Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER PROFESSIONNEL (DP)

Nom de naissance - *BEZIAT*
Nom d'usage - *BEZIAT*
Prénom - *Hervé*
Adresse - *541 c chemin du gavelier, 83119 Brue Auriac*

Titre professionnel visé

Concepteur Développeur d'Applications

MODALITÉ D'ACCÈS :

☐ Parcours de formation
☐ Validation des Acquis de l'Expérience (VAE)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel. **Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen**.

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.

2. du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.

3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation

4. de l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

DOSSIER PROFESSIONNEL (DP)

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

http://travail-emploi.gouv.fr/titres-professionnels

Sommaire

Exemples de pratique professionnelle

EXEMPLES DE PRATIQUE PROFESSIONNELLE

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 Développer une application sécurisée

Exemple n°1 ▸ *Mise en oeuvre de l'application SafeBase*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

SafeBase est une application web développée en PHP avec le framework Symfony, dont l'objectif est de gérer les sauvegardes automatiques de bases de données MySQL/MariaDB.
L'outil permet à des utilisateurs de :

- Sélectionner une ou plusieurs bases de données à sauvegarder
- Générer des dumps automatiquement ou manuellement
- Afficher un historique des sauvegardes réalisées
- Restaurer une version antérieure de la base
- Journaliser toutes les opération

Ce projet vise à garantir la sécurité des données, faciliter leur restauration rapide en cas de besoin, et offrir une interface utilisateur simple et efficace.
SafeBase a été développé dans un contexte d'apprentissage, mais avec des exigences proches de conditions professionnelles : gestion de projet, code versionné, documentation, etc.

Installer et configurer son environnement de travail en fonction du projet.
Dans un premier temps, j'ai mis en place un environnement local sur mon poste à l'aide de WampServer pour disposer rapidement d'un serveur Apache, de MySQL et de PHP. J'ai ensuite installé Symfony 7.1 via Composer, puis initialisé le projet avec `symfony new` .
J'ai configuré le projet pour être lancé via le serveur de développement Symfony (`symfony serve`).

Dès l'initialisation du projet, j'ai mis en place un système de gestion de versions à l'aide de Git, pour assurer le suivi des évolutions du code. Cela permettait également d'assurer la collaboration dans un contexte multo-développeurs.
J'ai ensuite créé un dépôt distant privé sur GitHub, que j'ai relié au dépôt local. Cela a permis de :

- synchroniser le projet entre plusieurs postes de travail
- conserver une copie distante à jour
- utiliser GitHub comme outil de collaboration avec gestion des branches, historique, pull requests.

J'ai organisé le travail avec une stratégie simple :

- une branche `main` pour le code stable
- des branches secondaires pour chaque nouvelle fonctionnalité
- des fusions (`merge`) après validation de chaque module

Une fois le projet avancé, j'ai souhaité rendre l'environnement de développement portable, reproductible et proche d'un futur environnement de production.
Pour cela, j'ai mis en place une architecture Dockerisée du projet.

J'ai commencé par créer les fichiers de configuration nécessaires pour automatiser la mise en place de l'environnement. Tout d'abord j'ai créé le fichier docker-compose.yml qui définit les 3 services principaux :

1. web (PHP + Apache) : Container principal qui exécute l'application Symfony
    - Il utilise un Dockerfile personnalisé, basé sur une image PHP avec Apache intégrée.
    - J'ai installé dans ce container les extensions PHP nécessaires à Symfony (pdo , mysqli , etc.)
    - un fichier vhost.conf utilisé pour configurer les hôtes virtuels Apache (rewrite des URLs Symfony)
    - Le volume .:var/www/html permet de synchroniser le code local avec le conteneur.
2. db (MariaDB / MySQL) : Fournit la base de données utilisée par l'application
    - Basé sur l'image officielle mariadb:10.6
    - Expose le port 3306
    - Initialisé avec un mot de passe root et un nom de base personnalisé (plateforme_safebase)
    - Permet de stocker les données de l'application de façon persistante
3. phpmyadmin
    - Utilise l'image officielle phpmyadmin/phpmyadmin
    - connecté au service db
    - accessible via le port 8081
    - Pratique pour vérifier les enregistrements et la structure des tables sans requêtes SQL manuelles

Les autres fichiers de configuration sont :

- Dockerfile : installe PHP, Apache, Composer, et les extensions PHP manquantes
- .env : fichier de configuration pour avoir accès à la base de donnée

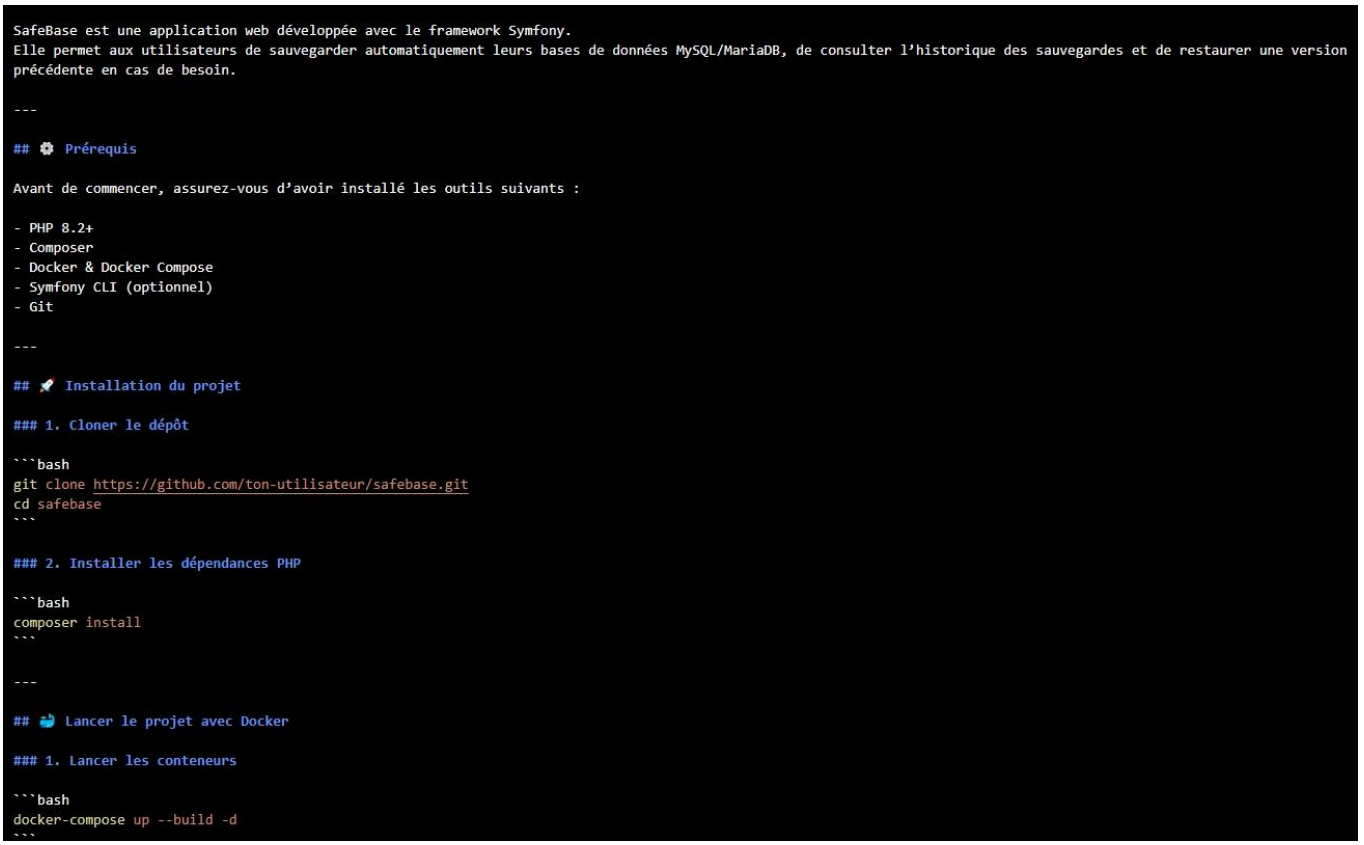
Les objectifs et les avantages de la dockerisation sont :

- Lancer toute l'application avec une seule commande docker-compose up une fois que l'image a été créée docker-compose –build
- Éviter les différences de versions entre les environnements de chaque développeur
- Travailler en temps réel (volume partagé) sans avoir à reconstruire l'image à chaque modification
- Préparer le terrain pour le futur déploiement en production sur un serveur Docker ou un cloud.

Une fois l'environnement de développement installé et configuré, j'ai rédigé un fichier README.md en français dans le répertoire racine du projet.
Ce document a pour objectif de guider tout nouveau développeur souhaitant installer, configurer et utiliser le projet SafeBase, que ce soit en local ou dans un environnement Dockerisé.
Le README contient notamment :

- Les étapes pour cloner le projet depuis GitHub
- Les commandes Composer nécessaires à l'installation des dépendances
- Les instructions pour lancer l'environnement Docker à l'aide de docker-compose
- Les informations pour accéder à l'interface phpMyAdmin

- Et une liste de commandes utiles pour le développement



```
SafeBase est une application web développée avec le framework Symfony.
Elle permet aux utilisateurs de sauvegarder automatiquement leurs bases de données MySQL/MariaDB, de consulter l'historique des sauvegardes et de restaurer une version
précédente en cas de besoin.

---

## ⚙️ Prérequis

Avant de commencer, assurez-vous d'avoir installé les outils suivants :

- PHP 8.2+
- Composer
- Docker & Docker Compose
- Symfony CLI (optionnel)
- Git

---

## 🚀 Installation du projet

### 1. Cloner le dépôt

```bash
git clone https://github.com/ton-utilisateur/safebase.git
cd safebase
```

2. Installer les dépendances PHP

```bash
composer install
```

🐳 Lancer le projet avec Docker

1. Lancer les conteneurs

```bash
docker-compose up --build -d
```
```

Développer des interfaces utilisateur

Une fois cette base technique en place, j'ai pu commencer le développement de l'application en respectant les maquettes établies en amont.

Le premier aspect sur lequel je me suis concentré est l'interface utilisateur, afin de structurer visuellement l'outil et permettre à l'utilisateur d'interagir facilement avec les différentes fonctionnalités proposées.

L'interface utilisateur de SafeBase a été conçue à partir d'un wireframe puis d'une maquette réalisée avec Figma.

La mise en œuvre respecte la structure et l'apparence définies dans cette maquette :

- Titre et descriptif clair clair sur la page d'accueil
- Liste des fonctionnalités principales sous forme de points
- Présence d'un menu de navigation latéral
- Utilisation d'une mise en page simple et épurée, adaptée à une cible technique

Le projet SafeBase utilise Symfony pour sa structure MVC, avec le moteur de template Twig pour générer les vues HTML.

Ce choix offre plusieurs avantages :

- Une séparation claire entre la logique métier (controller) et l'affichage (templates)
- Une réutilisation facilité des blocs de code grâce à l'héritage de templates ({% extends

'base.html.twig' %}).

- Une syntaxe simple et lisible, idéale pour intégrer dynamiquement des variables PHP dans les vues.

Chaque page de l'application est rendue depuis un controller Symfony, qui transmet les données nécessaires à une vue Twig, laquelle se charge de l'affichage HTML final.
Par exemple, la page d'accueil (/) est gérée par un contrôleur qui renvoie vers le fichier home/index.html.twig, lequel contient le contenu présenté à l'utilisateur, comme un titre <h2>, des paragraphes explicatifs et une liste des fonctionnalités principales.

L'application est destinée à être utilisée depuis un poste de travail (ordinateur), dans un contexte professionnel (développeur, administrateur système,...)
L'interface a donc été pensée pour :

- Une navigation intuitive via un menu latéral
- Une lisibilité optimisée sur écran large
- Une structure claire, permettant d'accéder rapidement aux actions principales (sauvegarde, restauration, historique)

Des tests fonctionnel ont été mis en place pour valider le bon affichage de l'interface utilisateur, notamment la page d'accueil.
Un test vérifie par exemple :

- Que la page / s'affiche correctement (code HTTP 200)
- Que le titre principal <h2> contient bien le texte requis
- Que la liste des fonctionnalités est présente avec 4 éléments ( )

Extrait du texte utilisé (HomeControllerTest.php) :

```php
public function testHomePageIsSuccessful(): void
{
    $client = static::createClient();
    $crawler = $client->request('GET', '/');

    // Vérifie que la page renvoie un code 200
    $this->assertResponseIsSuccessful();

    // Vérifie que le titre principal est bien affiché (h2 dans ce cas)
    $this->assertSelectorTextContains('h2', 'Bienvenue sur SafeBase');

    // Vérifie qu'un paragraphe contenant une description est bien présent
    $this->assertSelectorTextContains('p', 'SafeBase est un outil de gestion de sauvegardes');

    // Vérifie qu'il y a bien une liste de fonctionnalités
    $this->assertSelectorExists('ul');

    // Vérifie qu'il y a bien 4 éléments dans la liste
    $this->assertCount(4, $crawler->filter('ul li'));
}
```

Ce test garantit que l'utilisateur final voit bien les informations essentielles dès l'arrivée sur l'application. Il complète les tests métier en assurant que le rendu HTML est conforme à ce qui est attendu.

```
root@630487be2e6c:/var/www/html# bin/phpunit tests/Controller/HomeControllerTest.php --testdox
PHPUnit 9.6.20 by Sebastian Bergmann and contributors.

Testing App\Tests\Controller\HomeControllerTest
Home Controller (App\Tests\Controller\HomeController)
 ✓ Home page is successful

Time: 00:06.078, Memory: 34.50 MB

OK (1 test, 7 assertions)
```

Développer des composants métier

Après avoir mis en place l'interface utilisateur et validé son bon fonctionnement à travers des tests fonctionnels, j'ai poursuivi le développement de l'application en me concentrant sur la logique métier (backend).

Cette partie représente le cœur de fonctionnement de SafeBase car elle regroupe les entités, les règles , les services, et les traitements côté serveur.

Pour cela, j'ai respecté les principes de la programmation orientée objet (POO), tout en veillant à produire un code sécurisé, structuré et conforme aux bonnes pratiques du développement logiciel.

Le projet utilise Symfony, un framework PHP structuré autour de l'approche objet.

Les entités métier sont encapsulées, avec des propriétés privées, des getters/setters, et des contraintes de validation Symfony.

Exemple de propriété privée annotée, un getter et un setter de l'entité DatabaseConnection.php :

```php
#[ORM\Column(length: 255, nullable: true)]
private ?string $password = null;

#[ORM\Column(length: 255)]
#[Assert\NotBlank(message: "Le nom de la base de données ne peut pas être vide")]
private ?string $databaseName = null;
```

```php
public function removeBackupSchedule(BackupSchedule $backupSchedule): static
{
    if ($this->backupSchedules->removeElement($backupSchedule)) {
        if ($backupSchedule->getDatabaseConnection() === $this) {
            $backupSchedule->setDatabaseConnection(null);
        }
```

Les responsabilités sont également bien séparées :

- Les contrôleurs reçoivent les requêtes HTTP
- Les services traitent la logique métier comme la connexion à une base de données ou la création

d'un dump SQL.
- Le code métier est centralisé dans des classes de service injectées automatiquement par Symfony

Dans le fichier DatabaseConnectionController.php, la méthode testConnection() illustre bien la séparation des responsabilités.
Le contrôleur gère uniquement la requête entrante et la délègue aux services métiers injectés, responsables de la logique de test de connexion et de sauvegarde.

```php
#[Route('/api/test-connection', name: 'test_connection', methods: ['POST'])]
public function testConnection(Request $request): JsonResponse
{
    $data = json_decode($request->getContent(), true);

    $connection = new DatabaseConnection();
    $connection->setName($data['name'] ?? '');
    $connection->setHost($data['host'] ?? '');
    $connection->setPort($data['port'] ?? 3306);
    $connection->setUsername($data['username'] ?? '');
    $connection->setPassword($data['password'] ?? '');
    $connection->setDatabaseName($data['databaseName'] ?? '');

    try {
        $testResult = $this->connectionService->testConnection($connection);

        if (!$testResult['status']) {
            return $this->json($testResult, 400);
        }

        $saveResult = $this->managerService->saveConnection($connection);
        return $this->json($saveResult, $saveResult['success'] ? 200 : 400);
    } catch (\Throwable $e) {
        return $this->json([
            'status' => false,
            'message' => 'Erreur interne : ' . $e->getMessage(),
        ], 500);
    }
}
```

La méthode createBackup() centralise la logique métier liée à la génération d'un dump SQL.
Elle est entièrement encapsulée dans un service dédié, ce qui garantit une bonne séparation des responsabilités et une meilleure testabilité.
L'utilisation du composant Process de Symfony permet de sécuriser l'exécution système tout en gardant le contrôle sur les erreurs éventuelles.

```php
public function createBackup(DatabaseConnection $connection): Backup
{
    $backup = new Backup();
    $backup->setDatabaseConnection($connection);
    $backup->setStatus('pending');

    $filename = sprintf(
        '%s_%s.sql',
        $connection->getDatabaseName(),
        $backup->getCreatedAt()->format('Y-m-d_H-i-s')
    );
    $filePath = $this->backupDir . '/' . $filename;

    $command = sprintf(
        'mysqldump -h %s -P %s -u %s -p%s %s > %s',
        $connection->getHost(),
        $connection->getPort(),
        $connection->getUsername(),
        $connection->getPassword(),
        $connection->getDatabaseName(),
        $filePath
    );

    $process = Process::fromShellCommandline($command);
    $process->setTimeout(3600); // 1 hour timeout

    try {
        $process->mustRun();

        $backup->setFilename($filename);
        $backup->setFilePath($filePath);
        $backup->setSize(filesize($filePath));
        $backup->setStatus('completed');

        $this->entityManager->persist($backup);
        $this->entityManager->flush();

        return $backup;
    } catch (ProcessFailedException $exception) {
        $backup->setStatus('failed');
        $this->entityManager->persist($backup);
        $this->entityManager->flush();

        throw $exception;
    }
```

Cela illustre que la logique métier est isolable, testable, et respectueuse des bonnes pratiques objet.

Plusieurs mesures ont été prises pour garantir la sécurité côté serveur :

- Les formulaires Symfony filtrent automatiquement les données (protection contre XSS)
- Les identifiants transmis dans les formulaires ne sont jamais enregistrés
- L'accès aux fonctions sensibles est prévu pour être restreint à des utilisateurs connectés (future authentification via security.yaml)

Afin de garantir la lisibilité et la maintenabilité du code, j'ai appliqué les standards PSR-12 dans l'ensemble du projet SafeBase.
La norme PSR-12 est une recommandation officielle qui définit des règles précises pour structurer et écrire du code PHP de manière claire, lisible et homogène.
Elle couvre notamment le nommage (CamelCase , snake_case), l'organisation des classes, des namespaces, des blocs de code, l'usage des indentations et la mise en forme générale des fichiers PHP.

Dans ce même objectif de qualité, les méthodes critiques de mes services métiers sont systématiquement documentées à l'aide de blocs PHPDoc.
Ces commentaires indiquent le rôle de la méthode, les paramètres attendus, la valeur de retour et les exceptions levées si nécessaire.
Cette pratique facilite grandement la lecture et la reprise du projet par un autre développeur.

```php
/**
 * Crée un dump SQL de la base de données passée en paramètre,
 * et enregistre les informations dans l'entité Backup.
 *
 * @param DatabaseConnection $connection Objet contenant les informations de connexion
 * @return Backup Objet représentant le fichier de sauvegarde créé
 * @throws ProcessFailedException En cas d'échec de l'exécution de la commande mysqldump
 */

public function createBackup(DatabaseConnection $connection): Backup
```

Des tests unitaire ont été mis en place pour valider le bon fonctionnement des entités métiers, notamment les contraintes de validation.
Un exemple concret est le test de validation de l'entité DatabaseConnection.
Ce test vérifie que si le champ name est vide, la connexion ne peut pas être considérée comme valide.

```php
public function testEmptyNameShouldFailValidation(): void
{
    $connection = new DatabaseConnection();
    $connection->setName(''); // ← champ vide
    $connection->setHost('host.docker.internal'); // ← pour accéder à la BDD locale
    $connection->setPort(3306);
    $connection->setUsername('root');
    $connection->setPassword(null);
    $connection->setDatabaseName('mydb');

    $errors = $this->validator->validate($connection);

    $this->assertCount(1, $errors, 'Un champ vide pour "name" devrait générer une erreur de validation.');
    $this->assertSame('Le nom ne peut pas être vide', $errors[0]->getMessage());
```

Ce test permet de valider automatiquement les règles de l'annotation @Assert/NotBlank définie dans l'entité DatabaseConnection .

```php
#[ORM\Column(length: 255)]
#[Assert\NotBlank(message: "Le nom ne peut pas être vide")]
private ?string $name = null;
```

Il garantit que l'utilisateur ne peut pas enregistrer une connexion sans nom, même via une requête manuelle ou une interface externe.

Même si SafeBase ne dispose pas encore d'un système complet d'authentification, les points d'entrée critiques ont été sécurisés avec plusieurs bonnes pratiques :

- Validation serveur via Symfony
- Bloc try/catch pour intercepter les erreurs
- Réponse JSON génériques pour éviter les fuites d'informations techniques

Un exemple concret est visible dans le fichier DatabaseConnectionController.php

```php
try {
    $testResult = $this->connectionService->testConnection($connection);

    if (!$testResult['status']) {
        return $this->json($testResult, 400);
    }

    $saveResult = $this->managerService->saveConnection($connection);
    return $this->json($saveResult, $saveResult['success'] ? 200 : 400);
} catch (\Throwable $e) {
    return $this->json([
        'status' => false,
        'message' => 'Erreur interne : ' . $e->getMessage(),
    ], 500);
```

Ce bloc empêche les utilisateurs malveillants de voir des messages d'erreur système détaillés.
Cela limite les risques d'exposition d'informations sensibles comme la configuration du serveur, les chemins système ou les exceptions PHP internes.

Ces tests unitaires et ces contrôles côté serveur garantissent que les composants métier définies tout en protégeant l'application contres des comportements non souhaités et dangereux.
Cela prépare également le projet à l'intégration future d'un système complet d'authentification et d'autorisation.

Contribuer à la gestion d'un projet informatique
Le développement de SafeBase a été découpé en plusieurs étapes clés : analyse du besoin, conception, mise en place de l'environnement, développement de l'interface, logique métier, puis tests.

Pour gérer ce projet de manière structurée, j'ai mis en place une organisation inspirée de la méthode Kanban via un tableau Trello.
Ce tableau était divisé en colonnes :
À faire → En cours → À tester → Terminé

Cela m'a permis de visualiser l'état d'avancement en temps réel, de prioriser les tâches, et d'adapter le planning en cas de blocage technique (par exemple, pendant la mise en place de Docker).

Par la suite les procédures de qualité ont été mises en œuvre en appliquant les standards PSR-12 pour un code clair et uniforme.
La qualité est renforcée par :

- Des tests unitaires sur les entités métiers
- Des tests fonctionnels sur les interfaces utilisateur
- Une organisation de code en services pour séparer clairement la logique

L'architecture du projet repose sur une base Symfony, avec un environnement Dockerisé reproduisant les conditions d'un déploiement en production.
Les conteneurs mis en place sont :

- PHP 8.2 + Apache
- MariaDB
- phpMyAdmin

L'édition du code est faite via Visual Studio Code, avec des extensions adaptées à Symfony et au suivi de projet.

```yaml
services:
  app:
    build:
      context: .
    container_name: safebase_app
    ports:
      - "8080:80"  # pas de conflit avec Apache (WAMP reste sur 80)
    volumes:
      - .:/var/www/html
    depends_on:
      - db

  db:
    image: mysql:8.0
    container_name: safebase_db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: plateforme_safebase
      MYSQL_USER: symfony
      MYSQL_PASSWORD: symfony
    ports:
      - "3308:3306"  # on choisit un autre port libre (ni 3306, ni 3307)
    volumes:
      - db_data:/var/lib/mysql

  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: safebase_pma
    restart: always
    ports:
      - "8081:80"
    environment:
      PMA_HOST: db
      PMA_USER: symfony
      PMA_PASSWORD: symfony

volumes:
  db_data:
```

Grâce à une planification progressive, des outils adaptés et une documentation technique complète, j'ai assuré une gestion de projet fluide et conforme aux exigences professionnelles.
L'organisation Kanban, le suivi des tâches, l'environnement Dockerisé et les standards qualité ont permis de garantir un développement maîtrisé du projet SafeBase.

2. Précisez les moyens utilisés :

Outils utilisés :

- WampServer pour le serveur local Apache / PHP / MySQL (utilisé au départ)
- Symfoni CLI pour démarrer un serveur de dev simple
- Composer pour gérer les dépendances PHP
- Visual Studio Code pour l'écriture du code
- GitHub pour le versioning
- Docker

Documentations technique :

- Documentation officielle Symfony
- Utilisation de forums/documentation GitHub & Stack Overflow

Pour la gestion du projet, j'ai utilisé :

- Trello pour organiser les tâches
- Git pour le versionning
- GitHub comme plateforme de dépôt
- README.md pour centraliser la documentation technique

3. Avec qui avez-vous travaillé ?

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Ecole La Plateforme.*

Chantier, atelier, service ▸ *Formation.*

Période d'exercice ▸ Du : *01/09/2024* au : *30/06/2025*

5. Informations complémentaires *(facultatif)*

DOSSIER PROFESSIONNEL (DP)

Activité-type 2 — Concevoir et développer une application sécurisée organisée en couches

Exemple n° 1 — *Conception logicielle et organisation modulaire de l'application SafeBase*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Analyser les besoins et maquetter une application

Le projet SafeBase répond à un besoin métier clair : permettre à un utilisateur (développeur, administrateur système ou expert-comptable) de gérer facilement les sauvegardes de bases de données MySQL/MariaDB via une interface web simple et sécurisée.

Les besoins fonctionnels ont été identifiés à partir d'un mini cahier des charges, que j'ai moi-même rédigé après avoir défini les objectifs du projet.
Ils sont résumés comme suit :

- Ajouter et gérer des connexions à des bases de données
- Lancer une sauvegarde manuelle (dump sql)
- Restaurer une sauvegarde précédente
- Visualiser l'historique des sauvegardes effectuées
- Assurer la sécurité des accès et des traitements (validation des champs, traitement côté serveur)

À partir de ces besoins, j'ai réalisé plusieurs maquettes de l'interface utilisateur à l'aide de Figma, en français.
Elles ont permis de visualiser l'ergonomie de l'application avant la phase de développement.

Les maquettes comprennent notamment :

- Une page d'accueil avec présentation de l'outil
- Une page de connexion à la base de données
- Une interface de sauvegarde avec bouton d'action
- Une page de restauration
- Une page d'historique des dumps réalisés

L'enchaînement logique des écrans a été formalisé sous forme d'un schéma de navigation dans Figma, représentant le parcours utilisateur (UserFlow) typique.
Ce schéma montre par exemple que :

1. L'utilisateur arrive sur la page d'accueil
2. Il clique sur "Connexion DB"
3. Une fois connecté, il accède à l'interface de sauvegarde
4. Il peut ensuite aller consulter l'historique ou effectuer une restauration

Le projet repose sur une démarche de conception structurée :

- Utilisation de la méthode Merise pour concevoir la base de donnée (MCD, MPD, MLD)
- Structuration de l'application en couches (contrôleur, entité, service, vue)
- Mise en place des maquettes, suivies du schéma de navigation
- Documentation des besoins dans un mini cahier des charges fonctionnel.

Grâce à une analyse précise des besoins, une conception d'interface structurée et une démarche itérative, le projet SafeBase a pu être développé en répondant aux exigences exprimées dans le cahier des charges.
Les maquettes ont joué un rôle essentiel pour valider l'ergonomie, et le dossier de conception constitue une base solide pour les étapes de développement.

Définir l'architecture logicielle d'une application
Une fois les besoins définis et les maquettes validées, j'ai conçu une architecture logicielle structurée pour répondre aux exigences du projet.
Cette architecture respecte les principes d'une application sécurisée, modulaire et évolutive, en s'appuyant sur les bonnes pratiques du framework Symfony.

Le projet SafeBase repose sur une architecture logicielle monolithique, structurée selon le modèle MVC (Modèle-Vue-Contrôleur) fourni par le framework Symfony.
Ce choix s'explique par la nature du projet : une application autonome, légère, ne nécessitant pas (à ce stade) une architecture distribuée ou orientée microservices.

Couche	Rôle	Exemple dans SafeBase
Contrôleur	Gère les requêtes entrantes, fait appel aux services	DatabaseConnectionController.php
Service (métier)	Contient la logique métier (sauvegarde, restauration, validation)	DatabaseBackupService.php
Entité (modèle)	Représente les objets métiers persistés en base	DatabaseConnection.php
Vue (Twig)	Génère les pages HTML de l'interface	home.html.twig base.html.twig

Cette séparation permet de :

- Centraliser la logique métier
- Réutiliser les services sans dépendre du front

- Sécuriser les données (aucun traitement critique en front)
- Faciliter les tests

Même si le projet est individuel et modeste, certaines bonnes pratiques ont été intégrées :
TDD (Test-Driven Development) partiel :

- Des tests unitaires ont été écrits pour valider les entités
- Des tests fonctionnels vérifient le bon déroulement de certaines pages et requêtes

DDD (Domain Driven Design) simplifié :

- Les services métiers sont centrés sur les actions de domaine : créer une sauvegarde, tester une connexion, etc.
- Les entités représentent clairement les objets métier : connexion, sauvegarde

Dès la conception de l'application, plusieurs bonnes pratiques d'optimisation ont été appliquées afin d'améliorer la performance, la lisibilité du code et la fiabilité globale du projet :

- Injection de dépendances (DI) : tous les services sont injectés via les constructeurs. Cette approche évite les instanciations manuelles dans les contrôleurs, favorise le découplage, et facilite les tests unitaires grâce à la possibilité de simuler les dépendances (mocking).
- Services découplés : la logique métier est isolée dans des classes de service (comme DatabaseBackupService ou DatabaseConnectionManagerService). Cela rend le code plus maintenable, favorise la réutilisation, et respecte le principe de responsabilité unique (SRP).
- Exécution de commandes système optimisée : les sauvegardes SQL générées via mysqldump, exécuté à l'aide du composant Process de Symfony. Cela permet de gérer proprement les erreurs, de fixer un temps d'exécution maximal et d'éviter le blocage de l'application en cas d'échec.
- Validation centralisée des données : les entrées utilisateurs sont systématiquement contrôlées avant traitement (type, présence, format), ce qui renforce la sécurité tout en évitant des erreurs critiques côté base de données.
- Interface légère : aucun framework Javascript n'est utilisé. L'interface est développée uniquement avec Twig, HTML et un peu de CSS, ce qui réduit la bande passante, améliore la rapidité d'affichage et la compatibilité des navigateurs.

Même si le projet est modeste, plusieurs efforts ont été réalisés en faveur de l'éco-conception :

- L'interface utilisateur, très légère, permet de limiter les ressources nécessaires à l'affichage
- Aucun chargement de bibliothèques ou framework inutile
- Réduction du code redondant grâce à l'organisation en services réutilisables
- Utilisation de Docker pour maîtriser les ressources consommées par l'environnement de développement

Ces choix contribuent à réduire la consommation énergétique de l'application et à en améliorer la performance, même sur des machines peu puissantes.

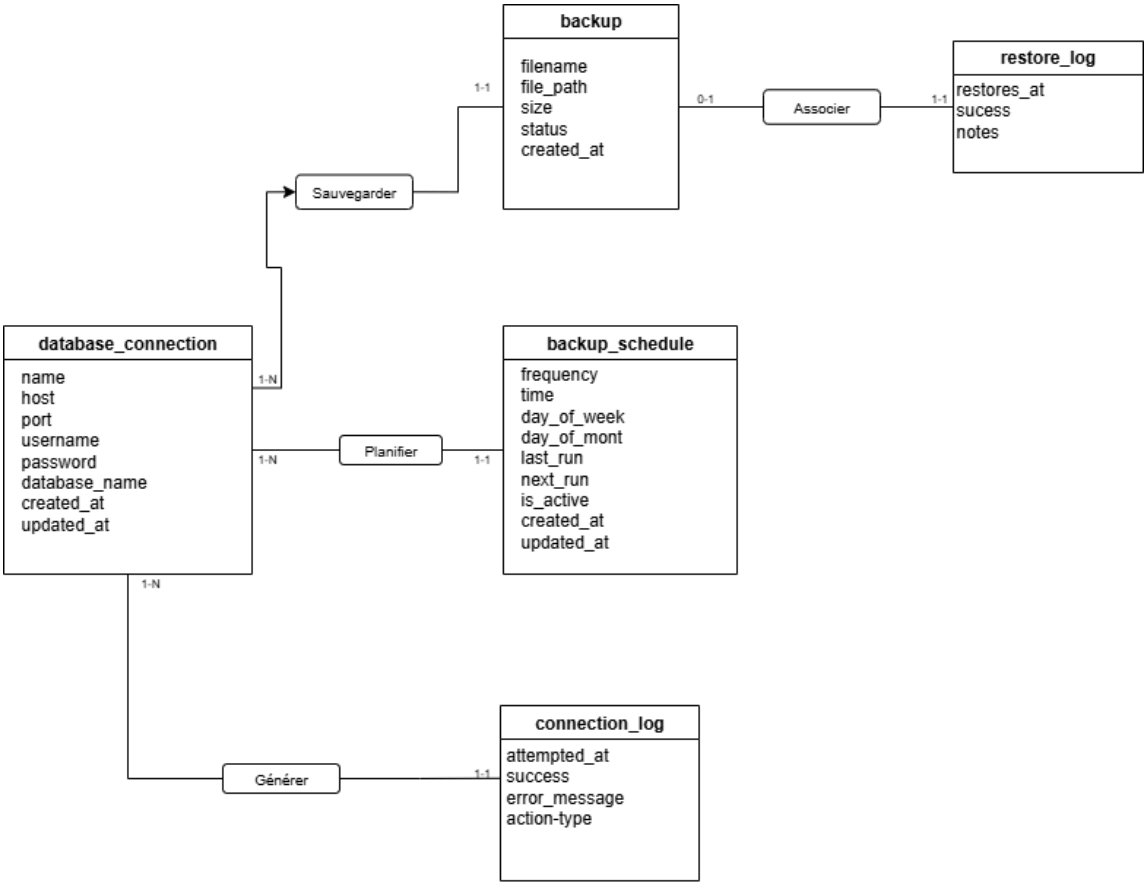
L'architecture et le code SafeBase ont été pensés pour garantir un bon équilibre entre performance, clarté et sécurité.
Les choix techniques adoptés s'inscrivent dans les recommandations des bonnes pratiques Symfony, tout en intégrant des réflexes d'éco-conception dès les premières lignes de code.
Ce socle technique solide assure une évolutivité facile vers des fonctionnalités futures (authentification, automatisation, etc.) sans remettre en cause l'architecture existante.

Concevoir et mettre en place une base de données relationnelle
Une fois l'architecture logicielle définie et les couches de l'application structurées, il a été essentiel de concevoir une base de données relationnelle adaptée aux besoins fonctionnels exprimés dans le cahier des charges.

Avant toute implémentation technique, la conception de la base de données a été réalisée à l'aide de la méthode Merise, en suivant les étapes classiques :

- MCD (Modèle Conceptuel de Données) pour identifier les entités principales du projet : DatabaseConnection, Backup
- MLD (Modèle Logique de Données) pour déterminer les types de relations et les clés étrangères
- MPD (Modèle Physique de Données) pour adapter le schéma au SGBD MariaDB/MySql

Visuel du MCD :



Le Schéma physique de la base de données répond aux besoins exprimés dans le cahier des charges :

- Une table database_connection contenant les informations nécessaires à la connexion (host, port, user, password, database_name)
- Une table backup liée à la précédente avec une clé étrangère, contenant le chemin du dump, sa date de création, sa taille et son statut.

Les noms de tables et de colonnes respectent les conventions de nommage SQL, avec des noms clairs, en anglais et en snake_case.

La base de données a été pensée pour garantir la confidentialité et la sécurité des données, via plusieurs mécanismes :

- Les mots de passe de connexion aux bases ne sont jamais affichés, même en cas d'erreur
- Les entrées sont validées et nettoyées avant d'être insérées en base
- Les identifiants critiques ne sont jamais exposés dans les URLs
- Possibilité d'évolution vers un hachage des données sensibles (via bcrypt ou sodium)

Concernant l'intégrité :

- Les relations sont assurées par des clés étrangères (backup.database_connection_id → database_connection.id)
- Chaque table possède une clé primaire auto-incrémentée

Dans le projet SafeBase, les tests automatisés (PHPUnit) ont été réalisés à l'aide d'une base MySQL dédiée, distincte de la base principale de l'application.
Plutôt que de créer un conteneur séparé, nous avons dupliqué la structure de la base existante (plateforme_safebase) pour créer une version de test (ex. : plateforme_safebase_test) dans le même conteneur Docker MySQL.

Cette base de test est utilisée automatiquement par Symfony lors de l'exécution des tests unitaires ou fonctionnels. Elle permet de :

- Exécuter des tests en toute sécurité, sans impacter les données de développement
- Tester la logique métier avec des jeux d'essai prévisibles (fixtures ou insertions manuelles)
- Vérifier la validité des contrôleurs, des entités et des règles de validation

En cas de besoin, cette base peut être recréée automatiquement à partir des entités Symfony, grâce aux commandes de migration Doctrine (doctrine:migrations:migrate).
Cela permet de générer la structure de la base à partir du modèle objet, sans avoir à manipuler manuellement de fichiers .sql.
Une fois la structure en place, la base peut être peuplée automatiquement à l'aide de fixtures Symfony, afin de disposer de jeux de données réalistes pour les tests.

L'organisation de la base de données est soigneusement documentée afin de faciliter sa prise en main par un autre développeur :

- Les entités PHP sont commentées à l'aide de PHPDoc, ce qui permet de comprendre le rôle de chaque propriété et leur lien avec la base de données (types, relations, etc.)
- Le modèle logique est directement visible dans le code à travers les annotations Doctrine, ce qui permet de visualiser rapidement les relations (@OneToMany, @ManyToOne, etc.) et les contraintes.
- Le fichier README.md contient toutes les instructions nécessaires pour créer et initialiser la base, que ce soit en environnement local ou Docker :
    1. Commandes de migration Doctrine
    2. Accès à PhpMyAdmin
    3. Variables d'environnement à configurer pour la connexion

Enfin, l'ensemble de la documentation technique est rédigée en français clair, afin d'être facilement compréhensible par un développeur.

En résumé, la base de données de SafeBase a été conçue selon les règles du relationnel, documentée proprement, et intégrée dans un environnement de développement automatisé et sécurisé, garantissant ainsi sa maintenabilité et sa fiabilité.

Développer des composants d'accès aux données SQL et NoSQL
L'application SafeBase repose sur une architecture en couches claire et sécurisée, déjà détaillée dans les sections précédentes.
Elle s'appuie sur le framework Symfony pour structurer le projet selon le modèle MVC, avec une séparation nette entre les contrôleurs, les services métiers et les entités.
Les traitements implémentés répondent aux besoins exprimés dans le dossier de conception.
Chaque fonctionnalité (connexion, sauvegarde, restauration) est isolée dans des services dédiés et protégée par des mécanismes de validation et de gestion des erreurs.
L'intégrité et la confidentialité des données sont assurées grâce à :

- L'utilisation de Doctrine ORM, qui protège des injections SQL
- La validation des entrées côté serveur via des contraintes sur les entités
- La gestion des exceptions dans les services critiques (try/catch)
- L'absence de stockage des mots de passe en clair côté client ou dans les vues

Des tests unitaires et fonctionnels ont été réalisés avec PHPUnit, en s'appuyant sur une base de test MySQL isolée, comme présenté dans le bloc précédent.
Ces tests permettent de garantir la stabilité du code tout en anticipant les cas d'usage inattendus.

En résumé, l'application respecte les principes d'une architecture sécurisée et modulaire, garantissant un traitement fiable des données, une bonne maintenabilité du code, et une protection des utilisateurs face aux erreurs ou aux attaques potentielles.

2. Précisez les moyens utilisés :

Figma
DOcker
Git et GitHub
Draw.io

3. Avec qui avez-vous travaillé ?

Ce fût dans le cadre d'un projet personnel au sein de l'école dans le but d'acquérir les compétences pour le passage du CDA.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Ecole La Plateforme*

Chantier, atelier, service ▸ *Formation*

Période d'exercice ▸ Du : *01/09/2024* au : *30/06/2025*

5. Informations complémentaires *(facultatif)*

Activité-type 3 Préparer le déploiement d'une application sécurisée

Exemple n° 1 – *Automatisation des tests, déploiement et supervision de l'application Safebase*

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Préparer et exécuter les plans de tests d'une application

Pour assurer la fiabilité de l'application SafeBase, un plan de tests a été rédigé sous forme d'un cahier des recettes, recensant l'ensemble des cas d'usage à tester. Ce document a servi de base pour structurer les vérifications fonctionnelles et garantir la conformité aux attentes définies en amont du projet.

N°	Fonctionnalité testée	Scénario de test	Données de test	Résultat attendu
1	Test de connexion à une base de données	L'utilisateur saisit les bons identifiants et teste la connexion	Host : localhost, User : root, Pass : 1234	Message de succès affiché, connexion réussie
2	Test de connexion invalide	L'utilisateur entre un mot de passe erroné	Host : localhost, User : root, Pass : badpass	Message d'erreur, aucune connexion ne doit être établie
3	Lancement manuel d'une sauvegarde	L'utilisateur clique sur "Créer une sauvegarde" pour une base valide	Base : platforme_safebase	Un fichier .sql est généré et sauvegardé dans le bon dossier
4	Accès à la page d'accueil	Un utilisateur accède à l'application via l'URL d'accueil	URL : /	Page d'accueil s'affiche avec un message de bienvenue
5	Validation du champ "Nom de la connexion"	L'utilisateur laisse le champ vide	Nom : ""	Message d'erreur "Le nom est requis"
6	Affichage des logs	L'utilisateur accède à la page "Logs des sauvegardes"	Aucun	Liste des sauvegardes avec dates, noms, statuts

Un environnement de tests dédié, isolé de l'environnement de production, a été mis en place à l'aide de conteneurs Docker. Cela a permis de recréer des conditions réelles d'utilisation tout en préservant l'intégrité des données sensibles.

Chaque module de l'application (connexion à une base de données, lancement manuel d'une sauvegarde, affichage des logs, etc.) a été soumis à des scénarios de tests manuels. Par exemple, la fonctionnalité de test de connexion à une base a été vérifiée avec des configurations valides, invalides, ou incomplètes.

Ces vérifications ont été complétées par des tests unitaires automatisés, principalement sur les entités Symfony, via le framework PHPUnit. Les validations métier (présence d'un nom, validité de l'hôte, etc.) ont été testées systématiquement.

Les résultats obtenus ont été conformes aux résultats attendus dans le cahier des tests. Les écarts constatés ont été documentés et corrigés, puis re-testés jusqu'à validation.

Enfin, le plan de tests a été conçu pour anticiper les évolutions futures (ajout de la planification automatique des sauvegardes, gestion multi-utilisateurs…) et pour prendre en compte les enjeux de sécurité, notamment autour des connexions à distance et de la confidentialité des identifiants de bases de données.

Préparer et documenter le déploiement d'une application
Le déploiement de l'application SafeBase a été anticipé dès les premières phases du projet, afin de faciliter sa mise en production et son évolutivité.

J'ai choisi d'utiliser Docker pour encapsuler les différents services nécessaires au fonctionnement de l'application (PHP, Symfony, MySQL, phpMyAdmin). Ce choix permet une reproductibilité des environnements, que ce soit pour le développement, les tests ou la production, et simplifie l'installation sur une nouvelle machine.

Une documentation technique de déploiement a été rédigée avec le projet. Elle décrit :

- les prérequis système,
- les étapes de mise en place de l'environnement Docker,
- les variables d'environnement à configurer,
- les commandes à exécuter pour lancer les services.

Des scripts de déploiement (fichiers Dockerfile, docker-compose.yml, et scripts de setup éventuels) ont été produits et commentés. L'objectif est de permettre à un tiers de déployer rapidement et de manière autonome l'application.

```dockerfile
FROM php:8.2-apache

# Install required PHP extensions
RUN apt-get update && apt-get install -y \
    git unzip zip libicu-dev libonig-dev libxml2-dev libzip-dev \
    && docker-php-ext-install intl pdo pdo_mysql zip

# Enable Apache mod_rewrite
RUN a2enmod rewrite

# Set working directory
WORKDIR /var/www/html

# Copy all project files
COPY . .
```

En parallèle, un environnement de test dédié a été mis en place à l'aide d'une configuration Docker distincte, afin d'exécuter les tests d'intégration et de vérifier le bon fonctionnement global du système.

Afin d'automatiser les tâches de test et de suivi du projet SafeBase, j'ai mis en place un pipeline CI/CD à l'aide de GitHub Actions.

Ce pipeline se déclenche automatiquement à chaque commit ou push sur la branche main. Il est défini dans un fichier YAML (.github/workflows/ci.yml) versionné avec le projet.

Le workflow exécute les étapes suivantes :

- Téléchargement du code : l'action checkout récupère le code source depuis le dépôt GitHub.
- Configuration de l'environnement PHP : la machine GitHub est configurée avec PHP 8.2, Composer, et les extensions nécessaires (pdo_mysql, mbstring) pour que le projet Symfony fonctionne.
- Installation des dépendances : composer install est exécuté pour installer les librairies du projet.
- Mise en place d'une base de données MySQL temporaire : un service MySQL est lancé dans le workflow, accessible localement, avec une base safebase prête à l'emploi.
- Configuration de l'environnement de test : un fichier .env.test.local est généré automatiquement avec les bonnes informations de connexion à la base MySQL.
- Exécution des tests unitaires : le framework PHPUnit est utilisé pour tester les entités Symfony et les règles métier définies dans le projet.
- Envoi de notifications par email :
- Si les tests réussissent : un email est envoyé pour signaler que la mise à jour est stable.
- Si les tests échouent : un autre email est envoyé avec le lien direct vers les logs de GitHub Actions pour analyser l'erreur.

Cette approche permet de :

- automatiser les tests sans intervention humaine,
- notifier les membres du projet des changements importants,
- et garantir que seules des versions stables du projet soient déployées par la suite.
- Cette mise en place s'inscrit dans une démarche DevOps, en favorisant l'automatisation, la fiabilité et la traçabilité du développement.

```yaml
name: SafeBase - CI Tests & Email

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  run-tests:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_ROOT_PASSWORD: root
          MYSQL_DATABASE: safebase
        ports:
          - 3306:3306
        options: >-
          --health-cmd="mysqladmin ping --silent"
          --health-interval=10s
          --health-timeout=5s
          --health-retries=5

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Set up PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.2'
        extensions: pdo, pdo_mysql, mbstring
        tools: composer

    - name: Install dependencies
      run: composer install --no-progress --prefer-dist
```

Enfin, une veille technique est régulièrement effectuée sur les outils de déploiement (Docker, GitHub Actions) et sur les enjeux de sécurité liés à l'environnement (mise à jour des images, gestion des identifiants, etc.).

Contribuer à la mise en production dans une démarche DevOps
L'approche DevOps adoptée pour le projet SafeBase a permis de renforcer la qualité du code et la fiabilité des déploiements.

Dans un premier temps, des outils de qualité de code ont été mis en place. Un linter PHP (via phpcs et les règles PSR-12) est utilisé localement pour garantir une base de code propre, lisible et cohérente. Cela permet de repérer rapidement les erreurs de style ou de syntaxe avant même les phases de test.

Des tests automatisés ont également été intégrés au projet grâce à PHPUnit, pour valider le bon fonctionnement des entités, des règles métiers et des interactions avec la base de données. Ces tests sont exécutés automatiquement via le pipeline CI/CD.

Ce pipeline CI/CD, mis en place via GitHub Actions, exécute plusieurs étapes clés à chaque commit ou push :

- Lancement d'un environnement de test complet (PHP + MySQL),
- Installation des dépendances et configuration du projet,
- Exécution des tests automatisés,
- Envoi de notifications par email selon le résultat.

Chaque exécution du pipeline génère un rapport détaillé (logs, statut, erreurs), qui est analysé directement depuis l'interface GitHub Actions pour valider ou corriger rapidement les livrables.

Par ailleurs, un suivi des logs applicatifs est effectué dans l'interface Symfony et via l'historique des sauvegardes (fonctionnalité native de SafeBase). Cela permet de tracer les actions critiques et de renforcer le contrôle sur les données sensibles.

Enfin, l'ensemble des outils utilisés (PHPUnit, GitHub Actions, Docker, Composer…) disposent d'une documentation officielle en anglais que j'ai su exploiter, en autonomie, pour configurer les environnements, écrire les scripts, et corriger les éventuelles erreurs détectées.

2. Précisez les moyens utilisés :

Docker
GitHub Actions
PHPUnit

3. Avec qui avez-vous travaillé ?

Ce fût dans le cadre d'un projet personnel au sein de l'école dans le but d'acquérir les compétences pour le passage du CDA.

4. Contexte

Nom de l'entreprise, organisme ou association ▸ *Ecole Laplateforme.*

Chantier, atelier, service ▸ *Formation.*

Période d'exercice ▸ Du : *01/09/2024* au : *30/06/2025i*

5. Informations complémentaires *(facultatif)*

Titres, diplômes, CQP, attestations de formation

(facultatif)

Intitulé	Autorité ou organisme	Date
Cliquez ici.	Cliquez ici pour taper du texte.	Cliquez ici pour sélectionner une date.

Déclaration sur l'honneur

Je soussigné(e) [prénom et nom] BEZIAT HERVÉ ,

déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis l'auteur(e) des réalisations jointes.

Fait à Brue-Auriac le 30/07/2025

pour faire valoir ce que de droit.

Signature :

BEZIAT HERVÉ

Documents illustrant la pratique professionnelle

(facultatif)

Intitulé
Cliquez ici pour taper du texte.

ANNEXES

(Si le RC le prévoit)