École Marocaine des Sciences de l'Ingénieur

EMSI – Marrakech

Projet de Fin de Semestre

Gestion Intelligente de Gare Routière

Filière :	Développement Digital et Systèmes d'Information
Méthodologie :	Agile Scrum
Technologies :	Spring Boot / Next.js / MySQL / Docker
Module :	Test qualité logiciel Dev

Réalisé par :

KOUBRI Hatim

LAYHI Rayan

Encadrant pédagogique :

Dr. Driss ESSABBAR

Année universitaire 2025–2026

Dédicace

Nous dédions ce modeste travail à nos parents pour leurs grands sacrifices,

leur soutien indéfectible et leurs encouragements tout au long

de notre parcours académique.

À nos familles et nos amis pour leur amour et leur présence permanente.

Aux professeurs de l'EMSI Marrakech qui ont consacré leur expertise

et leur disponibilité pour nous accompagner dans notre formation.

À tous ceux qui ont contribué de près ou de loin à l'élaboration de ce rapport.

Remerciements

C'est avec une grande satisfaction que nous souhaitons remercier toutes les personnes qui nous ont aidés, accompagnés et soutenus pour l'aboutissement de ce projet de fin de semestre.

Nous adressons en premier lieu nos sincères remerciements à notre encadrant pédagogique, **Dr. Driss ESSABBAR**, pour son accompagnement régulier, ses orientations techniques précieuses, ses encouragements et ses conseils avantageux tout au long de la réalisation de ce projet.

Nous remercions également l'ensemble du corps professoral de l'**EMSI Marrakech** pour la qualité de la formation dispensée durant notre cursus en Ingénierie Informatique et Réseaux, et plus particulièrement pour les enseignements relatifs aux systèmes distribués qui ont directement inspiré l'architecture de ce projet.

Nos remerciements s'adressent aux membres du jury pour l'intérêt porté à notre travail ainsi que pour le temps consacré à l'évaluation de ce projet.

Enfin, nous remercions nos familles et nos proches pour leur soutien moral et leurs encouragements permanents tout au long de notre parcours.

Résumé

Dans le cadre du Projet de Fin de Semestre réalisé à l'EMSI Marrakech, nous avons conçu et développé une plateforme intelligente de gestion de gare routière reposant sur une architecture distribuée moderne basée sur Spring Boot, Next.js, MySQL et Docker.

L'objectif principal du projet consiste à digitaliser les processus métiers liés à la gestion des trajets, des réservations, des chauffeurs, des voyageurs, des compagnies de transport ainsi que des opérations de contrôle et de supervision d'une gare routière moderne. Le système développé intègre plusieurs modules avancés, notamment l'authentification sécurisée par JWT, la gestion des rôles, la réservation multi-passagers, l'attribution intelligente des sièges, la génération de tickets PDF avec QR Code, les notifications temps réel via WebSocket, la reconnaissance OCR des plaques d'immatriculation, et les mécanismes de tarification dynamique.

Sur le plan des systèmes distribués, le projet implémente une communication synchrone via REST et RMI, une communication asynchrone via ActiveMQ et JMS, des mécanismes de résilience avec Resilience4j, la pagination des données ainsi qu'une architecture orientée services (SOA partielle), permettant ainsi de couvrir les principaux concepts du module Systèmes Distribués. Le projet a été conduit selon une méthodologie Agile Scrum organisée en sept sprints assurant une évolution progressive et une validation continue des fonctionnalités. Des tests automatisés avec JUnit 5, Selenium, et une analyse de la qualité du code avec SonarQube garantissent la robustesse et la maintenabilité de la solution.

Mots clés : Spring Boot, Next.js, JWT, ActiveMQ, RMI, Resilience4j, OCR, WebSocket, Docker, QR Code, Agile Scrum, Systèmes Distribués, SonarQube.

Informations du rapport :

- **Compte de mots :** 19 947 mots
- **Nombre de pages :** 81 pages
- **Github :** https://github.com/hatim-koubri/gare-routiere-intelligente/tree/develop

Abstract

As part of the academic semester project carried out at EMSI Marrakech, we designed and developed an intelligent bus station management platform based on a modern distributed architecture using Spring Boot, Next.js, MySQL and Docker.

The main objective of the project is to digitalize business processes related to trip management, reservations, drivers, passengers, transport companies and supervision operations within a modern bus station. The developed system integrates several advanced modules, including JWT authentication, role management, multi-passenger booking, intelligent seat allocation, PDF ticket generation with QR Codes, real-time WebSocket notifications, OCR license plate recognition, and dynamic pricing mechanisms.

From a distributed systems perspective, the project implements synchronous communication via REST and RMI, asynchronous communication via ActiveMQ and JMS, resilience mechanisms with Resilience4j, data pagination, and a service-oriented architecture (partial SOA), covering the main concepts of the Distributed Systems module. The project was conducted using the Agile Scrum methodology organized into seven sprints, ensuring progressive delivery and continuous validation of features. Automated tests with JUnit 5, Selenium, and code quality analysis with SonarQube guarantee the quality and robustness of the solution.

Keywords : Spring Boot, Next.js, JWT, ActiveMQ, RMI, Resilience4j, OCR, WebSocket, Docker, QR Code, Agile Scrum, Distributed Systems, SonarQube.

Report information :

- **Word count :** 19,947 words
- **Number of pages :** 81 pages
- **Github :** https://github.com/hatim-koubri/gare-routiere-intelligente/tree/develop

Glossaire

Terme / Acronyme	**Description**
API	Application Programming Interface – Interface de programmation permettant la communication entre services
ActiveMQ	Broker de messages open source implémentant le protocole JMS
Circuit Breaker	Mécanisme de résilience qui interrompt les appels vers un service défaillant
DTO	Data Transfer Object – Objet de transfert de données entre les couches applicatives
Docker	Plateforme de conteneurisation permettant l'isolation des services
E2E	End-to-End Testing – Tests de bout en bout simulant le comportement utilisateur
HTTP	Hypertext Transfer Protocol – Protocole de communication web
IHM	Interface Homme-Machine
JMS	Java Message Service – API de messagerie pour la communication asynchrone Java
JPA	Java Persistence API – Interface Java pour la persistance des données
JWT	JSON Web Token – Standard de sécurité pour l'authentification stateless
MOM	Message Oriented Middleware – Middleware orienté messages
MySQL	Système de gestion de base de données relationnelle open source
Next.js	Framework React pour le développement d'applications web modernes

Terme / Acronyme	**Description**
OCR	Optical Character Recognition – Reconnaissance optique de caractères
ORM	Object Relational Mapping – Technique de correspondance objets/tables
QR Code	Quick Response Code – Code-barres 2D contenant des informations encodées
RBAC	Role Based Access Control – Contrôle d'accès basé sur les rôles
REST	Representational State Transfer – Style d'architecture pour services web
Resilience4j	Bibliothèque de résilience pour applications Java (Retry, Circuit Breaker, Fallback)
RMI	Remote Method Invocation – Mécanisme Java d'invocation de méthodes distantes
RPC	Remote Procedure Call – Appel de procédure distante
Selenium	Outil d'automatisation des tests d'interfaces web via navigateur
SOA	Service Oriented Architecture – Architecture Orientée Services
SonarQube	Plateforme d'analyse continue de la qualité et de la sécurité du code source
Spring Boot	Framework Java facilitant le développement d'applications Spring
UML	Unified Modeling Language – Langage de modélisation unifié
WebSocket	Protocole de communication bidirectionnelle temps réel

Table des matières

Dédicace **1**

Remerciements **2**

Résumé **3**

Abstract **4**

Glossaire **5**

Introduction Générale **13**

1 Contexte Général du Projet **14**

Introduction 14

1.1 Présentation du projet 14

1.2 Vue d'ensemble du système 14

1.2.1 Schéma d'ensemble du système 15

1.3 Étude de l'existant 16

1.4 Problématique 17

1.5 Solution proposée 18

1.6 Enjeux du projet 18

1.7 Méthodologie Agile Scrum 19

1.7.1 Présentation de la méthode 19

1.7.2 Organisation des sprints 20

Conclusion 20

2 Analyse et Conception **21**

Introduction 21

2.1 Analyse des besoins fonctionnels 21

2.2 Analyse des besoins non fonctionnels 22

2.3 Backlog produit 23

2.4 Diagrammes de cas d'utilisation 24

2.4.1 Cas d'utilisation : Administrateur 24
2.4.2 Cas d'utilisation : Voyageur 25
2.4.3 Cas d'utilisation : Chauffeur 26
2.4.4 Cas d'utilisation : Responsable Compagnie 27
2.5 Diagramme BPMN : Processus global de réservation et embarquement . . . 29
2.6 Diagramme de séquence : Processus de réservation 30
2.7 Diagramme de séquence : Détection OCR et attribution de quai 31
2.8 Diagramme de classes 33
Conclusion 35

3 Architecture Technique et Technologies Utilisées 36
Introduction 36
3.1 Architecture générale du système 36
3.2 Architecture Backend – Spring Boot 38
3.3 Systèmes Distribués – Architecture et Communication 39
3.3.1 Vue d'ensemble des communications distribuées 40
3.3.2 Communication Synchrone – API REST 41
3.3.3 Sécurité – JWT et Spring Security 43
3.3.4 Communication Asynchrone – ActiveMQ et JMS 45
3.3.5 Invocation de Méthodes Distantes – RMI 46
3.3.6 Résilience – Resilience4j 48
3.3.7 Pagination REST 49
3.3.8 Notifications temps réel – WebSocket et STOMP 50
3.4 Technologies utilisées 52
Conclusion 53

4 Réalisation du Projet 54
Introduction 54
4.1 Sprint 1 : Infrastructure et Authentification 54
4.1.1 Structure du projet Backend 54
4.2 Sprint 2 : Gestion Administrateur et Dashboard 56
4.3 Sprint 3 : Module OCR et Gestion Chauffeur 57
4.4 Sprint 4 : Réservation Voyageur et Tickets 58
4.5 Sprint 5 : Paiement et Dashboard Financier 59
4.6 Sprint 6 : Gestion Responsable Compagnie 59
4.7 Sprint 7 : Tests, Qualité du Code et Optimisation 60
4.7.1 Tests unitaires avec JUnit 5 et Mockito 61
4.7.2 Tests d'automatisation UI avec Selenium 62
4.7.3 Analyse qualité du code – SonarQube et SonarCloud 64
4.7.4 Optimisations réalisées 65

4.8 Interfaces de l'application 66
4.8.1 Interface d'authentification 66
4.8.2 Dashboard Administrateur 66
4.8.3 Module de réservation Voyageur 67
4.8.4 Interface de paiement sécurisé 68
4.8.5 Interface du ticket électronique et QR Code 69
4.8.6 Module OCR – Reconnaissance de plaques 70
4.8.7 Interface Chauffeur – Validation d'embarquement 71
4.8.8 Dashboard Responsable Compagnie 72
4.8.9 Notifications temps réel 72
Conclusion 73

Conclusion Générale **74**

Bibliographie et Webographie **75**

Annexes **78**

Table des figures

1.1 Vue d'ensemble des acteurs et modules de la plateforme 15
1.2 Cycle de développement Agile Scrum 19

2.1 Diagramme de cas d'utilisation de l'administrateur 24
2.2 Diagramme de cas d'utilisation du voyageur 25
2.3 Diagramme de cas d'utilisation du chauffeur 26
2.4 Diagramme de cas d'utilisation du responsable compagnie 28
2.5 Diagramme BPMN du processus de réservation et embarquement 29
2.6 Diagramme de séquence du processus de réservation 31
2.7 Diagramme de séquence : Détection OCR et attribution automatique de quai 32
2.8 Diagramme de classes du système 33

3.1 Architecture générale multicouche du système 37
3.2 Architecture interne du backend Spring Boot 38
3.3 Schéma global des communications distribuées du système 40
3.4 Flux d'une requête REST sécurisée par JWT 42
3.5 Architecture d'authentification JWT avec Spring Security 44
3.6 Architecture de communication asynchrone avec ActiveMQ et JMS 45
3.7 Architecture RMI – Invocation distante pour la tarification 47
3.8 Mécanismes de résilience : Retry, Circuit Breaker et Fallback 48
3.9 Principe de pagination REST avec Spring Pageable 50
3.10 Architecture WebSocket STOMP – Notifications temps réel 51
3.11 Technologies utilisées dans le projet 53

4.1 Structure du projet Backend Spring Boot 55
4.2 Scénarios de tests automatisés avec Selenium 62
4.3 Tableau de bord SonarCloud – Analyse qualité du projet 65
4.4 Interface d'authentification – Page de connexion 66
4.5 Dashboard Administrateur – Vue de supervision globale 67
4.6 Interface de réservation – Recherche de trajet et sélection des sièges 68
4.7 Interface de paiement sécurisé 69

4.8 Interface du ticket électronique avec QR Code 70
4.9 Module OCR – Détection automatique de plaque et attribution de quai . . 71
4.10 Interface chauffeur – Validation des QR Codes 71
4.11 Dashboard Responsable Compagnie – Gestion opérationnelle 72
4.12 Interface des notifications temps réel 73

Liste des tableaux

1.1 Acteurs principaux du système 15
1.2 Comparaison des solutions existantes 17
1.3 Organisation des sprints du projet 20

2.1 Principaux besoins fonctionnels par module 22
2.2 Extrait du Product Backlog – User Stories principales 23

3.1 Exemples d'endpoints REST principaux 42

Introduction Générale

La transformation numérique des infrastructures de transport représente aujourd'hui un enjeu stratégique majeur pour améliorer l'efficacité opérationnelle, la qualité des services offerts aux usagers et l'expérience globale des voyageurs. Dans ce contexte, les gares routières jouent un rôle central dans l'organisation des déplacements interurbains et la gestion des flux de voyageurs et de compagnies de transport.

Cependant, de nombreux processus liés à la gestion des gares routières restent encore partiellement manuels. La gestion des réservations, l'attribution des quais, le suivi des trajets, le contrôle des bagages et la coordination entre les différents acteurs font partie des opérations qui nécessitent une automatisation et une digitalisation profondes.

C'est dans ce contexte que s'inscrit le projet « Gestion Intelligente de la Gare Routière », développé dans le cadre du Projet de Fin de Semestre à l'EMSI Marrakech. Ce projet propose une plateforme web moderne permettant la gestion centralisée et automatisée des différentes opérations d'une gare routière intelligente.

Sur le plan technique, la solution développée repose sur une architecture distribuée combinant un backend Spring Boot sécurisé par JWT, un frontend Next.js interactif, une base de données MySQL conteneurisée via Docker. L'architecture intègre également plusieurs composants des systèmes distribués modernes, notamment la communication asynchrone via ActiveMQ, l'invocation de méthodes distantes via RMI et les mécanismes de résilience via Resilience4j.

Le présent rapport est structuré en quatre chapitres principaux. Le premier chapitre présente le contexte général du projet, les problématiques identifiées et la méthodologie Agile Scrum adoptée. Le deuxième chapitre est consacré à l'analyse des besoins et à la conception du système à travers les diagrammes UML et le backlog produit. Le troisième chapitre détaille l'architecture technique et les technologies utilisées, en accordant une attention particulière aux aspects des systèmes distribués. Enfin, le quatrième chapitre présente la réalisation du projet à travers les différents sprints, les tests effectués et les interfaces développées.

Chapitre 1

Contexte Général du Projet

Introduction

Ce chapitre présente le cadre général dans lequel s'inscrit le projet de Gestion Intelligente de la Gare Routière. Nous y exposons la présentation du projet, l'étude de l'existant, les problématiques identifiées, la solution proposée ainsi que la méthodologie de conduite du projet adoptée.

1.1 Présentation du projet

Le projet « Gestion Intelligente de la Gare Routière » est une plateforme web développée dans le cadre du projet de fin de semestre à l'EMSI Marrakech en filière Ingénierie Informatique et Réseaux. L'objectif principal du système consiste à digitaliser les opérations liées à la gestion d'une gare routière moderne en proposant une solution centralisée permettant la gestion des voyageurs, des chauffeurs, des compagnies de transport, des trajets, des réservations, des quais ainsi que des opérations de contrôle et de supervision.

La plateforme repose sur une architecture distribuée moderne basée sur Spring Boot pour le backend, Next.js pour le frontend, MySQL pour la persistance des données et Docker pour la conteneurisation des services. Elle intègre plusieurs mécanismes de communication distribués tels que REST, ActiveMQ, RMI et WebSocket, permettant ainsi d'assurer la robustesse, la scalabilité et la résilience du système.

1.2 Vue d'ensemble du système

Le système développé s'articule autour de cinq acteurs principaux qui interagissent avec les différents modules de la plateforme, comme synthétisé dans le tableau 1.1.

TABLE 1.1 – Acteurs principaux du système

Acteur	**Rôle et responsabilités**
Administrateur	Assure la supervision globale de la gare, gère les quais, les annonces, les opérations OCR et les tableaux de bord financiers
Voyageur	Effectue les réservations, consulte les trajets, modifie ses tickets et reçoit des notifications en temps réel
Chauffeur	Consulte ses trajets, valide les tickets QR Code, déclare les départs et gère les bagages
Responsable compagnie	Gère les bus, les lignes, les trajets, les chauffeurs, les promotions et les statistiques de sa compagnie
Système OCR	Détecte automatiquement les plaques d'immatriculation et attribue les quais disponibles

1.2.1 Schéma d'ensemble du système

La figure suivante présente une vue d'ensemble des interactions entre les différents acteurs et les modules principaux de la plateforme.

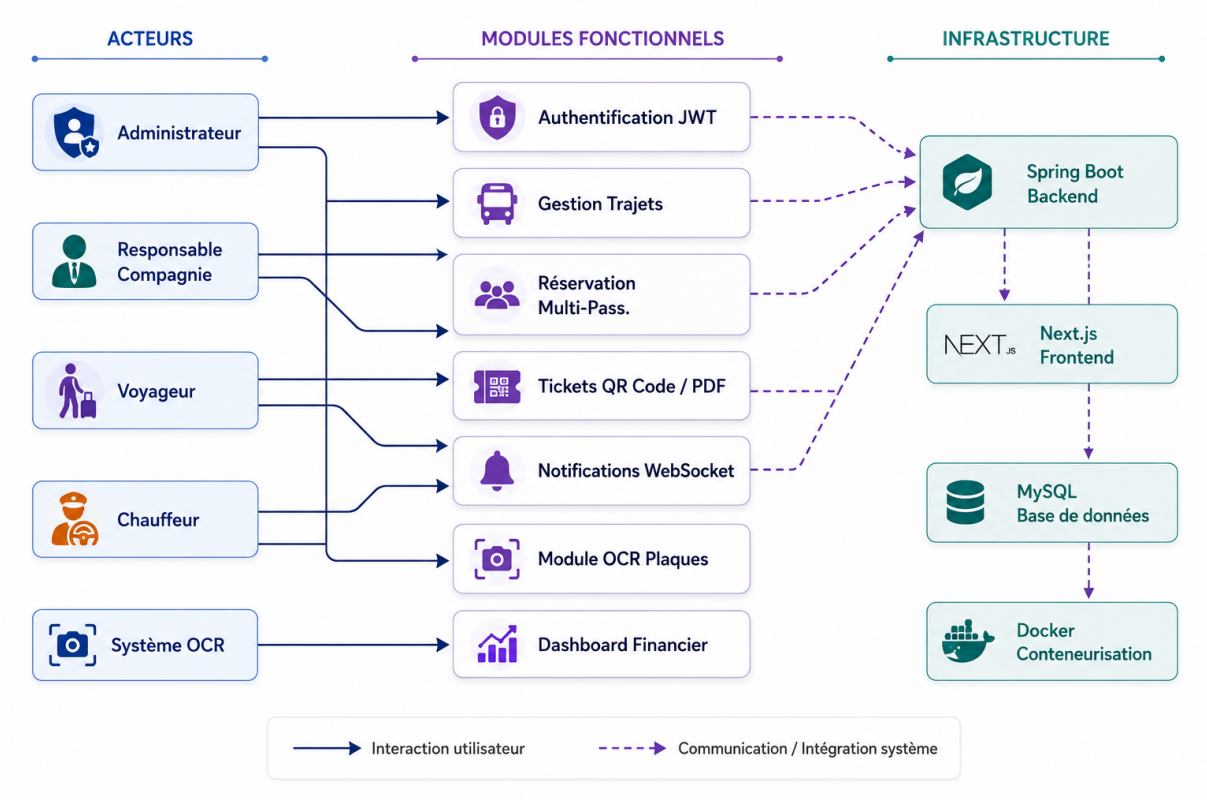


FIGURE 1.1 – Vue d'ensemble des acteurs et modules de la plateforme

La figure 1.1 illustre l'organisation globale de la plateforme. À gauche figurent les

cinq acteurs du système qui interagissent avec les modules métier centraux. Ces modules, représentés au centre, couvrent les principales fonctionnalités de la gare routière, allant de l'authentification jusqu'au tableau de bord financier. À droite, l'infrastructure technique assure le traitement et la persistance des données via Spring Boot, Next.js, MySQL et Docker. Les flèches en pointillés indiquent les dépendances techniques entre les modules et l'infrastructure sous-jacente.

1.3 Étude de l'existant

Afin de mieux cerner les besoins du système à développer, nous avons analysé les solutions existantes sur le marché marocain et international. Cette étude nous a permis d'identifier les forces et les limites des plateformes actuelles, et de définir les axes d'amélioration de notre solution.

CTM (Compagnie de Transport au Maroc) est la principale compagnie de transport routier au Maroc. Son application mobile et son site web permettent de réserver des billets de bus pour différentes destinations, de vérifier les horaires et la disponibilité des sièges, et de gérer les réservations. Cependant, l'application CTM ne propose pas de gestion côté gare (attribution de quais, supervision des chauffeurs, gestion OCR), et se limite à la relation avec le voyageur final. La gestion interne de la gare reste manuelle ou partiellement informatisée [16].

Markoub.ma est une plateforme marocaine de réservation de billets d'autocar permettant de comparer et réserver auprès de plusieurs compagnies. Elle offre un service de réservation en ligne, la comparaison de compagnies et la réception de billets électroniques. Toutefois, elle ne dispose d'aucun module de gestion de gare, de contrôle des quais ou de supervision des chauffeurs, et se positionne uniquement comme agrégateur de billets [17].

Digiparc est une solution marocaine de gestion de flotte et de transport proposant la géolocalisation GPS, le suivi des trajets et la gestion de la maintenance. Elle cible principalement les entreprises de transport de marchandises et ne couvre pas les besoins spécifiques d'une gare routière de voyageurs tels que la gestion des quais, les réservations multi-passagers ou la reconnaissance OCR [18].

Busbud est une plateforme internationale de réservation de bus présente dans plus de 80 pays. Elle permet la comparaison et la réservation de trajets interurbains à l'échelle mondiale. Bien qu'elle offre une expérience voyageur complète, elle ne propose aucune fonctionnalité de gestion opérationnelle de gare routière, restant exclusivement orientée vers l'achat de billets [19].

Le tableau 1.2 présente une synthèse comparative de ces solutions existantes par rapport à notre plateforme.

TABLE 1.2 – Comparaison des solutions existantes

Fonctionnalité	**CTM**	**Markoub**	**Digiparc**	**Busbud**	**Notre plateforme**
Réservation en ligne	✓	✓	✗	✓	✓
Gestion des quais	✗	✗	✗	✗	✓
Reconnaissance OCR	✗	✗	✗	✗	✓
Notifications temps réel	✗	✗	Partiel	✗	✓
Gestion chauffeurs	✗	✗	Partiel	✗	✓
Tickets QR Code PDF	✓	✓	✗	✓	✓
Architecture distribuée	✗	✗	✗	✗	✓
Dashboard financier	✗	✗	Partiel	✗	✓
Tarification dynamique	✗	✗	✗	Partiel	✓

Cette analyse comparative révèle que les solutions existantes sur le marché marocain et international se concentrent principalement sur la relation avec le voyageur final (réservation et billetterie), sans offrir de plateforme intégrée permettant la gestion opérationnelle complète d'une gare routière. C'est précisément ce vide fonctionnel que notre solution ambitionne de combler.

1.4 Problématique

Les systèmes classiques de gestion des gares routières présentent plusieurs limitations techniques et organisationnelles qui impactent directement la qualité des services et la fluidité des opérations. Les réservations sont souvent réalisées au guichet, engendrant de longues files d'attente et une faible traçabilité des transactions. La gestion des quais ainsi que la coordination entre les chauffeurs, les voyageurs et les compagnies de transport représentent des tâches complexes nécessitant une supervision humaine permanente. Par ailleurs, l'absence de solution centralisée limitant la traçabilité des opérations, la gestion des notifications en temps réel et le contrôle automatique des accès fragilise l'ensemble du système. L'absence d'outils analytiques et de tableaux de bord réduit également la capacité de supervision et de prise de décision des responsables. On peut ainsi résumer les problèmes majeurs identifiés : l'absence de supervision temps réel des opérations, les

difficultés dans la traçabilité des réservations, les conflits dans l'attribution des quais, le manque de coordination entre acteurs, la faible automatisation des traitements récurrents, les problèmes de sécurité des accès, et l'absence d'analyse statistique centralisée.

1.5 Solution proposée

Afin de répondre à l'ensemble des problématiques identifiées, nous avons conçu et développé une plateforme intelligente reposant sur une architecture distribuée moderne. La solution proposée s'appuie sur un backend Spring Boot exposant des APIs REST sécurisées par JWT, un frontend Next.js permettant une interaction fluide avec les utilisateurs, et une base de données MySQL conteneurisée via Docker. Elle intègre également des composants de systèmes distribués couvrant la communication synchrone (REST, RMI), la communication asynchrone (ActiveMQ/JMS), les mécanismes de résilience (Resilience4j) et les notifications temps réel (WebSocket).

Les fonctionnalités avancées développées incluent la reconnaissance OCR des plaques d'immatriculation pour l'attribution automatique des quais, la génération de tickets PDF avec QR Code pour le contrôle des voyageurs, les notifications temps réel via WebSocket pour tous les acteurs, la réservation multi-passagers avec choix des sièges sur plan interactif, la tarification dynamique basée sur le taux de remplissage, les tableaux de bord analytiques et financiers, et la gestion complète du cycle de vie des réservations (modification, annulation, remboursement).

1.6 Enjeux du projet

Le projet présente plusieurs enjeux techniques et fonctionnels majeurs. Sur le plan de la digitalisation, l'objectif est d'automatiser et de centraliser les opérations d'une gare routière pour améliorer la qualité du service rendu aux usagers. En matière de sécurité, il s'agit de garantir la confidentialité des données et la gestion sécurisée des accès via JWT et RBAC. Concernant la performance, le système doit assurer des temps de réponse acceptables malgré les traitements temps réel, OCR et les opérations concurrentes. La résilience constitue un autre enjeu majeur, en assurant la continuité du service même en cas de défaillance partielle grâce aux mécanismes de tolérance aux pannes. Enfin, la scalabilité impose de concevoir une architecture capable de supporter un nombre croissant d'utilisateurs et d'opérations simultanées.

1.7 Méthodologie Agile Scrum

1.7.1 Présentation de la méthode

Afin d'assurer une meilleure organisation du développement ainsi qu'une livraison progressive et continue des fonctionnalités, nous avons adopté la méthodologie Agile Scrum. Cette approche itérative permet de structurer le projet en plusieurs sprints successifs de durée fixe, facilitant ainsi le suivi de l'avancement, la validation continue des fonctionnalités développées et l'adaptation rapide aux nouveaux besoins.

La méthode Scrum favorise également une meilleure gestion des priorités grâce au Product Backlog, tout en assurant une amélioration continue du produit à travers les différentes cérémonies Agile telles que la planification, les réunions quotidiennes, les revues de sprint et les rétrospectives.

La figure suivante présente le cycle de développement Agile Scrum adopté dans le cadre de ce projet.

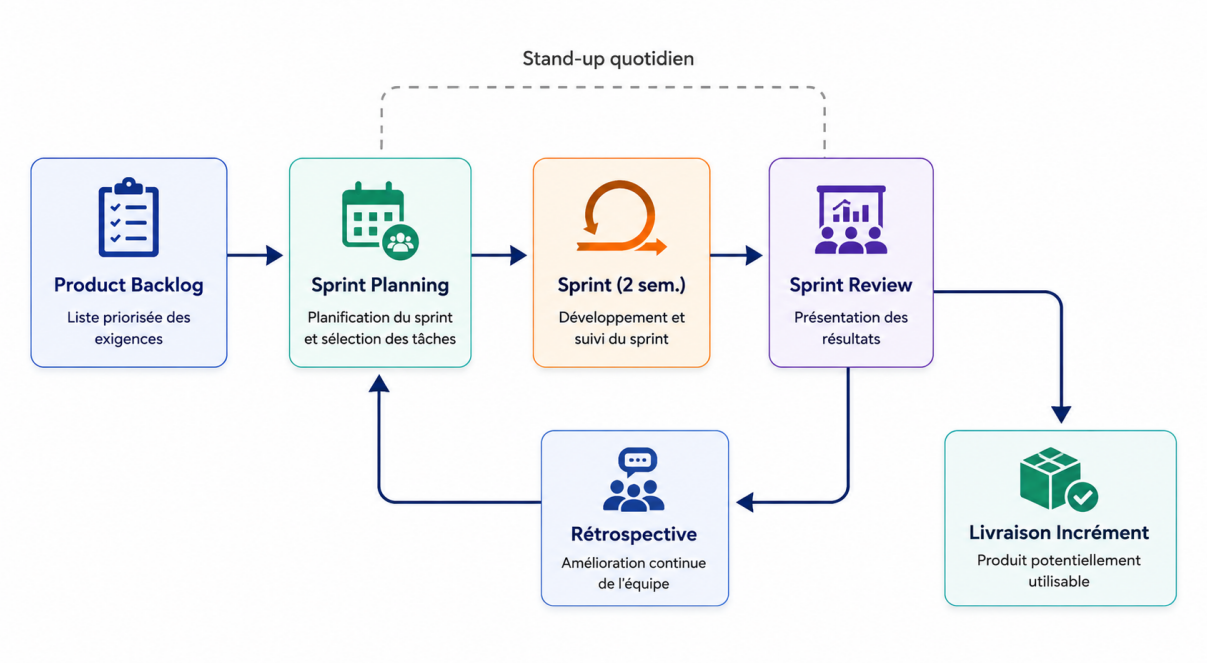


FIGURE 1.2 – Cycle de développement Agile Scrum

La figure 1.2 illustre le fonctionnement itératif de la méthodologie Scrum utilisée durant le développement de la plateforme. Le processus débute par la constitution du Product Backlog regroupant l'ensemble des fonctionnalités du système, priorisées selon leur importance métier et technique.

Chaque sprint commence par une phase de planification (Sprint Planning) durant laquelle les user stories à développer sont sélectionnées et estimées. Le développement est ensuite réalisé de manière collaborative tout en étant accompagné de réunions quotidiennes

(Daily Scrum) permettant de suivre l'avancement des tâches et d'identifier rapidement les éventuels blocages.

À la fin de chaque sprint, une revue (Sprint Review) est organisée afin de présenter l'incrément fonctionnel réalisé et de valider les fonctionnalités développées. Enfin, une rétrospective (Sprint Retrospective) permet d'analyser le déroulement du sprint dans une logique d'amélioration continue du processus de développement.

1.7.2 Organisation des sprints

Le projet a été réparti sur sept sprints couvrant l'ensemble des modules du système, comme présenté dans le tableau 1.3.

TABLE 1.3 – Organisation des sprints du projet

Sprint	**Fonctionnalités principales**	**Technologies clés**
Sprint 1	Infrastructure Docker, authentification JWT, gestion des rôles	Spring Boot, JWT, MySQL, Docker
Sprint 2	Dashboard administrateur, gestion des quais, annonces bilingues	REST API, Next.js
Sprint 3	Module OCR, gestion chauffeur, scan QR Code, gestion bagages	OpenCV, YOLOv8, ActiveMQ
Sprint 4	Réservation voyageur, choix sièges, tickets PDF, QR Code	iText, WebSocket
Sprint 5	Paiement, dashboard financier, tarification dynamique	REST, RMI, Resilience4j
Sprint 6	Gestion responsable compagnie, promotions, remboursements	Spring Security, JMS
Sprint 7	Tests, qualité SonarQube, optimisation, documentation	JUnit 5, Selenium, SonarQube

Conclusion

Dans ce chapitre, nous avons présenté le contexte général du projet de Gestion Intelligente de la Gare Routière. Nous avons exposé les problématiques liées aux systèmes classiques de gestion, analysé les solutions existantes sur le marché, présenté la solution distribuée proposée, les enjeux techniques majeurs ainsi que la méthodologie Agile Scrum adoptée pour la conduite du projet. Le chapitre suivant sera consacré à l'analyse des besoins et à la conception du système.

Chapitre 2

Analyse et Conception

Introduction

Ce chapitre présente le résultat du travail d'analyse et de conception du système. Nous y exposons les besoins fonctionnels et non fonctionnels, le backlog produit, les diagrammes de cas d'utilisation par acteur, les diagrammes de séquence des flux principaux ainsi que le diagramme de classes du système.

2.1 Analyse des besoins fonctionnels

Les besoins fonctionnels représentent l'ensemble des fonctionnalités que le système doit fournir pour répondre aux exigences des différents acteurs. Le tableau 2.1 synthétise les principaux modules fonctionnels du système.

TABLE 2.1 – Principaux besoins fonctionnels par module

Module	**Fonctionnalités**
Authentification	Inscription, connexion sécurisée, gestion des rôles (ADMIN, VOYAGEUR, CHAUFFEUR, RESPONSABLE)
Gestion trajets	Création, modification, planification des trajets et des lignes par les compagnies
Réservation	Réservation multi-passagers, choix des sièges, gestion des enfants, simulation tarifaire
Tickets	Génération QR Code, PDF, re-téléchargement, modification, annulation
Paiement	Paiement en ligne, gestion des remboursements, dashboard financier
OCR	Détection automatique des plaques, attribution des quais, facturation du stationnement
Notifications	Notifications push temps réel, rappels automatiques, alertes voyageurs et chauffeurs
Chauffeur	Consultation trajets, scan QR, validation arrêts, gestion bagages, déclaration départs
Administration	Supervision globale, gestion quais, annonces bilingues, statistiques, messagerie interne

2.2 Analyse des besoins non fonctionnels

En complément des fonctionnalités métier, le système doit satisfaire plusieurs exigences non fonctionnelles. En matière de sécurité, le système adopte une authentification stateless JWT, un contrôle d'accès RBAC, une protection contre les injections SQL et XSS, et un chiffrement des mots de passe via BCrypt. Concernant la performance, les temps de réponse doivent demeurer inférieurs à 500 ms pour les opérations critiques, avec une pagination des données volumineuses et une optimisation des requêtes JPA. La résilience est assurée par des mécanismes de Retry, Circuit Breaker et Fallback sur les services critiques. La scalabilité repose sur une architecture modulaire permettant l'évolution progressive du système. La disponibilité est garantie par la conteneurisation Docker assurant la portabilité et la cohérence des environnements. Le système propose également un support bilingue français/arabe avec gestion RTL/LTR pour l'internationalisation, et respecte une architecture multicouche (Controller, Service, Repository) basée sur les principes SOLID pour la maintenabilité.

2.3 Backlog produit

Le Product Backlog représente l'ensemble des user stories priorisées selon la méthode MoSCoW (Must, Should, Could). Le tableau 2.2 présente une sélection représentative des user stories principales.

TABLE 2.2 – Extrait du Product Backlog – User Stories principales

ID	User Story	Priorité	Sprint
US-01	S'inscrire et se connecter avec son rôle	Must	S1
US-02	Sécuriser les mots de passe avec BCrypt	Must	S1
US-20	Attribuer 5 quais fixes par compagnie	Must	S2
US-21	Créer des annonces bilingues (FR/AR)	Must	S2
US-25	Détecter un bus via caméra OCR (YOLOv8)	Must	S3
US-28	Attribuer automatiquement un quai après OCR	Must	S3
US-29	Notifier le chauffeur de son numéro de quai	Must	S3
US-35	Scanner le QR Code d'un voyageur à l'embarquement	Must	S3
US-45	Rechercher des trajets directs par ville et date	Must	S4
US-50	Faire une réservation multi-passagers	Must	S4
US-54	Voir une simulation tarifaire avant réservation	Must	S4
US-61	Recevoir un ticket PDF avec QR Code par email	Must	S4
US-64	Modifier la date/heure de son ticket	Must	S5
US-66	Annuler son ticket avec remboursement progressif	Must	S5
US-70	Recevoir des rappels automatiques avant le départ	Must	S5
US-86	Planifier les trajets (heures, arrêts, horaires)	Must	S6
US-93	Configurer la tarification dynamique	Must	S6
US-96	Gérer les remboursements des voyageurs	Must	S6

2.4 Diagrammes de cas d'utilisation

2.4.1 Cas d'utilisation : Administrateur

Le diagramme suivant présente les principales interactions entre l'administrateur et les différents modules fonctionnels de la plateforme intelligente de gestion de gare routière.

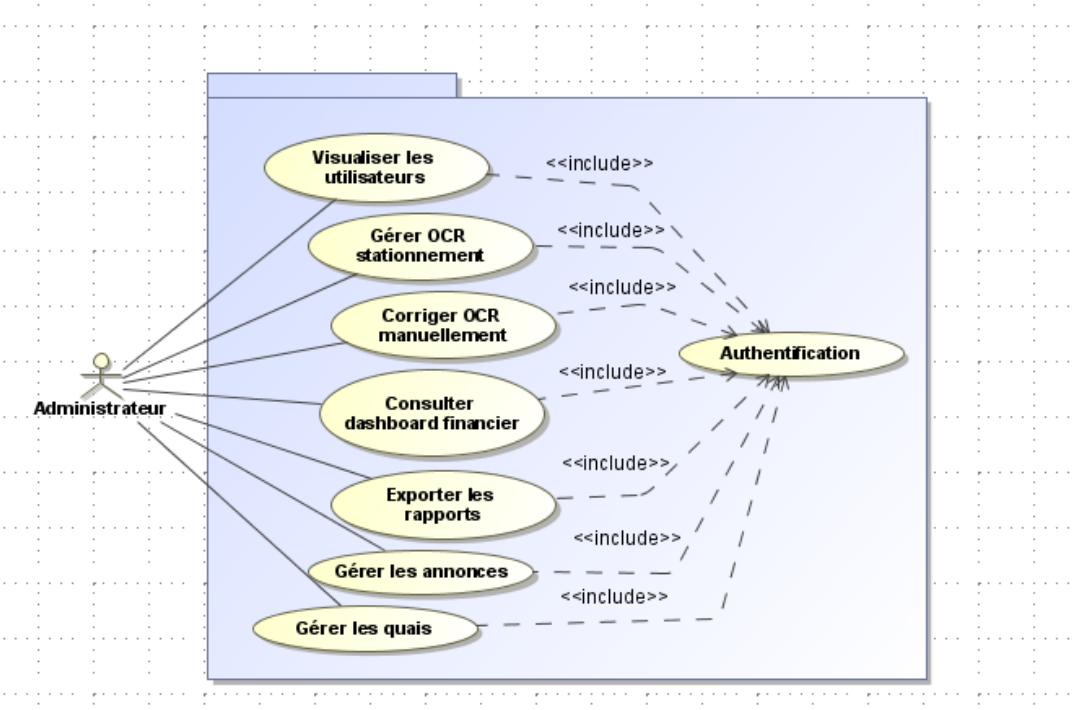


FIGURE 2.1 – Diagramme de cas d'utilisation de l'administrateur

La figure 2.1 illustre le diagramme de cas d'utilisation associé à l'administrateur, considéré comme l'acteur disposant des privilèges les plus élevés au sein du système. Cet acteur joue un rôle central dans la supervision et la gestion globale des opérations de la gare routière.

L'administrateur peut gérer les quais de stationnement, superviser les opérations de reconnaissance OCR des plaques d'immatriculation, corriger les éventuelles erreurs de détection, administrer les comptes utilisateurs, créer des annonces bilingues destinées aux voyageurs, ainsi que consulter les statistiques et tableaux de bord de supervision.

Le diagramme met également en évidence la relation *include* entre les différents cas d'utilisation et le module d'authentification. Cette relation traduit le fait que toutes les opérations sensibles nécessitent obligatoirement une authentification préalable afin de garantir la sécurité et le contrôle des accès au système.

Ce diagramme permet ainsi de visualiser l'ensemble des responsabilités attribuées à l'administrateur ainsi que son interaction avec les principaux services de supervision et d'administration de la plateforme.

2.4.2 Cas d'utilisation : Voyageur

Le diagramme suivant présente les différentes interactions possibles entre le voyageur et les principaux services proposés par la plateforme intelligente de gestion de gare routière.

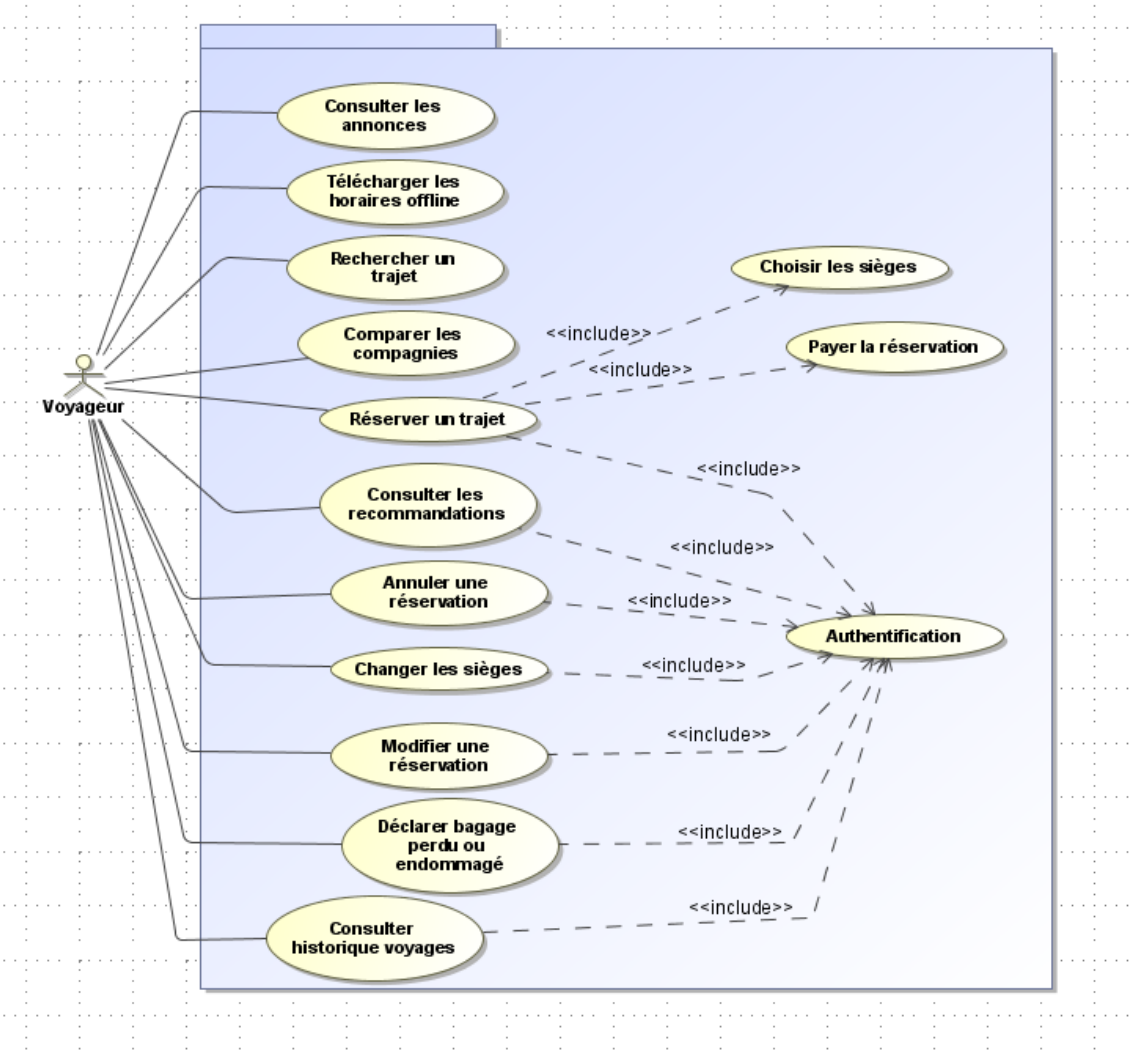


FIGURE 2.2 – Diagramme de cas d'utilisation du voyageur

La figure 2.2 illustre le diagramme de cas d'utilisation du voyageur, qui représente l'utilisateur principal de la plateforme côté client. Cet acteur interagit avec plusieurs modules métier permettant de simplifier et de digitaliser l'ensemble du processus de réservation et de gestion des voyages.

Le voyageur peut rechercher et comparer les trajets disponibles proposés par différentes compagnies de transport en fonction de la destination, de la date et des horaires souhaités. Après sélection d'un trajet, il peut effectuer une réservation multi-passagers, choisir les sièges disponibles à travers un plan interactif, puis procéder au paiement sécurisé en ligne.

Le système permet également au voyageur de gérer l'ensemble de ses tickets électroniques, notamment la consultation, la modification de la date ou de l'horaire, l'annulation

avec remboursement partiel ainsi que le re-téléchargement du ticket PDF contenant le QR Code d'embarquement.

En complément, le voyageur reçoit des notifications temps réel relatives à ses réservations, telles que les confirmations de paiement, les rappels avant départ, les changements de quai ou les éventuelles modifications de trajet.

Le diagramme met également en évidence que certaines opérations critiques, notamment les réservations et les paiements, nécessitent une authentification préalable afin d'assurer la sécurité des transactions et la protection des données personnelles des utilisateurs.

2.4.3 Cas d'utilisation : Chauffeur

Le diagramme suivant présente les différentes interactions entre le chauffeur et les modules opérationnels de la plateforme intelligente de gestion de gare routière.

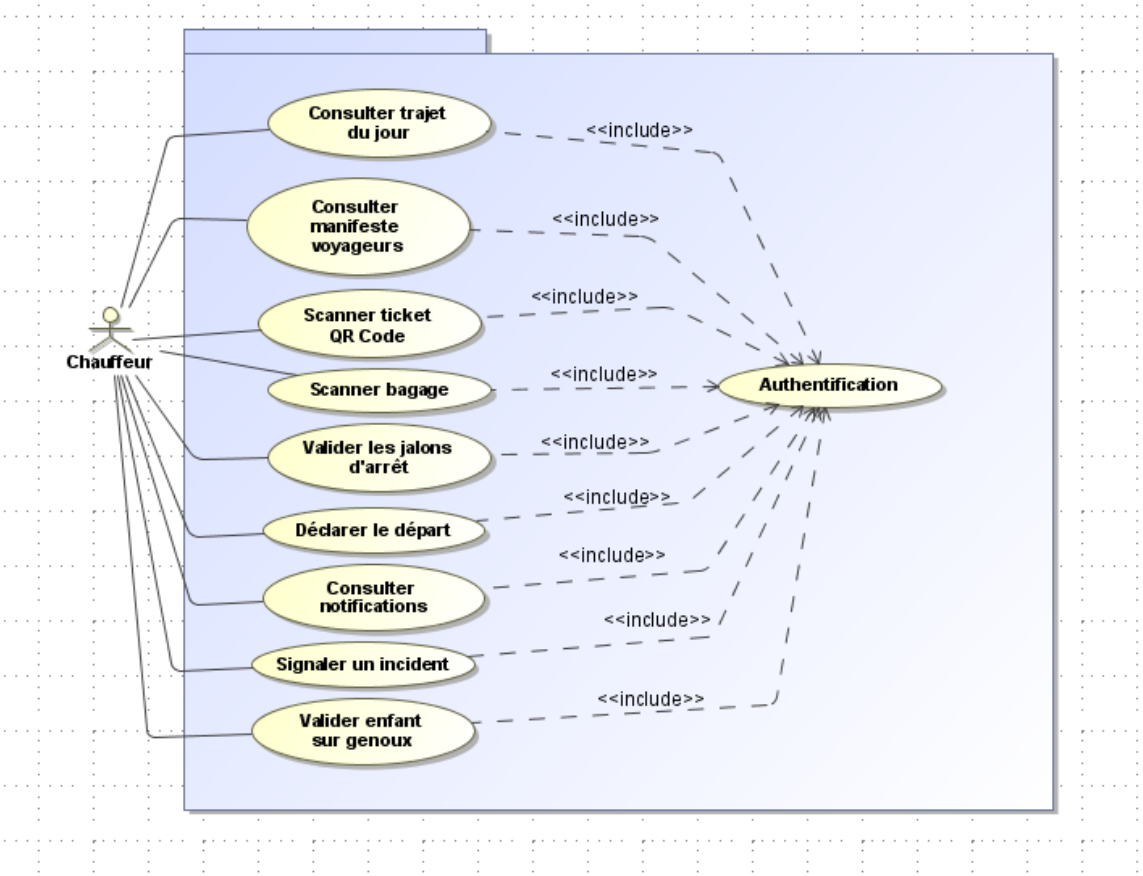


FIGURE 2.3 – Diagramme de cas d'utilisation du chauffeur

La figure 2.3 illustre le diagramme de cas d'utilisation associé au chauffeur, acteur essentiel dans les opérations de terrain liées au transport et à la gestion des voyageurs. Le chauffeur intervient principalement durant les phases d'embarquement, de suivi du trajet et de validation des opérations de transport.

À travers la plateforme, le chauffeur peut consulter les trajets qui lui sont attribués

ainsi que les informations relatives aux horaires, aux quais et aux voyageurs enregistrés. Lors de l'embarquement, il utilise le module de scan QR Code afin de vérifier l'authenticité des tickets électroniques et valider l'accès des voyageurs au bus.

Le système permet également au chauffeur de gérer les bagages des voyageurs, de confirmer les différentes étapes du trajet à travers la validation des jalons d'arrêt, ainsi que de déclarer le départ du véhicule, ce qui entraîne automatiquement la libération du quai occupé.

En cas de problème ou d'événement exceptionnel, le chauffeur peut également signaler des incidents directement depuis la plateforme afin d'informer les administrateurs et les responsables de compagnie en temps réel.

L'ensemble des fonctionnalités accessibles au chauffeur est protégé par une authentification sécurisée garantissant le contrôle des accès et la traçabilité des opérations effectuées au sein du système.

2.4.4 Cas d'utilisation : Responsable Compagnie

Le diagramme suivant présente les principales interactions entre le responsable compagnie et les différents modules de gestion opérationnelle de la plateforme.

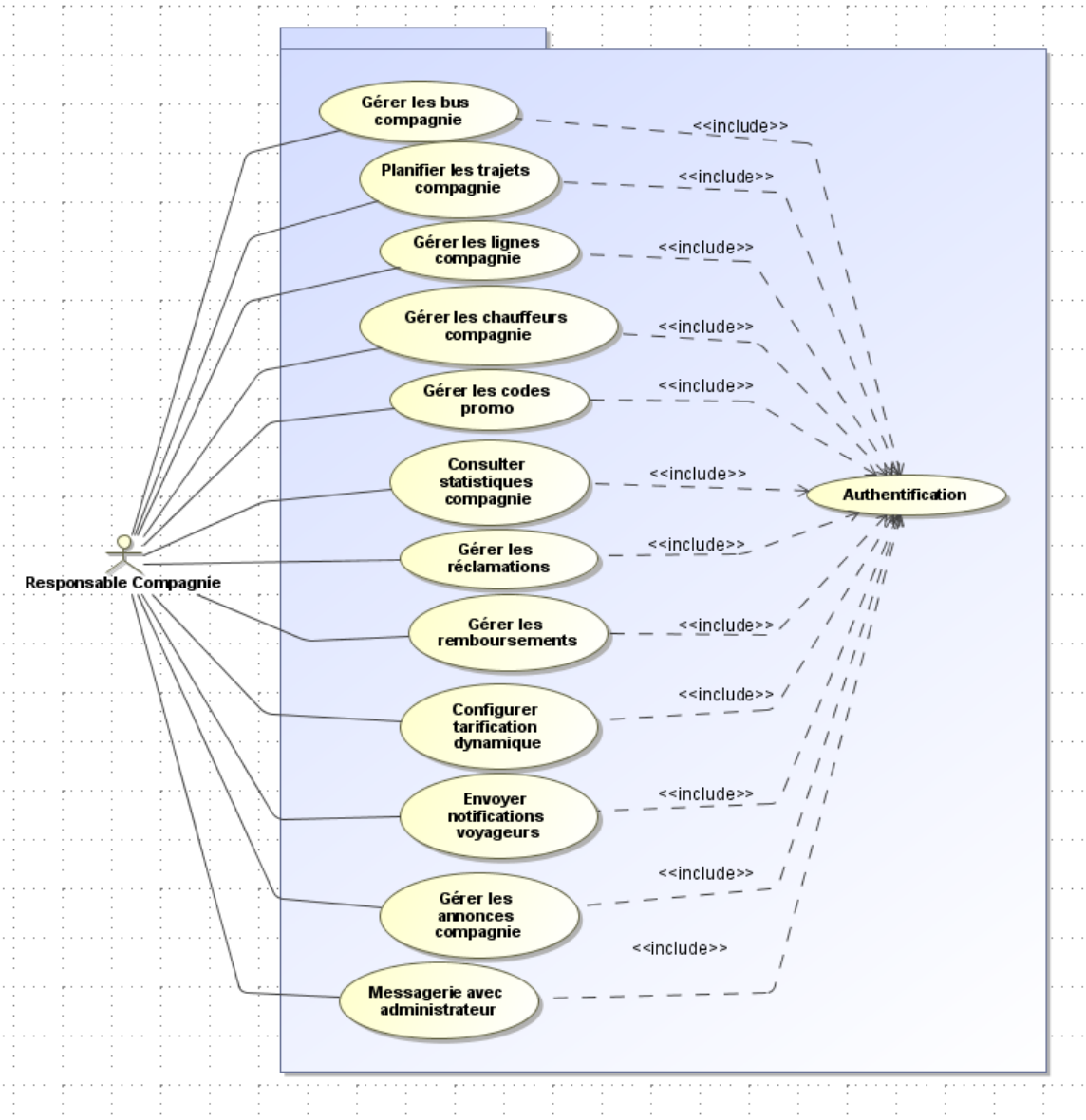


FIGURE 2.4 – Diagramme de cas d'utilisation du responsable compagnie

La figure 2.4 illustre le diagramme de cas d'utilisation du responsable compagnie, acteur chargé de la gestion complète des activités de sa compagnie de transport au sein de la gare routière intelligente.

Le responsable compagnie dispose de plusieurs fonctionnalités lui permettant d'assurer l'organisation et la supervision des opérations de transport. Il peut gérer le parc de bus de sa compagnie, administrer les informations relatives aux véhicules, planifier les trajets ainsi que les lignes de transport, et attribuer les chauffeurs aux différents trajets programmés.

Le système lui permet également de gérer les comptes chauffeurs, de suivre les activités opérationnelles de sa compagnie et de consulter des statistiques détaillées concernant les réservations, les revenus, le taux de remplissage des trajets et les performances globales de l'entreprise.

Par ailleurs, le responsable compagnie peut configurer des mécanismes de tarification

dynamique selon différents critères tels que la demande, le taux d'occupation ou la période de réservation. Il peut également créer et gérer des codes promotionnels afin d'améliorer l'attractivité commerciale de ses offres.

Enfin, la plateforme offre au responsable compagnie la possibilité de traiter les demandes de remboursement des voyageurs ainsi que de superviser les différentes opérations financières liées aux réservations et aux paiements.

Ce diagramme met ainsi en évidence le rôle stratégique du responsable compagnie dans la gestion administrative, commerciale et opérationnelle des services de transport proposés par la plateforme.

2.5 Diagramme BPMN : Processus global de réservation et embarquement

Le diagramme suivant présente le processus métier global de réservation et d'embarquement d'un voyageur au sein de la plateforme intelligente de gestion de gare routière.

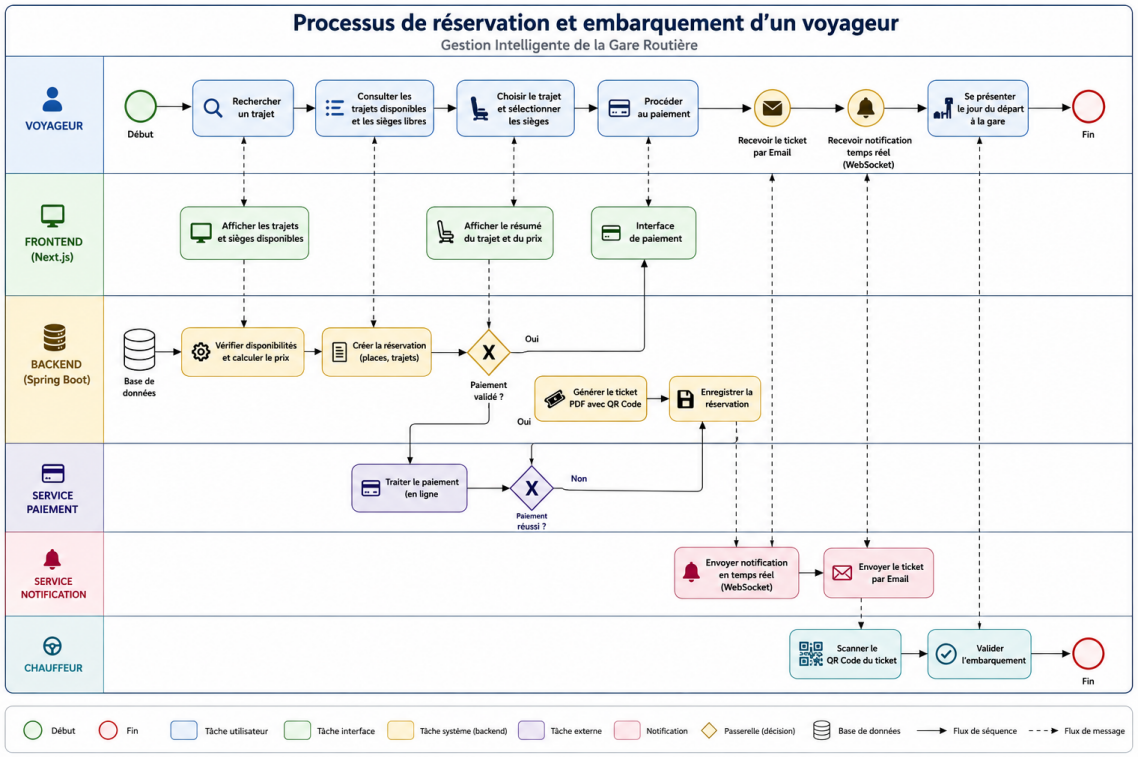


FIGURE 2.5 – Diagramme BPMN du processus de réservation et embarquement

La figure 2.5 illustre le diagramme BPMN représentant le processus complet de réservation et d'embarquement d'un voyageur au sein de la plateforme intelligente de gestion de gare routière. Ce diagramme modélise l'enchaînement des différentes activités métier ainsi que les interactions entre les principaux acteurs et services du système.

Le processus débute lorsque le voyageur effectue une recherche de trajet à travers l'interface frontend développée avec Next.js. Le système affiche alors les trajets disponibles, les horaires ainsi que les sièges libres proposés par les différentes compagnies de transport. Les requêtes sont transmises au backend Spring Boot qui interroge la base de données afin de vérifier les disponibilités en temps réel et calculer automatiquement les tarifs correspondants.

Après sélection du trajet et des sièges souhaités, le voyageur accède au module de paiement sécurisé en ligne. Une passerelle de décision BPMN permet de vérifier si le paiement a été validé avec succès. En cas d'échec, le processus retourne vers l'étape de paiement afin de permettre une nouvelle tentative. En revanche, lorsque le paiement est accepté, le backend procède automatiquement à la création de la réservation ainsi qu'à la génération d'un ticket électronique PDF contenant un QR Code unique associé au voyageur.

Le système enregistre ensuite les informations de réservation dans la base de données MySQL avant de déclencher plusieurs traitements asynchrones. Le service de notification envoie immédiatement une notification temps réel au voyageur via WebSocket afin de confirmer la réservation. Un email contenant le ticket PDF et le QR Code est également transmis automatiquement au voyageur.

Le jour du départ, le voyageur se présente à la gare routière muni de son ticket électronique. Le chauffeur utilise alors le module de scan QR Code afin de vérifier l'authenticité du ticket et de valider l'embarquement du voyageur. Une fois le contrôle effectué avec succès, le système confirme définitivement l'embarquement et clôture le processus métier.

Ce diagramme BPMN met ainsi en évidence la coordination entre plusieurs composants distribués de la plateforme, notamment le frontend Next.js, le backend Spring Boot, le service de paiement, le service de notification temps réel ainsi que les opérations de contrôle réalisées par le chauffeur. Il illustre également l'intégration des communications synchrones via API REST et des communications asynchrones via WebSocket et services de notification.

2.6 Diagramme de séquence : Processus de réservation

Le diagramme de séquence suivant illustre le flux complet du processus de réservation d'un voyageur, depuis la recherche du trajet jusqu'à la génération du ticket PDF avec QR Code.

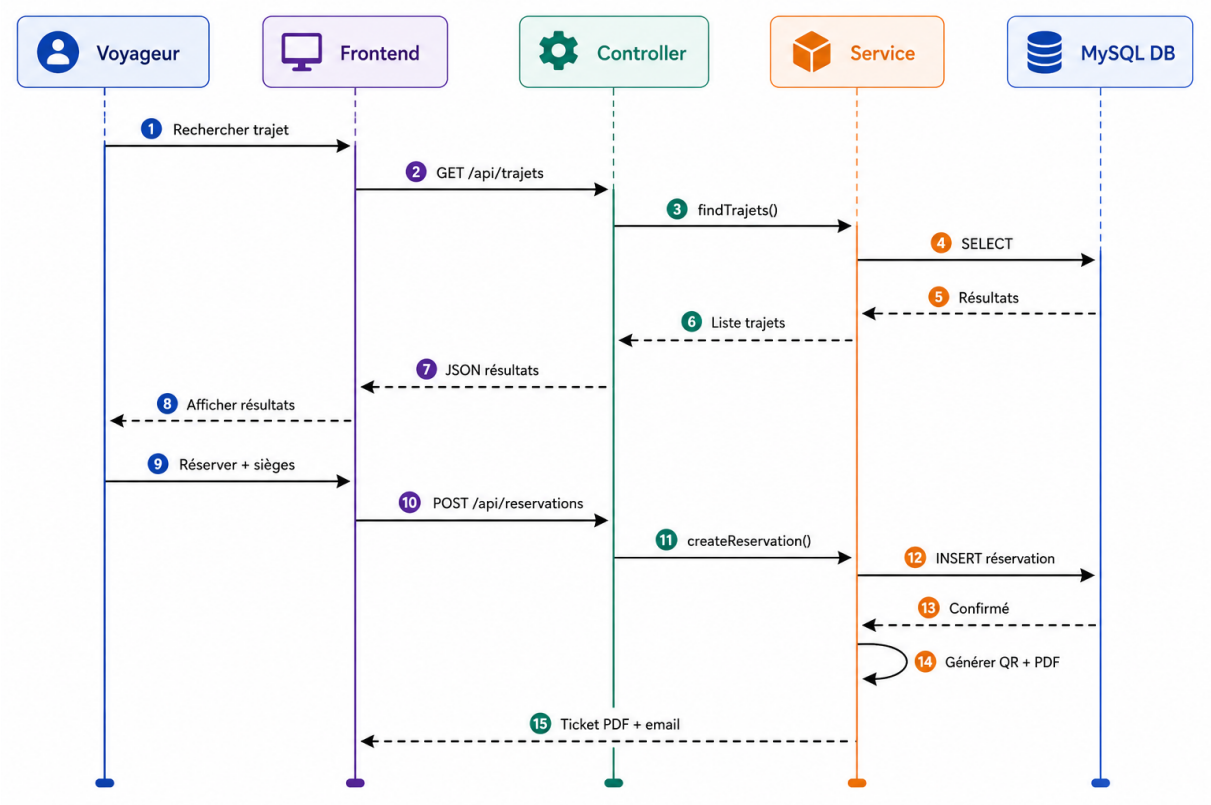


FIGURE 2.6 – Diagramme de séquence du processus de réservation

La figure 2.6 détaille le flux de réservation de bout en bout. Le voyageur initie une recherche de trajet via le frontend Next.js, qui interroge le contrôleur Spring Boot. Ce dernier délègue au service métier la recherche en base de données MySQL et retourne les résultats au voyageur sous forme JSON. Une fois le trajet sélectionné, le voyageur soumet sa réservation accompagnée des sièges choisis. Le service crée la réservation en base, génère automatiquement le QR Code et le ticket PDF, puis les transmet par email. Ce flux illustre la communication synchrone REST entre le frontend et le backend, et met en évidence le rôle central du service métier dans l'orchestration des traitements.

2.7 Diagramme de séquence : Détection OCR et attribution de quai

Le diagramme suivant présente le déroulement complet du processus automatisé de détection OCR des plaques d'immatriculation et d'attribution intelligente des quais au sein de la plateforme.

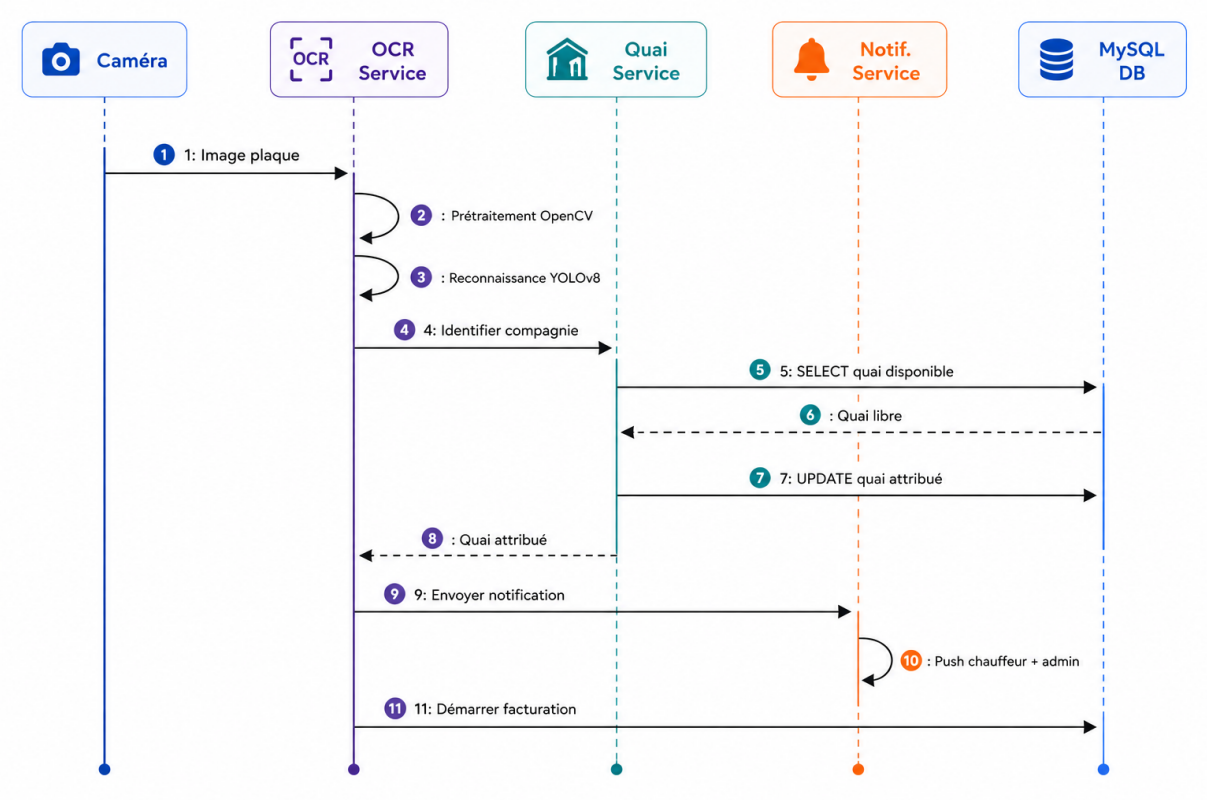


FIGURE 2.7 – Diagramme de séquence : Détection OCR et attribution automatique de quai

La figure 2.7 illustre le diagramme de séquence représentant le flux automatisé de détection OCR et d'attribution des quais dans la gare routière intelligente. Ce processus met en évidence la collaboration entre plusieurs composants distribués du système afin d'automatiser les opérations de contrôle et de gestion des bus à leur arrivée.

Le processus débute lorsqu'une image contenant la plaque d'immatriculation d'un bus est soumise au système via le module OCR. Le service OCR effectue tout d'abord un prétraitement de l'image à l'aide de la bibliothèque OpenCV afin d'améliorer la qualité visuelle, le contraste et la lisibilité des caractères détectés.

Une fois l'image optimisée, le modèle de reconnaissance YOLOv8 analyse la plaque et extrait automatiquement le numéro d'immatriculation du véhicule. Le backend utilise ensuite ce numéro afin d'identifier la compagnie de transport associée ainsi que les informations relatives au bus concerné.

Le service de gestion des quais interroge alors la base de données MySQL afin de rechercher un quai disponible correspondant aux règles d'attribution définies par le système. Après sélection du quai approprié, celui-ci est automatiquement réservé et affecté au bus détecté.

Le système déclenche ensuite plusieurs traitements asynchrones à travers le service de notification basé sur ActiveMQ et JMS. Des notifications temps réel sont envoyées au chauffeur ainsi qu'à l'administrateur afin de les informer du numéro de quai attribué et

de l'état de l'opération.

Par ailleurs, le système démarre automatiquement la facturation du stationnement avec enregistrement de l'horodatage d'entrée du véhicule dans la gare routière. Toutes les opérations effectuées sont enregistrées dans la base de données afin d'assurer la traçabilité et le suivi des activités.

Ce diagramme met ainsi en évidence l'intégration de plusieurs technologies distribuées au sein de la plateforme, notamment OpenCV, YOLOv8, Spring Boot, ActiveMQ, JMS et MySQL. Il démontre également l'utilisation combinée des communications synchrones et asynchrones afin d'assurer l'automatisation, la réactivité et la robustesse du système.

2.8 Diagramme de classes

Le diagramme suivant présente la structure statique globale du système ainsi que les principales entités métier composant la plateforme intelligente de gestion de gare routière.

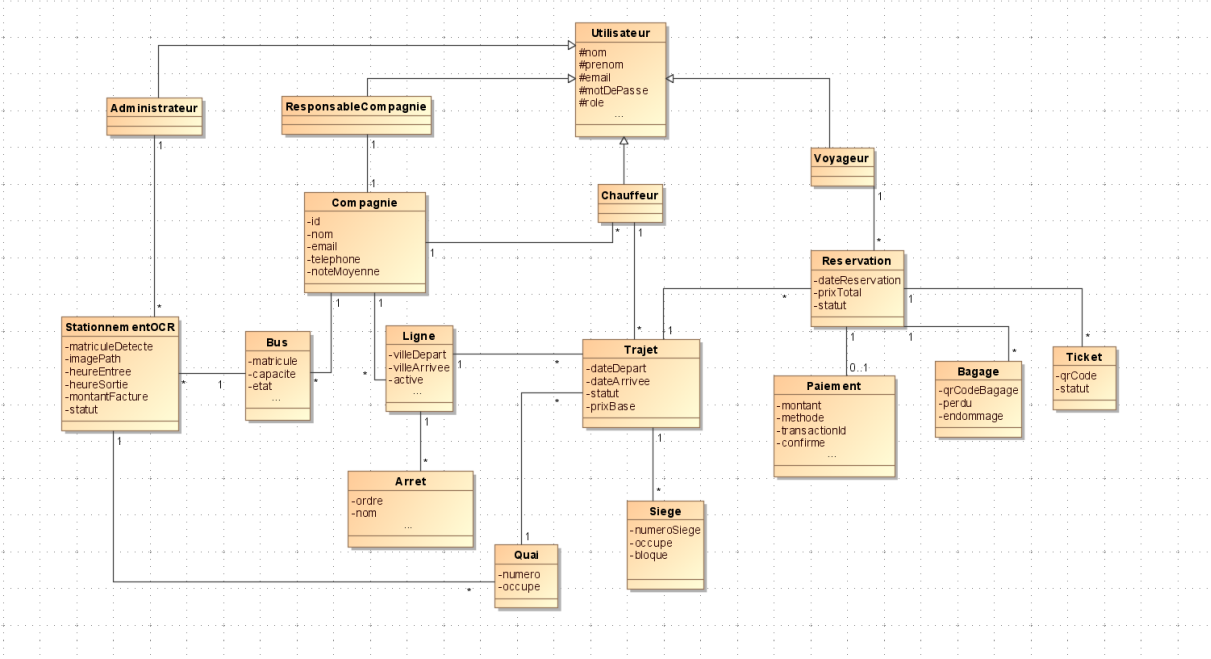


FIGURE 2.8 – Diagramme de classes du système

La figure 2.8 illustre le diagramme de classes global de la plateforme. Ce diagramme met en évidence les différentes entités métier du système ainsi que les relations d'association, d'héritage et de dépendance existantes entre elles. Il représente la structure logique des données manipulées par l'application et permet de modéliser les interactions entre les différents modules fonctionnels de la gare routière intelligente.

L'entité *Utilisateur* constitue la classe de base du système. Elle regroupe les informations communes aux différents profils de la plateforme, notamment le nom, le prénom, l'adresse email, le mot de passe ainsi que le rôle attribué à chaque utilisateur. À partir

de cette classe principale héritent plusieurs profils spécialisés tels que *Administrateur*, *Voyageur*, *Chauffeur* et *ResponsableCompagnie*. Cette approche permet de centraliser les mécanismes d'authentification, de gestion des accès et de contrôle des permissions.

La classe *ResponsableCompagnie* est associée à l'entité *Compagnie*, qui représente les différentes sociétés de transport enregistrées dans la plateforme. Chaque compagnie possède des informations telles que le nom, l'email de contact et les informations métier associées. Une compagnie peut gérer plusieurs bus, lignes et chauffeurs afin d'assurer l'organisation des trajets proposés aux voyageurs.

L'entité *Bus* représente les véhicules utilisés pour le transport des voyageurs. Chaque bus possède des caractéristiques propres telles que la marque, la capacité et l'état opérationnel. Les bus sont associés aux compagnies de transport et utilisés dans les différents trajets planifiés par le système.

La classe *Ligne* permet de modéliser les itinéraires de transport entre les villes de départ et d'arrivée. Chaque ligne peut contenir plusieurs arrêts intermédiaires représentés par l'entité *Arrêt*, permettant ainsi de gérer les différentes étapes du trajet.

La classe *Trajet* occupe une position centrale dans l'architecture du système. Elle relie plusieurs entités importantes telles que le bus affecté, le chauffeur responsable, la ligne empruntée, le quai de départ ainsi que les sièges disponibles. Chaque trajet contient également des informations relatives à la date de départ, à la date d'arrivée, au statut du trajet ainsi qu'au prix associé.

L'entité *Quai* représente les quais disponibles au sein de la gare routière. Chaque quai peut être attribué automatiquement à un trajet ou à un bus lors des opérations de stationnement et d'embarquement. Cette entité permet également de gérer l'occupation des quais en temps réel.

La classe *Siège* permet de gérer les places disponibles dans les bus. Chaque siège possède un numéro unique ainsi qu'un état indiquant s'il est libre ou occupé. Cette gestion facilite la réservation interactive des places par les voyageurs.

L'entité *Réservation* représente les réservations effectuées par les voyageurs. Elle contient des informations importantes telles que la date de réservation, le montant total payé ainsi que le statut de la réservation. Une réservation est liée à un voyageur et peut être associée à plusieurs services complémentaires du système.

La classe *Paiement* assure la gestion des transactions financières liées aux réservations. Elle stocke les informations relatives au montant payé, à la méthode de paiement utilisée ainsi qu'à la confirmation de la transaction.

L'entité *Ticket* représente le billet électronique généré automatiquement après validation du paiement. Chaque ticket contient un QR Code unique utilisé lors des opérations de contrôle d'embarquement effectuées par le chauffeur.

La classe *Bagage* permet de gérer les bagages enregistrés par les voyageurs. Elle contient des informations relatives au poids et à l'état des bagages afin d'assurer leur

suivi durant le trajet.

Enfin, l'entité *StationnementOCR* joue un rôle important dans les opérations intelligentes de la gare routière. Elle permet de gérer les informations issues du système de reconnaissance automatique des plaques d'immatriculation. Cette classe stocke notamment le numéro de plaque détecté, les horaires d'entrée et de sortie des véhicules, le montant facturé pour le stationnement ainsi que le statut de l'opération.

Ce diagramme de classes offre ainsi une vision globale de l'architecture métier du système et met en évidence les relations entre les différents composants fonctionnels de la plateforme intelligente de gestion de gare routière.

Conclusion

Ce chapitre a présenté l'analyse complète des besoins fonctionnels et non fonctionnels du système, les principales user stories du Product Backlog, les diagrammes de cas d'utilisation pour chacun des quatre acteurs, les diagrammes de séquence des flux principaux ainsi que le diagramme de classes du système. Le chapitre suivant sera consacré à la présentation de l'architecture technique et des technologies utilisées, avec une attention particulière portée aux aspects des systèmes distribués.

Chapitre 3

Architecture Technique et Technologies Utilisées

Introduction

Ce chapitre présente l'architecture technique globale du système ainsi que les technologies utilisées. Une attention particulière est accordée aux composants des systèmes distribués intégrés dans la solution, notamment la communication synchrone via REST et RMI, la communication asynchrone via ActiveMQ et JMS, et les mécanismes de résilience via Resilience4j.

3.1 Architecture générale du système

La plateforme développée repose sur une architecture distribuée multicouche assurant une séparation claire entre les différentes responsabilités du système. Cette architecture favorise la maintenabilité, la scalabilité, la modularité ainsi que l'évolutivité de la plateforme.

La figure suivante présente l'architecture générale multicouche adoptée pour le développement du système.

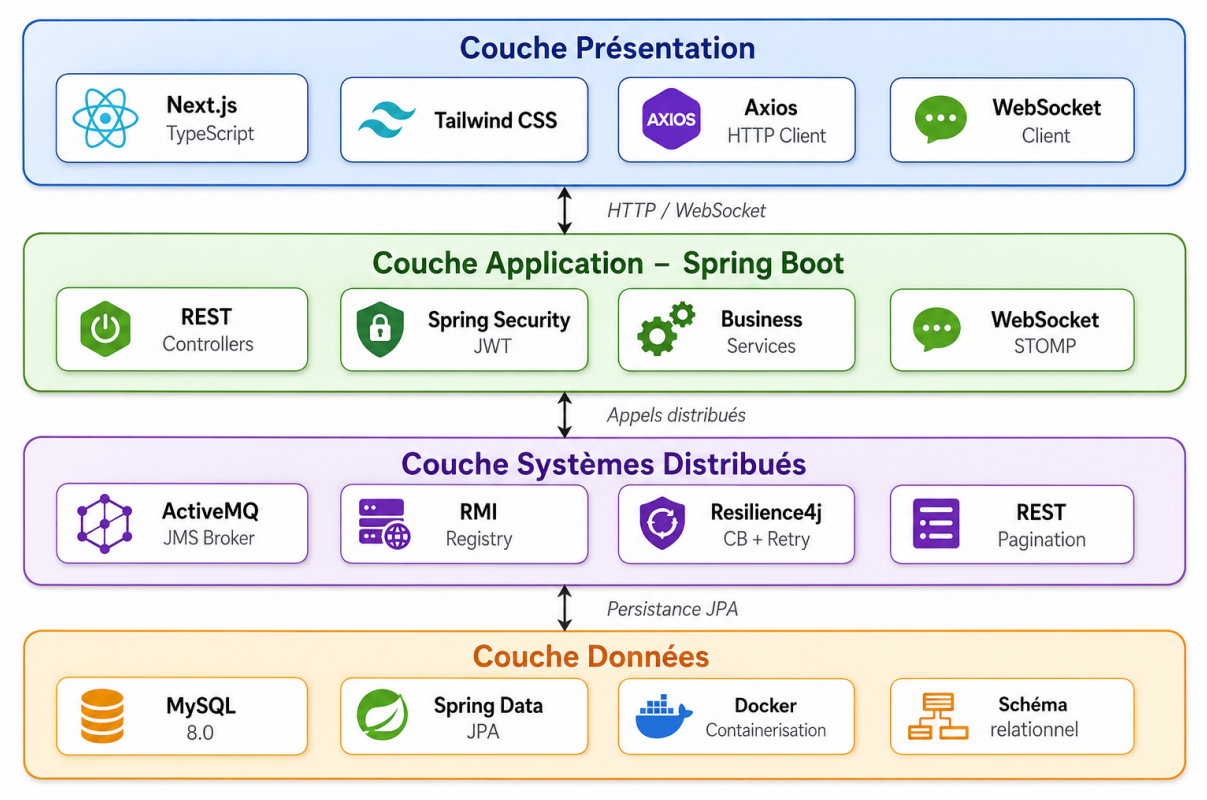


FIGURE 3.1 – Architecture générale multicouche du système

La figure 3.1 illustre l'architecture globale du système organisée selon un modèle multicouche permettant de séparer les différentes responsabilités techniques et fonctionnelles de la plateforme.

La première couche correspond à la couche présentation. Elle regroupe l'ensemble des composants liés à l'interface utilisateur développée avec Next.js et TypeScript. Cette couche utilise également le framework Tailwind CSS pour la conception de l'interface graphique ainsi que le client HTTP Axios pour assurer les échanges de données avec le backend via les API REST. Elle représente le point d'interaction principal entre les utilisateurs et la plateforme.

La deuxième couche représente la couche application basée sur Spring Boot. Elle contient les principaux traitements métiers du système, notamment les contrôleurs REST chargés de recevoir les requêtes HTTP, les services métiers responsables du traitement des données, les mécanismes de sécurité JWT assurant l'authentification et l'autorisation des utilisateurs, ainsi que les services WebSocket permettant les communications temps réel avec les clients connectés.

La troisième couche correspond à la couche systèmes distribués. Cette couche regroupe les différents composants distribués intégrés dans le cadre du module Systèmes Distribués. Elle inclut le broker de messagerie ActiveMQ utilisé pour les communications asynchrones via JMS, le registre RMI permettant l'invocation distante de méthodes, les mécanismes de résilience fournis par la bibliothèque Resilience4j ainsi que les mécanismes de pagination

REST permettant l'optimisation des échanges de données volumineuses.

Enfin, la quatrième couche représente la couche données. Cette couche assure la persistance et la gestion des informations du système à travers la base de données MySQL. L'accès aux données est assuré par Spring Data JPA qui facilite la manipulation des entités et des relations objets-relationnelles. La plateforme utilise également Docker afin de garantir la conteneurisation des services, la portabilité du système ainsi qu'une meilleure gestion des environnements de déploiement.

Cette architecture multicouche permet ainsi d'assurer une meilleure organisation du projet, une séparation claire des responsabilités, une meilleure maintenabilité du code ainsi qu'une grande flexibilité pour les futures évolutions de la plateforme.

3.2 Architecture Backend – Spring Boot

Le backend du système a été développé avec Spring Boot 3.4.4 en adoptant une architecture multicouche permettant une organisation claire, modulaire et rigoureuse des traitements métiers.

La figure suivante présente l'architecture interne du backend Spring Boot ainsi que les différentes couches techniques composant le serveur applicatif.

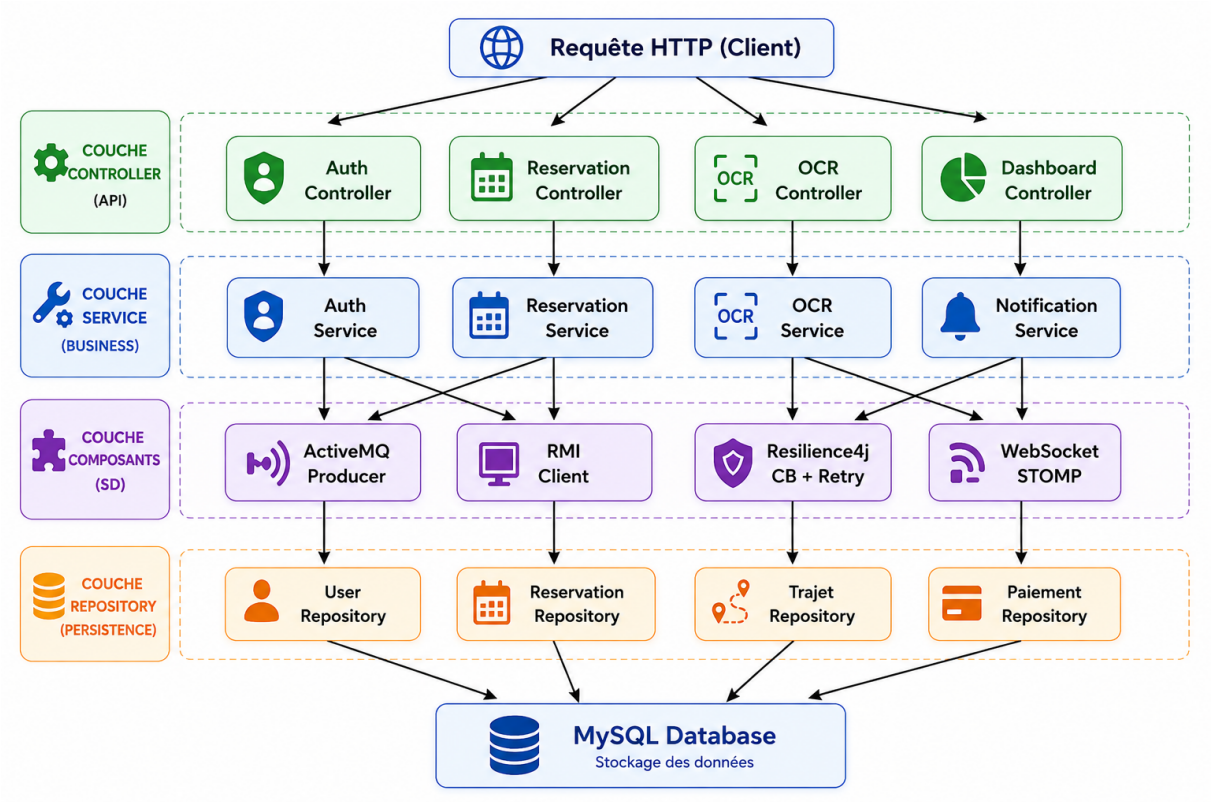


FIGURE 3.2 – Architecture interne du backend Spring Boot

La figure 3.2 illustre l'organisation interne du backend Spring Boot selon une architec-

ture structurée en plusieurs couches distinctes. Cette architecture permet de séparer les responsabilités techniques afin d'améliorer la maintenabilité, la lisibilité et l'évolutivité du système.

La première couche correspond à la couche contrôleur (*Controller Layer*). Elle regroupe les différents contrôleurs REST tels que *AuthController*, *ReservationController*, *OCRController* et *DashboardController*. Ces composants sont responsables de la réception des requêtes HTTP envoyées par le frontend Next.js, de la validation des données entrantes ainsi que du retour des réponses au format JSON.

Les contrôleurs délèguent ensuite les traitements métiers à la couche service (*Service Layer*). Cette couche contient l'ensemble de la logique métier de l'application. Les services assurent notamment la gestion des réservations, des paiements, des notifications, des opérations OCR, de l'authentification ainsi que des traitements liés à la supervision de la gare routière.

Afin de répondre aux besoins du module Systèmes Distribués, les services métiers communiquent avec plusieurs composants distribués intégrés dans le backend. Parmi ces composants figurent ActiveMQ pour la messagerie asynchrone, RMI pour les invocations distantes, Resilience4j pour les mécanismes de résilience et de tolérance aux pannes, ainsi que WebSocket permettant les communications temps réel avec les utilisateurs connectés.

La couche persistance (*Repository Layer*) est assurée par Spring Data JPA. Les repositories permettent d'interagir avec la base de données MySQL à travers les entités métier du système. Cette couche facilite les opérations CRUD, la gestion des relations entre les entités ainsi que l'abstraction de l'accès aux données.

Enfin, la base de données MySQL constitue la couche stockage du système. Elle assure la persistance des informations relatives aux utilisateurs, réservations, trajets, tickets, paiements, notifications et opérations OCR.

Cette séparation stricte des responsabilités respecte le principe de séparation des préoccupations (*Separation of Concerns*) et facilite considérablement la maintenance, la modularité ainsi que la testabilité de chaque composant du backend Spring Boot.

3.3 Systèmes Distribués – Architecture et Communication

L'un des aspects fondamentaux du projet concerne l'implémentation des mécanismes de communication distribués. Le système combine plusieurs paradigmes de communication afin d'assurer la robustesse, la scalabilité et la réactivité de la plateforme.

3.3.1 Vue d'ensemble des communications distribuées

La figure suivante présente une vue globale des différents mécanismes de communication distribuée intégrés dans la plateforme intelligente de gestion de gare routière.

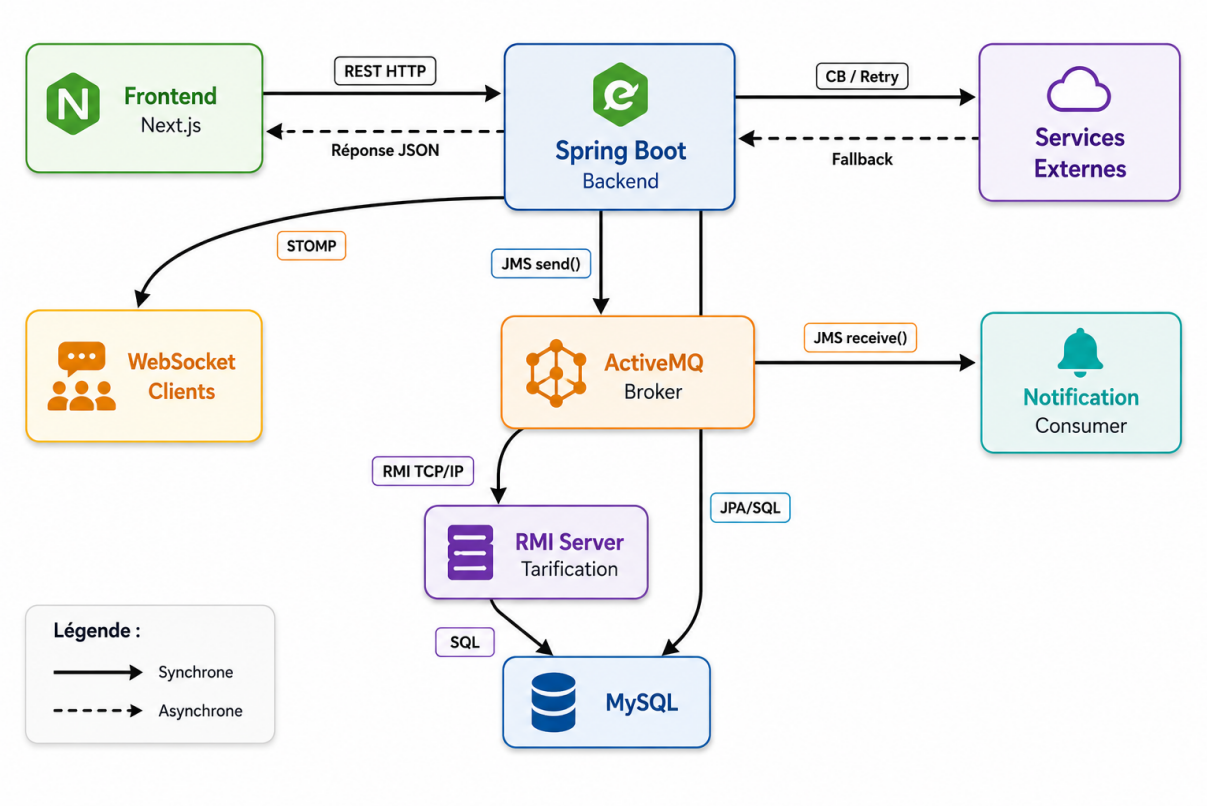


FIGURE 3.3 – Schéma global des communications distribuées du système

La figure 3.3 illustre l'architecture globale des communications distribuées mises en œuvre au sein du système. Cette architecture combine plusieurs paradigmes de communication afin d'assurer la réactivité, la scalabilité, la résilience ainsi que la continuité des services de la plateforme.

Le frontend développé avec Next.js communique avec le backend Spring Boot à travers des API REST via le protocole HTTP. Cette communication synchrone permet aux utilisateurs d'effectuer différentes opérations telles que l'authentification, la consultation des trajets, les réservations, les paiements et la gestion des tickets électroniques.

Le backend Spring Boot joue le rôle de composant central de traitement. Il reçoit les requêtes provenant du frontend, exécute les traitements métiers nécessaires puis interagit avec les différents composants distribués du système selon les besoins fonctionnels.

Pour les traitements asynchrones, le backend publie des messages vers le broker de messagerie ActiveMQ à travers l'API JMS. Ces messages sont ensuite consommés par le service de notification chargé de diffuser des alertes et notifications aux utilisateurs, notamment les confirmations de réservation, les rappels avant départ ou les changements

de quai. Cette communication asynchrone permet de découpler les services et d'améliorer les performances globales du système.

L'architecture intègre également un mécanisme d'invocation distante de méthodes via RMI (*Remote Method Invocation*). Ce mécanisme permet au backend d'interagir avec le service de tarification dynamique afin de calculer les prix des trajets selon plusieurs paramètres tels que la distance, le taux de remplissage ou les périodes de forte demande.

Les mécanismes de résilience fournis par la bibliothèque Resilience4j interviennent lors des appels vers les services externes afin d'assurer la tolérance aux pannes et la continuité des opérations. Les stratégies de *Retry*, *Circuit Breaker* et *Fallback* permettent de gérer efficacement les erreurs et les indisponibilités temporaires des services distants.

Enfin, les communications temps réel sont assurées par les WebSocket associés au protocole STOMP. Cette technologie permet une communication bidirectionnelle instantanée entre le serveur et les clients connectés afin de diffuser des notifications en temps réel sans nécessiter de rechargement des pages.

Cette architecture distribuée permet ainsi d'assurer une meilleure modularité du système, une forte réactivité des services ainsi qu'une meilleure gestion des traitements synchrones et asynchrones au sein de la plateforme.

3.3.2 Communication Synchrone – API REST

La communication synchrone principale du système repose sur une architecture REST exposée par le backend Spring Boot. Le frontend Next.js communique avec le backend via les méthodes HTTP standard (GET, POST, PUT, PATCH, DELETE) en échangeant des données au format JSON. Le tableau 3.1 présente les endpoints REST principaux du système.

TABLE 3.1 – Exemples d'endpoints REST principaux

Méthode	**Endpoint**	**Description**
GET	/api/trajets	Récupérer la liste des trajets disponibles avec pagination
POST	/api/auth/login	Authentification et génération du token JWT
POST	/api/reservations	Créer une nouvelle réservation multi-passagers
GET	/api/admin/reservations ?page=0&size=10	Lister les réservations avec pagination
PATCH	/api/reservations/{id}/cancel	Annuler une réservation
GET	/api/rmi/estimate ?distanceKm=300	Estimer le prix via invocation RMI
POST	/api/ocr/detect	Soumettre une image pour reconnaissance de plaque

La figure suivante présente le déroulement complet d'une requête REST sécurisée par le mécanisme d'authentification JWT au sein de la plateforme.

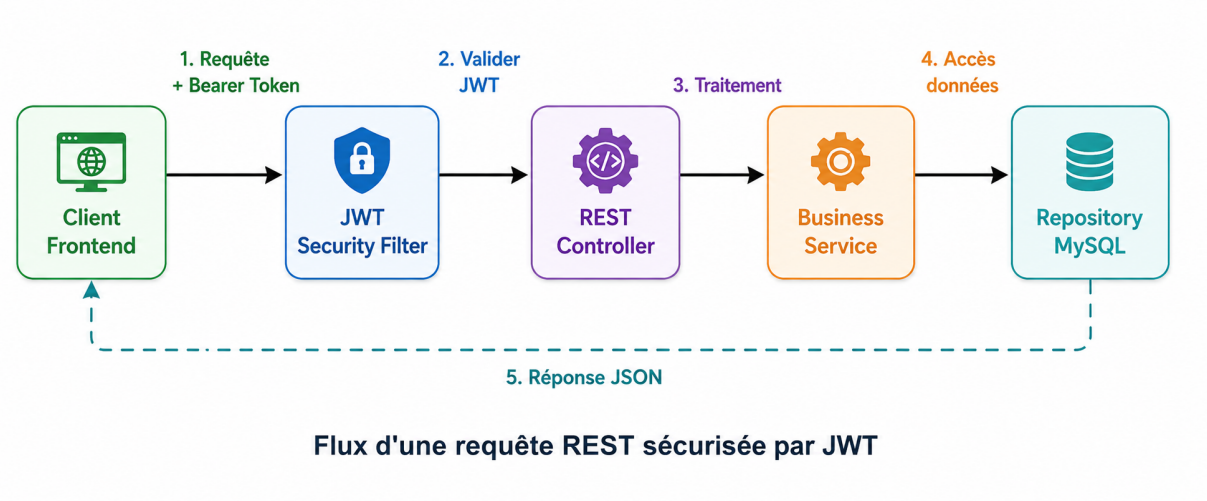


FIGURE 3.4 – Flux d'une requête REST sécurisée par JWT

La figure 3.4 illustre le flux d'exécution d'une requête REST sécurisée à travers le mécanisme JWT (*JSON Web Token*) implémenté avec Spring Security. Ce processus garantit la sécurisation des échanges entre le frontend et le backend tout en assurant un contrôle d'accès fiable aux ressources protégées du système.

Le processus débute lorsqu'un client envoie une requête HTTP vers le backend Spring Boot. Cette requête contient un *Bearer Token JWT* transmis dans l'en-tête *Authorization* de la requête HTTP. Ce token représente l'identité authentifiée de l'utilisateur ainsi que

les rôles et permissions associés.

À la réception de la requête, le filtre de sécurité Spring Security intercepte automatiquement la demande avant son traitement par les contrôleurs REST. Le système procède alors à la validation du token JWT en vérifiant plusieurs éléments importants, notamment la signature numérique du token, sa date d'expiration ainsi que les informations contenues dans les *claims*.

Si le token est valide, l'utilisateur est authentifié avec succès et les informations de sécurité sont injectées dans le contexte d'authentification Spring Security. Le système autorise alors l'accès au contrôleur REST concerné selon les rôles et permissions associés à l'utilisateur connecté.

Le contrôleur REST transmet ensuite la requête au service métier correspondant afin d'exécuter les traitements nécessaires. Les services métiers interagissent avec les repositories Spring Data JPA pour accéder aux données persistées dans la base de données MySQL.

Une fois le traitement terminé, la réponse est retournée au client sous forme de données JSON via HTTP. En cas de token invalide, expiré ou absent, le système refuse automatiquement l'accès à la ressource demandée et retourne une réponse d'erreur appropriée.

Ce mécanisme d'authentification stateless basé sur JWT permet d'assurer une meilleure sécurité des API REST tout en garantissant de bonnes performances et une grande scalabilité de la plateforme distribuée.

3.3.3 Sécurité – JWT et Spring Security

La figure suivante présente l'architecture d'authentification et de sécurisation des accès mise en place au sein de la plateforme à l'aide de JWT et Spring Security.

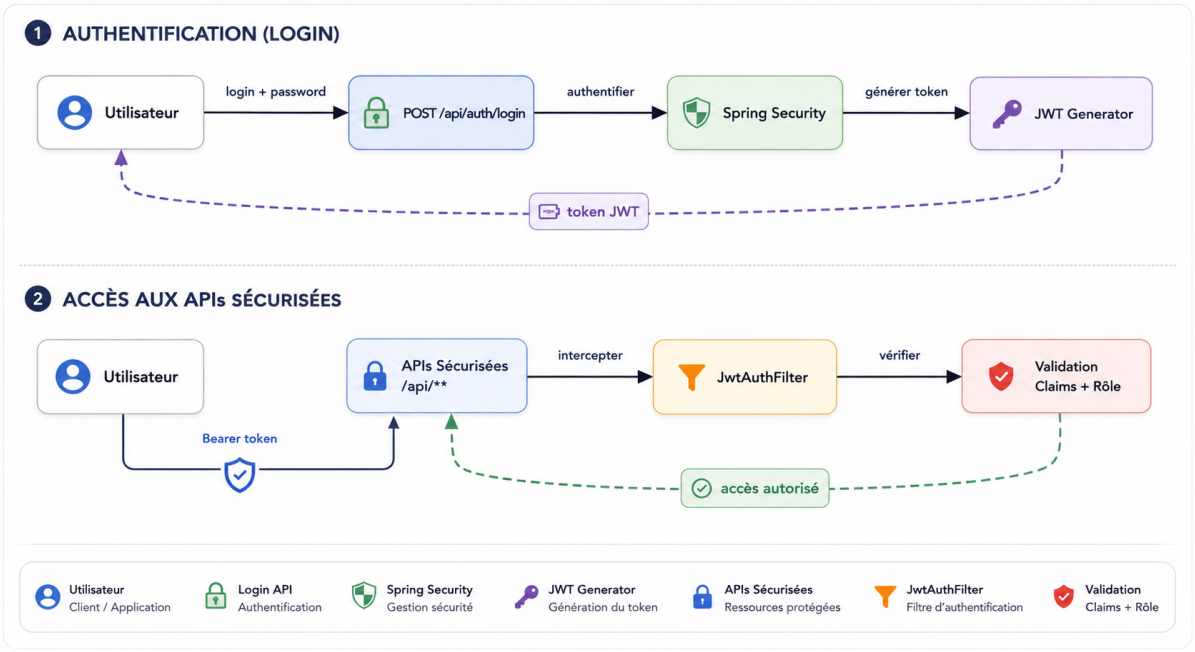


FIGURE 3.5 – Architecture d'authentification JWT avec Spring Security

La figure 3.5 illustre le mécanisme complet d'authentification sécurisé implémenté avec Spring Security et JWT (*JSON Web Token*). Cette architecture permet de garantir la sécurité des échanges entre le frontend et le backend tout en assurant un contrôle d'accès basé sur les rôles des utilisateurs.

Le processus d'authentification débute lors du premier accès de l'utilisateur à la plateforme. Celui-ci soumet ses identifiants de connexion, notamment son adresse email et son mot de passe, via le formulaire de login du frontend Next.js. Les informations sont ensuite transmises au backend Spring Boot à travers une requête HTTP sécurisée.

À la réception de la requête, Spring Security intercepte les données d'authentification et transmet les informations au composant *UserDetailsService*. Ce service se charge de récupérer les informations de l'utilisateur depuis la base de données MySQL ainsi que les rôles associés à son compte.

Le mot de passe fourni est ensuite comparé au mot de passe chiffré enregistré dans la base grâce à l'algorithme BCrypt. Si les identifiants sont valides, le système génère automatiquement un token JWT signé contenant les informations essentielles de l'utilisateur, notamment son identifiant, son rôle et la date d'expiration du token.

Le token JWT est alors retourné au client et stocké côté frontend afin d'être utilisé lors des requêtes ultérieures vers les API REST protégées du système.

Pour chaque nouvelle requête, le client inclut ce token dans l'en-tête HTTP *Authorization* sous la forme *Bearer Token*. Le filtre de sécurité *JwtAuthFilter* intercepte automatiquement la requête afin de vérifier la validité du token transmis.

Le système procède alors à plusieurs vérifications importantes, notamment la validation de la signature numérique du token, le contrôle de sa date d'expiration ainsi que la

vérification des rôles et permissions contenus dans les *claims*. Si toutes les validations sont correctes, l'accès à la ressource demandée est autorisé. Dans le cas contraire, le système bloque automatiquement l'accès et retourne une erreur d'authentification.

Cette architecture de sécurité basée sur JWT présente plusieurs avantages importants, notamment une authentification stateless, une meilleure scalabilité du système, une réduction de la charge serveur liée aux sessions ainsi qu'une sécurisation efficace des API REST de la plateforme distribuée.

3.3.4 Communication Asynchrone – ActiveMQ et JMS

La figure suivante présente l'architecture de communication asynchrone mise en œuvre dans la plateforme à l'aide du broker de messagerie ActiveMQ et de l'API JMS.

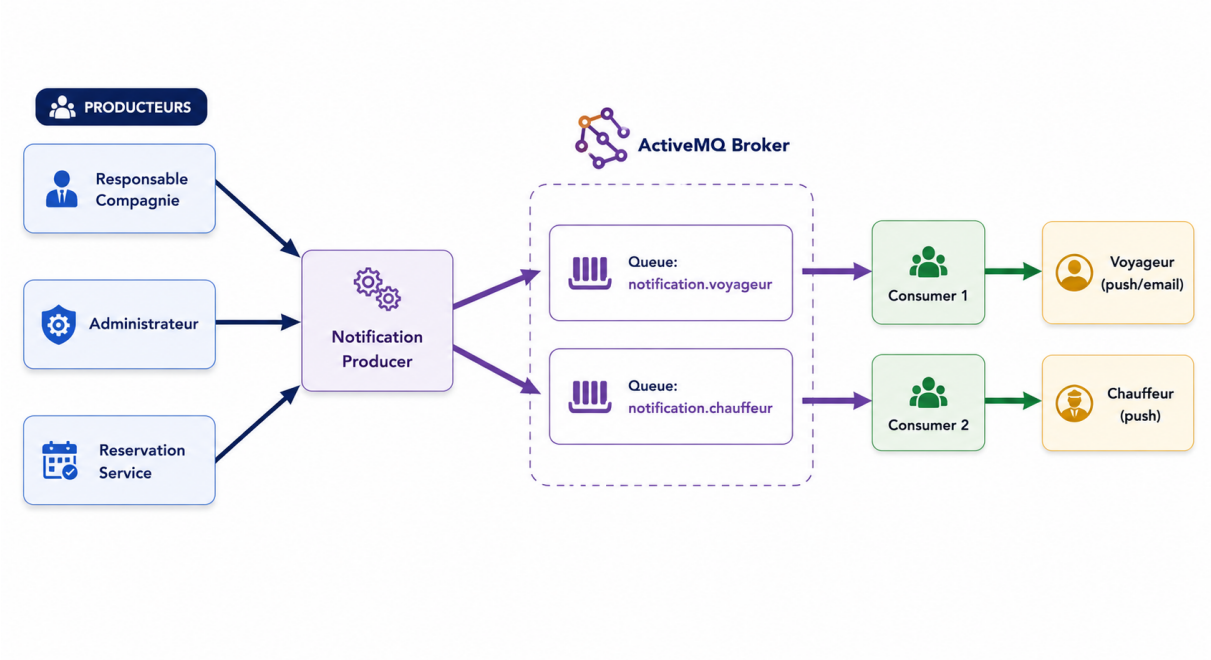


FIGURE 3.6 – Architecture de communication asynchrone avec ActiveMQ et JMS

La figure 3.6 illustre l'architecture de messagerie asynchrone utilisée au sein du système afin de gérer les notifications et les traitements distribués indépendamment des opérations principales de la plateforme.

Cette architecture repose sur l'utilisation du broker de messages ActiveMQ associé à l'API JMS (*Java Message Service*), permettant la mise en œuvre d'un modèle de communication asynchrone entre les différents services du système.

Les différents composants producteurs de notifications, notamment le *Responsable Compagnie*, l'*Administrateur* ainsi que le *Service de Réservation*, génèrent des messages correspondant aux événements métier importants de la plateforme. Ces événements peuvent concerner les confirmations de réservation, les rappels de départ, les changements de quai, les notifications destinées aux chauffeurs ou encore les alertes administratives.

Les messages produits sont envoyés vers un composant *Producer JMS* chargé de transmettre les données au broker ActiveMQ. Celui-ci assure ensuite le stockage temporaire et la distribution des messages dans différentes files d'attente (*queues*) dédiées.

L'architecture comporte deux queues principales : une première queue destinée aux notifications des voyageurs et une seconde réservée aux notifications des chauffeurs. Cette séparation permet d'organiser les traitements selon les différents types d'utilisateurs et de garantir une meilleure gestion des flux de communication.

Les composants consommateurs (*Consumers JMS*) écoutent en permanence les queues ActiveMQ afin de récupérer et traiter les messages entrants. Chaque consumer traite les notifications à son propre rythme, indépendamment des producteurs, ce qui permet un découplage fort entre les différents services du système.

Cette approche asynchrone présente plusieurs avantages importants. Elle améliore considérablement la réactivité globale de la plateforme en évitant de bloquer les traitements principaux lors de l'envoi des notifications. Elle permet également une meilleure tolérance aux pannes, car l'indisponibilité temporaire d'un consumer n'interrompt pas le fonctionnement des producteurs ni l'enregistrement des messages dans les queues.

Enfin, cette architecture distribuée facilite la scalabilité du système en permettant l'ajout de nouveaux consumers ou de nouvelles queues sans modification majeure de l'infrastructure existante.

3.3.5 Invocation de Méthodes Distantes – RMI

La figure suivante présente l'architecture de communication distante basée sur la technologie RMI utilisée pour le service de tarification dynamique de la plateforme.

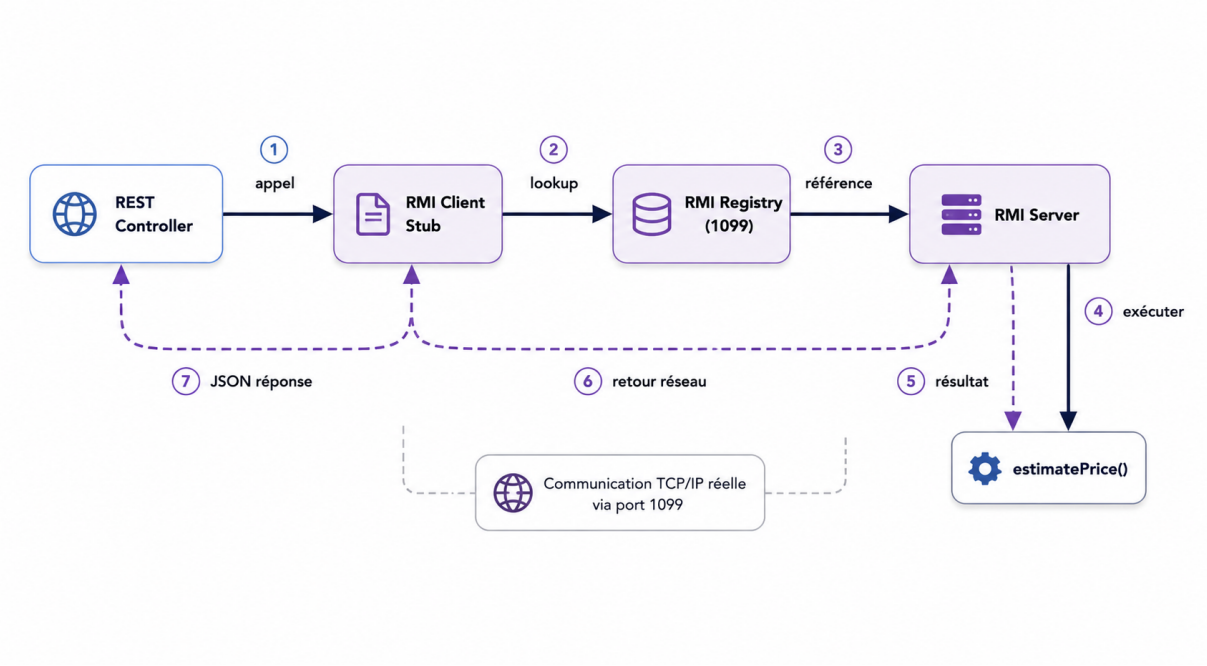


FIGURE 3.7 – Architecture RMI – Invocation distante pour la tarification

La figure 3.7 illustre le mécanisme d'invocation de méthodes distantes (*Remote Method Invocation – RMI*) utilisé dans le système afin de gérer le service de tarification dynamique des trajets. Cette architecture permet au backend Spring Boot d'exécuter des méthodes hébergées sur un serveur distant comme s'il s'agissait d'appels locaux.

Le processus débute lorsqu'un utilisateur effectue une demande d'estimation de prix à travers l'interface de réservation du système. La requête est envoyée au backend Spring Boot où le contrôleur REST concerné reçoit les paramètres nécessaires au calcul du tarif, notamment la distance du trajet ou les informations liées à la réservation.

Le contrôleur transmet ensuite la demande au composant client RMI à travers le *stub RMI*. Ce stub agit comme un proxy local représentant le service distant de tarification. Avant l'exécution de la méthode distante, le client RMI effectue un *lookup* auprès du registre RMI afin de localiser le service distant disponible sur le réseau.

Le registre RMI, généralement configuré sur le port 1099, retourne alors une référence vers le serveur RMI distant contenant l'implémentation réelle du service de tarification. Une fois la connexion établie, le client invoque à distance la méthode `estimatePrice()` en transmettant les paramètres nécessaires au calcul du prix.

Le serveur RMI exécute la méthode demandée et applique les règles de tarification dynamique définies dans le système. Le calcul peut prendre en compte plusieurs critères tels que la distance parcourue, le taux de remplissage du bus, la période de réservation ou encore les promotions actives.

Une fois le traitement terminé, le résultat calculé est renvoyé au client RMI à travers le réseau TCP/IP. Le stub récupère alors la réponse distante et la transmet au contrôleur REST qui retourne finalement le résultat au frontend sous forme de données JSON.

Cette architecture illustre pleinement le principe fondamental des systèmes distribués basé sur l'invocation distante de méthodes. Elle permet de répartir les traitements sur plusieurs services indépendants tout en assurant une communication transparente entre les différents composants du système.

L'utilisation de RMI présente également plusieurs avantages importants, notamment la modularité du système, la séparation des responsabilités métier ainsi que la possibilité de faire évoluer indépendamment les services distribués de la plateforme.

3.3.6 Résilience – Resilience4j

La figure suivante présente les mécanismes de résilience intégrés dans la plateforme afin d'assurer la continuité des services et la gestion des défaillances dans l'architecture distribuée.

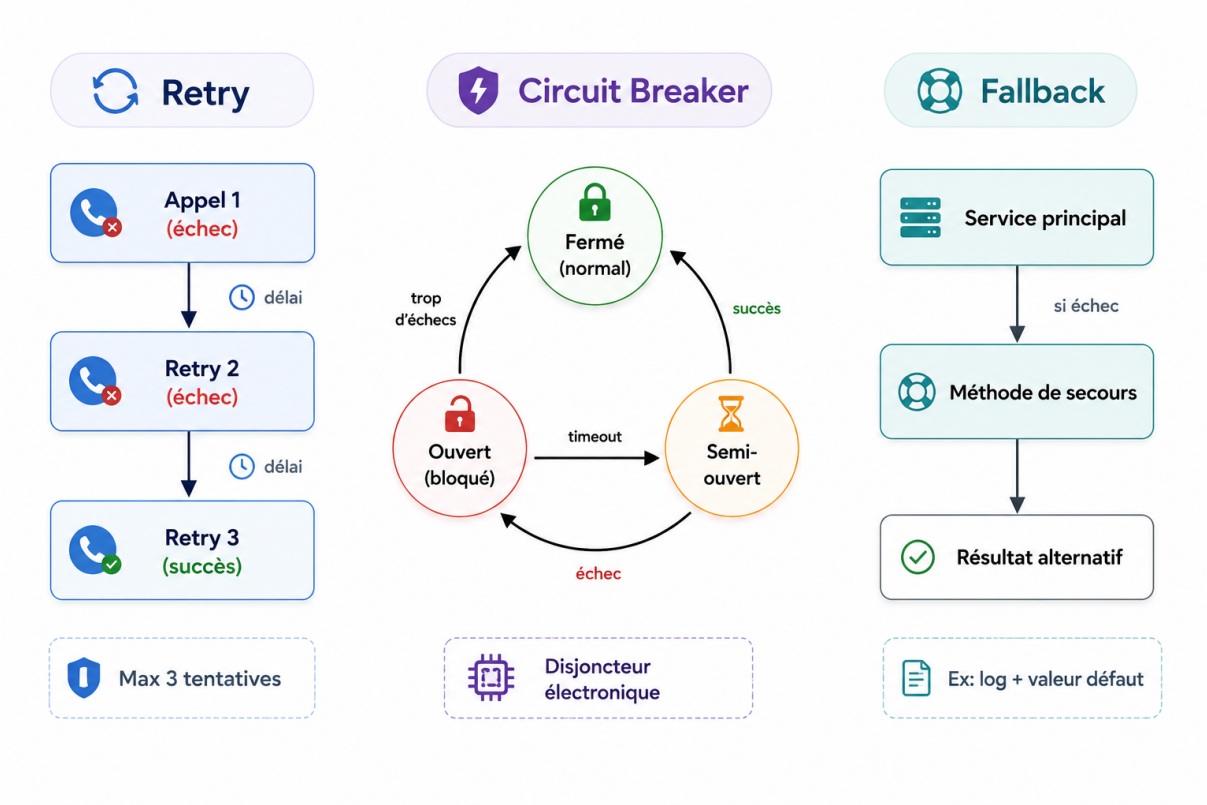


FIGURE 3.8 – Mécanismes de résilience : Retry, Circuit Breaker et Fallback

La figure 3.8 illustre les trois principaux mécanismes de résilience implémentés à l'aide de la bibliothèque Resilience4j afin d'améliorer la robustesse et la fiabilité de la plateforme distribuée.

Le premier mécanisme correspond au *Retry*. Ce mécanisme permet de relancer automatiquement un appel vers un service distant lorsqu'une erreur temporaire survient. En cas d'échec d'une requête, le système effectue jusqu'à trois nouvelles tentatives avec un

délai défini entre chaque essai. Cette stratégie permet de gérer efficacement les défaillances transitoires liées au réseau ou aux indisponibilités momentanées des services externes.

Le deuxième mécanisme présenté est le *Circuit Breaker*. Ce composant agit comme un disjoncteur intelligent chargé de protéger le système contre les cascades de défaillances. Lorsque le taux d'échecs d'un service dépasse un seuil prédéfini, le circuit passe automatiquement à l'état *Open*. Dans cet état, les nouvelles requêtes vers le service défaillant sont temporairement bloquées afin d'éviter une surcharge du système et de préserver les ressources disponibles.

Après une période de récupération, le circuit passe à l'état *Half-Open*. Cet état intermédiaire permet d'envoyer quelques requêtes de test afin de vérifier si le service distant est de nouveau opérationnel. Si les requêtes réussissent, le circuit revient à l'état *Closed* et les communications reprennent normalement. Dans le cas contraire, le circuit reste ouvert jusqu'à une nouvelle tentative de récupération.

Le troisième mécanisme correspond au *Fallback*. Cette stratégie permet de définir une méthode de secours exécutée automatiquement lorsqu'un appel distant échoue définitivement malgré les tentatives de reprise. Le système retourne alors une réponse alternative ou simplifiée afin d'éviter une interruption brutale du service pour l'utilisateur.

Dans le cadre de la plateforme intelligente de gestion de gare routière, le mécanisme de Fallback peut par exemple fournir une estimation tarifaire approximative, afficher un message d'indisponibilité temporaire ou retourner une réponse dégradée mais cohérente au lieu d'une erreur critique non gérée.

L'intégration combinée des mécanismes *Retry*, *Circuit Breaker* et *Fallback* permet ainsi d'assurer une meilleure tolérance aux pannes, une plus grande stabilité des services distribués ainsi qu'une amélioration globale de la disponibilité et de la résilience du système.

3.3.7 Pagination REST

Pour optimiser les performances du système lors de la manipulation de grandes quantités de données, le backend intègre un mécanisme de pagination basé sur Spring Data Pageable. Cette approche permet de limiter le nombre d'enregistrements retournés par les API REST afin de réduire la charge mémoire, d'améliorer les temps de réponse et d'optimiser les échanges réseau entre le frontend et le backend.

La figure suivante présente le fonctionnement général de la pagination REST implémentée dans la plateforme.

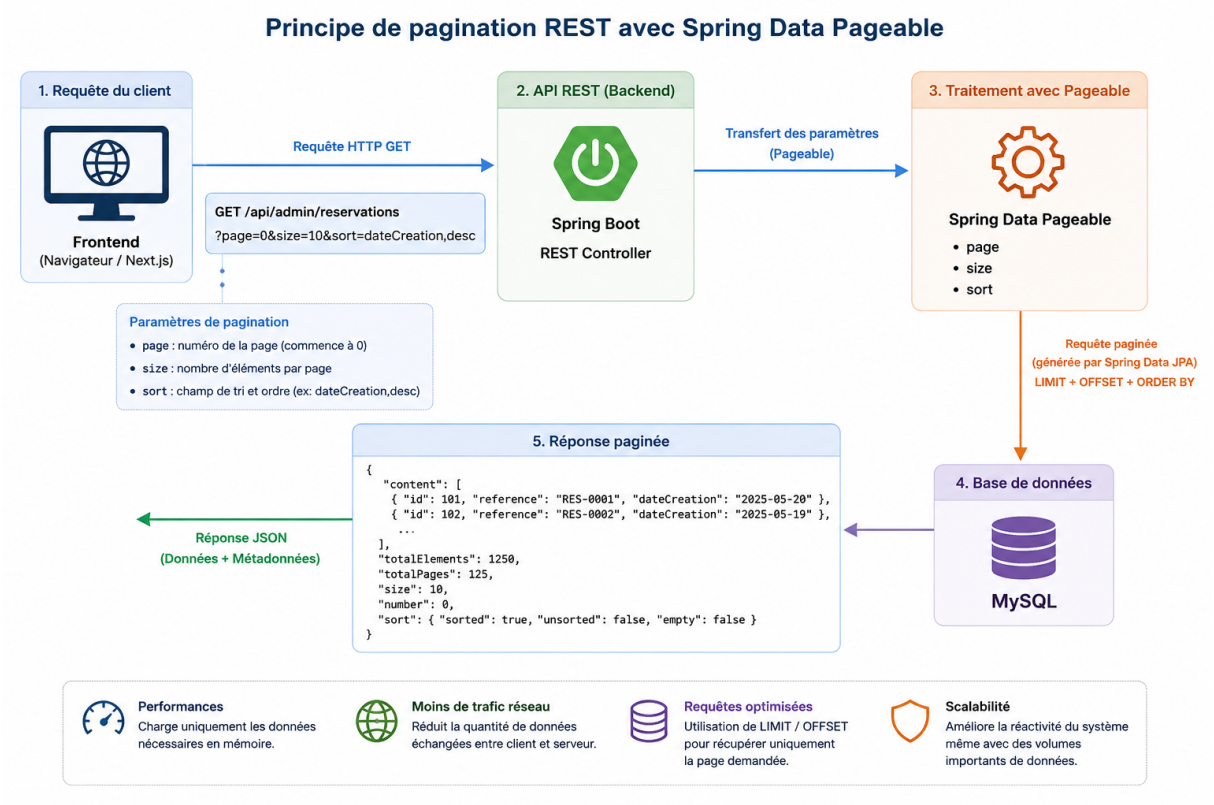


FIGURE 3.9 – Principe de pagination REST avec Spring Pageable

La figure 3.9 illustre le mécanisme de pagination utilisé dans les API REST du système. Lorsqu'un client effectue une requête vers une ressource volumineuse, le frontend transmet les paramètres de pagination tels que le numéro de page, la taille des données à récupérer ainsi que l'ordre de tri souhaité.

Le backend Spring Boot traite ensuite la requête à l'aide de l'interface Pageable fournie par Spring Data JPA. Seul un sous-ensemble des données est récupéré depuis la base MySQL, ce qui permet d'éviter le chargement complet de milliers d'enregistrements en mémoire.

La réponse retournée contient non seulement les données demandées, mais également plusieurs métadonnées importantes telles que le nombre total d'éléments, le nombre total de pages, la taille de chaque page ainsi que l'index de la page courante.

Cette approche améliore considérablement les performances globales de la plateforme, notamment pour les modules manipulant de grandes quantités de données comme les réservations, les trajets, les notifications ou les opérations OCR.

3.3.8 Notifications temps réel – WebSocket et STOMP

La figure suivante présente l'architecture de communication temps réel mise en œuvre dans la plateforme à l'aide des WebSocket et du protocole STOMP.

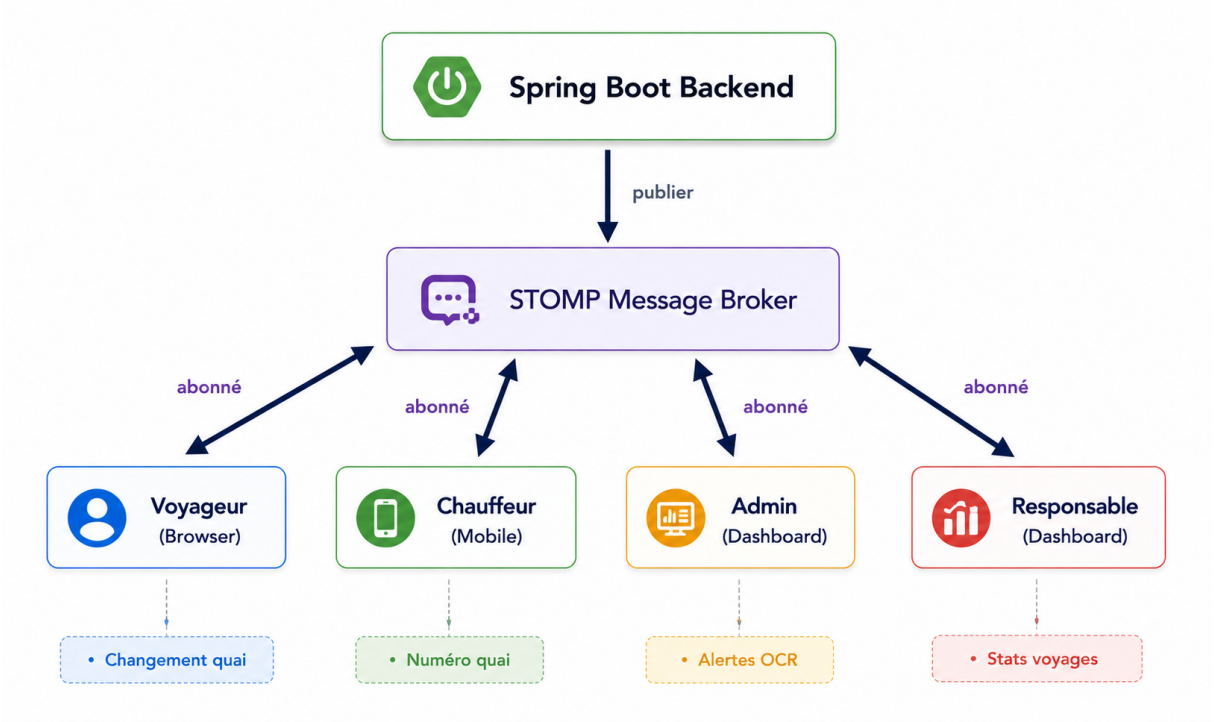


FIGURE 3.10 – Architecture WebSocket STOMP – Notifications temps réel

La figure 3.10 illustre l'architecture de communication temps réel utilisée au sein de la plateforme afin de diffuser instantanément les notifications et événements importants aux utilisateurs connectés.

Cette architecture repose sur la technologie WebSocket associée au protocole STOMP (*Simple Text Oriented Messaging Protocol*), permettant d'établir une communication bi-directionnelle permanente entre le serveur Spring Boot et les différents clients connectés.

Le backend Spring Boot agit comme serveur WebSocket et publie les messages de notification vers le broker STOMP intégré au système. Ce broker assure la gestion et la distribution des messages en temps réel vers les différents canaux de communication appelés *topics*.

Les différents utilisateurs de la plateforme, notamment les voyageurs, les chauffeurs, les administrateurs et les responsables de compagnie, s'abonnent aux topics correspondant à leurs besoins fonctionnels. Chaque client reçoit ainsi uniquement les notifications qui le concernent.

Contrairement aux requêtes HTTP classiques nécessitant des appels périodiques pour vérifier les nouvelles informations, l'architecture WebSocket adopte un modèle *push*. Le serveur envoie directement les messages aux clients dès qu'un événement se produit, sans attendre une requête explicite du frontend.

Cette approche permet d'améliorer considérablement la réactivité de la plateforme et de réduire la charge réseau liée aux requêtes répétitives. Les utilisateurs sont informés instantanément des événements importants tels que les changements de quai, les confirmations de réservation, les notifications OCR, les alertes administratives, les retards de

trajets ou encore les mises à jour liées aux voyages.

Dans le cadre du système, les notifications temps réel jouent un rôle essentiel dans la coordination des opérations de la gare routière intelligente. Elles permettent notamment d'assurer une meilleure synchronisation entre les différents acteurs du système ainsi qu'une diffusion rapide des informations critiques.

L'intégration combinée de WebSocket et du protocole STOMP permet ainsi de garantir une communication temps réel performante, scalable et adaptée aux besoins des systèmes distribués modernes.

3.4 Technologies utilisées

Le développement de la plateforme repose sur un ensemble de technologies modernes permettant d'assurer la performance, la sécurité, la maintenabilité ainsi que la scalabilité du système.

Spring Boot 3.4.4 constitue le framework principal du backend et facilite le développement des services REST, de la sécurité et des traitements métiers grâce à son architecture modulaire et ses mécanismes de configuration automatique. Le frontend a été développé avec Next.js et TypeScript afin de bénéficier du rendu côté serveur (SSR), d'une meilleure optimisation des performances et d'un typage fort du code. Tailwind CSS a été utilisé pour la conception de l'interface utilisateur moderne et responsive.

La persistance des données est assurée par MySQL 8.0 à travers Spring Data JPA et Hibernate, tandis que Docker permet la conteneurisation des différents services afin de garantir la portabilité et la reproductibilité de l'environnement de développement.

Le système intègre également plusieurs technologies issues des systèmes distribués telles qu'ActiveMQ, RMI, WebSocket/STOMP et Resilience4j afin d'assurer les communications asynchrones, les notifications temps réel ainsi que les mécanismes de résilience.

La figure suivante présente une synthèse des principales technologies utilisées dans le cadre du projet.

Technologies utilisées dans le projet

Technologie	Version	Utilisation dans le projet
Spring Boot	3.4.4	Backend REST, sécurité, services métier
Next.js	14+	Frontend, SSR, interface utilisateur
TypeScript	5+	Typage fort côté frontend
MySQL	8.0	Base de données relationnelle
Docker	25+	Conteneurisation de MySQL
Spring Security	6+	Authentification et autorisation
JWT	--	Tokens d'authentification stateless
ActiveMQ	5.18	Broker de messages JMS
Resilience4j	2+	Retry, Circuit Breaker, Fallback
RMI Java	--	Invocation de méthodes distantes
WebSocket/STOMP	--	Notifications temps réel
JUnit 5	5.10	Tests unitaires backend
Mockito	5+	Mocks pour les tests unitaires
Selenium	4+	Tests d'automatisation UI
SonarQube / SonarCloud	--	Analyse qualité et sécurité du code
Swagger/OpenAPI	3+	Documentation des APIs REST
iText	7+	Génération de tickets PDF
ZXing	--	Génération et scan de QR Code
Tailwind CSS	3+	Design et mise en page frontend

FIGURE 3.11 – Technologies utilisées dans le projet

La figure 3.11 illustre les différentes technologies intégrées dans la plateforme ainsi que leur rôle respectif dans l'architecture globale du système. Elle met en évidence les frameworks backend et frontend, les outils de sécurité, les technologies de communication distribuée, les outils de tests, ainsi que les solutions utilisées pour la documentation, la génération de fichiers PDF et la gestion des QR Codes.

Conclusion

Ce chapitre a présenté l'architecture technique globale du système ainsi que les technologies utilisées. Nous avons décrit en détail les composants des systèmes distribués intégrés dans la solution, notamment la communication synchrone REST et RMI, la communication asynchrone ActiveMQ/JMS, les mécanismes de résilience Resilience4j, la pagination REST et les notifications temps réel WebSocket. Le chapitre suivant présentera la réalisation concrète du projet à travers les différents sprints de développement.

Chapitre 4

Réalisation du Projet

Introduction

Ce chapitre présente la réalisation concrète du projet à travers les sept sprints de développement effectués durant le semestre. Nous y décrivons les fonctionnalités développées dans chaque sprint, les interfaces réalisées, les tests effectués ainsi que l'analyse qualité du code avec SonarQube.

4.1 Sprint 1 : Infrastructure et Authentification

Le premier sprint a été consacré à la mise en place de l'infrastructure de base du système ainsi qu'à l'implémentation du mécanisme d'authentification sécurisée. Les fonctionnalités réalisées comprennent l'initialisation et la configuration du projet Spring Boot avec Maven, la configuration du projet Next.js avec TypeScript et Tailwind CSS, la mise en place de la base de données MySQL conteneurisée via Docker, l'implémentation de l'authentification JWT avec Spring Security, la gestion des rôles (ADMIN, VOYAGEUR, CHAUFFEUR, RESPONSABLE) via RBAC, la sécurisation des APIs REST avec les annotations `@PreAuthorize`, et le hachage des mots de passe avec BCrypt.

4.1.1 Structure du projet Backend

La figure suivante présente l'organisation générale du projet backend développé avec Spring Boot ainsi que la répartition des différents composants techniques de l'application.

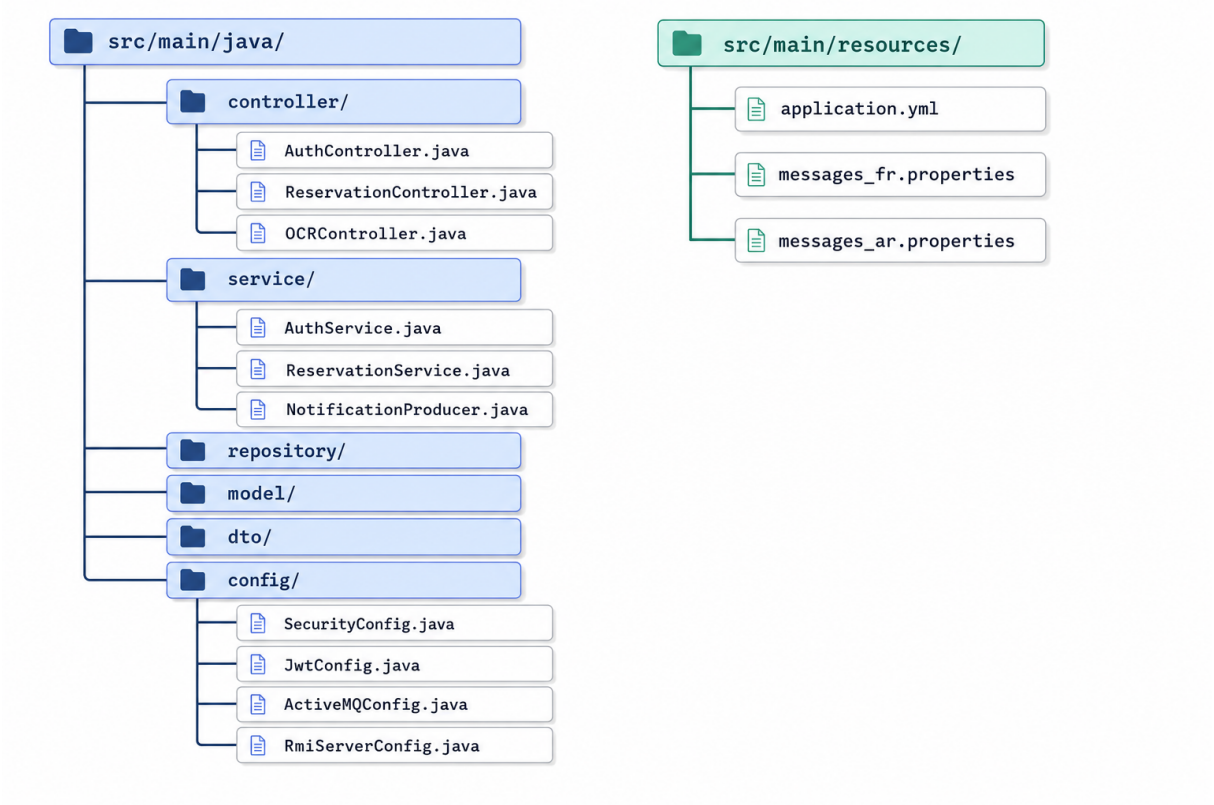


FIGURE 4.1 – Structure du projet Backend Spring Boot

La figure 4.1 illustre l'organisation interne du code source backend selon l'architecture standard recommandée par Spring Boot. Cette structure repose sur une séparation claire des responsabilités afin d'améliorer la maintenabilité, la modularité et l'évolutivité du projet.

Les contrôleurs REST sont regroupés dans le package *controller*. Ils assurent la réception des requêtes HTTP provenant du frontend Next.js ainsi que l'exposition des différentes API REST de la plateforme.

Les traitements métiers sont centralisés dans le package *service*. Cette couche contient l'ensemble de la logique fonctionnelle du système, notamment la gestion des réservations, des trajets, des paiements, des notifications, des opérations OCR et des mécanismes distribués.

Les accès à la base de données sont gérés à travers les repositories Spring Data JPA regroupés dans le package *repository*. Ces composants facilitent les opérations CRUD et les interactions avec les entités persistées dans MySQL.

Les différentes entités métier du système sont définies dans le package *entity*. Elles représentent les objets principaux manipulés par la plateforme tels que les utilisateurs, les trajets, les réservations, les tickets ou les opérations OCR.

Les classes de configuration sont organisées dans le package *config*. Elles permettent de centraliser les configurations liées à Spring Security, JWT, WebSocket, ActiveMQ, RMI et aux différents composants techniques de l'application.

Le projet contient également des fichiers de configuration YAML permettant de centraliser les paramètres d'exécution du système, notamment les configurations de base de données, les ports réseau, les paramètres JWT et les propriétés des services distribués.

Enfin, les fichiers de messages utilisés pour l'internationalisation permettent d'assurer la prise en charge de la bilingualité français/arabe intégrée dans la plateforme.

Cette organisation modulaire facilite considérablement le développement collaboratif, les opérations de maintenance ainsi que l'évolution future de l'architecture backend Spring Boot.

4.2 Sprint 2 : Gestion Administrateur et Dashboard

Le deuxième sprint a été consacré au développement des fonctionnalités destinées à l'administrateur du système. L'objectif principal de ce sprint était de mettre en place un espace de supervision permettant à l'administrateur de gérer les principaux éléments de la gare routière et d'assurer le suivi global de l'activité de la plateforme.

L'administrateur dispose d'un dashboard principal qui lui permet de consulter plusieurs indicateurs importants, notamment le nombre de voyageurs, le nombre de compagnies, les trajets disponibles, les revenus générés ainsi que les statistiques liées à l'activité de la gare. Ce tableau de bord offre une vision synthétique de l'état général du système et facilite la prise de décision.

Ce sprint a également permis d'intégrer la gestion des compagnies de transport. L'administrateur peut ajouter une nouvelle compagnie, consulter ses informations, modifier ses données et l'associer à un responsable compagnie. Chaque responsable est lié à une compagnie précise, ce qui permet d'organiser clairement les rôles et les responsabilités au sein de la plateforme.

La gestion des quais fait également partie des fonctionnalités importantes développées durant ce sprint. L'administrateur peut organiser les quais disponibles dans la gare et gérer leur affectation aux compagnies ou aux bus selon les besoins opérationnels. Cette fonctionnalité permet d'améliorer la gestion de l'espace physique de la gare et d'éviter les conflits d'affectation.

L'administrateur dispose aussi d'un module de communication interne lui permettant d'échanger des messages avec les responsables des compagnies. Cette messagerie facilite la coordination entre l'administration de la gare et les compagnies de transport, notamment en cas de changement de planning, de retard, d'incident ou de besoin d'information.

Un autre élément important de ce sprint concerne l'intégration du module OCR dans l'espace administrateur. Ce module permet de traiter les images des plaques d'immatriculation des bus afin d'identifier automatiquement la compagnie associée et de faciliter l'attribution d'un quai disponible.

Enfin, le dashboard d'analyse financière a été intégré afin de permettre à l'adminis-

trateur de suivre les revenus, les paiements, les remboursements et l'évolution globale des ventes. Des fonctionnalités d'exportation des rapports au format PDF et CSV ont également été prévues afin de faciliter l'analyse et l'archivage des données importantes.

Ainsi, ce sprint a permis de construire un espace administrateur complet, regroupant la gestion des compagnies, des responsables, des quais, de la communication interne, du module OCR et du suivi financier de la plateforme.

4.3 Sprint 3 : Module OCR et Gestion Chauffeur

Le troisième sprint a été consacré au développement du module OCR ainsi qu'aux fonctionnalités destinées aux chauffeurs. Ce sprint représente une étape importante dans la réalisation du projet, car il introduit une automatisation intelligente dans la gestion des bus entrant dans la gare routière.

Le module OCR permet la reconnaissance automatique des plaques d'immatriculation des bus à partir d'une image. Dans le cadre du projet, cette reconnaissance est simulée par l'upload d'une image depuis l'espace administrateur. L'objectif est d'identifier rapidement le bus entrant, de déterminer la compagnie à laquelle il appartient, puis de faciliter son affectation à un quai disponible.

Le processus commence par la soumission de l'image de la plaque d'immatriculation. Cette image est ensuite traitée à l'aide d'OpenCV afin d'améliorer sa qualité. Le prétraitement permet notamment d'ajuster le contraste, de réduire le bruit et de réaliser la binarisation de l'image afin de préparer la phase de reconnaissance.

Après cette étape, le modèle YOLOv8 est utilisé pour détecter et reconnaître les caractères présents sur la plaque d'immatriculation. Une fois le numéro de plaque identifié, le système recherche la compagnie associée au bus. Cette identification permet ensuite d'attribuer automatiquement un quai disponible selon les règles de gestion définies dans la plateforme.

Après l'attribution du quai, une notification est envoyée au chauffeur via ActiveMQ. Ce mécanisme permet d'assurer une communication asynchrone entre le système et le chauffeur concerné. Le démarrage de la facturation du stationnement peut également être déclenché automatiquement afin de suivre la durée d'occupation du quai.

En parallèle, plusieurs fonctionnalités ont été développées pour le chauffeur. Celui-ci peut consulter ses trajets du jour, scanner et valider les tickets QR Code lors de l'embarquement, gérer les bagages des voyageurs, valider les jalons d'arrêt, déclarer le départ du bus afin de libérer le quai, et signaler les incidents éventuels.

Le scan du QR Code permet de vérifier rapidement la validité du ticket avant l'embarquement. Cette fonctionnalité renforce la sécurité du processus, réduit les risques liés aux tickets invalides ou déjà utilisés, et facilite le contrôle des voyageurs.

Ainsi, ce sprint a permis de combiner l'automatisation OCR, la gestion intelligente

des quais et les fonctionnalités opérationnelles du chauffeur. Il contribue directement à améliorer la fluidité de gestion des bus au sein de la gare routière.

4.4 Sprint 4 : Réservation Voyageur et Tickets

Le quatrième sprint a été consacré au développement du module de réservation destiné aux voyageurs. Ce sprint représente l'une des parties les plus importantes de la plateforme, car il correspond au parcours principal suivi par l'utilisateur pour rechercher un trajet, réserver une place et obtenir son ticket électronique.

Le processus de réservation commence par la recherche d'un trajet. Le voyageur peut préciser la ville de départ, la ville d'arrivée, la date du voyage ainsi que certains filtres afin d'obtenir une liste de trajets correspondant à ses besoins. Les résultats affichés permettent ensuite de comparer les différentes compagnies disponibles selon plusieurs critères, notamment le prix, l'horaire, le rating et les avis des voyageurs.

Après avoir choisi un trajet, le système propose une simulation tarifaire permettant au voyageur d'avoir une estimation claire du montant à payer. L'utilisateur peut ensuite sélectionner le type de réservation souhaité : réservation individuelle, réservation avec accompagnants ou réservation pour une autre personne.

La plateforme prend également en compte les informations relatives aux passagers. Le voyageur peut définir le lien entre les personnes concernées par la réservation ainsi que certaines préférences de voisinage. Cette fonctionnalité permet d'améliorer l'organisation des sièges et d'offrir une expérience plus adaptée aux besoins des voyageurs.

Le choix des sièges s'effectue à travers un plan interactif du bus. Chaque siège possède un état représenté par une couleur spécifique : vert pour les sièges disponibles, bleu pour les sièges réservés par un homme, orange pour les sièges réservés par une femme, rouge pour les sièges associés à un enfant, et gris pour les sièges temporairement verrouillés. Cette représentation visuelle facilite la sélection et permet d'éviter les erreurs lors de la réservation.

Une fois la réservation confirmée et le paiement validé, le système génère automatiquement un ticket au format PDF contenant un QR Code. Ce ticket est envoyé par email au voyageur et peut ensuite être présenté au chauffeur lors de l'embarquement pour validation.

Ainsi, ce sprint a permis de mettre en place un parcours de réservation complet, reliant plusieurs modules essentiels du système : recherche de trajets, comparaison des compagnies, sélection des sièges, paiement, génération du ticket et validation par QR Code.

4.5 Sprint 5 : Paiement et Dashboard Financier

Le cinquième sprint a été consacré à l'implémentation des fonctionnalités liées au paiement, aux annulations, aux remboursements et au suivi financier de la plateforme. Cette étape a permis de renforcer la fiabilité du processus de réservation et d'assurer une meilleure gestion des transactions.

Le paiement en ligne sécurisé permet au voyageur de finaliser sa réservation directement depuis la plateforme. Lorsqu'un siège est sélectionné, il est temporairement verrouillé afin d'éviter qu'un autre utilisateur puisse réserver la même place au même moment. Si le paiement n'est pas effectué dans le délai autorisé, le siège est automatiquement libéré et redevient disponible pour les autres voyageurs.

La gestion des annulations a été mise en place avec un système de remboursement progressif. Le voyageur peut bénéficier d'un remboursement de 75% si l'annulation est effectuée avant 48 heures, de 50% le jour même, et d'aucun remboursement lorsque les conditions d'annulation ne sont plus respectées. Cette règle permet d'encadrer les annulations tout en protégeant les intérêts des compagnies de transport.

Le système permet également la modification de la date ou de l'heure du voyage. La première modification est gratuite, tandis qu'une deuxième modification peut entraîner des frais supplémentaires. Cette fonctionnalité offre plus de flexibilité au voyageur tout en limitant les changements excessifs.

Le voyageur peut aussi re-télécharger ses tickets depuis son historique et déclarer un bagage perdu ou endommagé afin d'en assurer le suivi. Ces fonctionnalités permettent d'améliorer le service après-réservation et de centraliser les informations liées au voyage.

Des notifications automatiques sont envoyées aux voyageurs à différentes étapes importantes. Le système envoie notamment des rappels avant le départ, à 24 heures, 2 heures, 30 minutes et 5 minutes. Ces rappels sont gérés à l'aide d'un scheduler Cron, ce qui permet d'automatiser l'envoi des notifications selon l'heure du trajet.

Le dashboard financier permet à l'administrateur de suivre les revenus générés, l'évolution des ventes, les paiements effectués, les remboursements et les indicateurs financiers globaux. L'estimation des prix est assurée par un service RMI, ce qui permet d'isoler une partie du calcul tarifaire dans un service distribué.

Ainsi, ce sprint a permis de sécuriser le cycle complet de réservation en intégrant la logique de paiement, de verrouillage temporaire des sièges, de remboursement, de notification et de suivi financier.

4.6 Sprint 6 : Gestion Responsable Compagnie

Le sixième sprint a été dédié au développement des fonctionnalités destinées au responsable compagnie. Cet acteur joue un rôle essentiel dans la plateforme, car il gère les

ressources propres à sa compagnie de transport tout en restant connecté à l'administration centrale de la gare routière.

La première fonctionnalité développée concerne la gestion du parc de bus. Le responsable compagnie peut ajouter, consulter, modifier et supprimer les bus appartenant à sa compagnie. Il peut également renseigner les informations relatives à la qualité du bus, ses options, son état et les opérations de maintenance préventive prévues.

La planification des trajets constitue une autre partie importante de ce sprint. Le responsable peut créer des trajets, définir les horaires de départ et d'arrivée, préciser les arrêts de pause, gérer les lignes et configurer des horaires récurrents. Cette fonctionnalité permet d'organiser l'offre de transport proposée aux voyageurs de manière structurée.

Le responsable compagnie peut également gérer les comptes chauffeurs associés à sa compagnie. Cette gestion permet d'affecter les chauffeurs aux trajets correspondants et de mieux suivre les opérations réalisées sur le terrain.

La plateforme intègre aussi la création et la gestion des codes promotionnels. Le responsable peut proposer des réductions temporaires afin d'attirer davantage de voyageurs ou de promouvoir certains trajets. En complément, la tarification dynamique permet d'adapter les prix selon plusieurs critères, comme le taux de remplissage du bus ou la proximité de la date de départ.

En cas de retard, d'annulation ou de changement de quai, le responsable peut envoyer des notifications aux voyageurs concernés via ActiveMQ. Ce mécanisme assure une communication rapide et asynchrone, ce qui améliore la réactivité de la plateforme face aux changements opérationnels.

Le responsable dispose également d'un accès aux statistiques de sa compagnie. Il peut consulter les données liées aux trajets, aux ventes, aux remboursements et aux performances globales, puis exporter ces informations au format PDF ou Excel. Une messagerie interne bidirectionnelle permet enfin de communiquer directement avec l'administrateur de la gare.

Ainsi, ce sprint a permis de fournir aux compagnies de transport un espace de gestion complet, couvrant les bus, les trajets, les chauffeurs, les promotions, les notifications et le suivi statistique.

4.7 Sprint 7 : Tests, Qualité du Code et Optimisation

Le dernier sprint a été consacré à la validation finale du système à travers les tests, l'analyse de la qualité du code et les optimisations techniques. L'objectif principal de cette phase était de vérifier le bon fonctionnement des modules développés, d'assurer la stabilité de l'application et d'améliorer la qualité globale du code source.

4.7.1 Tests unitaires avec JUnit 5 et Mockito

Les tests unitaires ont été réalisés avec JUnit 5 et Mockito afin de valider les traitements métiers du backend de manière isolée. Cette approche permet de tester chaque composant indépendamment tout en simulant les dépendances externes grâce aux mocks Mockito.

Les services testés comprennent notamment *AuthService*, *OCRService*, *ReservationService*, *PaiementService* et *NotificationService*. Les scénarios nominaux ainsi que les cas d'erreur ont été pris en compte afin de garantir la robustesse des traitements métiers.

Le listing suivant présente un exemple de test unitaire réalisé avec JUnit 5 et Mockito pour le service de réservation.

```
@ExtendWith(MockitoExtension.class)
class ReservationServiceTest {

    @Mock
    private ReservationRepository reservationRepository;

    @InjectMocks
    private ReservationService reservationService;

    @Test
    void testCreateReservation_Success() {

        ReservationDTO dto = new ReservationDTO();
        dto.setNbPassagers(2);

        when(reservationRepository.save(any()))
            .thenReturn(new Reservation());

        Reservation result =
            reservationService.createReservation(dto);

        assertNotNull(result);

        verify(reservationRepository, times(1))
            .save(any());
    }
}
```

Listing 4.1 – Exemple de test unitaire avec JUnit 5 et Mockito

Ce test vérifie le bon fonctionnement de la méthode de création de réservation. Mockito

permet de simuler le comportement du repository sans accéder directement à la base de données. Le test contrôle que la réservation est correctement créée et que la méthode `save()` du repository est invoquée une seule fois.

4.7.2 Tests d'automatisation UI avec Selenium

Les tests d'automatisation de l'interface utilisateur ont été réalisés avec Selenium WebDriver afin de valider les scénarios fonctionnels de la plateforme à travers un navigateur réel. Cette approche permet de reproduire les interactions des utilisateurs et de vérifier le comportement des fonctionnalités critiques du système.

Les tests couvrent notamment les processus d'authentification, de réservation, de paiement, de gestion OCR, de scan QR Code ainsi que les opérations administratives et les traitements liés aux trajets.

La figure 4.2 présente une synthèse des principaux scénarios automatisés avec Selenium WebDriver. Elle montre les différents parcours fonctionnels testés afin de vérifier la cohérence du système côté interface utilisateur.



FIGURE 4.2 – Scénarios de tests automatisés avec Selenium

Ces scénarios concernent plusieurs modules importants de la plateforme, notamment l'authentification des utilisateurs, les opérations OCR, les réservations multi-passagers, les paiements, la gestion des tickets électroniques, les fonctionnalités des responsables de compagnie ainsi que les opérations d'embarquement réalisées par les chauffeurs.

L'automatisation de ces tests contribue à détecter rapidement les régressions fonctionnelles et à garantir la stabilité de l'application lors des futures évolutions.

Le listing suivant présente un exemple de test automatisé réalisé avec Selenium WebDriver pour valider le processus de connexion d'un administrateur à la plateforme.

```
@Test
public void testAdminLogin() {

    WebDriver driver = new ChromeDriver();

    driver.get("http://localhost:3000/login");

    WebElement emailField =
        driver.findElement(By.id("email"));

    WebElement passwordField =
        driver.findElement(By.id("password"));

    emailField.sendKeys("admin@gare.ma");

    passwordField.sendKeys("Admin@1234");

    driver.findElement(By.id("submit-btn"))
          .click();

    WebDriverWait wait =
        new WebDriverWait(driver,
            Duration.ofSeconds(5));

    WebElement dashboard = wait.until(
        ExpectedConditions
            .visibilityOfElementLocated(
                By.id("admin-dashboard")));

    assertTrue(dashboard.isDisplayed());

    driver.quit();
}
```

Listing 4.2 – Exemple de test Selenium – Connexion administrateur

Ce test automatise le processus de connexion administrateur. Le script ouvre la page de connexion, saisit les identifiants, soumet le formulaire puis attend l'affichage du tableau de bord administrateur à l'aide de `WebDriverWait`.

La vérification finale confirme que l'authentification a réussi et que l'utilisateur est

correctement redirigé vers son espace. Ce type de test permet donc de valider l'interaction entre le frontend Next.js, le backend Spring Boot et le mécanisme de sécurité basé sur JWT.

4.7.3 Analyse qualité du code – SonarQube et SonarCloud

L'analyse de la qualité et de la sécurité du code source a été réalisée avec SonarQube Scanner en local et SonarCloud pour le suivi de la qualité du projet. Ces outils permettent de détecter les bugs potentiels, les vulnérabilités, les *code smells*, la dette technique ainsi que les duplications de code.

Configuration SonarQube Scanner

Le fichier suivant présente la configuration utilisée pour lancer l'analyse SonarQube Scanner sur le projet backend. Cette configuration précise notamment les répertoires sources, les tests, les fichiers compilés ainsi que le rapport de couverture généré par JaCoCo.

```
sonar.projectKey=gare-routiere
sonar.projectName=Gestion Intelligente Gare Routiere
sonar.projectVersion=1.0
sonar.sources=src/main/java
sonar.tests=src/test/java
sonar.java.binaries=target/classes
sonar.language=java
sonar.sourceEncoding=UTF-8
sonar.coverage.jacoco.xmlReportPaths=
    target/site/jacoco/jacoco.xml
```

Listing 4.3 – Configuration SonarQube – sonar-project.properties

Résultats de l'analyse SonarCloud

Après la configuration de l'analyse, le projet a été évalué à l'aide de SonarCloud afin d'obtenir une vision globale de la qualité du code. Le tableau de bord généré permet de consulter les indicateurs principaux liés à la sécurité, la fiabilité, la maintenabilité, la couverture des tests et la duplication.

La figure 4.3 présente les résultats obtenus après l'analyse du projet sur SonarCloud. Elle permet de visualiser rapidement le statut du *Quality Gate* ainsi que les principales métriques de qualité.

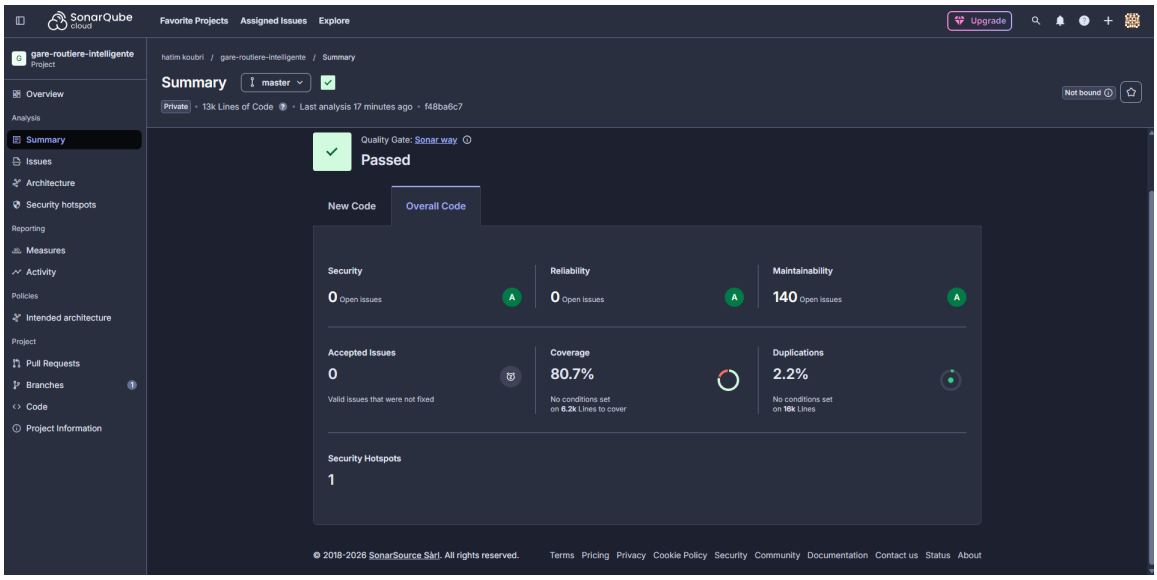


FIGURE 4.3 – Tableau de bord SonarCloud – Analyse qualité du projet

Les résultats montrent que le projet a passé avec succès le *Quality Gate*. Aucun problème ouvert n'a été détecté au niveau de la sécurité et de la fiabilité. La maintenabilité obtient également la note **A**, malgré la présence de 140 problèmes ouverts correspondant principalement à des améliorations possibles du code.

La couverture des tests atteint **80,7%**, ce qui indique qu'une partie importante du code est vérifiée par des tests automatisés. Le taux de duplication est de **2,2%**, ce qui reste faible pour un projet contenant environ 13 000 lignes de code. Enfin, un seul *Security Hotspot* a été détecté et nécessite une vérification manuelle afin de confirmer qu'il ne représente pas un risque réel.

Cette analyse confirme donc la bonne qualité générale du projet, tout en mettant en évidence quelques points d'amélioration liés principalement à la maintenabilité et à la vérification de sécurité.

4.7.4 Optimisations réalisées

Plusieurs optimisations ont été intégrées afin d'améliorer les performances et la stabilité du système. La réduction des requêtes N+1 a été réalisée grâce à l'utilisation des annotations `@EntityGraph` et des requêtes `JOIN FETCH` dans Spring Data JPA.

La pagination a également été mise en place sur les endpoints retournant des listes importantes afin d'éviter le chargement excessif de données. L'optimisation des requêtes SQL a été renforcée par l'ajout d'index sur les colonnes fréquemment utilisées dans les recherches.

Enfin, la mise en cache de certaines données statiques, comme les lignes et les compagnies, permet de réduire les appels répétitifs à la base de données et d'améliorer le temps de réponse global de l'application.

4.8 Interfaces de l'application

Cette section présente les principales interfaces graphiques développées pour la plateforme intelligente de gestion de gare routière. Les différentes interfaces illustrent le parcours complet des utilisateurs à travers le système, depuis l'authentification jusqu'aux opérations de réservation, de paiement, de contrôle d'embarquement et de supervision administrative.

4.8.1 Interface d'authentification

Le parcours utilisateur débute par l'interface d'authentification qui permet aux différents acteurs du système d'accéder de manière sécurisée à la plateforme selon leurs rôles et permissions.

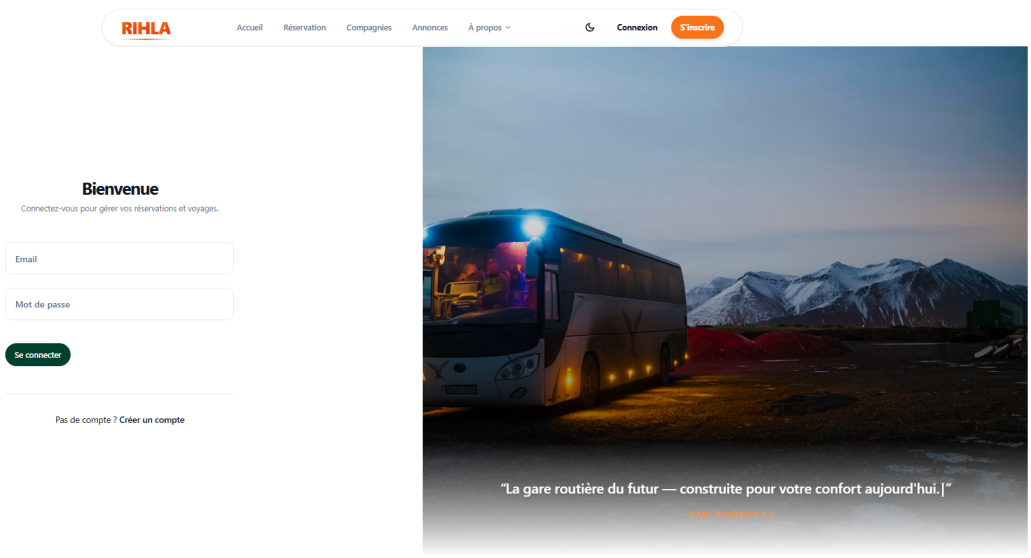


FIGURE 4.4 – Interface d'authentification – Page de connexion

La figure 4.4 présente l'interface de connexion de la plateforme. Le formulaire permet aux utilisateurs de s'authentifier à l'aide de leur adresse email et de leur mot de passe. Le système applique une authentification sécurisée basée sur JWT et Spring Security afin de protéger les accès aux différentes ressources de l'application.

Après validation des identifiants, chaque utilisateur est automatiquement redirigé vers son espace correspondant selon son rôle, notamment administrateur, voyageur, chauffeur ou responsable compagnie.

4.8.2 Dashboard Administrateur

Une fois authentifié, l'administrateur accède au tableau de bord principal lui permettant de superviser l'ensemble des opérations de la gare routière intelligente.

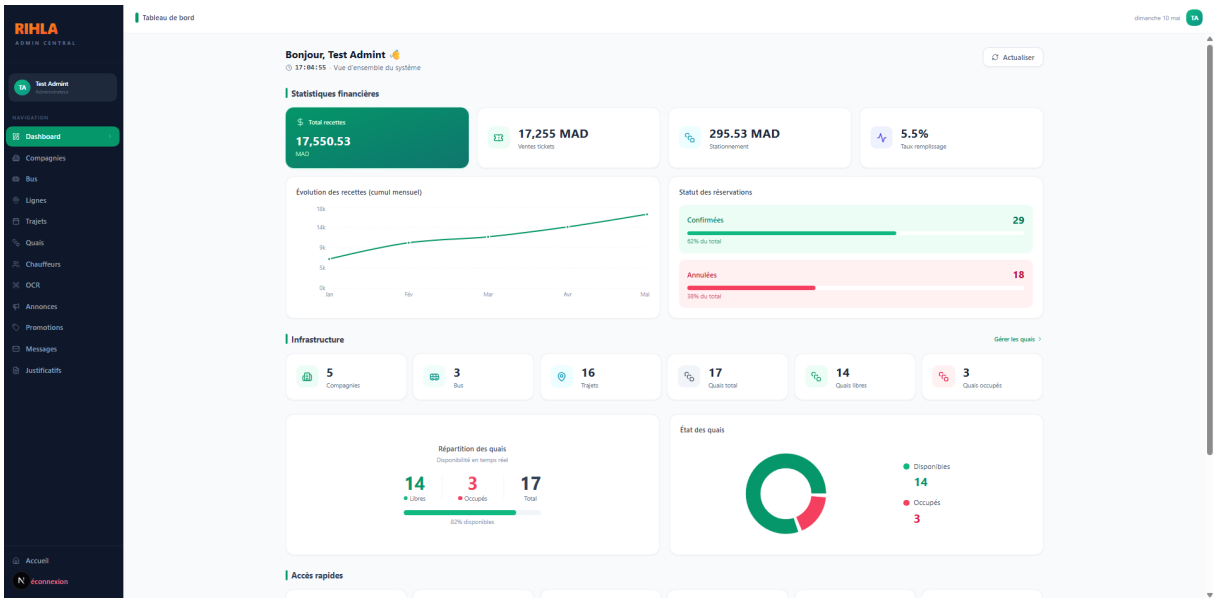


FIGURE 4.5 – Dashboard Administrateur – Vue de supervision globale

La figure 4.5 illustre le tableau de bord administrateur de la plateforme. Cette interface centralise les principaux indicateurs de performance du système tels que le nombre de voyageurs, les revenus générés, les trajets programmés ainsi que l'état d'occupation des quais.

Le dashboard intègre également plusieurs graphiques statistiques permettant de visualiser l'évolution des réservations, des revenus et des opérations réalisées au sein de la plateforme. Une zone de supervision affiche enfin les événements récents nécessitant une attention particulière de l'administrateur.

4.8.3 Module de réservation Voyageur

Après l'authentification, les voyageurs peuvent accéder au module de réservation afin de rechercher les trajets disponibles et effectuer leurs réservations en ligne.

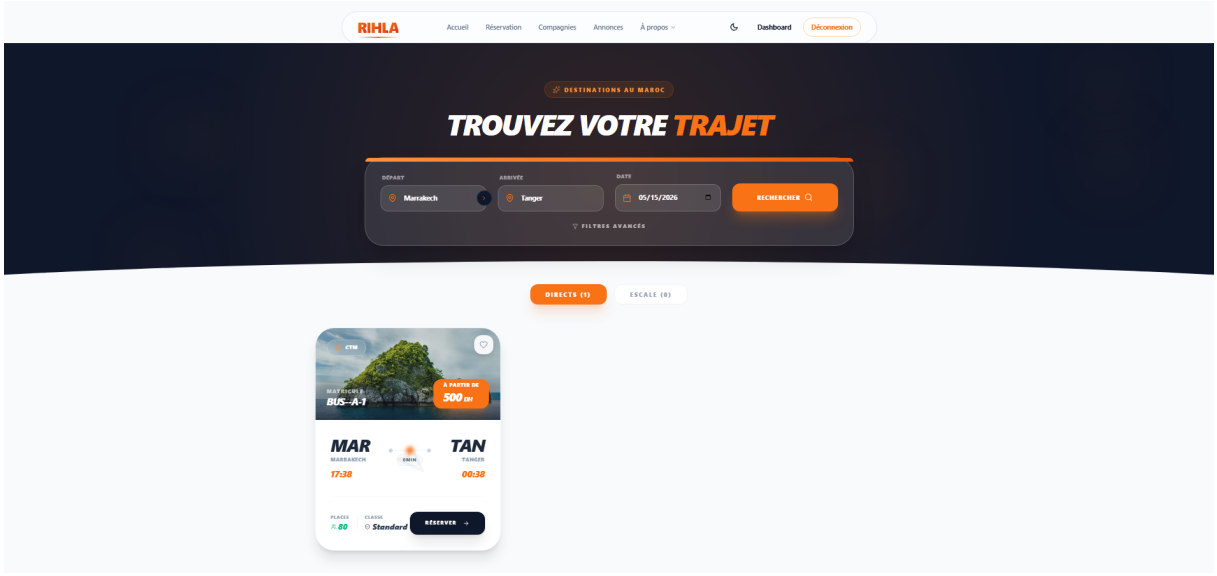


FIGURE 4.6 – Interface de réservation – Recherche de trajet et sélection des sièges

La figure 4.6 présente l'interface de réservation destinée aux voyageurs. Le formulaire de recherche permet de filtrer les trajets disponibles selon plusieurs critères tels que la ville de départ, la destination, la date du voyage ou les préférences de trajet.

Le système affiche ensuite les trajets disponibles ainsi qu'un plan interactif de sélection des sièges. Chaque place est représentée visuellement avec un code couleur indiquant son état en temps réel : disponible, réservée, verrouillée ou attribuée à une catégorie spécifique de voyageurs.

Cette interface permet également la réservation multi-passagers ainsi que le calcul dynamique du montant total du trajet.

4.8.4 Interface de paiement sécurisé

Après la sélection des sièges, le voyageur accède au module de paiement sécurisé permettant de finaliser la réservation du trajet.

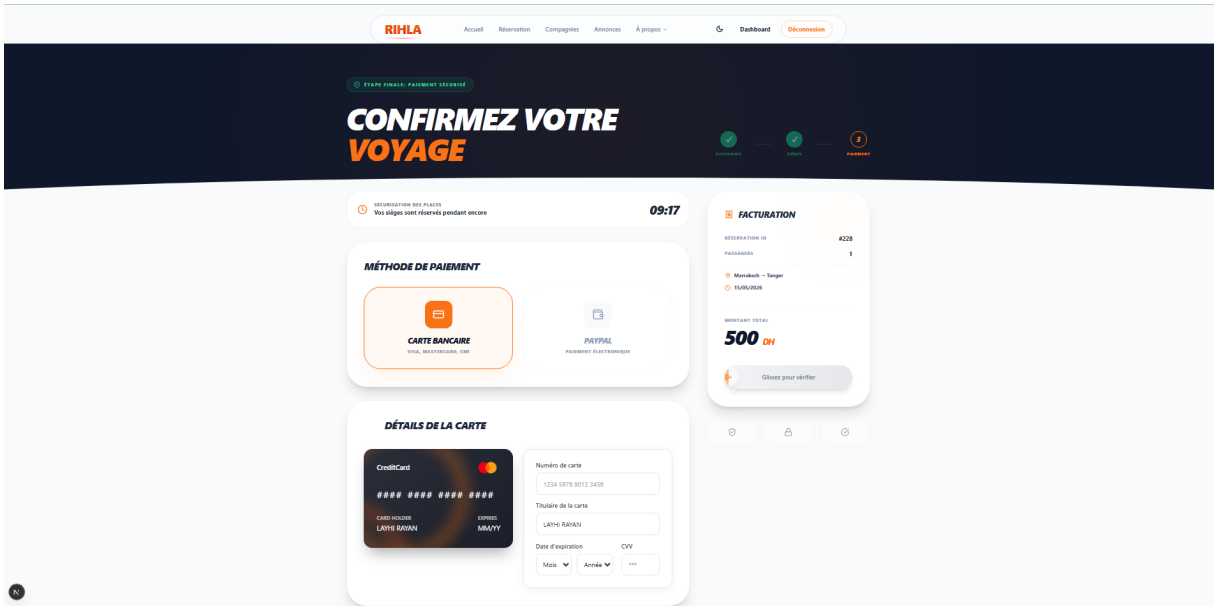


FIGURE 4.7 – Interface de paiement sécurisé

La figure 4.7 illustre l'interface de paiement intégrée dans la plateforme. Cette interface affiche un récapitulatif complet de la réservation incluant les informations du trajet, les sièges sélectionnés ainsi que le montant total à payer.

Afin d'éviter les conflits de réservation simultanée, le système applique un mécanisme de réservation temporaire des sièges. Lorsqu'un voyageur sélectionne une ou plusieurs places et accède à l'interface de paiement, les sièges concernés sont automatiquement verrouillés pendant une durée de 10 minutes. Durant cette période, aucun autre utilisateur ne peut réserver les mêmes places.

Ce mécanisme permet de garantir l'unicité des réservations et d'éviter que deux voyageurs réservent simultanément le même siège. Si le paiement est validé avant l'expiration du délai, la réservation devient définitive et les sièges sont confirmés. En revanche, si le voyageur ne finalise pas le paiement dans le temps imparti, les sièges sont automatiquement libérés et redeviennent disponibles pour les autres utilisateurs de la plateforme.

Le paiement est effectué à travers une passerelle sécurisée garantissant la protection des données bancaires des utilisateurs. Une fois la transaction validée, le système génère automatiquement le ticket électronique associé à la réservation.

4.8.5 Interface du ticket électronique et QR Code

Après confirmation du paiement, la plateforme génère automatiquement un ticket électronique contenant les informations du voyage ainsi qu'un QR Code unique.

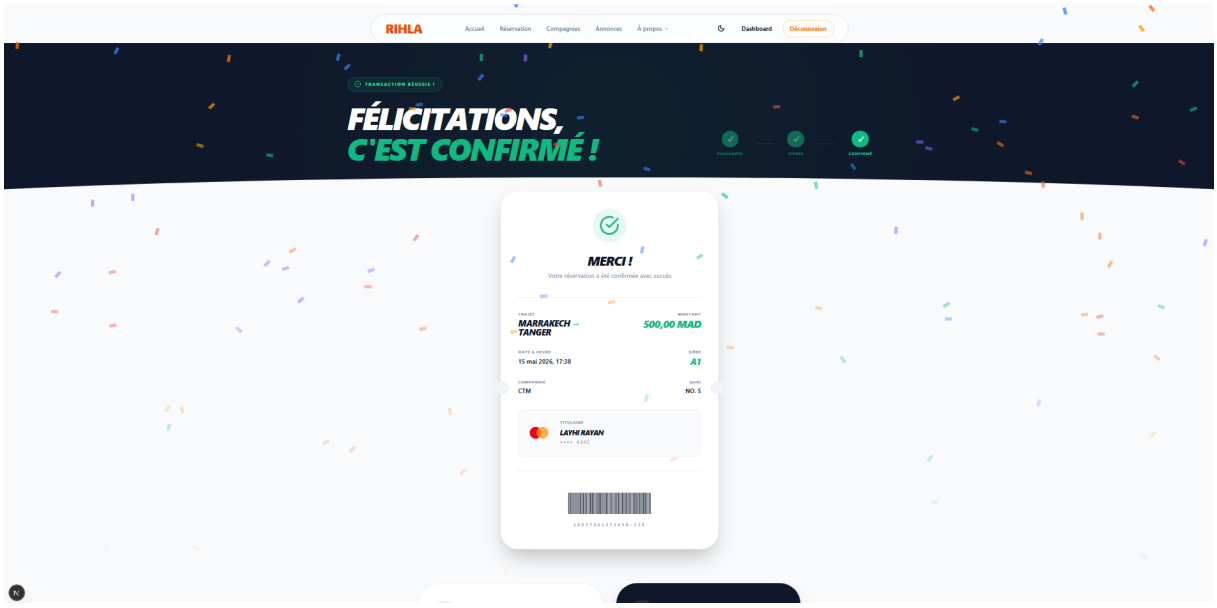


FIGURE 4.8 – Interface du ticket électronique avec QR Code

La figure 4.8 présente le ticket électronique généré après validation du paiement. Le ticket contient les informations principales du trajet telles que le numéro de réservation, les informations du voyageur, le siège attribué, les horaires du trajet ainsi qu'un QR Code unique utilisé lors des opérations d'embarquement.

Le voyageur peut télécharger le ticket au format PDF ou le conserver directement dans son espace personnel.

4.8.6 Module OCR – Reconnaissance de plaques

Parallèlement aux opérations de réservation, le système intègre un module OCR intelligent destiné à automatiser la reconnaissance des plaques d'immatriculation et l'attribution des quais.

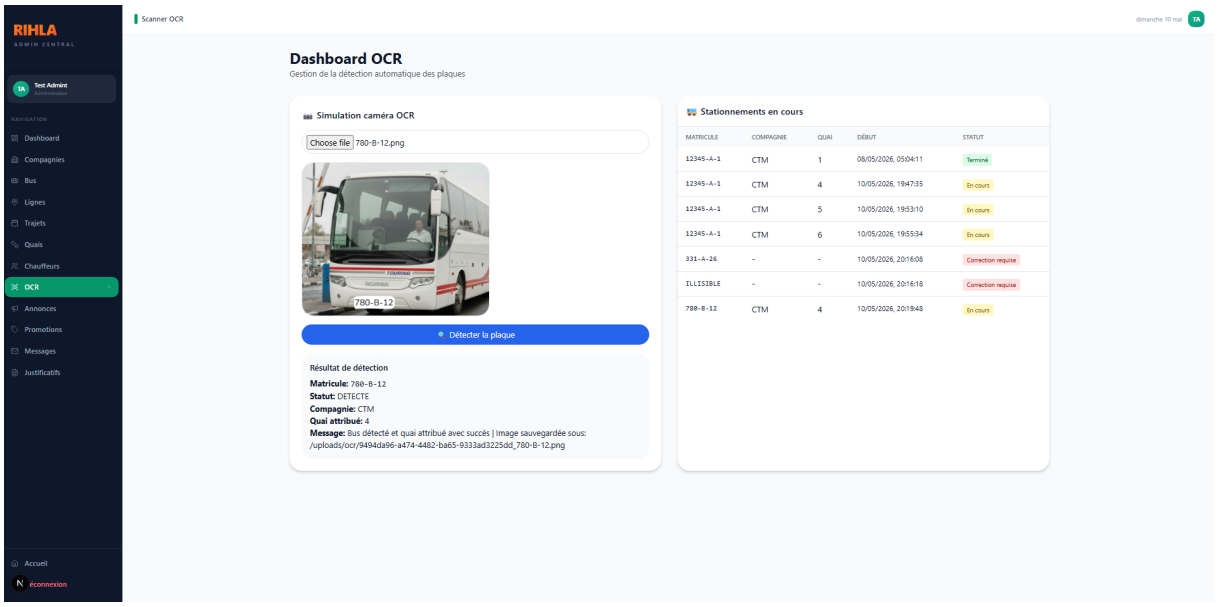


FIGURE 4.9 – Module OCR – Détection automatique de plaque et attribution de quai

La figure 4.9 montre l'interface du module OCR accessible à l'administrateur. Une image contenant la plaque d'immatriculation est soumise au système puis analysée par le modèle YOLOv8 après prétraitement avec OpenCV.

Le numéro détecté est affiché automatiquement avec les informations du bus et le quai attribué. En cas d'erreur de reconnaissance, l'administrateur peut corriger manuellement les données détectées.

4.8.7 Interface Chauffeur – Validation d'embarquement

Le jour du départ, le chauffeur utilise une interface dédiée afin de contrôler les tickets électroniques des voyageurs et valider les embarquements.

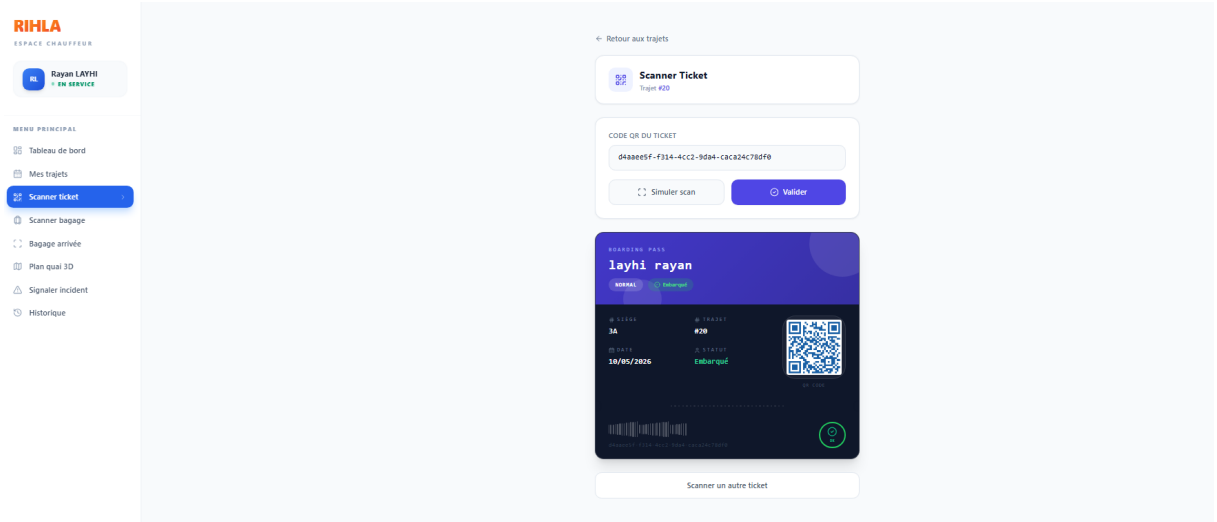


FIGURE 4.10 – Interface chauffeur – Validation des QR Codes

La figure 4.10 illustre l'interface utilisée par le chauffeur pour effectuer le contrôle des voyageurs avant le départ du bus. Le chauffeur peut scanner le QR Code présent sur chaque ticket électronique afin de vérifier son authenticité.

Une fois le ticket validé, le système confirme automatiquement l'embarquement du voyageur et met à jour les informations de réservation en temps réel.

4.8.8 Dashboard Responsable Compagnie

Le responsable compagnie dispose également d'un tableau de bord spécifique permettant de gérer les opérations liées à sa compagnie de transport.

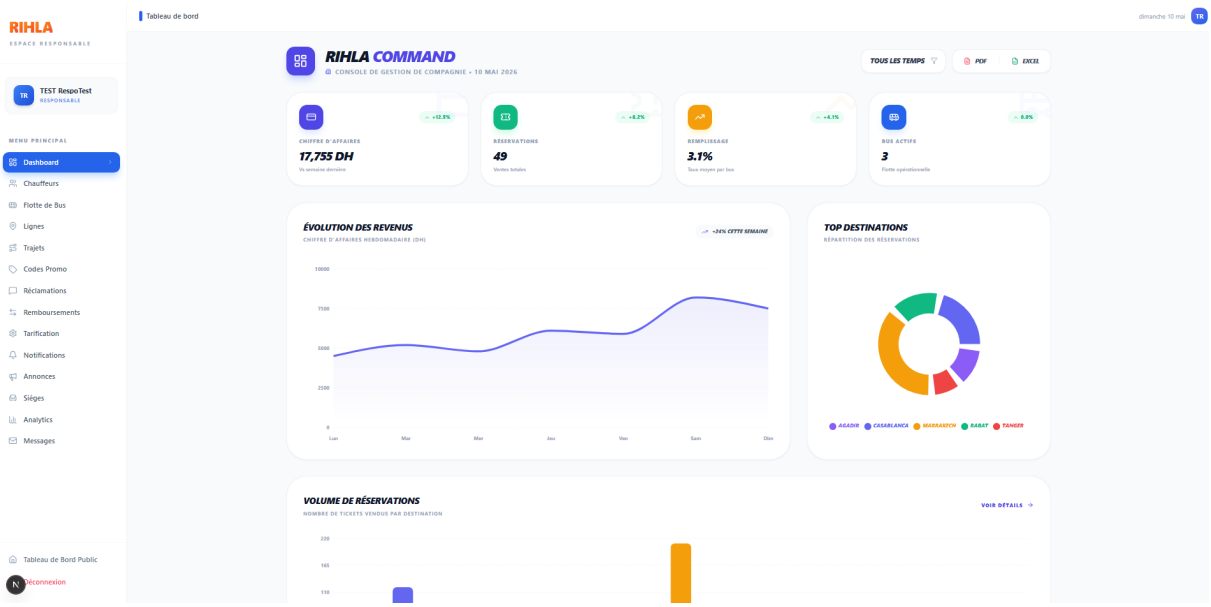


FIGURE 4.11 – Dashboard Responsable Compagnie – Gestion opérationnelle

La figure 4.11 présente l'interface du responsable compagnie. Ce tableau de bord centralise la gestion des bus, des chauffeurs, des trajets ainsi que les outils de configuration de la tarification dynamique et des codes promotionnels.

Des graphiques statistiques permettent également de suivre les revenus générés, le taux de remplissage des trajets et les performances globales de la compagnie.

4.8.9 Notifications temps réel

Afin d'assurer une meilleure communication entre les différents acteurs du système, la plateforme intègre un module de notifications temps réel basé sur WebSocket et STOMP.

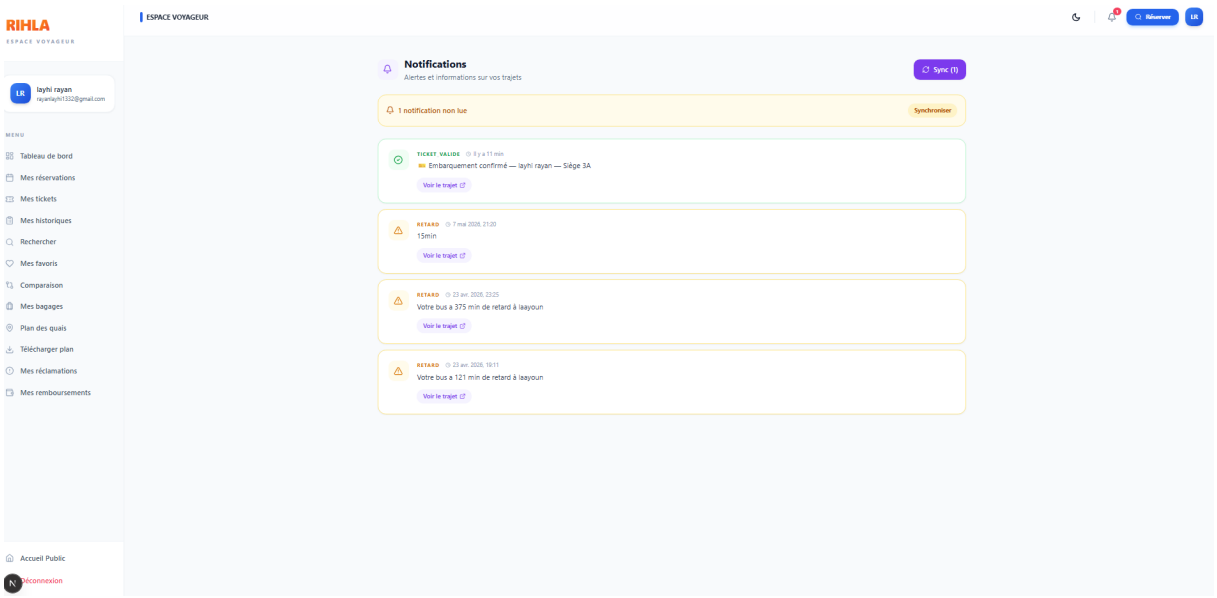


FIGURE 4.12 – Interface des notifications temps réel

La figure 4.12 illustre l'interface des notifications temps réel intégrée dans la plateforme. Les utilisateurs reçoivent instantanément les événements importants du système tels que les confirmations de réservation, les changements de quai, les rappels de départ, les notifications OCR ou les alertes administratives.

Cette interface améliore considérablement la réactivité du système ainsi que l'expérience utilisateur globale de la plateforme.

Conclusion

Dans ce chapitre, nous avons présenté la réalisation complète du projet à travers les sept sprints de développement. Nous avons décrit les fonctionnalités développées dans chaque sprint, les interfaces utilisateurs réalisées, les tests unitaires avec JUnit 5 et Mockito, les tests d'automatisation UI avec Selenium, l'analyse qualité avec SonarQube et SonarCloud, ainsi que les optimisations mises en place pour garantir la performance et la robustesse du système.

Conclusion Générale

Le projet « Gestion Intelligente de la Gare Routière » a permis de concevoir et de développer une plateforme moderne répondant aux besoins réels de digitalisation des opérations d'une gare routière. La solution développée repose sur une architecture distribuée robuste combinant Spring Boot, Next.js, MySQL et Docker, et intègre l'ensemble des concepts fondamentaux du module Systèmes Distribués.

Sur le plan des systèmes distribués, le projet couvre plusieurs paradigmes complémentaires. La communication synchrone est assurée par l'API REST sécurisée par JWT, qui constitue le canal principal d'échange entre le frontend et le backend, et par l'invocation de méthodes distantes via RMI pour le service de tarification. La communication asynchrone repose sur Apache ActiveMQ et JMS, permettant un découplage total entre les services producteurs de notifications et les services consommateurs, améliorant ainsi la réactivité et la scalabilité du système. Les mécanismes de résilience implémentés avec Resilience4j (Retry, Circuit Breaker et Fallback) garantissent la continuité du service même en cas de défaillance partielle. Enfin, la pagination REST optimise les performances lors de la manipulation de grandes quantités de données.

Sur le plan fonctionnel, la plateforme propose une gestion complète du cycle de vie des réservations, un module OCR innovant pour la reconnaissance automatique des plaques, des notifications temps réel via WebSocket, et une tarification dynamique adaptée aux besoins des compagnies de transport. La qualité du code a été assurée grâce à une stratégie de tests multicouche combinant les tests unitaires JUnit 5 avec Mockito, les tests d'automatisation UI avec Selenium, et l'analyse continue de la qualité et de la sécurité via SonarQube et SonarCloud.

Ce projet nous a permis d'approfondir de nombreuses compétences liées au développement web moderne, à l'architecture logicielle distribuée, à la sécurité applicative et aux tests logiciels. Il constitue une base solide sur laquelle plusieurs évolutions peuvent être envisagées, notamment l'intégration de paiements réels, le déploiement en microservices indépendants, l'ajout de mécanismes de machine learning pour l'optimisation des prix et la prédiction des flux de voyageurs, et l'implémentation de WebSocket pour le suivi GPS temps réel des bus.

Bibliographie et Webographie

Bibliographie

[1] VMware, Inc., *Spring Boot Reference Documentation – Version 3.4.x*, [En ligne], 2024. Disponible sur : https://spring.io/projects/spring-boot [Consulté le 15 avril 2025].

[2] VMware, Inc., *Spring Security Reference Documentation*, [En ligne], 2024. Disponible sur : https://spring.io/projects/spring-security [Consulté le 15 avril 2025].

[3] Vercel Inc., *Next.js Documentation – Getting Started*, [En ligne], 2024. Disponible sur : https://nextjs.org/docs [Consulté le 10 avril 2025].

[4] Apache Software Foundation, *Apache ActiveMQ Classic Documentation*, [En ligne], 2024. Disponible sur : https://activemq.apache.org/ [Consulté le 20 avril 2025].

[5] R. Winkler, *Resilience4j Documentation*, [En ligne], 2024. Disponible sur : https://resilience4j.readme.io/ [Consulté le 20 avril 2025].

[6] Oracle Corporation, *Java Remote Method Invocation Tutorial*, [En ligne], 2024. Disponible sur : https://docs.oracle.com/javase/tutorial/rmi/ [Consulté le 18 avril 2025].

[7] Oracle Corporation, *MySQL 8.0 Reference Manual*, [En ligne], 2024. Disponible sur : https://dev.mysql.com/doc/ [Consulté le 10 avril 2025].

[8] Docker Inc., *Docker Documentation*, [En ligne], 2024. Disponible sur : https://docs.docker.com/ [Consulté le 10 avril 2025].

[9] JUnit Team, *JUnit 5 User Guide*, [En ligne], 2024. Disponible sur : https://junit.org/junit5/docs/current/user-guide/ [Consulté le 5 mai 2025].

[10] SeleniumHQ, *Selenium WebDriver Documentation*, [En ligne], 2024. Disponible sur : https://www.selenium.dev/documentation/ [Consulté le 5 mai 2025].

[11] SonarSource, *SonarQube Documentation*, [En ligne], 2024. Disponible sur : https://docs.sonarsource.com/sonarqube/ [Consulté le 8 mai 2025].

[12] SonarSource, *SonarCloud – Continuous Code Quality*, [En ligne], 2024. Disponible sur : https://sonarcloud.io/ [Consulté le 8 mai 2025].

[13] SmartBear Software, *Swagger / OpenAPI Documentation*, [En ligne], 2024. Disponible sur : https://swagger.io/docs/ [Consulté le 12 avril 2025].

[14] A. S. Tanenbaum and M. Van Steen, *Distributed Systems : Principles and Paradigms*, 3rd ed. CreateSpace Independent Publishing Platform, 2017.

[15] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, Internet Engineering Task Force (IETF), May 2015. [En ligne]. Disponible sur : https://datatracker.ietf.org/doc/html/rfc7519.

[16] CTM Maroc, *Application mobile CTM – Réservation de billets*, [En ligne], 2024. Disponible sur : https://www.ctm.ma/ [Consulté le 2 mai 2025].

[17] Markoub.ma, *Réservation de billets d'autocar au Maroc*, [En ligne], 2024. Disponible sur : https://markoub.ma/ [Consulté le 2 mai 2025].

[18] Digiparc, *Logiciel de gestion de flotte et transport*, [En ligne], 2024. Disponible sur : https://digiparc.com/ [Consulté le 3 mai 2025].

[19] Busbud Inc., *Busbud – Book Bus Tickets Worldwide*, [En ligne], 2024. Disponible sur : https://www.busbud.com/ [Consulté le 3 mai 2025].

[20] Le Brief, *Gares routières : vers une modernisation numérique et efficace*, [En ligne], février 2025. Disponible sur : https://www.lebrief.ma/gares-routieres-vers-une-modernisation-numerique-et-efficace-100066488/ [Consulté le 4 mai 2025].

Annexes

Annexe A – Configuration Docker MySQL

```yaml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: gare-routiere-mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: gare_routiere
      MYSQL_USER: gare_user
      MYSQL_PASSWORD: gare_password
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    restart: unless-stopped

volumes:
  mysql_data:
```

Listing 4.4 – Docker Compose – Configuration MySQL

Annexe B – Configuration ActiveMQ

```java
@Configuration
@EnableJms
public class ActiveMQConfig {

    @Bean
    public ActiveMQConnectionFactory connectionFactory() {
        ActiveMQConnectionFactory factory =
```

```
            new ActiveMQConnectionFactory();
        factory.setBrokerURL("tcp://localhost:61616");
        factory.setUserName("admin");
        factory.setPassword("admin");
        return factory;
    }

    @Bean
    public JmsListenerContainerFactory<?>
       jmsListenerContainerFactory(
            ActiveMQConnectionFactory connectionFactory) {
        DefaultJmsListenerContainerFactory factory =
            new DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setConcurrency("3-10");
        return factory;
    }
}
```

Listing 4.5 – Configuration ActiveMQ – Spring Boot

Annexe C – Exemple NotificationProducer

```
@Service
@RequiredArgsConstructor
public class NotificationProducer {

    private final JmsTemplate jmsTemplate;
    private static final String QUEUE =
        "notification.voyageur";

    public void sendNotification(NotificationDTO dto) {
        jmsTemplate.convertAndSend(QUEUE, dto);
        log.info("Notification envoyee: {}", dto.getMessage());
    }
}

@Component
public class NotificationConsumer {

    @JmsListener(destination = "notification.voyageur")
```

```
    public void receiveNotification(NotificationDTO dto) {
        log.info("Notification recue: {}", dto.getMessage());
        // Envoi push ou email au voyageur
    }
}
```

Listing 4.6 – NotificationProducer et Consumer – ActiveMQ JMS

Annexe D – Exemple Resilience4j

```
@Service
public class EmailService {

    @Retry(name = "emailService",
           fallbackMethod = "emailFallback")
    @CircuitBreaker(name = "emailService",
                    fallbackMethod = "emailFallback")
    public void sendEmail(String to, String subject,
                          String body) {
        // Logique d'envoi d'email
        // Si echec, retry automatique (max 3 tentatives)
    }

    public void emailFallback(String to, String subject,
                              String body, Exception ex) {
        log.error("Impossible d'envoyer l'email a: {}", to);
        // Sauvegarder en base pour retry ulterieur
    }
}
```

Listing 4.7 – Configuration Resilience4j – EmailService