ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Rapport de Projet Java Avancé

Application de Gestion d'Hôtel

Module : Java Avancé / Programmation Orientée Objet
Filière : 4IIR
Année Universitaire : 2025 – 2026

Réalisé par :
Adam Labrahmi

Encadré par :
Mr. Abderrahim Larhlimi

Remerciements

Nous tenons à exprimer nos sincères remerciements à notre encadrant pédagogique, **M. Abderrahim Larhlimi**, pour son accompagnement, ses conseils avisés et son suivi tout au long de la réalisation de ce projet. Ses orientations et ses remarques pertinentes ont grandement contribué à la réussite de ce travail.

Nous remercions également l'ensemble du corps professoral ainsi que l'administration de l'**EMSI** pour la qualité de la formation dispensée et pour les moyens pédagogiques mis à notre disposition.

Enfin, nous adressons nos remerciements à toutes les personnes ayant contribué, de près ou de loin, à l'aboutissement de ce projet.

Table des matières

Table des figures

Chapitre 1

Introduction Générale

1.1 Contexte du projet

Ce projet s'inscrit dans le cadre de l'enseignement pratique du module **Java Avancé** dispensé au sein de l'École Marocaine des Sciences de l'Ingénieur (**EMSI**), en quatrième année du cycle ingénieur. L'objectif pédagogique est de concevoir une application de gestion robuste mettant en œuvre des concepts techniques complexes tels que la programmation orientée objet avancée, la manipulation des flux (*Streams*), la gestion de la concurrence (*Multithreading*) et la persistance des données.

1.2 Problématique

Dans le secteur de l'hôtellerie, la gestion traditionnelle des opérations quotidiennes (disponibilité des chambres, réservations, suivi des paiements) repose encore trop souvent sur des méthodes manuelles ou des outils peu intégrés. Ce mode de fonctionnement expose l'établissement à plusieurs problématiques majeures :

- **Incohérence des données :** Risques élevés de sur-réservation (*overbooking*) ou d'erreurs dans le calcul des factures.
- **Manque de réactivité :** Lenteur de l'interface lors du traitement des données volumineuses, impactant la qualité de service.
- **Sécurité et Persistance :** Difficulté à centraliser et à sécuriser les informations sensibles des clients et des transactions financières.

L'enjeu est donc de proposer une solution logicielle performante, capable d'automatiser ces processus tout en offrant une expérience utilisateur fluide.

1.3 Objectifs

L'objectif principal de ce travail est de concevoir et réaliser l'application **MangmentHotel**. Pour répondre aux défis identifiés, les objectifs spécifiques suivants ont été définis :

- **Automatisation des processus :** Développer un moteur de facturation automatique capable de calculer les montants en temps réel selon la durée du séjour.
- **Optimisation des performances :** Implémenter le *multithreading* via les *Tasks* JavaFX pour garantir que l'interface reste réactive lors des accès à la base de données.
- **Architecture Modulaire :** Adopter le design pattern **MVC** (Modèle-Vue-Contrôleur) et une architecture en couches pour assurer la maintenabilité et l'évolutivité du code.
- **Persistance fiable :** Mettre en place une couche DAO (*Data Access Object*) utilisant JDBC et Hibernate pour une interaction sécurisée avec MySQL.
- **Standardisation du déploiement :** Utiliser la technologie Docker pour assurer une portabilité totale de la solution, quel que soit l'environnement système.

Chapitre 2

Analyse et Conception

2.1 Spécification des besoins

Cette section détaille les attentes du système, tant sur le plan métier (fonctions) que sur le plan qualitatif (performance et sécurité).

2.1.1 Besoins fonctionnels

Les besoins fonctionnels décrivent les actions que les utilisateurs (Administrateurs ou Clients) peuvent effectuer sur le système.

Gestion des Chambres

- **Gestion CRUD :** Ajouter, modifier (numéro, type, prix) et supprimer une chambre.
- **Disponibilité :** Suivi en temps réel du statut de la chambre (Libre, Occupée, Nettoyage).
- **Visualisation :** Affichage dynamique de la liste des chambres avec recherche multicritères.

Gestion des Réservations et Clients

- **Suivi des Clients :** Enregistrement et gestion des informations personnelles des clients.
- **Calendrier des Réservations :** Création de réservations avec distinction automatique des tarifs selon la durée et le type de chambre.
- **Validation métier :** Empêcher la réservation d'une chambre déjà occupée.

Automatisation et Administration

- **Facturation Automatique :** Génération automatique de la facture dès la création d'une réservation (Calcul du montant total : $Prix \times Jours$).

— **Reporting Factures :** Visualisation, mise à jour du statut de paiement et suppression des factures.
— **Gestion des Plaintes :** Permettre aux clients de soumettre des réclamations et aux administrateurs de les traiter.

2.1.2 Besoins non fonctionnels

Les besoins non fonctionnels concernent les contraintes techniques et les critères de qualité du logiciel.

— **Performance et Réactivité :** Utilisation du *multithreading* (Threads) pour charger les listes de données en arrière-plan sans bloquer l'interface utilisateur.
— **Ergonomie et UX :** Interface graphique intuitive développée avec JavaFX (CSS moderne) pour minimiser le nombre de clics.
— **Maintenabilité :** Architecture en couches (N-Tier) permettant des mises à jour aisées (ex : changer de base de données sans impacter l'interface).
— **Robustesse :** Gestion rigoureuse des exceptions métier et validation des formulaires pour éviter les données incohérentes.
— **Portabilité :** Utilisation de Docker pour garantir que l'application s'exécute de manière identique sur Windows, Linux ou macOS.

2.2 Conception UML

2.2.1 Diagramme de classes

Le diagramme de classes illustre la structure statique de l'application de gestion d'hôtel. Il permet de représenter les différentes classes du système, leurs attributs, leurs méthodes, ainsi que les relations qui existent entre elles, telles que l'héritage, l'association ou la composition.

Dans notre application, les principales entités identifiées sont :

— **Chambre** : représente les informations relatives aux chambres de l'hôtel, telles que le numéro, le type, le prix et l'état (libre, occupée).
— **Client** : contient les informations personnelles du client et ses préférences.
— **Réservation** : gère les détails d'une réservation, incluant la date d'arrivée, la date de départ et le lien avec le client et la chambre concernée.
— **Personne** : classe générique dont héritent certains types d'utilisateurs, comme le client ou le personnel de l'hôtel.

Ce diagramme facilite la compréhension globale du système, en mettant en évidence l'architecture logicielle et les interactions principales entre les entités. Il constitue un support précieux pour le développement et la maintenance de l'application.

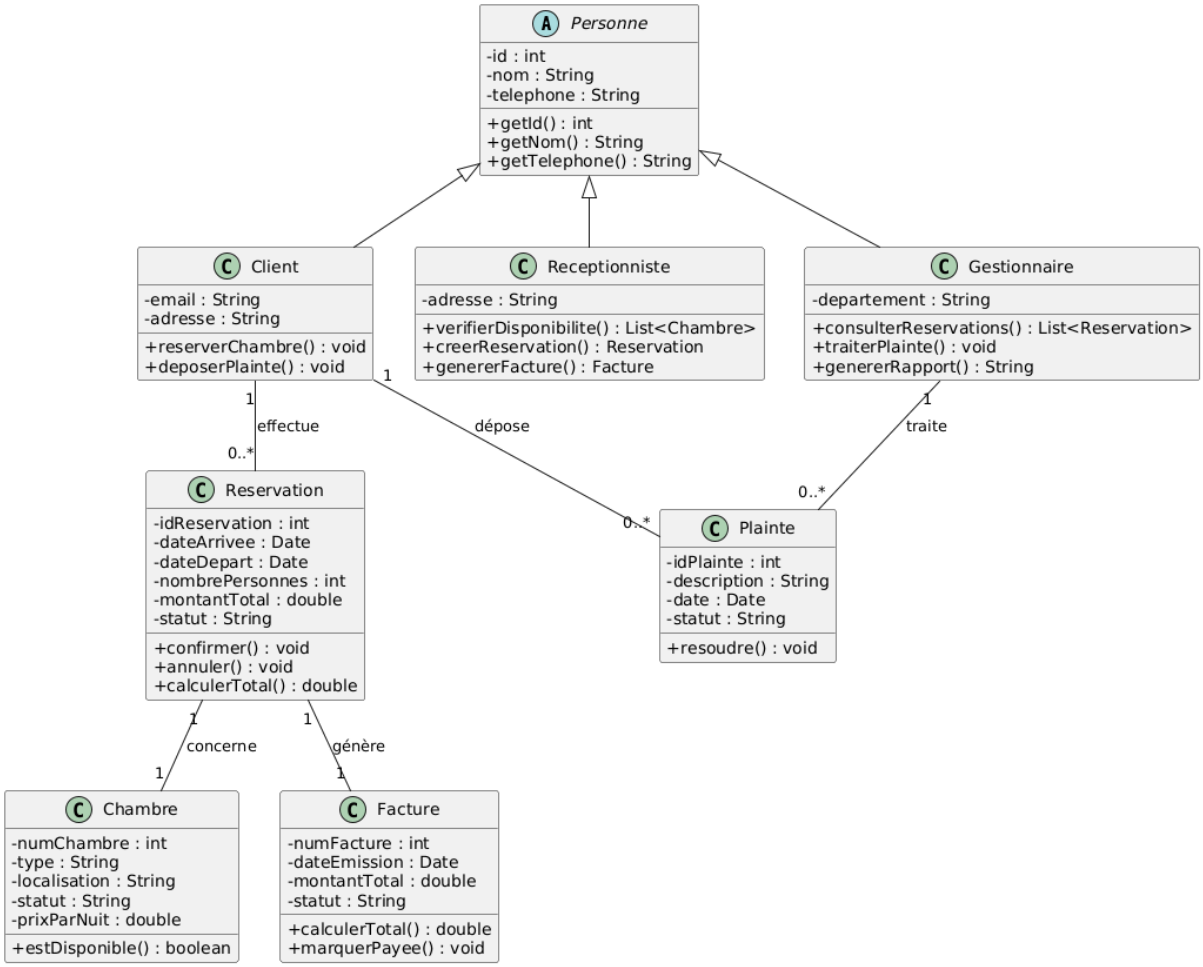

FIGURE 2.1 – Diagramme de classes de l'application ManagementHotel

2.3 Conception de la base de données

La base de données de l'application a été conçue à partir du diagramme de classes UML afin d'assurer une cohérence entre le modèle objet et le modèle relationnel. Chaque classe métier a été transformée en une table relationnelle, et les relations entre les classes ont été implémentées à l'aide de clés primaires et de clés étrangères.

Cette conception permet de garantir l'intégrité des données, d'éviter la redondance et de faciliter l'évolution future du système.

2.3.1 Modèle logique de données

Le modèle logique de données est composé des tables suivantes :

— **Personne** : Cette table contient les informations communes à tous les acteurs du système. Elle constitue la table mère dans la hiérarchie d'héritage.
  — id (clé primaire)
  — nom

  - téléphone
- **Client** : Représente les clients de l'hôtel. Cette table hérite de la table *Personne* et stocke les informations spécifiques aux clients.
  - id (clé primaire et clé étrangère vers Personne)
  - email
  - adresse
- **Receptionniste** : Contient les informations relatives aux réceptionnistes chargés de la gestion des réservations et des factures.
  - id (clé primaire et clé étrangère vers Personne)
  - adresse
- **Gestionnaire** : Représente les gestionnaires responsables du traitement des plaintes et de la supervision générale.
  - id (clé primaire et clé étrangère vers Personne)
  - departement
- **Chambre** : Cette table contient les informations relatives aux chambres de l'hôtel.
  - numChambre (clé primaire)
  - type
  - localisation
  - statut
  - prixParNuit
- **Reservation** : Stocke les réservations effectuées par les clients. Chaque réservation est associée à un client et à une chambre.
  - idReservation (clé primaire)
  - dateArrivee
  - dateDepart
  - nombrePersonnes
  - montantTotal
  - statut
  - client_id (clé étrangère vers Client)
  - numChambre (clé étrangère vers Chambre)
- **Facture** : Représente les factures générées à partir des réservations.
  - numFacture (clé primaire)
  - dateEmission
  - montantTotal
  - statut
  - reservation_id (clé étrangère vers Reservation)
- **Plainte** : Contient les réclamations déposées par les clients et traitées par les gestionnaires.
  - idPlainte (clé primaire)

— description
— date
— statut
— client_id (clé étrangère vers Client)
— gestionnaire_id (clé étrangère vers Gestionnaire)

2.4 Environnement Technique

Ce projet a été développé en utilisant un ensemble de technologies modernes garantissant performance, portabilité et maintenabilité.

— **Langage :** Java (JDK 23) - Utilisation des dernières fonctionnalités comme les *Record Patterns* et les *Unnamed Variables*.
— **IDE :** IntelliJ IDEA - Pour une gestion efficace du cycle de vie du projet.
— **Framework UI :** JavaFX - Utilisé avec le pattern MVC pour une interface utilisateur réactive et moderne.
— **SGBD :** MySQL - Pour la persistance des données via JDBC et Hibernate.
— **Build :** Maven - Pour la gestion automatisée des dépendances et du packaging.
— **Docker & Docker Compose :** pour l'orchestration de l'application et de la base de données.

Chapitre 3

Architecture et Implémentation

3.1 Architecture Logicielle

Pour garantir la flexibilité et la maintenabilité de l'application **MangmentHotel**, nous avons adopté une architecture logicielle de type **N-Tier** (en couches) couplée au pattern **MVC** (Modèle-Vue-Contrôleur). Cette séparation permet de modifier un composant technique (ex : changer de base de données) sans impacter l'interface utilisateur.

- **Couche Présentation (Controller & View) :** Définie par les fichiers FXML pour le design et les classes *Controller* pour la gestion de l'interactivité. Elle gère l'affichage des données et la capture des entrées utilisateur.
- **Couche Service (Métier) :** Centralise la logique métier de l'application. Par exemple, c'est ici que sont calculés les montants des factures et que les règles de validation (chambre occupée, format des dates) sont appliquées.
- **Couche Accès aux Données (DAO / Repository) :** Abstraite via une interface générique `IDao<T, ID>`, elle gère les interactions avec MySQL via JDBC et Hibernate.
- **Couche Modèle (Entities) :** Représente les données métier de l'hôtel (Chambres, Clients, Réservations, Plaintes) sous forme de classes Java .

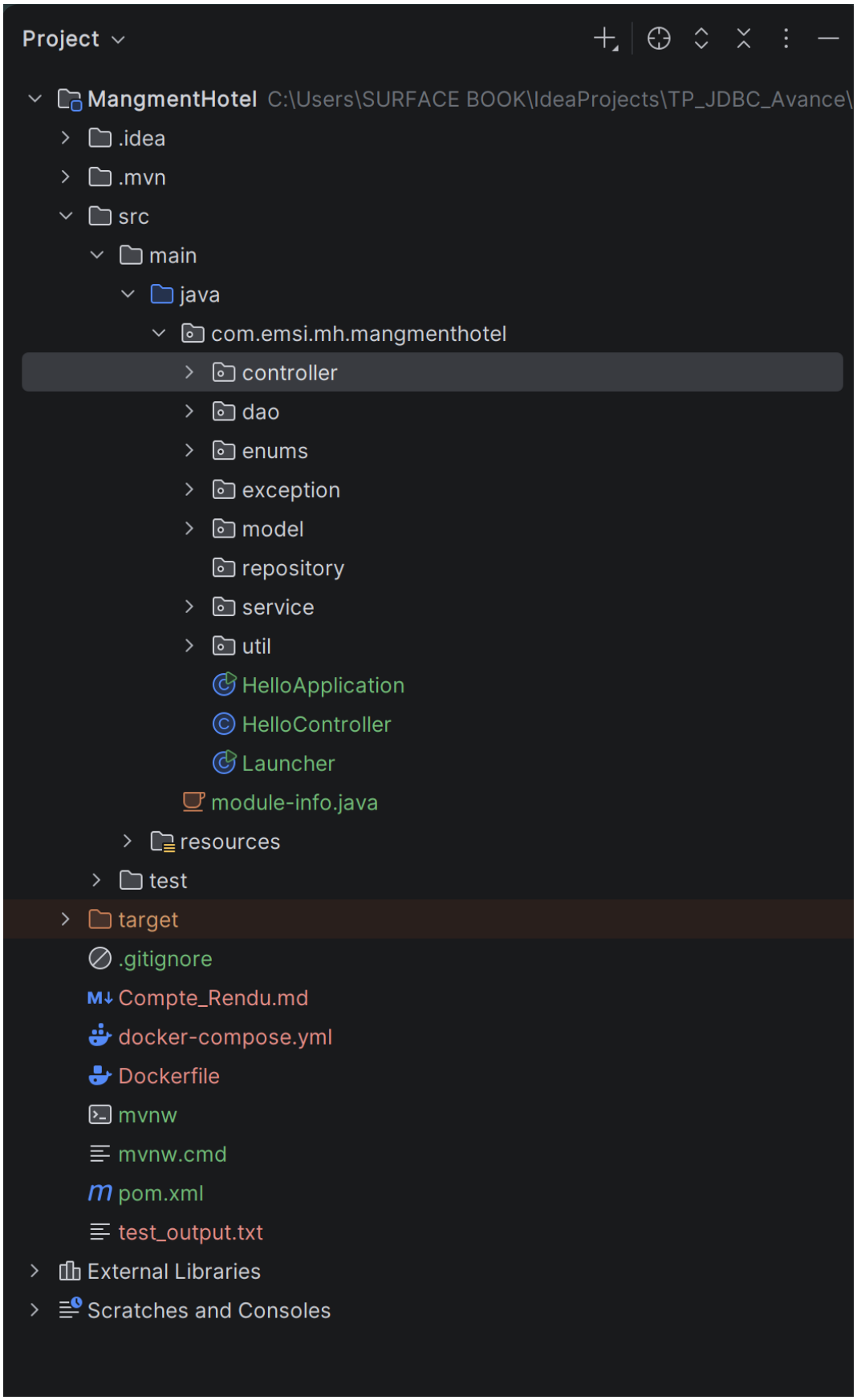

FIGURE 3.1 – Architecture N-Tier et pattern MVC de l'application ManagementHotel

Voici une explication détaillée de la structure de votre projet **MangmentHotel**. Le projet suit une architecture logicielle professionnelle de type **N-Tier** (en couches) avec le pattern **MVC** (Modèle-Vue-Contrôleur).

3.1.1 Fichiers à la Racine du Projet

- **pom.xml** : Le cœur du projet Maven. Contient la liste des dépendances et les instructions de compilation.
- **Dockerfile** : Crée l'image Docker de l'application.
- **docker-compose.yml** : Lance l'application et MySQL simultanément.
- **.gitignore** : Exclut les fichiers temporaires et configurations locales.
- **mvnw / mvnw.cmd** : Maven Wrapper pour compiler sans Maven installé.

3.1.2 Dossier `src/main/java` (Code Source)

Packages principaux

- **model** : Classes métier (`Chambre.java`, `Client.java`, `Facture.java`).
- **dao** : Interaction avec la base de données (`ChambreDAO.java`).
- **repository** : Abstraction au-dessus du DAO.
- **service** : Logique métier et automatisation des factures.
- **controller** : Liaison Vue Service.
- **enums** : Types constants (`StatutChambre`).
- **util** : Outils transversaux (`DBConnection.java`).
- **exception** : Classes d'erreurs personnalisées.

Fichiers spéciaux

- **HelloApplication.java** : Point d'entrée JavaFX.
- **Launcher.java** : Démarreur JavaFX.
- **module-info.java** : Configuration des modules Java.

3.1.3 Dossier `src/main/resources` (Interface)

- **view/** : Fichiers `.fxml` (fenêtres, boutons, tableaux).
- **com/emsi/mh/mangmenthotel/** : Correspond aux packages Java pour lier contrôleurs et vues.

3.1.4 Autres dossiers utiles

- **src/test** : Tests unitaires.
- **target/** : Généré par Maven, contient le `.jar` exécutable.

— **.idea/** : Configuration IntelliJ IDEA.

3.1.5 Résumé du flux de données

`FXML (Vue)` → `Controller` → `Service` → `DAO/Repository` → `Base de données`

3.1.6 Modèle de Données (`Chambre.java`, `Reservation.java`, etc.)

Les classes du modèle représentent les entités métier du système. Elles respectent le principe d'encapsulation et utilisent des annotations **JPA** pour le mapping objet-relationnel.

Afin de réduire le code répétitif (*boilerplate*), le projet utilise la bibliothèque **Lombok** à travers des annotations telles que `@Data`, `@Builder`, `@NoArgsConstructor` et `@AllArgsConstructor`.

```
@Data
@Entity
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Chambre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String numChambre;

    @Enumerated(EnumType.STRING)
    private TypeChambre type;

    private Double prixParNuit;

    @Enumerated(EnumType.STRING)
    private StatutChambre statut;
}
```

3.1.7 Abstraction et Généricité (`IDao.java`)

Afin de respecter le principe de réutilisabilité et de réduire la duplication du code, une interface générique `IDao<T, ID>` a été mise en place. Elle définit un contrat commun pour toutes les opérations de persistance (*CRUD*) quel que soit le type de l'entité manipulée.

```
public interface IDao<T, ID> {
    void create(T entity);
```

```
    T findById(ID id);
    List<T> findAll();
    void update(T entity);
    void deleteById(ID id);
}
```

Cette approche permet de changer facilement la technologie de persistance (JDBC ou Hibernate) sans impacter la couche métier.

3.1.8 Logique Métier et Utilisation des Streams

La couche service centralise la logique métier de l'application. Elle exploite l'API **Stream** de Java afin de traiter efficacement les collections d'objets métier.

Un exemple concret est le filtrage des réservations associées à un client spécifique :

```
public List<Reservation> getReservationsByClient(Long clientId) {
    return dao.findAll().stream()
            .filter(r -> r.getClient().getId().equals(clientId))
            .collect(Collectors.toList());
}
```

L'utilisation des Streams améliore la lisibilité du code et permet un traitement déclaratif plus performant que les boucles classiques.

3.1.9 Multithreading et Réactivité de l'Interface

Pour garantir une interface utilisateur fluide et éviter le gel (*freeze*) lors des accès à la base de données, les traitements lourds sont exécutés dans des Threads d'arrière-plan à l'aide de la classe `Task` de JavaFX.

```
private void loadChambres() {
    Task<ObservableList<Chambre>> task = new Task<>() {
        @Override
        protected ObservableList<Chambre> call() {
            return FXCollections.observableArrayList(
                chambreService.getAllChambres()
            );
        }
    };
    task.setOnSucceeded(e -> chambreTable.setItems(task.getValue())
       );
    new Thread(task).start();
}
```

Cette technique permet de maintenir une expérience utilisateur fluide, même lors du chargement de volumes importants de données.

Chapitre 4

Guide d'Exécution

4.1 Compilation avec Maven

```
mvn clean package
```

4.2 Exécution locale

```
mvn javafx:run
```

Chapitre 5

Couche de Persistance et Base de Données

Le projet prend en charge deux modes de persistance afin d'offrir flexibilité et contrôle selon les besoins techniques.

5.1 JDBC Classique

La persistance JDBC est assurée via la classe `DatabaseConnection.java`, offrant un contrôle précis sur les requêtes SQL exécutées.

5.2 Hibernate (ORM)

L'ORM Hibernate est utilisé via la classe `HibernateUtil.java`. Il permet un mapping objet–relationnel fluide entre les entités Java et la base de données MySQL.

5.3 Configuration Docker

Afin de faciliter le déploiement, Docker est utilisé pour lancer l'application et la base de données dans des conteneurs isolés.

```
1 docker compose up --build
```

5.4 Design Patterns Appliqués

Afin de produire un code propre et réutilisable, plusieurs patrons de conception (*Design Patterns*) ont été implémentés :

1. **Patron MVC (Model-View-Controller) :** Utilisé comme base pour séparer l'interface JavaFX, la logique de contrôle et les données.
2. **Patron Singleton :** Appliqué à la classe `DBConnection` pour garantir qu'une seule instance de connexion à la base de données est ouverte à travers toute l'application, optimisant ainsi les ressources.
3. **Patron DAO (Data Access Object) :** Utilisé pour isoler la couche de persistance. Cela permet d'utiliser indifféremment JDBC ou Hibernate sans modifier le code des services.
4. **Patron Builder (via Lombok) :** Utilisé pour la création d'objets complexes (comme `Reservation`) de manière lisible et sécurisée, évitant les constructeurs à rallonge.

5.5 Optimisations et Fonctionnalités Avancées

5.5.1 Multithreading et Réactivité

L'un des défis majeurs était de maintenir une interface fluide lors de la récupération de gros volumes de données. Nous avons utilisé les **Tasks** de JavaFX permettant d'exécuter les requêtes SQL dans des **Threads** d'arrière-plan. Cela évite le gel (*freeze*) de la fenêtre principale pendant les chargements.

5.5.2 API Streams et Traitement des Données

L'API **Streams** de Java a été privilégiée pour manipuler les collections de données. Elle est particulièrement utile pour :

— Filtrer les réservations spécifiques à un client connecté.
— Calculer dynamiquement le montant total des ventes.
— Rechercher des chambres libres selon des critères complexes.

5.5.3 Conteneurisation avec Docker

Le projet intègre des fichiers `Dockerfile` et `docker-compose.yml`, permettant de déployer l'application et sa base de données MySQL dans des conteneurs isolés. Cela garantit une portabilité totale et facilite le déploiement sur n'importe quel poste de travail.

Conclusion Générale

Le présent travail a porté sur la conception et la réalisation de l'application **MangmentHotel**, une solution logicielle destinée à automatiser et optimiser la gestion d'un établissement hôtelier. Ce projet a constitué une opportunité précieuse pour mettre en pratique des concepts avancés de développement logiciel dans un contexte professionnel.

Au cours de cette réalisation, nous avons atteint les objectifs fixés en relevant plusieurs défis techniques :

- **Sur le plan architectural :** L'adoption du pattern **MVC** et d'une structure en couches a permis de produire un code modulaire, facile à maintenir et à faire évoluer.
- **Sur le plan technique :** L'utilisation judicieuse du **Multithreading** (Threads JavaFX) a garanti une interface fluide, tandis que l'API **Streams** a simplifié le traitement complexe des données métier.
- **Sur le plan du déploiement :** L'intégration de **Docker** a assuré une portabilité totale de l'application, répondant aux standards actuels du génie logiciel.

Ce projet ne se limite pas à une simple interface de gestion ; il intègre des automatismes critiques tels que la génération de factures en temps réel et le suivi rigoureux des statuts de chambres, réduisant ainsi considérablement les risques d'erreurs humaines.

En guise de perspectives, cette solution pourrait être enrichie par l'ajout d'un module de **Reporting Décisionnel** (tableaux de bord statistiques sur le taux d'occupation) ou par le développement d'une interface **Web/Mobile** permettant aux clients d'effectuer leurs réservations à distance en toute autonomie.

Pour conclure, ce projet m'a permis de consolider mes compétences en tant qu'élève ingénieur, en alliant rigueur algorithmique, conception logicielle et outils de conteneurisation modernes.