Thesis submitted to the

UNIVERSITY OF MOHAMED BOUDIAF – MSILA



FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

In partial fulfillment of the requirements for the degree of

Master in **Computer science**

By

Soheyb, HOUICHI

Wahid Wail Abdessamed, YATOUDJI

Title of the thesis

Designing and Architecting a Cloud-Based Multi-Tenant Field Service Management Solution

---

Under the supervision of

Ahlam BOUZAROURA

And the co-supervisor

Moukhtar ZAGHBA

Composition of the jury

K. Derdour	University of Msila	President
R. Mahdjoubi	University of Msila	Examiner

July, 2026

Abstract

Field Service Management (FSM) is essential for organizations that deploy technicians to customer sites, yet many still rely on manual or disconnected tools that cause weak traceability, limited visibility, and fragmented data. In parallel, modern applications increasingly need to serve several organizations from a single shared platform. This thesis addresses both issues by designing and implementing FieldOps, a cloud-hosted, multi-tenant FSM platform that centralizes the service-request lifecycle and provides distinct, role-based interfaces for administrators, managers, employees, technicians, and customers. After reviewing FSM concepts and multi-tenant cloud architectures, the system was analyzed and designed using UML, and a three-tier architecture was adopted. A shared-database, shared-schema tenancy model with a "TenantId" discriminator, secured through JWT-based access control, ensures logical data isolation between tenants. The platform was implemented as an ASP.NET Core Web API backend (SQL Server, Entity Framework Core), a React/TypeScript web application, and a Flutter/Dart mobile application, and was deployed online. The result is an accessible, scalable, and low-cost solution suited to small and medium-sized organizations, particularly in the Algerian context.

Keywords: Field Service Management; Multi-tenancy; Cloud computing; SaaS; Data isolation; shared-database shared-schema.

الملخص

تُعد إدراة الخدمات الميدانية (Field Service Management) نشاطًا أساسيًا للمؤسسات التي توفد تقنيها إلى مواقع العملاء لإنجاز أعمال الصيانة والتركيب والإصلاح والمعاينة، غير أنّ العديد منها لا يزال يعتمد على أساليب يدوية أو أدوات غير مترابطة، ممّا يؤدّي إلى ضعف تتبّع العمليات، ومحدودية الرؤية الآنية، وتشتّت البيانات. وفي الوقت نفسه، أصبحت التطبيقات الحديثة مطالَبة بخدمة عدّة مؤسسات انطلاقًا من منصّة واحدة مشتركة. تعالج هذه الأطروحة هذين الإشكالين من خلال تصميم وإنجاز منصّة «FieldOps»، وهي منصّة لإدارة الخدمات الميدانية متعدّدة المستأجرين (Multi-tenant) ومستضافة سحابيًا، تُوحّد دورة حياة طلب الخدمة وتوفّر واجهات متمايزة قائمة على الأدوار للمسؤولين والمدراء والموظفين والتقنيين والعملاء. وبعد دراسة مفاهيم إدارة الخدمات الميدانية والبنى السحابية متعدّدة المستأجرين، جرى تحليل النظام وتصميمه باستخدام لغة النمذجة الموحّدة

(UML)، واعتُمدت بنية ثلاثية الطبقات. وقد اعتُمد نموذج «قاعدة بيانات مشتركة بمخطّط مشترك» مع مُميِّز TenantId، مؤمَّنًا عبر المصادقة المعتمدة على الرموز (JWT) لضمان العزل المنطقي لبيانات كلّ مستأجر. وأُنجِزت المنصّة في صورة خدمة خلفية بواجهة برمجية ASP.NET Core Web API (مع SQL Server و Entity Framework Core)، وتطبيق ويب بلغة React/TypeScript، وتطبيق محمول بإطار Flutter/Dart، ثمّ نُشِرت عبر الإنترنت. والنتيجة حلٌّ متاحٌ وقابلٌ للتوسّع ومنخفض التكلفة يناسب المؤسسات الصغيرة والمتوسطة، ولا سيّما في السياق الجزائري.

الكلمات المفتاحية: إدارة الخدمات الميدانية، تعدّد المستأجرين، الحوسبة السحابي،البرمجيات كخدمة، عزل البيانات، قاعدة بيانات مشتركة مع مخطط ثابت.

Dedications

قال الله تعالى ﴿ وَقُل رَّبِّ زِدْنِي عِلْمًا ﴾ [طه: 114]

الحمد لله الذي علّم بالقلم، علّم الإنسان ما لم يعلم،
وشرّف أهل العلم ورفعهم درجات، وجعل طلب العلم عبادةً يُتقرب بها إليه،
والصلاة والسلام على من بُعث معلّمًا وهادياً ورحمةً للعالمين

وقال تعالى ﴿ وَقَضَىٰ رَبُّكَ أَلَّا تَعْبُدُوا إِلَّا إِيَّاهُ وَبِالْوَالِدَيْنِ إِحْسَانًا﴾ [الاسراء: 23]

إلى والديّ الكريمين، اللذين كانا سببًا بعد توفيق الله في تربيتي على القيم والإيمان، وغرسا في نفسي حب العلم والاجتهاد، وكان دعاؤهما لي سندًا لا ينقطع، أُهدي ثمرة هذا الجهد المتواضع عرفانًا بفضلهما وإحسانهما

وإلى اخي وشريكي في هذه المذكرة صهيب، الذي كان نعم العون والسند في إنجاز هذا العمل، فله مني خالص الشكر والتقدير

Wahid Wail Abdessamed, YATOUDJI

قال الله تعالى ﴿ رَبِّ أَوْزِعْنِي أَنْ أَشْكُرَ نِعْمَتَكَ الَّتِي أَنْعَمْتَ عَلَيَّ وَعَلَىٰ وَالِدَيَّ وَأَنْ أَعْمَلَ صَالِحًا تَرْضَاهُ ﴾ [النمل: 19]

الحمد لله الذي بنعمته تتمّ الصالحات، أحمده حمداً كثيراً طيباً مباركاً فيه، حمداً يليق بجلاله وعظيم سلطانه، وأشكره على ما أنعم به عليّ وأتمّ من فضله.

إلى والديّ الكريمين، أمي الغالية التي لم ينقطع دعاؤها لي لحظةً واحدة، وكان حبّها الدافئ يُضيء لي الطريق في أصعب اللحظات، وأبي العزيز قدوتي ومثلي الأعلى، الذي علّمني الصبر والمثابرة بسيرته قبل كلامه، وحمل الكثير في صمت لأنعم أنا بالدراسة والراحة، ما وصلتُ إلا بتضحياتكما ودعائكما، فهذا الإنجاز لكما قبل أن يكون لي.

إلى إخوتي الأعزاء **لؤي**، **محمد**، **يوسف**، الذين كانوا سنداً لي في كل مرحلة، يُشاركونني أفراحي ويُخففون عني أحزاني، أتمنى لكم من الله كل توفيق ونجاح، وأن يبلّغكم أعلى ما تتمنّون.

إلى أخي وشريكي في هذه المذكرة **وحيد**، نِعمَ العون كنتَ ونِعمَ السند، فلك مني خالص الشكر والتقدير على كل لحظة تعاون وكل جهد مشترك.

Soheyb, HOUICHI

Acknowledgments

We are deeply indebted to our thesis supervisors, Mrs. **Ahlam BOUZAROURA** and Mr. **Moukhtar ZAGHBA**, whose unwavering support, guidance, and inspiration have been invaluable throughout this project. Their expertise, encouragement, and dedication have greatly contributed to the successful completion of this work. We sincerely thank them for their continuous assistance and commitment to helping us overcome the challenges encountered during this study.

We would also like to express our sincere gratitude to the members of the thesis jury for their time, effort, and valuable evaluation of our work. Their insightful comments, constructive feedback, and academic expertise are greatly appreciated and have contributed to improving the quality of this research.

Special thanks go to our friends and families for their patience, understanding, and constant encouragement. Their support helped us endure the demanding moments and challenges we faced throughout the course of this research project.

Contents

List of figures v
Introduction 1
Chapter 1: Field Service Management 3
1.1 Introduction 3
1.2 Definition of FSM 3
1.3 Main Actors in FSM 4
1.4 Core Processes in FSM 5
1.5 Importance of FMS System 6
1.6 Challenges of field operations 6
1.6.1 Task Coordination 7
1.6.2 Lack of Real-Time Monitoring 7
1.6.3 Communication Issues 7
1.6.4 Incident Management 7
1.6.5 Data Fragmentation 7
1.7 Existing FSM solutions 7
1.8 Need for an Improved FSM System 8
1.9 Conclusion 9
Chapter 2: Multi-Tenant Architecture & Cloud Systems 10
2.1 Introduction 10
2.2 Definition of Multi-Tenancy 10
2.3 Limitations of Traditional Architectures 11
2.4 Multi-Tenant Models 11
2.4.1 Shared Database with Shared Schemas 11
2.4.2 Shared Database with Separate Schemas 12
2.4.3 Separate Database per Tenant 12
2.4.4 Hybrid Model 13
2.5 Comparison of Multi-Tenant Models 14
2.6 Fundamental Requirements of Multi-Tenant Systems 15
2.6.1 Data Isolation 15
2.6.2 Scalability 15
2.6.3 Security 16
2.6.4 Maintainability 16
2.6.5 Customization 16
2.7 Benefits and Challenges of Multi-Tenant Architecture 16
2.8 Cloud Computing Overview 17
2.8.1 Cloud Service Models and Architectural Aspects 18

2.9 Relationship Between Cloud Computing and Multi-Tenancy 18
2.10 Conclusion 19
Chapter 03: Conceptual Design of a Field Service Management System 20
3.1 Introduction 20
3.2 System Overview and Objectives 20
3.3 Requirements Analysis 21
3.3.1 System Actors 21
3.3.2 Functional Requirements 22
3.3.3 Non-Functional Requirements 24
3.3.4 Use Case Diagram 25
3.4 System Design 30
3.4.1 System Architecture 30
3.4.2 Selected Multi-Tenant Model 31
3.4.3 Database Design 32
3.4.4 Class Diagram 36
3.4.5 Sequence Diagrams 38
3.4.6 State Diagram 42
3.5 Conclusion 44
Chapter 04: System Implementation 45
4.1 Introduction 45
4.2 Development Environment 45
4.2.1 Development Stack 46
4.2.2 Deployment Services 48
4.3 Presentation of the Application 48
4.3.1 Backend Presentation 48
4.3.2 Web Platform Presentation 54
4.3.3 Mobile Application Presentation 69
4.4 Conclusion 78
Conclusion 80
References 82

List of figures

Figure1.1: Key Features of Dynamic 365 Field Service Management 4
Figure2.1: Shared Database with Shared Schemas...……...11
Figure2.2: Shared Database with Separate Schemas 12
Figure2.3: Separate Database per Tenant 13
Figure2.4: Hybrid Model 13
Figure 3.1. System Administrator Use Case Diagram……25
Figure 3.2 Actor Generalization Hierarchy 26
Figure 3.3. Tenant Admin Use Case Diagram 27
Figure 3 4. Manager Use Case Diagram 27
Figure 3.5. Employee Use Case Diagram 28
Figure 3.6. Field Ops Mobile Application Use Case Diagram 29
Figure 3.7. System Architecture Diagram (Three-Tier Client-Server Architecture)..... 31
Figure 3.8. Entity relations Diagram (ERD) of the Proposed System 35
Figure 3.9. Class Diagram of proposed system 38
Figure 3.10. Multi-Tenant Authentication Sequence Diagram 40
Figure 3.11. HTTP Request Lifecycle Sequence Diagram 41
Figure 3.12. Service Request Status State Diagram 43
Figure 4.1. REST API Endpoints for Users …….. 49
Figure 4.2. REST API Endpoints for service request 50
Figure 4.3.GET Employee by Dynamic Search using postman 51
Figure 4.4. Service Request Entity 52
Figure 4.5. The methods used to change the status of a service request 53
Figure 4.6. Status Transition class 54
Figure 4.7. Web Platform Authentication Interface 55
Figure 4.8. System Administrator sidebar 56
Figure 4.9. System Administrator Dashboard Interface 57
Figure 4.10. Tenant Management interface 58
Figure 4.11. Create New Tenant Form 59
Figure 4.12. Tenant Administrator Sidebar 60
Figure 4.13. Tenant Administrator Dashboard 61
Figure 4.14. Manager Sidebar 61

Figure 4.15. Service Requests Interface 62
Figure 4.16. Service Request Items 62
Figure 4.17. First Taps of Creating New Request 63
Figure 4.18. Last Tab of creating new request 63
Figure 4.19. New Service Request 64
Figure 4.20. Completed Service request 64
Figure 4.21. Rejection Window 65
Figure 4.22. Re-assignment Window 65
Figure 4.23. Scheduling Window 65
Figure 4.24. Assignment service request to employee window 66
Figure 4.25. Users Management interface (Employees) 66
Figure 4 26. Create New Employee Account Form 67
Figure 4.27. Employee Profile interface 67
Figure 4.28. Employee Sidebar 68
Figure 4.29. Employee service request interface 69
Figure 4.30. Self-Assignment Window 69
Figure 4.31. Authentication interface of the mobile application 70
Figure 4.32. Customer Registration Request Form 70
Figure 4.33. Customer Dashboard interface 71
Figure 4.34. Form for creating a service request (customer) 72
Figure 4.35. List of the customer's service requests 72
Figure 4.37. List of the customer's assets 73
Figure 4.36. Form for adding an asset 73
Figure 4.38. Customer Profile Interface 74
Figure 4.39. Technician Dashboard Interface 74
Figure 4.40. List of the technician's service requests 75
Figure 4.41. Detailed view of a service request (technician) 75
Figure 4.42. Form for adding an attachment 76
Figure 4.43. List of attachments 76
Figure 4.44. Technician Profile Interface 77
Figure 4.45. Tenant-information Interface 78
Table 2.1. Comparison of Multi-Tenancy Models……15

Introduction

The continuous evolution of information and communication technologies has significantly transformed the way organizations manage their operational activities. In sectors such as construction, telecommunications, utilities, maintenance, and technical support, a large portion of work is performed in the field by geographically distributed teams of technicians and contractors. In such environments, ensuring efficient coordination, real-time information exchange, and continuous monitoring of field operations has become essential for maintaining service quality, operational efficiency, and timely task execution.

Despite the availability of modern digital solutions, many organizations still rely on traditional methods such as manual reporting, phone-based communication, spreadsheets, or disconnected applications. These practices often lead to several limitations, including weak traceability of operations, lack of real-time visibility, inefficient communication between stakeholders and difficulties in tracking task progress. As a result, organizations may face delayed responses to incidents, data inconsistency, duplication of information, and reduced control over field activities, which ultimately affects productivity and service quality.

Although Field Service Management systems have been introduced to address these issues, modern requirements extend beyond improving internal operations. With the growing adoption of cloud-based systems, organizations increasingly require scalable platforms capable of supporting multiple entities within a shared environment.

This thesis addresses two main problems. The first problem is **the poor management of field service operations** because many tasks are still handled using manual or disconnected methods. The second problem is **how to build the system as a multi-tenant platform** so that many companies can use the same application safely while keeping their data separate.

To address these challenges, this thesis proposes the design and implementation of a **Multi-Tenant Field Service Management System** that provides a unified platform for managing field operations. The proposed system supports the main stages of field service management, including service request handling, task assignment, task status monitoring, reporting, and performance tracking. In addition, it integrates a tenant-aware architecture that allows multiple organizations to share the same platform while maintaining isolated data and secure access boundaries.

This thesis is organized as follows:

- **Chapter 1** presents the concept of Field Service Management, its importance, challenges, and existing solutions.
- **Chapter 2** presents the main concepts of multi-tenant architecture, covering its fundamental concepts, architectural approaches, key requirements, and its role in designing scalable, secure, and cloud-based systems.
- **Chapter 3** focuses on the analysis and design of the proposed system, including functional requirements, UML diagrams, system architecture, and database schema.
- **Chapter 4** describes the implementation of the application, and presents the application obtained results.

1 Chapter 1: Field Service Management

1.1 Introduction

Field Service Management (FSM) has become an important discipline in modern organizations whose activities depend on operations performed outside the company's main premises. It refers to the planning, coordination, and control of tasks carried out by field personnel at customer sites or other external locations, such as equipment installation, maintenance interventions, repair operations, and technical support services.

In modern organizational contexts, where organizations require quick responses, accurate scheduling, and effective coordination, FSM systems help move from manual, paper-based workflows toward integrated digital platforms. These platforms support the management of field operations by improving task organization, intervention follow-up, information flow, and communication between office staff and field workers, thereby enhancing operational efficiency and service quality.

1.2 Definition of FSM

Field Service Management (FSM) can be defined as the planning, coordination, and control of all activities performed by field personnel at customer sites or in the field. This includes managing technicians, scheduling service visits, tracking vehicles and mobile workers, and ensuring that the right tools and spare parts are available at the right time and place. [1]

From an operational point of view, FSM focuses on ensuring that the right technician, equipped with the right tools and information, is assigned to the right task at the right time and location.[2] This makes FSM a key element in improving responsiveness, resource utilization, and service quality.

From a technical perspective, FSM represents a complete management approach that covers the entire lifecycle of a field intervention, starting from the reception of a service request and ending with task completion, reporting, and work order closure. It also involves the integration of field activities with organizational information systems in order to ensure better coordination between customers, field teams, assets, and administrative services. [1],[3]

A typical FSM system supports several core functions: [1], [2]

- **Task creation and assignment**, through the generation of work orders and their allocation to technicians.
- **Workforce scheduling and dispatching**, by assigning the most suitable technician to each job according to availability, location, or specialization.
- **Task status tracking and follow-up**, in order to monitor the progress of field activities.
- **Reporting and feedback collection**, including intervention reports and service-related observations.
- **Resource and asset management**, such as tracking equipment, spare parts, and tools required for field operations.

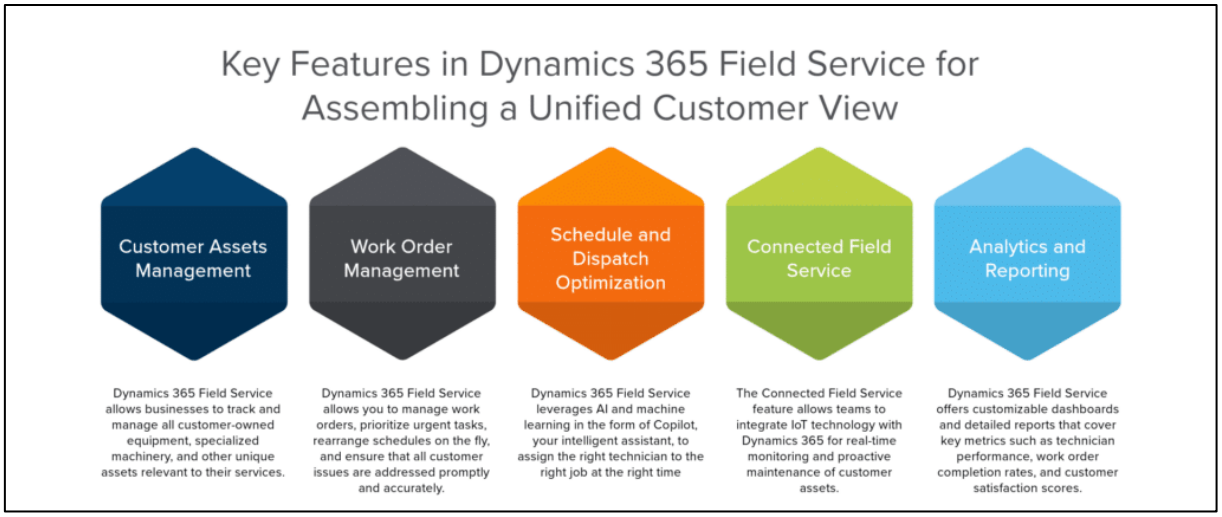


***Figure1.1**: Key Features of Dynamic 365 Field Service Management. [4]*

1.3 Main Actors in FSM

Field Service Management involves several actors who contribute to the planning, execution, and monitoring of field operations. These actors interact throughout the service process, from the identification of a service need to the completion of the intervention and the evaluation of its results.[5], [6]

The main actors involved in FSM are:

- **Service provider organization**: The organization responsible for delivering field services and managing the overall workflow.
- **Field worker or technician**: The person who performs tasks in the field, such as installation, maintenance, repair, or inspection.

- **Supervisor or dispatcher**: The actor responsible for assigning tasks, scheduling interventions, and monitoring field activities.
- **Customer or service recipient:** The person or organization that requests the service or receives the intervention.
- **Manager or decision-maker**: The actor who monitors overall performance, analyzes reports, and supports operational or strategic decisions.

These actors work together to ensure that field operations are properly planned, executed, and monitored. [6]

1.4 Core Processes in FSM

Field Service Management is based on a set of core processes that organize the lifecycle of field operations from the initial request to the completion of the service. These processes help ensure that field activities are planned, executed, and monitored in a structured and efficient way.[7]

The main core processes in FSM are the following: [7], [8]

- **Service request**: The process begins when a customer, user, or organization reports a problem or requests a service. This request usually contains basic information about the issue, the location, and the type of intervention needed.
- **Work order creation**: After the request is received and reviewed, it is transformed into a work order. The work order defines the task to be performed and includes important details such as the type of service, required resources, priority level, and deadline.
- **Task assignment**: Once the work order is created, it is assigned to the most appropriate field worker or technician. This assignment is generally based on factors such as availability, skills, location, and urgency of the task.
- **Task execution**: In this stage, the technician performs the required intervention in the field. The execution process may include installation, maintenance, repair, inspection, or any other service activity depending on the operational context.

- **Reporting and closure**: After completing the task, the field worker submits a report describing the work performed, the result of the intervention, and any

relevant observations. This step ensures proper documentation and allows the organization to evaluate performance and close the work order.

These core processes form the operational foundation of any FSM system. They ensure that field activities follow a logical sequence and that all interventions are recorded, monitored, and completed in an organized manner.

1.5 Importance of FMS System

FSM systems are important because they bridge the gap between office-based management and on-site execution. By giving managers, a clear view of what is happening in the field, they reduce uncertainty, improve planning, and support faster decision-making.[9], [1]

FSM is widely recognized as a key enabler for improving service delivery performance and reducing operational costs through automation and enhanced coordination. Research on e-management and service systems highlights that digital FSM platforms help organizations streamline workflows, minimize duplicated effort, and better align field operations with strategic objectives. [1],[10]

From a practical viewpoint, FSM systems offer several advantages: [9] ,[10]

- Improved productivity by matching the right technician with the right job according to skill, location, and availability.
- Enhanced visibility into field operations, allowing managers to monitor progress and intervene when needed.
- Faster response times to customer requests and incidents, leading to higher customer satisfaction.
- Better communication between managers and field workers through mobile apps, alerts, and shared information.
- Data-driven decision-making based on real-time performance metrics and analytics.

1.6 Challenges of field operations

Despite the benefits of Field Service Management, managing field operations remains a complex task, especially in large or geographically distributed organizations.

Several challenges continue to affect the efficiency and quality of field activities.[1],[11], [12]

1.6.1 Task Coordination

One of the main challenges is assigning the right technician to the right task at the right time. This becomes difficult when multiple factors must be considered, such as skills, location, priority, and service requirements. Poor coordination may lead to scheduling conflicts, wasted travel time, and reduced productivity.[1], [11]

1.6.2 Lack of Real-Time Monitoring

Traditional methods such as phone calls, paper forms, and delayed reports make it difficult to know the exact status of ongoing tasks. This limited visibility can lead to missed appointments, delayed interventions, and lower service quality. Modern FSM systems aim to improve this through mobile updates and real-time tracking.[11], [12]

1.6.3 Communication Issues

Ineffective communication between managers and field workers can lead to misunderstandings, delays, and execution errors. Clear and timely communication is essential for ensuring coordination and improving service performance.[1], [11]

1.6.4 Incident Management

Unexpected events such as equipment failures, safety issues, or urgent service requests require fast and organized responses. However, traditional systems often lack proper mechanisms for handling such incidents efficiently. Modern FSM platforms address this through alerting and escalation workflows.[11], [12]

1.6.5 Data Fragmentation

Relying on multiple disconnected tools such as spreadsheets, standalone mobile applications, or local databases often results in fragmented and inconsistent data. Consolidating field data within a unified platform ensures that customer history, asset records, and service outcomes are consistently accessible where needed, a principle widely emphasized in research on e-management and service delivery systems. [11]

1.7 Existing FSM solutions

Several Field Service Management solutions, both commercial and open-source, have been developed to help organizations manage field operations more efficiently.

These systems aim to digitalize activities performed outside traditional office environments and improve coordination, visibility, and operational control.[5], [11]

In general, existing FSM platforms provide core functionalities such as task scheduling and dispatching, mobile access for field workers, task status updates, reporting, and location-based tracking.[6], [8] Some solutions also include support for offline operation, document attachment, and performance monitoring, which help organizations improve service execution and resource utilization.[9]

Despite these advantages, existing FSM solutions still present several limitations. Many of them are costly to implement and maintain, especially for small and medium-sized organizations. Others offer limited flexibility and may be difficult to adapt to specific operational workflows or sector requirements.[9]

In addition, scalability and multi-tenant support remain important challenges. Some systems are not well suited for environments where multiple organizations need to share the same platform while maintaining strict data isolation and independent operation.

Here in Algeria, the availability of locally developed and widely adopted FSM solutions appears to be limited. Most existing platforms are developed for international markets and may not fully address the operational, financial, and organizational realities of Algerian companies. In addition, many commercial FSM solutions are relatively expensive in terms of licensing, deployment, and maintenance costs, which makes their adoption difficult for many small and medium-sized companies in Algeria.

This situation highlights the need for FSM solutions that are more accessible, adaptable, flexible, and scalable, while being better suited to the local context and supported by appropriate architectural designs.

1.8 Need for an Improved FSM System

Although the progress made by existing FSM solutions, several limitations still affect their suitability for different operational contexts. Many platforms remain costly, insufficiently flexible, or not well adapted to environments that require scalability and multi-tenant support.

In the Algerian context, this need is even more significant due to the limited availability of locally adapted FSM solutions and the high cost of many commercial platforms. Therefore, there is a clear need for a more accessible, adaptable, flexible, and scalable FSM system that fits local organizational requirements and relies on appropriate architectural design.

1.9 Conclusion

Field Service Management has emerged as a key approach for organizations that rely on field-based activities such as maintenance, installation, inspection, and repair, as it structures the entire lifecycle of service requests from initiation to closure and monitoring. By formalizing processes such as task scheduling, resource allocation, and performance tracking, FSM aims to improve operational efficiency, responsiveness, and service quality.

Despite these benefits, managing field operations remains challenging in practice. Organizations continue to face coordination difficulties when assigning the right technician to the right task, limited real-time visibility into ongoing interventions, recurrent communication issues between office staff and field workers, and fragmented data scattered across heterogeneous tools and formats. These factors can lead to delays, inefficient routing, duplicated effort, and suboptimal decision-making.

Several commercial and open-source FSM solutions have been developed to address these challenges by digitalizing field operations and supporting centralized planning, tracking, and reporting. However, many of these systems still exhibit notable limitations related to cost, customization, scalability, and adaptability to specific organizational or regional contexts. In particular, for small and medium-sized enterprises and for environments such as the Algerian market, the high licensing and deployment costs of international platforms, combined with their limited alignment with local operational realities, represent significant barriers to adoption.

These observations highlight the need for FSM solutions that are more accessible, flexible, and scalable, while offering better support for heterogeneous organizations and local constraints. They motivate the exploration of architectural approaches capable of hosting multiple organizations on a shared platform without compromising data isolation, security, or configurability. This rationale naturally leads to the study of multi-tenant architectures and cloud-based systems, which form the focus of the next chapter and provide the technological foundation for the proposed platform

2 Chapter 2: Multi-Tenant Architecture & Cloud Systems

2.1 Introduction

With the increasing demand for scalable and cloud-based software systems, modern applications are expected to support multiple clients or organizations within a single infrastructure. This requirement has led to the emergence of multi-tenancy architecture, a design approach that enables a single software instance to serve multiple tenants while ensuring logical separation of their data and configurations.

In the context of Field Service Management (FSM) systems, multi-tenancy is particularly relevant, as service providers often manage operations for multiple organizations simultaneously.

Therefore, adopting a multi-tenant architecture allows for improved scalability, reduced infrastructure costs, and simplified system maintenance.

This chapter presents the fundamental concepts of multi-tenancy, its main architectural approaches, advantages, challenges, and its relevance to the design of the proposed FSM system.

2.2 Definition of Multi-Tenancy

Multi-tenancy is a software architecture model in which a single instance of an application serves multiple independent users or organizations, known as **tenants**. Each tenant operates within a shared environment but has its own isolated data, configurations, and user management.[13]

A **tenant** can be defined as:

A group of users, typically belonging to the same organization, who share access to a common system while maintaining logical data separation from other groups.

In a multi-tenant system, all tenants share the same application codebase and infrastructure, but their data is segregated either logically or physically depending on the architectural approach. [14]

2.3 Limitations of Traditional Architectures

Traditional application architectures, such as monolithic systems, are typically designed to serve a single organization. While they can be simple to implement in small-scale environments, they often become difficult to scale and maintain when the number of users or organizations increases.

These limitations have encouraged the transition toward more flexible architectures, such as multi-tenancy, which are better suited for modern distributed systems. [15]

2.4 Multi-Tenant Models

Multi-tenant architecture can be implemented using different approaches, depending on the required level of data isolation, system complexity, and scalability.[13] Each approach represents a trade-off between performance, cost, and security. The main models are described below.[16]

2.4.1 Shared Database with Shared Schemas

In this model, all tenants share the same database and the same tables. Data belonging to different tenants is distinguished using a specific identifier, commonly referred to as a (**tenant_id)**, which is included in each record. This means that all application data is stored together, but logically separated through filtering mechanisms.[13], [16]



***Figure2.1**: Shared Database with Shared Schemas*

This model is highly efficient in terms of resource utilization, as it minimizes database overhead and simplifies deployment. It is particularly suitable for systems that need to support a large number of tenants with relatively similar requirements. However, it requires strict implementation of security rules to ensure that each tenant can only access its own data. Any failure in enforcing these rules may lead to data leakage between tenants.[17]

2.4.2 Shared Database with Separate Schemas

In this model, a single database is used, but each tenant is assigned a separate schema. A schema represents a logical grouping of tables, meaning that each tenant has its own set of tables within the same database instance.[13]

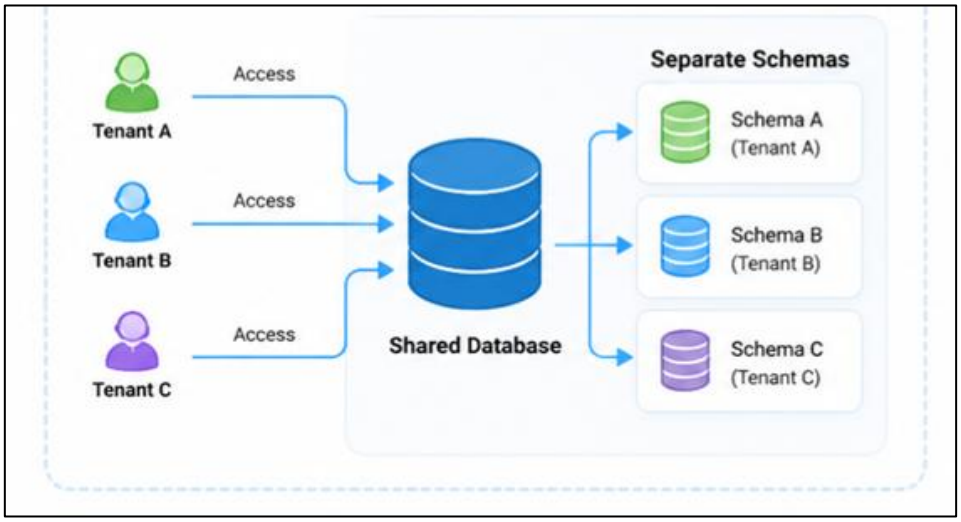


***Figure2.2**: Shared Database with Separate Schemas*

This model provides a higher level of data isolation compared to the shared tables approach, as tenant data is separated at the schema level rather than within the same tables. It also allows some level of customization for each tenant without affecting others. However, managing multiple schemas increases system complexity, especially when applying updates or changes, as these must be replicated across all schemas. [16]

2.4.3 Separate Database per Tenant

In this model, each tenant has its own dedicated database. This ensures complete physical separation of data, providing the highest level of security and isolation. [13] Each organization operates independently, and database-level operations such as backup, recovery, and scaling can be handled individually. [16]

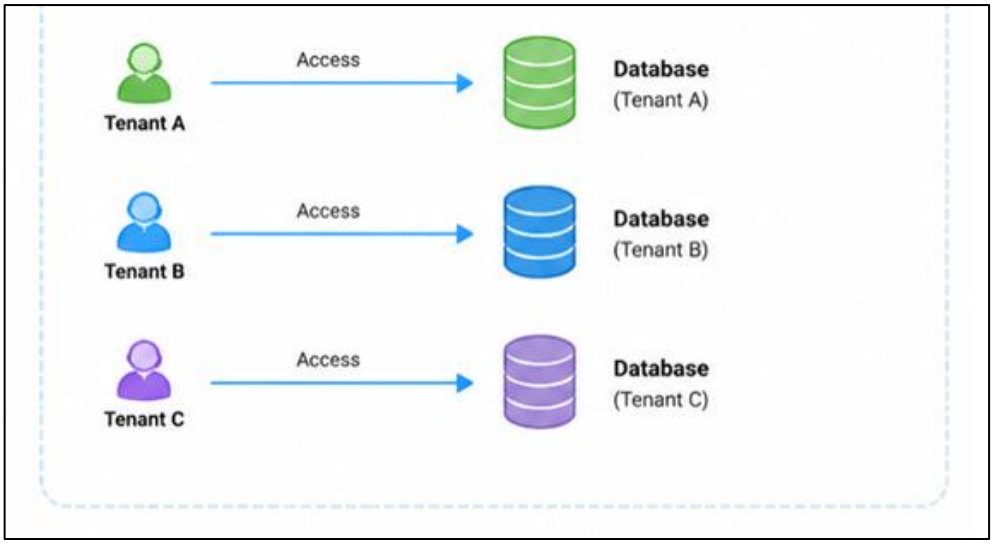


***Figure2.3**: Separate Database per Tenant*

While this model offers strong data protection and flexibility, it comes with higher infrastructure and maintenance costs. Managing a large number of databases can become complex and may reduce overall system scalability. For this reason, it is often used in systems where data security and compliance requirements are critical.[17]

2.4.4 Hybrid Model

In some cases, systems adopt a hybrid approach that combines elements of the previously mentioned models. For example, smaller tenants may share a database, while larger tenants are assigned dedicated databases.[18] This allows the system to balance cost efficiency and performance based on tenant needs.[16]

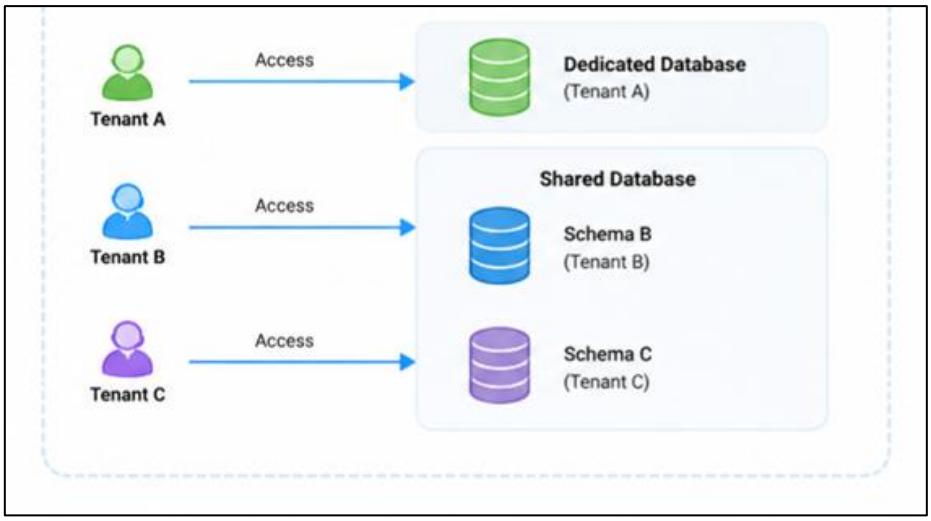


***Figure2.4.** Hybrid Model*

The hybrid model provides flexibility and can be adapted as the system evolves. However, it also introduces additional complexity in system design and management, as different tenants may follow different storage strategies.[13]

Overall, the choice of a multi-tenant approach depends on the specific requirements of the system, including scalability, security, cost, and maintainability. In the context of Field Service Management systems, selecting the appropriate approach is essential to ensure efficient operation while supporting multiple organizations within a unified platform.[3]

2.5 Comparison of Multi-Tenant Models

The choice of a multi-tenancy model depends on multiple factors, including data isolation requirements, scalability needs, operational cost, and system complexity. There is no universally optimal solution, as each approach represents a different balance between these constraints.[13]

The shared database with shared tables approach is the most resource-efficient, as all tenants use the same data structure. However, it requires strong application-level controls to ensure proper data separation. The shared database with separate schemas approach improves isolation by separating tenant data at the schema level, offering a balance between structure and flexibility, but it increases maintenance complexity. In contrast, the separate database per tenant approach provides the strongest isolation and customization capabilities, but at the cost of higher infrastructure usage and reduced scalability efficiency.[17]

In general, multi-tenancy design involves a trade-off between efficiency, isolation, and scalability. As system requirements grow, the architectural choice must carefully balance these factors.

Model	Data Isolation Level	Operational Cost	Scalability Behavior	Complexity
Shared Database with Shared Tables	Logical separation within shared structure	Low infrastructure demand	Supports large number of tenants efficiently	Simple structural design but requires strict application-level control
Shared Database with Separate Schemas	Schema-level separation per tenant	Moderate infrastructure and maintenance effort	Suitable for medium-scale growth scenarios	Increased management effort due to multiple schemas
Separate Database per Tenant	Physical and complete isolation	High infrastructure requirements per tenant	Limited efficiency when scaling to many tenants	High administrative and deployment complexity
Hybrid	Configurable — combines logical and physical isolation based on tenant tier or data sensitivity	Cost scales with chosen mix; premium tenants drive higher spend	Highly scalable — standard tenants share resources while high-value tenants get dedicated instances	Requires orchestration logic to route tenants across models; most complex to build and operate

Table 2.1 *Comparison of Multi-Tenancy Models. [13],[18]*

2.6 Fundamental Requirements of Multi-Tenant Systems

A well-designed multi-tenant system must satisfy a set of fundamental requirements to ensure efficiency, reliability, and secure operation. [13] These requirements are essential for maintaining system performance while supporting multiple tenants within a shared environment.[16]

2.6.1 Data Isolation

Data isolation is one of the most critical requirements in multi-tenant systems. [13] Each tenant must only be able to access its own data, without any possibility of interference or data leakage between tenants. This isolation can be achieved through logical mechanisms, such as tenant identifiers, or through structural separation in the database. Ensuring proper data isolation is essential for maintaining privacy, trust, and compliance with data protection standards.[16]

2.6.2 Scalability

Scalability refers to the ability of the system to handle an increasing number of tenants, users, and operations without significant degradation in performance.[19] A scalable multi-tenant system should be able to expand efficiently as demand grows,

whether by adding more computational resources or optimizing system performance. This requirement is particularly important for cloud-based systems that are expected to support continuous growth.[13], [16]

2.6.3 Security

Security in multi-tenant systems involves protecting both the system and tenant data from unauthorized access. This includes implementing strong authentication mechanisms to verify user identities and authorization controls to ensure that users can only access resources within their assigned tenant. Additional security measures, such as encryption and monitoring, are often required to protect sensitive information and prevent potential attacks.[13]

2.6.4 Maintainability

Maintainability refers to the ease with which the system can be updated, corrected, and improved over time. In a multi-tenant environment, updates and bug fixes should be applied centrally without disrupting the operation of individual tenants. A well-designed system should allow continuous improvement while minimizing downtime and ensuring service availability for all users.[13]

2.6.5 Customization

Customization allows different tenants to adapt the system according to their specific needs. This may include configuring workflows, user roles, or interface elements. A flexible multi-tenant system should support such variations without affecting the core system or other tenants. This requirement is important for supporting organizations with different operational models within the same platform.[18]

2.7 Benefits and Challenges of Multi-Tenant Architecture

Multi-tenant architecture offers a set of significant advantages that make it highly suitable for modern scalable systems, particularly in cloud-based environments. [13] Its primary benefits include cost efficiency through shared infrastructure, simplified maintenance due to centralized deployment and updates, improved scalability with the ability to onboard new tenants easily, optimized resource utilization, and faster deployment of new clients without the need for duplicating system components.[19]

However, despite these advantages, multi-tenancy also introduces important technical and architectural challenges that must be carefully addressed. These include security risks related to improper data isolation between tenants, performance contention where resource consumption by one tenant may impact others, increased system complexity due to the need for tenant-aware logic across multiple layers, and challenges associated with data migration and scaling, which require careful planning to ensure system consistency and reliability.[16], [20]

Overall, while multi-tenancy provides a powerful foundation for scalable and efficient system design, its successful implementation depends on rigorous attention to security, performance management, and architectural complexity.[13], [16]

2.8 Cloud Computing Overview

Cloud computing refers to a model that enables on-demand access to a shared pool of configurable computing resources, such as servers, storage, networks, and applications, which can be rapidly provisioned and released with minimal management effort. [14] Instead of relying on local infrastructure, organizations can host and operate their applications on remote cloud platforms, benefiting from increased flexibility, scalability, and cost efficiency.[19]

Cloud computing is characterized by several essential features. It provides on-demand self-service, allowing users to access resources whenever needed without requiring direct interaction with service providers. It also relies on resource pooling, where computing resources are shared among multiple users in a centralized infrastructure. Another important characteristic is rapid elasticity, which enables systems to scale dynamically based on workload demands. In addition, cloud platforms offer measured services, where resource usage is monitored, controlled, and optimized.[14]

These characteristics make cloud computing a fundamental enabler of modern distributed systems, especially those that require scalability and efficient resource utilization.

2.8.1 Cloud Service Models and Architectural Aspects

Cloud computing is commonly structured into different service models, which define how resources and responsibilities are distributed between the provider and the user. [14] The main models include:

- **Infrastructure as a Service (IaaS),** which provides virtualized computing resources such as servers, storage, and networking. Users have control over the operating systems and deployed applications, while the cloud provider manages the underlying infrastructure.[14]
- **Platform as a Service (PaaS),** which offers a development and deployment environment where users can build and run applications without managing the underlying hardware or operating systems.[14]
- **Software as a Service (SaaS),** which delivers complete applications over the internet. Users access the software through a web interface, while the provider manages the entire system, including infrastructure and application logic. [13], [16]

From an architectural perspective, cloud systems are designed to support scalability, high availability, fault tolerance, and load balancing. These capabilities are essential for ensuring that applications remain responsive and reliable even under high demand. Cloud environments also promote centralized deployment and management, which simplifies system updates and maintenance.[16], [19]

2.9 Relationship Between Cloud Computing and Multi-Tenancy

Cloud computing and multi-tenancy are closely related but distinct concepts that together support modern large-scale distributed systems. [14] While cloud computing focuses on infrastructure provisioning, multi-tenancy defines how applications are designed to serve multiple clients efficiently. Their combination is especially important in the context of Software as a Service (SaaS), where a single application is delivered over the internet to multiple organizations through a shared platform.[16]

From an infrastructure standpoint, cloud computing provides on-demand access to virtualized computing resources such as processing power, storage, and networking. These resources are elastically allocated based on workload requirements, enabling

systems to scale dynamically while maintaining high availability and fault tolerance. This abstraction reduces dependency on physical hardware and allows centralized management of distributed environments.[14]

At the application level, multi-tenancy is an architectural model in which a single application instance serves multiple independent tenants. [13] Each tenant's data and configuration are logically isolated, despite sharing the same underlying system. [16] This isolation is enforced through mechanisms such as tenant-aware data partitioning and access control policies, ensuring security and operational independence.

In enterprise applications, particularly in Field Service Management systems, this model enables the delivery of a single, centrally managed platform that can support multiple organizations simultaneously. [3] Each tenant operates within its own isolated context while benefiting from shared infrastructure, simplified deployment, and easier system maintenance.

2.10 Conclusion

Multi-tenant architecture represents an important approach for designing modern software systems that must support multiple organizations within a shared environment while preserving data isolation, security, and operational independence.

Cloud computing provides the technical foundation required for such systems through scalability, flexibility, and centralized resource management. The combination of cloud computing and multi-tenancy therefore offers an effective basis for building efficient and scalable platforms.

These concepts are particularly relevant to the proposed Field Service Management system, whose objective is not only to improve the management of field operations, but also to support multiple organizations within the same platform. They therefore provide the theoretical and architectural basis for the analysis and design of the proposed solution

3 Chapter 03: Conceptual Design of a Field Service Management System

3.1 Introduction

After presenting the theoretical background of Field Service Management and multi-tenant architecture, the next step is to analyze and design the proposed system. This phase is essential because it transforms the identified needs and architectural concepts into a structured system model.

The objective of this chapter is to define the main requirements of the proposed platform, identify its actors, and describe the interactions that take place within the system. It also presents the overall architecture, the database design, and the main design decisions adopted for the multi-tenant Field Service Management solution.

3.2 System Overview and Objectives

The proposed system, named **FieldOps**, is a cloud-hosted, multi-tenant web and mobile application designed to digitize and centralize the management of field service operations. It is intended for organizations that deploy teams of technicians to customer sites to perform activities such as maintenance, installation, repair, and inspection.

FieldOps is accessible through two main interfaces: a web application intended for administrative and managerial users, and a mobile application dedicated to field operatives and customers. This dual-interface design supports both back-office management and field-level interaction within a unified platform.

The platform is structured around this main functional domain:

1. **Tenant and subscription management**, handled by System Administrators.
2. **Personnel and team management**, performed through the web application.
3. **Service request lifecycle management**, covering request creation, scheduling, assignment, execution, and closure.
4. **Field-level interaction**, supported through the mobile application.
5. **Customer visibility and service tracking**, allowing customers to monitor the progress of their service requests.
6. **Customer feedback management**, enabling customers to submit feedback after service completion.

7. **Reporting and traceability management**, ensuring that field activities, status changes, and work reports are properly recorded and monitored.
8. **Coordination and operational follow-up**, improving communication between managers, employees, technicians, and customers.

Multi-tenancy is enforced at the architecture level so that each subscribing organization - tenant - operates within a logically isolated environment while sharing the same underlying infrastructure. This design ensures data separation, secure access, and efficient resource utilization across the platform.

3.3 Requirements Analysis

Requirements analysis is the process of identifying, studying, and organizing the needs that the proposed system must satisfy. It helps define the expected system behavior, the services provided to users, and the constraints that must be respected during development.

3.3.1 System Actors

FieldOps involves several actors, each with a specific role in using and managing the platform. These actors interact with the system according to their responsibilities and access permissions within the multi-tenant environment.

The identified system actors are as follows:

- **System Administrator**: Manages the platform at the global level. This actor creates tenants, creates an admin account for each tenant, manages subscriptions, and monitors the system.
- **Tenant Administrator**: Responsible for managing the internal environment of a specific tenant. This includes managing users, employees, technicians, teams, and organizational settings within the tenant.
- **Manager**: Oversees daily operations within the tenant. This actor supervises employees and technicians, assigns tasks, monitors progress, and helps coordinate service activities.
- **Employee**: Handles service request-related tasks such as scheduling, prioritization, and coordination. This actor may create service requests when a customer contacts the organization by phone, update request information, and support the coordination of technicians and teams.

- **Technician** (Mobile Application): Performs field tasks such as maintenance, installation, repair, and inspection. This actor views assigned tasks, updates their status, and sends work reports.
- **Customer** (Mobile Application): Creates service requests, manages assets and addresses, tracks service progress, and provides feedback.

These actors represent the main users of FieldOps system and reflect both the administrative and operational aspects of the platform. Their identification is essential for defining system functions, access control rules, and user interactions.

3.3.2 Functional Requirements

Functional requirements describe the services and actions that the system must provide to its users. They define what the system should do in order to support user needs and business processes

System Administrator Requirements

The system must allow the Administrator to perform the following functions:

- ✓ Review registration requests.
- ✓ Create, edit, activate, and deactivate tenants.
- ✓ Create, edit, activate, and deactivate Tenant Administrator accounts.
- ✓ Manage subscription plans by creating, updating, deleting, and viewing them.
- ✓ Apply discounts and manage tenant plan changes or cancellations.

Tenant Admin Requirements (Web Application)

The system must allow the Tenant Administrator to perform the following functions:

- ✓ Create, edit, delete, activate, and deactivate Manager accounts.
- ✓ View manager information.
- ✓ Manage tenant-specific organizational settings (Company informations).

Manager Requirements (Web Application)

The system must allow the Manager to perform the following functions:

- ✓ Manage technician profiles by creating, viewing, editing, deleting, and activating or deactivating them.
- ✓ Manage employee accounts by creating, viewing, editing, deleting, and activating or deactivating them.

- ✓ Manage teams by creating, viewing, editing, deleting, assigning technicians, and defining a team leader.
- ✓ Manage customer records by creating, viewing, editing, and deleting customer information.
- ✓ Manage service requests by creating, viewing, editing, scheduling, canceling, and assigning them to technicians or teams.
- ✓ Assign service requests to employees when needed, especially when they are responsible for handling or updating the request information.
- ✓ Define service request details such as priority, due date, and cost.
- ✓ Track the status history of service requests.
- ✓ View the operational dashboard.

Employee Requirements (Web Application)

- ✓ Create service requests when a customer calls by phone.
- ✓ Create Employee Accounts.
- ✓ View service requests assigned to him and unassigned service requests.
- ✓ Assign unassigned service requests to himself.
- ✓ Manage service requests by creating, viewing, editing, scheduling, canceling, and assigning them to technicians or teams.
- ✓ Manage customer-related data (assets, locations.).
- ✓ Support daily service and administrative operations.

Technician / Field Operator Requirements (Mobile Application)

- ✓ View assigned service requests.
- ✓ Update the status of a service request.
- ✓ Manage work reports: create, view, edit, and delete reports attached to a request.
- ✓ Manage attachments: upload, view, edit, and delete files linked to a request or report.

Customer

- ✓ Submit a new service request by entering request information, selecting an asset, and setting the priority.
- ✓ View submitted requests and cancel a pending request.
- ✓ Manage personal assets by adding, viewing, editing, and deleting them.

- ✓ Manage personal addresses.
- ✓ Track service progress.
- ✓ Provide feedback after service completion.

3.3.3 Non-Functional Requirements

Non-functional requirements describe the quality attributes and constraints of the system. They define how the system should perform and the conditions it must satisfy to ensure reliability, security, usability, and efficiency.

The main non-functional requirements of the proposed system are as follows:

- **Security**: The system must ensure secure authentication and authorization for all users. Each actor must only access the functions and data allowed by their role, and tenant data must remain isolated from other tenants.
- **Performance**: The system must respond quickly to user actions and support smooth execution of operations such as request creation, assignment, and status updates.
- **Scalability**: The system must be able to support an increasing number of tenants, users, service requests, and mobile clients without significant degradation in performance.
- **Availability**: The system should remain accessible and operational for users whenever needed, especially for service coordination and field activities.
- **Usability**: The web and mobile interfaces must be simple, clear, and easy to use for all actors, including administrators, managers, technicians, and customers.
- **Reliability**: The system must provide consistent operation and preserve data accuracy during task execution, report submission, and service tracking.
- **Maintainability**: The system should be easy to update, improve, and maintain over time, especially as tenant needs evolve.
- **Data Integrity**: The system must ensure that stored data remains accurate, complete, and consistent across different operations and users.
- **Compatibility**: The system must support both web and mobile platforms and function correctly across commonly used devices and browsers.

3.3.4 Use Case Diagram

Use case Diagram describe the main interactions between the system and its actors. They provide a high-level representation of the services offered by the system and show how each actor uses these services to achieve specific goals.

In this work, three use case diagrams were developed to represent the main functional areas of the platform. These diagrams correspond to three principal actor groups: the System Administrator, the web application users, and the mobile application users.

- **System Administrator Use Case Diagram**

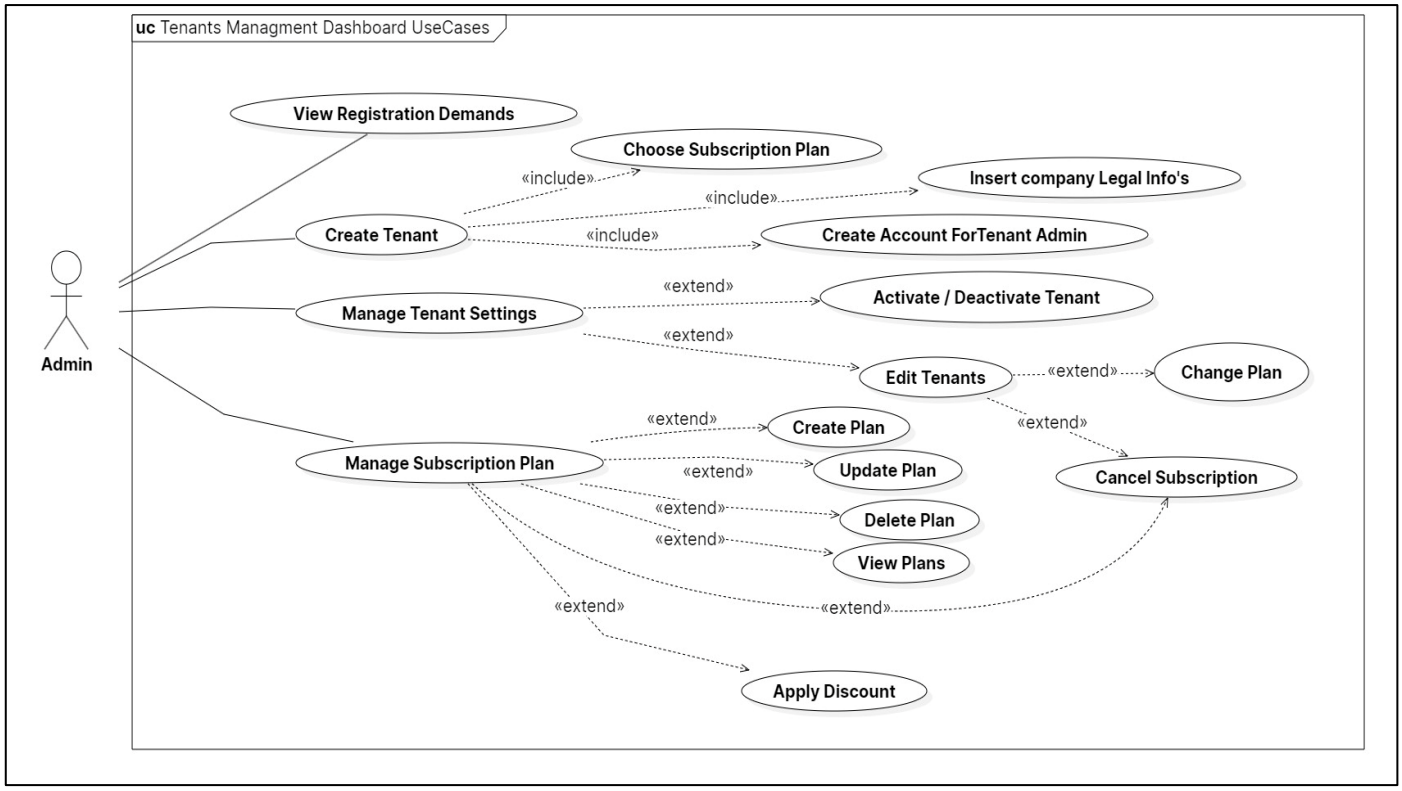


Figure 3.1. *System Administrator Use Case Diagram*

Figure 3.1 presents the use case diagram associated with the System Administrator. This actor interacts with the system at the platform level and is responsible for managing functions that affect all tenants.

The diagram is organized around Two main functional areas: **Tenant Management**, **Subscription Plan Management**.

- ❖ Tenant Management includes creating new tenants, editing tenant information, activating or deactivating tenant accounts, and managing tenant settings.
- ❖ The tenant creation process may also include selecting a subscription plan, entering commercial information, and generating login credentials.
- ❖ Subscription Plan Management includes creating, updating, deleting, and viewing subscription plans. It also covers plan changes, subscription cancellations, and discount management for subscribed tenants. Platform User Management includes creating, viewing, editing, deleting, and activating or deactivating platform-level user accounts.

- ➢ **Web Application Use Case Diagram**

The system defines a three-tier actor hierarchy in which TenantAdmin sits at the apex, Manager inherits from Employee, and TenantAdmin in turn inherits from Manager, as represented through UML generalization arrows in the actor diagram (Figure 3.2).

The hollow-headed arrows point downward from child to parent Employee points to Manager, and Manager points to TenantAdmin following standard UML generalization notation. In this hierarchy, TenantAdmin can perform all the roles and responsibilities of both Manager and Employee, while neither Manager nor Employee can perform the roles above them.

Similarly, Manager can perform all the roles of Employee, but Employee cannot perform the roles of Manager. Each subordinate actor inherits the privileges of its parent while extending the system's functional scope with responsibilities specific to its own role.

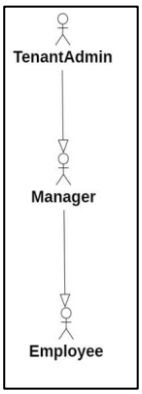


***Figure 3. 2** Actor Generalization Hierarchy*

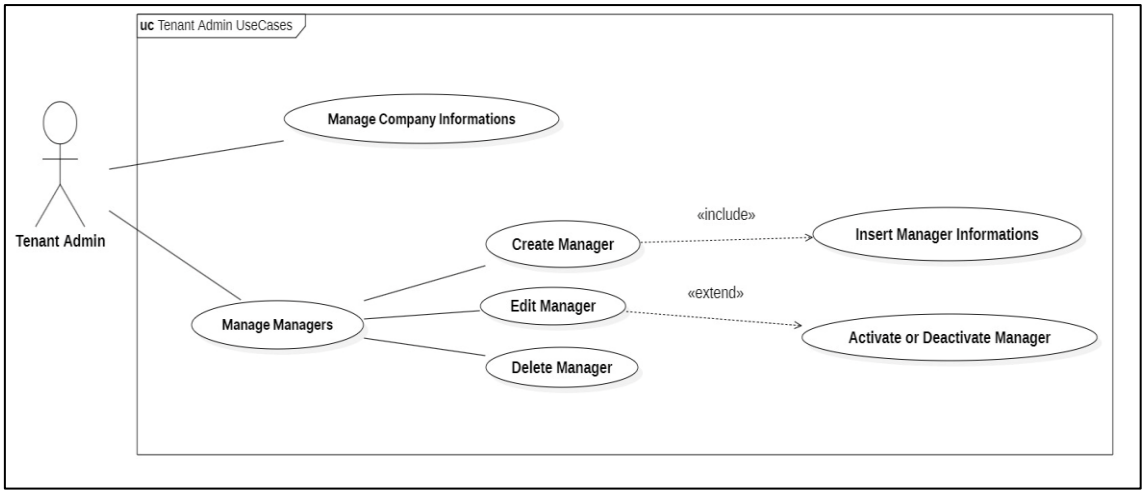


Figure 3.3. *Tenant Admin Use Case Diagram*

The Tenant Admin, as illustrated in **Figure 3.3**, is assigned two principal functional clusters. The first, Manage Company Information's, is an atomic standalone use case for configuring organizational profile data. The second, Manage Managers, governs the full lifecycle of manager accounts, decomposed into Create Manager, which mandatorily includes Insert Manager Information's alongside Edit Manager, which conditionally extends to Activate or Deactivate Manager, and Delete Manager.

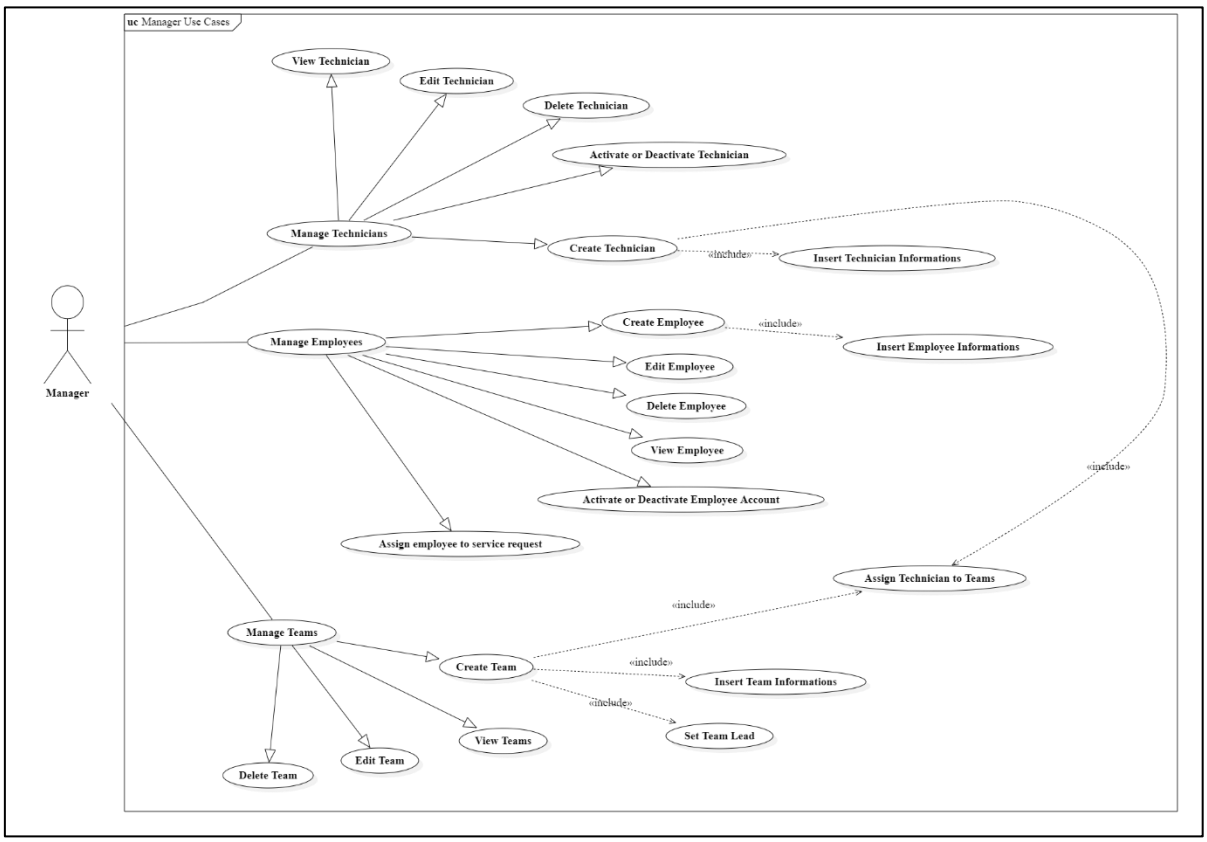


Figure 3 4*. Manager Use Case Diagram*

The Manager actor, as shown in **Figure 3.4**, operates across three management domains. Under Manage Technicians and Manage Employees, the Manager controls the full account lifecycle of both roles creation, editing, deletion, viewing, and activation where creation in both cases mandatorily includes the insertion of the corresponding profile information. The Manager is further authorized to assign employees to service requests and to administer operational teams through Manage Teams, where Create Team mandatorily includes Insert Team Informations, Set Team Lead, and Assign Technician to Teams, while Edit, View, and Delete Team complete the domain.

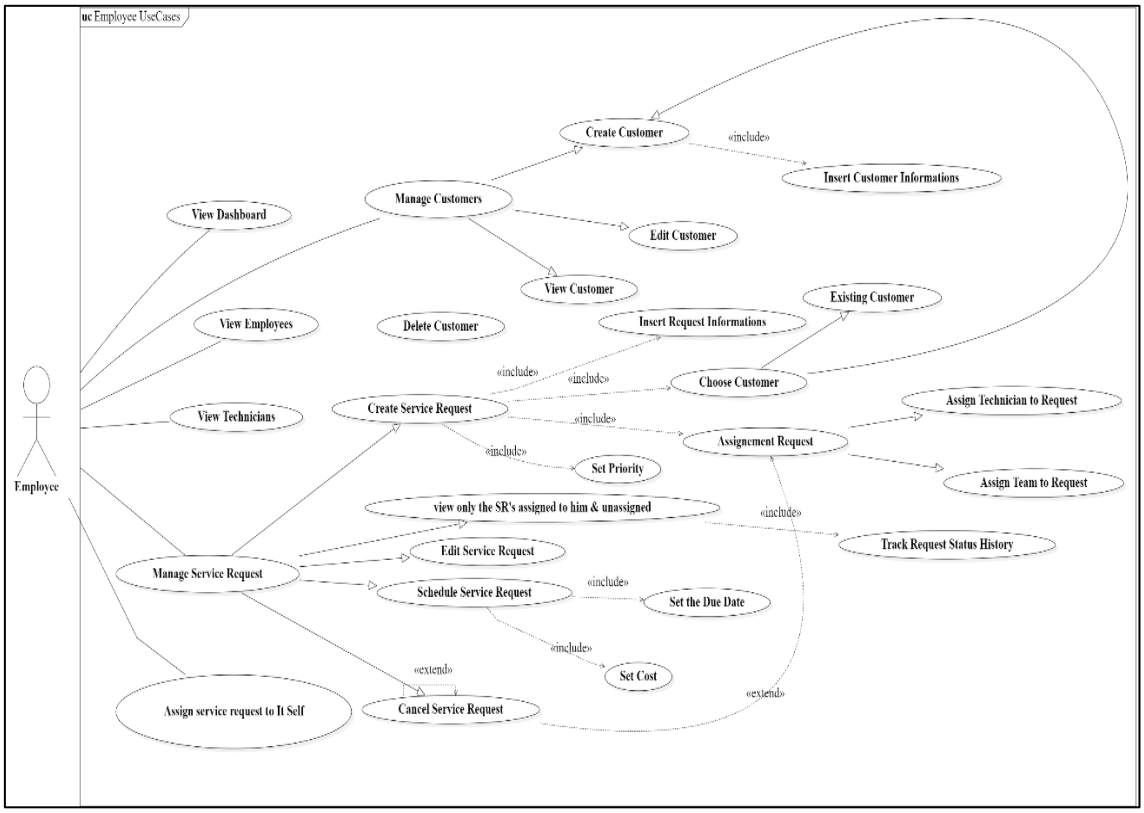


Figure 3.5. *Employee Use Case Diagram*

The Employee actor, depicted in **Figure 3.5**, interacts with the widest operational surface of the system across three domains. In customer management, the Employee creates, edits, views, and deletes customer records, where creation mandatorily includes inserting customer information. In service request management the core of the Employee's workflow the Employee creates service requests by mandatorily including the insertion of request information, customer selection, priority assignment, request assignment, and status history tracking, while optionally extending into editing, scheduling with a due date, cancellation with cost setting, and self-assignment. The

Employee also holds read-only access to the dashboard, employee, and technician directories for situational awareness.

- **Mobile Application Use Case Diagram**

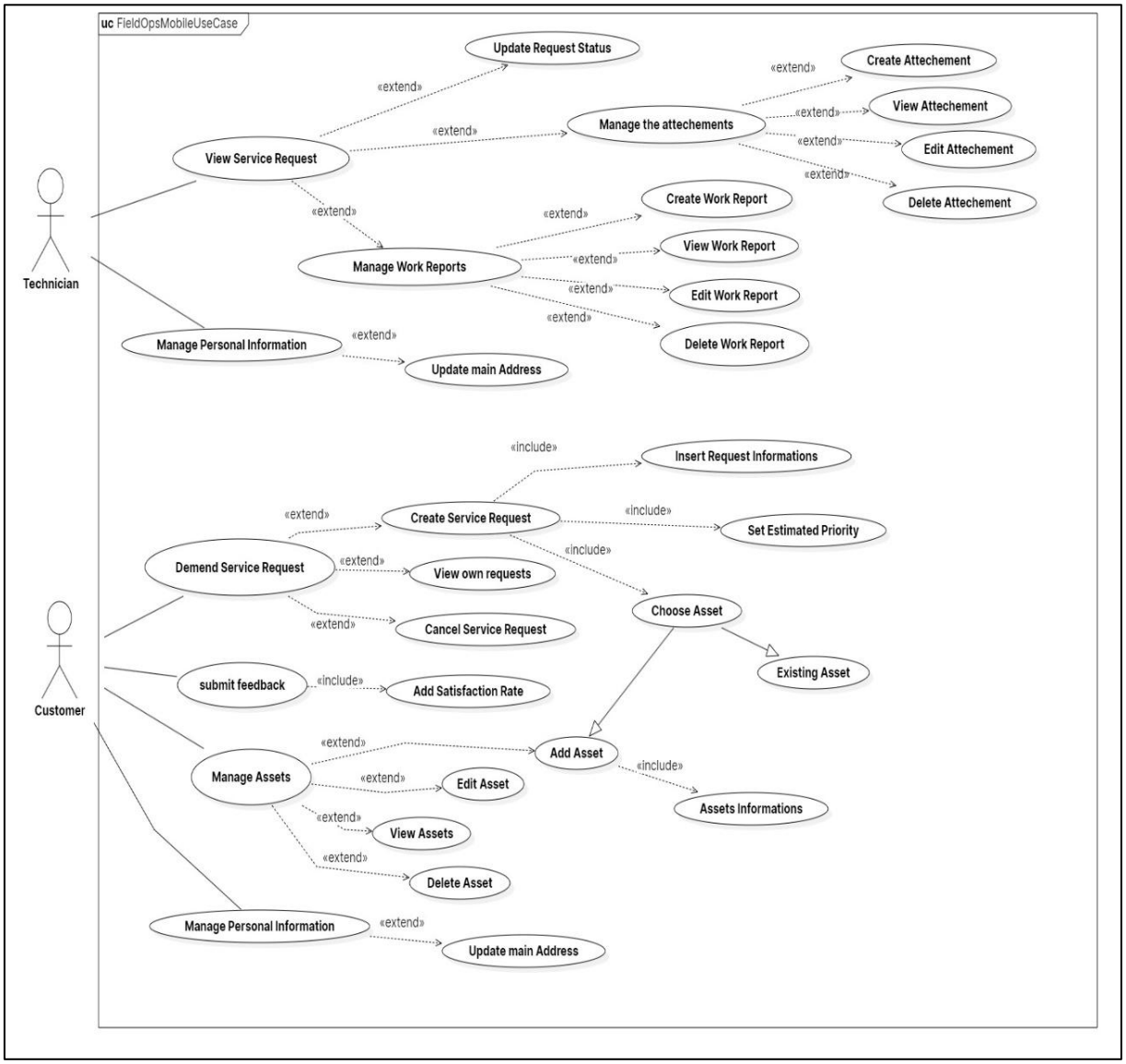


Figure 3.6. *Field Ops Mobile Application Use Case Diagram*

Figure 3.6 presents the use case diagram of the mobile application. This diagram involves two main actors: the Technician and the Customer.

The Technician uses the mobile application to access assigned service requests, update request status, and manage work reports and attachments related to field interventions. The Customer uses the mobile application to submit service requests, enter

request details, select or specify an asset, set request priority, view submitted requests, cancel pending requests, manage personal assets, and follow service progress.

3.4 System Design

In the proposed multi-tenant Field Service Management system (MT-FSM) , the design phase focuses on building a platform that supports multiple tenants, ensures data isolation, and provides dedicated functionalities for different actors through web and mobile applications. It also defines how the system handles service requests, user management, reporting, and communication between the different parts of the platform.

The purpose of this section is to present the overall design of the system and describe its main architectural elements. It provides a structured view of how the proposed solution is organized in order to satisfy operational needs, support scalability, and ensure secure and efficient service delivery.

3.4.1 System Architecture

The proposed MT-FSM system is built on a layered architecture that separates the user interface, business logic, and data management concerns. This structure makes the system easier to maintain, scale, and extend while supporting both web and mobile access.

Presentation Layer

The presentation layer includes two client applications: a **web application** for administrative and operational users, and a **mobile application** for technicians and customers. These clients communicate with the backend through **RESTful APIs**, which ensures a clear separation between the interface and the server-side logic.

Application Layer

The application layer contains the core business logic of the system. It processes requests, applies validation rules, manages user actions, and coordinates platform operations. It also enforces tenant isolation by using the tenant context extracted from the authenticated user token.

Data Layer

The data layer stores and manages all persistent information in a relational database. It contains tenants, users, service requests, assets, work reports, subscriptions,

and other operational data. The database follows the selected multi-tenant strategy, where tenant-specific records are separated logically using a **TenantId** field.

This architecture was selected because it provides a clear separation of concerns, supports secure multi-tenant operation, and allows the system to grow more easily over time.

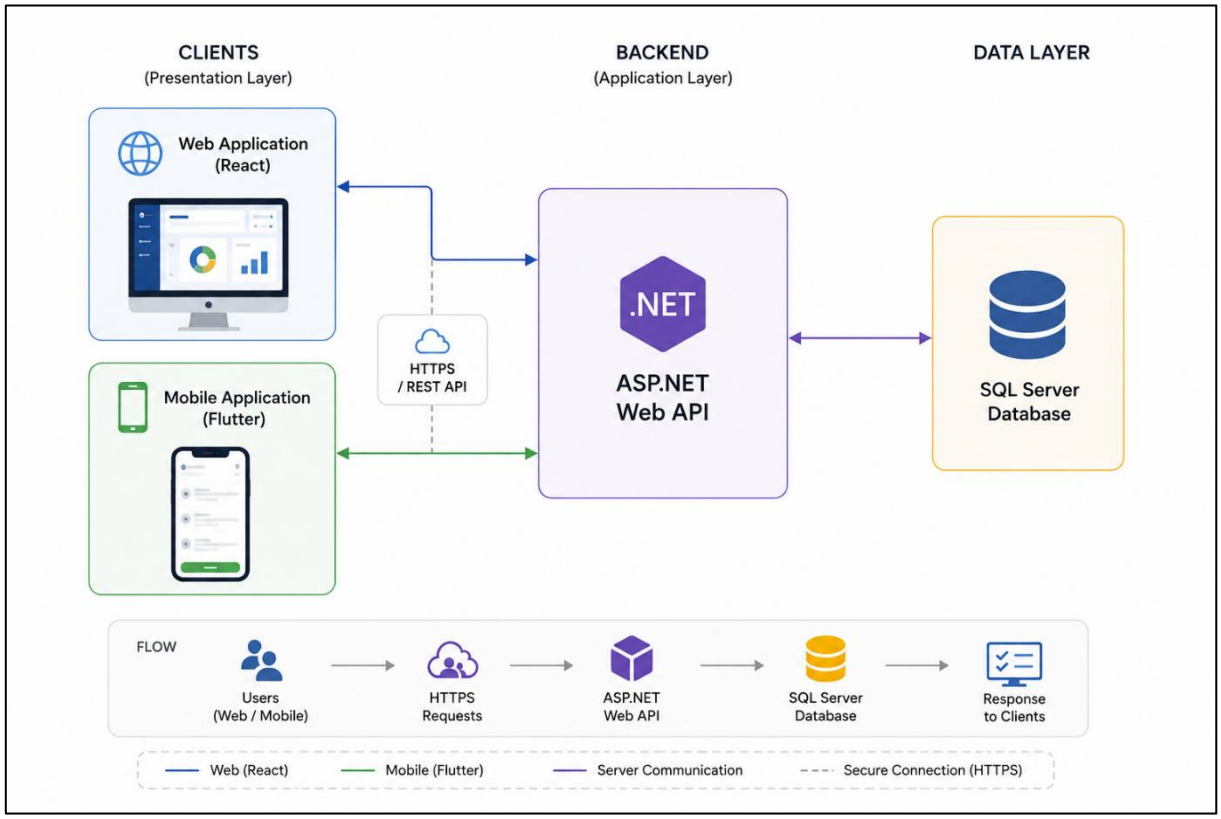


Figure 3.7. *System Architecture Diagram (Three-Tier Client-Server Architecture)*

3.4.2 Selected Multi-Tenant Model

The proposed system adopts the **shared database with shared schema** multi-tenant approach, enriched with a **TenantId** discriminator column. In this model, all tenants share the same relational database and the same schema, while each tenant-specific record is associated with a **TenantId** that identifies the tenant to which the data belongs. This mechanism makes it possible to separate tenant data logically while maintaining a unified database structure.

This approach was selected for several reasons:

- ✓ **Cost efficiency**: A single database instance reduces infrastructure and operational costs, which makes this approach suitable for startups and small or medium-sized SaaS platforms.

- ✓ **Deployment simplicity**: There is no need to create a separate database or schema whenever a new tenant is added. Tenant onboarding can be handled directly through the tenant creation workflow.
- ✓ **Ease of maintenance**: Since all tenants use the same schema, updates, backups, and maintenance operations are easier to manage.
- ✓ **Scalability**: The system can support growth by increasing database resources and deploying multiple application server instances when needed.

To ensure robust tenant isolation, the system enforces access control at the application layer through the use of JSON Web Tokens (JWT). A JWT is a compact, cryptographically signed standard used for securely transmitting information between parties as a JSON object. Upon user authentication, the unique `TenantId` is securely embedded within the token's payload as a specific "claim." Because the token is digitally signed by the server, its integrity is guaranteed. Subsequently, with every incoming request, the system validates the JWT and extracts this claim. By doing so, all underlying database queries and business operations are automatically scoped exclusively to the authenticated tenant's context, inherently preventing cross-tenant data leakage and fortifying the platform's overall security.

This approach has **one main limitation**: it provides less physical isolation than the separate-database model, since all tenant data is stored in the same database structure. However, this limitation is reduced through strict tenant filtering, role-based access control, and input validation at the application level. Considering the expected scale of the first deployment, this trade-off is acceptable and well suited to the proposed platform.

3.4.3 Database Design

The database design defines how data is organized, stored, and related within the proposed system. It provides the structural foundation needed to support the main platform functions.

System entities

The database contains several main entities that represent the core components of the platform. These entities include:

- ❖ **Tenant**: Represents each organization using the platform.
- ❖ **User**: Stores authentication and account information for system users.

- **Role**: Defines user permissions and access levels within the system.
- **Service Request**: Represents a customer request for maintenance, repair, installation, or another field service.
- **Service Request Status History**: Stores the different status changes of a service request over time.
- **Work Report**: Contains reports created by technicians after or during an intervention.
- **Customer Feedback**: Stores feedback submitted by customers after service completion.
- **Attachment**: Stores files linked to service requests or work reports.
- **Employee**: Represents administrative or operational staff within a tenant.
- **Technician**: Represents field workers responsible for carrying out interventions.
- **Customer**: Represents the client who requests or receives the service.
- **Team**: Represents a group of technicians organized to perform service tasks.
- **Address**: Stores hierarchical location information related to customers, service requests, and technicians (City, State, Country).
- **Asset**: Represents customer equipment or service-related items that require intervention.

Each entity stores information required for a specific part of the system and is linked to other entities through well-defined relationships.

Main Relations

The main relations between the system entities are defined according to the business logic of the proposed platform. These relations ensure data consistency and reflect how different actors and processes interact within the system.

The principal relations are as follows:

- A **Tenant** can have many Users (Employees, Technicians, Customers), Teams, Assets, Location, and Service Requests.
- A **User** is associated with one **Role**, and a role can be assigned to many users.
- A **Customer** can have many **Addresses** and many **Assets**.
- A **Customer** can submit many **Service Requests**.
- A **Service Request** belongs to one **Customer** and may be linked to one **Asset**.
- A **Service Request** can be assigned to one **Technician** or one **Team**.

- A **Service Request** can have many **Status History** records.
- A **Service Request** can have many **Work Reports**.
- A **Service Request** can have many **Attachments**.
- A **Service Request** can receive one **Customer Feedback**.
- A **Team** can contain many **Technicians**.
- A **Technician** can belong to one **team** (he can be the team leader).
- A **Work Report** is created by one **Technician** and belongs to one **Service Request**.
- An **Attachment** can be linked to one **Service Request** or one **Work Report**.
- An **Asset** belongs to one **Customer** and may be linked to several **Service Requests**.
- An **Address** Linked to a specific **City** and can be associated with a **customer** or used as **a service intervention location**. It represents a physical location within a structured geographic hierarchy.
- **City** Belongs to a **State** and serves as a container for multiple addresses within its boundary.
- **State** Belongs to a **Country** and groups multiple cities under a regional administrative division.
- **Country** is the top-level geographic entity that contains multiple states.

Entity relations Diagram (ERD) of the Proposed System

The Entity Relationship Diagram (ERD) presents a visual representation of the database structure of the proposed system. It illustrates the main entities, their attributes, and the relationships between them.

In the proposed system, the ERD reflects the selected multi-tenant approach by showing the inclusion of tenant-related associations in the main operational entities. It therefore plays an important role in ensuring data consistency, tenant isolation, and proper support for the functional requirements of the platform.

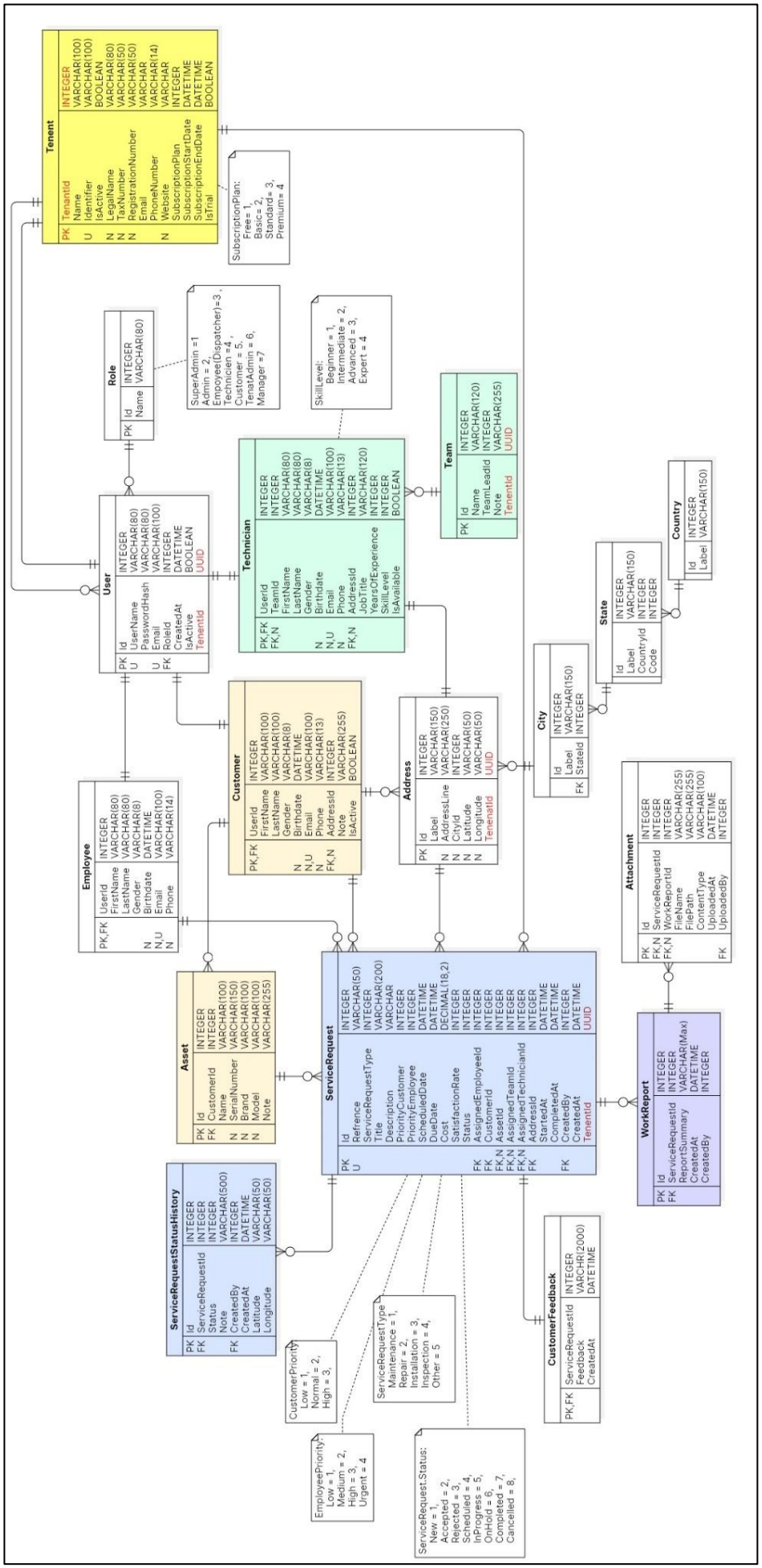


Figure 3.8. *Entity relations Diagram (ERD) of the Proposed System*

3.4.4 Class Diagram

The class diagram shown in the figure below illustrates the static structure of the proposed multi-tenant Field Service Management system. It presents the main classes of the system, their principal attributes, and the relationships between them. This diagram follows object-oriented design principles and provides a conceptual model of the software structure. It also serves as a basis for understanding the internal organization of the platform before implementation.

The main classes are described as follows:

- **Tenant**: Represents an organization subscribed to the platform. It stores information such as the tenant's name, unique identifier, subscription plan, subscription start date, subscription end date, Legal Informations and activation status. It is the main class that defines the organizational context of tenant-specific data.
- **User**: Represents the base class for all authenticated users of the system. It stores common account information such as username, email, password, activation status, and role. This class is extended by specialized user classes according to the responsibilities of each actor.
- **Employee**: Represents administrative staff involved in internal support and service-related operations. This class is associated with request handling, customer data management, and daily operational support.
- **Technician**: Represents the field worker responsible for carrying out maintenance, repair, installation, and inspection tasks. This class stores information such as technical profile, availability, experience, and assignment to a team. A technician can be assigned to service requests and can create work reports.
- **Customer**: Represents the service requester or service recipient. This class stores personal and contact information and is linked to assets, addresses, service requests, and feedback. It allows the customer to interact with the system through the mobile application.
- **Team**: Represents a group of technicians organized to perform field tasks. It stores team information such as name, label, and team leader. A team can be assigned to service requests and may include several technicians.

- **ServiceRequest**: Represents the central operational class of the system. It stores information related to a service intervention request, such as title, description, request type, priority, current status, due date, scheduling information, cost, customer, asset, and assignment details. It is linked to several other classes that describe the lifecycle of the intervention.
- **ServiceRequestStatusHistory**: Represents the history of status changes of a service request. It stores the different states through which a request passes, such as pending, assigned, scheduled, in progress, completed, or canceled. This class helps track the evolution of the service process over time.
- **WorkReport**: Represents the report created by a technician during or after an intervention. It contains operational details about the work performed, observations, technical notes, and completion information related to a service request.
- **Attachment**: Represents a file associated with a service request or a work report. It stores file-related information such as file name, path, type, upload date, and the user responsible for the upload.
- **Customer Feedback**: Represents the feedback submitted by a customer after the completion of a service request. It may include comments, ratings, or satisfaction information used to evaluate service quality.
- **Asset**: Represents customer equipment or service-related items that may require maintenance, repair, or inspection. It stores information such as asset name, serial number, brand, model, and description. An asset is linked to a customer and may be associated with one or more service requests.
- **Address**: Represents a physical address associated with a customer or used as a service intervention location. It stores detailed location information required for service delivery.
- **City**: Represents the city to which an address belongs. This class helps organize location data in a structured way and supports geographical classification.
- **State**: Represents the regional or administrative division that contains multiple cities. It is used to structure the geographical hierarchy of addresses.
- **Country**: Represents the highest geographical level in the address hierarchy. It contains one or more states and helps standardize location management across the system.

- **AuthService**: Represents the authentication service of the platform. It is responsible for login, token generation, logout, and access control. It also supports tenant-aware authentication by including the tenant context in the authentication process.

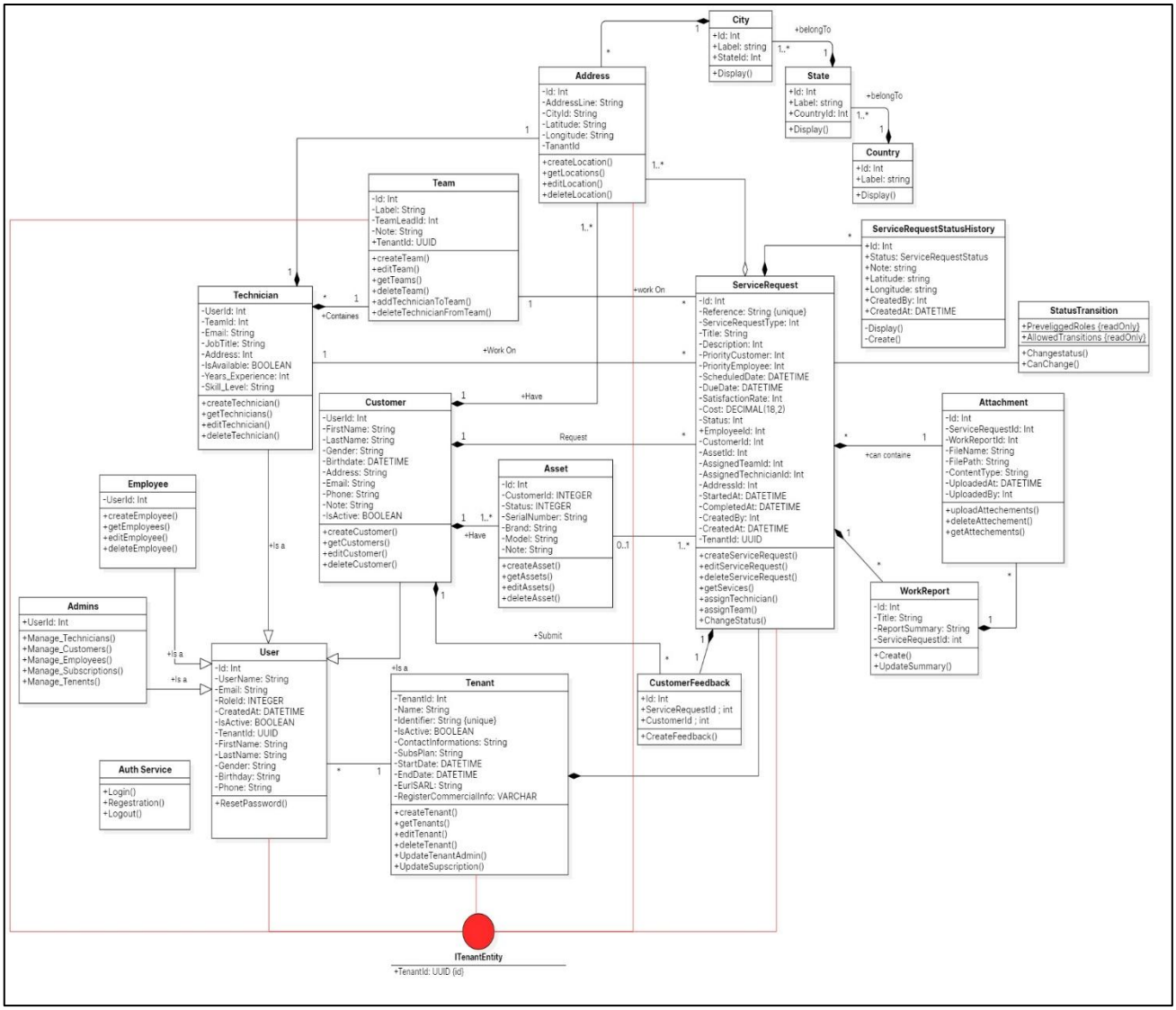


Figure 3.9. *Class Diagram of proposed system*

3.4.5 Sequence Diagrams

Sequence describes the dynamic interactions between system components over time. Two sequence diagrams were developed to represent the most important runtime behaviors of the MT-FSM system: the authentication flow and the HTTP request lifecycle.

Multi-Tenant Authentication Sequence Diagram

Figure 3.5 illustrates the login sequence for the multi-tenant authentication mechanism. The design uses a JWT-based approach in which the tenant context is included in the authentication token to support stateless and tenant-aware request processing throughout the application.

The sequence begins when the user enters an email address and password through the web or mobile application. The application sends a login request to the authentication endpoint. The authentication service then searches for the user by email in the database.

If the user is not found, the system returns an unauthorized response to the client. If the user exists, the authentication service verifies the password using a secure hashed comparison. If the password is invalid or the tenant is inactive, the system also returns an unauthorized response.

If the credentials are valid, the authentication service retrieves the user role, claims, and tenant information. It then generates a signed access token and creates a refresh token. Before saving the new refresh token, the system deletes any old refresh tokens associated with the same user. The new refresh token is then stored in the database.

Finally, the authentication service returns the access token and refresh token to the client, and the login response is completed successfully.

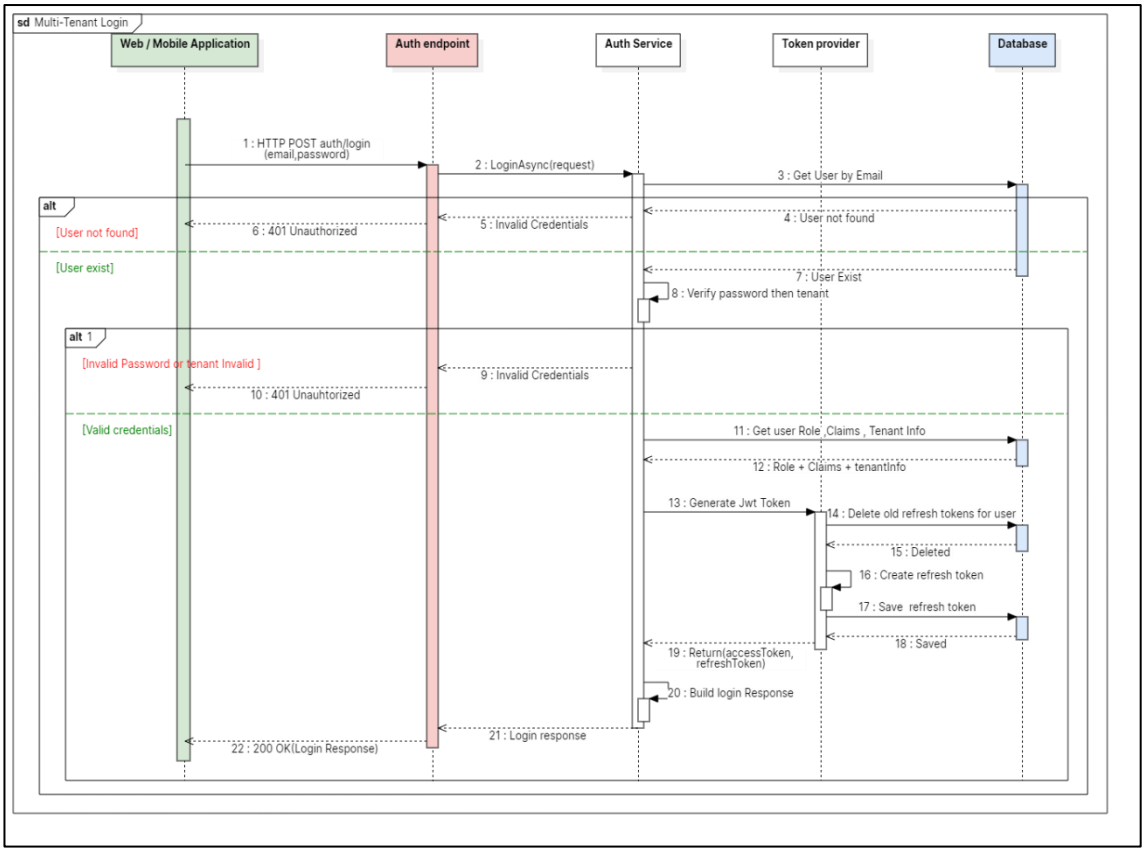


Figure 3.10. *Multi-Tenant Authentication Sequence Diagram*

HTTP Request Lifecycle Sequence Diagram

The second sequence diagram illustrates the flow of an HTTP request within the MT-FSM system after authentication. It shows how the tenant context is extracted from the JWT token and propagated through the request processing pipeline to ensure tenant-aware data access.

The sequence begins when the web or mobile application sends an HTTP GET request to retrieve the list of technicians. The tenant middleware intercepts the request and extracts the tenant identifier from the JWT token. If the tenant identifier is invalid, the system returns a 401 Unauthorized response to the client.

If the tenant identifier is valid, the middleware stores it in the request context and forwards the request to the technician endpoint. The technician endpoint then calls the technician service to process the request. The technician service retrieves the tenant identifier from the request context and uses it to query the database for technicians belonging to that tenant only.

After the database returns the technicians data, the service builds the response and sends the technicians list back through the endpoint and middleware layers. Finally, the web or mobile application receives a successful 200 OK response containing the list of technicians.

This diagram highlights how tenant isolation is enforced during request processing. It also shows that every request is scoped to the current tenant before accessing the database, which prevents cross-tenant data access.

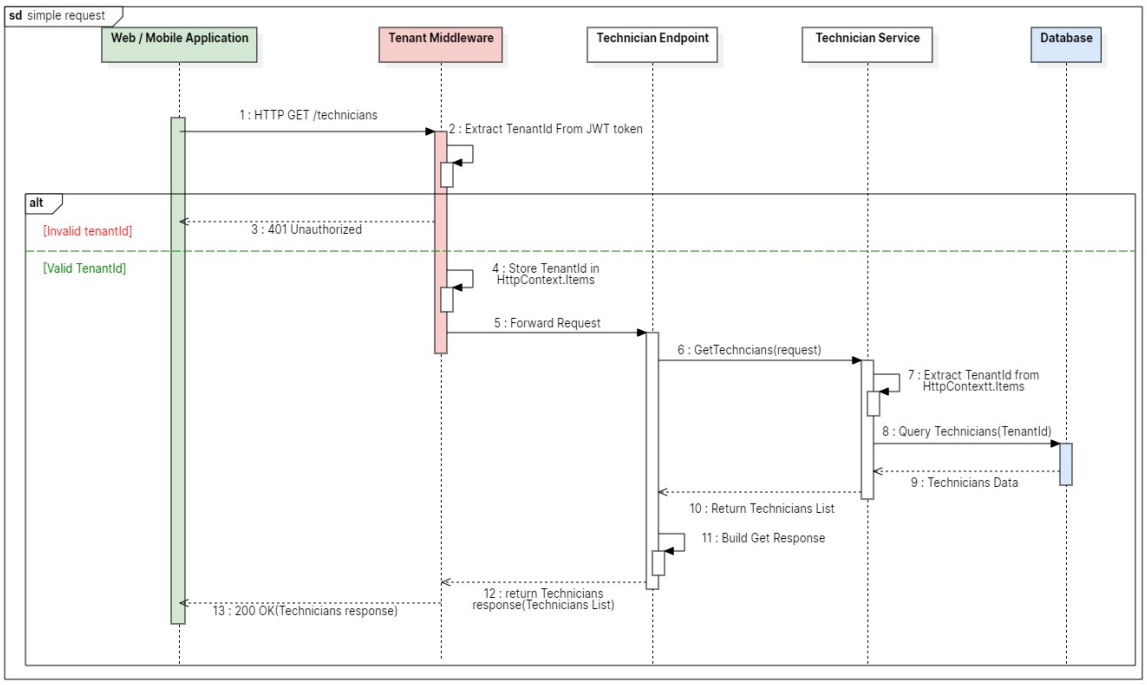


Figure 3.11. *HTTP Request Lifecycle Sequence Diagram*

3.4.6 State Diagram

The lifecycle of a single object by showing the states it can occupy and the transitions triggered by events or conditions. In the MT-FSM system, the most important lifecycle belongs to the ServiceRequest entity, whose evolution from creation to completion involves several actors and role-based transition rules.

Service Request States

The ServiceRequest entity can occupy the following principal states during its lifecycle:

- **New**: The initial state after a service request is created by a customer or an employee. The request is waiting to be processed.
- **Scheduled**: The request has been reviewed and scheduled by an employee.
- **Worker Accepted**: A technician has accepted the scheduled request and is ready to handle it.
- **InProgress**: The technician has started working on the service request.
- **On Hold**: The technician has temporarily paused the work on the request.
- **Completed**: The technician has finished the intervention successfully.
- **Rejected**: The request has been rejected by an employee or a technician, depending on the current stage.
- **Cancelled**: The request has been cancelled by the customer or an employee.
- **Failed**: The intervention could not be completed successfully due to a technical or operational issue.

The service request is governed by the **Service Request Status** logic, which defines the valid transitions according to the user role. For example,

- ✓ Only an Employee can move a request from New to **Scheduled** or **Rejected**.
- ✓ A Customer or an Employee can cancel a request from the **New** or **Scheduled** state.
- ✓ Once a request is **Scheduled**, only a technician can move it to **WorkerAccepted** or **Rejected**.
- ✓ The Technician can then move the request from **WorkerAccepted** to **InProgress**, **Rejected**, or **Cancelled**.
- ✓ During execution, the Technician can also move the request from **InProgress** to **OnHold**, **Completed**.

- ✓ From **OnHold**, the Technician can return the request to **InProgress** or mark it as **Failed**.
- ✓ Finally, a request in the **Rejected** state can still be moved to **Cancelled** by an Employee or a Customer.

These transition rules ensure that the service request lifecycle remains consistent and that only authorized roles can perform specific state changes. The state diagram therefore provides a clear view of the allowed progressions and exceptional paths within the proposed system. It also complements the sequence diagrams by showing the lifecycle of the request from a state-based perspective.

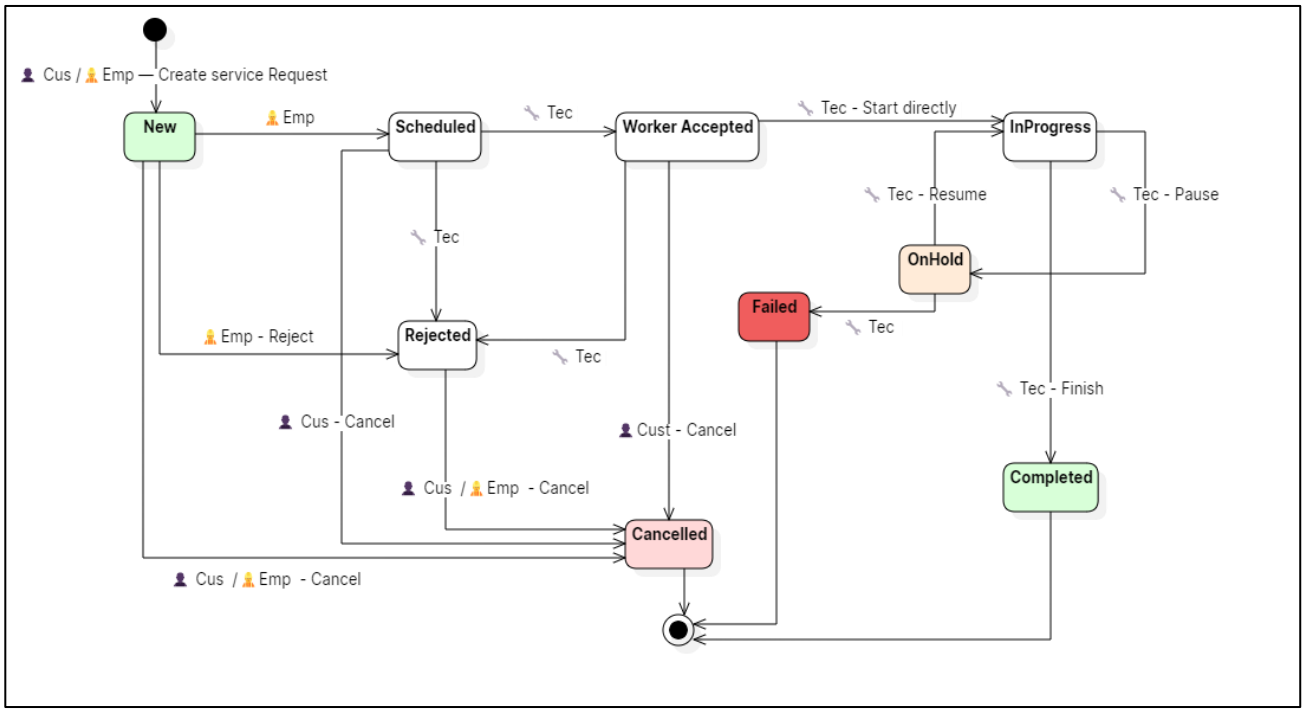


Figure 3.12. *Service Request Status State Diagram*

3.5 Conclusion

This chapter presented the analysis and design of the Multi-Tenant Field Service Management system. It began by identifying the main actors of the platform and defining the functional requirements associated with each one. It then described the selected multi-tenant architecture and justified the use of the Shared Database with Shared Schema approach as a practical solution for cost, scalability, and maintainability.

The UML models developed in this chapter provide a complete view of the proposed system. The use case diagrams define the functional scope of the platform from the perspective of each actor. The class diagram presents the main domain classes and highlights the tenant-based data organization through the TenantId field. The sequence diagrams describe the authentication flow and the request processing lifecycle, showing how the different system components interact at runtime. The state diagram models the lifecycle of the ServiceRequest entity and clarifies the valid status transitions according to user roles and business rules.

Together, these analysis and design artefacts establish a solid foundation for the implementation phase. The architectural choices made in this chapter, especially the multi-tenant strategy, the token-based authentication mechanism, and the modular structure of the system, guide the development decisions presented in the next chapter.

4 Chapter 04: System Implementation

4.1 Introduction

This chapter presents the implementation of the proposed Multi-Tenant Field Service Management system. It translates the analysis and design results presented in the previous chapter into a concrete software solution by describing how the system architecture, functional requirements, and multi-tenant mechanisms were realized during development.

This chapter therefore highlights the technologies, tools, and implementation strategies adopted to develop the proposed system. It also presents the main modules of the platform and explains how they cooperate to support tenant management, user management, service request processing, and secure tenant-aware access to data.

This section presents the main technologies and tools used to design, develop, test, and deploy the proposed system. It includes the development environment, the software stack adopted for the backend and frontend, and the deployment services used to make the application accessible.

4.2 Development Environment

The development environment consists of a set of software tools that support coding, testing, debugging, version control, and API validation during the implementation process.

- **Visual Studio Code (VS Code):** Visual Studio Code is an open-source source code editor developed by Microsoft. It was used for developing the Flutter mobile application and the React web application. It provides several useful features for developers, such as syntax highlighting, code completion, debugging support, extensions, integrated terminal access, and version control integration.
- **Visual Studio 2022**: Visual Studio 2022 was used for developing the ASP.NET Web API backend. It provides a complete integrated development environment with advanced features such as project management, debugging tools, code refactoring, IntelliSense, and built-in support for .NET technologies.
- **Google Chrome**: A widely used web browser that was used for testing and debugging the web application. It offers powerful developer tools that allow

developers to inspect page elements, monitor network requests, analyze performance, and detect frontend errors efficiently.

- **Postman**: A popular platform for API development and testing. It was used to test the backend endpoints, verify request and response formats, and ensure proper communication between the client applications and the server.
- **Git**: A distributed version control system used to track changes in the source code during development. It helps manage different versions of the project and supports collaboration between developers in an organized and efficient manner.
- **GitHub**: A cloud-based platform for hosting Git repositories. It was used to store the project source code remotely, manage updates, and support collaboration through features such as commit history, branch management, and issue tracking.

4.2.1 Development Stack

a) Backend Stack

- **ASP.NET Core Web API**: A framework for building HTTP-based services and RESTful APIs. It enables communication between the server and client applications by exposing endpoints, processing requests, and returning responses.[21]
- **C#**: A modern object-oriented programming language developed by Microsoft. It is widely used for building backend applications, web services, and enterprise software on the .NET platform.[22]
- **Entity Framework Core:** An Object-Relational Mapping (ORM) framework for .NET. It provides a way to interact with relational databases through objects and classes instead of writing all database queries manually.[23]
- **SQL Server:** A relational database management system developed by Microsoft. It is designed to store, organize, manage, and retrieve structured data efficiently and securely.[24]
- **Swagger**: A set of tools built around the OpenAPI Specification for designing, documenting, visualizing, and testing REST APIs. It provides interactive API documentation that helps developers explore available endpoints, parameters, request formats, and responses more easily.[25]

b) Frontend Stack

Web Stack

- **HTML**: HyperText Markup Language, is the standard markup language for creating the structure of web pages. It defines elements such as headings, paragraphs, forms, buttons, and tables.[26]
- **TypeScript**: A superset of JavaScript that adds static typing and advanced development features. It helps improve code quality, readability, and maintainability in large frontend applications.[27]
- **React**: A JavaScript library for building interactive user interfaces. It is based on reusable components and is commonly used for developing dynamic single-page applications.[28]
- **Tailwind CSS:** Tailwind CSS is a utility-first CSS framework that provides predefined classes for styling user interfaces. It helps developers design responsive and modern layouts more quickly.[29]
- **Axios:** A promise-based HTTP client used to communicate with backend APIs. It simplifies sending asynchronous requests, handling responses, and managing errors in web applications.[30]
- **React Router:** A routing library for React applications that enables navigation between pages and supports dynamic routing in single-page applications.[31]
- **React Query:** A data-fetching and state synchronization library used for managing server-side data. It provides features such as caching, background synchronization, and automatic data updates.[32]
- **Zustand:** A lightweight state management library for React applications. It provides a simple and scalable way to manage global application state without the complexity of larger state management solutions.[33]

Mobile Stack

- **Dart:** Dart is a programming language developed by Google. It is the main language used in Flutter for creating application logic, user interfaces, and interactive mobile features.[34]
- **Flutter:** Flutter is an open-source UI software development kit created by Google. It is used to build cross-platform mobile applications from a single codebase.[35]

- **Dio:** A powerful HTTP networking library for Dart and Flutter applications. It supports API communication, interceptors, request cancellation, file uploading, and advanced error handling.[36]
- **GetIt:** A service locator package used for dependency injection in Flutter applications. It helps manage and access services, repositories, and shared resources efficiently across the application.[37]
- **Go_Router:** A declarative routing package for Flutter used to manage navigation and routing between application screens. It supports nested navigation, route guards, and deep linking.[38]

4.2.2 Deployment Services

MonsterASP.NET: A hosting platform for ASP.NET-based applications. In this project, it serves as the deployment service for publishing and running the backend API so that it can be accessed online by the web and mobile applications.[39]

4.3 Presentation of the Application

The implemented application is presented from both functional and technical perspectives. It includes the backend API, its main endpoints, and the frontend user interface through screenshots and selected code excerpts that illustrate the core implementation of the system.

4.3.1 Backend Presentation

This presents the backend of the system, which is developed as a Web API. It includes the main endpoints responsible for handling requests, processing business logic, and returning responses to the client applications.

Main API Endpoints

The API endpoints of the system are documented using Swagger, which provides a clear and interactive view of the available backend operations. The Swagger interface lists the endpoints according to their corresponding controllers, making it easier to understand the structure of the Web API and the purpose of each request.

The first figure shows the authentication endpoint together with the CRUD operations for customers and employees. It provides an overview of the main authentication and user management services available in the backend.

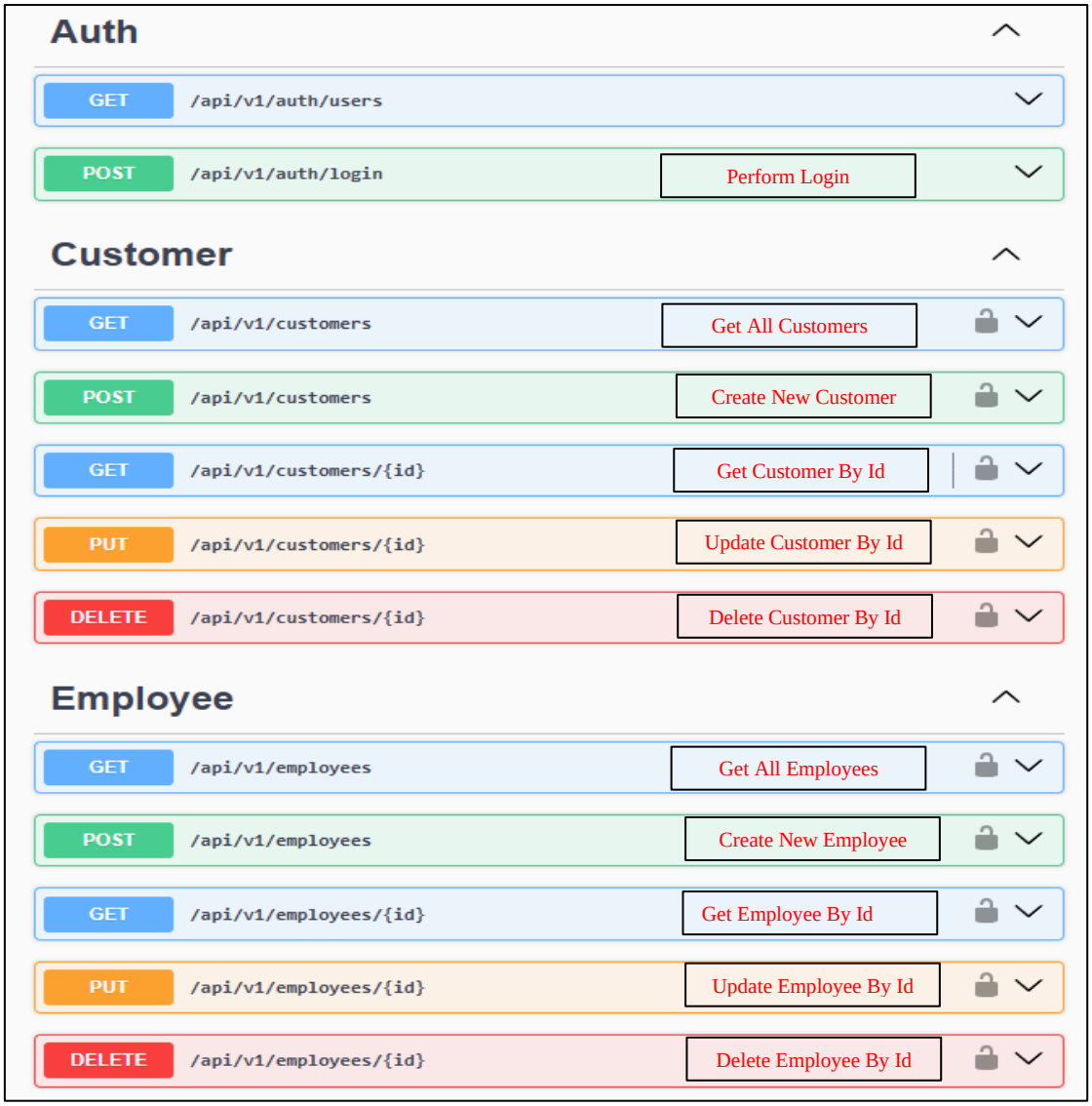


Figure 4.1. *REST API Endpoints for Users*

The second figure presents the service request endpoints. In addition to the standard CRUD operations, it also includes methods used to update the status of a service request, such as assign Technician, start working, Cancelling…

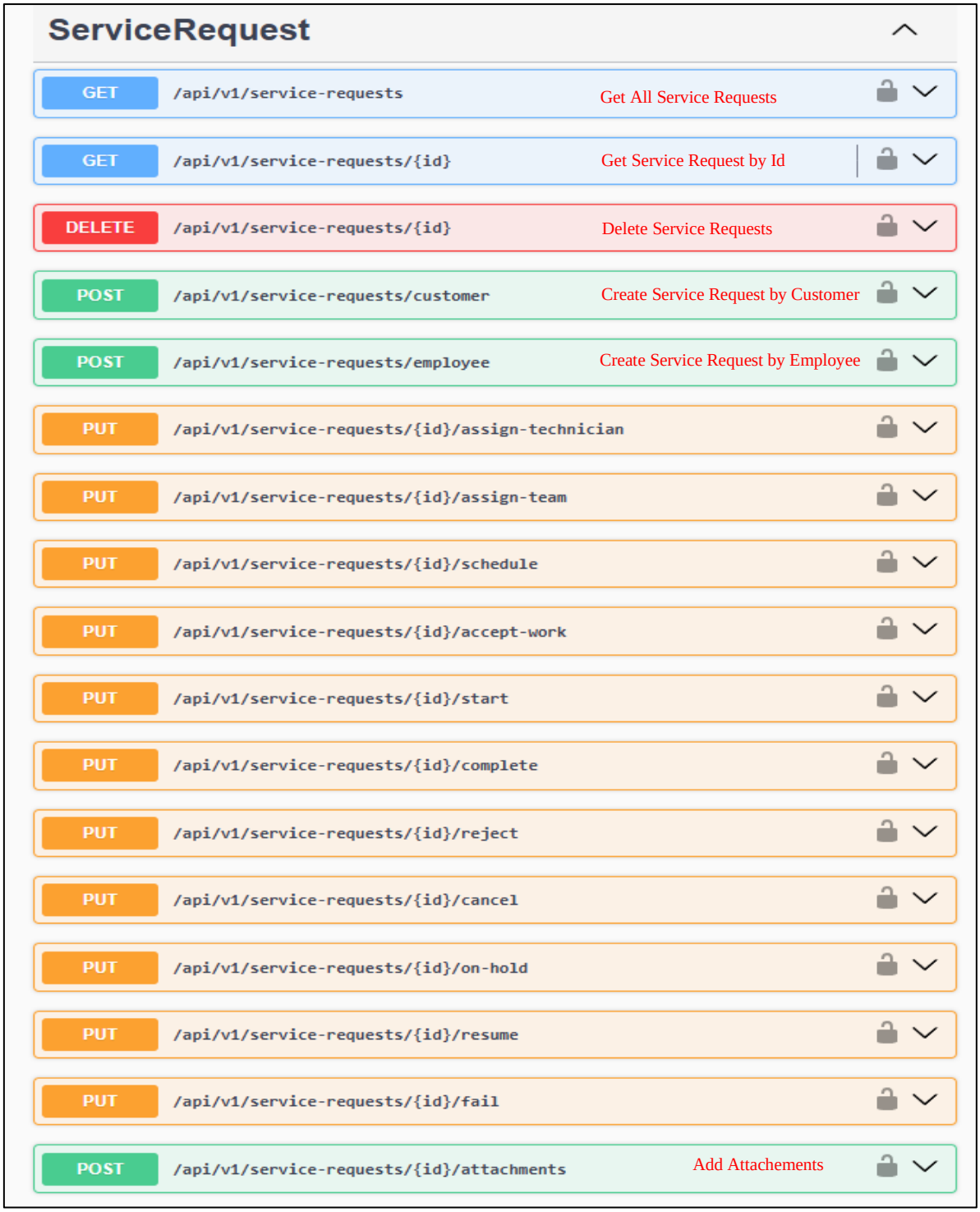


Figure 4.2. *REST API Endpoints for service request*

Moreover, all **GET endpoints** are designed to support dynamic filtering through query parameters. These parameters enable **filtering**, **pagination**, and **sorting** of the returned data, which improves the flexibility and usability of the API when handling large or complex datasets.

The figure below shows an example of this mechanism through the endpoint **/employees?pageSize=10&pageNumber=1&search[firstname]=Yacine**, where the returned employee records are filtered by first name and limited according to the specified pagination parameters.

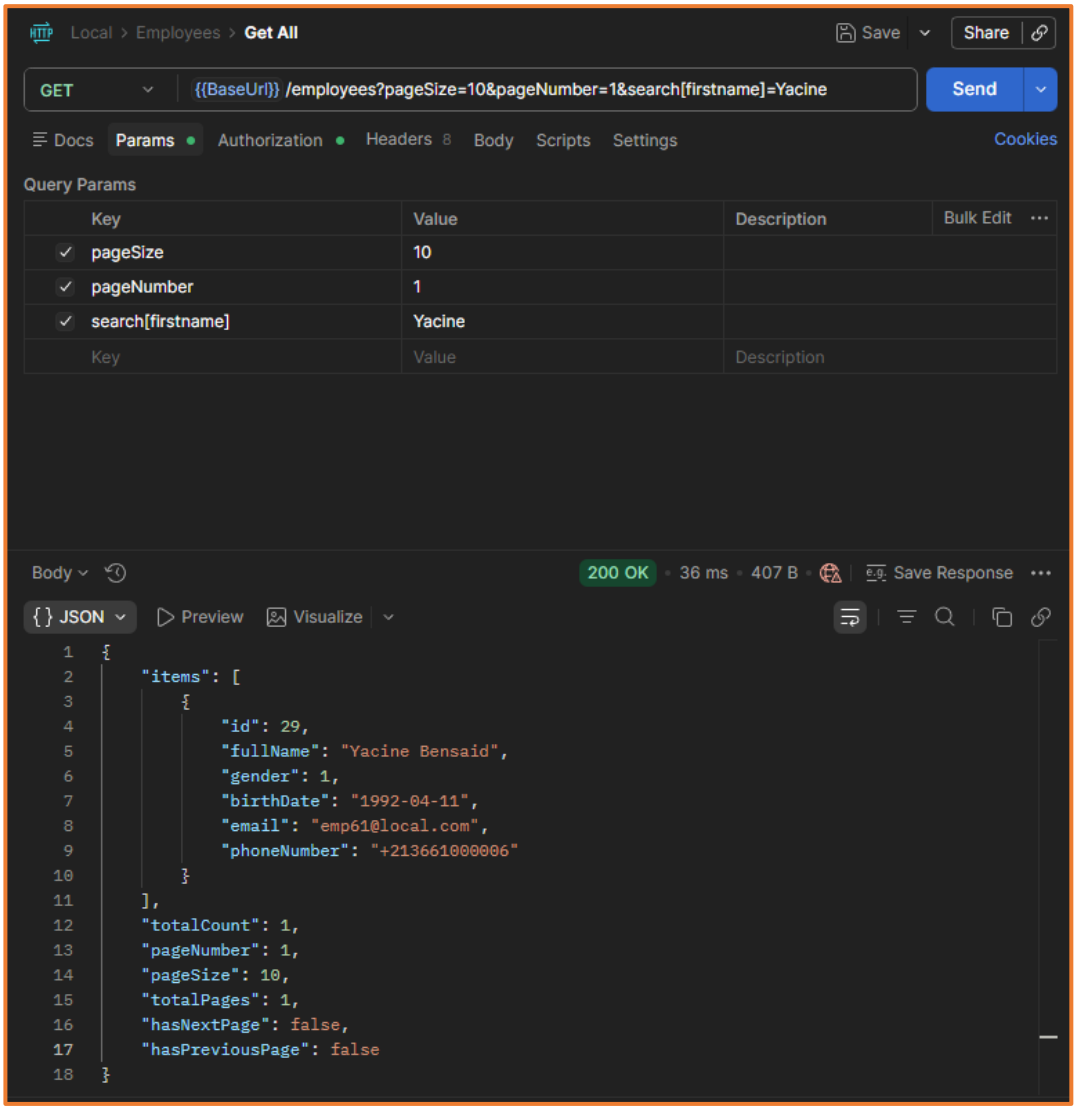
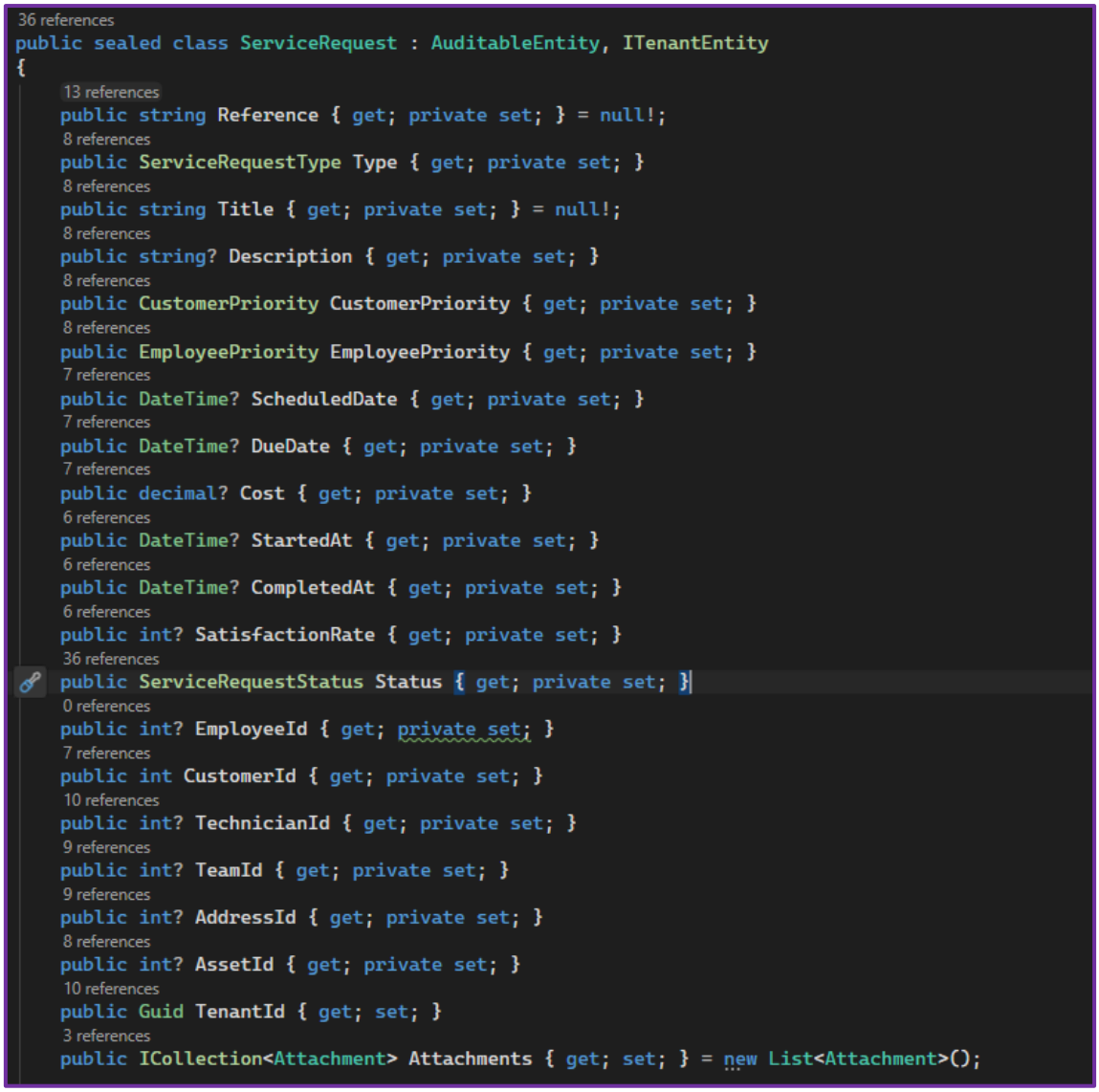


Figure 4.3.*GET Employee by Dynamic Search using postman*

Service Request Entity and Status Transition Management

The figure below presents the Service Request entity, which defines the main data structure used to represent a service request in the system. It includes essential attributes such as the reference, type, title, description, priorities, scheduled and due dates...
These fields allow the system to store and manage all the information required to follow a service request from creation to completion.



```
36 references
public sealed class ServiceRequest : AuditableEntity, ITenantEntity
{
    13 references
    public string Reference { get; private set; } = null!;
    8 references
    public ServiceRequestType Type { get; private set; }
    8 references
    public string Title { get; private set; } = null!;
    8 references
    public string? Description { get; private set; }
    8 references
    public CustomerPriority CustomerPriority { get; private set; }
    8 references
    public EmployeePriority EmployeePriority { get; private set; }
    7 references
    public DateTime? ScheduledDate { get; private set; }
    7 references
    public DateTime? DueDate { get; private set; }
    7 references
    public decimal? Cost { get; private set; }
    6 references
    public DateTime? StartedAt { get; private set; }
    6 references
    public DateTime? CompletedAt { get; private set; }
    6 references
    public int? SatisfactionRate { get; private set; }
    36 references
    public ServiceRequestStatus Status { get; private set; }
    0 references
    public int? EmployeeId { get; private set; }
    7 references
    public int CustomerId { get; private set; }
    10 references
    public int? TechnicianId { get; private set; }
    9 references
    public int? TeamId { get; private set; }
    9 references
    public int? AddressId { get; private set; }
    8 references
    public int? AssetId { get; private set; }
    10 references
    public Guid TenantId { get; set; }
    3 references
    public ICollection<Attachment> Attachments { get; set; } = new List<Attachment>();
```

Figure 4.4. *Service Request Entity*

The second figure shows the methods used to change the status of a service request. Before updating the status, each method calls the transition logic to verify whether the change is valid. Once the transition is accepted, the status is updated and the related data, such as the start date, completion date, cost, satisfaction rate, or cancellation reason, is added according to the operation being performed

```
public Result<Updated> Start(Roles role)
{
    var transition = ServiceRequestStatusTransition.Change(Status, ServiceRequestStatus.InProgress, role);
    if (transition.IsError)
        return transition.Errors;

    Status = transition.Value;
    StartedAt = DateTime.UtcNow;
    AddStatusHistory(Status);

    return Result.Updated;
}

3 references
public Result<Updated> Complete(Roles role, decimal? cost, int? satisfactionRate)
{
    if (satisfactionRate is < 0 or > 5)
        return ServiceRequestErrors.InvalidSatisfactionRate;

    var transition = ServiceRequestStatusTransition.Change(Status, ServiceRequestStatus.Completed, role);
    if (transition.IsError)
        return transition.Errors;

    Status = transition.Value;
    CompletedAt = DateTime.UtcNow;
    Cost = cost;
    SatisfactionRate = satisfactionRate;

    AddStatusHistory(Status);

    return Result.Updated;
}

2 references
public Result<Updated> Cancel(Roles role, string reason)
{
    var transition = ServiceRequestStatusTransition.Change(Status, ServiceRequestStatus.Cancelled, role);
    if (transition.IsError)
        return transition.Errors;

    Status = transition.Value;
    AddStatusHistory(Status, reason);

    return Result.Updated;
}
```

Figure 4.5. *The methods used to change the status of a service request*

The third figure presents the **status transition class**, which contains the transition rules that define which status changes are allowed and which roles are authorized to perform them. This class ensures that only correct transitions are accepted, preserving the consistency of the service request workflow.

```
public static class ServiceRequestStatusTransition
{
    private static readonly HashSet<Roles> PrivilegedRoles = [Roles.TenantAdmin, Roles.Manager];

    private static readonly Dictionary<(ServiceRequestStatus, ServiceRequestStatus), HashSet<Roles>> Transitions = new()
    {
        [(ServiceRequestStatus.New, ServiceRequestStatus.Scheduled)] = [Roles.Employee],
        [(ServiceRequestStatus.New, ServiceRequestStatus.Rejected)] = [Roles.Employee],
        [(ServiceRequestStatus.New, ServiceRequestStatus.Cancelled)] = [Roles.Customer, Roles.Employee],

        [(ServiceRequestStatus.Scheduled, ServiceRequestStatus.WorkerAccepted)] = [Roles.Technician],
        [(ServiceRequestStatus.Scheduled, ServiceRequestStatus.Rejected)] = [Roles.Technician],
        [(ServiceRequestStatus.Scheduled, ServiceRequestStatus.Cancelled)] = [Roles.Customer, Roles.Employee],

        [(ServiceRequestStatus.WorkerAccepted, ServiceRequestStatus.InProgress)] = [Roles.Technician],
        [(ServiceRequestStatus.WorkerAccepted, ServiceRequestStatus.Rejected)] = [Roles.Technician],
        [(ServiceRequestStatus.WorkerAccepted, ServiceRequestStatus.Cancelled)] = [Roles.Customer],

        [(ServiceRequestStatus.InProgress, ServiceRequestStatus.OnHold)] = [Roles.Technician],
        [(ServiceRequestStatus.InProgress, ServiceRequestStatus.Completed)] = [Roles.Technician],

        [(ServiceRequestStatus.OnHold, ServiceRequestStatus.InProgress)] = [Roles.Technician],
        [(ServiceRequestStatus.OnHold, ServiceRequestStatus.Failed)] = [Roles.Technician],

        [(ServiceRequestStatus.Rejected, ServiceRequestStatus.Cancelled)] = [Roles.Employee, Roles.Customer],
    };

    9 references
    public static Result<ServiceRequestStatus> Change(ServiceRequestStatus current, ServiceRequestStatus next, Roles role)
    {
        if (current == next)
            return current; // idempotent

        if (!CanChange(current, next, role))
            return ServiceRequestErrors.InvalidStatusTransitionF(current, next, role);

        return next;
    }

    1 reference
    public static bool CanChange(ServiceRequestStatus current, ServiceRequestStatus next, Roles role)
    {
        if (PrivilegedRoles.Contains(role))
            return true;

        return Transitions.TryGetValue((current, next), out var roles)
               && roles.Contains(role);
    }
}
```

Figure 4.6. *Status Transition class*

4.3.2 Web Platform Presentation

This section presents the main interfaces of the web platform. In order to respect the principle of role-based access, the system offers a distinct workspace for each category of user. The following figures illustrate, for every profile, the navigation side bar and the principal screens to which the user has access.

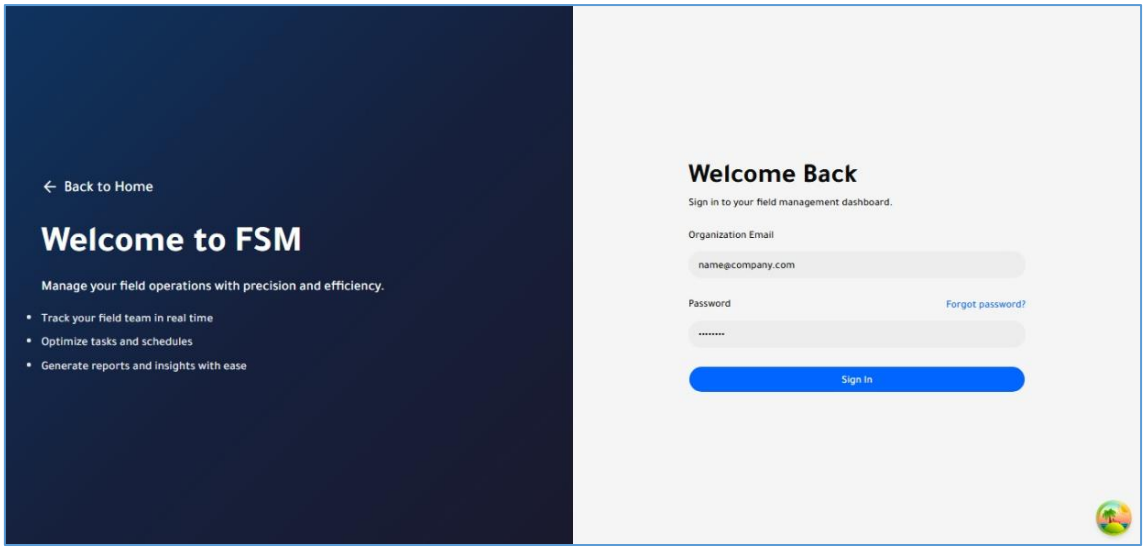


Figure 4.7. *Web Platform Authentication Interface*

The figure above presents the authentication interface of the web platform, which constitutes the common entry point for every category of user. The screen is divided into two parts: a presentation panel on the left that introduces the system and summarizes its main benefits, and a sign-in form on the right in which the user enters the organization e-mail address and the password. Once the credentials are validated, the user is redirected to the workspace that corresponds to his or her role.

- **System Administrator Interfaces**

The System Administrator is responsible for the management of the platform as a whole and, more particularly, for the supervision of the different organizations, called tenants.

a) Side Bar

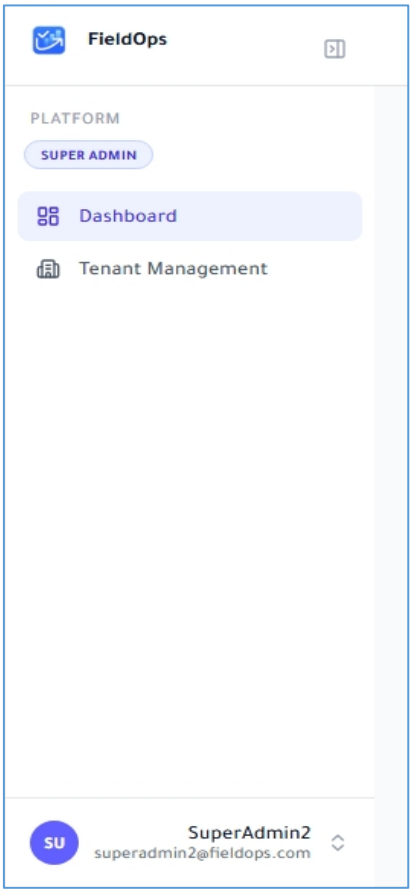


Figure 4.8. *System Administrator sidebar*

The (Figure 4.8) shows the navigation side bar associated with the System Administrator role. It contains only two main entries, the Dashboard and the Tenant Management module, which reflects the limited but strategic scope of this profile. The lower part of the bar displays the identity of the connected administrator.

b) Dashboard

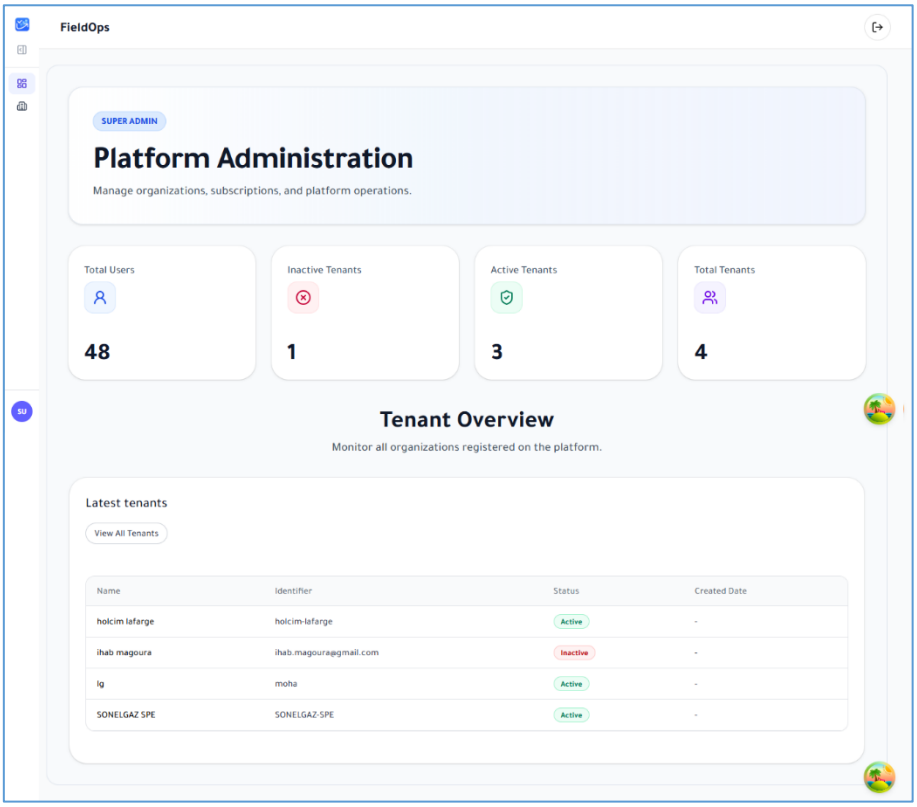


Figure 4.9. *System Administrator Dashboard Interface*

(Figure 4.9) illustrates the dashboard of the System Administrator. The upper section provides summary indicators concerning the whole platform, such as the total number of users and the number of active and inactive tenants. The lower section, entitled Tenant Overview, lists the most recently registered organizations together with their status, which gives the administrator a global and immediate view of the system.

c) Tenant Management

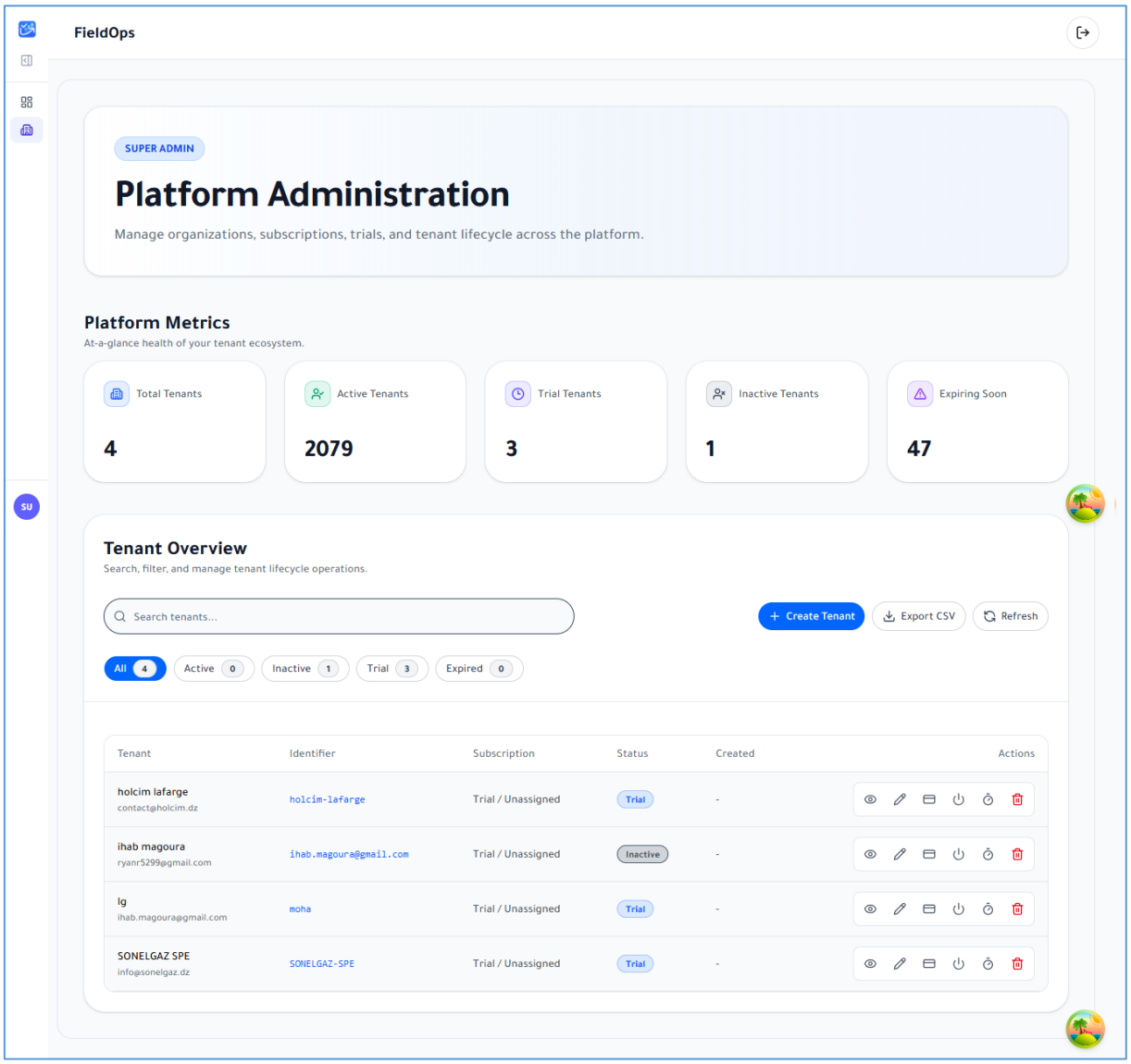


Figure 4.10. *Tenant Management interface*

The figure above presents the Tenant Management interface, through which the System Administrator supervises the life cycle of all the organizations. A set of metrics summarizes the number of tenants according to their state (active, in trial, inactive or expiring soon), while the table below lists each tenant with its identifier, subscription type and status. The action icons on the right allow the administrator to view, modify, activate, deactivate or delete a tenant.

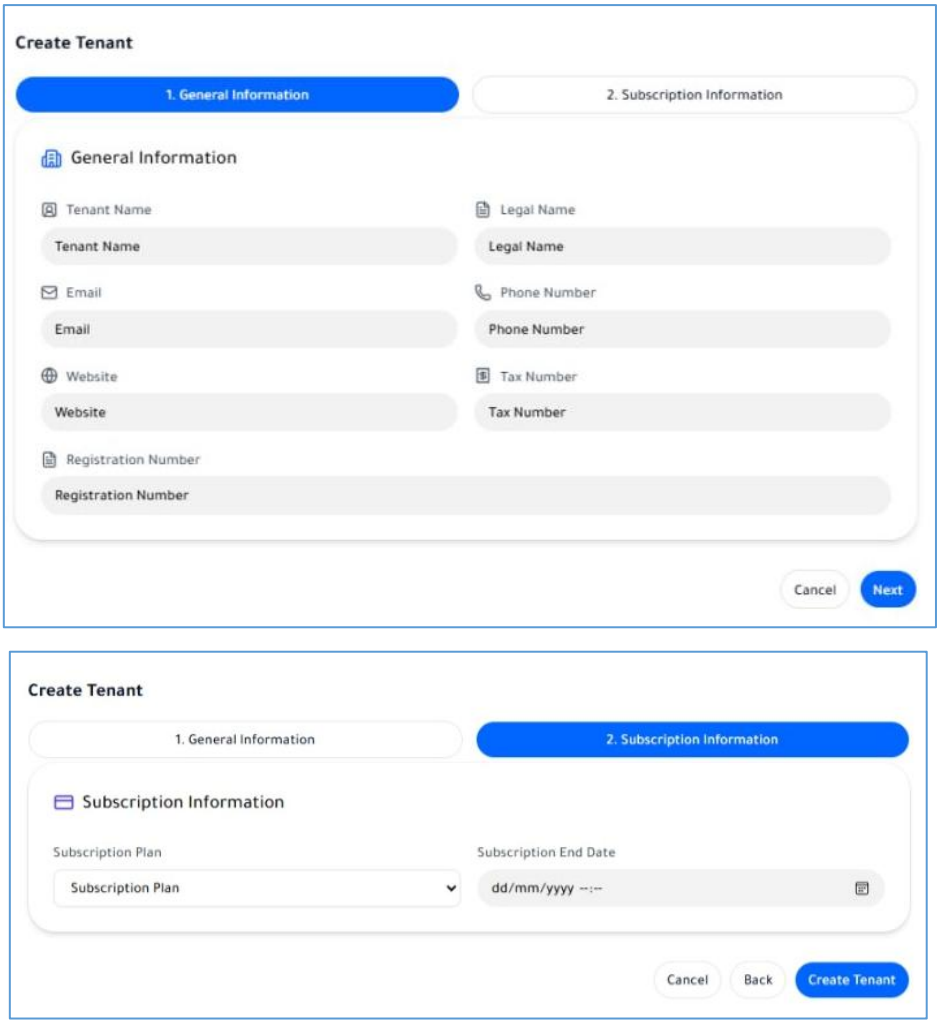


Figure 4.11. *Create New Tenant Form*

The (Figure 4.11) show the form used to create a new tenant. In the first step, which is devoted to general information, the administrator fills in the main attributes of the organization, namely its name, legal name, e-mail, telephone number, web site, tax number and registration number. In the second step, which concerns the subscription information, the administrator selects the subscription plan and specifies its end date, after which the new tenant is created and added to the platform.

- **Tenant Administrator Interfaces**

The Tenant Administrator manages a single organization and has access to all of its resources, including the users, the teams, the requests and the reports.

a) Side Bar

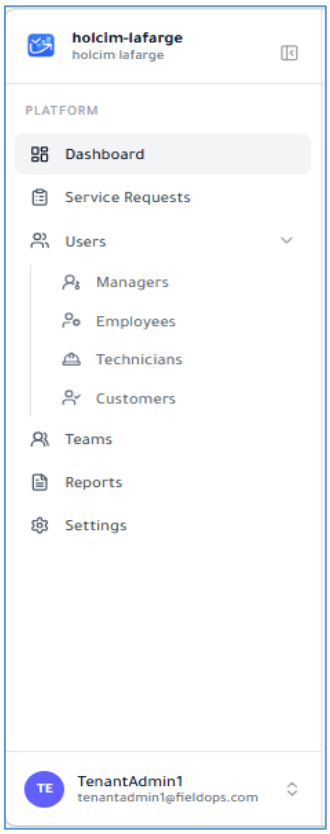


Figure 4.12. *Tenant Administrator Sidebar*

The (Figure 4.12) presents the side bar of the Tenant Administrator. Compared with the System Administrator, this profile has a much wider scope: in addition to the Dashboard and the Service Requests, it provides access to the management of users (managers, employees, technicians and customers), to the teams, to the reports and to the general settings of the organization.

b) Dashboard

The (Figure 4.13) illustrates the dashboard of the Tenant Administrator. It gathers the key performance indicators of the organization, in particular the total number of requests, the open and completed requests and the average resolution time. A bar chart represents the evolution of the requests over the last months, while a side panel reports the most recent activities and the most productive technician of the period.

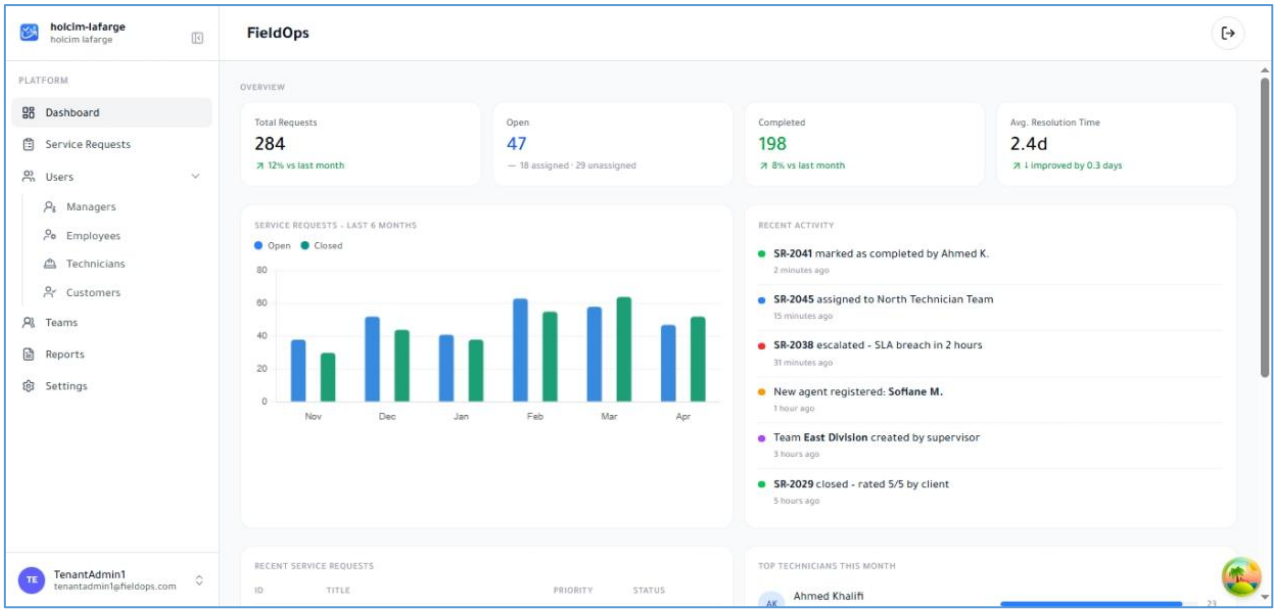


Figure 4.13. *Tenant Administrator Dashboard*

- **Manager Interfaces**

The Manager ensures the day-to-day operational follow-up of the organization, mainly the handling of service requests and the management of the employees, the technicians and the teams.

a) Side Bar

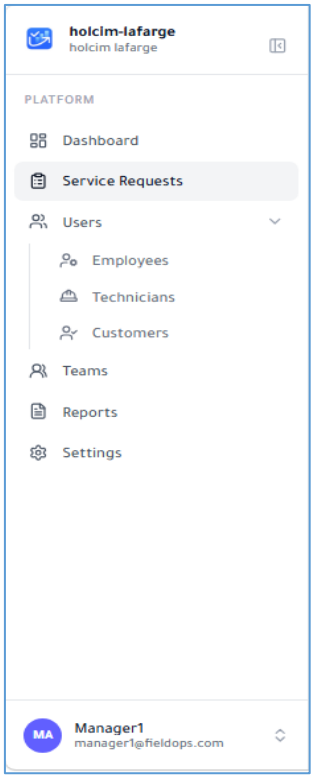


Figure 4.14. *Manager Sidebar*

(Figure 4.14) shows the navigation side bar of the Manager. Its structure is similar to that of the Tenant Administrator, except that the management of other managers is no longer available, since the Manager operates at an intermediate level of responsibility. The remaining entries give access to the service requests, to the employees, technicians and customers, and to the teams and reports.

b) Service Request

(Figure 4.15) presents the Service Requests interface as displayed for the Manager. Summary counters at the top indicate the total number of requests and their distribution by state. The table lists each request with its reference, title, client, priority, status, due date, cost and assignees, which enables the Manager to monitor and follow the whole set of interventions.

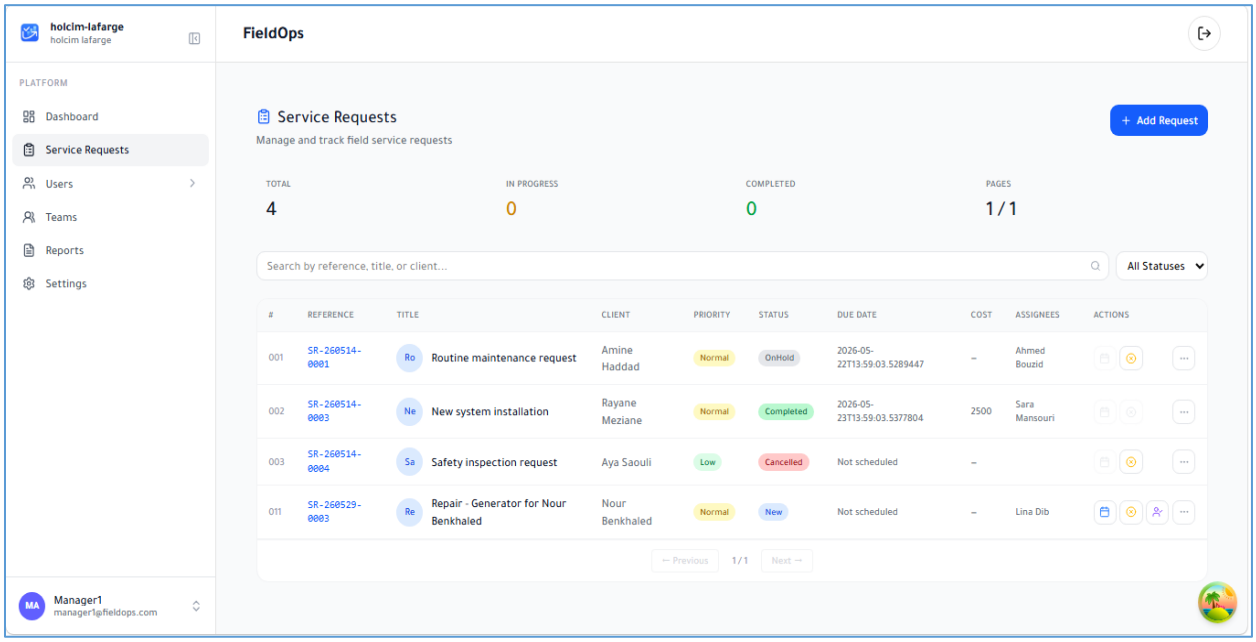


Figure 4.15. *Service Requests Interface*

The figures above concern the management of service requests by the Manager.

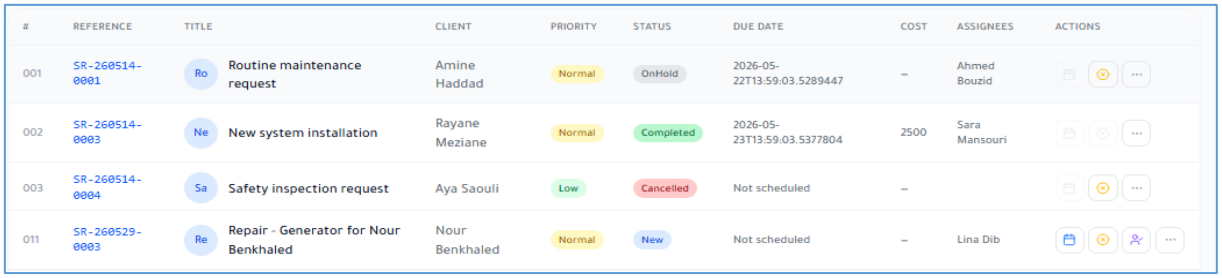


Figure 4.16. *Service Request Items*

(Figure 4.16) provides a closer view of the requests table, in which color-coded labels express the priority and the status of each request.

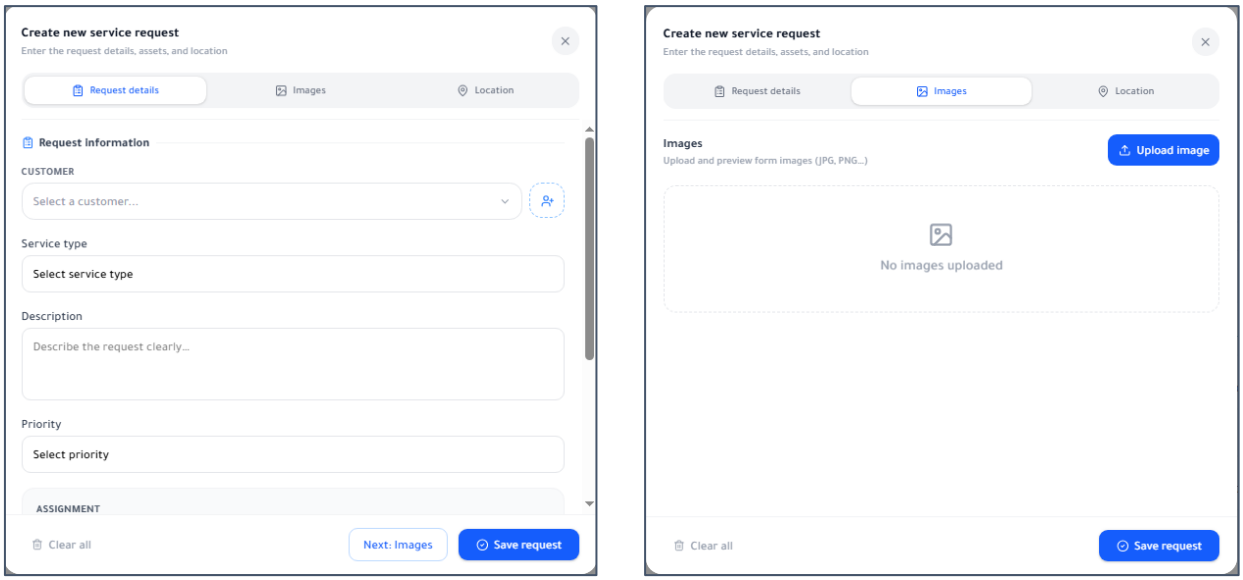


Figure 4.17. *First Taps of Creating New Request*

(Figure 4.17) present the form used to create a new request: the first tab gathers the request details, namely the customer, the type of service, the description and the priority, while the second tab allows images to be attached in order to document the reported problem.

The figures above complete the creation of a service request and illustrate the actions available on an existing one.

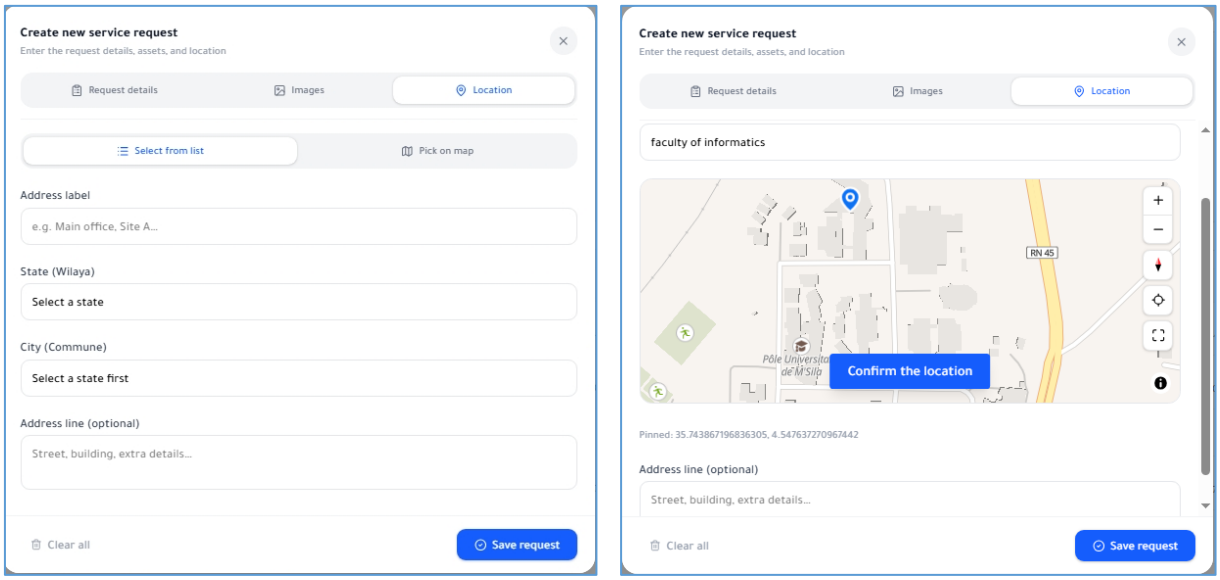


Figure 4.18. *Last Tab of creating new request*

(Figure 4.18) show the location tab: the place of the intervention can be entered manually by choosing the state and the commune, **or** selected directly on an interactive map and then confirmed.

(Figure 4.19) presents the action panel that appears when the Manager opens a service request with the status **New**. Since the request is at its initial stage and has not yet been processed, all possible actions are displayed simultaneously, giving the Manager complete visibility over what can be done at this point.

The available actions are:

- Assign – assign the request to an available employee.
- Schedule – define the priority, start date and due date.
- Reject – reject the request, with the option to reassign or close it.
- Cancel – cancel the request entirely.
- Consult Details – view the full information of the request.

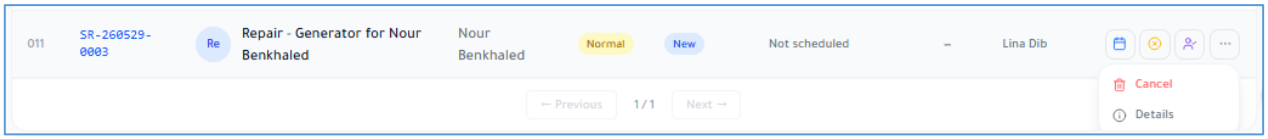


Figure 4.19. *New Service request*

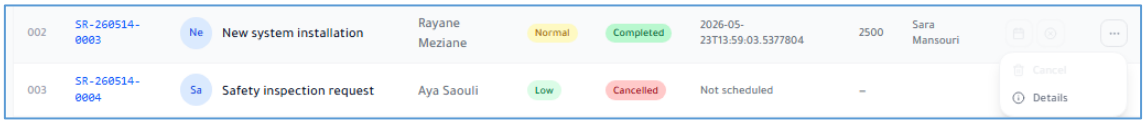


Figure 4.20. *Completed Service request*

(Figure 4.20) presents the action panel on a service request whose status is **Completed**. In contrast to (Figure 4.19), where the new status exposed the full set of available actions, this figure illustrates the opposite situation: since the request has already been completed and is no longer active, the menu is reduced to its minimum, displaying only the Details action, which allows the Manager to consult the information of the request in read-only mode. This confirms that the action panel is dynamic and status-driven the actions it proposes are not fixed but adapt automatically to the current state of each request, ensuring that the Manager can only perform operations that are logically valid at each stage of the workflow.

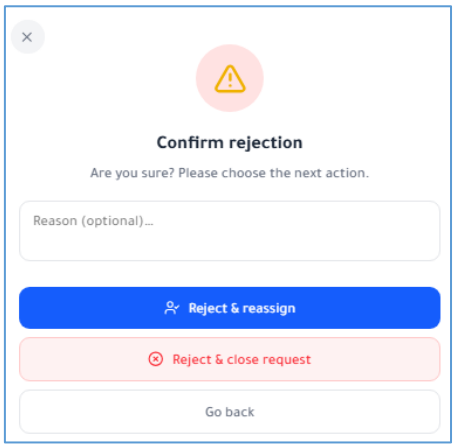


Figure 4.21. *Rejection Window*

(Figure 4.21) is the rejection window, in which the Manager chooses between rejecting and reassigning the request or rejecting and closing it.

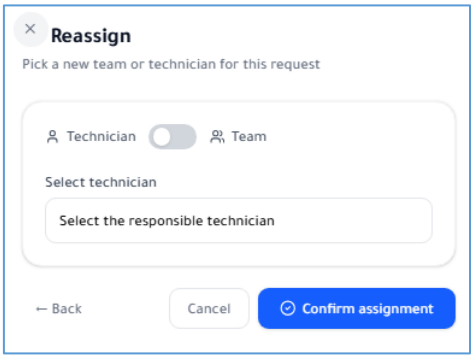


Figure 4.22. *Re-assignment Window*

(Figure 4.22) is the reassignment window, used to select a new technician or a new team.

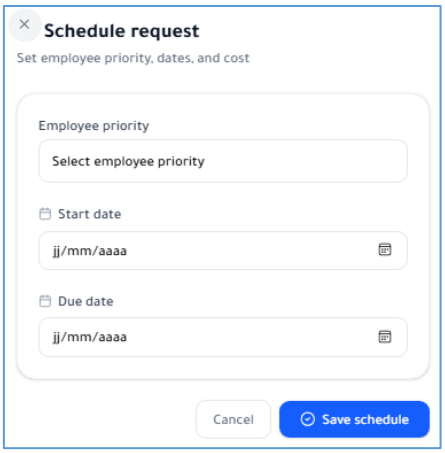


Figure 4.23. *Scheduling Window*

(Figure 4.23) is the scheduling window, in which the priority of the employee as well as the start and due dates are defined.

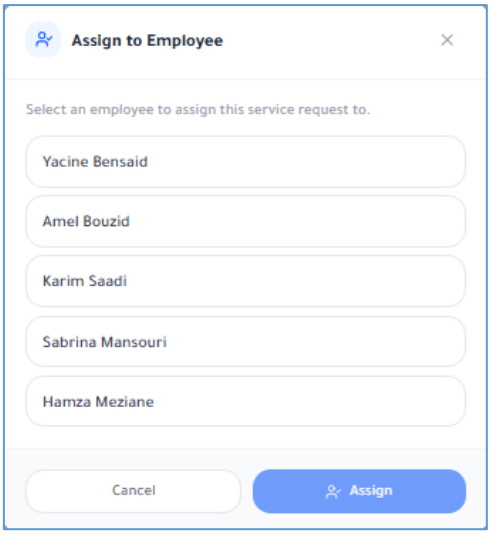


Figure 4.24. *Assignment service request to employee window*

The figure above presents the window used to assign a request to an employee. A list of the available employees is displayed and the Manager selects the most appropriate person to carry out the intervention.

c) Users (Employee)

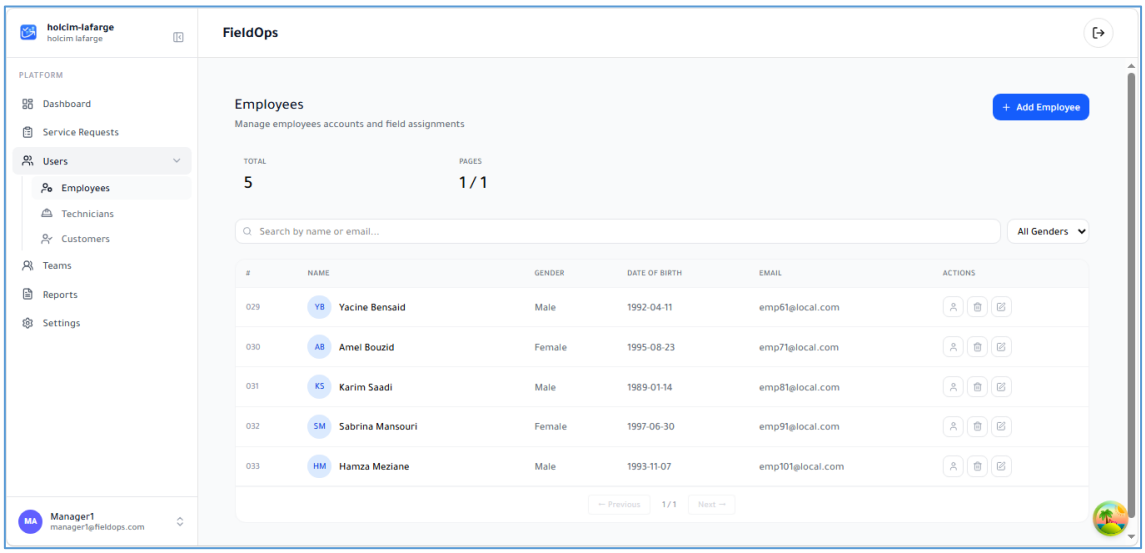


Figure 4.25. *Users Management interface (Employees)*

The figure above shows the list of employees managed by the Manager. Each line presents the main information of an employee, namely the name, the gender, the date of birth and the e-mail address, together with a set of actions. A search field and a filter facilitate the location of a particular employee.

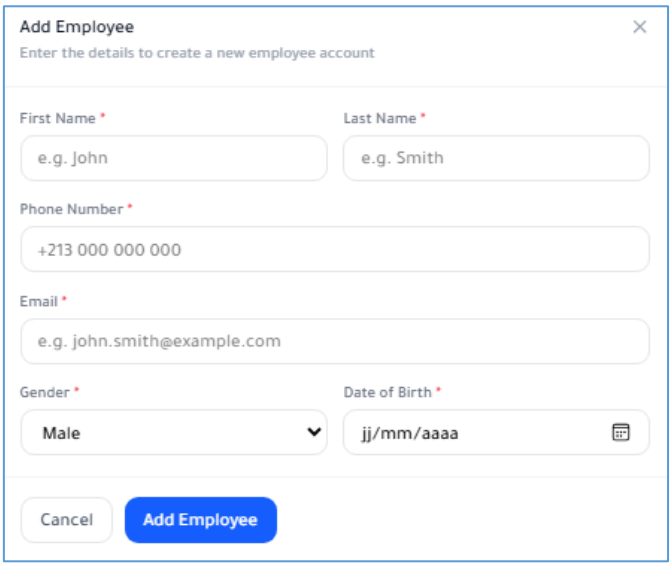


Figure 4 26. *Create New Employee Account Form*

The figure above presents the form used to create a new employee account. The Manager enters the first name, the last name, the telephone number, the e-mail address, the gender and the date of birth, the required fields being clearly indicated.

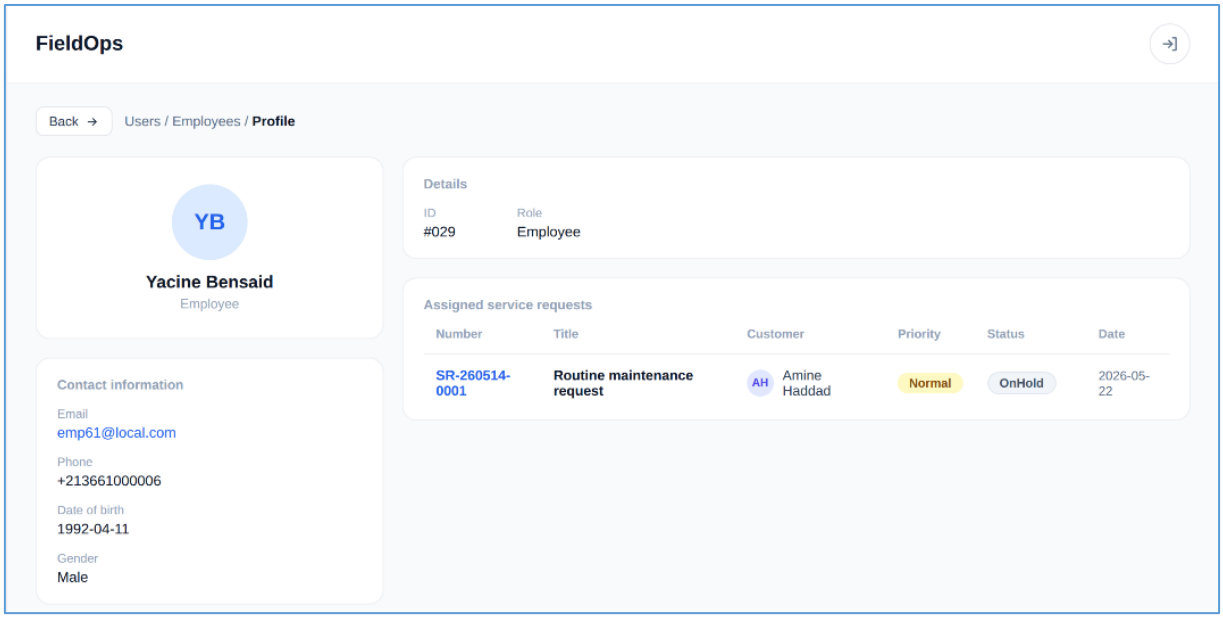


Figure 4.27. *Employee Profile interface*

The figure above illustrates the individual profile of an employee. The left part gathers the personal and contact information, while the right part displays the identification details and, above all, the list of the service requests assigned to this employee with their reference, title, customer, priority, status and date.

- **Employee Interfaces**

The Employee occupies an operational position and intervenes directly in the processing of the service requests that are assigned to him or her.

a) Side Bar

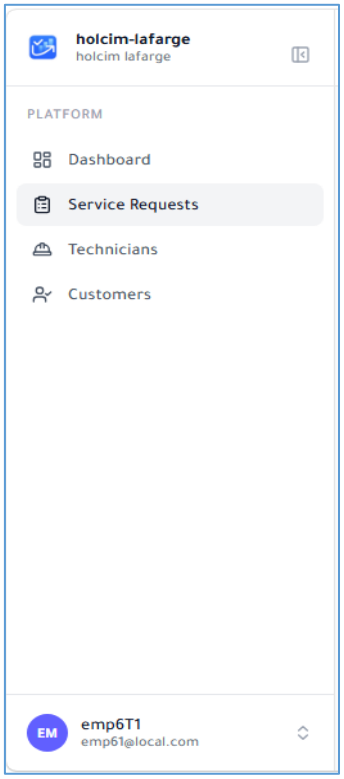


Figure 4.28. *Employee Sidebar*

The (Figure 4.28) shows the navigation side bar of the Employee. Its scope is deliberately restricted: the employee only has access to the dashboard, to the service requests and to the consultation of the technicians and customers, in accordance with his or her operational role.

b) Service request

The (Figure 4.29) presents the service-requests interface as seen by the employee. The requests are displayed with their reference, title, client, priority and status, and an indication specifies whether each request is already assigned to the employee or still available.

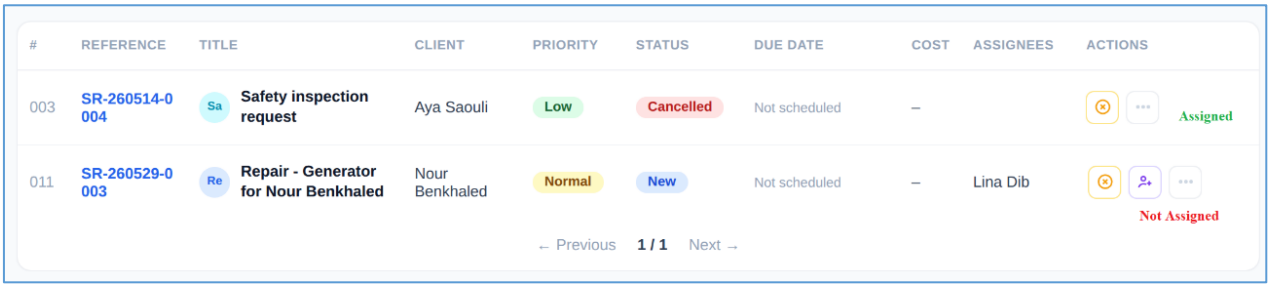


Figure 4.29. *Employee service request interface*

The figure above shows the window through which an employee assigns a request to himself or herself. By confirming this action, the employee becomes responsible for completing the corresponding service request.

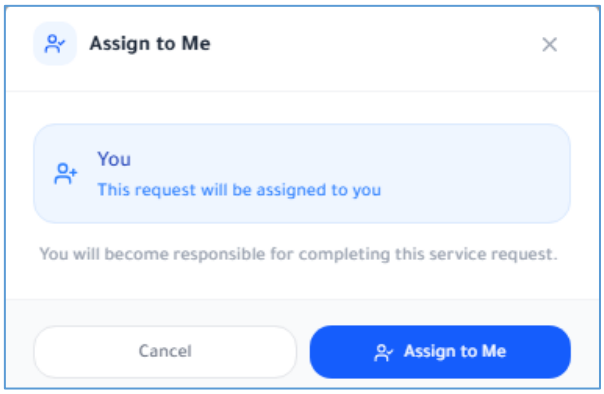


Figure 4.30. *Self-Assignment Window*

4.3.3 Mobile Application Presentation

This section presents the main interfaces of the mobile application, which is intended for the two profiles that operate directly in the field, namely the customer and the technician. In accordance with the principle of role-based access, the application offers a distinct workspace to each of them. The following figures illustrate, for every profile, the principal screens to which the user has access.

The two figures above present the common entry interfaces of the mobile application.

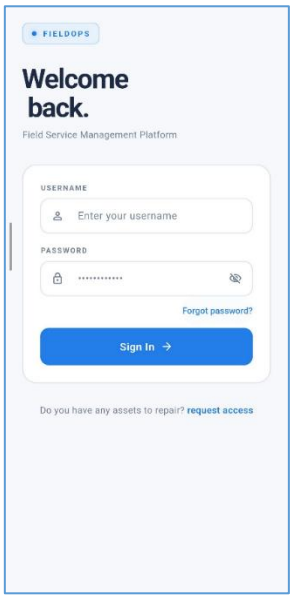


Figure 4.32. *Authentication interface of the mobile application*

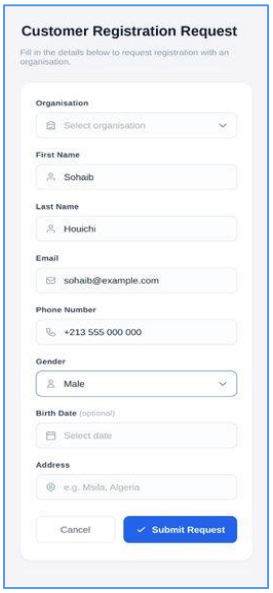


Figure 4.31.*Customer Registration Request Form*

(Figure 4.31) is the authentication screen, which constitutes the shared access point for every user: the user enters the e-mail address and the password and, once the credentials are validated, is redirected to the workspace that corresponds to his or her role.

A link placed beneath the form gives a new user access to the second screen, namely the customer registration-request form (Figure 4.32), through which a person who does not yet belong to an organization asks to join one by selecting the target organization and by providing the personal information, namely the first name, e-mail address, the telephone number, the gender, the date of birth and the address; the request is then submitted for validation.

- **Customer Interfaces**

a) Dashboard

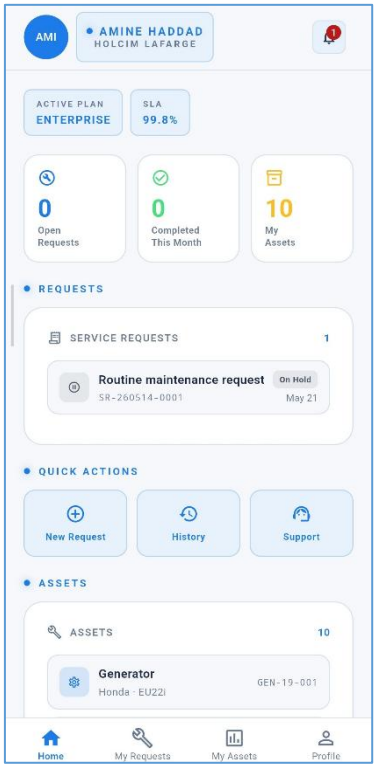


Figure 4.33. *Customer Dashboard interface*

The figure above illustrates the dashboard of the customer. The upper part recalls the subscription plan of the customer and the associated service-level indicator, then summarizes the activity by means of three counters: the number of open requests, the number of requests completed during the month and the number of registered assets. The central part, entitled Requests, displays the most recent service requests with their reference and status, while a set of quick actions allows a new request to be created or the history and the support to be consulted. The lower part, entitled Assets, lists the equipment belonging to the customer. A navigation bar at the bottom of the screen gives access to the home page, the requests, the assets and the profile.

b) Service Requests

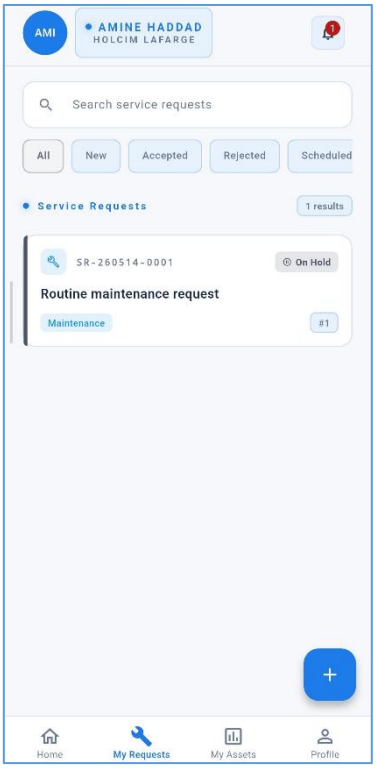


Figure 4.34. *List of the customer's service requests*

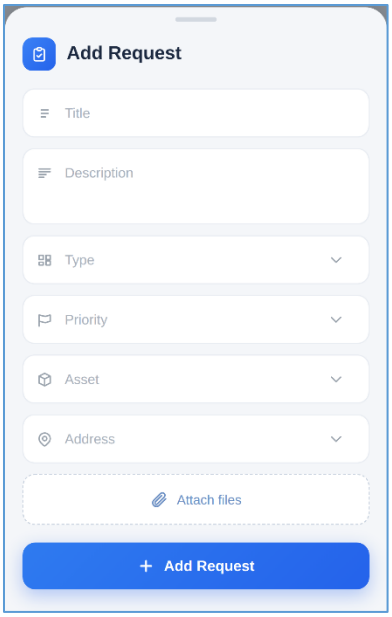


Figure 4.35. *Form for creating a service request (customer)*

The two figures above concern the management of the service requests by the customer. (Figure 4.35) presents the form used to create a new request, in which the customer fills in the title, the description, the type of service, the priority and the asset concerned, indicates the address of the intervention and, if necessary, attaches one or more files in order to document the problem. (Figure 4.34) shows the list of the requests submitted by the customer: a search field and a series of filters by status facilitate navigation, and each request is displayed with its reference, its title, its status and its priority.

c) Assets

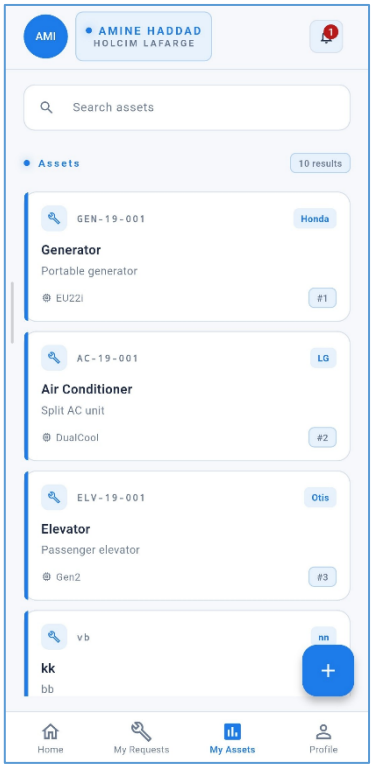


Figure 4.36. *List of the customer's assets*

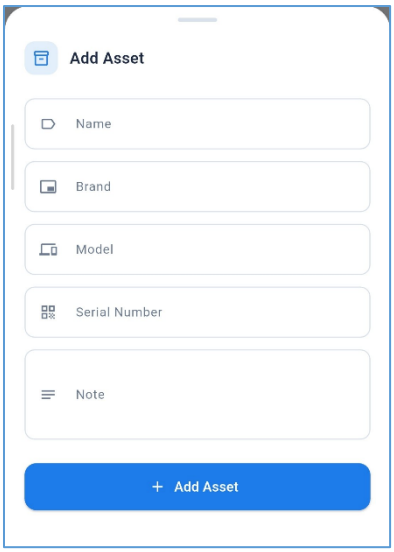


Figure 4.37. *Form for adding an asset*

(Figure 4.36) presents the form through which the customer declares a new asset by entering its name, its brand, its model, its serial number and a free note. (Figure 4.37) displays the list of the registered assets; each line shows the main characteristics of an asset, namely its reference, its designation and a short description, while a search field allows a particular item of equipment to be located.

d) Profile

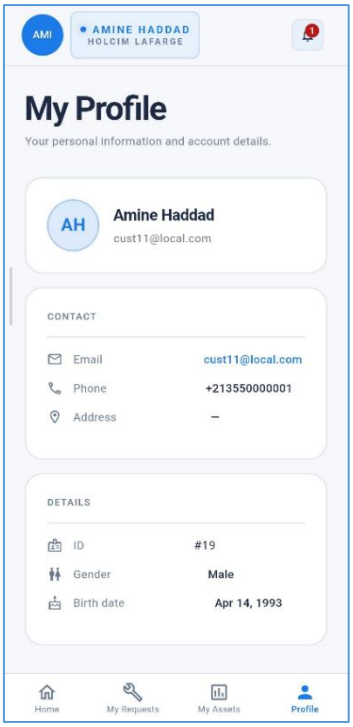


Figure 4.38. *Customer Profile Interface*

The figure above illustrates the profile of the customer. The upper part recalls the identity of the connected user, the central part gathers the contact information, namely the e-mail address, the telephone number and the address, and the lower part displays the identification details, namely the identifier, the gender and the date of birth.

- **Technician Interfaces**

a) Dashboard

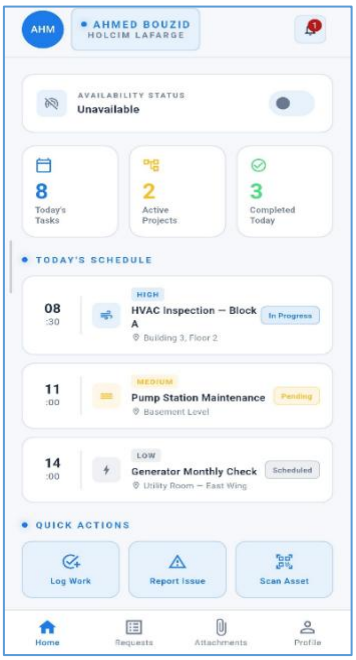


Figure 4.39. *Technician Dashboard Interface*

The figure above illustrates the dashboard of the technician. At the top, an availability indicator allows the technician to declare whether he or she is available. Three counters then summaries the daily workload, namely the number of tasks scheduled for the day, the number of active projects and the number of interventions completed during the day. The central part, entitled Today's Schedule, lists the interventions planned for the day with their time, their title, their location and their status, which may be in progress, pending or scheduled, while a set of quick actions allows work to be logged, an issue to be reported or an asset to be scanned.

b) Service Requests

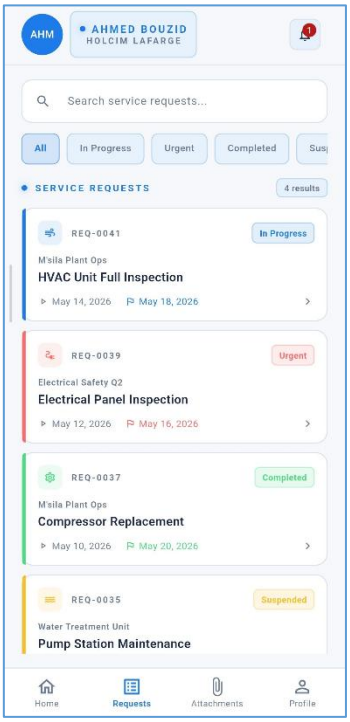


Figure 4.40. *List of the technician's service requests*

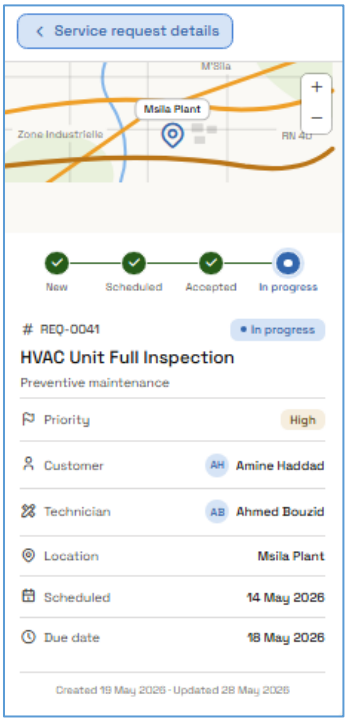


Figure 4.41. *Detailed view of a service request (technician)*

The two figures above present the service requests as seen by the technician. (Figure 4.40) displays the list of the requests assigned to the technician; a search field and several filters by status facilitate consultation, and each request is shown with its reference, its title, its status and its priority. (Figure 4.41) shows the detailed view of a request: a map locates the place of the intervention, a progress bar indicates the stage that has been reached, from new to scheduled, accepted and in progress, and a descriptive sheet gathers the main information, namely the priority, the customer, the assigned technician, the location, the scheduled date and the due date.

c) Attachments

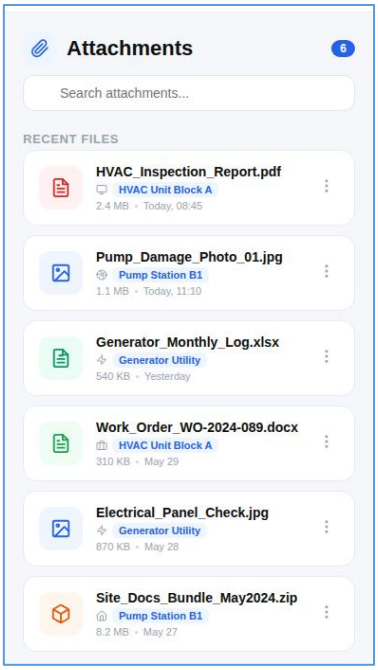


Figure 4.42. *List of attachments*

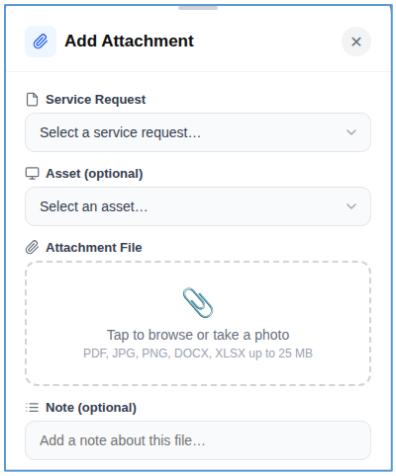


Figure 4.43. *Form for adding an attachment*

The two figures above concern the management of the attachments. (Figure 4.42) presents the form used to add a new document: the technician associates the file with a service request and, where appropriate, with an asset, then selects or photographs the document to be attached and may add an explanatory note. (Figure 4.43) shows the list of the recent files, each one being displayed with its name, the request or the asset to which it is linked, its size and its date; the supported formats include, in particular, document, image, spreadsheet and archive files.

d) Profile

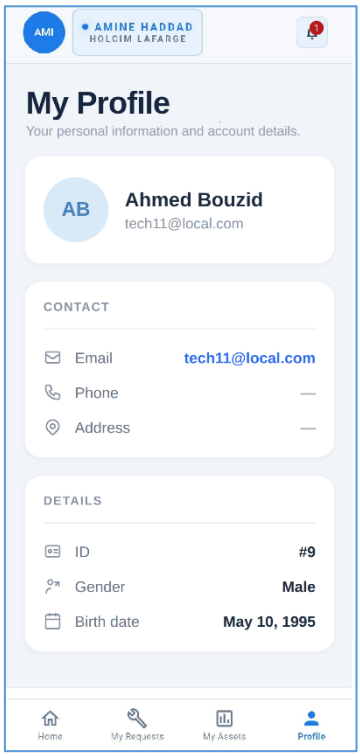


Figure 4.44. *Technician Profile Interface*

The figure above illustrates the profile of the technician. As for the customer, the upper part recalls the identity of the connected user, the central part gathers the contact information and the lower part displays the identification details, namely the identifier, the gender and the date of birth.

- **Tenant Information interface**

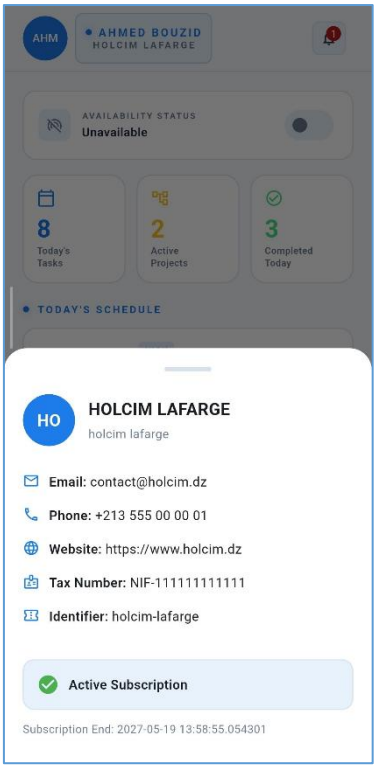


Figure 4.45. *Tenant-information Interface*

The figure above presents the tenant-information interface, which is displayed in the form of a panel that opens over the workspace of the technician. It summarizes the data of the organization to which the user belongs, namely its name, its e-mail address, its telephone number, its web site, its tax number and its identifier. The lower part indicates the state of the subscription and its end date, which gives the user a clear view of the situation of his or her organization.

4.4 Conclusion

This chapter presented the implementation of the Multi-Tenant Field Service Management system, covering the development environment, the technology stack, and the main modules of the platform.

The backend was built with ASP.NET Core Web API and C#, using Entity Framework Core for database interaction and SQL Server for persistent data storage. The API endpoints were structured around the system's core entities users, tenants, and service requests and were documented through Swagger to ensure clarity and ease of testing. A key implementation detail was the status transition mechanism for service requests, which enforces workflow consistency by validating every state change against predefined rules and role-based permissions before it is applied.

The web frontend was developed with React and TypeScript, relying on Tailwind CSS for styling and React Query and Zustand for data and state management. The mobile application was built with Flutter and Dart, targeting customers and technicians who operate in the field. Both interfaces implement role-based access control, providing each user profile System Administrator, Tenant Administrator, Manager, Employee, Customer, and Technician with a tailored workspace limited to the operations relevant to their responsibilities.

Taken together, the implemented system demonstrates that the architectural decisions made during the design phase were successfully translated into a working application. The separation of concerns between backend and frontend layers, the use of a shared multi-tenant data model, and the dynamic, status-driven behavior of the service request workflow all contribute to a platform that is both functionally complete and technically sound. The following chapter will address the testing and validation of this implementation.

Conclusion

The management of field service operations is a critical activity for organizations whose work is carried out away from their main premises, such as maintenance, installation, repair, and inspection. As shown throughout this work, many organizations still rely on manual or disconnected methods, which lead to weak traceability of operations, limited real-time visibility, recurrent communication gaps, and fragmented data. At the same time, modern software is increasingly expected to serve several organizations from a single shared platform. This thesis was therefore guided by two questions: how to improve the management of field operations through an integrated digital platform, and how to design that platform as a multi-tenant system in which several companies can share the same application while keeping their data isolated and secure a need that is particularly significant in the Algerian context, where affordable and locally adapted FSM solutions remain scarce.

To answer these questions, the work was organized into four complementary stages. The first two chapters established the conceptual foundations: Chapter 1 examined the discipline of Field Service Management its actors, core processes, benefits, recurring operational challenges, and the limitations of existing commercial and open-source solutions while Chapter 2 studied multi-tenant architecture and cloud computing, comparing the main tenancy models and identifying the fundamental requirements of data isolation, scalability, security, maintainability, and customization.

Building on this foundation, Chapter 3 presented the analysis and design of the proposed system, named FieldOps. The actors, the functional and non-functional requirements, and the use-case, class, sequence, and state diagrams were defined, and a three-tier client–server architecture was adopted. A shared-database, shared-schema multi-tenancy model, based on a TenantId discriminator and reinforced by JWT-based access control, was selected as the best compromise between cost, scalability, and ease of maintenance. Chapter 4 then translated this design into a working system: an ASP.NET Core Web API backend with SQL Server and Entity Framework Core, a React/TypeScript web application for administrative and managerial users, and a Flutter/Dart mobile application for technicians and customers, deployed online through the MonsterASP.NET hosting service.

The resulting platform centralizes the complete service-request lifecycle from creation, scheduling, and assignment to execution, closure, and reporting within a single, role-aware environment accessible through both web and mobile interfaces. Above all, it demonstrates that multiple organizations can operate on the same instance while their data remain logically isolated, thereby meeting the two objectives set out at the beginning of this work.

Like any project carried out within limited time and resources, the proposed solution presents some limitations. The shared-schema model offers less physical isolation than a separate-database approach, even though this risk is mitigated by strict tenant filtering, role-based access control, and token validation. Moreover, the system was validated mainly through functional demonstration; it has not yet been subjected to large-scale performance, load, and security testing, nor to a full deployment with real companies under operational conditions.

These limitations open several directions for future work. The platform could be enriched with real-time notifications and GPS-based technician tracking, an offline mode for the mobile application, more advanced (possibly AI-assisted) scheduling and dispatching, and analytical dashboards for performance monitoring. From an architectural standpoint, a hybrid tenancy model and horizontal scaling could be introduced for larger tenants, accompanied by systematic performance and security evaluation. Finally, the integration of subscription billing and an Arabic user interface would further improve the platform's suitability for the local market. Beyond these extensions, this work shows that a modern, affordable, and scalable multi-tenant Field Service Management platform can be designed and implemented with widely available technologies, offering a solid basis for both practical adoption and further research.

References

[1] G. Owusu, P. O'Brien, J. McCall, et N. F. Doherty, Éd., *Transforming Field and Service Operations: Methodologies for Successful Technology-Driven Business Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-44970-3.

[2] J. A. Castillo-Salazar, D. Landa-Silva, et R. Qu, « Workforce scheduling and routing problems: literature survey and computational study », *Ann. Oper. Res.*, vol. 239, nº 1, p. 39-67, avr. 2016, doi: 10.1007/s10479-014-1687-2.

[3] M. Vössing et J. Kunze Von Bischhoffshausen, « Field Service Technician Management 4.0 », in *Operations Research Proceedings 2016*, A. Fink, A. Fügenschuh, et M. J. Geiger, Éd., in Operations Research Proceedings. , Cham: Springer International Publishing, 2018, p. 63-68. doi: 10.1007/978-3-319-55702-1_10.

[4] « Overview of Dynamics 365 Field Service ». [En ligne]. Disponible sur: https://learn.microsoft.com/en-us/dynamics365/field-service/overview

[5] SAP, « What is Field Service Management? » Consulté le: 4 mai 2026. [En ligne]. Disponible sur: https://www.sap.com/resources/what-is-field-service-management

[6] Fluix, « Field Service Management Explained ».

[7] J. E. Collins et E. M. Sisley, « Automated assignment and scheduling of service personnel », *IEEE Expert*, vol. 9, nº 2, p. 33-39, avr. 1994, doi: 10.1109/64.294133.

[8] ServicePower, « What is the Field Service Work Order Lifecycle? » Consulté le: 4 mai 2026. [En ligne]. Disponible sur: https://www.servicepower.com/blog/field-service-work-order-lifecycle

[9] S. Agnihothri, N. Sivasubramaniam, et D. Simmons, « Leveraging technology to improve field service », *Int. J. Serv. Ind. Manag.*, vol. 13, nº 1, p. 47-68, mars 2002, doi: 10.1108/09564230210421155.

[10] aidarsolutions, « field service management best practices ». Consulté le: 4 mai 2026. [En ligne]. Disponible sur: https://aidarsolutions.com/field-service-management-best-practices/

[11] « field-services-management-challenges ». [En ligne]. Disponible sur: https://www.netsuite.com/portal/resource/articles/erp/field-services-management-challenges.shtml

[12] Y. Wang, J.-P. Skeete, et G. Owusu, « Understanding the implications of artificial intelligence on field service operations: a case study of BT », *Prod. Plan. Control*, vol. 33, nº 16, p. 1591-1607, déc. 2022, doi: 10.1080/09537287.2021.1882694.

[13] C.-P. Bezemer et A. Zaidman, « Multi-tenant SaaS applications: maintenance dream or nightmare? », in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and*

International Workshop on Principles of Software Evolution (IWPSE), Antwerp Belgium: ACM, sept. 2010, p. 88-92. doi: 10.1145/1862372.1862393.

[14] P. M. Mell et T. Grance, « The NIST definition of cloud computing », National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-145, 2011. doi: 10.6028/NIST.SP.800-145.

[15] I. Gorton, R. Xu, Y. Yang, H. Liu, et G. Zheng, « Experiments in Curation: Towards Machine-Assisted Construction of Software Architecture Knowledge Bases », in *2017 IEEE International Conference on Software Architecture (ICSA)*, Gothenburg, Sweden: IEEE, avr. 2017, p. 79-88. doi: 10.1109/ICSA.2017.27.

[16] Huixin Chen, « Architecture strategies and data models of Software as a Service: A review », in *2016 3rd International Conference on Informative and Cybernetics for Computational Social Systems (ICCSS)*, Jinzhou, Liaoning, China: IEEE, août 2016, p. 382-385. doi: 10.1109/ICCSS.2016.7586486.

[17] P. Mika, « Big Data Conferences, Here We Come! », *IEEE Internet Comput.*, vol. 17, nº 3, p. 3-5, mai 2013, doi: 10.1109/MIC.2013.45.

[18] Microsoft Docs, « SaaS tenancy app design patterns ». [En ligne]. Disponible sur: https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns

[19] M. Armbrust *et al.*, « A view of cloud computing », *Commun. ACM*, vol. 53, nº 4, p. 50-58, avr. 2010, doi: 10.1145/1721654.1721672.

[20] S. A. Noor, R. Hasan, et M. M. Haque, « CellCloud: A Novel Cost Effective Formation of Mobile Cloud Based on Bidding Incentives », in *2014 IEEE 7th International Conference on Cloud Computing*, Anchorage, AK, USA: IEEE, juin 2014, p. 200-207. doi: 10.1109/CLOUD.2014.36.

[21] « Microsoft. ASP.NET Core documentation ». [En ligne]. Disponible sur: https://learn.microsoft.com/aspnet/core/web-api/

[22] « Microsoft. C# documentation ». [En ligne]. Disponible sur: https://learn.microsoft.com/dotnet/csharp/

[23] « Entity Framework Core — Microsoft. Entity Framework Core documentation. » [En ligne]. Disponible sur: https://learn.microsoft.com/ef/core/

[24] « SQL Server — Microsoft. SQL Server documentation. » [En ligne]. Disponible sur: https://learn.microsoft.com/sql/sql-server/

[25] « Swagger — SmartBear. Swagger documentation ». [En ligne]. Disponible sur: https://swagger.io/docs/

[26] « HTML — WHATWG. HTML Living Standard ». [En ligne]. Disponible sur: https://html.spec.whatwg.org/

[27] « TypeScript — Microsoft. TypeScript documentation. » [En ligne]. Disponible sur: https://www.typescriptlang.org/docs/

[28] « React — Meta (Facebook) and community. React documentation ». [En ligne]. Disponible sur: https://react.dev/

[29] « Tailwind CSS — Tailwind Labs. Tailwind CSS documentation. » [En ligne]. Disponible sur: https://tailwindcss.com/docs

[30] « Axios — Axios (open-source project). Axios documentation. » [En ligne]. Disponible sur: https://axios.rest/pages/getting-started/first-steps.html

[31] « React Router — React Router / Remix team. React Router documentation. » [En ligne]. Disponible sur: https://reactrouter.com/

[32] « React Query — TanStack (Tanner Linsley). TanStack Query documentation. » [En ligne]. Disponible sur: https://tanstack.com/query/latest

[33] « Zustand — pmndrs (Poimandres). Zustand documentation/repository. » [En ligne]. Disponible sur: https://github.com/pmndrs/zustand

[34] « Dart — Google. Dart documentation. » [En ligne]. Disponible sur: https://dart.dev/

[35] « Flutter — Google. Flutter documentation. » [En ligne]. Disponible sur: https://docs.flutter.dev/

[36] « Dio — official package page on pub.dev. » [En ligne]. Disponible sur: https://pub.dev/packages/dio

[37] « GetIt — official package page on pub.dev. » [En ligne]. Disponible sur: https://pub.dev/packages/get_it

[38] « Go_Router — official package page on pub.dev (published by flutter.dev). » [En ligne]. Disponible sur: https://pub.dev/packages/go_router

[39] « MonsterASP.NET — MonsterASP.NET. MonsterASP.NET (official website). » [En ligne]. Disponible sur: https://www.monsterasp.net/