Projet réalisé dans le cadre du passage du Titre Professionnel de

CONCEPTEUR DÉVELOPPEUR D'APPLICATIONS

Présenté par Jessy CHARLET

La_Plateforme - Toulon 2025

TOURNAMENT VB

APPLICATION COMPAGNON POUR LES JOUEURS ET CLUBS DE VOLLEY BALL

INTRODUCTION

Jessy Charlet, âgé de 35 ans, et je suis passionné par la technologie, la logique et la création. Mon parcours professionnel est riche et varié, marqué par une évolution constante et une adaptation à différents domaines tenant au commerce et à la technologie.

MON PARCOUR PROFESSIONNEL

Début de carrière dans la gravure pour bijoux (4 ans)

J'ai commencé ma carrière professionnelle à l'âge de 16 ans dans la gravure pour bijoux. Ce métier artisanal m'a permis de développer un sens aigu du détail et de la précision.

Évolution vers la maintenance et la création visuelle en ligne

Ensuite, j'ai évolué vers des responsabilités plus techniques et créatives en m'occupant de la maintenance de la boutique en ligne de l'entreprise. J'ai aussi créé la partie visuelle et promotionnelle de cette boutique, ce qui m'a permis de combiner mes compétences techniques et artistiques.

Ouverture d'une boutique d'impression à Hyères (5 ans)

Fort de cette expérience, j'ai ouvert ma propre boutique dans le centre-ville de Hyères, spécialisée dans l'impression sur tous supports et destinée aux clients particuliers. Durant 5 ans, j'ai appris à gérer cette entreprise, répondant aux besoins variés de mes clients ainsi qu'en développant mes compétences en gestion d'entreprise.

Expérience chez Micromania (1 an)

Par la suite, je suis resté un an chez Micromania, une expérience qui m'a permis de me familiariser avec le secteur du jeu vidéo et de la pop culture.

Création d'une entreprise de produits dérivés de la pop culture (7 ans)

Ma passion pour la pop culture m'a conduit à ouvrir une entreprise de vente de produits dérivés sur divers marketplaces comme Amazon, AliExpress, et Fnac. Au cours de ces 7 années, j'ai dirigé cette entreprise, tout en gérant la chaîne d'approvisionnement, le marketing, et les ventes en ligne.

Passion pour la Technologie et le Développement

Tout au long de ma carrière professionnelle, j'ai toujours eu énormément d'attrait pour la technologie et la création. Cette passion m'a conduit à vouloir approfondir mes compétences en développement. Actuellement, je me concentre sur le développement web, tout en m'intéressant également beaucoup au développement d'applications et de logiciels.

Objectifs de Formation

Après avoir travaillé de nombreuses années dans le commerce, j'ai décidé de franchir le pas et de me lancer dans le développement. J'ai obtenu mon diplôme de **Développeur Web et Web Mobile** (DWWM) l'année dernière et je passe actuellement celui de **Concepteur Développeur d'Applications** (CDA) dans ce but.

Mon objectif est de combiner mon expérience diversifiée en commerce et en technologie avec des compétences avancées en développement, afin de créer des solutions innovantes et répondre aux besoins actuels du marché.

RÉSUMÉ DU PROJET

Dans le monde du Volley Ball il est parfois difficile de se tenir au courant des différents tournois proposés autour de soi. Il faut souvent être abonné à tous les réseaux sociaux des différents clubs proches. En plus d'être inondé de publications qui ne nous intéressent pas forcément, on peut facilement passer à côté de celles qui publient un tournoi intéressant et encore plus difficilement s'y inscrire.
Grâce à l'application Tournament VB, il est maintenant possible de trouver instantanément tous les tournois proches de soi.
Il en va de même pour les clubs qui doivent gérer la communication autour de leurs tournois ainsi que les inscriptions, la sportive, etc...
L'objectif est donc de faciliter en tous points l'expérience des clubs et des joueurs.

PROCESSUS DE DÉVELOPPEMENT

Pour s'assurer que l'application réponde aux besoins réels des joueurs, une phase de recherche approfondie a été menée. J'ai rejoint plusieurs communautés de joueurs

d'Altered sur différents serveurs Discord afin de recueillir des retours d'expérience et identifier les besoins non satisfaits.

Les discussions avec les joueurs ont révélé un désir fort pour une solution qui leur permettrait d'avoir un aperçu clair et rapide de la proposition de tournois environnants. Les clubs, quant à eux, ont exprimé leur frustration face à la visibilité globale de leurs propres tournois et au manque d'outils centralisés pour gérer ceux-ci, et ont montré un grand intérêt pour une application qui pourrait combler ce vide.

CAHIER DES CHARGES

OBJECTIFS

Offrir une plateforme de vente spécialisée dans les chaises gaming :

- Proposer une large gamme de chaises gaming de haute qualité.
- Répondre aux besoins spécifiques des gamers en termes de confort, d'ergonomie et de style.

Améliorer l'expérience utilisateur :

- Faciliter la recherche et la comparaison de produits.
- Offrir des options de personnalisation pour répondre aux préférences individuelles des clients.

Augmenter la satisfaction client :

- Fournir des descriptions détaillées et des avis clients pour chaque produit.
- Offrir un suivi de commande transparent et en temps réel.

MVP (Minimum Viable Product)

Le MVP pour l'application Tournament VB doit se concentrer sur les fonctionnalités essentielles qui répondent aux besoins immédiats des joueurs et des clubs en leur offrant une valeur ajoutée significative dès le lancement. Voici les éléments clés du MVP :

1. Gestion social

- Amis : Envoyer des demandes d'amis aux autres joueurs afin de faciliter la créations d'équipes pour les joueurs.
- Équipes : Créer et sauvegarder des équipes afin de pouvoir les inscrire à un tournoi en un seul clic.
- Club : Enregistrer un club afin d'avoir accès à sa gestion et celle de ses tournois.

2. Gestion des tournois

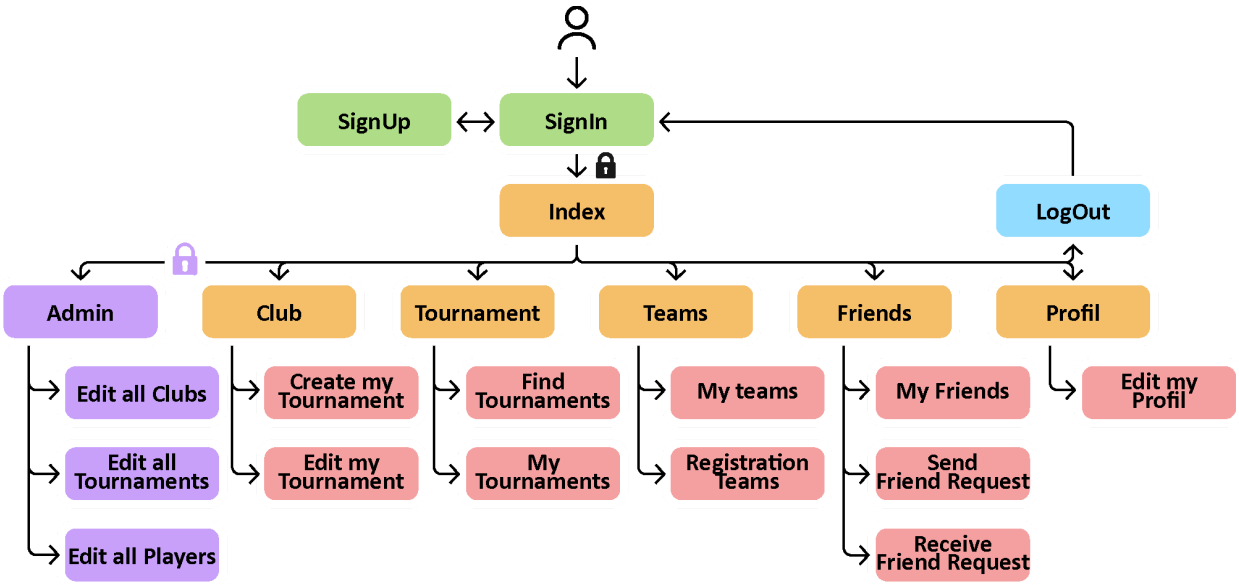
- Pour les joueurs : pouvoir identifier les tournois qui leurs correspondent (féminin, masculin, 4x4, 6x6, etc...) et s'y inscrire facilement.
- Pour les clubs : Publier et promouvoir facilement un tournoi, récupérer les inscriptions pour faciliter la création de la sportive du tournoi.

3. Interface Utilisateur Simple et Intuitive

- Navigation Facile : Concevoir une interface utilisateur claire et facile à naviguer, permettant aux joueurs d'accéder rapidement aux informations dont ils ont besoin.
- Compatibilité Multi-Plateforme : Assurer que l'application fonctionne de manière fluide sur les principaux appareils mobiles (iOS et Android) et sur le web.

MISE EN PLACE DE L'APPLICATION

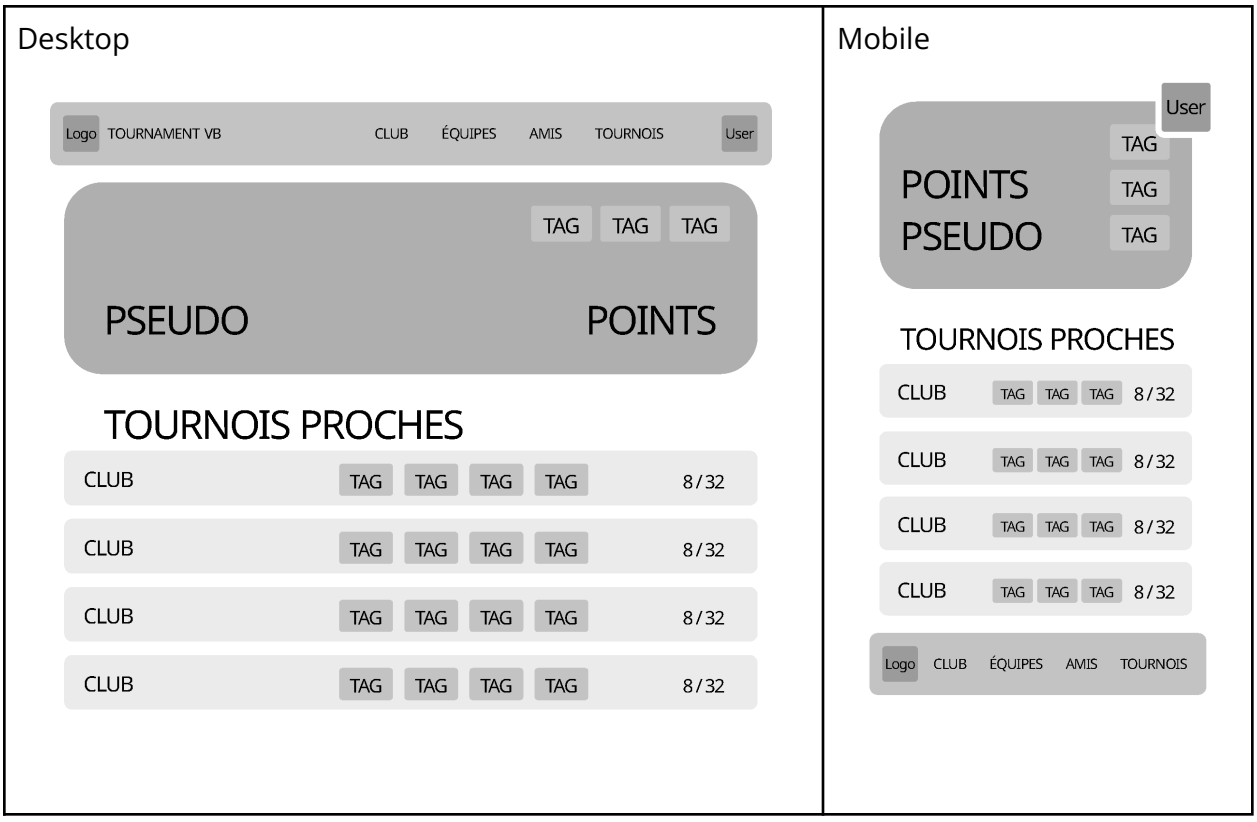
ARBORESCENCE



WIREFRAMES

Comme évoqué précédemment dans les fonctionnalités, mon souhait était de concevoir une application responsive mobile / tablette. Pour ce faire, j'ai abordé la conception des Wireframes selon une approche mobile-first.

Chaque joueur doit pouvoir lire son terrain facilement, ce qui a nécessité de scinder l'écran en deux pour que chaque utilisateur puisse avoir un sens de lecture adapté au mieux.



UTILISATEURS ET USER STORIES

PUBLIC VISÉ

Tous les joueurs, ainsi que tous les clubs de Volley Ball, du niveau amateur au niveau Elite. En conséquence, cela permet de toucher un très large public.

RÔLES

Player : utilisateur de base, il représente la majorité des users. Il à accès aux amis, à la création d'équipe et à la recherche et inscription aux tournois.

Club : Accessible aux représentants d'un club de Volley Ball. Il à accès aux mêmes fonctionnalités que le Player, mais dispose en plus du droit de créer et gérer ses propres tournois.

Admin: Accessible uniquement au gérant de Tournament VB. Il à accès aux mêmes fonctionnalités que Club, mais peut gérer tous les tournois, clubs et players.

USER STORIES

En tant que	je souhaite	afin de
player	visualiser les tournois proches	gagner du temps par rapport à une recherche classique sur les réseaux de tous les clubs
player	ajouter des amis	facilement créer des équipes en piochant dans mes amis disponibles et éligibles
player	créer des équipes	m'inscrir en un clic à un tournoi avec une équipe déjà enregistrée
player	modifier mon profil	qu'il soit toujours à jour si je change de poste ou de niveau
player	inscrire mon club	avoir accès à la gestion de celui ci
club	publier un tournoi	toucher un plus large public et gagner du temps sur la promotion de celui ci
club	récupérer la liste des équipes inscrites	faciliter la création de la sportive
admin	valider l'inscription d'un club	garantir la conformité du contenu disponible sur l'application
admin	blacklister un player ou un club	garder un environnement sain en cas de non respect des règles

VERSIONING

La gestion de version permet de conserver l'historique du code source et de travailler plus facilement en équipe. Même seul, il était très important de garder les versions précédentes du code, notamment en cas de gros problème avec une fonctionnalité suite à des

modifications dans le code. Pour cette gestion, j'ai utilisé Git.
J'ai au départ utilisé GitHub puis, par la suite, j'ai utilisé la liaison entre mon compte et intelliJ afin de créer des commits et push plus rapidement. Travaillant seul sur ce projet, je n'ai pas eu à utiliser de Pull Request afin de contrôler les merges de branches, ni même de Git Issues.

CHOIX TECHNOLOGIQUES

FRONT-END

Mon application repose sur une stack front-end simple et efficace : **HTML** pour structurer les pages, **CSS** pour gérer la mise en forme et le design responsive, et **JavaScript** pour dynamiser l'interface et offrir une meilleure interactivité à l'utilisateur. Cette base me permet de construire une interface claire, moderne et facile à maintenir.

BACK-END

Pour le back-end, j'ai choisi une stack centrée sur **Java** et le framework **Spring Boot**, qui permet de développer rapidement des applications web robustes et modulaires. J'utilise **Maven** pour la gestion des dépendances et le cycle de build. Pour les appels interservices, j'ai intégré **Feign**, qui simplifie la communication REST entre microservices. Enfin, j'ai opté pour **Thymeleaf** comme moteur de templates, afin de générer des vues dynamiques directement côté serveur. Cette combinaison me permet de mettre en place une architecture claire et évolutive..

BASE DE DONNÉES

Pour la partie base de données, j'ai utilisé une approche mixte avec **MySQL** pour la gestion des données relationnelles et **MongoDB** pour les données non structurées nécessitant plus de flexibilité. L'accès et la manipulation des données relationnelles se font via **Spring Data JPA**, ce qui facilite l'implémentation des opérations CRUD tout en réduisant le code "boilerplate". Cette combinaison permet de bénéficier à la fois de la fiabilité d'un SGBD relationnel et de la souplesse d'une base NoSQL, en fonction des besoins de l'application.

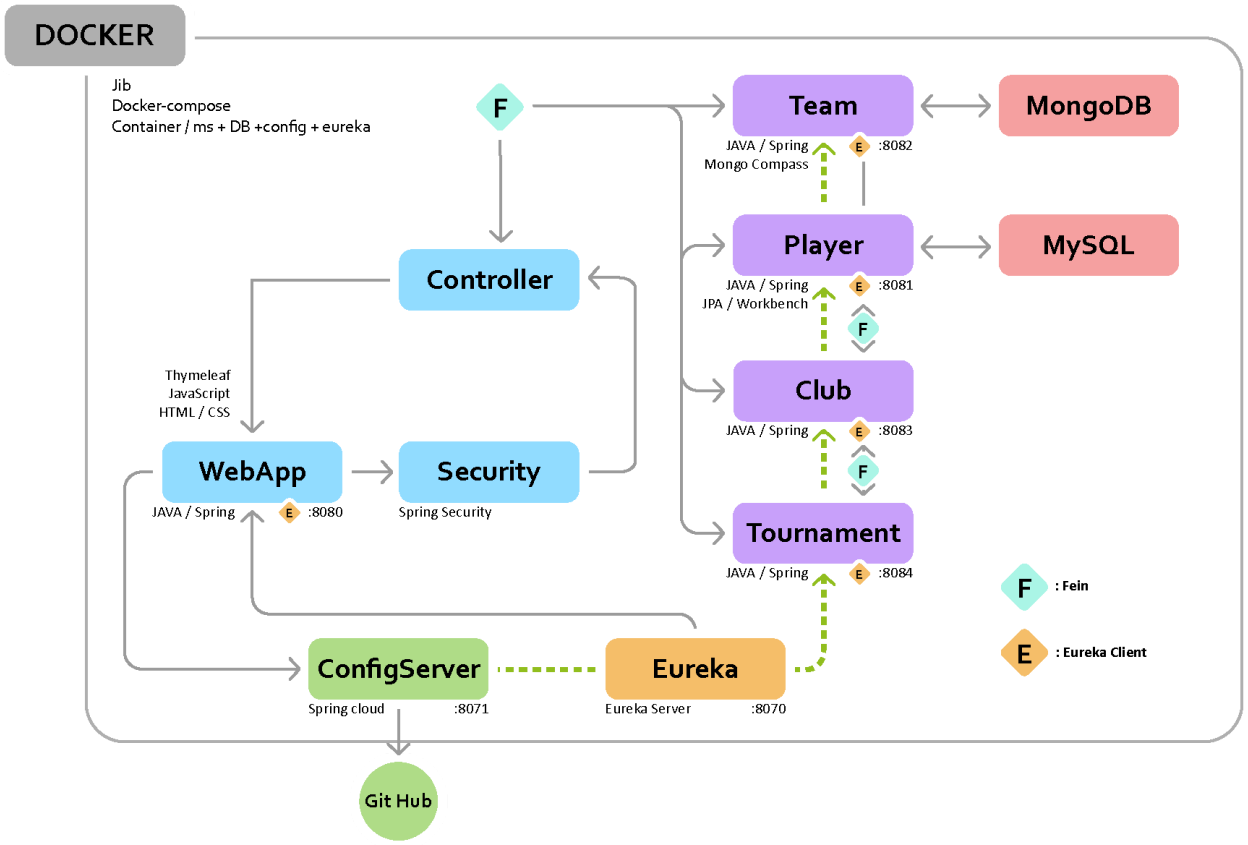
DEV OPS

Pour l'aspect DevOps, j'ai mis en place plusieurs outils afin d'assurer la gestion, le déploiement et la supervision de l'application. J'utilise **GitHub** comme plateforme de gestion de code source, ce qui permet un suivi clair des versions. L'enregistrement et la découverte des services se font grâce à **Netflix Eureka**, qui joue un rôle central dans l'architecture microservices en permettant aux services de se localiser dynamiquement entre eux. Enfin, j'ai intégré **Docker** afin de containeriser l'application et ses dépendances, ce qui garantit une exécution identique sur différents environnements (développement, test, production) et simplifie le déploiement.

ARCHITECTURE

Mon projet repose sur une architecture **microservices** déployée via **Docker**, permettant l'isolation et la scalabilité des différents services. Chaque microservice est développé en **Java/Spring** et gère une entité spécifique : **Team**, **Player**, **Club** et **Tournament**. Les bases de données sont choisies selon le type de service : **MongoDB** pour la gestion des équipes et **MySQL** pour les joueurs, clubs et tournois.

Le projet intègre également un **service de configuration centralisé (ConfigServer)** et un **Eureka Server** pour la découverte des services. La **WebApp**, développée en Spring avec Thymeleaf, HTML/CSS et JavaScript, interagit avec les microservices via un **Controller** et est sécurisée grâce à **Spring Security**. L'ensemble est orchestré dans des conteneurs Docker pour faciliter le déploiement et la maintenance.



GESTION DES BDD

MODÈLE PHYSIQUE DE DONNÉES

Mon modèle physique de données est organisé autour de plusieurs entités principales liées à la gestion de tournois et des joueurs :

- **Player** constitue l'entité centrale, contenant les informations personnelles et sportives des joueurs. Elle est reliée à **Club** (appartenance à un club) et à **Users** (compte applicatif et authentification).

- Les relations sociales sont gérées via la table **Friends**, qui permet de modéliser les liens entre joueurs.

- L'organisation des compétitions repose sur l'entité **Tournament**, qui stocke toutes les caractéristiques d'un tournoi (dates, lieu, format, contraintes, prix, etc.).

- La participation des joueurs et des équipes est représentée par les tables d'association **Team_registered** et **Team_registered_player**, ce qui permet de gérer respectivement l'inscription des équipes à un tournoi et la composition des équipes par joueur.

- Les résultats des matchs sont conservés dans la table **Result**, liée directement aux tournois et aux équipes participantes.

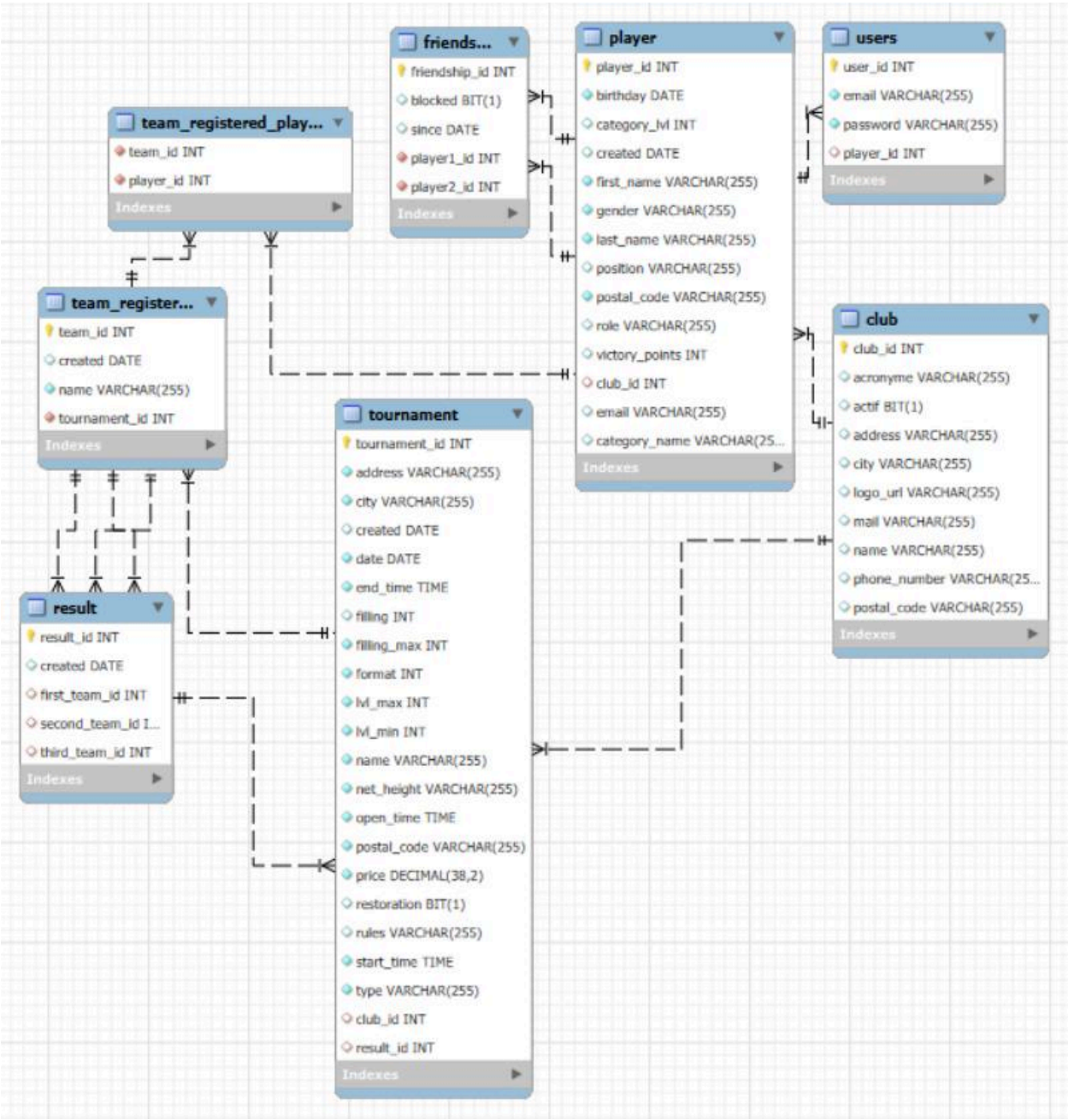
Ce modèle respecte les principes de **normalisation** :

- séparation claire des entités (joueurs, clubs, tournois, équipes, résultats),

- utilisation de tables d'association pour gérer les relations N-N,

- cohérence et extensibilité (possibilité d'ajouter de nouveaux attributs ou types de compétitions sans remise en cause de la structure).

Cette modélisation offre une base robuste et évolutive pour gérer l'ensemble des données liées aux joueurs, aux compétitions et aux résultats.



JPA

L'implémentation du modèle physique de données a été réalisée avec **Spring Data JPA**, ce qui a permis de mapper les tables de la base en **entités Java**. Chaque table (par exemple **Player**, **Club**, **Tournament**, **Result**) correspond à une classe annotée avec **@Entity**. Les clés primaires sont définies à l'aide de l'annotation **@Id** et gérées automatiquement grâce à **@GeneratedValue**.

Les relations entre entités respectent le modèle :

- Les relations **OneToMany** et **ManyToOne** sont utilisées pour lier, par exemple, un **Club** à ses **Tournaments**, ou un **Tournament** à ses **Results**.
- Les relations **ManyToMany** sont gérées par des tables intermédiaires, comme **Team_registered_player**, et sont implémentées via l'annotation **@ManyToMany** avec une entité de jointure.
- Les relations **OneToOne**, comme entre **Player** et **Users**, sont représentées avec l'annotation **@OneToOne**.

Grâce à **Spring Data JPA**, les opérations **CRUD** (création, lecture, mise à jour, suppression) sont fortement simplifiées : il suffit de définir des interfaces **Repository** (par exemple **PlayerRepository**, **TournamentRepository**), qui héritent de **JpaRepository**, pour bénéficier de méthodes prédéfinies (findAll(), save(), deleteById(), etc.) sans écrire de code **SQL** explicite.

Cette approche permet de :

- réduire le code répétitif lié aux requêtes SQL,
- garantir la cohérence entre le modèle objet et la base relationnelle,
- faciliter l'évolution du schéma en ne modifiant que les entités, sans réécrire toutes les requêtes.

JPA m'a permis d'assurer une correspondance directe entre mon **MPD** et mon code Java, tout en gardant une architecture claire et maintenable.

```java
@Repository  4 usages  Jessy Charlet *
public interface FriendshipRepository extends JpaRepository<Friendship, Integer> {

    @Query("SELECT f FROM Friendship f WHERE f.player1.id = :playerId OR f.player2.id = :playerId")
    List<Friendship> findAllByPlayerId(@Param("playerId") Integer playerId);


    @Query("SELECT COUNT(f) > 0 FROM Friendship f " +  no usages  new *
            "WHERE (f.player1.id = :playerId1 AND f.player2.id = :playerId2) " +
            "   OR (f.player1.id = :playerId2 AND f.player2.id = :playerId1)")
    boolean existsFriendship(@Param("playerId1") Integer playerId1,
                             @Param("playerId2") Integer playerId2);
}
```

```java
@Entity  Jessy Charlet *
@Getter
@Setter
@ToString
@AllArgsConstructor
@NoArgsConstructor
@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")
public class TeamRegistered {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "team_id")
    private Integer id;

    @NotBlank(message ="Le nom de l'équipe est obligatoire.")
    private String name;

    @NotNull(message ="Une équipe doit être rattachée à un tournoi.")
    @ManyToOne
    @JoinColumn(name = "tournament_id", nullable = false)
    @JsonBackReference
    private Tournament tournament;

    @ManyToMany
    @JoinTable(
            name = "team_registered_player",
            joinColumns = @JoinColumn(name = "team_id"),
            inverseJoinColumns = @JoinColumn(name = "player_id")
    )
    @JsonBackReference
    private List<Player> players = new ArrayList<>();

    private LocalDate created;
}
```

LE CODE

CSS

J'ai structuré mon CSS de manière moderne et efficace. J'ai utilisé des **variables CSS** pour centraliser les couleurs, les tailles et les marges, ce qui facilite les modifications et assure la cohérence du design. J'ai également intégré **des calculs dynamiques avec calc()**, permettant d'ajuster certains éléments de manière flexible selon la taille de l'écran. Enfin, j'ai travaillé le **responsive design** grâce aux **media queries**, afin que le site s'adapte correctement sur différents supports, du mobile à l'écran large. Cette approche rend le CSS à la fois maintenable et évolutif.

```css
/*__globals variables_____*/
:root {
    --max-width: 1200px;
    --radius: 1.5rem;
    --padding: 0.2rem;
    --margin: 1rem;
    --font-size: 0.8rem;
    color-scheme: light dark;
    --white: rgb(230, 230, 230);
    --white-dark: #EAEAEA;
    --black: #000000;
    --black-light: #1F1F1F;

    --font: light-dark(var(--black), var(--white));
    --background: light-dark(var(--white), var(--black));
    --alt: light-dark(var(--white-dark), var(--black-light));
    --primary: light-dark(#FF2E63, #D6FF00);
    --secondary: light-dark(#D6FF00, #FF2E63);
    --link: #08D9D6;
    --yes: #48FF00;
    --no: #FF0004;
    --male: #00D0FF;
```

```css
a, button {
    height: 2rem;
    padding: calc(var(--padding) * 2) calc(var(--padding) * 4);
    border: none;
    text-decoration: none;
    display: flex;
    gap: 0.5rem;
    align-items: center;
    color: light-dark(var(--black-light), var(--white));
    border-radius: calc(var(--radius) * 2 - var(--padding) * 2) calc(var(--radius) - var(--padding) * 2);

    .header-icon {
        height: calc(var(--radius) + var(--padding));
        width: calc(var(--radius) + var(--padding));
        color: var(--primary);
    }

    &:hover, &:focus {
        background-color: var(--primary);
        color: var(--alt);
```

THYMELEAF

J'ai utilisé **Thymeleaf** pour rendre mes pages HTML dynamiques et interactives. J'ai exploité des attributs comme **th:text** et **th:value** pour afficher des données côté serveur, et **th:field** pour lier facilement les formulaires à mes objets Java. Les boucles **th:each** et les conditions **th:if / th:unless** permettent de gérer l'affichage conditionnel et répétitif de manière claire. J'ai également utilisé **th:action** et **th:object** pour simplifier la soumission des formulaires, et **th:replace** pour réutiliser des fragments HTML. Enfin, **th:id** m'a permis de générer des identifiants dynamiques pour mes éléments, ce qui rend le code plus modulable et maintenable.

J'ai utilisé un **ViewController** pour gérer la communication entre le serveur et la vue. Grâce à **ModelAndView**, je peux préparer les données côté serveur et les transmettre à la page

Thymeleaf. J'utilise **@ModelAttribute** pour lier automatiquement les objets Java aux formulaires, ce qui simplifie la récupération et l'affichage des informations. Ainsi, le contrôleur centralise la logique métier tout en permettant à la vue de rester dynamique et réactive.

```
<div th:if="${(user != null)}" id="my-teams-content" class="nav-inside-content active">
    <div th:each="team : ${teams}" th:if="${(teams != null)}" class="team">
        <div th:text="${team.getName()}"></div>
        <div th:text="${team.getFormat()} + 'x' + ${team.getFormat()}"></div>
        <div th:text="${team.getType()}"></div>
        <form th:action="@{/team/delete}" th:object="${DeleteTeamForm}" method="post">
            <input type="hidden" th:id="${team.getId()}" th:value="${team.getId()}" name="teamId"/>
            <button type="submit" value="X" class="no">×</button>
        </form>
        <div>
            <div th:each="player : ${team.getPlayers()}" th:if="${(team.getPlayers() != null)}" class="tag">
                <span th:text="${(player.getFirstName())}" class="capitalize">/span>
                <span th:text="${(player.getLastName())}" class="uppercase"></span>
            </div>
        </div>
    </div>
    <div th:unless="${(teams != null)}" class="empty">
        Tu n'as pas encore créé d'équipe...
    </div>
</div>
```

CONFIGURATION

Pour centraliser et simplifier la gestion de la configuration, j'ai mis en place un **Config Server** avec Spring Cloud. Celui-ci permet de charger les paramètres de l'application à partir d'un **dépôt Git**, ce qui rend la configuration versionnée, traçable et facilement partageable. Cette approche me permet de séparer le code de la configuration et de modifier certains réglages (comme les URL de base de données, les clés d'API ou les profils d'environnement) sans avoir besoin de redéployer l'application. J'ai également utilisé **Eureka Server** comme registre de services, ce qui facilite la **découverte automatique** et la communication entre microservices, tout en réduisant le couplage. Grâce à cette organisation, mon application est plus **souple, maintenable et adaptable** à différents environnements (dev, test, prod).

```yaml
server:
    port: 8071
spring:
    application:
        name: configServer
    profiles:
        active: git
    cloud:
        config:
            server:
                git:
                    uri: "https://github.com/jessy-charlet/Tournament-config"
                    default-label: main
                    timeout: 5000
                    clone-on-start: true
                    force-pull: true
management:
    endpoints:
        web:
            exposure:
                include: health,info
    health:
        readiness-state:
```

Tournament-config / **eureka.yml**

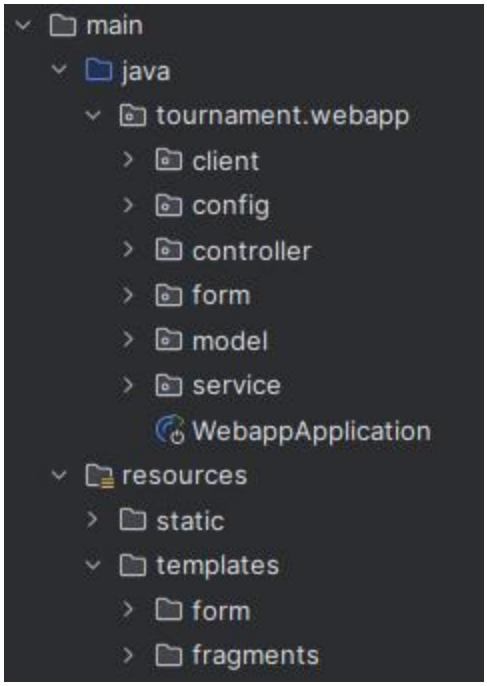
Jessy-Charlet Add Configs

Code | Blame | 32 lines (31 loc) · 691 Bytes

```yaml
server:
    port: 8070
eureka:
    instance:
        hostname: localhost
    client:
        fetchRegistry: false
        registerWithEureka: false
        serviceUrl:
            defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
management:
    endpoints:
        web:
            exposure:
                include: health,info
    health:
        readiness-state:
            enabled: true
        liveness-state:
            enabled: true
    endpoint:
        health:
            probes:
                enabled: true
    info:
        env:
```

ARCHITECTURE EN COUCHES

Chaque micro-service suit une **architecture en couches** afin de séparer les responsabilités et de rendre le code plus maintenable. La couche **Controller** gère les requêtes et fait le lien avec la vue. La couche **Service** contient la logique métier et centralise les traitements. La couche **Repository** s'occupe de la communication avec la base de données. Les **Models** représentent mes entités principales, tandis que les **Forms** me servent à gérer et valider les données issues des formulaires. La couche **Client** permet d'interagir avec des services externes si nécessaire. Enfin, la couche **Config** regroupe la configuration technique (sécurité, mapping, dépendances, etc.). Cette organisation rend le projet plus clair, évolutif et facile à tester.



FEIGN

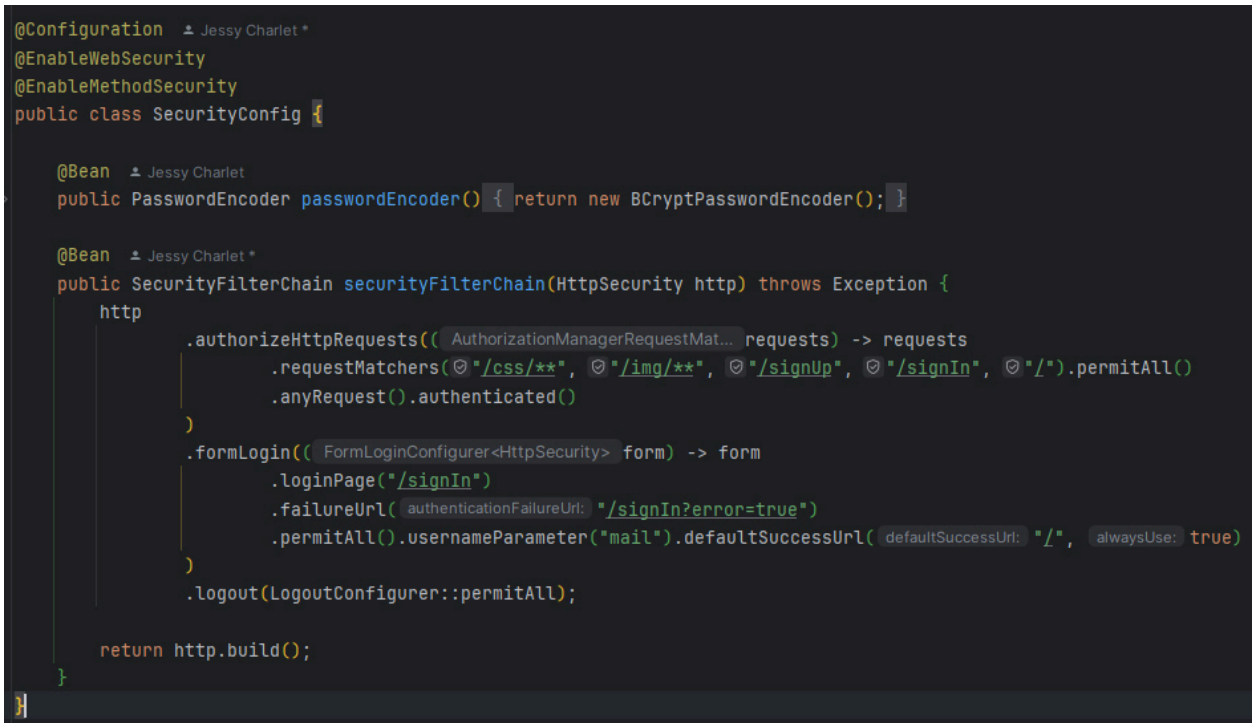
J'ai utilisé **Feign** comme client HTTP afin de simplifier la communication entre les micro-services. Grâce à ses **interfaces annotées**, je peux définir facilement les appels aux API sans avoir à écrire de code bas niveau pour gérer les requêtes ou les réponses. Feign s'intègre directement avec Spring, ce qui me permet de récupérer les données distantes et de les injecter dans ma logique métier de manière propre et maintenable. Cette approche réduit le couplage et rend le code plus lisible.

```java
@SpringBootApplication  Jessy Charlet
@EnableFeignClients
@RefreshScope
public class WebappApplication {

    public static void main(String[] args) { SpringApplication.run(WebappApplication.class, args); }

}
```

```java
@FeignClient("player")  12 usages  Jessy Charlet
public interface PlayerFeignClient {
    @RequestMapping(method = RequestMethod.POST, value = "player/byMail", consumes = "application/json")
    Optional<PlayerDTO> findPlayerByEmail(String email);

    @RequestMapping(method = RequestMethod.POST, value = "player/add", consumes = "application/json")
    Player addPlayer(SignUpForm form);

    @RequestMapping(method = RequestMethod.POST, value = "player/byId", consumes = "application/json")
    PlayerDTO findPlayerById(Integer id);

    @RequestMapping(method = RequestMethod.POST, value = "player/friend/add", consumes = "application/json")
    PlayerDTO addFriend(AddFriendForm form);
}
```

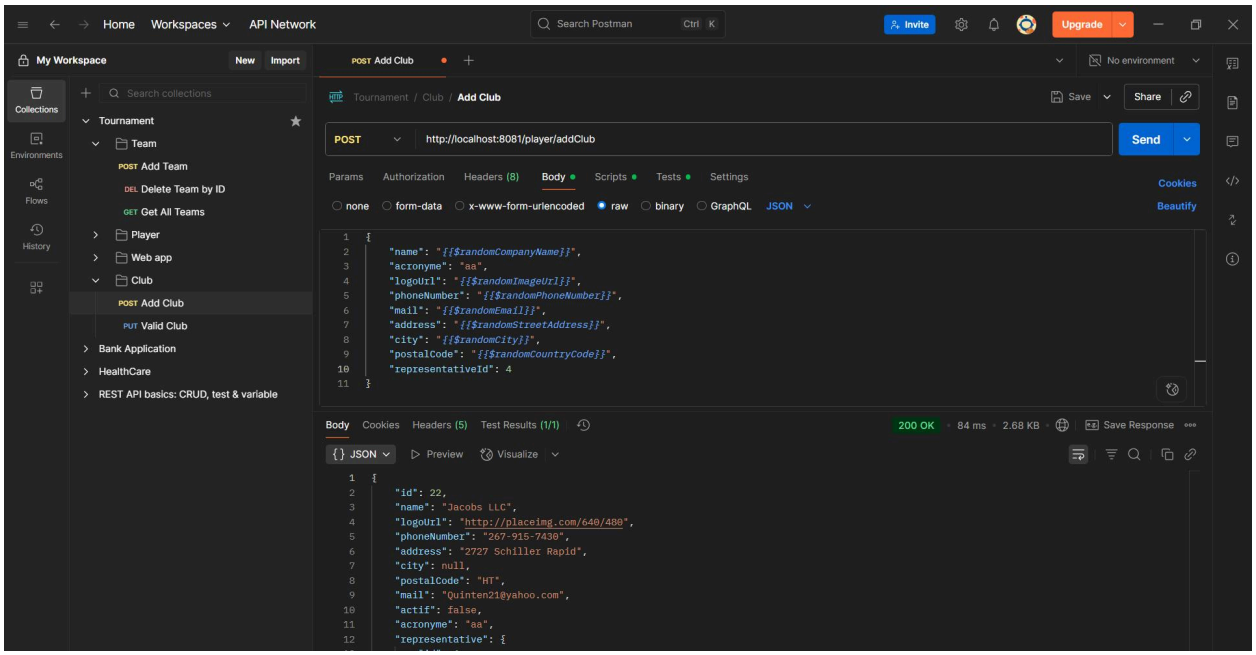
SECURITY

Pour sécuriser mon application, j'ai mis en place **Spring Security**. J'ai créé une classe **SecurityConfig** afin de définir les règles d'authentification et d'autorisation, par exemple pour restreindre l'accès à certaines pages selon les rôles des utilisateurs. Le mécanisme d'**authentication** me permet de gérer l'identification des utilisateurs et la validation de leurs identifiants. Enfin, avec **MvcConfig**, j'ai configuré les vues liées à la sécurité, comme les pages de connexion ou de déconnexion. Cette configuration assure une gestion claire et centralisée de la sécurité dans mon application.



TESTS CRUD

Pour valider le bon fonctionnement de mes opérations **CRUD**, j'ai utilisé **Postman**. J'ai testé chaque endpoint en envoyant des requêtes **POST**, **GET**, **PUT** et **DELETE** afin de vérifier respectivement la création, la lecture, la mise à jour et la suppression des données. Postman me permet aussi d'inspecter facilement les réponses JSON et les statuts HTTP pour m'assurer que les contrôleurs, services et repositories interagissent correctement avec la base de données. J'ai également utilisé des **variables dynamiques** comme **{{$randomEmail}}** pour générer automatiquement des données aléatoires lors des tests, ce qui m'a permis de diversifier les cas de test et d'automatiser davantage la validation. Ces tests garantissent la fiabilité de l'application et facilitent la détection d'éventuelles erreurs.



TEST D'INTÉGRATIONS

Pour assurer la fiabilité de mon code, j'ai également mis en place des **tests unitaires**. Contrairement aux tests d'intégration, ils se concentrent sur une méthode ou une classe précise, sans charger tout le contexte applicatif. J'ai utilisé **JUnit** et **Mockito** pour isoler les dépendances et vérifier que ma logique métier fonctionne correctement. Cela me permet de détecter rapidement les erreurs, de garantir la qualité du code et de faciliter la maintenance au fil du temps.

```java
class PlayerServiceTest {

    @Mock 2 usages
    private FriendshipRepository friendshipRepository;

    @InjectMocks 2 usages
    private PlayerService playerService;

    private Player player1; 4 usages
    private Player player2; 4 usages
    private Player player3; 4 usages

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks( testClass: this);

        player1 = new Player();
        player1.setId(1);

        player2 = new Player();
        player2.setId(2);

        player3 = new Player();
        player3.setId(3);
    }

    @Test
    void testGetFriendsByPlayerId_withFriends() {
        // Arrange
        Friendship f1 = new Friendship();
        f1.setPlayer1(player1);
        f1.setPlayer2(player2);
        Friendship f2 = new Friendship();
        f2.setPlayer1(player3);
        f2.setPlayer2(player1);

        when(friendshipRepository.findAllByPlayerId(1))
                .thenReturn(Arrays.asList(f1, f2));

        // Act
        List<Player> friends = playerService.getFriendsByPlayerId(1);

        // Assert
        assertEquals( expected: 2, friends.size());
        assertTrue(friends.contains(player2));
        assertTrue(friends.contains(player3));
    }

    @Test
    void testGetFriendsByPlayerId_noFriends() {
        // Arrange
        when(friendshipRepository.findAllByPlayerId(1))
                .thenReturn(Collections.emptyList());

        // Act
        List<Player> friends = playerService.getFriendsByPlayerId(1);

        // Assert
        assertNotNull(friends);
        assertTrue(friends.isEmpty());
    }
}
```

DOCKER

Pour faciliter le déploiement et la portabilité de mon application, j'ai utilisé **Docker**. Chaque composant (application, base de données, services externes…) est encapsulé dans un **conteneur**, ce qui garantit un environnement stable et identique quel que soit le poste ou le serveur. J'ai également mis en place **Docker Compose** pour orchestrer plusieurs conteneurs en même temps : par exemple, lancer mon application avec sa base de données et ses services associés en une seule commande. Enfin, j'utilise un **registry Docker** (Docker Hub) pour publier et partager mes images, ce qui permet de déployer facilement l'application sur différents environnements ou serveurs. Cette approche simplifie énormément le développement, les tests et le déploiement, tout en rendant l'application plus **fiable et reproductible**.

```yaml
networks:
  backend:
  frontend:

services:

  # Config Server
  configserver:
    image: configserver:latest
    build:
      context: ./ConfigServer
    ports:
      - "8071:8071"
    environment:
      - SPRING_PROFILES_ACTIVE=dev
    networks:
      - backend
    depends_on:
      - db_mysql
      - db_mongo

  # Eureka Server
  eurekaserver:
    image: eurekaserver:latest
    build:
      context: ./Eureka
    ports:
      - "8070:8070"
    networks:
      - backend
    depends_on:
      - configserver

  # Microservices
  player:
    image: player:latest
    build:
      context: ./Player
          ports:
      - "8081:8081"
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://db_mysql:3306/playerdb
      SPRING_DATASOURCE_USERNAME: playeruser
      SPRING_DATASOURCE_PASSWORD: player123
    networks:
      - backend
    depends_on:
      - eurekaserver
```