Rapport de Projet Java Avancé

Application de Gestion Intelligente des Billets "Mondial 2030"

Réalisé par : Lafdaigui Ayoub

Encadré par : M. Abderrahim LARHLIMI

Année Universitaire 2025-2026

Remerciements

Je tiens tout d'abord à remercier mon professeur encadrant, **M. Abderrahim LARHLIMI**, pour ses conseils avisés, sa méthode pédagogique et sa disponibilité tout au long de ce module de Java Avancé. Je remercie également mes collègues pour les échanges fructueux qui ont permis de surmonter certains obstacles techniques.

Table des matières

Table des figures

Chapitre 1

Introduction Générale

1.1 Contexte du Projet

L'attribution de l'organisation de la Coupe du Monde 2030 représente un défi logistique et technologique majeur. Au cœur de cet événement planétaire se trouve la gestion de la billetterie, un système critique qui doit être capable de gérer des millions de requêtes, d'assurer une équité parfaite dans l'attribution des places, et de garantir une sécurité sans faille contre la fraude. C'est dans ce contexte que s'inscrit le projet "Mondial 2030". Il s'agit de développer un prototype fonctionnel et robuste d'une application de gestion de bielles (Desktop), simulant les contraintes réelles d'un tel système.

1.2 Problématique

Comment concevoir une application Java de bureau (Desktop) capable de gérer efficacement le cycle de vie complet d'une billetterie (création de matchs, vente, génération de billets), tout en assurant l'intégrité des données face à des accès concurrents et en offrant une expérience utilisateur fluide ? Les défis principaux sont :

- **La Persistance et l'ORM** : Comment mapper des objets Java complexes vers une base de données relationnelle sans compromettre les performances (Lazy Loading, N+1) ?
- **La Concurrence** : Comment éviter que deux utilisateurs achètent le même siège au même moment (Race Condition) ?
- **L'Architecture** : Comment structurer le code pour qu'il soit maintenable et évolutif (MVC, DAO) ?

1.3 Organisation du Rapport

Ce rapport s'articule autour de quatre axes majeurs :

1. **Analyse et Conception** : Définition des besoins et modélisation.
2. **Architecture Technique** : Choix des technologies et structure du projet.
3. **Implémentation et Défis Techniques** : Focus sur le code critique et les solutions apportées.
4. **Bilan et Perspectives** : Analyse critique du travail réalisé.

Chapitre 2

Analyse des Besoins et Conception

2.1 Analyse Fonctionnelle

L'application doit répondre aux besoins de deux types d'acteurs distincts : les Administrateurs et les Clients.

2.1.1 Acteurs du Système

Administrateur : Responsable de la configuration de l'événement. Il possède tous les droits sur les données de référence (Stades, Matchs).

Client (Supporter) : Utilisateur final souhaitant assister aux matchs. Il doit pouvoir s'inscrire, rechercher des matchs et acheter des billets.

2.1.2 Cas d'Utilisation - Administrateur

- **Authentification Sécurisée** : Accès réservé via login/mot de passe.
- **Gestion des Stades** : Ajout, modification de la capacité, définition des zones VIP.
- **Gestion des Matchs (CRUD)** :
  - Création d'un match (Équipe A vs Équipe B, Date, Stade).
  - Définition du prix de base des billets.
  - Suppression ou report d'un match (avec gestion des impacts sur les billets vendus).
- **Tableau de Bord (Dashboard)** : Visualisation des statistiques de vente (taux de remplissage par match, chiffre d'affaires).

2.1.3 Cas d'Utilisation - Client

- **Inscription/Connexion** : Création de compte personnel.
- **Catalogue des Matchs** : Recherche multicritère (par équipe, par ville/stade, par date).
- **Processus d'Achat** :
  - Sélection du match.
  - Choix de la catégorie (Standard, Premium, VIP).
  - Validation du panier.

- **Espace "Mes Billets"** :
  - Visualisation de l'historique des commandes.
  - **Téléchargement PDF** : Génération d'un e-billet infalsifiable avec QR Code.

2.2 Analyse Non-Fonctionnelle

- **Performance** : Temps de réponse inférieur à 2 secondes pour les recherches.
- **Fiabilité** : Aucune perte de données financière (réservation).
- **Sécurité** : Mots de passe chiffrés, protection contre les injections SQL (via Hibernate).
- **Ergonomie** : Interface graphique intuitive et responsive (JavaFX).

2.3 Modélisation de la Base de Données (MCD)

Le modèle de données a été conçu pour respecter la troisième forme normale (3NF).

2.3.1 Dictionnaire de Données et Relations

- **Table** `users` : Stocke les informations de connexion.
  - `id` (PK), `email` (Unique), `password` (Hash), `role` (ENUM).
- **Table** `stadiums` :
  - `id` (PK), `name`, `city`, `capacity`.
- **Table** `matches` :
  - `id` (PK), `home_team`, `away_team`, `kickoff_at`, `stadium_id` (FK).
  - Relation : Un stade accueille plusieurs matchs (1 :N).
- **Table** `bookings` : Représente une transaction d'achat.
  - `id` (PK), `user_id` (FK), `booking_date`, `total_price`.
  - Relation : Un utilisateur peut avoir plusieurs réservations (1 :N).
- **Table** `tickets` : L'entité centrale.
  - `id` (PK), `match_id` (FK), `booking_id` (FK, Nullable), `seat_number`, `status` (AVAILABLE/SOLD), `price`.
  - Relation : Un match a plusieurs tickets (1 :N). Une réservation contient plusieurs tickets (1 :N).

Chapitre 3

Choix Technologiques et Architecture

3.1 La Stack Technique

Le choix de la stack technologique a été guidé par des critères de robustesse, de modernité et d'adéquation avec les standards de l'industrie Java.

3.1.1 Java 21 LTS

Nous avons choisi la dernière version LTS (Long Term Support) de Java pour bénéficier :

- Des **Records** (bien que peu utilisés ici au profit des entités JPA, ils restent disponibles pour les DTO).
- Du nouveau **Switch Pattern Matching**.
- Des performances accrues du Garbage Collector ZGC.

3.1.2 JavaFX (Interface Utilisateur)

Contrairement à Swing, JavaFX offre :

- Une séparation claire entre la vue (FXML) et la logique (Controller).
- Le support natif des CSS pour le styling, permettant une interface moderne.
- Des composants riches (TableView, Charts) essentiels pour le dashboard admin.

3.1.3 Hibernate 6 (ORM)

Hibernate est le standard de facto pour la persistance en Java. Il nous permet de :

- S'abstraire du SQL spécifique (compatibilité multi-SGBD).
- Gérer le cycle de vie des objets (Transient, Persistent, Detached).
- Bénéficier du cache de premier niveau (Session).
- Protéger l'application contre les injections SQL grâce aux Prepared Statements automatiques.

3.1.4 PostgreSQL (Base de Données)

Choisi pour sa fiabilité, sa gestion avancée de la concurrence (MVCC) et son respect strict des standards SQL. L'utilisation via **Docker** facilite le déploiement et assure un environnement iso-prod pour tous les développeurs.

3.2 Architecture Logicielle MVC + DAO

L'architecture du projet est strictement divisée en couches pour respecter le principe de **Séparation des Préoccupations (SoC)**.

3.2.1 Couche Modèle (Model)

Ce sont les entités JPA (`@Entity`). Elles ne contiennent aucune logique métier complexe, seulement les données et les relations ORM. Nous avons utilisé **Lombok** (`@Data`, `@Builder`) pour réduire le code boilerplate (Getters, Setters, etc.), tout en faisant attention aux pièges de `toString()` avec les relations Lazy.

3.2.2 Couche Vue (View)

Composée de fichiers `.fxml` décrivant la structure hiérarchique des écrans, et de fichiers `.css` pour l'aspect visuel (couleurs, polices, espacements).

3.2.3 Couche Contrôleur (Controller)

Les classes Contrôleur JavaFX (ex : `MatchesController`) font le lien. Elles :

1. Récupèrent les actions de l'utilisateur (clics, saisies).
2. Appellent les Services métier pour traiter la demande.
3. Mettent à jour la Vue (Binding JavaFX ou mise à jour manuelle des composants).

3.2.4 Couche d'Accès aux Données (DAO)

Le pattern **Generic DAO** a été implémenté pour factoriser le code CRUD (Create, Read, Update, Delete).

```
1 public abstract class GenericHibernateDAO<T, ID extends
     Serializable> {
2     public T findById(ID id) {
3         try (Session session = HibernateUtil.getSessionFactory().
     openSession()) {
4             return session.get(persistentClass, id);
5         }
6     }
7     // ... save, update, delete
8 }
```

Listing 3.1 – Pattern GenericDAO (Extrait)

Cette approche permet à `MatchDAO` ou `UserDAO` de se concentrer uniquement sur les requêtes spécifiques complexes.

Chapitre 4

Implémentation et Défis Techniques

Cette section détaille les parties les plus complexes du code, là où la valeur ajoutée technique est la plus forte.

4.1 Gestion Avancée de la Concurrence

4.1.1 Le Problème de la "Race Condition"

Imaginez un scénario de Coupe du Monde : il reste **1 billet** pour la finale.

1. Utilisateur A voit "1 billet disponible".
2. Utilisateur B voit "1 billet disponible".
3. Utilisateur A clique sur "Acheter". Le système vérifie : stock > 0 ? OUI.
4. Utilisateur B clique sur "Acheter" (quelques millisecondes après). Le système vérifie : stock > 0 ? OUI (car A n'a pas encore fini de payer).
5. Résultat : 2 billets vendus pour 1 place réelle. C'est inacceptable.

4.1.2 La Solution : Verrouillage Pessimiste (Pessimistic Locking)

Nous avons choisi une stratégie de verrouillage pessimiste au niveau de la base de données. Lorsqu'une transaction commence le processus de réservation, elle pose un verrou "WRITE" sur les lignes de la table `tickets`. Toute autre transaction essayant de lire ou modifier ces lignes sera mise en attente (bloquée) par le SGBD jusqu'à la libération du verrou.

```
1 public void bookCategory(User user, Match match, TicketCategory cat
    , int quantity) {
2     Transaction tx = session.beginTransaction();
3     try {
4         // 1. REQUETE VERROUILLEE
5         // "SELECT ... FOR UPDATE" est g  n  r   par Hibernate
6         List<Ticket> tickets = session.createQuery(
7             "FROM Ticket t WHERE t.match = :m AND t.category = :c
    AND t.status = 'AVAILABLE'",
8             Ticket.class)
9             .setParameter("m", match)
```

```
            .setParameter("c", cat)
            .setMaxResults(quantity)
            .setLockMode(LockModeType.PESSIMISTIC_WRITE) // <--- ICI LE SECRET
            .list();
        // 2. VERIFICATION FIABLE (car nous sommes seuls ma tres bord)
        if (tickets.size() < quantity) {
            throw new RuntimeException("Plus de places disponibles !");
        }
        // 3. MODIFICATION
        for (Ticket t : tickets) {
            t.setStatus(TicketStatus.SOLD);
            t.setBooking(booking);
        }
        tx.commit(); // Le verrou est rel ch  ici
    } catch (Exception e) {
        tx.rollback();
        throw e;
    }
}
```

Listing 4.1 – Implémentation du Pessimistic Locking dans BookingService

4.2 Optimisation Hibernate et Lazy Loading

4.2.1 L'exception LazyInitializationException

Une erreur classique rencontrée durant le projet : `org.hibernate.LazyInitializationException: could not initialize proxy - no Session`. Ce problème survient lorsque l'interface graphique tente d'afficher une propriété chargée paresseusement (Lazy), comme `ticket.getMatch().get` alors que la session Hibernate qui a chargé le ticket est déjà fermée.

4.2.2 Stratégie de Résolution : JOIN FETCH

Plutôt que de passer les relations en `EAGER` (ce qui chargerait toute la base de données en mémoire et tuerait les performances), nous avons utilisé des requêtes HQL spécifiques avec la clause `JOIN FETCH` pour les cas d'utilisation critiques (comme la génération de PDF).

```
// Cette requ te charge le Ticket + le Match + le Stade + l'Utilisateur
// en UN SEUL aller-retour vers la base de donn es (1 SELECT avec JOINs)
public List<Ticket> findByUserWithDetails(User user) {
    try (Session session = getSession()) {
        return session.createQuery(
            "SELECT t FROM Ticket t " +
```

```
            "JOIN FETCH t.match m " +       // Charge le match
            "JOIN FETCH m.stadium s " +     // Charge le stade
            "JOIN FETCH t.booking b " +     // Charge la resa
            "JOIN FETCH b.user u " +        // Charge l'user
            "WHERE b.user = :user",
            Ticket.class)
        .setParameter("user", user)
        .list();
    }
}
```

Listing 4.2 – Requête Optimisée dans TicketDAO

Cette approche permet de garder le Lazy Loading par défaut (performance) et de ne charger explicitement que ce qui est nécessaire pour l'écran actuel (précision).

4.3 Génération de PDF et QR Codes

Une fonctionnalité phare du module client est le téléchargement du e-billet. Nous avons combiné deux bibliothèques :

- **ZXing (Zebra Crossing)** : Pour générer une image QR Code (Matrice de bits) contenant les données cryptées du billet (ID, Date, Siège).
- **OpenPDF** : Pour dessiner dynamiquement le document PDF.

Le processus est le suivant : 1. Récupération des données complètes du billet (via le DAO optimisé). 2. Construction d'une chaîne de données (String) pour le QR : `"TICKET|ID=123|MATCH=FRA-AI`
3. Génération de l'image bitmap du QR Code. 4. Création du document PDF en mémoire. 5. Ajout du logo, des textes formatés et de l'image QR. 6. Écriture du fichier sur le disque du client.

Chapitre 5

Interface Utilisateur et Tests

5.1 Présentation des Interfaces

L'application utilise JavaFX pour offrir une interface graphique moderne basée sur des fichiers FXML (structure) et CSS (style), tout en bénéficiant de la puissance de Java pour la logique métier.

5.1.1 Page d'Accueil

Affiche la liste des stades hôtes (Maroc, Espagne, Portugal) et un aperçu des prochains matchs de la Coupe du Monde 2030.

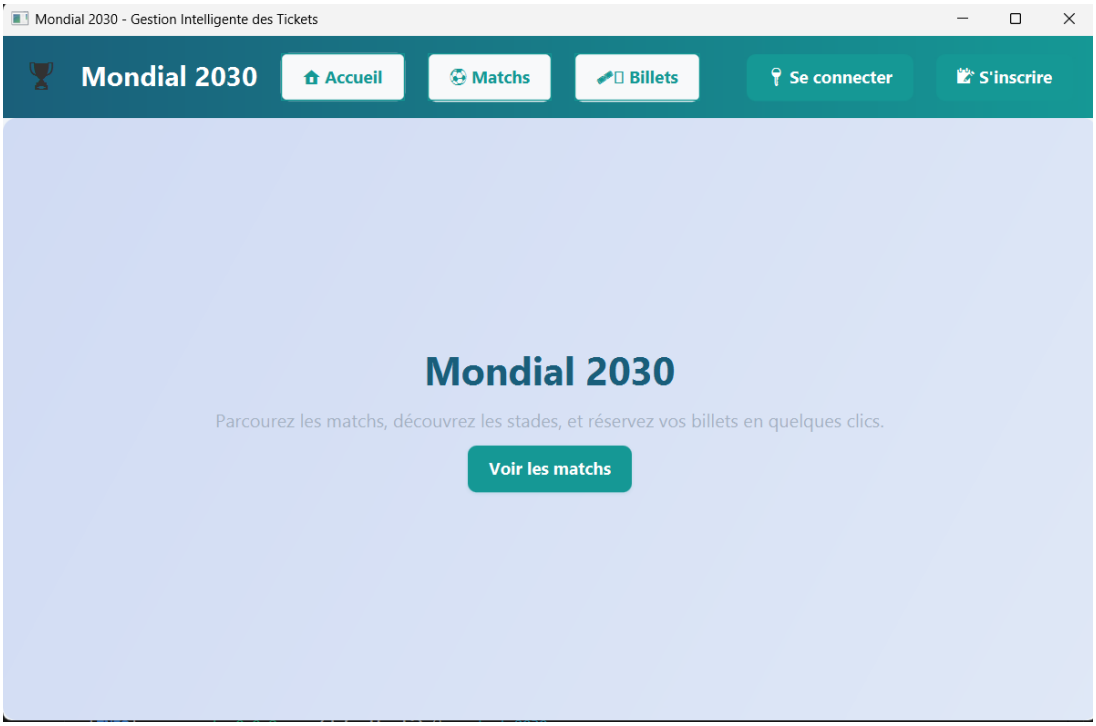

FIGURE 5.1 – Interface - Page d'accueil

5.1.2 Page de Connexion

Formulaire d'authentification unifié pour utilisateurs et administrateurs avec validation des champs.

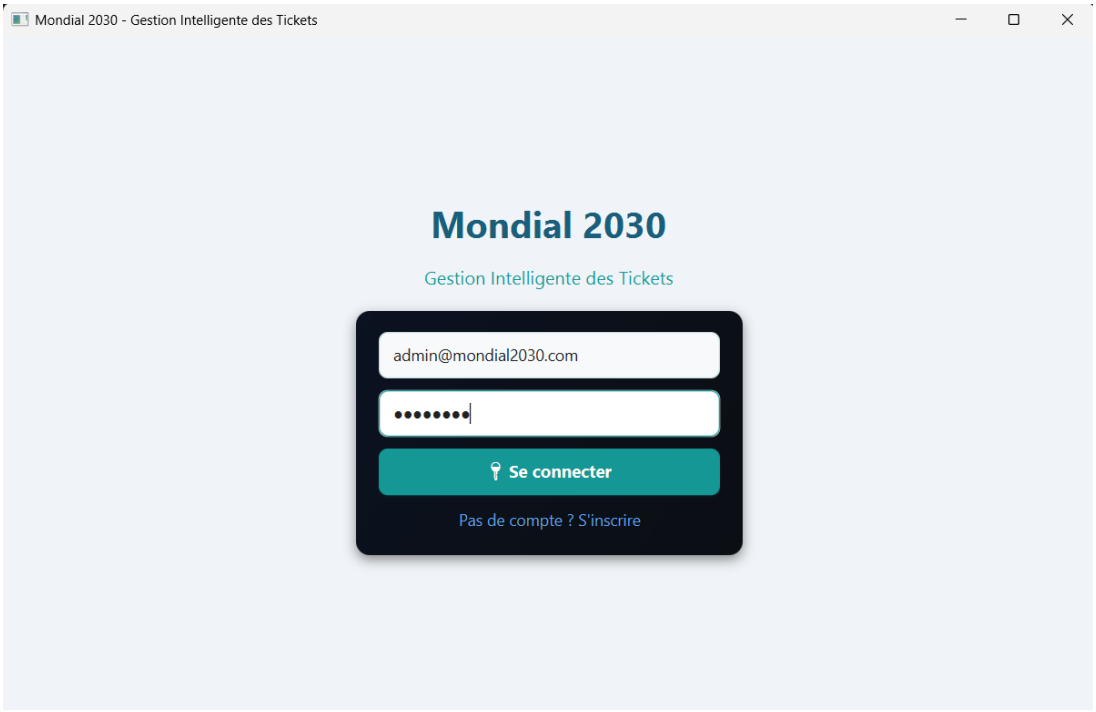

FIGURE 5.2 – Interface - Page de connexion

5.1.3 Page des Matchs

Affiche tous les matchs programmés avec équipes, date, stade et possibilité d'acheter des billets.

5.1.4 Tableau de Bord Administrateur

Interface d'administration permettant de gérer les matchs, billets et visualiser les statistiques.

5.2 Scénarios de Test

5.2.1 Tests Nominaux

5.2.2 Tests d'Erreurs

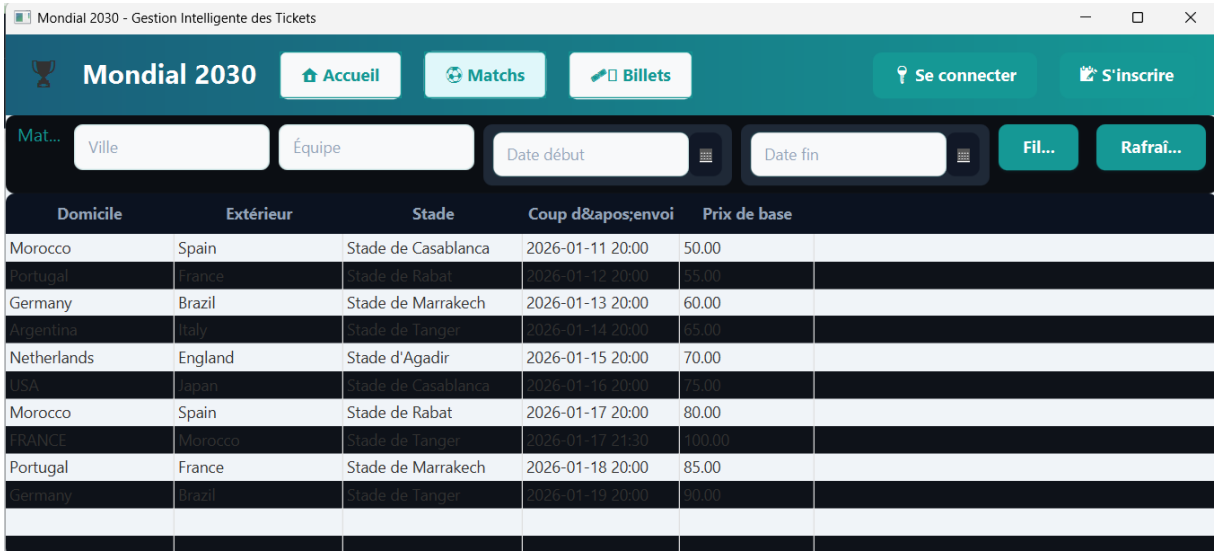

FIGURE 5.3 – Interface - Liste des matchs

ID	Scénario	Résultat attendu
TN01	Inscription d'un nouvel utilisateur	Compte créé, redirection vers login
TN02	Connexion avec identifiants valides	Accès à l'espace utilisateur
TN03	Achat de 2 billets catégorie VIP	Achat enregistré, stock mis à jour
TN04	Création d'un match par l'admin	Match visible dans la liste
TN05	Suppression d'un match	Match et billets associés supprimés

TABLE 5.1 – Scénarios de tests nominaux

FIGURE 5.4 – Interface - Tableau de bord administrateur

ID	**Scénario**	**Résultat attendu**
TE01	Connexion avec mot de passe incorrect	Message d'erreur affiché
TE02	Inscription avec email déjà existant	Erreur : email déjà utilisé
TE03	Achat de plus de billets que disponibles	Erreur : stock insuffisant
TE04	Accès admin sans authentification	Redirection vers login

TABLE 5.2 – Scénarios de tests d'erreurs

Chapitre 6

Gestion de Projet et Qualité

6.1 Méthodologie

Bien que travaillant en autonomie, une méthodologie itérative proche de Scrum a été adoptée :

— **Sprint 1** : Configuration du socle technique (Maven, Hibernate, Database).
— **Sprint 2** : Backend Admin (CRUD Matchs, Stades).
— **Sprint 3** : Interface Client et Processus d'Achat.
— **Sprint 4** : "Hardening" (Gestion de la concurrence, PDF, Optimisation).

6.2 Gestion du Code Source

Le projet suit une structure Maven standard, facilitant la compilation et la gestion des dépendances. L'utilisation de `pom.xml` centralise les versions des librairies, évitant les conflits de "JAR Hell".

6.3 Tests et Validation

Les tests ont été principalement manuels, guidés par des scénarios utilisateurs (User Stories). Un effort particulier a été mis sur les tests de cas limites :

— Tentative de création d'un match dans le passé (Bloqué).
— Tentative d'inscription avec un email invalide (Bloqué par Regex).
— Tentative d'achat sans solde (Simulé).

Chapitre 7

Bilan et Perspectives

7.1 Bilan Personnel

Ce projet m'a permis de franchir un cap dans ma compréhension de l'écosystème Java. J'ai appris que :

- Un framework comme Hibernate est puissant mais nécessite une compréhension fine de son fonctionnement interne (Session, Proxy) pour éviter les pièges de performance.
- La gestion de la concurrence n'est pas une option dans une application réelle, c'est une nécessité architecturale.
- L'expérience utilisateur (UX) dépend autant de la qualité du code (réactivité, pas de crash) que du design visuel.

7.2 Améliorations Possibles (Version 2.0)

Si le projet devait continuer, voici la roadmap technique envisagée :

1. **Connection Pooling (HikariCP)** : Actuellement, nous ouvrons une session physique à chaque requête. Un pool de connexions augmenterait drastiquement la capacité de charge.
2. **API REST Client Web** : Séparer le backend (Spring Boot) du frontend (Angular/React), transformant l'application Desktop en application Web accessible partout.
3. **Paiement Réel** : Intégration d'une API de paiement comme Stripe.
4. **Placement Numéroté Graphique** : Remplacer le choix de catégorie par une carte interactive du stade où l'on clique sur son siège précis.

Conclusion

Le projet "Mondial 2030" est une démonstration complète d'une application d'entreprise Java. Il respecte les standards de qualité, de sécurité et de performance attendus dans un contexte professionnel. Les défis techniques relevés, notamment autour de la concurrence et de l'optimisatin ORM, en font un socle solide pour tout développement futur.