables de la base de données avec des données initiales.

### Dossier /public

Ce dossier contient le fichier index.php qui est le point d'entrée de toutes les requêtes de l'application. Il contient également les ressources comme les images, les fichiers JavaScript et CSS.

## Dossier /resources

- **/views** : Contient les templates Blade de notre application. Blade est le moteur de template de Laravel.
- **/lang** : Contient les fichiers de langues pour l'internationalisation de notre application.
- **/sass, /js** : Contiennent les fichiers sources SASS/SCSS et JavaScript qui seront compilés et placés dans le dossier public.

## Dossier /routes

Contient tous les fichiers de routage de l'application. Laravel sépare le routage en plusieurs fichiers comme web.php pour les routes web et api.php pour les routes d'API.

## Fichier .env

Un fichier de configuration des variables d'environnement. Cela inclut les configurations de base de données, les clés API, et d'autres données sensibles.

## Fichier composer.json

Définit les dépendances PHP du projet et quelques scripts supplémentaires pour Composer.

## Fichier artisan

Un script PHP en ligne de commande qui permet d'exécuter des tâches Laravel, comme la création de contrôleurs, de modèles, de migrations, et d'exécuter des migrations ou des seeders.

## MISE EN PLACE DANS L'IDE

L'utilisation de PhpStorm comme environnement de développement intégré (IDE) pour notre projet Laravel offre une multitude d'avantages, notamment pour la gestion du code, l'intégration de la base de données etc.. Nous devons cependant faire quelque réglages et configuration avant d'aller plus loin dans le développement du projet

### Ouverture du projet

- Ouvrir PhpStorm et sélectionner File > Open.
- Naviguez jusqu'au dossier contenant notre projet
- Faire un clic droit sur le dossier et choisir « Open Folder as PhpStorm Project ».
- Notre projet est maintenant ouvert dans PhpStorm, prêt pour le développement.

### Configuration de la Base de Données

- Cliquer sur l'onglet Database à droite de l'IDE.
- Cliquer sur le + pour ajouter une nouvelle source de données et sélectionner MySQL.
- Dans la fenêtre de configuration nous devons renseigner les informations de connexion :
  - o **Host** : localhost
  - o **Port** : 3306 (port par défaut de MySQL)
  - o **User** : root
  - o **Password** : Laisser vide car nous n'avons pas défini de mot de passe

Tester la connexion pour s'assurer que tout est correctement configuré et appliquer les changements.

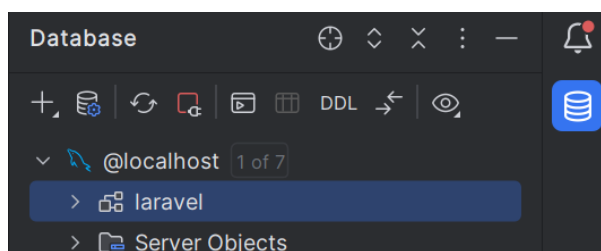


Figure 14 BDD connexion PhpStorm

## VARIABLES D'ENVIRONNEMENT (.ENV)

Parmi les commandes exécutées par composer à l'installation de laravel, il y en a une qui s'occupe de copier le fichier .env.exemple pour créer un fichier .env et ce fichier est automatiquement ajouté dans le .gitignore

Le fichier .env joue un rôle crucial dans la configuration de notre application Laravel, notamment pour les accès à la base de données. Pour le configurer :

Ouvrir le fichier .env dans PhpStorm, modifier les valeurs suivantes enregistrer les modifications.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

## CONFIGURATION DE TAILWIND CSS

Avant de pouvoir utiliser Tailwind CSS, nous devons nous assurer que Node.js et npm sont correctement installés, car ils seront nécessaires pour installer Tailwind CSS et ses dépendances.

### Installation de node.js et npm

- Aller sur le site node.js et télécharger la version dernière version LTS pour notre environnement (Windows) ici la version **20.11.1-x64** et lancer l'installation en s'assurant que « **npm package manager** » est bien sélectionné lors de l'installation
- Les variables d'environnement node.js et npm seront automatiquement ajoutées aux variables d'environnement Windows pour pouvoir exécuter les commandes liées depuis le terminal

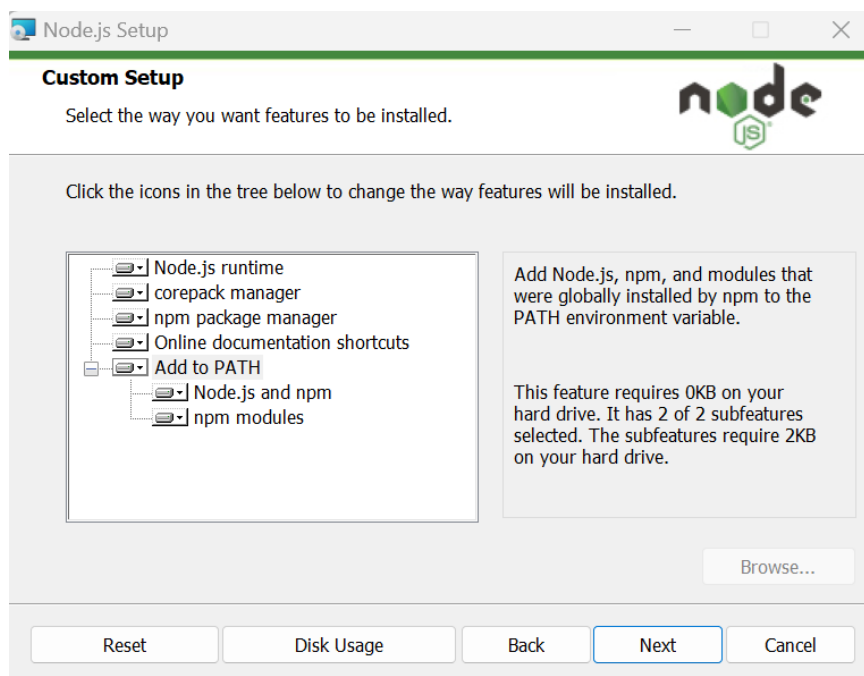


Figure 15 Node.js installation

Une fois l'installation terminée, ouvrir un terminal et exécuter les commandes suivantes :

```
node -v
npm -v
```

```
PS C:\Users\schei> node -v
v20.11.1
PS C:\Users\schei> npm -v
10.2.4
PS C:\Users\schei>
```

Figure 16 node et npm versions

### Installation de tailwind CSS :

Dans le dossier du projet, exécuter la commande :

```
npm install -D tailwindcss postcss autoprefixer
```

Une fois l'installation terminée, exécuter la commande suivante :

```
npx tailwindcss init -p
```

Dans le fichier *tailwind.config.js*, ajouter les chemins de fichiers qui doivent être prise en compte lors du build :

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    './resources/**/*.blade.php',
    './resources/**/*.js',
    './resources/**/*.vue',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Enregistrer et ajouter ensuite les directives Tailwind CSS en haut du fichier css (*/resources/css/app.css*) :

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Il faut ensuite ajouter la ligne suivante dans la balise head (*resources/views/welcome.blade.php*)

```
@vite('resources/css/app.css')
```



Ensuite, pour vérifier que tout est fonctionnel, il nous suffit de rajouter un petit bloc de code et quelque classe Tailwind dans le fichier `resources/views/welcome.blade.php` :

```
<div class="bg-red-600 flex justify-center">
  <h1 class="text-green-600 text-9xl">HelloWorld</h1>
</div>
```

Exécuter la commande pour build :

```
npm run dev
```

Lancer le serveur intégré :

```
php artisan serve
```

Aller sur notre navigateur à l'adresse **http://127.0.0.1:8000/** et observer le résultat :

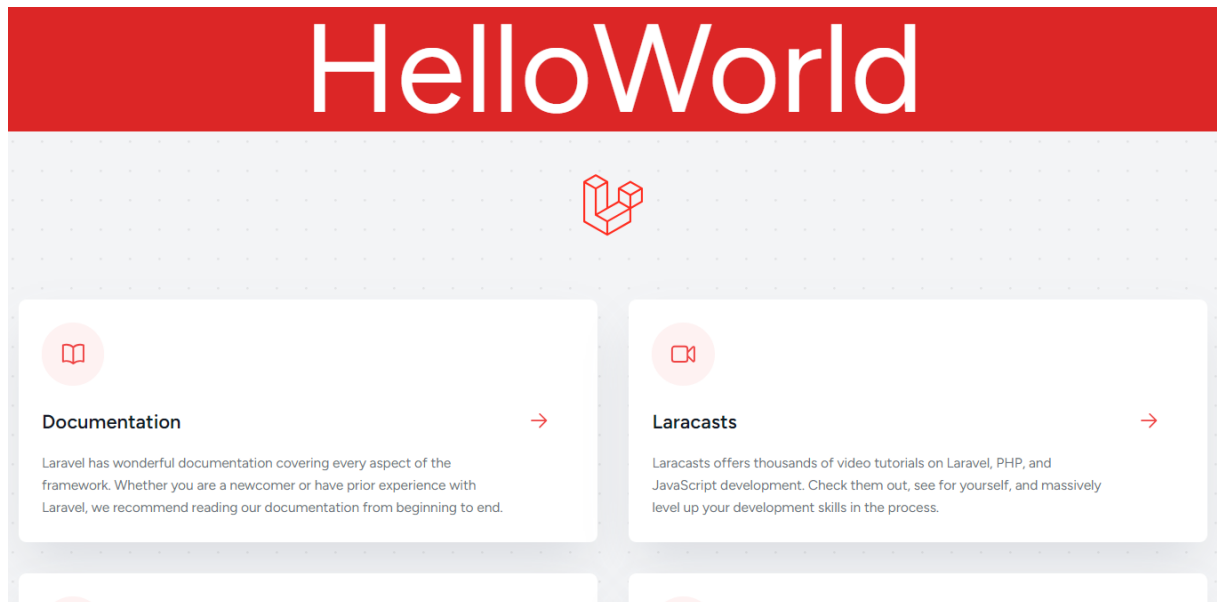


Figure 17 Premier build Tailwind CSS

## IMPLÉMENTATION DE LA BASE DE DONNÉES

Lors de l'installation initiale, Laravel crée automatiquement quatre tables, qui servent de fondement à de nombreuses fonctionnalités essentielles de l'application. Ces tables sont :

### users :

- Cette table stocke les informations essentielles des utilisateurs, comme les noms, les adresses email, et les mots de passe. Bien que notre application n'implémente pas immédiatement un système d'authentification, la présence de cette table facilite grandement l'ajout de cette fonctionnalité plus tard, sans nécessiter de configuration supplémentaire.

### password\_resets :

- Elle joue un rôle clé dans le processus de réinitialisation des mots de passe, en stockant les jetons de réinitialisation.

### failed\_jobs :

- Laravel utilise cette table pour enregistrer les tâches planifiées qui échouent.

### personal\_access\_tokens :

- Elle stocke les jetons d'accès personnel utilisés pour l'authentification API, offrant une méthode sécurisée pour accéder aux services de l'application sans exposer les identifiants de connexion de l'utilisateur.

## CRÉATIONS DES MIGRATIONS DE NOTRE BASE DE DONNÉES

Dans le développement d'application web avec Laravel, les fichiers de migration sont des composants essentiels pour la gestion de la base de données, ils permettent de :

- Créer et modifier des tables
- Maintenir la cohérence
- Faciliter le déploiement et le rollback
- Automatiser le processus de développement

Selon le schéma relationnel établis lors de la phase de conception, nous devons créer 3 tables.

- Projects
- Results
- Source

Lors de l'implémentation de tables, il est crucial de prêter attention à l'ordre de création et aux relations entre elles. Il est recommandé de commencer par créer les tables qui n'entretiennent pas de dépendances directes. Par conséquent, l'ordre de création suggéré est le suivant :

1. Projects
2. Sources
3. Results

Cette organisation assure que les tables **projects** et **sources** peuvent être établies sans dépendre d'autres tables, tandis que la table **results**, qui peut nécessiter des clés étrangères reliant **projects** ou **sources**, est créée en dernier. Cet ordre reflète une approche méthodique pour assurer l'intégrité relationnelle et faciliter la gestion des dépendances dans la base de données.

## TABLE PROJECTS

Nous allons maintenant créer la table projects :

- 1- Générer la migration : exécuter la commande suivante directement dans le terminal de PhpStorm pour créer le fichier de migration :

```
php artisan make:migration create_projects_table --create=projects
```

L'option **--create=projects** indique à Artisan de préparer un fichier de migration qui contiendra par défaut un squelette pour la table projects dans la base de données.

Cette commande crée un fichier de migration dans le répertoire *database/migrations* avec un nom qui commence par une date et une heure, suivies de **create\_projects\_table**

L'ordre des migrations s'effectuent selon la date et l'heure de création des fichiers de migrations.

*database/migrations/2024\_02\_23\_073540\_create\_projects\_table.php* :

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('projects', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('projects');
    }
};
```

**Déclarations use** : Importent les classes nécessaires pour définir la structure de la migration

**up()** : Définit les opérations à effectuer lorsque la migration est appliquée (**php artisan migrate**)

**down()** : Définit les opérations à effectuer pour annuler la migration (**php artisan migrate:rollback**)

**timestamp()** : La méthode crée deux colonnes équivalentes nommées **'created\_at'** et **'updated\_at'**, nous n'en avons pas besoin dans le cadre de notre projet nous pouvons supprimer cette ligne.

**Id()** : La méthode id() est un raccourci pour la méthode bigIncrements. Par défaut, cette méthode génère une colonne id servant de clé primaire auto-incrémentée.

Nous ajouterons uniquement la colonne **'name'** qui contiendra du texte (string) : `$table->string('name');`

## TABLE SOURCES

Même processus que pour la table **projects** :

Création de la migration :

```
php artisan make:migration create_sources_table --create=sources
```

Puis, suppression de la ligne **timestamps()** et ajout de la ligne **'name'** :

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('sources', function (Blueprint $table) {
            $table->id();
            $table->string('name');
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('sources');
    }
};
```

## TABLE RESULTS

Nous avons maintenant 2 tables **sources** et **projects** créées, nous pouvons ajouter la 3<sup>ème</sup> tables **results** ainsi que les associations.

```
php artisan make:migration create_results_table --create=results
```

Dans le fichier de migration créée, nous allons modifier le méthode **up()** :

```
public function up(): void
{
    Schema::create('results', function (Blueprint $table) {
        $table->id();
        $table->foreignId('project_id')->constrained()->onDelete('cascade');
        $table->foreignId('source_id')->constrained()->onDelete('cascade');
        $table->longText('data');
        $table->timestamps();
    });
}
```

Associations :

- **foreignId()** : Cette méthode crée une colonne de type entier signé qui est destinée à être utilisée comme une clé étrangère dans la table.
- **constrained()** : Cette méthode applique automatiquement la contrainte de clé étrangère, en liant la colonne de clé étrangère à sa table parent correspondante.

- **onDelete()** : Cette méthode définit le comportement à adopter lorsque l'entrée référencée dans la table parente est supprimée.

Ajout de la colonne '**data**' de type **string** pour accueillir les données reçues de l'api et nous laissons la méthode **timestamps()** pour connaître la date des données reçues, cette méthode va créer 2 colonnes **created\_at** et **updated\_at** de type **timestamp**.

## EXÉCUTER LES MIGRATIONS

Maintenant que nous avons créé les fichiers de migrations, nous allons pouvoir les exécuter à l'aide d'une commande Artisan.

Dans le terminal, exécuter la commande :

```
php artisan migrate
```

Lors de l'exécution de la commande `php artisan migrate` pour appliquer les migrations à la base de données, une erreur de type `Illuminate\Database\QueryException` a été rencontrée, indiquant un problème de syntaxe ou une violation d'accès avec le message suivant :

```
SQLSTATE[42000]: Syntax error or access violation: 1071 La clé est trop longue. Longueur maximale: 1000
(Connection: mysql, SQL: alter table `users` add unique `users_email_unique`(`email`))
```

Type : Erreur de syntaxe SQL

Cause : Cette erreur survient lorsque la longueur de la clé utilisée pour un index unique dépasse la limite maximale autorisée par MySQL

Solution : ajuster la configuration de notre base de données :

Dans le fichier `app/Providers/AppServiceProvider.php` :

Ajouter l'import suivant :

```
use Illuminate\Support\Facades\Schema;
```

Et dans la fonction **boot()** ajouter la ligne suivante :

```
Schema::defaultStringLength(191);
```

Enregistrer et relancer les migrations :

Lors du premier essai de migration, la table `users` a été créée, nous avons désormais une nouvelle erreur :

```
SQLSTATE[42S01]: Base table or view already exists: 1050 La table 'users' existe déjà
```

Il suffit juste d'exécuter la commande de migration avec l'option `fresh`. Cette commande va supprimer les tables existantes (fonction `down()` des fichiers de migration)

```
php artisan migrate:fresh
```

```

PS C:\Sites\TPI> php artisan migrate:fresh

Dropping all tables ..... 18ms DONE

[INFO] Preparing database.

Creating migration table ..... 8ms DONE

[INFO] Running migrations.

2014_10_12_000000_create_users_table ..... 44ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 5ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 47ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 44ms DONE
2024_02_23_073540_create_projects_table ..... 5ms DONE
2024_02_23_081555_create_sources_table ..... 7ms DONE
2024_02_23_082625_create_results_table ..... 167ms DONE

```

Figure 18 Résultat migration

Notre migration s'est déroulée avec succès, et nous pouvons notifier que les tables ont bien été supprimées avant d'être créées. Nous pouvons contrôler directement avec un « refresh » de la base de données dans l'IDE :

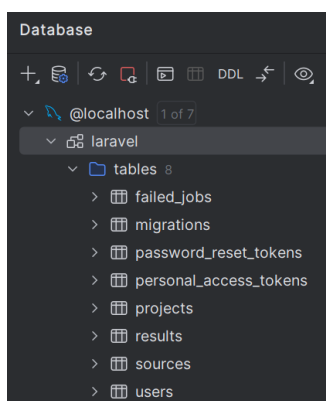


Figure 19 BDD after migration

## MODÈLES

Dans le cadre du framework Laravel, les modèles jouent un rôle crucial dans l'interaction avec la base de données au sein de l'architecture MVC (Modèle-Vue-Contrôleur). Voici les principaux rôles des modèles dans Laravel :

- Interaction avec la base de données
- Représentation des tables
- Gestion des relations
- Validation des données
- Comportements personnalisés
- Événements et Observateurs

Nous allons donc créer un modèle pour les 3 tables que nous avons créées dans le cadre de notre projet dans notre base de données, pour se faire, il faut exécuter les commandes suivantes :

```
php artisan make:model Source
php artisan make:model Project
php artisan make:model Result
```

### Modèle Source et Project :

Ces 2 modèles sont similaires, ils ont tous 2 une relation avec le modèle Result :

```
class Project extends Model
{
    public $timestamps = false;

    public function results()
    {
        return $this->hasMany(Result::class);
    }
}
```

**hasMany()** : source et project sont relatifs à plusieurs résultats

Nous n'avons pas intégré les colonnes timestamps dans cette migration, nous devons donc désactiver cette fonction dans les 2 modèles :

```
public $timestamps = false;
```

### Modèle Result :

```
class Result extends Model
{
    public function project()
    {
        return $this->belongsTo(Project::class);
    }

    public function source()
    {
        return $this->belongsTo(Source::class);
    }
}
```

**belongsTo()** : Un résultat appartient à un project et à une source

## FACTORIES

Pour vérifier l'intégrité des données, nous allons insérer des « fausses » données dans nos 3 tables créées dans le cadre de ce projet. Nous allons utiliser les seeders et factories, des concepts propres à Laravel et très utiles lors des phases de développement.

Pour créer les factories, nous allons exécuter les commandes suivantes :

```
php artisan make:factory ProjectFactory --model=Project
php artisan make:factory SourceFactory --model=Source
php artisan make:factory ResultFactory --model=Result
```

### Factories Project et source :

```
class ProjectFactory extends Factory
{
    protected $model = Project::class;

    public function definition(): array
    {
        return [
            'name' => $this->faker->word
        ];
    }
}
```

**Faker()** méthodes qui génère des fausses données

### Factory Results :

```
use App\Models\Project;
use App\Models\Source;

class ResultFactory extends Factory
{
    public function definition(): array
    {
        return [
            'project_id' => Project::factory(),
            'source_id' => Source::factory(),
            'data' => $this->faker->paragraph,
        ];
    }
}
```

Importation du modèle de Projet et Source

Lors de l'exécution de ce code, les factories de project et source seront aussi appelé, ce qui aura pour résultat de créer des projets et sources directement, qui seront liés aux résultats



## SEEDERS

Ici pas besoin de créer des seeders pour source et projet, car nous allons créer directement ces enregistrements via le seeder de Result, c'est lui qui s'occupera de créer des instances via la factory de result

```
php artisan make:seeder ResultSeeder
```

**Result :**

```
use App\Models\Result;

class ResultSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        Result::factory()->count(20)->create();
    }
}
```

**count(20)** représente le nombre de fois qu'on veut appeler la factory

Puis dans le fichier `database/seeder/DatabaseSeeder.php`

```
namespace Database\Seeders;

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        $this->call([
            ResultSeeder::class,
        ]);
    }
}
```

On appelle le seeder Result et une fois la création terminée, on peut aller dans la base de données contrôler que nous avons bien 20 results mais aussi 20 sources et 20 projects.

sources		projects		results	
id	name	id	name	id	project_id
1	1 saepe	1	1 cupiditate	1	1
2	2 doloremque	2	2 quis	2	2
3	3 veniam	3	3 ut	3	3
4	4 libero	4	4 saepe	4	4
5	5 voluptatem	5	5 commodi	5	5
6	6 vero	6	6 enim	6	6
7	7 sed	7	7 esse	7	7
8	8 consequatur	8	8 dolorem	8	8
9	9 quis	9	9 error	9	9
10	10 sit	10	10 officiis	10	10
11	11 placeat	11	11 eos	11	11
12	12 ut	12	12 ab	12	12
13	13 tempore	13	13 in	13	13
14	14 deleniti	14	14 ratione	14	14
15	15 velit	15	15 et	15	15
16	16 rem	16	16 expedita	16	16
17	17 id	17	17 tenetur	17	17
18	18 quis	18	18 consequatur	18	18
19	19 explicabo	19	19 est	19	19
20	20 sit	20	20 illo	20	20

Figure 20 Données de tests

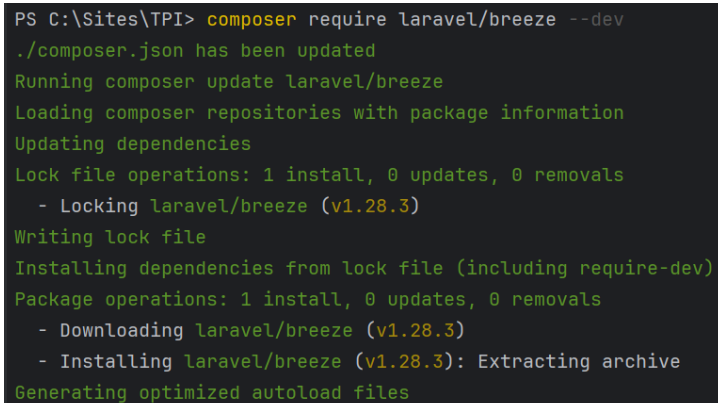
## AUTHENTIFICATION AVEC LARAVEL BREEZE

Laravel Breeze est un système d'authentification qui offre une installation simplifiée et un ensemble complet de fonctionnalités d'authentification, telles que la connexion, l'inscription, la réinitialisation de mot de passe, et la vérification d'email,

### INSTALLATION

Pour installer Laravel Breeze dans le terminal exécuter la commande suivante :

```
composer require laravel/breeze --dev
```



```
PS C:\Sites\TPI> composer require laravel/breeze --dev
./composer.json has been updated
Running composer update laravel/breeze
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking laravel/breeze (v1.28.3)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Downloading laravel/breeze (v1.28.3)
  - Installing laravel/breeze (v1.28.3): Extracting archive
Generating optimized autoload files
```

Figure 21 Installation Breeze

Une fois le package Breeze installé, nous devons "publier" les ressources nécessaires au fonctionnement de Breeze, telles que les vues, les routes, et les contrôleurs. Cela se fait en exécutant la commande suivante :

```
php artisan breeze:install
```

## CRÉATION DE LA PAGE DASHBOARD

Dans le dossier ressources/view, nous avons pour l'instant une page « welcome.blade.php » nous allons la renommer en dashboard.blade.php car nous ne souhaitons pas avoir une page welcome mais une page vue d'ensemble pour nos domaines, nous reviendrons sur sa composition plus tard et nous l'appelons « dashboard »

Une fois renommée, nous pouvons relancer le serveur de développement et se rendre sur notre navigateur :

Nous avons maintenant une erreur :

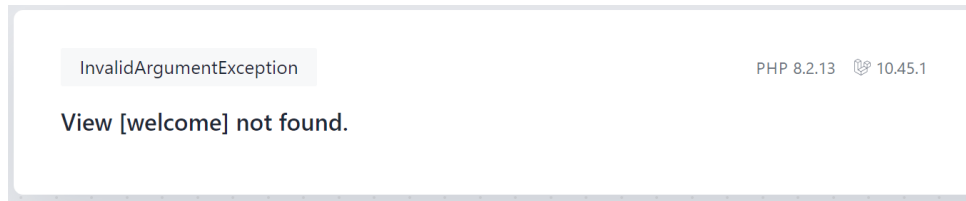


Figure 22 Erreur route Breeze

Il faut configurer la route pour qu'elle pointe sur la nouvelle page « dashboard », actuellement elle pointe sur welcome, qui n'existe plus.

## ROUTES ET MIDDLEWARES

Dans le fichier de routes, nous devons commencer par rediriger l'utilisateur vers la vue login s'il n'est pas connecté, sinon il sera redirigé sur la page dashboard

```
Route::get('/', function () {
    return auth()->check() ? redirect('/dashboard') : view('auth.login');
});
```

Ensuite nous allons mettre en place les middleware 'auth' et 'verified'

- Le middleware **auth** s'assure que l'utilisateur est connecté. Si un utilisateur non authentifié tente d'accéder au tableau de bord, il sera redirigé vers la page de connexion.
- Le middleware **verified** vérifie si l'email de l'utilisateur a été vérifié. C'est une étape supplémentaire pour renforcer la sécurité de votre application en s'assurant que seuls les utilisateurs ayant validé leur adresse email peuvent accéder au tableau de bord.

```
Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');
```

## CRÉATION D'UN UTILISATEUR AVEC LES SEEDERS

Dans le fichier DatabaseSeeder que nous avons utilisé précédemment pour peupler notre base de données, nous allons créer un utilisateur de base pour notre application :

```
use App\Models\User;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        User::factory()->create([
            'name' => 'Florent',
            'email' => 'fscheibler@firstpoint.ch',
            'password' => '$2y$12$IjjXFP.9DZzcVuGrqC0pW.0TuK5v4rrY0q6G50VzcseR9G50li1SW',
        ]);
    }
}
```

Pour le hashage du mot de passe, nous avons utiliser php thinker en utilisant bcrypt

```
php artisan tinker
Hash::make('motDePasse');
```

Il suffit ensuite de copier/coller le résultat dans le seeder et nous executons la commande suivante pour vider la base de données des données de test et pour créer notre utilisateur :

```
php artisan migrate:fresh --seed
```

Et nous pouvons tester de nous connecter avec notre utilisateur en relançant le serveur (php artisan server) :

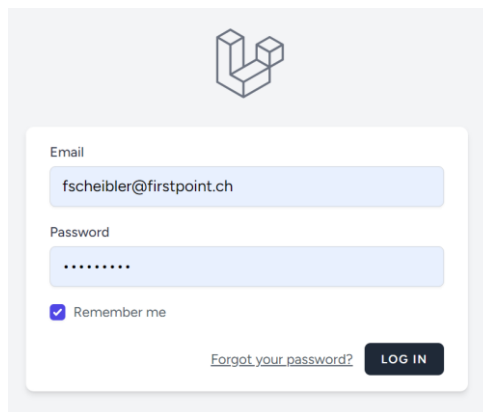


Figure 23 Page login

Et nous sommes désormais redirigé vers notre page « dashboard » :

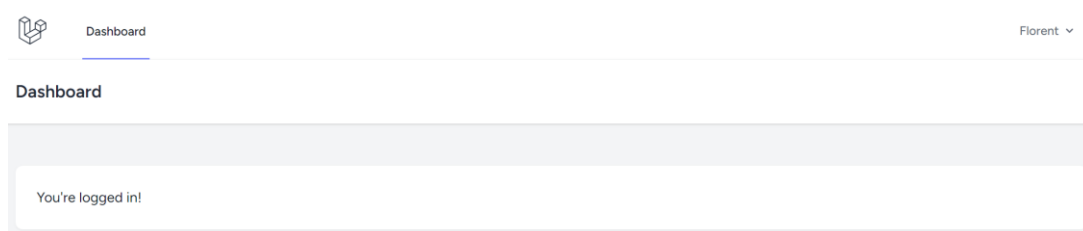


Figure 24 Vue Dashboard

## CRÉATION D'UN UTILISATEUR AVEC PHP ARTISAN TINKER :

Dans un terminal :

```
php artisan tinker
```

Et exécuter la commande ci-dessous avec les informations souhaitées :

```
User::create(["name"=>"laravel","email"=>"laravel@laravel.com","password"=>bcrypt("123456")]);
```

Pour contrôler que notre utilisateur est bien créé, toujours avec php artisan tinker, exécuter la commande suivante :

```
App\Models\User::where('email', 'laravel@laravel.com')->first();
```

L'utilisateur sera alors retourné :

```
> App\Models\User::where('email', 'laravel@laravel.com')->first();
= App\Models\User {#5061
  id: 2,
  name: "laravel",
  email: "laravel@laravel.com",
  email_verified_at: null,
  #password: "$2y$12$rZmF5/zCf06KAJ29LYV0yeBaQnk/BI.VXwuKuWnYp5AMLYeJyvV0u",
  #remember_token: null,
  created_at: "2024-02-25 15:38:38",
  updated_at: "2024-02-25 15:38:38",
}
```

Figure 25 Php tinker affichage utilisateur

## IMPLÉMENTATION UI – LAYOUT DE BASE

Tout au long de cette phase, le serveur de développement sera lancé pour avoir un aperçu direct des modifications en live. De plus, il sera peut-être nécessaire de build le projet pour que les changements soient pris en compte.

### CHANGEMENT DU LOGO

Dans `resources/views/components/application-logo.blade.php` :

Supprimer le contenu existant et ajout du code svg de notre logo

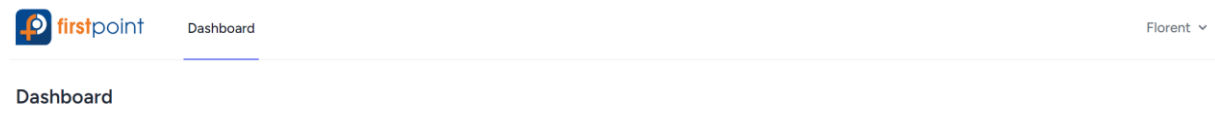


Figure 26 Header logo

### SUPPRESSION MENU NAV

Dans le cadre de notre projet, nous n'avons pas besoin d'avoir un menu de navigation (entre le logo et le menu utilisateur, nous allons donc le supprimer (desktop et mobile)

Dans le composant `resources/views/layouts/navigation.blade.php`

#### Nav Desktop :

- Supprimer la ligne 14 à 18

#### Nav responsive :

- Supprimer la ligne 69 à 73

#### Suppression du titre de la page « dashboard »

Dans la vue de la page `resources/views/dashboard.blade.php`

- Supprimer ligne 2 à 7

### FOOTER

#### Ajout d'un footer

Dans le layout 'app' `resources/views/layouts/app.blade.php`

Ligne 34 : création du footer avec le même style que le header :

```
<footer class="bg-white shadow">
  <div class="max-w-7xl mx-auto py-6 px-4 sm:px-6 lg:px-8">
    Test
  </div>
</footer>
```

**Lien pour la documentation :**

Nous allons maintenant ajouter un lien à gauche du footer pour la documentation avec un icône pour mettre en avant le fait que c'est un lien :

```
<footer class="bg-white shadow">
  <div class="flex justify-end max-w-7xl mx-auto py-6 px-4 sm:px-6 lg:px-8">
    <a href="/" class="inline-flex items-center hover:text-blue-600 transition-colors
duration-200 ease-in-out">
      Documentation
      <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-
width="1.5" stroke="currentColor" class="w-4 h-4 ml-1">
        <path stroke-linecap="round" stroke-linejoin="round" d="M13.5 6H5.25A2.25 2.25
0 0 0 3 8.25v10.5A2.25 2.25 0 0 0 5.25 21h10.5A2.25 2.25 0 0 0 18 18.75V10.5m-10.5 6L21 3m0
0h-5.25M21 3v5.25" />
      </svg>
    </a>
  </div>
</footer>
```

Div :

- **Flex**, et **justify-end** pour aligner à gauche
- Ajout de margin et padding pour les différents écran (sm et lg)

**Lien (a)**

- Inline flex pour ne pas prendre toute la largeur mais uniquement la largeur du texte
- Hover : animation pour mettre en avant le lien et changement de couleur du texte

**SVG**

- Ajout d'un icône et taille
- Marge à gauche pour le séparer un peu de « documentation »

## PAGE

Suppression de la classe max-w-7xl dans le header et le footer pour que ces 2 éléments prennent toute la largeur.

### Tests

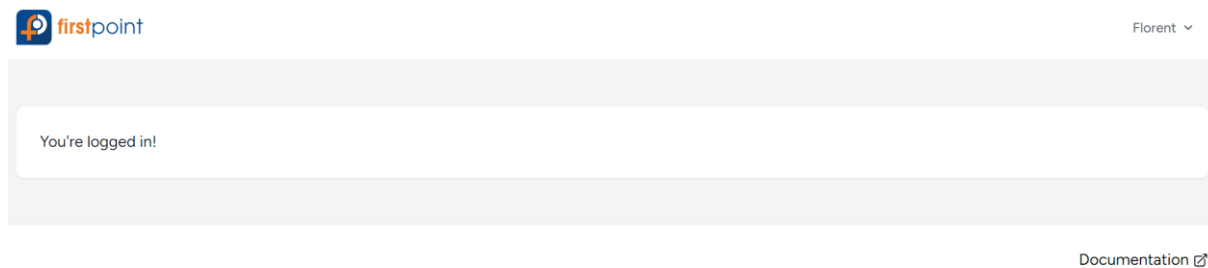


Figure 27 Test desktop layout

### Responsive

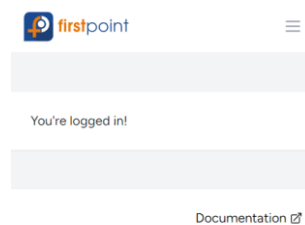


Figure 28 Test responsive layout



## INTÉGRATION SERVICES TIERS :

Afin d'intégrer des services tiers, nous allons avoir besoin d'un client http afin d'effectuer des requêtes, nous allons utiliser guzzle, effectuer la commande suivante pour l'installer :

```
composer require guzzlehttp/guzzle
```

## CONFIGURATION ET VARIABLE D'ENVIRONNEMENT

Ajout de variables d'environnement dans le .env (token api) pour Oh Dear et Flare :

```
OHDEAR_API_KEY=93cIxHhwJYwZ1S*****95rbuga23d5173
FLARE_API_KEY=N1bLvX4sa1dIzXm*****SHt8TxbstoljstilQi4
```

### fichier .env.exemple :

Il est important de modifier aussi le fichier .env.exemple pour refléter les ajouts et variables nécessaires au fonctionnement de l'application

```
OHDEAR_API_KEY=clé_api_ici
FLARE_API_KEY=clé_api_ici
```

Pour chaque service, nous allons créer un fichier de configuration

Dans un nouveau dossier (*config/provider*), créer un fichier de configuration pour stocker les configurations relatives à nos services

*config/provider/ohdear.php*

```
<?php
return [
    'api_key' => env('OHDEAR_API_KEY'),
];
```

*config/provider/flare.php*

```
<?php
return [
    'api_key' => env('FLARE_API_KEY'),
];
```

Création du fichier config (*config/sites.php*) et ajout d'une base de configuration pour exemple

```
<?php
return [
    'firstpoint.ch' => [
        'oh_dear' => [
            'enabled' => true,
            'site_id' => '12313432432', // <- L'ID du site pour les appels d'API Oh Dear
        ],
        'flare' => [
            'enabled' => 'false', // <-- Si jamais on désactive pour ce site
        ]
    ]
];
```

```
    ],
];
```

## SERVICE CONTAINER OH DEAR

Création du Service Container pour l'API Oh Dear

### Service provider

Dans le terminal, exécuter la commande suivante pour créer un service provider

```
php artisan make:provider OhDearServiceProvider
```

### Service client

Dans le nouveau service provider créé, enregistrer une service client personnalisé dans la méthode **register()** en utilisant le service container de Laravel :

```
public function register()
{
    $this->app->singleton(OhDearService::class, function ($app) {
        return new OhDearService(config('provider.ohdear.api_key'));
    });
}
```

Et importer la classe

```
use App\Services\OhDearService;
```

### Configuration du service client :

Créer une classe OhDearService dans un nouveau dossier *app/Services*

```
<?php

namespace App\Services;

class OhDearService
{
}
```

Implémentation de la classe OhDearService :

```
use GuzzleHttp\Client;

class OhDearService
{
    private Client $client;

    public function __construct()
    {
        $this->client = new Client([
            'base uri' => 'https://ohdear.app/api/',
            'headers' => [
                'Authorization' => 'Bearer ' . config('provider.ohdear.api_key'),
                'Accept' => 'application/json',
                'Content-Type' => 'application/json',
            ],
        ]);
    }

    public function getSiteData($siteName)
    {
        $config = config("sites");
        $siteId = $config[$siteName]['oh_dear']['site_id'];

        try {
            $response = $this->client->request('GET', "sites/{$siteId}", [
                'verify' => false,
            ]);
            return json_decode($response->getBody()->getContents(), true);
        } catch (\Exception $e) {
            Log::error("Erreur lors de la récupération des détails du site pour oh dear : {$e->getMessage()}");
            return null;
        }
    }
}
```

### Test Implémentation Oh Dear

Créer une commande Artisan :

```
php artisan make:command TestOhDear
```

Dans la commande créée, importation du services :

```
use App\Services\OhDearService;
```

Dans le méthode *handle()* :

```
public function handle()
{
    $ohDearService = new OhDearService();

    $siteId = 61225;

    $siteDetails = $ohDearService->getSiteData($siteId);

    $this->info("Détails du site : " . print_r($siteDetails, true));
}
```

Puis, exécuter la commande dans le terminal :

```
php artisan app:test-oh-dear
```

Résultat :

```
Détails du site : Array
(
    [id] => 61225
    [url] => https://firstpoint.ch
    [sort_url] => firstpoint.ch
    [label] => firstpoint.ch
    [team_id] => 14838
    [group_name] => FirstPoint
    [tags] => Array
        (
        )
    [notes] =>
    [latest_run_date] => 2024-02-26 13:13:57
    [summarized_check_result] => succeeded
    [uses_https] => 1
    [checks] => Array
        (
            [0] => array(
                'name' => 'SSL',
                'status' => 'ok',
                'details' => 'SSL is valid for 14 days more'
            )
        )
)
```

Tout est ok, nous avons bien les données retournées

## SERVICE CONTAINER FLARE

Création du Service Container pour l'API Flare

### Service provider

Dans le terminal, exécuter la commande suivante pour créer un service provider

```
php artisan make:provider FlareServiceProvider
```

### Service client

Dans le nouveau service provider créé, enregistrer un service client personnalisé dans la méthode **register()** en utilisant le service container de Laravel :

```
public function register()
{
    $this->app->singleton(FlareService::class, function ($app) {
        return new FlareService(config('provider.flare.api_key'));
    });
}
```

Importer la classe :

```
use App\Services\FlareService;
```

### Configuration du service client :

Créer une classe FlareService dans un nouveau dossier *app/Services*

Configuration de l'appel

```
public function __construct()
{
    $this->apiToken = config('provider.flare.api_key');
    $this->client = new Client([
        'base_uri' => 'https://flareapp.io/api/',
        'headers' => [
            'Accept' => 'application/json',
        ],
        'verify' => false,
    ]);
}
```

Pour l'implémentation de l'API, nous devons procéder en 2 étapes :

- Récupérer l'id d'un projet
- Récupérer les erreurs à l'aide de l'id

Récupération de l'ID :

```
public function getProjectIdByName($siteName)
{
    try {
        // Récupérer tous les projets
        $response = $this->client->get('projects?api_token=' . $this->apiToken);
        $projects = json_decode($response->getBody()->getContents(), true);

        // Filtrer pour trouver le projet par son nom
```

```

        foreach ($projects['data'] as $project) {
            if ($project['name'] === $siteName) {
                Log::info("Projet id pour {$siteName} : {$project['id']}");
                return $project['id'];
            }
        }

        Log::info("Aucun projet trouvé pour {$siteName}.");
        return null;
    } catch (\Exception $e) {
        Log::error("Erreur lors de la récupération du projet {$siteName} : " . $e-
>getMessage());
        return null;
    }
}

```

Récupérer l'erreur avec l'id

```

public function getSiteData($siteName)
{
    $projectId = $this->getProjectIdByName($siteName);

    if (!$projectId) {
        return null;
    }

    try {
        $response = $this->client->get("projects/{$projectId}?api_token=" . $this-
>apiToken."&status=open");

        return json_decode($response->getBody()->getContents(), true);
    } catch (\Exception $e) {
        Log::error("Erreur lors de la récupération des erreurs pour le projet {$siteName} : "
. $e->getMessage());
        return null;
    }
}

```

Importer les classes suivantes

```

use GuzzleHttp\Client;
use Illuminate\Support\Facades\Log;

```

Test implémentation Flare

Dans le terminal, exécuter :

```
php artisan make:command TestFlare
```

Puis implémenter l'appel

```

public function handle()
{
    $flareService = new FlareService();
    $siteName = 'blanc-labo.com';

    $errors = $flareService->getProjectErrors($siteName);

    $this->info("Erreurs pour {$siteName} : " . print_r($errors, true));
}

```

## IMPLÉMENTATION CRON

Maintenant que nous avons implémenter les fonctions pour la récupération des données via les 2 api, nous devons insérer ces résultats en base de données

## CRÉATION D'UN SERVICE DE GESTION DES RÉSULTATS

Afin de gérer les résultats et l'insertion en base de données, nous allons créer un services dédié.

Séparation des préoccupations : Ce service se concentre uniquement sur la gestion des résultats, rendant votre code plus organisé et plus facile à maintenir.

Réutilisabilité : Si vous devez enregistrer des résultats à partir d'autres commandes ou parties de votre application, vous pouvez réutiliser ce service sans dupliquer la logique d'enregistrement.

Testabilité : Il est plus facile de mocker un service dans des tests unitaires ou d'intégration, permettant de tester la logique d'enregistrement des résultats de manière isolée.

Dans le dossier app/services créer le service ResultService.php et implémenter une méthode **saveResults()** qui s'occupera d'enregistrer les résultats :

Importer les modèles de result, project et source :

```
use App\Models\Project;
use App\Models\Source;
use App\Models\Result;
```

```
public function saveResults($siteName, $sourceName, $data)
{
    $project = Project::firstOrCreate(['name' => $siteName]);
    $source = Source::firstOrCreate(['name' => $sourceName]);

    $result = new Result();
    $result->project_id = $project->id;
    $result->source_id = $source->id;
    $result->data = json_encode($data);
    $result->save();
}
```

Afin de pouvoir assigner des valeurs « en masse » nous devons rajouter les colonnes assignable :

Ajout de l'attribut « fillable » dans la classe dans les modèles Source et Project :

```
protected $fillable = ['name'];
```

## IMPLÉMENTATION DE LA COMMANDE CRON

Dans le terminal, exécuter la commande :

```
php artisan make:command CheckSitesCommand
```

Importer :

```
use App\Services\FlareService;
use App\Services\OhDearService;
use App\Services\ResultService;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
```

Et implémenter la méthode *handle()*

```
public function handle()
{
    $resultService = app(ResultService::class);

    $sites = config('sites');

    foreach ($sites as $siteName => $services) {
        foreach ($services as $serviceName => $serviceConfig) {
            if (!$serviceConfig['enabled']) {
                continue;
            }

            $studlyServiceName = Str::studly($serviceName);

            $serviceClass = "App\\Services\\" . $studlyServiceName . "Service";

            $service = app($serviceClass);

            $data = $service->getSiteData($siteName);

            if ($data) {
                $resultService->saveResults($siteName, $serviceName, $data);
            } else {
                Log::warning("{StudlyServiceName}: Aucune donnée récupérée pour {SiteName}.");
            }
        }
    }
}
```

## PLANIFICATEUR DE TÂCHES

Dans le fichier `app/Console/Kernel.php` :

```
protected function schedule(Schedule $schedule): void
{
    $schedule->command('app:check-sites-command')->everyMinute();
}
```

Exécuter la commande dans le terminal pour lancer la tâche :

```
php artisan schedule:run
```

## CONFIGURATION DU CRON SUR LE SERVEUR

Sur votre serveur, vous devez ajouter une entrée cron qui exécute la commande `schedule:run` de Laravel toutes les minutes. Cela permettra au planificateur de tâches de Laravel de vérifier et d'exécuter toutes les tâches programmées selon leur configuration.

Délai de requête

Afin d'éviter des erreurs type `too many requests`, nous allons ajouter un délai dans l'envoi des requêtes :

Dans la méthode `handle()` de `CheckSiteCommands.php` :

Ajouter un délai d'une seconde et avant l'appel API mettre en attente :

```
$delayInSeconds = 1;

sleep($delayInSeconds);
```



Dans le cadre de ce projet, nous n'avons pas besoin d'implémenter une autre stratégie (job, queue etc..) si le nombre de site devait augmenter drastiquement, l'implémentation d'un autre système serait nécessaire

## LARAVEL TELESCOPE

Ajout de telescope à notre projet :

```
composer require laravel/telescope
```

Publier les assets :

```
php artisan telescope:install
```

Migrer la base de données

```
php artisan migrate
```

Une fois configuré, nous pouvons relancer la commande d'appel d'api sans délai et voir que l'erreur est bien présente et enregistrée :



Figure 29 Telescope

Sécurisation de telescope :

Afin de sécuriser laravel telescope, nous allons ajouter un accès conditionnel :

Dans le fichier app/Providers/TelescopeServiceProvider.php

Dans la méthode **gate()**, rajouter l'email :

```
protected function gate(): void
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            "fscheiber@firstpoint.ch"
        ]);
    });
}
```

Ajout de la variable d'environnement dans le fichier .env et .env.exemple :

```
TELESCOPE_ENABLED=true
```

**LIVEWIRE :**

Ajouter livewire au projet :

```
composer require livewire/livewire
```

Publier les assets :

```
php artisan vendor:publish --tag=livewire:assets
```

Inclusion des Scripts

Pour que Livewire fonctionne, nous devons inclure ses scripts dans nos vues.

Ouvrir le fichier de layout principal de notre application *resources/views/layouts/app.blade.php* et ajouter les directives suivantes :

- Avant la fermeture de la balise <head> : @livewireStyles
- Avant la fermeture de la balise <body> : @livewireScripts

**SELECTEURS DE PROJETS**

Afin de sélectionner un projet pour l'afficher détaillé de ses données, nous allons créer un composant livewire (dropdown) pour afficher tout les projets.

Création du composant :

```
php artisan make:livewire ProjectsDropdown
```

Cette commande va créer deux fichiers :

- un pour la classe du composant dans *app/Livewire/ProjectsDropdown.php*
- un pour la vue dans *resources/views/livewire/projects-dropdown.blade.php*.

```
PS C:\Sites\TPI> php artisan make:livewire ProjectsDropdown
COMPONENT CREATED 🍌

CLASS: app/Livewire//ProjectsDropdown.php
VIEW: C:\Sites\TPI\resources\views\livewire\projects-dropdown.blade.php

--
 /  o  \   II  O           O   --
|_ \  / _|   II  II  \  /  / _ \  \  /  II  | ~ ~  / _ \
| ' ' |   II  II   \  \  \ ^ /   II  II   \  \

Congratulations, you've created your first Livewire component! 🎉🎉🎉
```

Figure 30 Livewire first component

Modifier la classe du composant, en incluant le modèle Project

```
namespace App\Livewire;

use Livewire\Component;
use App\Models\Project;

class ProjectsDropdown extends Component
{
    public $projects;

    public function mount()
```

```

    {
        $this->projects = Project::all();
    }

    public function render()
    {
        return view('livewire.projects-dropdown');
    }
}

```

### Création de la vue

```

<div>
    <select class="form-control">
        @foreach ($projects as $project)
            <option value="{{ $project->id }}">{{ $project->name
        }}</option>
        @endforeach
    </select>
</div>

```

Inclure le composant, dans notre projet il se situe dans le header entre le logo et le menu utilisateur

Dans le fichier de navigation resources/views/layouts/navigation.blade.php

Ajouter le composant à la fin du div du logo

```
@livewire('projects-dropdown')
```

Et admirer le résultat :

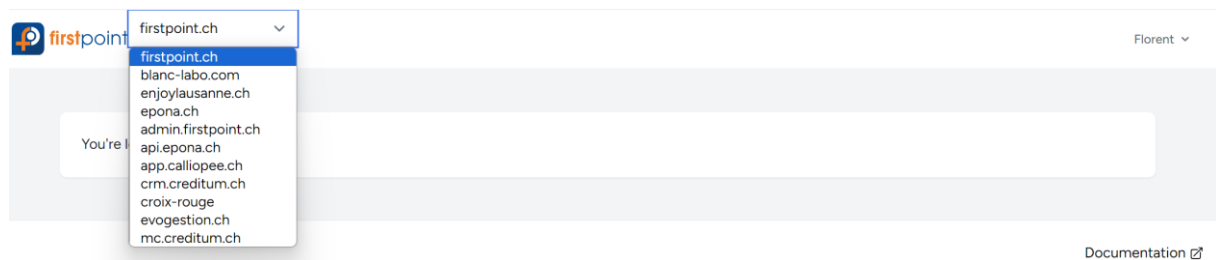


Figure 31 Dropdown projects

## ROUTE ET PAGE DÉTAILS D'UN PROJET

Afin de rediriger l'utilisateur vers une page détails pour un projet, nous devons créer une page et configurer une nouvelle route

Dans la classe du composant, définir une méthode redirectToProject()

```

public $selectedProject;
public function redirectToProject()
{
    if (empty($this->selectedProject)) {
        return redirect()->route('dashboard');
    } else {

```

```

        return redirect()->route('projects.show', ['project' => $this->selectedProject]);
    }
}

```

Création d'un controller project :

```
php artisan make:controller ProjectController --resource
```

L'option `--resource` créera directement les méthodes pour les opérations index, create, store, show, edit, update, et destroy déjà définies.

Configuration de la route :

```
Route::resource('projects', ProjectController::class);
```

Il ne faut pas oublier d'importer le controller, de plus, nous mettons cette route en sécurité dans un middleware

Création de la page show pour les projets : `resources/views/projects/show.blade.php`

```

<x-app-layout>
    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ $project->name }}
        </h2>
    </x-slot>

    <div class="py-12">
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white overflow-hidden shadow-sm sm:rounded-lg">
                <div class="p-6 text-gray-900">
                    Hello World
                </div>
            </div>
        </div>
    </div>
</x-app-layout>

```



## INDEX

Laravel

TailwindCSS

Alpine.js

Livewire

Oh Dear

Flare

Composer

NPM

MVC

Vue

Modèle

Controlleur

API

GitHub

PHP

SQL

Cron

Dropdown

Seeders

CRUD

TALL

Versionning

Framework

Commit

Branche

Breeze

Back-end

Front-end

Eloquent

Wamp

Apache

Mysql

Template

Dépendance(s)

IDE

.gitignore

README.md

Node.js

Npm

LTS

Build

Jetons

Artisan

String

Timestamp

ORM

## TABLE DES ILLUSTRATIONS

Figure 1 Schéma BDD modèle conceptuel .....	8
Figure 2 Schéma BDD modèle relationnel .....	8
Figure 3 Maquette authentification .....	9
Figure 4 Maquette vue d'ensemble .....	10
Figure 5 Maquette détails.....	10
Figure 6 WampServer .....	24
Figure 7 MySQL config .....	25
Figure 8 MySQL databases.....	26
Figure 9 PHP version.....	26
Figure 10 Composer version .....	27
Figure 11 Laravel install .....	28
Figure 12 Laravel version.....	28
Figure 13 Laravel page principale .....	29
Figure 14 BDD connexion PhpStorm.....	30
Figure 15 Node.js installation.....	31
Figure 16 node et npm versions .....	32
Figure 17 Premier build Tailwind CSS.....	33
Figure 18 Résultat migration.....	38
Figure 19 BDD after migration.....	38



## SOURCES & BIBLIOGRAPHIES

Tutoriel base de données : <https://www.youtube.com/watch?v=RTRMXxBnOAO>

WampServer download : <https://www.wampserver.com/en/download-wampserver-64bits/>

Composer : <https://getcomposer.org/download/>

Node.js : <https://nodejs.org/en>

PhpStorm : <https://www.jetbrains.com/fr-fr/phpstorm/>

TailwindCSS: <https://tailwindcss.com/docs/guides/laravel>

Laravel : <https://laravel.com/docs/10.x>

Laravel migrations : <https://laravel.com/docs/10.x/migrations>

MySQL timestamp : <https://dev.mysql.com/doc/refman/8.0/en/datetime.html>

Erreur migrations : <https://stackoverflow.com/questions/42244541/laravel-migration-error-syntax-error-or-access-violation-1071-specified-key-wa>

Laravel models : <https://laravel.com/docs/10.x/eloquent#introduction>

Laravel Seeders : <https://laravel.com/docs/10.x/seeding#writing-seeders>

Laravel factories : <https://laravel.com/docs/10.x/eloquent-factories>

Icones : <https://heroicons.com/>

Laravel httpclient : <https://laravel.com/docs/10.x/http-client>

Laravel tasks : <https://laravel.com/docs/10.x/scheduling>

Laravel Telescope : <https://laravel.com/docs/10.x/telescope>