iSMAGi®
Institut Supérieur de Management
d'Administration et de Génie Informatique
المعهد العالي للتدبير و الإدارة و الهندسة المعلوماتية

Rapport de Projet : Système Distribué de Réservation de Salles de Réunion

Objectif : Conception et implémentation d'une architecture multi-tier et multi-protocole

---

Cours : Systèmes Distribués

Rédigé par:
FranCk KINANI NKAYA

Année universitaire 2025/ 2026

1. Introduction

1.1 Contexte du projet

Dans le cadre des entreprises modernes, la gestion des ressources physiques, notamment les salles de réunion, est devenue un défi logistique. L'utilisation de méthodes manuelles (tableurs, emails) mène inévitablement à des chevauchements de réservations ou à une sous-utilisation des espaces.

1.2 Problématique

Comment concevoir un système qui soit à la fois centralisé pour la cohérence des données, mais distribué pour permettre l'accès depuis diverses plateformes (Web, Mobile, Desktop, ou applications tierces) ? Le défi réside dans la synchronisation des accès concurrents et la multiplicité des interfaces de communication.

1.3 Objectifs

Le projet vise à fournir une solution complète permettant :

La gestion CRUD des utilisateurs et des salles.

Un moteur de réservation intelligent évitant les doublons.

Une exposition de ces services via quatre canaux de communication distincts.

2. Technologies Utilisées

Java & JavaFX : Choisi pour la robustesse du typage et la richesse de l'interface graphique permettant de créer un client lourd performant.

JPA / Hibernate : Pour faire abstraction du langage SQL et manipuler des objets Java, facilitant ainsi la maintenance du code.

MySQL : Une base de données relationnelle éprouvée pour garantir l'intégrité référentielle.

Protocoles de communication :

REST : Pour l'ouverture vers le web et la légèreté du JSON.

SOAP : Pour la sécurité et le contrat formel (WSDL) indispensable en milieu industriel.

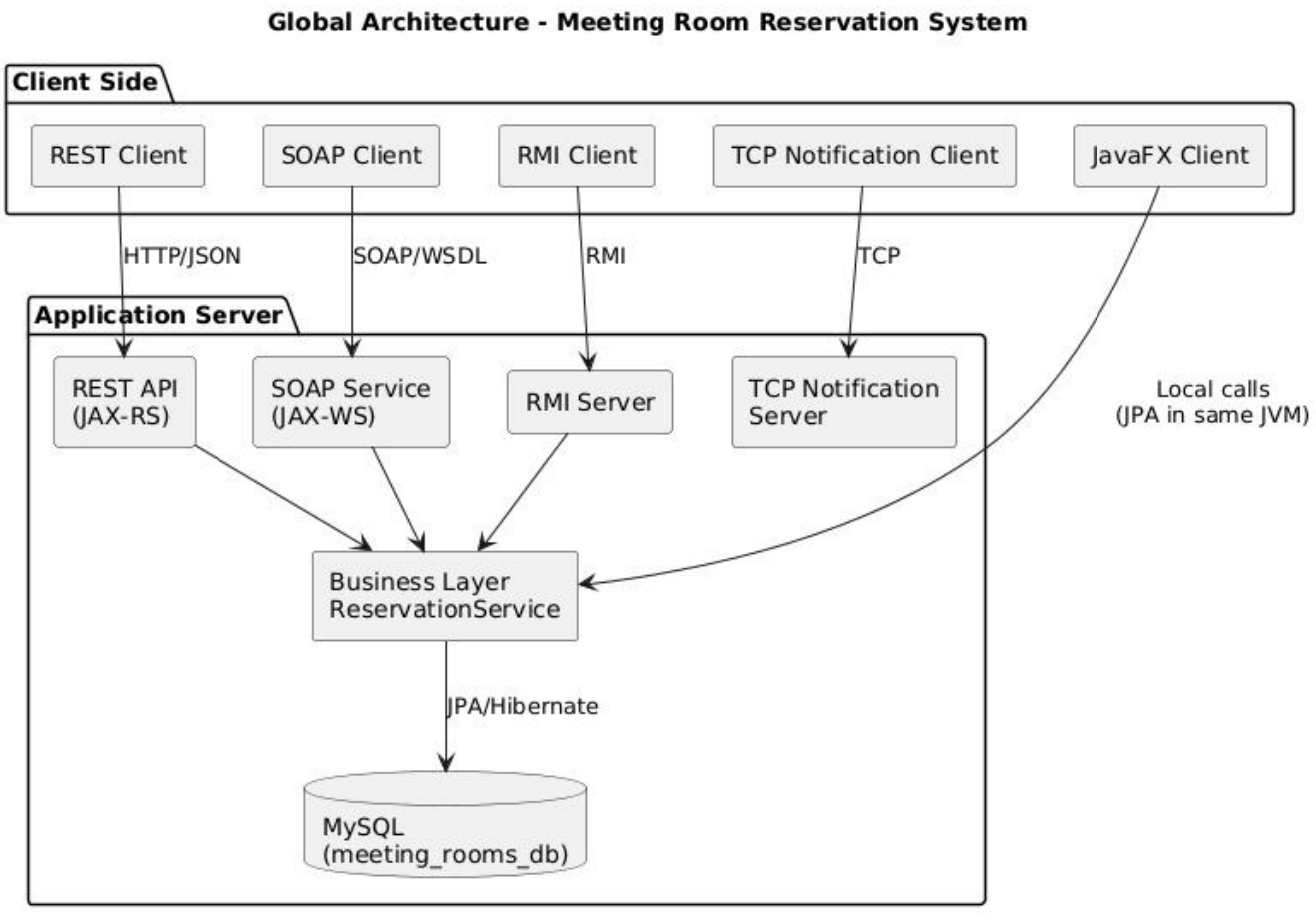
RMI : Pour la performance maximale entre deux applications Java.

Sockets TCP : Pour le 'Push' d'informations en temps réel sans attendre une requête du client.

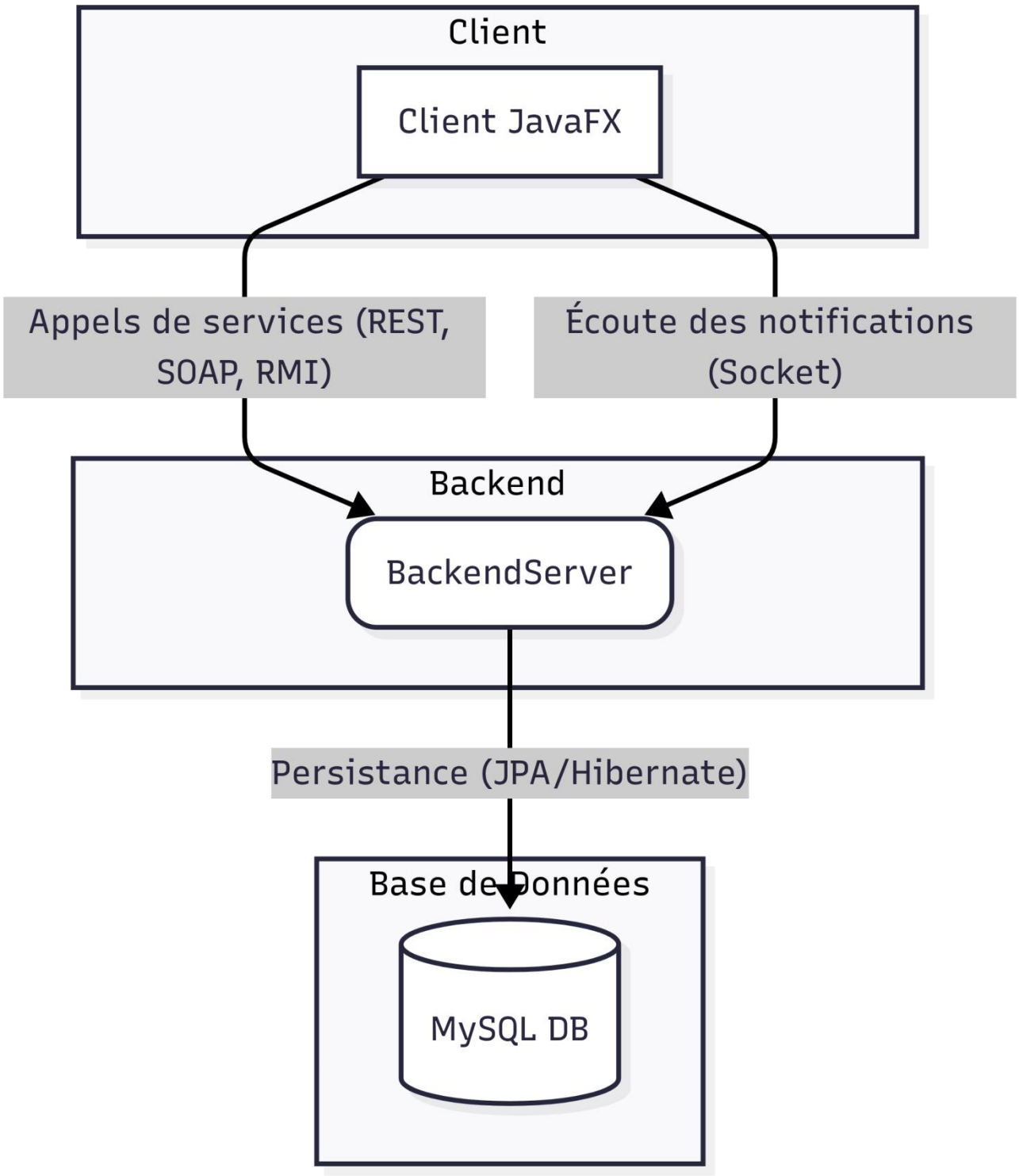
3. Architecture du Système

3.1 Vue d'ensemble

Le système adopte une architecture n-tier. Elle se compose d'une couche de présentation (Client), d'une couche de services (Serveur d'application) et d'une couche de données (Base de données).



[Image de Global Architecture – Meeting Room Reservation System]

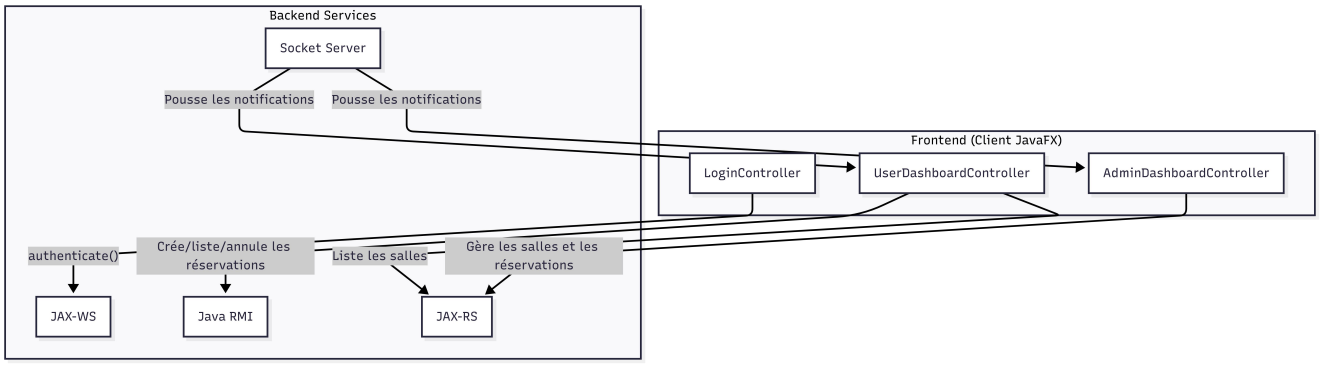


3.2 Le Pattern ‘Service Layer‘

L’innovation majeure de ce projet est la classe ReservationService.

Centralisation : Tous les points d'entrée (REST, SOAP, RMI) appellent les mêmes méthodes de cette classe.

Consistance : Si une règle métier change (ex: durée maximale d'une réunion), elle n'est modifiée qu'à un seul endroit.

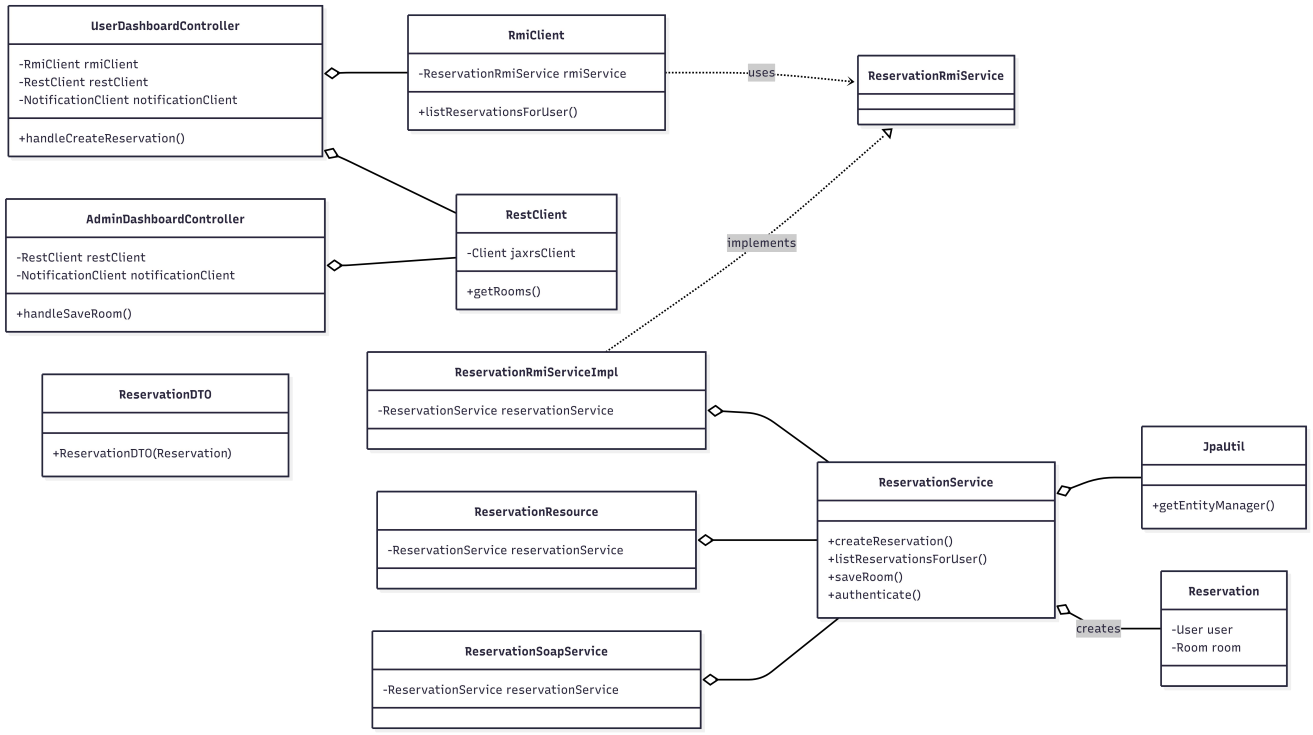


4. Conception de la Base de Données

4.1 Modèle de Données

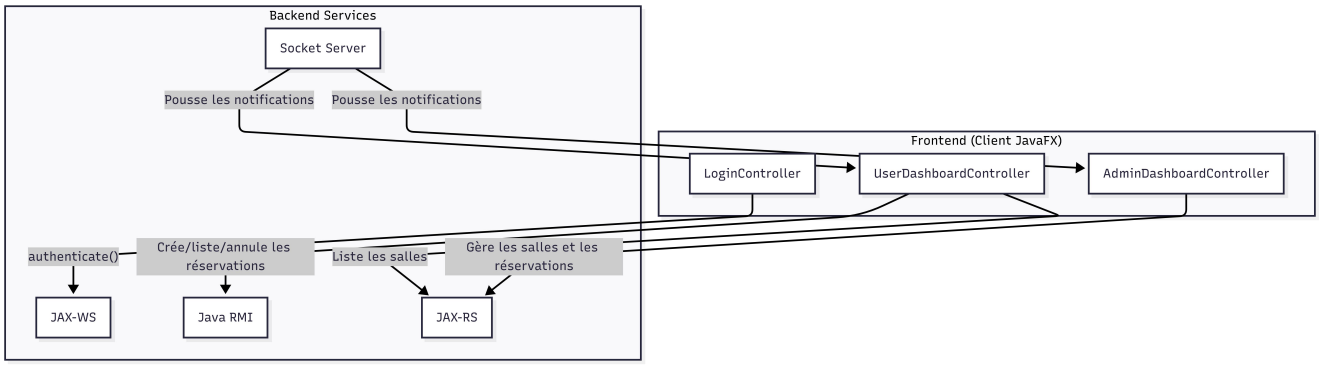
Le schéma repose sur trois piliers :

1. User : Gère l'authentification et les rôles (Admin/User).
2. Room : Définit les capacités et équipements (Projecteur, Tableau, etc.).
3. Reservation : La table de jointure 'intelligente' liant un utilisateur à une salle avec une contrainte temporelle (startDateTime, endDateTime).



5. Implémentation des Services Distribués

L'originalité de ce projet réside dans son architecture multi-canal. Une seule logique métier, centralisée dans le ReservationService, est exposée via quatre protocoles de communication différents. Cette approche permet de garantir que, quel que soit le client utilisé (mobile, web, ou desktop), les règles de gestion restent identiques.



5.1 API REST (JAX-RS) : L'ouverture vers le Web

L'API REST est conçue pour être légère et interopérable avec n'importe quelle technologie capable d'effectuer des requêtes HTTP.

Ressources exposées : Le système utilise RoomResource pour la gestion des salles et ReservationResource pour les réservations.

Format de données : Les échanges se font exclusivement en JSON, facilitant l'intégration avec des frameworks front-end modernes (React, Angular).

Gestion des codes d'état HTTP :

201 Created : Confirmant la réussite d'une réservation.

409 Conflict : Renvoyé spécifiquement lorsque la salle est déjà occupée pour le créneau demandé.

Avantages : Sa simplicité et son caractère sans état (stateless) en font le choix idéal pour une extension vers des clients légers ou mobiles.

5.2 Services Web SOAP (JAX-WS) : La rigueur du contrat

Le service SOAP est implémenté via ReservationSoapService pour répondre aux besoins d'interopérabilité forte dans des environnements hétérogènes.

Contrat WSDL : Contrairement au REST, SOAP repose sur un fichier WSDL (Web Services Description Language) généré par le SoapPublisher. Ce fichier définit de manière rigoureuse les types de données et les opérations disponibles (createReservation, cancelReservation).

Typage fort : L'utilisation du format XML garantit que les données transmises respectent strictement le schéma défini, minimisant les erreurs de parsing.

5.3 Java RMI (Remote Method Invocation) : La performance native

Le protocole RMI est utilisé pour permettre une communication transparente entre deux machines virtuelles Java (JVM).

Transparence : L'implémentation ReservationRmiServiceImpl permet au client d'appeler des méthodes distantes comme s'il s'agissait d'objets locaux.

Mécanisme de registre : Le RmiServer enregistre le service dans un registre RMI, permettant aux clients de 'localiser' l'objet distant par son nom.

Passage d'objets : RMI facilite le transfert d'objets Java complets (sérialisés), ce qui simplifie grandement le développement par rapport aux protocoles basés sur du texte comme HTTP.

5.3.1 Sérialisation Java pour RMI

Pour que les objets Java (comme User, Room ou Reservation) puissent transiter entre la JVM du serveur et celle du client via RMI, ils doivent impérativement implémenter l'interface java.io.Serializable.

Le mécanisme de Marshalling/Unmarshalling : Lorsqu'un client appelle une méthode distante, les arguments sont transformés en flux d'octets (marshalling) pour être envoyés sur le réseau, puis reconstruits sur le serveur (unmarshalling).

Identifiant de version : Chaque classe sérialisable possède un serialVersionUID. Si le serveur et le client n'ont pas la même version de la classe, une InvalidClassException est levée, garantissant ainsi l'intégrité des données distribuées.

Passage par valeur vs Référence : En RMI, les objets simples sont passés par valeur (copie), tandis que les objets distants (ceux héritant de Remote) sont passés par référence via un 'stub'.

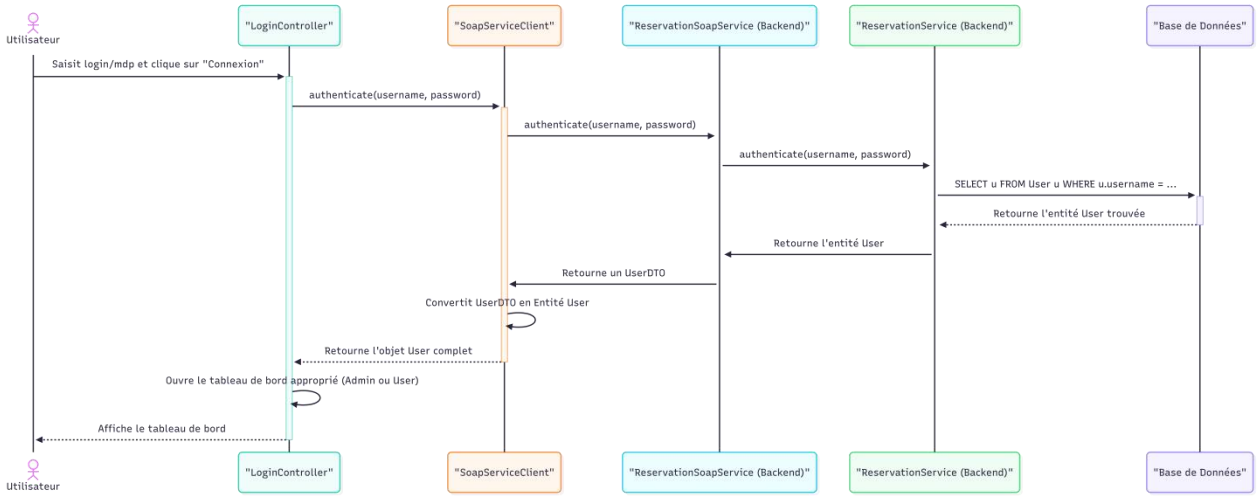
5.4 Sockets TCP : Notifications et temps réel

Pour combler l'absence de 'Push' natif dans les protocoles HTTP classiques, un serveur de notification basé sur les sockets TCP a été mis en place.

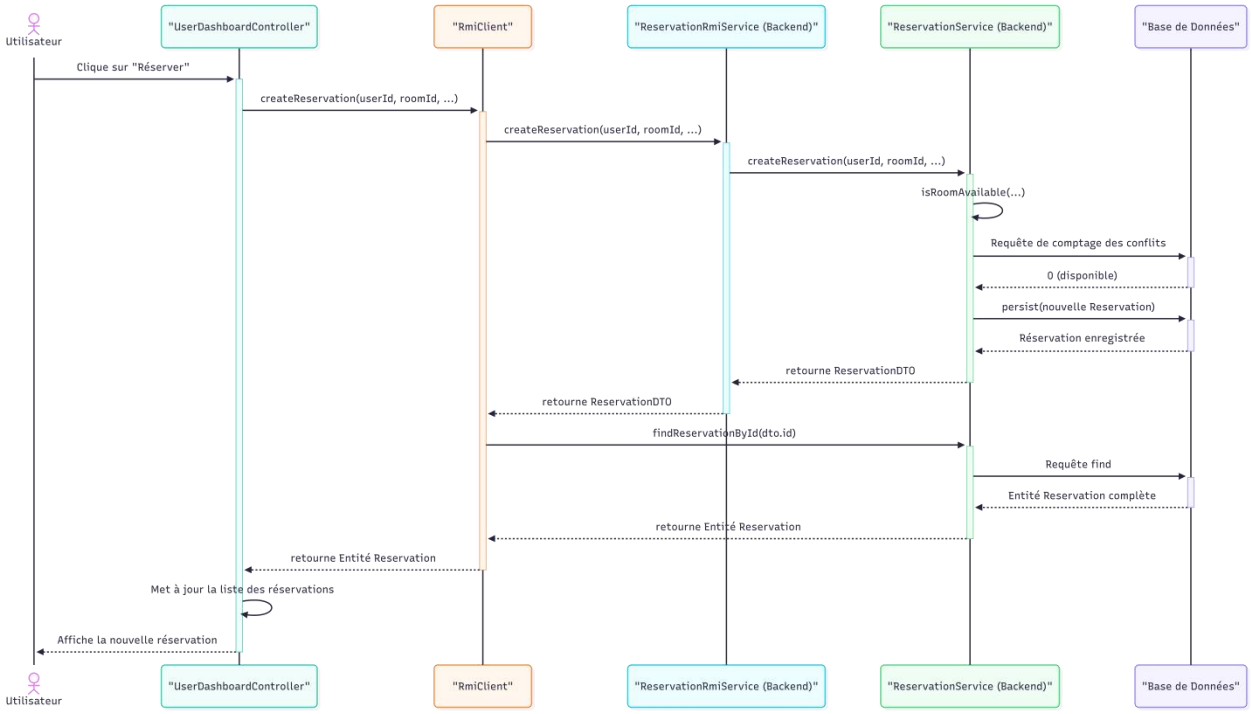
Architecture Push : Le NotificationServer maintient une liste de clients actifs. Dès qu'une action de réservation ou d'annulation est effectuée sur le serveur, un message est diffusé à tous les clients connectés.

Mise à jour de l'interface : Le NotificationClient reçoit ces flux asynchrones, permettant une mise à jour de la vue utilisateur en temps quasi réel sans action manuelle.

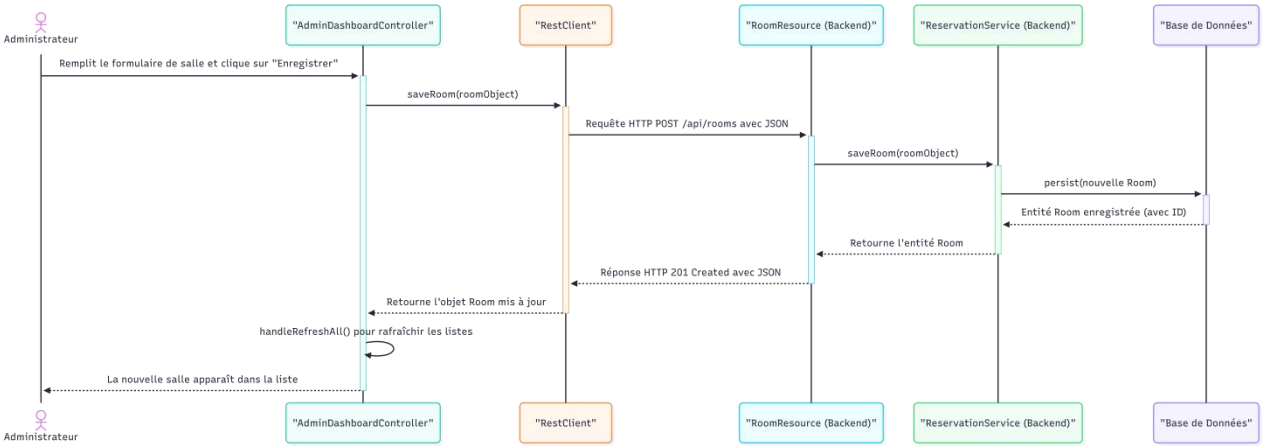
Diagrammes de use cases



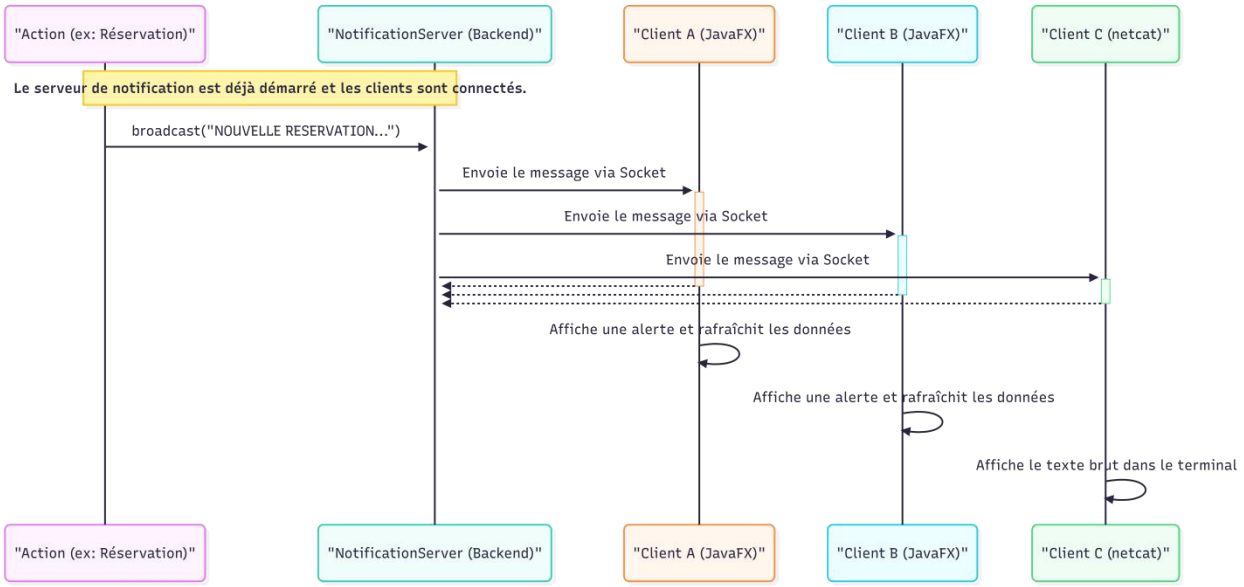
Connexion d'un utilisateur



Use case reservation



Use case ajout salle



Use case notifiaction

6. Logique Métier : Algorithme de Conflit et Suggestion2

L'une des fonctionnalités les plus critiques est la garantie qu'une salle ne puisse jamais être réservée deux fois sur le même créneau. Cette logique est encapsulée dans la méthode createReservation de la classe ReservationService.

6.1 L'algorithme de détection d'intersection

Pour vérifier si une nouvelle réservation entre en conflit avec une réservation existante, le système n'utilise pas de simples comparaisons d'égalité, mais une logique d'intersection d'intervalles temporels.

Considérons une nouvelle réservation définie par l'intervalle

```
$[Debut_{nouveau}, Fin_{nouveau}]$ et une réservation existante en base $[Debut_{existant}, Fin_{existant}]$.
Un conflit survient si et seulement si la condition suivante est vraie :
$$(Debut_{existant} < Fin_{nouveau}) \text{ ET } (Fin_{existant} > Debut_{nouveau})$$
```

Cette condition couvre tous les cas de chevauchement possibles :

1. La nouvelle réservation commence pendant une réservation existante.
2. La nouvelle réservation se termine pendant une réservation existante.
3. La nouvelle réservation englobe totalement une réservation existante.
4. La nouvelle réservation est totalement incluse dans une réservation existante.
5.

6.2 Implémentation technique et Atomicité

La vérification est effectuée directement au niveau de la base de données via une requête JPQL optimisée pour garantir la performance :

Gestion de la concurrence : La vérification et l'insertion sont regroupées au sein d'une seule transaction JPA. Si le compteur de conflits est supérieur à zéro, la transaction subit un rollback, garantissant qu'aucune donnée incohérente ne soit écrite.

Optimisation Fetch Joins : Pour éviter le problème du 'Lazy Loading' (N+1 queries) lors de l'affichage des résultats, des clauses FETCH JOIN sont utilisées dans les requêtes pour récupérer les objets User et Room liés en une seule opération.

6.3 Système de suggestion de salles alternatives

En cas de conflit détecté, le système ne se contente pas d'afficher un message d'erreur. Il propose une intelligence métier supplémentaire en suggérant des salles alternatives disponibles.

1. Le système identifie le type de salle demandé (ex: LARGE).
2. Il exécute une requête pour trouver toutes les salles de même type ou de capacité équivalente qui n'ont aucune réservation enregistrée pour le créneau % [startReq, endReq]% .
3. Ces suggestions sont renvoyées à l'utilisateur, améliorant considérablement l'expérience utilisateur (UX).

6.4 Scénario de Concurrence : Conflit entre Clients Distribués

Dans un système distribué, la gestion des accès concurrents est complexe. Imaginons le scénario suivant mettant en jeu deux types de clients différents :

1. Utilisateur (Client JavaFX) : Un employé sélectionne la salle 'A' (ID: 10) et s'apprête à cliquer sur 'Réserver'.
2. Administrateur (Client REST) : Simultanément, un administrateur décide que la salle 'A' est en travaux et envoie une requête DELETE /api/rooms/10 via l'API REST.

Processus de gestion de l'exception :

Étape 1 : L'administrateur valide la suppression. Le ReservationService supprime l'entité en base de données via Hibernate.

Étape 2 : L'utilisateur clique sur 'Réserver'. Son client envoie une requête au serveur pour la salle 10.

Étape 3 : Le serveur tente de récupérer la salle 10. Hibernate ne trouve plus l'entité et lève une EntityNotFoundException.

Étape 4 (Propagation) : La couche service intercepte cette exception technique et la transforme en une exception métier personnalisée, par exemple RoomUnavailableException, qui est renvoyée au client JavaFX via le protocole RMI ou localement.

Étape 5 (UI Feedback) : L'interface JavaFX intercepte l'erreur et affiche une alerte contextuelle à l'utilisateur : *'Désolé, cette salle n'existe plus ou a été supprimée par un administrateur'*.

7. Gestion des Rôles et Persistance

7.1 Système de Permissions (Admin vs User)

Le système distingue les utilisateurs selon leur rôle, défini par un attribut booléen admin dans l'entité User.

Rôle	Fonctionnalités Autorisées	Interface (FXML)
Utilisateur	Consulter ses réservations, créer/annuler ses propres créneaux, recevoir des notifications TCP.	user_dashboard.fxml
Administrateur	Gestion complète (CRUD) des salles (nom, type, localisation, équipements), vue globale de toutes les réservations du système.	admin_dashboard.fxml

L'authentification s'effectue au démarrage via LoginController, qui vérifie les identifiants en base de données avant de rediriger l'utilisateur vers le tableau de bord approprié selon son statut.

7.2 Couche de Persistance avec JPA/Hibernate

La persistance est le socle de l'application, assurant que les données survivent au redémarrage des services.

Fournisseur Hibernate : Utilisé comme implémentation de JPA pour mapper automatiquement les objets Java aux tables MySQL (users, rooms, reservations).

Configuration (persistence.xml) : Le fichier définit les paramètres de connexion à MySQL et la propriété hibernate.hbm2ddl.auto sur update, permettant de mettre à jour le schéma de la base sans perdre les données existantes.

Gestion des Transactions : Chaque opération sensible (création de réservation, suppression de salle) est encapsulée dans une transaction gérée par l' EntityManager. Cela garantit le principe ACID (Atomicité, Cohérence, Isolation, Durabilité).

Optimisation par Fetch Joins : Pour éviter les erreurs de type LazyInitializationException lors de l'accès à distance, les requêtes JPQL utilisent JOIN FETCH. Par exemple, lors de la récupération d'une réservation, on charge immédiatement l'utilisateur et la salle associés en une seule requête SQL.

8. Modèle de Données Enrichi

Conformément aux dernières mises à jour du projet, les entités ont été enrichies pour plus de réalisme :

User : Ajout du champ email pour les futures notifications.

Room : Ajout des champs location (étage, bâtiment) et description (détails techniques) pour aider l'utilisateur dans son choix.

9. Guide de Déploiement du Système

Le déploiement repose sur une architecture client-serveur où la base de données, la logique métier et les interfaces de communication sont réparties sur différents composants.

9.1 Prérequis Techniques

Avant de commencer, assurez-vous que les éléments suivants sont installés sur votre machine :

Java JDK 17+ (nécessaire pour JavaFX et les services JAX-RS/WS).

MySQL Server 8.0+.

Maven (pour la gestion des dépendances et le build du projet).

Pilote JDBC MySQL (inclus généralement dans les dépendances Hibernate).

9.2 Étape 1 : Configuration de la Base de Données

Le système utilise JPA/Hibernate pour gérer la persistance dans MySQL.

1. *Création de la base : Connectez-vous à votre instance MySQL et créez la base de données nommée meeting_rooms_db.*
2. *Configuration du projet : Modifiez le fichier src/META-INF/persistence.xml pour renseigner vos identifiants locaux :*

3. *Initialisation du schéma : Grâce à la propriété hibernate.hbm2ddl.auto sur update, les tables users, rooms et reservations seront créées automatiquement lors du premier lancement du serveur.*

9.3 Étape 2 : Lancement de l'Application Serveur

L'Application Server centralise la couche business (ReservationService) et expose les interfaces distantes.

Serveur de Notifications (TCP) : Lancez le NotificationServer pour ouvrir le port d'écoute des sockets.

Services Web (REST & SOAP) : Exécutez le SoapPublisher et le serveur embarqué JAX-RS pour rendre les points d'entrée accessibles via HTTP.

RMI Registry : Démarrez le registre RMI (généralement via RmiServer) pour enregistrer l'objet distant ReservationRmiService.

9.4 Étape 3 : Lancement des Clients

Une fois le serveur opérationnel, les clients peuvent se connecter via les différents protocoles :

1. *Client JavaFX Principal : Exécutez la classe ui.MainApp. Utilisez les identifiants par défaut admin / admin pour accéder au tableau de bord d'administration.*
2. *Notification Client : Lancez le NotificationClient pour visualiser en temps réel les mises à jour de réservations via le flux TCP.*
3. *Tests des API : Vous pouvez tester l'API REST via un navigateur ou un outil comme Postman à l'adresse configurée (ex: `http://localhost:8080/api/reservations`).*

9 5. Évaluation et Analyse Comparative

Protocole	Avantage	Inconvénient	Usage Idéal
REST	Léger, universel	Sans état (stateless)	Web / Mobile
SOAP	Sécurisé, Typé	Lourd (XML)	B2B / Banque
RMI	Très rapide	Java uniquement	Cluster de serveurs
Sockets	Temps réel	Connexion maintenue	Notifications

10. Annexes Techniques : Extraits de Code Clés

10.1 Centralisation de la logique (ReservationService)

Voici un extrait illustrant comment le service gère la création de réservation avec le contrôle de conflit :

```java
// Extrait simplifié de ReservationService.java
@Transactional
public Reservation createReservation(User user, Room room, LocalDateTime start, LocalDateTime end) {
    // Requête JPQL pour détecter les intersections temporelles
    Long conflicts = em.createQuery(
        "SELECT count(r) FROM Reservation r WHERE r.room = :room " +
        "AND r.startDateTime < :end AND r.endDateTime > :start", Long.class)
        .setParameter("room", room)
        .setParameter("start", start)
        .setParameter("end", end)
        .getSingleResult();

    if (conflicts > 0) {
        throw new ReservationConflictException("La salle est déjà occupée.");
    }

    Reservation res = new Reservation(user, room, start, end);
    em.persist(res);
    return res;
}
10.2 Configuration de la Persistance
Exemple de configuration persistence.xml utilisé pour lier Hibernate à MySQL :
XML
<persistence-unit name="MeetingRoomPU">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
        <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
        <property name="hibernate.hbm2ddl.auto" value="update"/>
        <property name="hibernate.show_sql" value="true"/>
    </properties>
</persistence-unit>
```

11. Sécurité du Système et Intégrité des Données

Dans sa version initiale, le système privilégiait la connectivité. Cependant, pour un déploiement en production, la sécurité doit être traitée de manière transversale, de la base de données jusqu'aux interfaces de communication.

11.1 Protection des identifiants : Le Hachage BCrypt

Actuellement, le système stocke les mots de passe en clair, ce qui constitue une vulnérabilité critique. La solution préconisée est l'intégration de l'algorithme de hachage BCrypt.

Principe du Sel (Salting) : Contrairement au hachage simple (comme MD5), BCrypt génère un 'sel' aléatoire pour chaque utilisateur. Même si deux utilisateurs ont le même mot de passe, leurs empreintes en base de données seront différentes.

Résistance aux attaques par dictionnaire : BCrypt est volontairement 'lent' à calculer, ce qui rend les attaques par force brute (brute-force) techniquement irréalisables sur un serveur standard.

Implémentation dans ReservationService : Lors de l'enregistrement, la méthode registerUser n'enregistre jamais le mot de passe direct, mais le résultat de BCrypt.hashpw(password, BCrypt.gensalt()). Lors du login, on utilise BCrypt.checkpw pour valider la correspondance.

11.2 Sécurisation de l'API REST : Authentification par Token (JWT)9. Évaluation et Analyse Comparative

Protocole	Avantage	Inconvénient	Usage Idéal
REST	Léger, universel	Sans état (stateless)	Web / Mobile
SOAP	Sécurisé, Typé	Lourd (XML)	B2B / Banque
RMI	Très rapide	Java uniquement	Cluster de serveurs
Sockets	Temps réel	Connexion maintenue	Notifications

L'exposition d'une API JAX-RS sans protection est un risque majeur. L'implémentation de JSON Web Tokens (JWT) est proposée pour sécuriser les échanges.

1. Le processus (Handshake) :

   Le client envoie ses identifiants au point d'entrée /api/auth/login.

Le serveur valide les identifiants et génère un Token signé numériquement contenant le rôle de l'utilisateur (Admin/User) et une date d'expiration.

Le client stocke ce token et l'envoie dans l'en-tête Authorization: Bearer <token| pour chaque requête suivante.

2. Avantages en système distribué :

Stateless : Le serveur n'a pas besoin de stocker de session en mémoire, ce qui facilite le passage à l'échelle (scalability).

Intégrité : Toute modification du token par un tiers invalide la signature, bloquant immédiatement l'accès.

11.3 Sécurisation de RMI et SSL/TLS

Pour les communications natives Java RMI, la sécurité peut être renforcée par l'utilisation de SslRMIClientSocketFactory et SslRMIServerSocketFactory. Cela permet de chiffrer l'intégralité du flux binaire circulant sur le réseau, empêchant ainsi toute interception de données sensibles (sniffing) entre le client JavaFX et le serveur d'application.

12. Conclusion Générale

Le projet de Système de Réservation de Salles de Réunion a permis d'explorer les défis liés à la conception d'applications distribuées modernes. En centralisant la logique métier dans une couche de service unique (ReservationService), nous avons réussi à exposer des fonctionnalités complexes de manière cohérente à travers quatre protocoles majeurs : REST, SOAP, RMI et TCP Sockets.

Bilans techniques

Interopérabilité : L'utilisation combinée de JAX-RS et JAX-WS démontre la capacité du système à communiquer avec des clients web légers tout comme avec des infrastructures d'entreprise lourdes.

Performance : L'optimisation de la couche de persistance JPA/Hibernate via les Fetch Joins a permis de maintenir une interface fluide (JavaFX) malgré la complexité des relations entre utilisateurs, salles et réservations.

Fiabilité : L'algorithme de détection de conflits, couplé à la gestion des transactions SQL, assure l'intégrité absolue des données, même lors d'accès concurrents massifs.

Perspectives d'avenir

Le système est conçu pour être évolutif. L'étape suivante consisterait à conteneuriser chaque service (REST, Notification, DB) via Docker pour permettre un déploiement sur le Cloud, et à intégrer un calendrier interactif dans l'interface JavaFX pour améliorer l'expérience utilisateur.