# Documentation PariBasket

Projet du 27 au 31 janvier 2025

LENOIR Julie

SUIRE Enzo

LEONARD Chloé

# Sommaire :

# 1. Introduction

## 1.1 Contexte du projet

Le projet consiste à concevoir et développer une application web/mobile de paris sportifs dédiée à la NBA, l'une des ligues de basketball les plus suivies au monde. L'objectif principal est de créer une application fonctionnelle et ergonomique permettant aux utilisateurs de consulter les statistiques des équipes NBA, de simuler des matchs, et de placer des paris fictifs sur les résultats.

En plus de l'aspect technique, le projet encourage la collaboration en groupe, la mise en pratique des bonnes pratiques de développement logiciel, ainsi que l'application des principes DevOps pour le déploiement. Les apprenants doivent également produire une documentation technique complète, incluant les détails de la conception, des tests et du déploiement de l'application.

## 1.2 Description du besoin

Les paris sportifs représentent un secteur en forte croissance, et la NBA attire une large audience grâce à son dynamisme, ses joueurs emblématiques et ses statistiques détaillées. Une application dédiée aux paris sportifs fictifs sur la NBA répond à plusieurs besoins :

- Engagement des utilisateurs : Permettre aux fans de basketball de s'immerger davantage dans les performances des équipes et des joueurs en les incitant à analyser les statistiques et à faire des prévisions.
- Simulation et divertissement : Offrir une expérience ludique et éducative à travers des simulations de matchs basées sur des données réelles.
- Gestion de portefeuille virtuel : Apprendre aux utilisateurs à gérer leur solde virtuel en analysant les risques et les opportunités liés aux paris.

L'application, tout en restant fictive, offre un environnement réaliste et captivant, conçu pour répondre à des attentes spécifiques :

1. Une interface utilisateur fluide et intuitive pour consulter les statistiques et placer des paris.
2. Une approche sécurisée et modulaire permettant une gestion efficace des comptes et des paris.

## 2. Conception

### 2.1 Architecture générale

Pour développer l'application, nous allons utiliser Symfony pour le backend. Nous avons choisi d'aller chercher des données sur une API appelée SportRadar qui contient la liste des différents matchs ainsi que les détails de chacun. Pour la base de données nous avons choisi MySQL.

Nous avons choisi ces outils car ce sont ceux qui nous ont paru les plus adaptés à l'application demandée et ceux où nous étions le plus à l'aise afin de visualiser plus facilement le contenu des tâches qui nous attendaient.

Back-end :

PHP (Framework Symfony)

Front-end :

HTML / CSS /Bootstrap

Base de données :

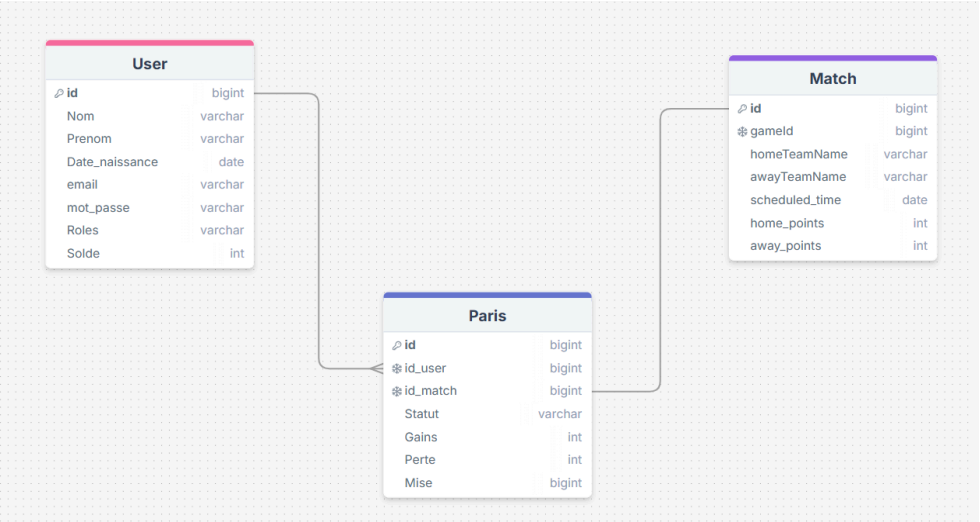
MySQL

APi :

Sportradar

### 2.2 Conception base de données

Pour commencer, nous avons choisi d'utiliser le site drawsql.app afin d'obtenir un visuel des données que nous devons stocker dans la base de données. Cela nous a permis de faire un travail de réflexion sur les données à récupérer directement depuis l'API ou celles à stocker en base de données afin de les réutiliser.



Nous avons décidé de partir sur trois tables pour la conception de la base de données

### 1.User

**Première table : User,** qui concerne la partie utilisateurs. Pour la création de cette table, nous nous sommes appuyés sur le bundle SecurityBundle de Symfony, qui permet de mettre en place un système d'inscription et de connexion pour les utilisateurs. Nous avons également ajouté à cette table plusieurs champs, dont le Solde.

**Attributs :**

id
email
roles (rôles de l'utilisateur)
password (mot de passe)
nom, prenom, Date_naissance, solde (informations personnelles et solde)

**Relations :**

Un utilisateur peut avoir plusieurs Paris (relation OneToMany avec l'entité Paris).

### 2. Paris

Deuxième table : Paris, qui concerne la partie des paris. Elle comprend plusieurs champs (Statut, Gains, Mise, Perte, etc.). Cette table a pour objectif de stocker les paris et d'assurer leur suivi dans la partie administration.

**Attributs :**

id
Statut (statut du pari)
Gains, Perte, Mise (informations Budget)
equipe (équipe sur laquelle le pari est placé)
soldeCloture (solde après clôture du pari)

**Relations :**

Un pari est associé à un User (relation ManyToOne avec l'entité User).
Un pari est associé à un Matchs (relation ManyToOne avec l'entité Matchs).

### 3. Matchs

Troisième table : Matchs, qui concerne la partie des matchs. Plusieurs champs ont été mis en place (homeTeamName, awayTeamName, scheduled_time) afin de récupérer les données de l'API Sportradar et de les stocker ensuite en base de données.

**Attributs :**

id
home Team Name, awayTeamName (noms des équipes)
scheduled_time (heure prévue du match)
home_points, away_points (points des équipes)
Game_id (identifiant du match)

**Relations :**

Un match peut avoir plusieurs Paris (relation OneToMany avec l'entité Paris).

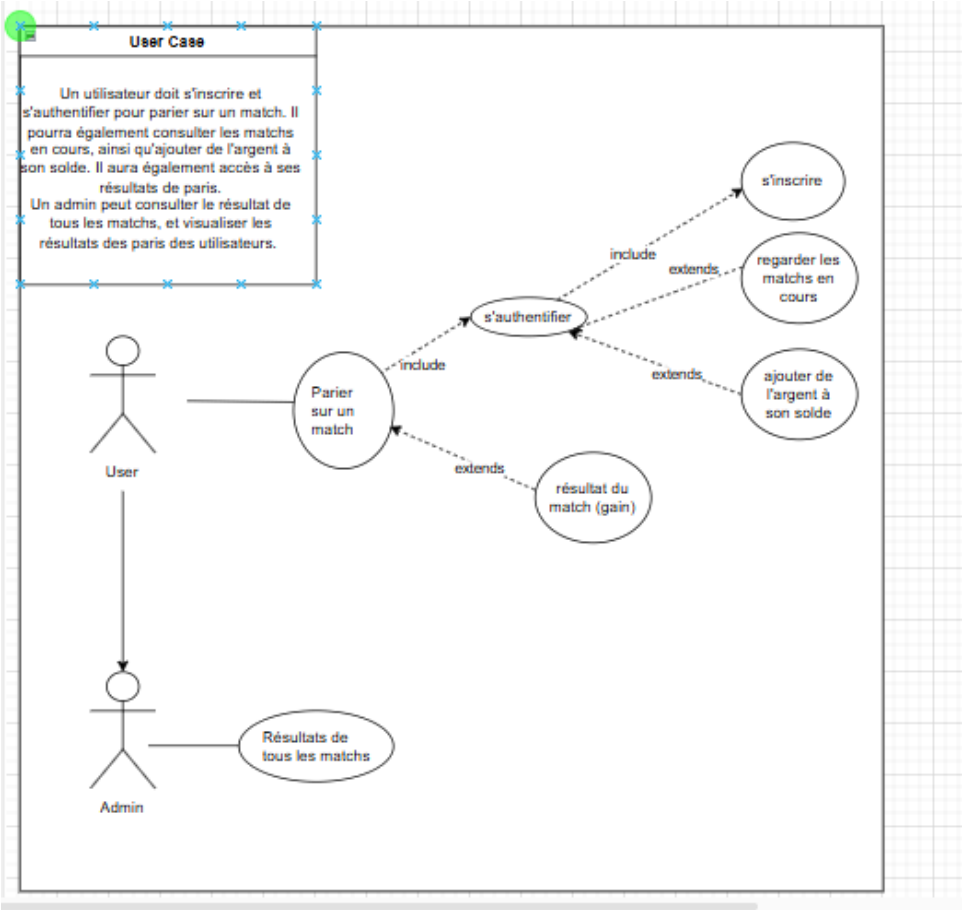
*Relations entre les entités :*

**User / Paris :** Un utilisateur peut effectuer plusieurs paris, mais chaque pari est associé à un seul utilisateur.
**Paris / Matchs :** Un pari est associé à un seul match, mais un match peut avoir plusieurs paris.

### 2.3 Diagramme UML

*Usercase*

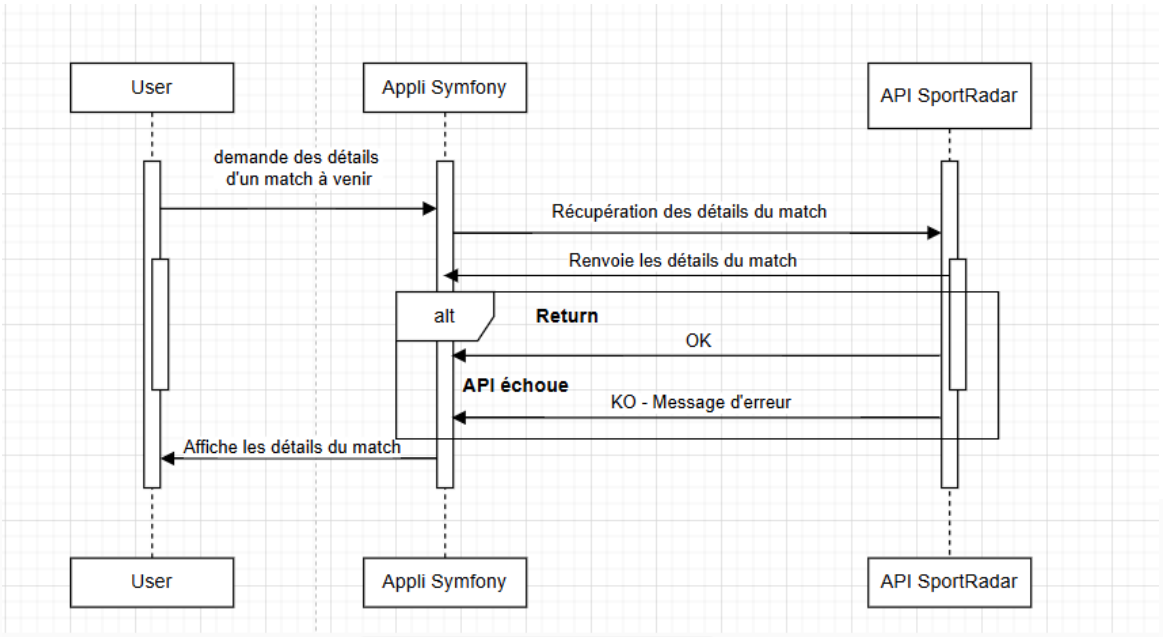
Nous avons ensuite fait un diagramme de cas d'utilisation (usercase) afin de faciliter la gestion des droits et des accès des différents utilisateurs :



Pour finir, nous avons fait un diagramme de séquence afin de visualiser le chemin parcouru par les données.

### *Diagramme de Séquence - CAS 1 Détails match à venir*

l'application est développée en Symfony, les données sont stockées dans une base de données relationnelles MySql et certaines informations sont récupérées via une API (SportRadar).
Lorsqu'un utilisateur souhaite avoir des détails d'un match à venir alors nous allons chercher les informations via l'API. Message d'erreur si le retour API échoue, Affichage des résultats si OK.
Pas de stockage en bdd.

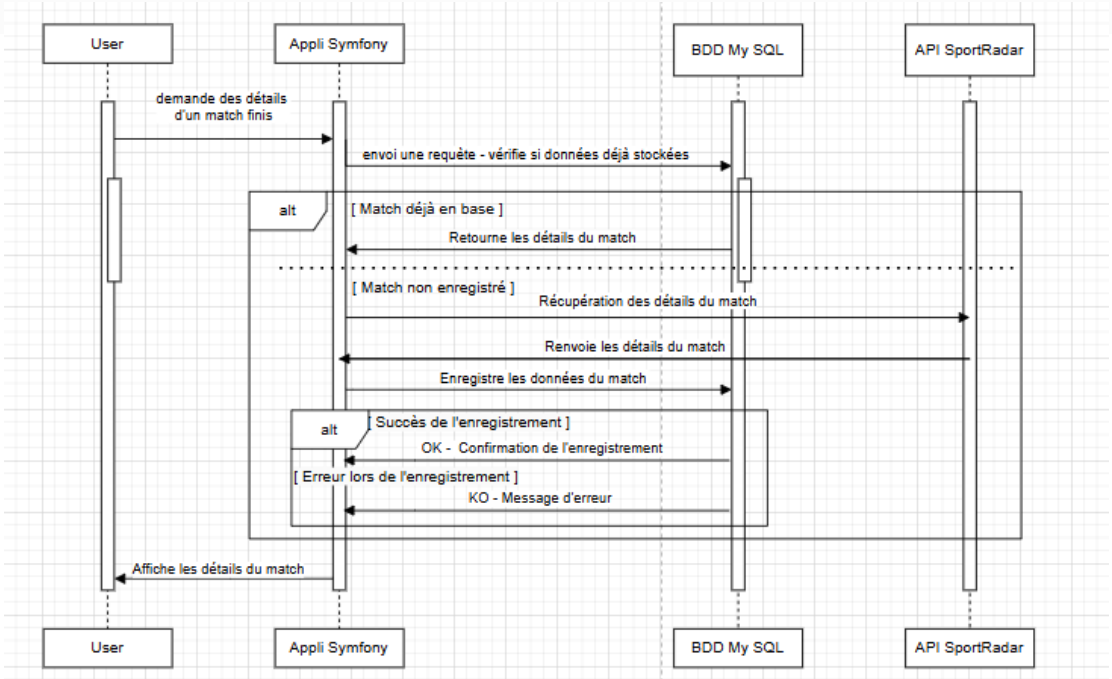



### *Diagramme de Séquence - CAS 2 Pari sur une match*

l'application est développée en Symfony, les données sont stockées dans une base de données relationnelles MySql et certaines informations sont récupérées via une API (SportRadar).
Lorsqu'un utilisateur souhaite parier sur un match alors .

- si déjà en bdd alors la bdd retourner les détails et ils sont affichés au USER
- si pas en bdd, on fait une requête à l'API qui renvoie les données puis on stocke les données en bdd. On affiche ensuite les détails au user.

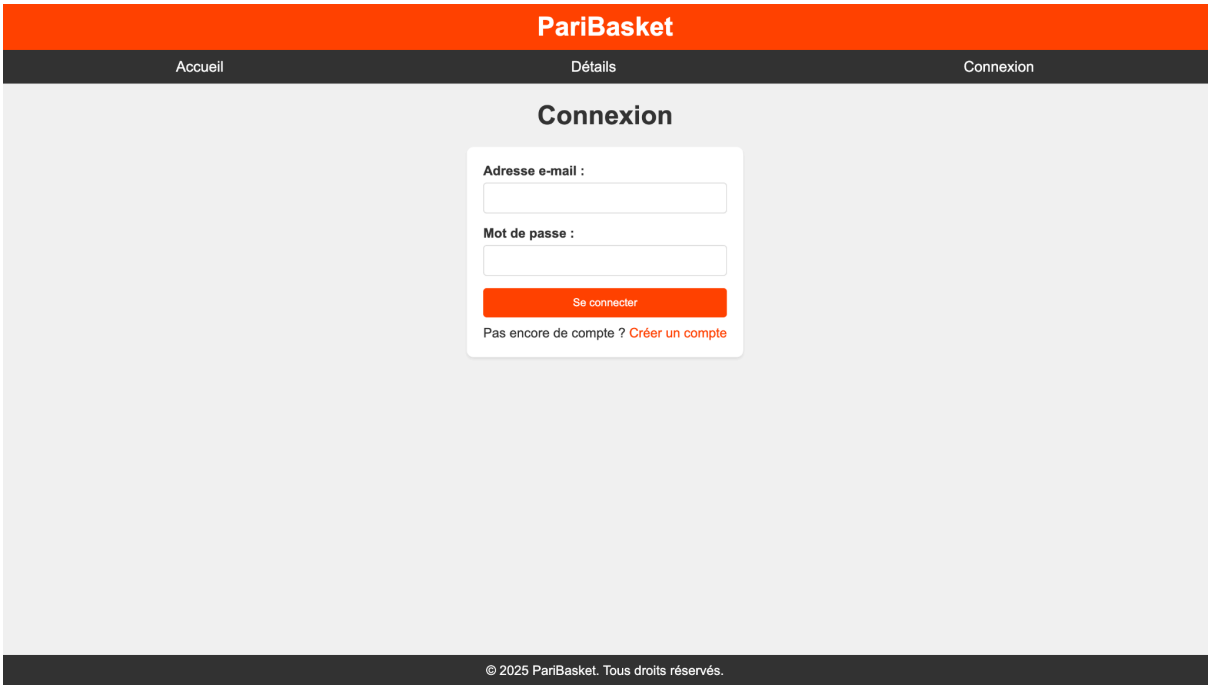
Un message OK ou KO est prévu pour vérifier si les données sont bien enregistrées en bdd



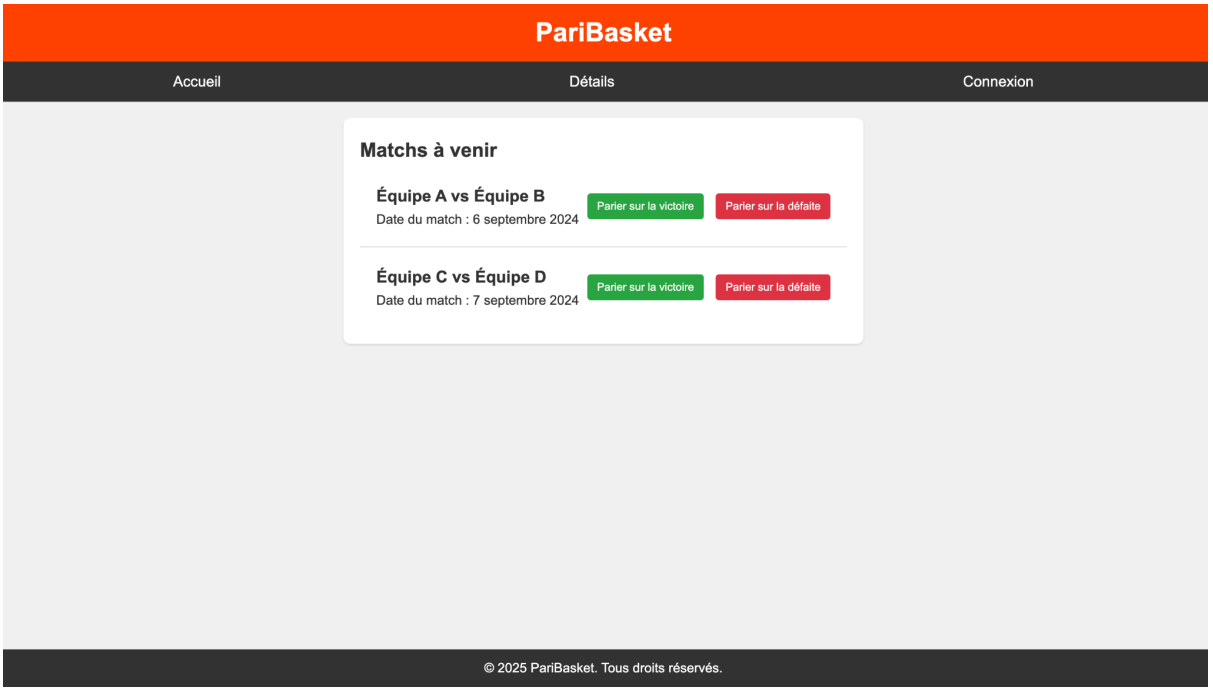
### 2.4 Wireframes

Afin de gagner du temps dans le développement, nous avons commencé par faire les wireframes des pages principales (page de connexion, accueil du site, détail d'un match).

Aperçu de la page de connexion :




Aperçu de la page d'accueil du site (après connexion):



Aperçu des détails d'un match :

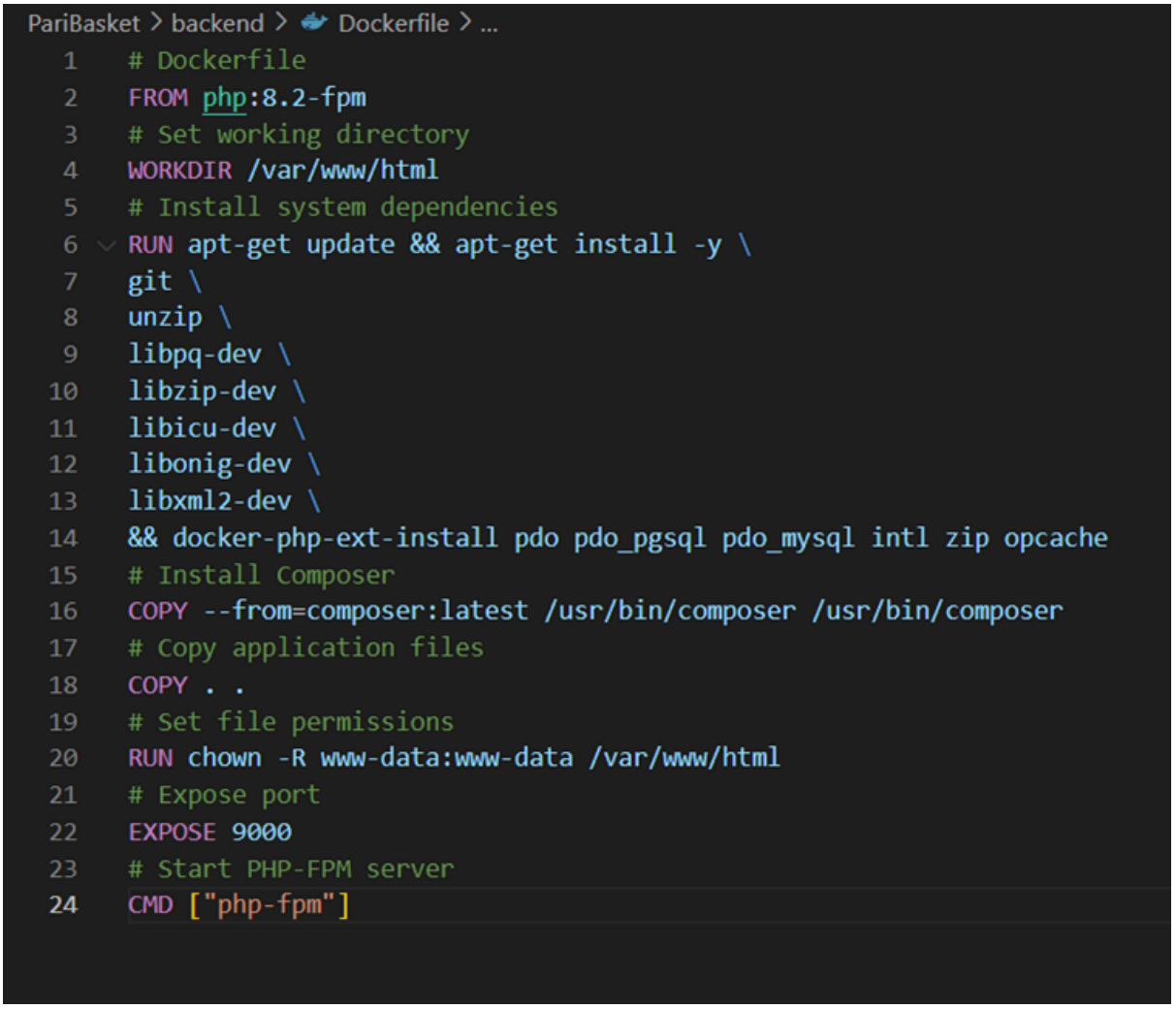

### 2.5 Scénarios de tests

Nous avons mis en place trois tests. Ils couvrent différents aspects de l'application Symfony. Deux tests unitaires, 'UserTest' qui est un test unitaire vérifiant la logique métier de la classe `User`, comme la détection d'emails déjà utilisés et le hachage des mots de passe, en isolant les dépendances avec des mocks. 'ParisServiceTest' qui vérifie la mise à jour des paris après un match, en simulant des scénarios de paris gagnants ou perdants.
'AuthenticationTest' est quant à lui un test fonctionnel qui simule des requêtes HTTP pour valider le comportement de l'application, comme la redirection vers la page de connexion et l'accès après authentification. L'objectif est de garantir que chaque composant fonctionne correctement.

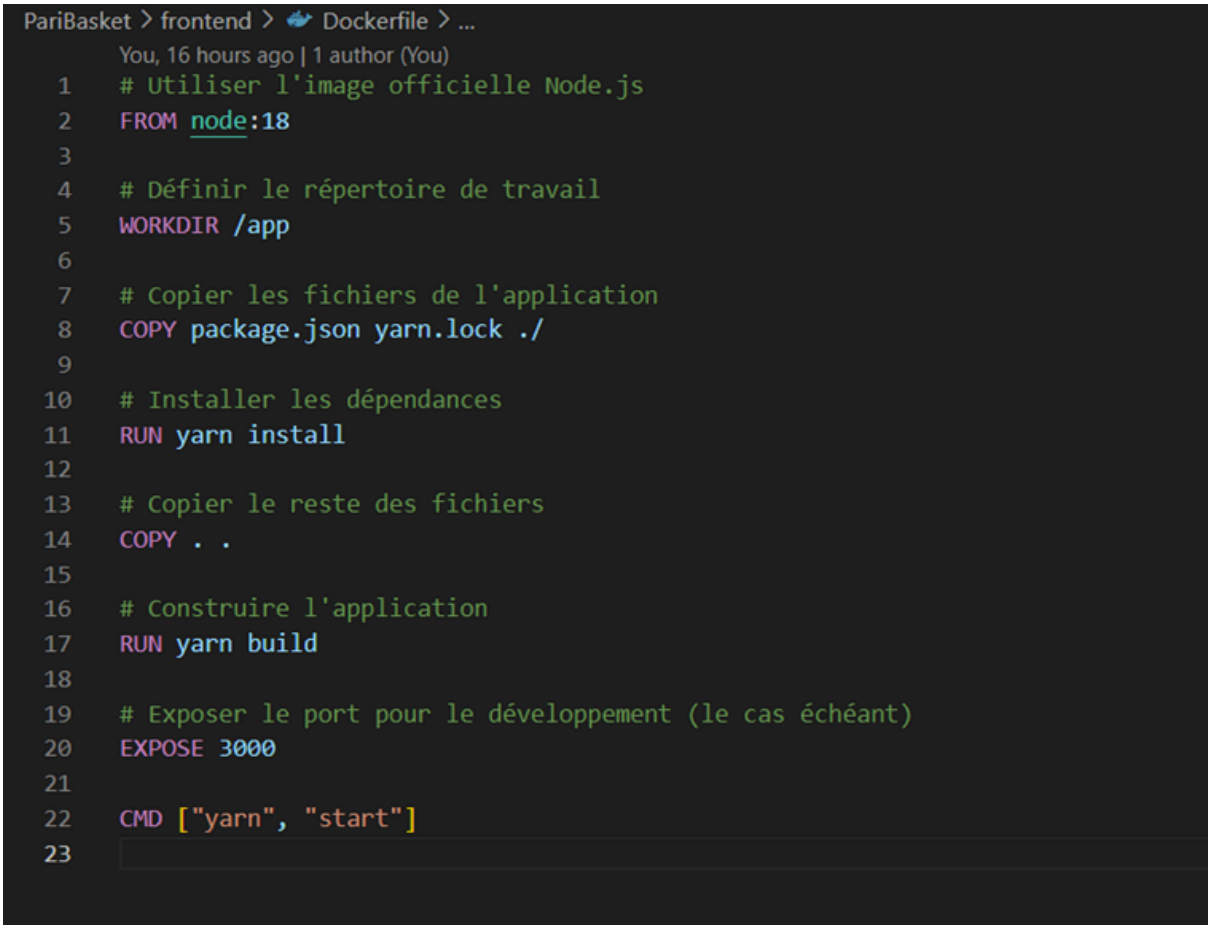
## 3. Déploiement

### 3.1 Conteneurisation avec Docker

Pour commencer nous avons installé docker sur nos machines. Une fois l'installation terminée, nous avons créé des dossiers afin de stocker les Dockerfile du backend et du frontend séparément. Ensuite, nous avons créé le Dockerfile correspondant au backend. Pour cela, nous nous sommes appuyés sur le fichier classique php :8.22-fpm qui existe déjà par défaut dans Docker. Cependant, nous avons rajouté des installations supplémentaires à faire au lancement de ce fichier (ex : git, pdo_mysql, ….). Nous avons modifié les permissions afin de s'assurer que l'utilisateur puisse accéder à l'arborescence que nous avons créée. Pour finir, nous avons ouvert le port 9000 par défaut afin de pouvoir tester le docker.




Pour le frontend (dans le cas ou nous aurions eu le temps d'utiliser React Native), nous avons utilisé l'image officielle de Node.js en copiant les fichiers de configuration (package.json et yarn.lock) afin de récupérer et installer toutes les dépendances présentes dans le projet. On récupère ensuite le reste des fichiers afin de construire l'application. Enfin, nous exposons cette fois le port 3000 afin de visualiser l'application sur le navigateur.




Pour lancer ces fichiers et installer directement l'environnement nécessaire, nous avons créé un fichier docker-compose.yml. Ce fichier sert à regrouper tous les services nécessaires au fonctionnement de l'application. Désormais, le lancement du fichier créera automatiquement des containers docker contenant le backend, le frontend et la base de données.

### Comment créer un fichier docker-compose.yml ?

Un fichier docker-compose.yml est utilisé pour définir et orchestrer plusieurs **services** Docker qui composent une application. Il permet de gérer facilement leurs interactions (par exemple, la communication entre un backend et une base de données).

Dans ce fichier, nous déclarons les **services** (par exemple : backend, frontend, base de données), ainsi que leurs configurations (ports, dépendances, volumes, variables d'environnement, etc.). L'utilisation de docker-compose simplifie le déploiement de projets complexes.

### Qu'est-ce qu'un volume en Docker ?

Un volume est une fonctionnalité de Docker qui permet de partager des données entre le conteneur et la machine hôte ou entre plusieurs conteneurs. Il est utilisé pour stocker des données de manière persistante, même si le conteneur est supprimé ou recréé.

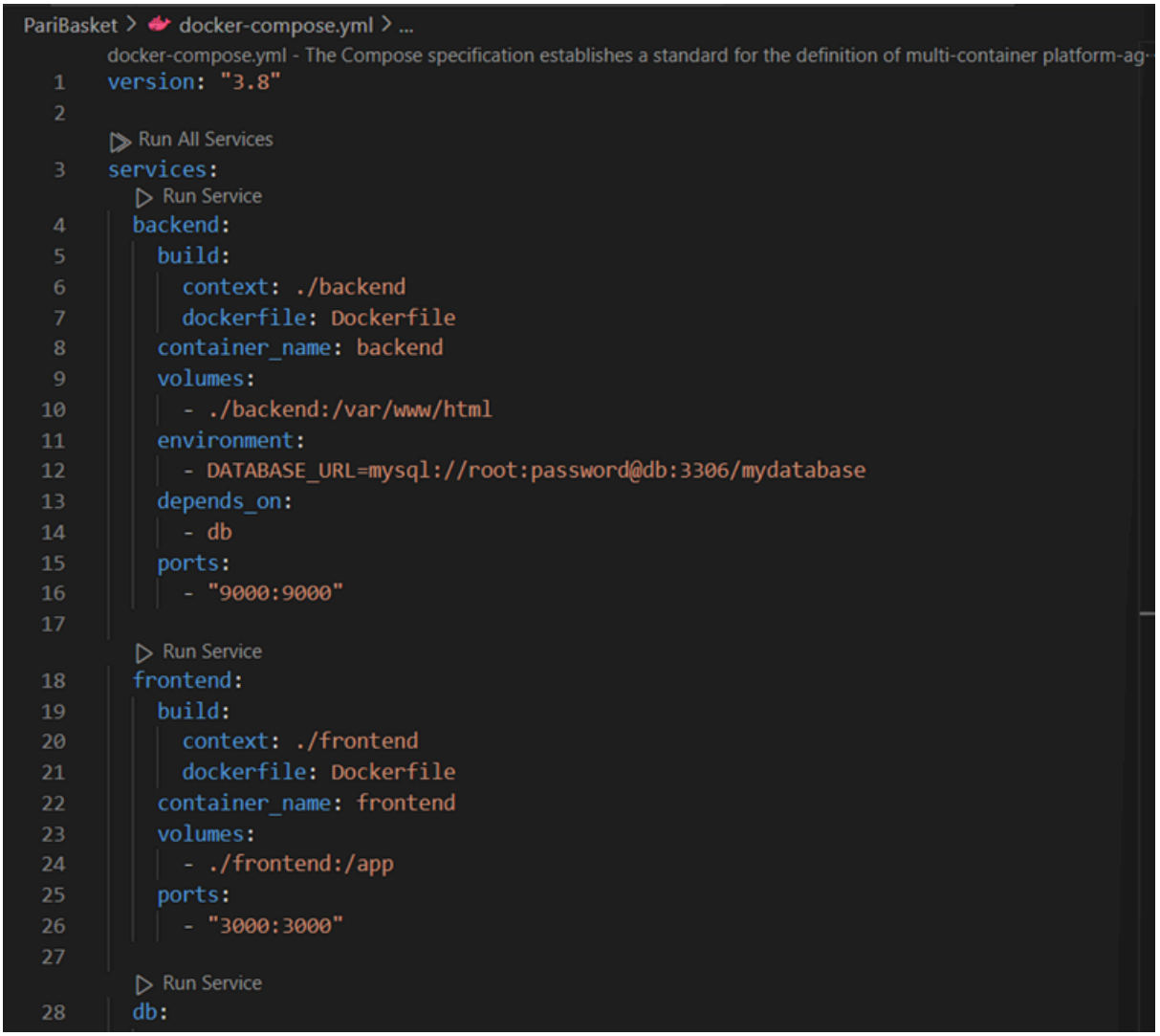
**Cela permet :**

- **De la persistance de données**
- **Du partage de données**
- **Séparation de la logique d'application et des données** : Les fichiers de l'application peuvent être gérés indépendamment des données persistantes.

**Exemple de volume dans le cas de mon service backend :**

Si tu montes un volume entre /var/www/html dans un conteneur (emplacement où le code est attendu) et un répertoire sur ta machine hôte (./backend), tu peux modifier les fichiers dans ./backend directement sur ton ordinateur, et ces modifications seront immédiatement disponibles dans le conteneur.

Pour le frontend, le fonctionnement est identique. Quant à la base de données, elle utilisera l'image officielle de mysql au lieu d'un Dockerfile. Il faudra aussi préciser les variables de l'environnement pour le nom de la base et le mot de passe.



```yaml
    image: mysql:8
    container_name: db
    environment:
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: mydatabase
    volumes:
      - db_data:/var/lib/mysql
    ports:
      - "3306:3306"

volumes:
  db_data:

```

De cette façon, il faudra simplement écrire la commande `"docker-compose up --build"` pour créer les containers nécessaires et donner l'accès aux projets.

### 3.2 Automatisation avec GitHub Actions

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8
        env:
          MYSQL_ROOT_PASSWORD: projet
          MYSQL_DATABASE: Paribasket
        ports:
          - 3306:3306

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

      # Configuration du backend (Symfony)
      - name: Setup Backend
        working-directory: .
        run: |
          composer install
          php bin/console doctrine:database:create --if-not-exists
          php bin/console doctrine:schema:update --force

      # Lancer les tests Symfony
      - name: Run Backend Tests
        working-directory: ./backend
        run: php bin/phpunit

  deploy:
    runs-on: ubuntu-latest
    needs: build

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

      # Déployer les services avec Docker Compose
      - name: Deploy with Docker Compose
        run: |
          docker-compose pull
          docker-compose up -d
```

Pour automatiser les tests et le déploiement, nous avons créé un fichier ci-cd.yml qui va récupérer le contenu de la branche main et initialiser les services dont nous avons eu besoin (configuration de la base mysql,installation de Symfony et des dépendances, ….). Ensuite, le script lance les tests de la partie Symfony en lançant php bin/phpunit. La configuration du front est en commentaire car nous n'avons pas de projet react native pour le moment. Pour finir, le script vérifie le code, si celui-ci trouve un docker-compose dans le dossier docker,il construit l'application à l'aide de Docker Compose.

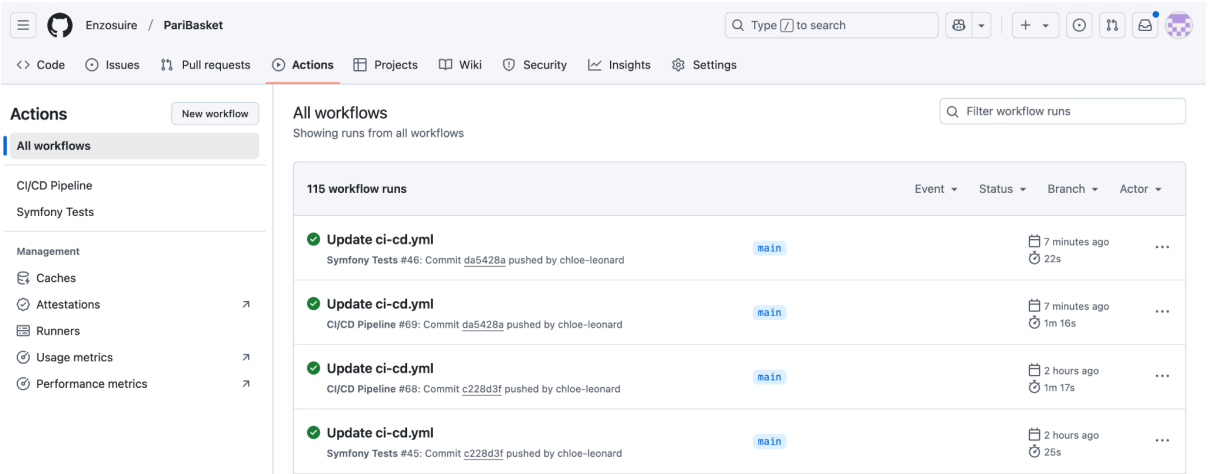
**Explication détaillé :**

Pour commencer, la partie 'on' sert à définir à quel moment nous allons lancer l'exécution (ici, le fichier s'exécute à chaque push effectué dans 'main'). Nous avons ensuite 2 jobs :

- build : qui va paramétrer le services mysql, puis effectuer les commandes peu à peu. On commence par la récupération du code, l'installation de PHP et les paramétrages (dépendances, extensions, ….). Ensuite, on configure MySQL avec les données qui se trouve dans le fichier .env.test. On vide les caches afin d'accélérer le montage des containers docker. On crée la base de données sur mySQL, puis on lance les migrations avec doctrine:schema:update. Pour finir, on lance les tests avec phpunit.
- deploy : qui va permettre de déployer l'application à l'aide du docker-compose existant. Pour cela, si le build s'est bien passé, on récupère le code du projet et on installe docker compose. Enfin, si docker-compose.yml existe dans le dossier docker, je pull (récupération des données) puis je lance 'docker-compose up -d ' pour créer mes containers et monter mon application.

Nous avons déplacé le docker-compose.yml dans un fichier pour éviter des conflits avec le fichier "compose.yaml" présents dans les projets symfony.

Nous avons un autre pipeline nommé "Symfony_tests" qui lance seulement l'exécution des tests unitaires.



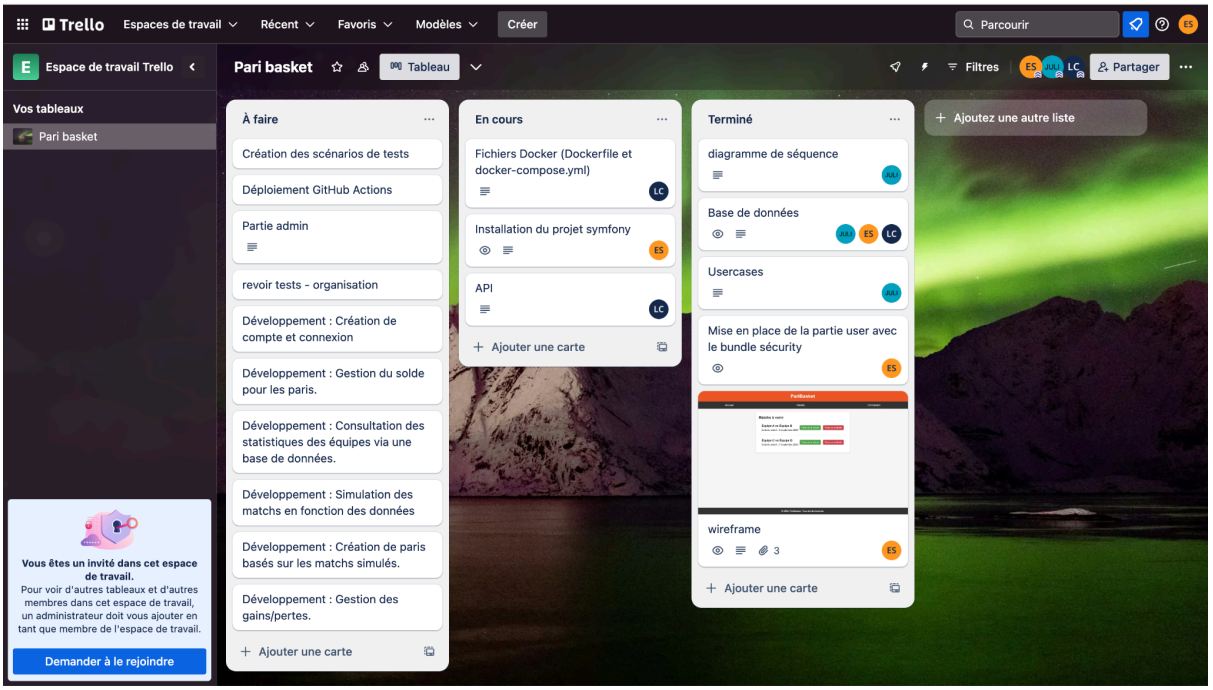
## 4. Collaboration

### 4.1 Utilisation de Git/GitHub

Pour simplifier les échanges de fichiers 😅😉, nous avons utilisé GitHub. Cela nous a permis de pouvoir travailler à plusieurs chacun sur une branche différentes. Une fois le développement d'une partie terminée, nous envoyons le code vers la branche main et gérons les conflits s'il y en a.
Nous avons beaucoup appris dans la manipulation de GitHub, cette semaine. En effet, à plusieurs reprises nous avons été confrontés à des problèmes,encore inconnus jusque là. Nous avons, dans ce contexte, appris de nouvelles commandes.

### 4.2 Gestion du projet en groupe

Pour ce projet, nous avons commencé par travailler ensemble sur la conception de la base de données afin d'être sûr qu'elle convienne à tout le monde. Cela à réduit aussi la possibilité de trouver des erreurs plus tard lors du développement.
Ensuite, nous avons créé un trello afin de lister toutes les tâches à faire pendant ce projet. Cela permet de visualiser l'avancée du projet. Nous nous sommes répartis les tâches petit à petit pendant la semaine, tout en prévoyant des temps de collaboration afin de montrer l'avancée de chaque personne. Nous en avons profité également pour faire des points quotidiens afin de partager nos compétences.



**Tâches Réalisées par :**

**Julie :**
Développement : Gestion des gains/pertes.
Développement : Création de paris
Conception - diagramme de séquence
Développement : Gestion du solde pour les paris
Conception - User Cases
Création des scénarios de tests

**Chloé :**
Conception - Fichiers Docker (Dockerfile et docker-compose.yml)
Développement : Consultation des statistiques des équipes via une base de données.
Développement : Simulation des matchs en fonction des données.
Développement : Affichage des données de l' API (Parti Matchs)
Création des scénarios de tests

**Enzo :**
Développement : Admin (Gestion des équipes et des statistiques ,suivi utilisateurs)
Développement : Installation du projet symfony (création base de données)
Développement : Mise en place de la partie Utilisateur avec le bundle security
Développement : Gestion du solde dans la rubrique profil
Conception - Wireframe

**Groupe :**
Conception de la base de données (Groupe)
Déploiement GitHub Actions(Groupe)
Développement : Structure de template et mise en forme css (Groupe)

Récapitulatif des outils utilisés :

*Conception*

1. **DrawSQL :** Outil de conception visuelle de schémas de bases de données pour modéliser et documenter des structures de bases de données relationnelles.
2. **Draw.io (Diagrams.net)** : Outil de création de diagrammes en ligne pour concevoir des schémas, organigrammes et architectures visuelles.

*Développement*

3. **IDE (Visual Studio Code)** : Éditeur de code
4. **Wamp/Mamp :** Environnement de développement local pour exécuter un serveur web, une base de données et PHP.
5. **phpMyAdmin :** Interface web pour gérer des bases de données MySQL/MariaDB, permettant de créer, modifier et exécuter des requêtes SQL.
6. **GitHub** (Actions) : Plateforme d'hébergement de code avec intégration CI/CD pour automatiser les workflows de développement et de déploiement.

*Communication/Organisation*

7. **Slack** : Outil de communication d'équipe pour discuter, partager des fichiers et intégrer des applications tierces.

8. **Trello :** Plateforme de gestion de projet en tableaux Kanban pour organiser des tâches et suivre l'avancement des projets.

## 5. Tests et Validation

Pour rappel, voici la liste des tests que nous avons effectués :

- deux tests unitaires
  - UserTest
  - ParisService
- un test fonctionnel
  - Authentication Text

Un test unitaire permet de tester une petite unité de code de manière isolée, en utilisant des mocks pour simuler les dépendances (comme l'EntityManager ou le UserPasswordHasher). Il vérifie le bon fonctionnement de chaque composant.
Le test fonctionnel (ou test d'intégration) teste le comportement de l'application dans son ensemble, en simulant des requêtes HTTP et en vérifiant les réponses du système (comme les redirections ou les accès aux pages).

### 5.1 Test Unitaire

UserTest

Le test 'UserTest' se décompose en trois parties principales

- configuration des mocks dans la méthode setUp()
- test unitaire dans la méthode testEmailAlreadyTaken()
- assertion pour vérifier la véracité d'une condition du test

Il se concentre sur la class User et ne teste pas les contrôleurs, les routes ou les vues.

La première méthode setUp(), permet de créer un mock.
Les mocks sont utilisés pour simuler les dépendances du service. On simule donc *EntityManager* de Doctrine, le repository de *User* puis on lui définit ces méthodes. On définit également un mock pour la class qui permet d'hasher le mot de passe.

```php
protected function setUp(): void
{
    // Mock de l'EntityManager
    $this->entityManager = $this->createMock(EntityManagerInterface::class);

    // Mock du UserRepository
    $this->userRepository = $this->createMock(EntityRepository::class);

    // Simuler `getRepository(User::class)` pour retourner notre mock
    $this->entityManager->method('getRepository')
        ->willReturn($this->userRepository);

    // Mock du password hasher
    $this->passwordHasher = $this->createMock(UserPasswordHasherInterface::class);
}
```

Puis pour simuler la création de compte avec le même mail, nous avons affecté une création d'utilisateur avec l'adresse "test@example.com". On vérifie donc que cet utilisateur existe puis on crée un deuxième utilisateur avec le même mail. Le test renvoie donc un message précisant que ce compte existe déjà. C'est la méthode *assertEquals()* qui permet de vérifier que le code se comporte comme attendu.

```php
0 references | 0 overrides
public function testEmailAlreadyTaken(): void
{
    // Simuler un utilisateur existant en BDD
    $existingUser = new User();
    $existingUser->setEmail(email: 'test@example.com');

    $this->userRepository->method('findOneBy')      You, 23 minutes ago • Aj
        ->with(['email' => 'test@example.com'])
        ->willReturn($existingUser);

    // Simuler une nouvelle tentative d'inscription avec le même email
    $newUser = new User();
    $newUser->setEmail(email: 'test@example.com');

    // Vérifier que l'utilisateur existe déjà
    $this->assertNotNull(
        $this->userRepository->findOneBy(['email' => $newUser->getEmail()]),
        'L\'email existe déjà en base de données.'
    );
}
```

Dans ce test, nous vérifions également le comportement du code lorsqu'il s'agit d'un ajout d'un utilisateur, et surtout que le mot de passe est bien hashé. Nous utilisons les mêmes mocks que précédemment ainsi que la méthode *assertEquals()* pour vérifier que le mot de passe saisi est bien hashé en base de données.

### 5.2 Test Fonctionnel

AuthentificationTest

Ce test permet d'éprouver l'ensemble de la stack Symfony (contrôleurs, routes, sécurité, base de données, etc.). Il vérifie que les fonctionnalités de l'application (comme l'authentification) fonctionnent comme prévu.

Il utilise *WebTestCase* (une extension de PHPUnit pour Symfony) pour les tests fonctionnels ainsi que le client HTTP de Symfony (*$client = static::createClient()*) pour simuler des requêtes. et enfin il vérifie des réponses HTTP avec des assertions spécifiques à Symfony (*assertResponseRedirects(), assertResponseIsSuccessful()*).

Pour vérifier qu'un utilisateur à seulement accès aux pages importantes s'il est connecté nous utilisons 2 tests.
Sur le premier, nous créons un client qui va tenter d'ouvrir le lien sans s'identifier.

```php
0 references | 0 overrides
public function testAuthenticationRequiredForBetting(): void
{
    $client = static::createClient();

    // Tentative d'accès à la page de profil sans être authentifié
    $client->request('GET', '/user');

    // Vérifier que l'utilisateur est redirigé vers la page de connexion
    $this->assertResponseRedirects('/login');
}
```

Sur le second, nous allons utiliser les identifiants d'un compte de test en base de données "test@example.com" pour simuler la connexion de cet utilisateur et de vérifier l'accès à la route '/user' qui est censé être accessible au seul utilisateur connecté. Une base de données de test a été créée au préalable, des accès sont spécifiés dans le fichier .env.test de l'application.
Cette méthode, *testAccessAfterLogin()*, commence par initialiser un client HTTP, avec la méthode *createClient()*, cela simule un navigateur. Les services et classes nécessaires, comme *l'EntityManager* et le *repository User* sont également récupérés pour interagir avec la base de données et l'entité User, afin d'avoir accès aux méthodes pour manipuler les données.
Ensuite, elle vérifie si un utilisateur de test existe déjà en base de données ; sinon, elle le crée dans la bdd de test avec les méthodes *persist()* et *flush()* et hache son mot de passe. Une fois l'utilisateur prêt, la méthode simule sa connexion avec *loginUser()* et tente d'accéder à une page protégée (/user).
Enfin, elle vérifie que l'assertion précédente est remplie. c'est-à-dire un user connecté accède bien à la route '/user'. La méthode *assertResponseIsSuccessful()* assure cette vérification.

```php
public function testAccessAfterLogin()
{
    $client = static::createClient();
    $entityManager = static::getContainer()->get('doctrine')->getManager();
    $userRepository = $entityManager->getRepository(User::class);

    // Vérifier si l'utilisateur existe déjà
    $testUser = $userRepository->findOneBy(['email' => 'test@example.com']);
    if (!$testUser) {
        $testUser = new User();
        $testUser->setEmail('test@example.com');
        $testUser->setRoles(['ROLE_USER']);

        // Hasher le mot de passe
        $passwordHasher = static::getContainer()->get('security.password_hasher');
        $testUser->setPassword($passwordHasher->hashPassword($testUser, 'password123'));

        $entityManager->persist($testUser);
        $entityManager->flush();
    }

    // Vérifier que l'utilisateur existe bien
    $this->assertNotNull($testUser, "L'utilisateur test@example.com n'a pas été trouvé en base.");

    // Simuler la connexion
    $client->loginUser($testUser);

    // Accéder à la page utilisateur après authentification
    $client->request('GET', '/user');

    // Vérifier que la page est bien accessible
    $this->assertResponseIsSuccessful();
}
```

A la fin de cette semaine, nous avons pu faire les pages de l'utilisateur, connexion/déconnexion, matchs, détails du matchs, ainsi que la gestion de paris et de soldes (gains/pertes). Du côté de GitHub, nous avons créé 2 pipelines:

- Symfony Tests gère l'installation et les paramètres de PHP et mySQL, la création et l'initialisation de la base de données puis le lancement des tests.
- CI/CD pipeline est identique à Symfony Tests cependant celui-ci gère également le job de déploiement vers docker-compose.

Une page Admin a été créée avec du contenu. Cependant, son accès n'a pas été géré. C'est-à-dire que n'importe quel utilisateur non connecté peut y accéder.

Difficultés rencontrées :

Nous avons mis du temps à adapter la base de données en fonction de ce que l'on récupérait par API et de ce qu'on avait besoin de stocker en base pour le réutiliser.
De plus, l'utilisation de github pour les partages de fichiers a été compliqué par le fait que personne dans le groupe ne l'utilise régulièrement. Nous avons donc passé du temps à débloquer le GitHub.

## Guide de déploiement :

Pour lancer l'application, il faudra cloner le projet avec la commande :

*git clone https://github.com/Enzosuire/PariBasket.git*

Ensuite, nous devrions pouvoir lancer le fichier docker-compose.yml (dans le dossier docker).Pour cela il faut se déplacer dans le projet, dans /docker et lancer la commande docker-compose up -d .
Le projet devrait alors s'installer et l'appli devrait être accessible sur http://localhost:80.

SI le docker-compose ne fonctionne pas, il faudra cloner le projet puis utiliser un serveur symfony en local. Pour cela, il faudra créer la base de données puis lancer la commande " symfony server:start -d".