


# **Rapport** **Bases de Données en Environnement Distribué**

Ducoeur Lancelot - Dupré Thibault

13 Décembre 2023



## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Architecture de l'Application</b>	<b>2</b>
2.1	Rôle de WildFly dans l'Architecture . . . . .	2
2.2	Composants de l'Architecture . . . . .	3
2.3	Intégration et Flux de Données . . . . .	3
2.4	Diagramme de l'Architecture . . . . .	3
<b>3</b>	<b>Implémentation EJB</b>	<b>4</b>
3.1	EJB de Type Message . . . . .	4
3.2	Différences entre EJB 2 et EJB 3 . . . . .	4
3.3	Gestion des Transactions Distribuées avec EJB . . . . .	5
<b>4</b>	<b>Mécanismes de Persistance</b>	<b>5</b>
4.1	Gestion de la Persistance dans EJB . . . . .	5
4.1.1	BMP avec AccountBmp.java . . . . .	6
4.1.2	Container-Managed Persistence (CMP) . . . . .	6
4.2	Comparaison entre JPA et Hibernate . . . . .	6
4.3	Choix de Persistance et Implications . . . . .	6
<b>5</b>	<b>Transactions et Sécurité</b>	<b>7</b>
5.1	Gestion des Transactions avec JPA . . . . .	7
5.2	Sécurité des Transactions . . . . .	7
<b>6</b>	<b>Tests et Validation</b>	<b>8</b>
6.1	Validation Manuelle avec Captures d'Écran . . . . .	8
<b>7</b>	<b>Analyse, Synthèse et Conclusion</b>	<b>10</b>
7.1	Évaluation des Technologies EJB . . . . .	10
7.2	Conclusions et Perspectives . . . . .	10

## 1 Introduction

Dans le cadre de l'évolution continue des technologies de l'information, l'**informatique bancaire** se doit de s'adapter pour offrir des services toujours plus performants et sécurisés. Ce rapport détaille le développement d'une application bancaire, permettant de *consulter le solde d'un compte bancaire* via une interface web, en utilisant les **Enterprise Java Beans** (EJB) version 3. L'application se concentre sur la simplicité d'utilisation tout en assurant une *sécurité* et une *fiabilité* optimales pour les utilisateurs finaux.

Nous présenterons l'architecture de l'application, qui démontre une intégration efficace des *serv-lets* et des différents types d'EJB, notamment les **Beans de session**, les **Beans entités** et les **Beans pilotés par message** (MDB). Ces composants travaillent ensemble pour gérer les opérations courantes telles que les transactions bancaires, l'administration des comptes et la persistance des données.

La conception de cette application est le fruit d'une analyse approfondie des besoins actuels en matière de *services bancaires en ligne*. Elle vise à démontrer une compréhension claire des *meilleures pratiques* et des *modèles de conception* dans le développement d'applications d'entreprise avec Java EE.

## 2 Architecture de l'Application

L'architecture de notre système bancaire en ligne est conçue pour optimiser les interactions entre les utilisateurs et le système de gestion de la base de données, tout en assurant un haut niveau de performance, de sécurité et de disponibilité. Elle repose sur les principes de l'Enterprise JavaBeans (EJB) pour la logique métier et les servlets pour le traitement des requêtes HTTP.

### 2.1 Rôle de WildFly dans l'Architecture

WildFly, un serveur d'applications Java EE flexible et performant, sert de plateforme pour déployer et exécuter nos EJBs et servlets. En tant que serveur d'applications, WildFly offre plusieurs avantages :

- **Support EJB complet** : WildFly assure une gestion optimale des différentes catégories d'EJBs utilisées dans notre application, y compris les Beans de session, les Beans entités, et les MDBs.
- **Performance et scalabilité** : WildFly est conçu pour des performances élevées et peut être configuré pour s'adapter à des charges de travail croissantes, garantissant ainsi la réactivité et l'évolutivité de notre application.
- **Gestion des transactions et de la sécurité** : WildFly intègre des mécanismes avancés pour la gestion des transactions et la sécurité, ce qui est essentiel pour les opérations bancaires.

L'utilisation de WildFly comme serveur d'applications influence l'ensemble de l'architecture, en facilitant la gestion des composants EJB, en améliorant la sécurité et en garantissant la performance nécessaire pour une application bancaire en ligne.

## 2.2 Composants de l'Architecture

Les composants principaux de notre architecture sont :

- **Servlets** : Servent de façade pour les interactions entre le client web et les couches de service EJB.
- **AdminServlet et ExampleServlet** : Gèrent les opérations d'administration et les exemples d'utilisation standard.
- **Beans EJB sans état (Stateless Session Beans)** : *ReaderBean* et *WriterBean* fournissent des fonctionnalités de lecture et d'écriture pour la manipulation des données.
- **InitializerBean** : Configure l'état initial de l'application et prépare les données nécessaires au démarrage.
- **Message-Driven Bean (MDB)** : Traite les messages de manière asynchrone à travers la file d'attente JMS, permettant une gestion des opérations bancaires sans blocage.

Ces composants sont interconnectés de manière à permettre une évolutivité et une maintenabilité accrues.

## 2.3 Intégration et Flux de Données

Les servlets reçoivent les requêtes des clients web et mobilisent les EJBs pour exécuter la logique métier. Les beans *Reader* et *Writer* interagissent avec la base de données pour effectuer les opérations requises. L'*InitializerBean* s'occupe de la préparation de l'environnement d'exécution initial.

## 2.4 Diagramme de l'Architecture

Le diagramme suivant illustre la structure interne et les interactions entre les différents composants du système :

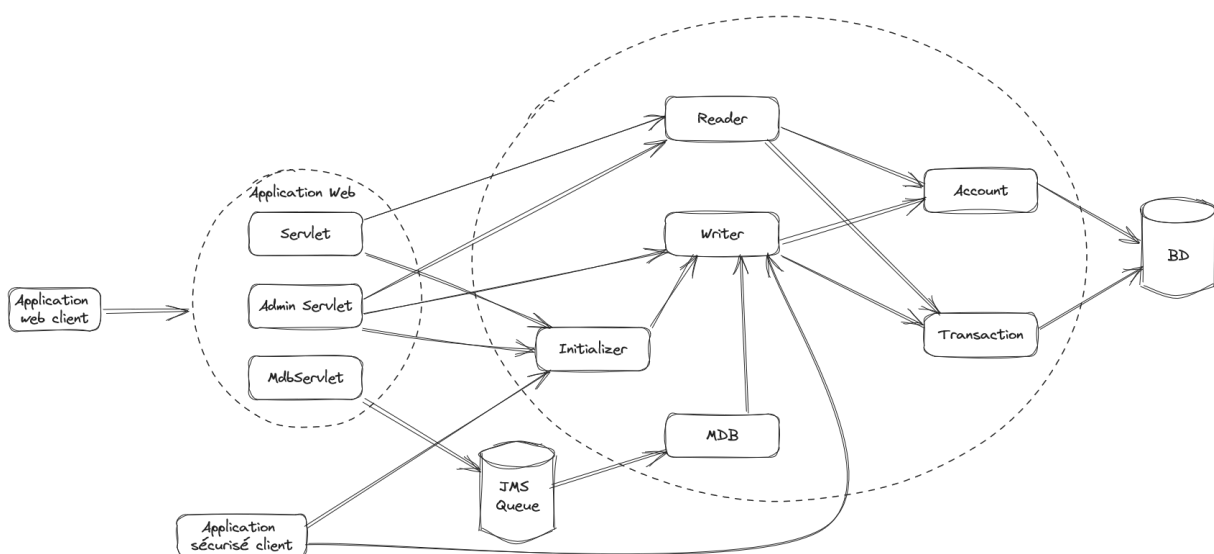


FIGURE 1 – Diagramme illustrant l'architecture de l'application bancaire.

Le schéma ci-dessus représente l'architecture de notre application bancaire en ligne, mettant en avant une conception modulaire et une séparation claire des responsabilités. Au premier plan, nous

avons les *Servlets* qui servent de point de contact initial pour l'utilisateur, gérant les requêtes et les réponses entre le client web et le serveur.

Le **AdminServlet** est conçu spécifiquement pour gérer les opérations administratives, telles que la création et la gestion des comptes utilisateurs, tandis que l'*ExampleServlet* traite les requêtes standard de l'utilisateur, telles que la consultation des soldes et l'exécution des transactions.

Au cœur de notre système, les **EJBs** jouent un rôle crucial dans l'orchestration de la logique métier. Le *ReaderBean* est responsable de toutes les opérations de lecture, extrayant les informations nécessaires de la base de données, tandis que le *WriterBean* s'occupe des opérations d'écriture, mettant à jour les données en réponse aux actions de l'utilisateur.

L'**InitializerBean** assure que l'application démarre avec un état cohérent et pré-configuré, établissant les paramètres initiaux et les données de référence nécessaires au bon fonctionnement de l'application.

La communication asynchrone est gérée par le **Message-Driven Bean (MDB)**, qui écoute une file d'attente JMS et traite les messages reçus, facilitant ainsi un traitement des transactions non bloquant et efficace.

La base de données est le réceptacle de toutes les entités métier, telles que les *Accounts* et les *Transactions*, et est interfacée par les EJBs pour une persistance et une récupération optimales des données.

## 3 Implémentation EJB

### 3.1 EJB de Type Message

Dans le cadre de l'application bancaire, les **Enterprise JavaBeans (EJB)** de type message jouent un rôle essentiel dans le traitement asynchrone des opérations bancaires. Le *JMSMessageBean.java*, un **Message-Driven Bean (MDB)**, incarne cette catégorie d'EJB. Son rôle est d'écouter les messages placés dans une file d'attente JMS et de réagir en conséquence, ce qui permet de découpler le processus de traitement de la demande client de la réponse du serveur. Cette approche apporte une amélioration significative de la performance et de la réactivité de l'application, car elle permet de gérer les charges de travail importantes sans bloquer les utilisateurs durant les opérations de traitement des données.

### 3.2 Différences entre EJB 2 et EJB 3

La migration de la spécification EJB 2 vers EJB 3 a introduit plusieurs améliorations qui ont eu un impact positif sur la qualité du code et la productivité du développement. Les différences notables sont :

- **Simplification du développement** : EJB 3 utilise des annotations pour la déclaration des beans, éliminant ainsi le besoin de fichiers de déploiement descriptifs et réduisant la complexité du code.

- **Injection de dépendances** : Elle permet une meilleure gestion des dépendances et une intégration plus facile des composants EJB, ce qui contribue à une architecture plus propre et à un couplage plus faible.
- **Persistance simplifiée** : Avec l'introduction de l'API Java Persistence (JPA), la gestion des entités et des opérations CRUD est devenue plus intuitive et moins verbeuse.
- **Meilleures pratiques et patterns** : EJB 3 encourage l'utilisation de patterns de conception modernes, qui favorisent un code bien structuré et maintenable.

Ces améliorations ont abouti à un code plus lisible et plus facile à maintenir, réduisant le risque d'erreurs et accélérant le cycle de développement. C'est pourquoi nous avons décidé d'utiliser EJB3.

### 3.3 Gestion des Transactions Distribuées avec EJB

La gestion des transactions distribuées dans les applications utilisant les Enterprise Java Beans (EJB) est un élément clé pour assurer la cohérence et l'intégrité des données. Ces transactions, impliquant plusieurs composants et ressources, sont cruciales lorsque de multiples opérations doivent être traitées de manière atomique.

Les EJB gèrent ces transactions complexes via la Java Transaction API (JTA). Cette intégration permet d'assurer que toutes les opérations entreprises par les beans, telles que les lectures et écritures dans la base de données, sont soit toutes confirmées (commit) soit toutes annulées (rollback) en cas de défaillance. Cette stratégie garantit l'intégrité des données dans des environnements où la fiabilité des transactions est primordiale.

## 4 Mécanismes de Persistance

En utilisant Oracle comme système de gestion de base de données, notre application EJB 3 bénéficie de fonctionnalités adaptées à la gestion des transactions et à l'optimisation des requêtes, ce qui est crucial dans le contexte de la Bean Managed Persistence. Oracle, avec ses capacités de traitement de grandes quantités de données, apporte une solution viable pour la gestion de données complexes dans notre application, tout en offrant les outils nécessaires pour une gestion efficace de la persistance.

### 4.1 Gestion de la Persistance dans EJB

Dans le cadre de l'EJB 3 avec BMP, envisager un modèle de conception comme l'ObjectFactory pourrait offrir une centralisation bénéfique de la création d'objets. Cela permettrait de modifier aisément la logique d'instanciation sans perturber les composants de l'application. Bien que non utilisé dans notre architecture actuelle, ce modèle pourrait être envisagé pour augmenter la flexibilité et la maintenabilité des processus de persistance à l'avenir.

La persistance des données est un aspect fondamental de notre application bancaire, et différentes approches ont été explorées pour gérer cette persistance, notamment la Bean-Managed Persistence (BMP) et la Container-Managed Persistence (CMP).

#### 4.1.1 BMP avec AccountBmp.java

Dans *AccountBmp.java*, la persistance est gérée manuellement par le développeur, démontrant l'approche BMP. Cette stratégie offre un contrôle détaillé sur les interactions avec la base de données. La méthode de suppression d'un compte en est un exemple typique :

```
public void delete() {
    try {
        InitialContext ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup("java:/OracleDS");

        try (Connection conn = ds.getConnection();
            PreparedStatement stmt = conn.prepareStatement(
                "DELETE FROM account WHERE id = ?")) {
            stmt.setLong(1, this.id);
            stmt.executeUpdate();
        }
    } catch (NamingException e) {
        System.err.println("Erreur JNDI : " + e.getMessage());
    } catch (SQLException e) {
        System.err.println("Erreur de connexion a la BD : " + e.getMessage());
    }
}
```

#### 4.1.2 Container-Managed Persistence (CMP)

En complément de BMP, CMP a également été utilisée dans certaines parties de l'application. Avec CMP, le conteneur EJB gère automatiquement la persistance des entités, ce qui allège la charge des développeurs pour les interactions avec la base de données. CMP fournit une abstraction élevée et simplifie la gestion des données, mais peut manquer de flexibilité pour des cas d'utilisation spécifiques ou des optimisations de performances.

### 4.2 Comparaison entre JPA et Hibernate

La Java Persistence API (JPA) et Hibernate sont employés pour faciliter la persistance des données. JPA, utilisée dans *Account.java* et *Transaction.java*, définit le mappage entre les objets Java et les tables de base de données. Hibernate, en tant qu'implémentation de JPA, offre des fonctionnalités supplémentaires et des optimisations de performance.

- **JPA** : Simplifie la définition des entités et des relations à l'aide d'annotations ou de fichiers XML, favorisant ainsi une approche standardisée et portable de l'ORM.
- **Hibernate** : Propose une implémentation de JPA avec des extensions pour des cas d'utilisation plus complexes et des performances accrues, notamment grâce à ses stratégies de caching avancées.

### 4.3 Choix de Persistance et Implications

Le choix entre BMP, CMP, et l'utilisation de JPA avec ou sans Hibernate dépend des besoins spécifiques de l'application. BMP offre un contrôle maximal sur les opérations de base de données, CMP simplifie la gestion des données mais avec moins de flexibilité, tandis que JPA fournit un équi-

libre entre simplicité et contrôle, surtout avec l'appui de Hibernate pour des fonctionnalités étendues.

Pour une gestion optimale des connexions au système de gestion de base de données (SGBD) dans un environnement bancaire, deux stratégies clés peuvent être adoptées :

1. **Mise en place d'un pool de connexions** : Cette technique évite la création d'une nouvelle connexion pour chaque requête en réutilisant les connexions existantes. Elle implique le maintien d'un ensemble de connexions disponibles, attribuées aux requêtes selon les besoins. Cela permet de réduire le temps nécessaire pour établir de nouvelles connexions et d'améliorer la gestion des ressources du SGBD.
2. **Implémentation d'un système de réplication de bases de données** : Cette approche vise à répartir la charge des requêtes sur plusieurs serveurs de base de données. Elle se base sur la synchronisation des données entre les serveurs pour garantir que tous soient constamment à jour et aptes à répondre efficacement aux requêtes.

## 5 Transactions et Sécurité

### 5.1 Gestion des Transactions avec JPA

La gestion des transactions dans notre application bancaire est illustrée par l'entité *Account*, qui utilise JPA pour interagir avec la base de données de manière transactionnelle. Les transactions sont gérées par l'EJB Container, qui utilise le contexte de persistance pour s'assurer que les opérations sur les entités sont effectuées dans le cadre d'une transaction. Voici un exemple de gestion des transactions dans *Account.java* :

```
@Entity
public class Account implements Serializable {
    // ... (attributs de la classe)

    public void addTransaction(String typeTransaction, float amount) {
        Transaction transaction = new Transaction(typeTransaction, amount, this);
        // La persistance de la transaction est gérée automatiquement par JPA
        // grâce à la relation 'OneToMany' avec cascade 'CascadeType.ALL'
        transactions.add(transaction);
    }

    // ... (autres méthodes)
}
```

Dans cet extrait, l'ajout d'une transaction à un compte se fait par la méthode *addTransaction*, qui crée une nouvelle instance de *Transaction* et l'ajoute à la collection de transactions associée au compte. Grâce à l'annotation *@OneToMany* avec l'option *cascade=CascadeType.ALL*, toutes les opérations effectuées sur l'entité *Account* sont propagées à ses transactions, y compris la persistance, ce qui assure l'intégrité de nos données lors des opérations de mise à jour.

### 5.2 Sécurité des Transactions

La sécurité des transactions est renforcée par des mécanismes de verrouillage optimiste dans JPA, qui préviennent les conflits de données dans un environnement concurrent. Des contrôles d'accès basés sur les rôles peuvent être intégrés pour limiter les opérations selon les privilèges de l'utilisateur, offrant ainsi une couche supplémentaire de sécurité.



```
@Entity
@NamedQueries({
    // ... (Requetes nommees)
})
public class Account implements Serializable {
    // ... (attributs et methodes)

    // Methodes de gestion des acces et de la securite peuvent etre ajoutees ici
}
```

En combinant les fonctionnalités de JPA avec les mécanismes de sécurité intégrés dans les EJB, notre application offre un cadre robuste pour effectuer des transactions bancaires sécurisées et maintenir l'intégrité des données.

## 6 Tests et Validation

La validation des fonctionnalités de notre application bancaire est essentielle pour garantir une expérience utilisateur sans faille. Au lieu de tests unitaires traditionnels, nous avons opté pour une série de tests d'intégration et de validations manuelles, dont les résultats sont capturés sous forme de captures d'écran démontrant le bon fonctionnement de l'application.

### 6.1 Validation Manuelle avec Captures d'Écran

Les captures d'écran suivantes montrent l'application en action, validant le comportement attendu lors de l'exécution des opérations bancaires et l'interaction avec le Message-Driven Bean (MDB).



Nom du compte	Solde	Opérations
Lancelot	10035.0 euros	Retrait de 15.0 euros Ajout de 50.0 euros
Thibault	535.0 euros	Retrait de 15.0 euros Ajout de 50.0 euros

Ajouter un account

FIGURE 2 – Liste des comptes avant l'ajout de nouvelles opérations.

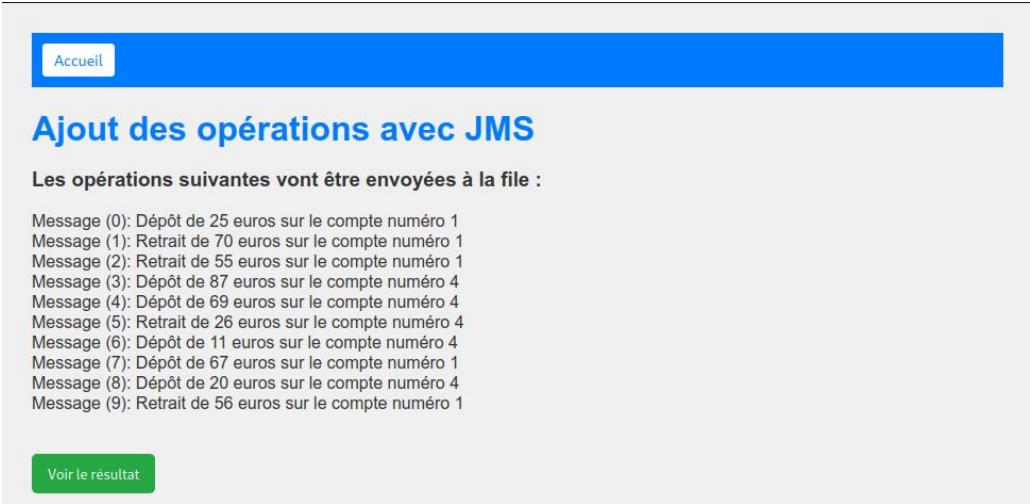


FIGURE 3 – Vue de la file d’attente JMS montrant les messages traités par le MDB.

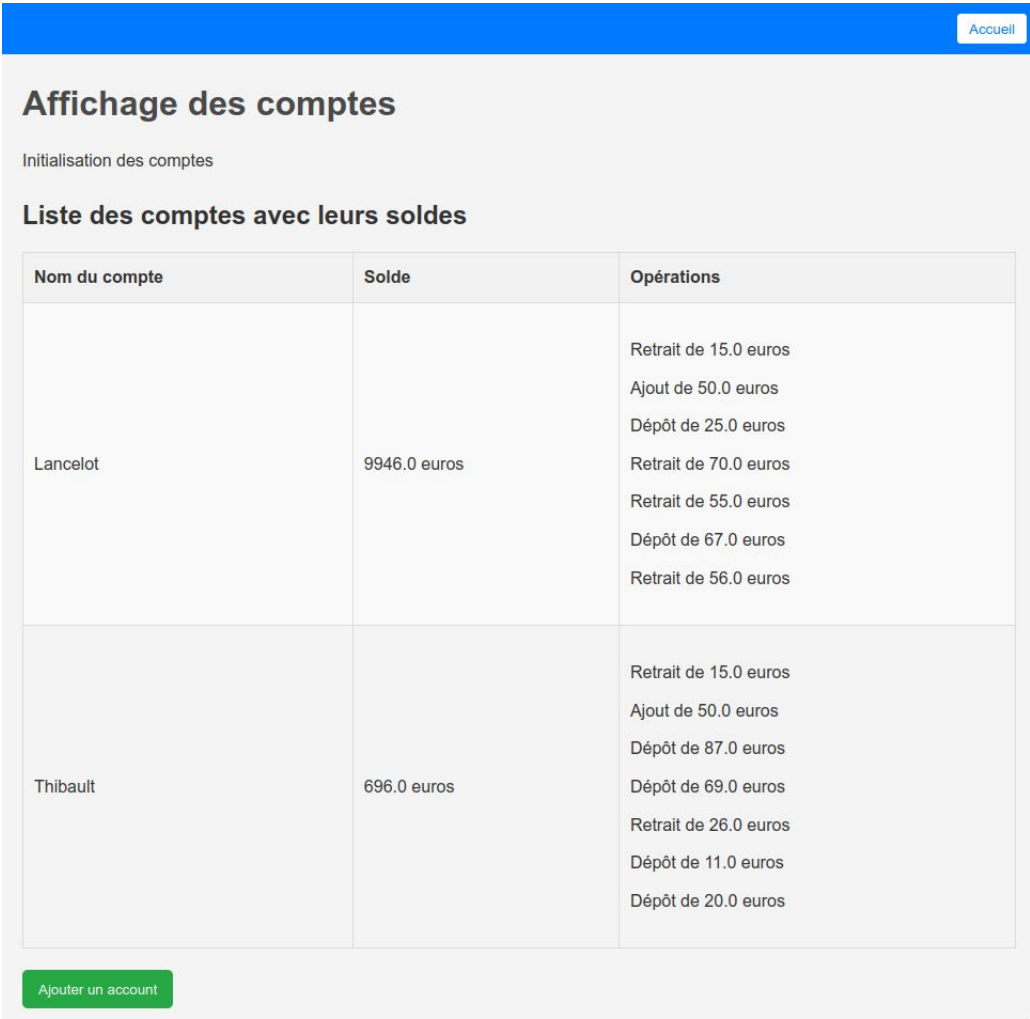


FIGURE 4 – Liste des comptes après l’ajout de nouvelles opérations avec JMS.

## 7 Analyse, Synthèse et Conclusion

Le projet de développement d'une application bancaire utilisant les Enterprise JavaBeans (EJB) a permis d'explorer en profondeur les capacités et les défis associés à ces technologies dans un contexte d'entreprise.

### 7.1 Évaluation des Technologies EJB

Les EJBs se sont avérés être des outils efficaces pour structurer la logique métier et gérer les transactions, la sécurité et la persistance des données. Leur capacité à offrir une abstraction de haut niveau pour ces aspects complexe de l'application a amélioré la qualité et la maintenabilité du code. Cependant, la complexité inhérente à la configuration et à la gestion des EJBs, notamment en ce qui concerne le débogage et les tests, représente un défi notable. La compréhension approfondie de leur fonctionnement est essentielle pour exploiter pleinement leur potentiel.

### 7.2 Conclusions et Perspectives

Ce projet a démontré l'importance d'une architecture bien conçue et d'une planification détaillée, en particulier pour les fonctionnalités telles que la gestion des transactions distribuées et l'intégration des MDB. La mise en œuvre des EJBs, malgré ses défis, offre un cadre robuste pour les applications d'entreprise, en assurant une gestion efficace et sécurisée des opérations.

Pour terminer, l'expérience acquise à travers ce projet offre des perspectives précieuses pour le développement futur d'applications d'entreprise. Les leçons tirées en termes de conception, de gestion des transactions, et de sécurité des applications sont des connaissances essentielles pour tout développeur travaillant dans l'environnement Java EE. Elles soulignent l'importance d'une compréhension approfondie des technologies sous-jacentes pour la création d'applications fiables et performantes dans un environnement professionnel.