EMSI - École Marocaine des Sciences de l'Ingénieur

Filière : 4IIR - Ingénierie Informatique et Réseaux

Module : Java Avancé / Programmation Orientée Objet

Système de Gestion des Billets Coupe du Monde FIFA 2026

Application Desktop JavaFX avec Hibernate ORM

Réalisé par :

Chouaib IKTACHE

Encadré par :

Pr. Abderrahim LARHLIMI

Année Universitaire : 2025-2026

Remerciements

Je tiens à exprimer ma sincère gratitude à toutes les personnes qui ont contribué à la réalisation de ce projet.

Mes remerciements s'adressent tout d'abord à **Pr. Abderrahim LARHLIMI**, mon encadrant, pour ses conseils précieux, sa disponibilité et son accompagnement tout au long de ce projet.

Je remercie également l'administration de l'**EMSI** pour la qualité de la formation dispensée et les moyens mis à notre disposition.

Enfin, je remercie ma famille et mes amis pour leur soutien continu et leurs encouragements.

Chouaib IKTACHE

Table des matières

Chapitre 1

Introduction Générale

1.1 Contexte du Projet

La Coupe du Monde FIFA 2026, organisée conjointement par les **États-Unis**, le **Mexique** et le **Canada**, représente un événement sportif majeur nécessitant une gestion efficace des billets pour les millions de spectateurs attendus.

Dans ce contexte, la gestion manuelle des billets (vente papier, réservations téléphoniques) présente de nombreuses limitations : lenteur, risques d'erreurs, difficulté de suivi, et absence de statistiques en temps réel.

Ce projet a été réalisé dans le cadre du module **Java Avancé** de la filière 4IIR à l'EMSI, afin de mettre en pratique les concepts de programmation orientée objet, les design patterns, et les frameworks modernes.

1.2 Problématique

Comment développer une application robuste et ergonomique permettant :

— La gestion centralisée des stades, matchs et billets ?

— La distinction claire entre administrateurs et clients ?

— Le suivi en temps réel des ventes et réservations ?

— Une expérience utilisateur fluide et moderne ?

1.3 Objectifs du Projet

1. Développer une application desktop moderne avec **JavaFX**
2. Implémenter une architecture en couches (**DAO-Service-View**)
3. Utiliser **Hibernate ORM** pour la persistance des données
4. Appliquer les **design patterns** (DAO, Singleton)
5. Exploiter les **Collections Java** et l'**API Stream**
6. Offrir une interface utilisateur intuitive et responsive

Chapitre 2

Analyse et Conception

2.1 Spécification des Besoins

2.1.1 Besoins Fonctionnels

ID	Fonctionnalité	Acteur
BF01	S'authentifier (login/logout)	Admin, Client
BF02	S'inscrire (créer un compte)	Client
BF03	Gérer les stades (CRUD)	Admin
BF04	Gérer les matchs (CRUD)	Admin
BF05	Générer des billets pour un match	Admin
BF06	Consulter les statistiques	Admin
BF07	Gérer les utilisateurs	Admin
BF08	Consulter les matchs disponibles	Client
BF09	Rechercher un match par équipe	Client
BF10	Acheter un billet	Client
BF11	Réserver un billet	Client
BF12	Consulter ses billets/réservations	Client
BF13	Modifier son profil	Client
BF14	Exporter les données en CSV	Admin

TABLE 2.1 – Besoins fonctionnels

2.1.2 Besoins Non-Fonctionnels

— **Performance** : Temps de réponse < 1 seconde pour les opérations courantes
— **Ergonomie** : Interface intuitive, navigation fluide
— **Sécurité** : Mots de passe stockés (à améliorer avec hachage)
— **Portabilité** : Application multiplateforme (Windows, macOS, Linux)
— **Maintenabilité** : Code structuré, architecture en couches

2.2 Conception UML

2.2.1 Diagramme de Cas d'Utilisation

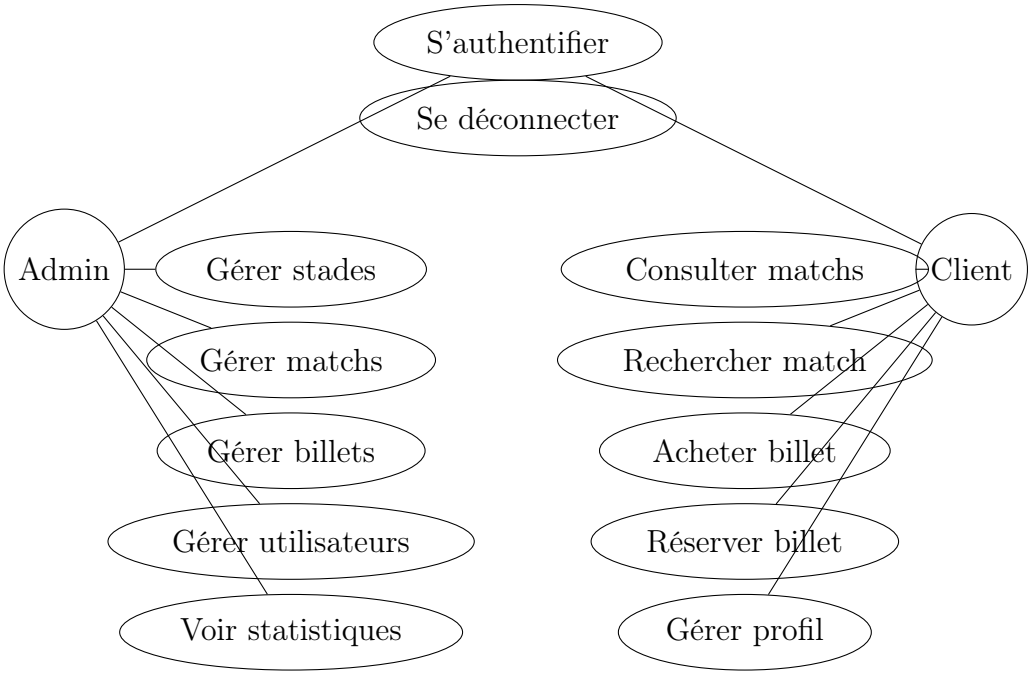

FIGURE 2.1 – Diagramme de cas d'utilisation

2.2.2 Diagramme de Classes

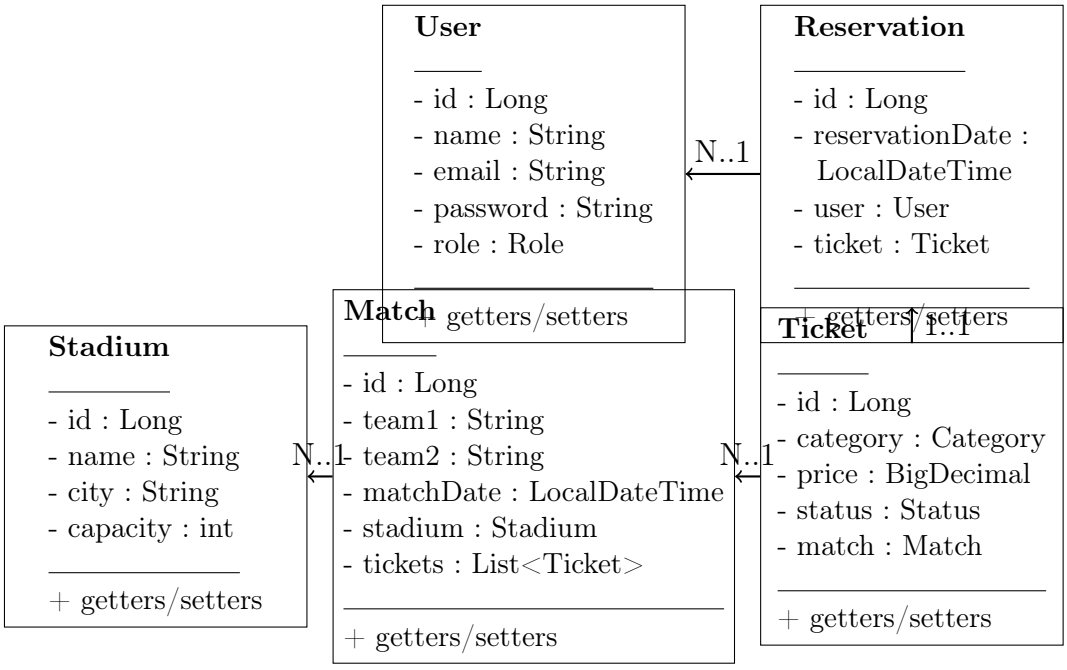

FIGURE 2.2 – Diagramme de classes simplifié

2.2.3 Énumérations

```java
// Dans User.java
public enum Role {
    ADMIN, CLIENT
}

// Dans Ticket.java
public enum Category {
    VIP, STANDARD, ECONOMY
}

public enum Status {
    AVAILABLE, RESERVED, SOLD
}
```

Listing 2.1 – Énumérations du modèle

2.3 Conception de la Base de Données

2.3.1 Modèle Logique de Données (MLD)

```
users (id PK, name, email UNIQUE, password, role)
stadiums (id PK, name UNIQUE, city, capacity)
matches (id PK, team1, team2, match_date, stadium_id FK)
tickets (id PK, category, price, status, match_id FK)
reservations (id PK, reservation_date, user_id FK, ticket_id FK)
```

2.3.2 Dictionnaire de Données

Table	Champ	Type	Taille	Contrainte
users	id	INTEGER	-	PK, AUTO_INCREMENT
users	name	VARCHAR	100	NOT NULL
users	email	VARCHAR	100	UNIQUE, NOT NULL
users	password	VARCHAR	255	NOT NULL
users	role	VARCHAR	20	NOT NULL
stadiums	id	INTEGER	-	PK, AUTO_INCREMENT
stadiums	name	VARCHAR	100	UNIQUE, NOT NULL
stadiums	city	VARCHAR	100	NOT NULL
stadiums	capacity	INTEGER	-	NOT NULL, > 0
tickets	id	INTEGER	-	PK, AUTO_INCREMENT
tickets	category	VARCHAR	20	NOT NULL
tickets	price	DECIMAL	10,2	NOT NULL, $>= 0$
tickets	status	VARCHAR	20	NOT NULL
tickets	match_id	INTEGER	-	FK

TABLE 2.2 – Dictionnaire de données (extrait)

Chapitre 3

Environnement Technique

3.1 Langage et Version

Technologie	Version	Justification
Java	JDK 17 (LTS)	Version stable avec support long terme

Table 3.1 – Langage de programmation

3.2 Frameworks et Bibliothèques

Framework	Version	Justification
JavaFX	21.0.1	Interface graphique moderne, stylisable avec CSS, composants riches (TableView, Charts)
Hibernate ORM	6.4.4	Mapping objet-relationnel automatique, gestion des transactions, indépendance BDD
Hibernate Validator	8.0.1	Validation des entités avec annotations (@NotBlank, @Email, @Min)
SQLite JDBC	3.45.1	Base de données embarquée, déploiement simplifié
Apache Commons CSV	1.10.0	Export des données au format CSV
JUnit Jupiter	5.10.0	Tests unitaires

Table 3.2 – Frameworks et bibliothèques

3.3 Outils de Développement

Outil	Version	Utilisation
Maven	3.9+	**Automatisation** de la gestion des dépendances et du build
VS Code	Latest	IDE de développement avec extensions Java
Git	2.x	Contrôle de version
SQLite	3.x	Base de données embarquée légère

TABLE 3.3 – Outils de développement

3.4 Extrait du pom.xml

```xml
<dependencies>
    <!-- JavaFX Controls -->
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>21.0.1</version>
    </dependency>

    <!-- Hibernate ORM -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.4.Final</version>
    </dependency>

    <!-- SQLite JDBC Driver -->
    <dependency>
        <groupId>org.xerial</groupId>
        <artifactId>sqlite-jdbc</artifactId>
        <version>3.45.1.0</version>
    </dependency>

    <!-- Hibernate SQLite Dialect -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-community-dialects</artifactId>
        <version>6.4.4.Final</version>
    </dependency>
</dependencies>
```

Listing 3.1 – Dépendances Maven clés (pom.xml)

Chapitre 4

Architecture et Implémentation

4.1 Architecture Logicielle

L'application suit une **architecture en couches** (Layered Architecture) :

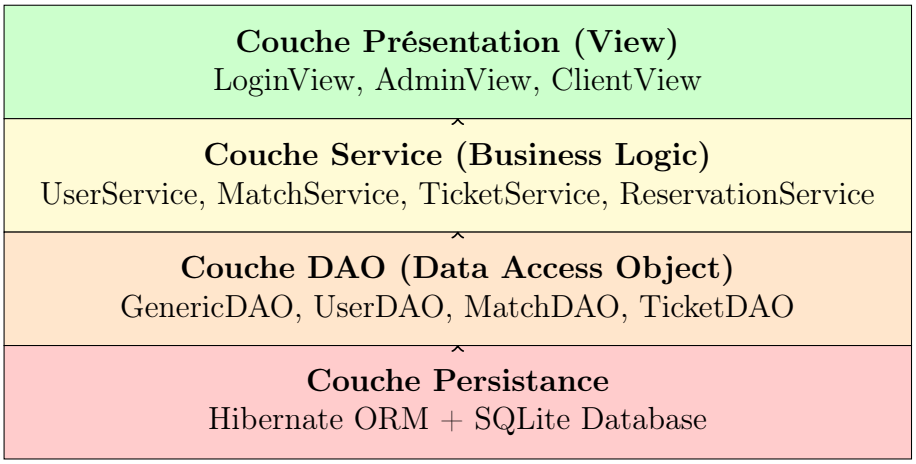

FIGURE 4.1 – Architecture en couches

4.1.1 Organisation des Packages

```
com.example/
|-- MainApp.java          # Point d'entree JavaFX
|-- Launcher.java         # Lanceur (contourne module JavaFX)
|-- entity/               # Entites JPA (POJO)
|   |-- User.java
|   |-- Stadium.java
|   |-- Match.java
|   |-- Ticket.java
|   |-- Reservation.java
|-- dao/                  # Data Access Objects
|   |-- GenericDAO.java       # Interface generique
|   |-- GenericDAOImpl.java   # Implementation generique
|   |-- UserDAO.java, UserDAOImpl.java
|   |-- MatchDAO.java, MatchDAOImpl.java
|   |-- TicketDAO.java, TicketDAOImpl.java
|   |-- ...
|-- service/              # Logique metier
```

```
|   |-- UserService.java
|   |-- MatchService.java
|   |-- TicketService.java
|   |-- ReservationService.java
|-- view/                  # Interfaces graphiques
|   |-- LoginView.java
|   |-- AdminView.java
|   |-- ClientView.java
|   |-- RegisterView.java
|-- util/                  # Utilitaires
    |-- HibernateUtil.java     # Singleton SessionFactory
    |-- CsvExporter.java       # Export CSV
```

Listing 4.1 – Structure des packages

4.2 Design Patterns Utilisés

4.2.1 Pattern Singleton - HibernateUtil

Le pattern **Singleton** garantit une instance unique de la SessionFactory Hibernate :

```java
package com.example.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    // Instance unique (Singleton)
    private static final SessionFactory sessionFactory =
   buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Cree la SessionFactory a partir de hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory()
   ;
        } catch (Exception ex) {
            System.err.println("Creation SessionFactory echouee: " + ex)
   ;
            throw new ExceptionInInitializerError(ex);
        }
    }

    // Point d'acces global
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

Listing 4.2 – HibernateUtil.java (Lignes 1-28)

Justification : La connexion à la base de données est une ressource coûteuse. Le Singleton évite la création multiple de SessionFactory et centralise la gestion.

4.2.2 Pattern DAO (Data Access Object)

Le pattern **DAO** isole le code d'accès aux données du reste de l'application :

```
1  package com.example.dao;
2
3  import java.util.List;
4  import java.util.Optional;
5
6  public interface GenericDAO<T, ID> {
7      void save(T entity);
8      void update(T entity);
9      void delete(T entity);
10     Optional<T> findById(ID id);
11     List<T> findAll();
12 }
```

Listing 4.3 – GenericDAO.java - Interface générique (Lignes 1-12)

```
1  @Override
2  public void save(T entity) {
3      Transaction transaction = null;
4      try (Session session = HibernateUtil.getSessionFactory().openSession
       ()) {
5          transaction = session.beginTransaction();
6          session.persist(entity);      // Ligne 22 : Persistance
7          transaction.commit();          // Ligne 23 : Commit
8      } catch (Exception e) {
9          if (transaction != null) transaction.rollback();  // Rollback si
       erreur
10         throw e;
11     }
12 }
13
14 @Override
15 public void update(T entity) {
16     Transaction transaction = null;
17     try (Session session = HibernateUtil.getSessionFactory().openSession
       ()) {
18         transaction = session.beginTransaction();
19         session.merge(entity);        // Ligne 35 : Mise a jour
20         transaction.commit();
21     } catch (Exception e) {
22         if (transaction != null) transaction.rollback();
23         throw e;
24     }
25 }
```

Listing 4.4 – GenericDAOImpl.java - Implémentation (Lignes 17-40)

Justification : Le DAO permet de changer la technologie de persistance (ex : passer de SQLite à MySQL) sans modifier la couche Service.

4.3 Extraits de Code Clés

4.3.1 Utilisation des Streams Java (API Fonctionnelle)

Les **Streams**, introduits en Java 8, permettent de traiter les collections de manière déclarative.

Exemple 1 : Filtrage avec filter()

Fichier : `TicketService.java` — **Lignes 58-61**

```
public List<Ticket> getSoldTickets() {
    return ticketDAO.findAll().stream()          // Ligne 59: Conversion en Stream
            .filter(t -> t.getStatus() == Ticket.Status.SOLD)  // Ligne 60: Filtre
            .collect(Collectors.toList());       // Ligne 61: Reconversion en List
}
```

Listing 4.5 – Filtrage des billets vendus

Explication :

- `stream()` : Convertit la List en Stream pour traitement fonctionnel
- `filter(t -> ...)` : Expression lambda qui garde uniquement les billets avec statut SOLD
- `collect(Collectors.toList())` : Opération terminale qui reconvertit en List

Exemple 2 : Transformation avec map() et réduction avec reduce()

Fichier : `TicketService.java` — **Lignes 68-71**

```
public BigDecimal getTotalRevenue() {
    return getSoldTickets().stream()             // Ligne 69: Stream des billets vendus
            .map(Ticket::getPrice)               // Ligne 70: Extraction du prix (reference de methode)
            .reduce(BigDecimal.ZERO, BigDecimal::add);  // Ligne 71: Somme
}
```

Listing 4.6 – Calcul du revenu total

Explication :

- `map(Ticket::getPrice)` : Transforme chaque Ticket en son prix (BigDecimal)
- `Ticket::getPrice` : Référence de méthode, équivalent à `t -> t.getPrice()`
- `reduce(BigDecimal.ZERO, BigDecimal::add)` : Agrège tous les prix en partant de 0

Exemple 3 : Chaînage complet

Fichier : `AdminView.java` — **Lignes 151-155**

```java
double totalRevenue = ticketService.getAllTickets().stream()  // Ligne 151
        .filter(t -> t.getStatus() == Ticket.Status.SOLD)     // Ligne 152: Filtre
        .map(Ticket::getPrice)                                 // Ligne 153: Extraction
        .mapToDouble(BigDecimal::doubleValue)                  // Ligne 154: Conversion
        .sum();                                                // Ligne 155: Somme
```

Listing 4.7 – Pipeline Stream complet

Exemple 4 : Recherche multicritère

Fichier : `ClientView.java` — **Lignes 223-228**

```java
List<Match> filtered = matches.stream()
    .filter(m -> m.getTeam1().toLowerCase().contains(searchText) ||
                 m.getTeam2().toLowerCase().contains(searchText) ||
                 m.getStadium().getName().toLowerCase().contains(searchText))
    .toList();  // Methode Java 16+ equivalente a collect(Collectors.toList())
```

Listing 4.8 – Filtrage par recherche texte

4.3.2 Tableau Récapitulatif des Streams

Fichier	Méthode	Lignes	Opérations
TicketService.java	getSoldTickets()	58-61	stream → filter → collect
TicketService.java	getAvailableTickets()	63-66	stream → filter → collect
TicketService.java	getTotalRevenue()	68-71	stream → map → reduce
TicketService.java	getTicketsByUser()	73-76	stream → filter → collect
AdminView.java	showDashboard()	145-155	stream → filter → count/map/sum
ClientView.java	filterMatches()	223-228	stream → filter → toList

TABLE 4.1 – Utilisation des Streams dans le projet

4.3.3 Gestion des Transactions (try-with-resources)

Fichier : `GenericDAOImpl.java` — **Lignes 17-27**

```java
@Override
public void save(T entity) {
    Transaction transaction = null;
    // try-with-resources : fermeture automatique de la session
    try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        transaction = session.beginTransaction();  // Debut transaction
        session.persist(entity);                   // Operation
        transaction.commit();                      // Validation
    } catch (Exception e) {
```

```
10          if (transaction != null) transaction.rollback();  // Annulation
    si erreur
11          throw e;  // Re-lancer l'exception
12      }
13  }
```

Listing 4.9 – Gestion des transactions avec try-with-resources

Points importants :

— `try-with-resources` (Java 7+) : Ferme automatiquement la Session

— `transaction.rollback()` : Annule les modifications en cas d'erreur

— Pattern robuste pour la gestion des ressources

Chapitre 5

Interface Utilisateur et Tests

5.1 Présentation des Interfaces

5.1.1 Page de Connexion (LoginView)

— Champs : Email, Mot de passe
— Bouton "Sign In"
— Lien vers inscription
— Affichage des identifiants de démonstration
— Design moderne avec dégradé de couleurs

5.1.2 Dashboard Administrateur (AdminView)

— **Navbar** : Logo, nom utilisateur, bouton déconnexion
— **Sidebar** : Navigation (Dashboard, Users, Stadiums, Matches, Tickets, Stats)
— **Cards statistiques** : Total users, stades, matchs, billets vendus, revenus
— **Tableaux** : TableView JavaFX avec données interactives
— **Actions** : Boutons CRUD (Add, Edit, Delete)

5.1.3 Interface Client (ClientView)

— **Browse Matches** : Liste des matchs avec recherche
— **My Tickets** : Billets achetés
— **My Reservations** : Réservations en cours
— **Profile** : Modification des informations personnelles

5.2 Composants JavaFX Utilisés

Composant	Utilisation
BorderPane	Layout principal (top, left, center)
VBox / HBox	Organisation verticale / horizontale
TableView<T>	Affichage des données en tableau
TableColumn<T,S>	Colonnes du tableau
Button	Actions utilisateur
TextField / PasswordField	Saisie de texte
ComboBox<T>	Liste déroulante
DatePicker	Sélection de date
Dialog / Alert	Popups et confirmations
ScrollPane	Zone de défilement

Table 5.1 – Composants JavaFX

5.3 Scénarios de Test

5.3.1 Tests Nominaux (Cas positifs)

ID	Scénario	Résultat attendu	Statut
T01	Connexion avec admin@worldcup.com / admin123	Redirection vers AdminView	✓
T02	Connexion avec client@test.com / client123	Redirection vers ClientView	✓
T03	Ajouter un nouveau stade	Stade visible dans la liste	✓
T04	Créer un match avec billets	Match et billets créés	✓
T05	Client achète un billet	Statut passe à SOLD	✓
T06	Client réserve un billet	Statut passe à RESERVED	✓
T07	Recherche match par équipe	Matchs filtrés correctement	✓

Table 5.2 – Tests nominaux

5.3.2 Tests d'Erreurs (Cas limites)

ID	Scénario	Résultat attendu	Statut
E01	Connexion avec mauvais mot de passe	Message "Invalid credentials"	✓
E02	Inscription avec email déjà existant	Message d'erreur	✓
E03	Acheter un billet déjà vendu	Exception "Ticket not available"	✓
E04	Champs obligatoires vides	Alert de validation	✓
E05	Capacité stade négative	Validation @Min(1) bloque	✓

Table 5.3 – Tests d'erreurs

Chapitre 6

Conclusion et Perspectives

6.1 Bilan Technique

Le cahier des charges initial a été **entièrement respecté** :

✓ Application desktop fonctionnelle avec JavaFX
✓ Architecture en couches (DAO, Service, View)
✓ Persistance avec Hibernate ORM et SQLite
✓ Design patterns implémentés (Singleton, DAO)
✓ Collections Java et API Stream exploitées
✓ Interface utilisateur moderne et ergonomique
✓ Gestion complète du cycle de vie des billets
✓ Séparation Admin / Client

6.2 Compétences Acquises

— Maîtrise de **JavaFX** pour le développement desktop
— Compréhension approfondie de **Hibernate ORM**
— Application des **design patterns** (Singleton, DAO, Factory)
— Maîtrise des **Streams Java** et programmation fonctionnelle
— Gestion de projet avec **Maven**
— Structuration d'un projet en couches

6.3 Difficultés Rencontrées

— **Configuration Hibernate/SQLite** : Nécessité d'utiliser le dialect communautaire
— **JavaFX Modules** : Résolu avec le Launcher séparé
— **Relations JPA** : Gestion du FetchType EAGER/LAZY

6.4 Perspectives et Améliorations

Avec plus de temps, les améliorations suivantes pourraient être apportées :

— **Sécurité** : Hachage des mots de passe (BCrypt)
— **Multithreading** : Tasks JavaFX pour opérations longues
— **PDF** : Génération de billets imprimables
— **Paiement** : Intégration Stripe/PayPal
— **Notifications** : Emails de confirmation
— **Version Web** : Spring Boot + Angular/React
— **Application Mobile** : Version Android/iOS

Webographie

1. **Documentation Oracle Java 17**
   https://docs.oracle.com/en/java/javase/17/
2. **Documentation JavaFX**
   https://openjfx.io/javadoc/21/
3. **Hibernate ORM Documentation**
   https://hibernate.org/orm/documentation/6.4/
4. **SQLite Documentation**
   https://www.sqlite.org/docs.html
5. **Maven Central Repository**
   https://mvnrepository.com/
6. **Baeldung - Java Tutorials**
   https://www.baeldung.com/
7. **Stack Overflow**
   https://stackoverflow.com/
8. **GitHub - Projets JavaFX**
   https://github.com/topics/javafx

Annexe A

Annexe : Configuration Hibernate

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Connexion SQLite -->
        <property name="hibernate.connection.driver_class">
            org.sqlite.JDBC
        </property>
        <property name="hibernate.connection.url">
            jdbc:sqlite:worldcup_tickets.db
        </property>

        <!-- Dialect SQLite -->
        <property name="hibernate.dialect">
            org.hibernate.community.dialect.SQLiteDialect
        </property>

        <!-- Pool de connexions -->
        <property name="hibernate.connection.pool_size">1</property>

        <!-- Gestion du schema -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Affichage SQL -->
        <property name="hibernate.show_sql">true</property>

        <!-- Mapping des entites -->
        <mapping class="com.example.entity.User"/>
        <mapping class="com.example.entity.Stadium"/>
        <mapping class="com.example.entity.Match"/>
        <mapping class="com.example.entity.Ticket"/>
        <mapping class="com.example.entity.Reservation"/>
    </session-factory>
</hibernate-configuration>
```

Listing A.1 – hibernate.cfg.xml complet

Annexe B

Annexe : Données Initiales

B.1 Stades de la Coupe du Monde 2026

Stade	**Ville, Pays**	**Capacité**
MetLife Stadium	New Jersey, USA	82 500
AT&T Stadium	Dallas, USA	80 000
SoFi Stadium	Los Angeles, USA	70 000
Estadio Azteca	Mexico City, Mexico	87 000
Hard Rock Stadium	Miami, USA	65 000
Lumen Field	Seattle, USA	69 000
Gillette Stadium	Boston, USA	65 000
BMO Field	Toronto, Canada	45 000
BC Place	Vancouver, Canada	54 000
Estadio BBVA	Monterrey, Mexico	53 500

Table B.1 – Stades officiels

B.2 Tarification des Billets

Phase	**VIP ($)**	**Standard ($)**	**Economy ($)**
Phase de groupes	800	350	150
Huitièmes de finale	1 000	500	200
Quarts de finale	1 500	750	300
Demi-finales	2 000	1 000	400
Finale	3 000	1 500	600

Table B.2 – Grille tarifaire