UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

DESIGN AND DEVELOPMENT OF A FOOD DELIVERY WEB APPLICATION BASED ON MICROSERVICES

HECTOR EMANUEL PEÑA CUEVAS

Thesis supervisor
CRISTINA GÓMEZ SEOANE (Department of Service and Information System Engineering)

Degree
Bachelor's Degree in Informatics Engineering (Software Engineering)

Bachelor's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

22/01/2025

Abstract

This project aims to relate the design and implementation of a microservices-based web application, highlighting the knowledge acquired in the Software Engineering specialization. The latest trends and technologies in web development have been applied.

The developed application addresses the growing demand for food delivery orders, with a focus on scalability and ease of use, ensuring an efficient and satisfying experience for the end user.

Resumen

Este proyecto tiene como objetivo presentar el diseño e implementación de una aplicación web basada en microservicios, destacando así los conocimientos adquiridos en la especialidad de Ingeniería del Software. Se han aplicado las últimas tendencias y tecnologías en el ámbito del desarrollo web.

La aplicación desarrollada aborda la creciente demanda de pedidos de comida a domicilio, con un enfoque en la escalabilidad y la facilidad de uso, asegurando una experiencia eficiente y satisfactoria para el usuario final.

Resum

Aquest projecte té com a objectiu presentar el disseny i la implementació d'una aplicació web basada en microserveis, destacant així els coneixements adquirits en l'especialitat d'Enginyeria del Software. S'han aplicat les últimes tendències i tecnologies en l'àmbit del desenvolupament web.

L'aplicació desenvolupada respon a la creixent demanda de comandes de menjar a domicili, amb un enfocament en l'escalabilitat i la facilitat d'ús, assegurant una experiència eficient i satisfactòria per a l'usuari final.

Contents

Index of Figures

Index of Tables

1. Introduction

This project is a bachelor's thesis in Informatics Engineering from the *Facultat d'Informàtica de Barcelona* (FIB), part of the *Universitat Politècnica de Catalunya* (UPC). It falls within the Software Engineering specialization and is supervised by Cristina Gómez Seoane from the *Enginyeria de Serveis i Sistemes d'Informació* (ESSI) department. This thesis is made under the A modality.

1.1. Context

In today's fast-paced world, particularly following the 2020 pandemic, the demand for food delivery services has experienced an unprecedented surge [1]. This growth is largely driven by consumers' increasing preference for convenience, seeking to save time on cooking and avoiding the need to go out. As people adapt to busier lifestyles and the comfort of having meals delivered to their doorsteps, this industry faces significant challenges that must be addressed to keep pace with this rising demand.

To gain a broader perspective, we'll see statistics from Uber Eats, one of the most popular options. This chart (Table 1) shows off the increasing number of users it has acquired over the years.

Year	Users (millions)
2016	5
2017	9
2018	15
2019	21
2020	66
2021	81
2022	85
2023	88

Table 1: Uber Eats Revenue and Usage Statistics (2024). [2]

These numbers represent data from just one of the most popular options. In Spain, this industry holds so much potential, and the statistics presented in Table 1 are expected to continue growing even more in the coming years [3].

Food delivery applications may seem simple from the consumer's perspective, but they hide a significant amount of complex software that sustains their well functioning.

1.2. The Problem

Food delivery apps have become an integral part of daily life, yet many platforms still lack essential functionalities that could significantly enhance the user experience. While basic features like live order tracking are common, these apps often fall short in providing tools that enable users to effectively manage their spending. For instance, customers may struggle to **track their total expenditures** over time or identify spending trends.

In addition, **allergen information**—which is vital for users with dietary restrictions—remains incomplete or hard to find on many platforms. When allergen details are not readily available, users with specific needs may be put at risk, undermining their trust and overall safety.

Furthermore, many apps do not include a **loyalty points system**, a valuable feature that could boost user engagement by allowing customers to earn points for each order. These points could then be redeemed for discounts on delivery fees or service charges, fostering a sense of reward and encouraging customer loyalty. Additionally, the absence of a **PIN verification** process for deliveries can lead to errors, such as misdelivery to the wrong customer or location, compromising both security and service quality.

Finally, the rapid growth of food delivery services has also introduced several critical challenges related to **scalability**. As platforms grow, existing systems often struggle to handle large volumes of users and orders simultaneously, leading to slower processing times and potential service disruptions. On another hand, **maintainability** becomes a significant concern as platforms expand, with growing complexity leading to increased operational costs and difficulties in updating or improving the system efficiently. Platforms that fail to address these challenges risk losing their competitive edge.

1.3. Stakeholders

Stakeholders are individuals or groups who have an interest in or are impacted by the project's outcomes and processes.

Each stakeholder plays a vital role in the overall functionality of the application, contributing to various aspects of its lifecycle. The primary stakeholders for this food delivery application are listed below:

- **Restaurants:** Potential users of the application. These establishments will provide their menus and process orders through the platform.

- **Courier:** Individuals responsible for delivering food from the restaurant to the consumer.

- **Consumers:** Potential users of the application. These are individuals who will place food orders through the platform.

- **Bachelor's thesis director:** Cristina Gómez, from the *Enginyeria de Serveis i Sistemes d'Informació* (ESSI) department, who look after the quality and correctness of the project.

- **Developer:** The individual responsible for designing and implementing the application.

1.4. Motivation

Since I was a kid, I have always been fascinated by anything related to technology. I have long admired my cousin Rafael, who is now an established software developer. When he first introduced me to computers, it was a pivotal moment in my life; that's when I realized I wanted to build a career in technology. Then, early in my adolescence — quite earlier than most people I must say — I set my goal on becoming a software engineer.

That's where my passion for this sector comes from, but the idea for this bachelor's thesis comes from my nearly three years of experience working at McDonald's. During this time, I interacted with the type of application I am developing from the restaurant's perspective, and, like many others, I have also been a consumer of such services. From the restaurant point of view, I observed the dynamic nature of the food delivery industry and how it has transformed the business. For instance, the McDonald's restaurant where I worked was located near the beach, so during winter or on cold and rainy days when almost nobody is going out, the restaurant could still generate revenue by delivering food to customers' doorsteps.

Finally, this project is an opportunity for me to learn new skills, explore the world of software architecture design, and apply the knowledge I have gained during my studies at the FIB. My goal is to enhance my abilities as a developer and grow professionally through this experience.

2. Justification

The chapter begins with a detailed *State of the Art* review, analyzing prominent mainstream platforms to identify strengths, weaknesses, and opportunities for innovation. Then, the *Project Justification* section articulates the necessity for this application, highlighting its distinctive value proposition.

2.1. State of the Art

This section reviews the current landscape of food delivery applications, providing an analysis of existing solutions and industry standards. We examine leading platforms such as Uber Eats, Glovo, and Just Eat, exploring their features, strengths, and limitations. By analyzing these solutions, we aim to understand how they meet user needs and the gaps that remain in the market.

2.1.1. Uber Eats

Uber Eats [5] is a global platform launched by Uber Technologies in 2014. It connects users with a vast array of local restaurants through its app. The platform features real-time order tracking and a broad selection of restaurants. It supports various payment options and frequently offers promotional deals to enhance user experience.

Cons:

- **Lack of Consumption Graphics:** Users cannot view a graphical representation of their spending on orders.
- **Absence of Loyalty Points System:** There is no system for earning and redeeming loyalty points to save on delivery or service fees.
- **No Mandatory Allergen Information:** Some restaurants do not display allergen information for menu dishes.

2.1.2. Glovo

Glovo [6] is a Spanish on-demand delivery service that operates across multiple countries, providing users with the convenience of ordering not only food but also groceries and other items from local businesses. It features real-time tracking and a user-friendly interface that accommodates both food and non-food deliveries.

Cons:

- **Lack of PIN Verification for Delivery:** There is no PIN verification process to ensure that deliveries are made to the correct customer.
- **Lack of Consumption Graphics.**
- **Absence of Loyalty Points System.**
- **No Mandatory Allergen Information.**

2.1.3. Just Eat

Just Eat [7] is a UK-based service that connects customers with a large network of local restaurants. Just Eat operates in several countries and focuses exclusively on food delivery.

Cons:

- **Outdated User Interface:** The user interface is not as modern or visually appealing as other apps.
- **Lack of PIN Verification for Delivery.**
- **Lack of Consumption Graphics.**
- **No Mandatory Allergen Information.**

2.2. Project Justification

We do not want to reinvent the wheel, instead, we seek to create an application that fills specific gaps in the market while being reliable. It is essential that users can trust that their orders will be processed smoothly. Furthermore, the application should be able to handle more users and orders as demand grows, maintaining a fast and efficient experience. Through these efforts, we intend to deliver a service that meets both current needs and future expectations.

In the rapidly evolving market of applications, distinguishing oneself from other solutions often involves offering unique functionalities. In this project, we emphasize the importance of managing allergen information to protect the safety of users with dietary restrictions—an aspect many current apps overlook, despite its critical real-world impact. Alongside this allergen management, incorporating features such as secure PIN-based verification, graphical spending summaries, and a loyalty rewards system significantly enhances user engagement and satisfaction. Furthermore, providing insights into customer spending fosters greater awareness and promotes responsible financial habits. By implementing these features, our application establishes a distinctive niche among existing solutions.

Finally, while addressing these market needs, the student aims to deepen their understanding of the technical methodologies required to achieve these goals. This project serves as both a practical solution to a real-world problem and an opportunity for the developer to learn and apply cutting-edge techniques, ensuring the application's success both from a business and a personal growth perspective.

2.3. Obstacles and Risks

Developing this application presents several challenges and potential risks that must be addressed to ensure the success of the project. The following list outlines the primary obstacles and risks associated with this project:

1. **Time Limitation:** Given the scope of the project and the time constraints of the bachelor's thesis, there is a risk that certain features may not be fully implemented or refined. Proper time management and prioritization of core functionalities are crucial to ensure the project meets its primary objectives within the allocated time frame.

2. **Application Complexity:** The application involves multiple microservices, each responsible for different aspects of the system. Managing the complexity of service interactions, ensuring data consistency, and maintaining system performance are challenges that could result in delays or potential issues during development and testing.

3. **Real-time Tracking and Notifications:** Real-time order tracking and notifications for restaurants and couriers introduces additional layers of complexity. Implementing reliable real-time updates and ensuring that notifications are synchronized across all platforms may pose technical challenges.

4. **Single Developer:** Being the sole developer responsible for the entire system adds a significant workload and risk of bottlenecks. With no team to share the workload, balancing development, testing, and troubleshooting becomes more difficult, which can lead to increased time pressure and reduced code quality.

5. **Unforeseen Technical Challenges:** Unanticipated technical issues, such as system integrations, performance bottlenecks, or scalability problems, could arise and require additional time and resources to resolve. Additionally, for different phases of the development we will be using a variety of technologies that we are not familiar with, which may introduce unforeseen difficulties.

6. **Cloud Services:** For this project, we will utilize the AWS Free Tier to host our application. This tier comes with limitations that may restrict our access to necessary resources, particularly during periods of high demand or extensive testing.

3. Scope

Once we have provided the context, it is now time to define the scope we are addressing in order to better manage the tasks we must complete, as well as the time and effort required to accomplish them.

3.1. Objectives

The core objective of this thesis is to design and develop a food delivery web application that enables users to easily and safely place their orders, ensuring a seamless user experience.

Primary objectives are to implement the functionalities outlined in previous sections, including requiring allergen information for each dish, providing PIN-based verification, graphical spending summaries, and a loyalty rewards system.

However, while these functionalities are important, the true essence and value of this bachelor's thesis go beyond merely integrating these features.

Although there are various alternatives for developing this application, the primary sub-objective of this thesis is to learn how to design and develop a sophisticated system characterized by its scalability, reliability, and maintainability, utilizing a microservices-based architecture. Using this approach, the complexity of developing this application lies in ensuring that multiple microservices communicate effectively, managing data consistency across services, and maintaining high performance even under heavy usage.

On top of that, the project aims to demonstrate advanced software engineering principles by addressing challenges related to service orchestration, data management, and fault tolerance. By focusing on these aspects, we look forward to gaining a better understanding of modern software development practices and developing the ability to apply them in real-world contexts.

3.2. Requirements

This section outlines the essential criteria to be met, divided into functional and non-functional requirements.

3.2.1. Functional Requirements

Functional requirements define the core features and functionalities that the application must deliver to meet user needs and achieve operational goals. They outline what the system should do, detailing the essential tasks and services it must perform to fulfill its intended purpose.

- **User Registration and Authentication**
    - Users should be able to register for an account using email or social media accounts.
    - Users must be able to log in and log out securely.
    - Users should be able to manage their profiles.
- **Restaurant Management**
    - Restaurants should be able to add, update, or remove menu dishes.
    - Restaurants should be able to set pricing of their dishes.
    - Restaurants should receive real-time notifications of new orders.
    - Restaurants should be able to update the order status (e.g., preparing, ready for delivery).
- **Menu Browsing and Ordering**
    - Customers should be able to browse restaurant menus and view dish details.
    - Customers should be able to search for restaurants and menu dishes.
    - Customers should be able to apply filters to refine search results (e.g., cuisine type).
    - Customers should be able to place orders, specify delivery addresses, and choose payment methods.
    - Customers should be able to view their order history and reorder previous dishes.
    - Customers should be able to see information about their spending.
    - Customers should receive a unique PIN when their order is assigned to a courier.
    - Customers should earn loyalty points after each purchase.
    - Customers should be able to use their loyalty points to reduce delivery fees.

- Customers should be able to track their orders in real-time and see the order updates.

- **Delivery Management**
  - Couriers should be able to update the status of deliveries (e.g., picked up).
  - Couriers should be able to enter the PIN provided by the customer to confirm that the order has been delivered.

- **Ratings**
  - Customers should be able to rate restaurants.
  - Restaurants should be able to view feedback.

- **Admin Dashboard**
  - Administrators should be able to manage user accounts, restaurant profiles, and delivery personnel.
  - Administrators should have access to system analytics, including order volumes and user activity.

3.2.2. Non-Functional Requirements

Non-functional requirements emphasize the performance attributes and quality standards the application must meet. In other words, they address how the system should perform rather than the specific functions it offers, making them essential for delivering a high-quality, robust application. The non-functional requirements we aim to address are:

NFR 1 - Reliability

Description	The system's ability to continue functioning correctly, even in the event of failures or unexpected issues.
Justification	For food delivery services, reliability is critical to ensure orders are placed and fulfilled accurately without failures.
Acceptance criteria	The system must achieve 99.9% uptime.

NFR 2 - Scalability

Description	The system's capacity to handle increasing workloads without performance degradation, whether by adding more users, data, or processing demands.
Justification	Especially during peak hours, the system must be able to scale efficiently to avoid slowdowns or crashes. Scalability ensures that the application can grow alongside its user base without sacrificing performance.
Acceptance criteria	The system should support 200 requests, ensuring that the average response time remains under 2s.

NFR 3 - Maintainability

Description	It refers to how easily the system can be updated, modified, and improved over time.
Justification	A maintainable system ensures that developers can quickly adapt to new business requirements, fix bugs, and make improvements. This reduces long-term costs and ensures the system remains up-to-date and secure. This becomes even more critical as the complexity of the application increases.
Acceptance criteria	Code must follow clean coding practices and modular architecture, with clear documentation provided. The application will also include automated testing to ensure that no critical bugs are introduced.

NFR 4 - Appearance

Description	It refers to the visual design and layout of the application, including its user interface elements, color scheme, and overall aesthetic.
Justification	A well-designed appearance enhances user experience by making the application intuitive, engaging, and visually appealing. Consistent and attractive design can improve usability and reduce user frustration.
Acceptance criteria	It should receive positive feedback (more than 3.5 on average in a 1-5 scale) from the test users (10 test users approx.). The design must be responsive, ensuring a seamless experience across various devices and screen sizes.

NFR 5 - Ease of Use

Description	It refers to how intuitive and user-friendly the application is, including the simplicity of navigation, clarity of instructions, and overall user experience.
Justification	A user-friendly application ensures that users can quickly understand and interact with the system without extensive training or confusion.
Acceptance criteria	The system should receive an average rating of more than 3.5 (on a 1-5 scale) from test users (10 test users approx.) in usability testing. The application must have an intuitive interface with clear navigation and accessible features.

4. Methodology

Now, we will explain the methodology we are following in terms of work and development. Adhering to a strict methodological framework will allow us to complete the work more efficiently.

4.1. Work Methodology

There are a lot of project management methodologies [8]: Waterfall, Agile, Critical Path Method (CPM), Extreme Programming (XP), etc. But for the purpose and context of this project we will use Scrum. This decision is primarily based on the fact that this methodology provides significant flexibility, which is essential for a project of this nature. In the words of Scrums's creators, "Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems" [9 p. 3].

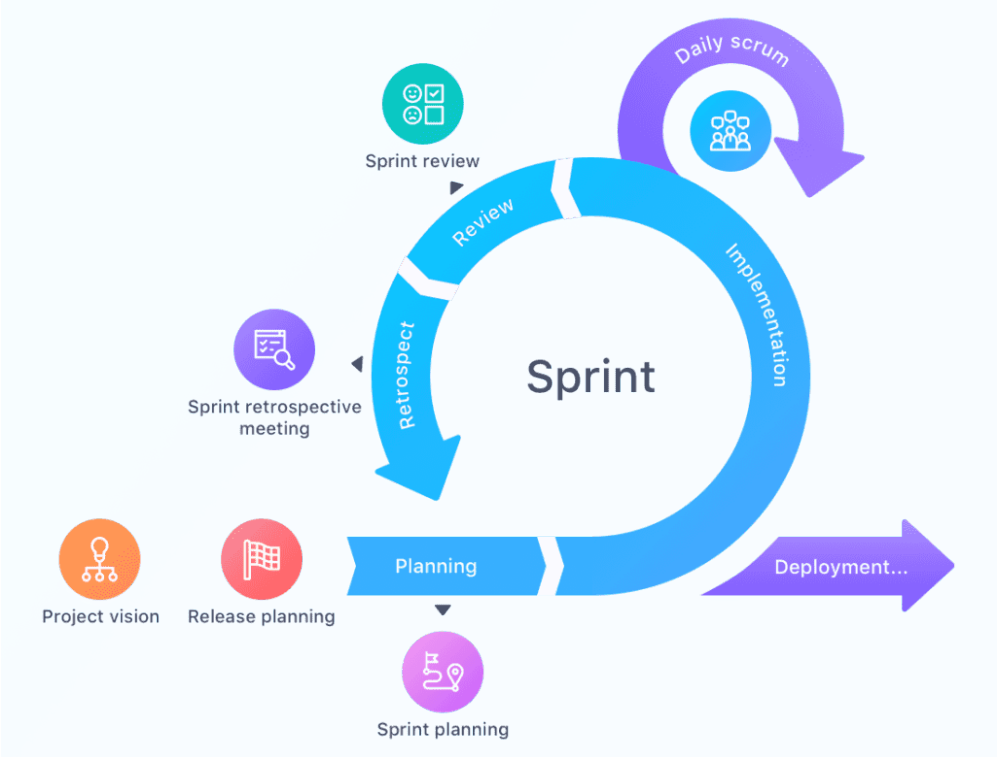


Figure 1: Scrum Cycle. [10]

As illustrated in Figure 1, Scrum operates through iterative cycles known as **sprints**, each consisting of the following key phases [11]:

- **Planning:** During this phase, the team collaborates to define the sprint goal and select the tasks from the *product backlog* (pending tasks) that will be completed during the sprint.

- **Implementation:** The development team works on executing the tasks, turning product backlog items into potentially shippable *increments* (sprint results). During

this phase **Daily Scrum** meetings are held to keep the team informed about the progress of the sprint.

- **Review and Retrospective:** At the end of the sprint, the team conducts a sprint review to demonstrate the completed work to stakeholders and gather feedback. Also, the team reflects on the sprint, identifying what went well, what could be improved, and how to make the next sprint more effective.

- **Deployment:** After successful implementation and review, the completed work is deployed to the production environment, making it available to users or clients.

Then, another important part of this methodology are the roles [12]:

- **Product Owner**: The person who has this role is responsible for maximizing the value of the product by managing the *product backlog*. We don't have that role due to the context of the project.

- **Scrum Master**: The Scrum Master facilitates the Scrum process, ensuring that the team follows Scrum principles and practices, and that deadlines are met. This role will be assumed by Cristina, the director.

- **Development Team**: Usually a group of between 3 and 9 individuals [13]. They are responsible for delivering *product increments* at the end of each sprint. This role will be assumed by me and I will be responsible for all the possible roles that may be required within the team.

Important adaptations of the methodology are needed. For instance, Daily Scrum meetings will not be held, as there is just one person fulfilling the development team role. Additionally, sprint reviews will take place every 1 or 2 weeks with the Scrum Master, so we can do both the revision and retrospective meeting. We will use **Google Meet** for our meetings and **Google Docs** for sharing documents.

4.2. Tools

To develop this application, we will use the following tools to enhance our efficiency and productivity:

- We will employ **Jira** to plan and manage each sprint, and we will utilize a Scrum board to manage the sprint tasks effectively. Jira will help us maintain control over the objectives and requirements that need to be addressed.

- To develop the code and complete our tasks, we will use **Visual Studio Code** as our integrated development environment (**IDE**). For **version control**, we will utilize **Git**,

a tool that tracks changes to files and facilitates collaboration. Additionally, we will host our code repository on GitHub and follow the Gitflow workflow.

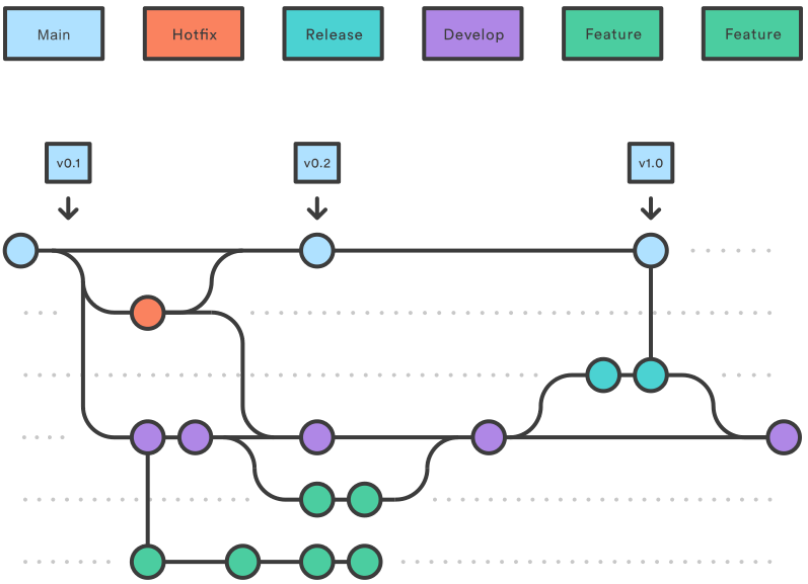


Figure 2: Gitflow Workflow. [14]

Gitflow is a branching model for Git that outlines a structured workflow with specific branches. The *master* branch represents the production-ready state of the project, while the *develop* branch serves as the integration branch for features, where new developments are merged before release. *Feature* branches are created from the *develop* branch to develop new features, and once complete, they are merged back into *develop*. *Release* branches are created from *develop* when preparing for a new release, allowing for final adjustments before merging into *master*. Lastly, *hotfix* branches are created from *master* to quickly address production issues and are subsequently merged back *master* after the fix is applied.

5. Time Planning

Time planning is a crucial aspect of the project management process. It allows us to estimate how long tasks will take and adjust the workload to balance with the available time.

Our initial approach is to structure and manage the workload according to the *European Credit Transfer System* (ECTS) guidelines set by the *Facultat d'Informàtica de Barcelona* (FIB). The project is assigned a total of 18 ECTS, with 15 credits allocated to the thesis itself and 3 credits for the Project Management subject. According to FIB's regulations [15], each ECTS corresponds to 30 hours of work, meaning the thesis should be completed within approximately 540 hours.

The university provides a clear timeline with key dates and deliverables that must be met to ensure the project is completed by the given deadline. Specifically, we have a total of 540 hours to allocate between September 11th, 2024 (the first college week), and January 13th, 2025 (one week before the thesis defense period), which gives us approximately 17 weeks to complete the work. We should dedicate around 6 to 7 hours per day, from Monday to Friday. However, it may be necessary to work on weekends to ensure deadlines are met.

Our intention is to manage and adapt the project timeline based on these 540 hours. Nevertheless, this estimate may fluctuate as the project progresses due to unforeseen challenges or changes in scope. Flexibility in time management will be critical to ensure that the project is completed successfully within the expected timeframe while maintaining high quality.

5.1. Description of Tasks

The tasks will be divided into three groups, corresponding to the major phases of the project.

5.1.1. Project Management

This phase corresponds to the first stage, where we will focus on planning, organization, and setting the foundation for the project.

- **PM1 - Contextualization and scope**: Define the overall context of the project and objectives while establishing the scope we will encompass. Determine the work methodology we will utilize.

- **PM2 - Time planning**: Create a project timeline that ensures sufficient time is allocated per each phase of the project.

- **PM3 - Budget management and sustainability**: Develop an accurate budget estimate covering resources and costs. Additionally, create a sustainability report.

- **PM4 - Project management final report**: Compile the previous documents into a final report, incorporating feedback from the Project Management tutor to ensure all requirements are met.

- **PM5 - System's specification**: Document the detailed requirements and functionalities of the system, including user stories, use cases and non-functional requirements.

5.1.2. Development

In this phase, the core of the project will be addressed, including the design, implementation and development of the application. Sprint retrospectives (including meetings with the director) and documentation tasks are included in the Estimates chapter, as they remain consistent throughout all sprints.

5.1.2.1. *Inception*

- **I1 - Architecture design**: Create a high-level design of the system architecture that details the microservices that will be developed, their interactions, and the technologies that will be utilized.

- **I2 - Set up Jira**: Set up Jira for project management by configuring boards, creating the product backlog, and establishing workflows and sprints.

- **I3 - Set up Github repository**: Initialize and configure a GitHub repository for version control and organizing branches.

- **I4 - User interface mock-ups**: Develop visual mock-ups of the user interface, illustrating key screens and user interactions to provide a clear design reference for implementation.

5.1.2.2. *Sprint 1*

- **DV1 - Restaurant Management:** Implement the core features to allow the restaurants to create and update their dishes.

- **DV2 - Search restaurants and view dishes:** Develop functionality for users to search for restaurants and browse their menus, including dish details and pricing.

- **DV3 - Admin Dashboard:** Provide a dashboard for administrators to monitor and manage platform activities and user accounts.

- **DV4 - CI/CD implementation:** Set up Continuous Integration (CI) and Continuous Deployment (CD) by automating the build, testing, and deployment processes.

5.1.2.3. *Sprint 2*

- **DV5 - Register and authentication:** Implement user registration and authentication features, allowing users to sign up and log in.

- **DV6 - Place an order:** Develop the functionality for users to place orders by selecting dishes from a restaurant's menu and processing payment.

- **DV7 - Assign order to courier:** Implement the logic to assign placed orders to available couriers.

- **DV8 - Update order status:** Implement functionality for updating the status of an order, allowing restaurants and couriers to mark orders as prepared, en route, or delivered.

5.1.2.4. *Sprint 3*

- **DV9 - Order tracking:** Enable real-time order tracking for customers, allowing them to monitor their order state throughout the delivery process.

- **DV10 - Notifications:** Set up a notification system to alert users about important updates, such as order status changes.

- **DV11 - Profile management:** Create features that allow users to manage their personal information, including updating their contact details or delivery address.

- **DV12 - View order history:** Develop a feature that allows consumers to view their previous orders and their details.

- **DV13 - View spending insights**: Develop a feature that provides consumers with visual insights and analytics about their spending habits over time.

- **DV14 - Loyalty points system**: Implement a system where customers can earn loyalty points after each purchase, which they can later redeem to save on delivery fees.

5.1.2.5. *Sprint 4*

- **DV16 - Ratings:** Implement a rating system where consumers can rate restaurants. Additionally, restaurants should be able to see their ratings.

- **DV17 - PIN:** Add functionality for customers to be able to confirm receipt of their order. Couriers should be able to introduce a PIN provided by the consumer to verify this confirmation.

- **DV18 - Final touches:** Perform final adjustments, bug fixes, and UI refinements to ensure the application is fully polished and functional.

5.1.3. Project Documentation and Defense Preparation

The final phase will focus on preparing the thesis report and the presentation for the thesis defense.

- **DC5 - Final report:** Write and finalize the thesis report, integrating all previously completed documentation. Review the final report to ensure it meets the requirements set by the FIB.

- **PM9 - Defense prep:** Prepare for the thesis defense by creating the presentation and practicing it to ensure a strong understanding of all project aspects. Additionally, anticipate and prepare for potential questions from the panel.

5.2. Resources

This chapter outlines the essential resources required for the successful development and completion of the project.

5.2.1. Human Resources

The developer (the student) will work approximately 30 to 35 hours per week until the project is completed. These hours will primarily be distributed from Monday to Friday, though weekend work may be required if necessary.

Test users will be needed to verify that certain requirements are effectively met. Finally, the thesis director, Cristina Gómez Seoane, will be an essential contributor.

5.2.2. Material Resources

First of all, a computer with an internet connection will be required. In this case, we will be using a MacOS 15 device. Additionally, an iPhone 13 (iOS 18) smartphone will be available for testing mobile-oriented features, if necessary.

5.2.3. Software Resources

G Suite applications will be used for documentation, meetings, emails, presentations, and graphics. This decision is driven by the fact that the university email is integrated with G Suite, making it more convenient to use these tools in the academic environment with our institutional email.

Zotero will be used to manage and track information sources throughout the project's lifecycle. Draw.io will be the tool for creating diagrams, while GanttProject will be used for managing the Gantt chart.

Visual Studio Code will be the IDE for development, and GitHub Desktop will simplify Git version control and repository management. Figma will be used for designing user interfaces. Docker Desktop will be essential for maintaining a local development environment, and Insomnia will be used to test API calls. For cloud-based tasks, we will rely on Amazon Web Services (AWS).

5.3. Estimates and Gantt diagram

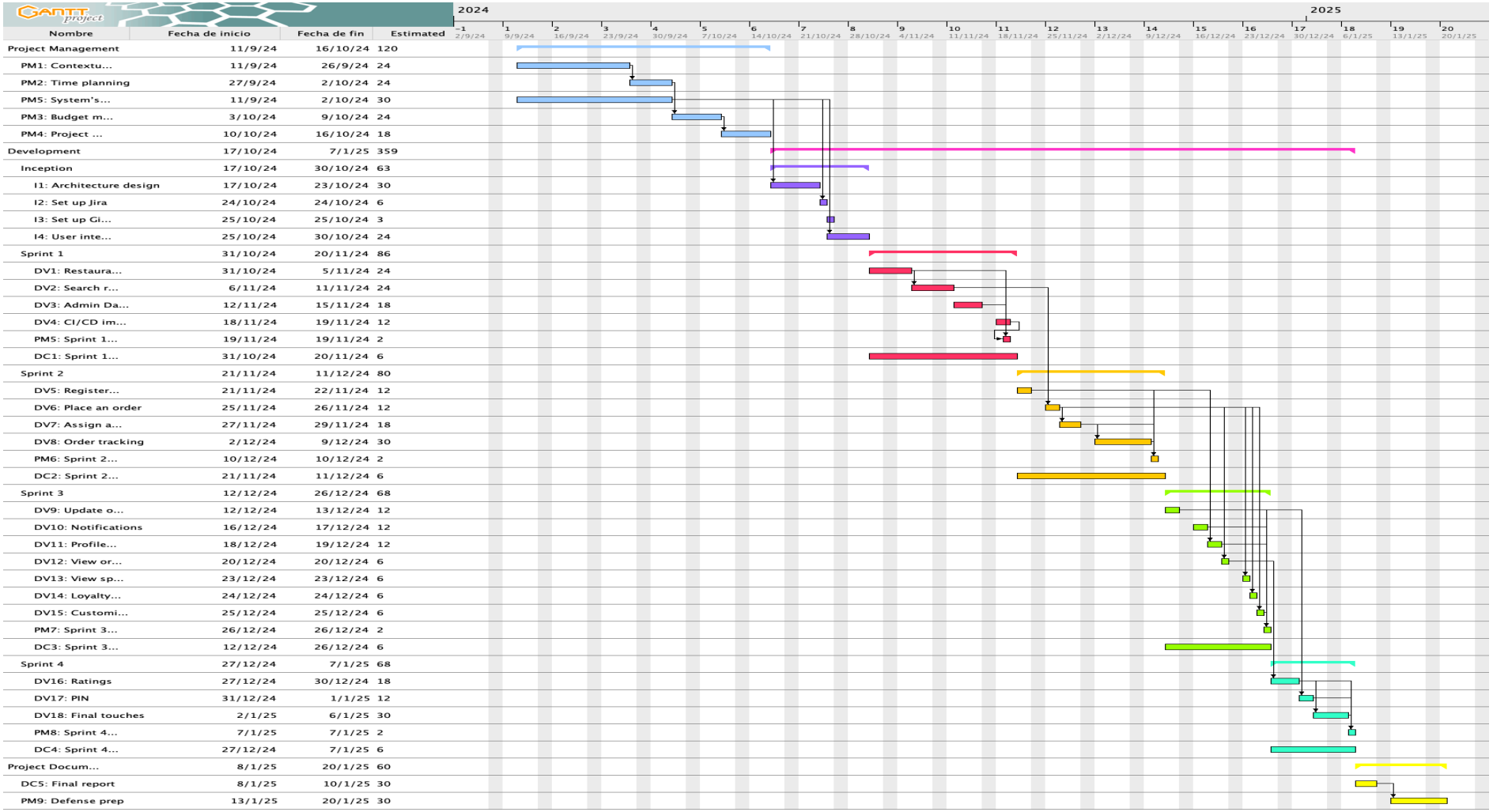
This chapter provides a detailed overview of the project's time estimates. It highlights the expected time needed for each task and phase, as well as the resources required. The estimated time is calculated in multiples of 6, with each multiple representing one full day of work. For instance, 18 hours would be an estimate of 3 days of work.

Code	Task	Time	Dependencies	Resources
	Project Management	**Total: 120**		
PM1	Contextualization and scope	24	-	G Suite
PM2	Time planning	24	PM1	G Suite, GanttProject
PM3	Budget management and sustainability	24	PM2	G Suite
PM4	Project management final report	18	PM3	G Suite
PM5	System’s specification	30	PM1	G Suite, Draw.io
	Development	**Total: 365**		
I	***Inception***	**Total: 63**		
I1	Architecture design	30	PM5	G Suite, Draw.io
I2	Set up Jira	6	PM5	Jira
I3	Set up Github repository	3	-	GitHub
I4	User interface mock-ups	24	PM5	Figma
S1	***Sprint 1***	**Total: 86**		
DV1	Restaurant Management	24	-	Visual Studio, GitHub, Docker
DV2	Search restaurants and view dishes	24	DV1	Visual Studio, GitHub, Docker
DV3	Admin Dashboard	18	-	Visual Studio, GitHub, Docker
DV4	CI/CD implementation	12	DV2	Visual Studio, GitHub, AWS
PM5	*Sprint 1* retrospective	2	DV1, DV2, DV3, DV4	G Suite, Jira
DC1	*Sprint 1* documentation	6	-	G Suite

S2	***Sprint 2***	**Total: 80**		
DV5	Register and authentication	12	-	Visual Studio, GitHub, Docker
DV6	Place an order	12	DV2, DV5	Visual Studio, GitHub, Docker
DV7	Assign order to courier	18	DV6	Visual Studio, GitHub, Docker
DV8	Update order status	30	DV7	Visual Studio, GitHub, Docker
PM6	*Sprint 2* retrospective	2	DV5, DV6, DV7, DV8	G Suite, Jira
DC2	*Sprint 2* documentation	6	-	G Suite
S3	***Sprint 3***	**Total: 62**		
DV9	Order tracking	12	-	Visual Studio, GitHub, Docker
DV10	Notifications	12	-	Visual Studio, GitHub, Docker
DV11	Profile management	12	DV5	Visual Studio, GitHub, Docker
DV12	View order history	6	DV6	Visual Studio, GitHub, Docker
DV13	View spending insights	6	DV6	Visual Studio, GitHub, Docker
DV14	Loyalty points system	6	DV6	Visual Studio, GitHub, Docker
PM7	*Sprint 3* retrospective	2	DV9, DV10, DV11, DV12, DV13, DV14, DV15	G Suite, Jira
DC3	*Sprint 3* documentation	6		G Suite
S4	***Sprint 4***	**Total: 68**		
DV16	Ratings	18	DV12	Visual Studio, GitHub, Docker
DV17	PIN	12	DV9	Visual Studio, GitHub, Docker
DV18	Final touches	30	DV14, DV15	Visual Studio, GitHub, Docker
PM8	*Sprint 4* retrospective	2	DV14, DV15, DV16	G Suite, Jira
DC4	*Sprint 4* documentation	6	-	G Suite
	Project Documentation and Defense Preparation	**Total: 60**		
DC5	Final report	30	-	G Suite, Jira
PM9	Defense prep	30	DC5	G Suite
TOTAL		**539**		

Table 2. Summary of the project's tasks

A Gantt diagram [16] is a project management tool that visually maps tasks and milestones along a timeline, helping to schedule, track progress, and manage deadlines, dependencies, and resources. Figure 3 illustrates the Gantt diagram.



Nombre	Fecha de inicio	Fecha de fin	Estimated
Project Management	11/9/24	16/10/24	120
PM1: Contextu...	11/9/24	26/9/24	24
PM2: Time planning	27/9/24	2/10/24	24
PM5: System's...	11/9/24	2/10/24	30
PM3: Budget m...	3/10/24	9/10/24	24
PM4: Project ...	10/10/24	16/10/24	18
Development	17/10/24	7/1/25	359
Inception	17/10/24	30/10/24	63
I1: Architecture design	17/10/24	23/10/24	30
I2: Set up Jira	24/10/24	24/10/24	6
I3: Set up Gi...	25/10/24	25/10/24	3
I4: User inte...	25/10/24	30/10/24	24
Sprint 1	31/10/24	20/11/24	86
DV1: Restaura...	31/10/24	5/11/24	24
DV2: Search r...	6/11/24	11/11/24	24
DV3: Admin Da...	12/11/24	15/11/24	18
DV4: CI/CD im...	18/11/24	19/11/24	12
PM5: Sprint 1...	19/11/24	19/11/24	2
DC1: Sprint 1...	31/10/24	20/11/24	6
Sprint 2	21/11/24	11/12/24	80
DV5: Register...	21/11/24	22/11/24	12
DV6: Place an order	25/11/24	26/11/24	12
DV7: Assign a...	27/11/24	29/11/24	18
DV8: Order tracking	2/12/24	9/12/24	30
PM6: Sprint 2...	10/12/24	10/12/24	2
DC2: Sprint 2...	21/11/24	11/12/24	6
Sprint 3	12/12/24	26/12/24	68
DV9: Update o...	12/12/24	13/12/24	12
DV10: Notifications	16/12/24	17/12/24	12
DV11: Profile...	18/12/24	19/12/24	12
DV12: View or...	20/12/24	20/12/24	6
DV13: View sp...	23/12/24	23/12/24	6
DV14: Loyalty...	24/12/24	24/12/24	6
DV15: Customi...	25/12/24	25/12/24	6
PM7: Sprint 3...	26/12/24	26/12/24	2
DC3: Sprint 3...	12/12/24	26/12/24	6
Sprint 4	27/12/24	7/1/25	68
DV16: Ratings	27/12/24	30/12/24	18
DV17: PIN	31/12/24	1/1/25	12
DV18: Final touches	2/1/25	6/1/25	30
PM8: Sprint 4...	7/1/25	7/1/25	2
DC4: Sprint 4...	27/12/24	7/1/25	6
Project Docum...	8/1/25	20/1/25	60
DC5: Final report	8/1/25	10/1/25	30
PM9: Defense prep	13/1/25	20/1/25	30

Figure 3. Gantt diagram.

5.4. Risk Management

This section revisits the potential risks previously mentioned in the Scope chapter and offers strategies to mitigate them. Alternative solutions are provided to keep the project on schedule and minimize the impact of unforeseen challenges. Table 3 provides a summary of the risks, their impact on the project, and the probability of occurrence.

Risk/Obstacle	Impact	Probability
Time Limitation	Medium-High	High
Application Complexity	Medium	High
Real-time Tracking and Notifications	Low-Medium	Medium
Single Developer	High	High
Unforeseen Technical Challenges	Medium	Medium-High
Cloud Services	Low	Low

Table 3. Risks probability and impact

5.4.1. Time Limitation

To address time constraints in the bachelor's thesis, we will implement a phased approach, prioritizing core functionalities in the first two sprints. By **focusing on a minimum viable product** (MVP), we can ensure essential features are completed on time. This strategy may extend the initial development period as we allocate more time to these sprints. We will also dedicate time for comprehensive project planning and design, using project management tools and CI/CD to optimize our workflow. If needed, we will reduce the project's scope as the last resource.

5.4.2. Application Complexity

Managing multiple microservices is complex, particularly for service interactions and data consistency. To address this, we will create documentation to outline service interactions and **define clear API contracts**. While this may slightly extend the development phase, it will lead to a more stable system.

5.4.3. Real-time Tracking and Notifications

Implementing real-time tracking and notifications adds complexity, particularly in synchronization across platforms. To manage this, we will explore existing **frameworks** like Firebase [17] and Websockets, which **simplify real-time communications**. Additionally, we may need to dedicate time on weekends to learn how to use these tools effectively.

5.4.4. Single Developer

Being the sole developer presents risks in workload management and bottlenecks. To mitigate these, we will use Scrum methodology for regular assessments and priority adjustments. Additionally, utilizing libraries for non-critical frontend components will reduce the workload. The student has also spent the summer learning microservices architecture and related technologies, enabling more time for practical work during development, rather than focusing excessively on learning and research.

5.4.5. Unforeseen Technical Challenges

Unanticipated technical issues, such as integration difficulties, may arise during development. To mitigate these challenges, we will implement Scrum methodology for continuous progress assessment through sprints and reviews. Version control branching will also allow us to isolate new features, minimizing disruption to the main development branch.

5.4.6. Cloud Services

Utilizing the AWS Free Tier [18] for hosting our application has limitations, especially during high-demand periods. To handle this, we will implement auto-scaling mechanisms within AWS to adjust resources based on traffic. If setting this up becomes too complex or time-consuming, a simpler alternative is to upgrade to a paid AWS plan temporarily to increase available resources, or to change our cloud provider. This would be an easy, quick fix to ensure the application remains reliable during peak times, without requiring complex configurations. Another option is to reduce features or limit certain functionalities during high traffic to keep the system stable.

6. Budget and Sustainability

Once the time planning is completed, we can move on to estimating the project's costs and assessing its sustainability impact across social, economic, and environmental dimensions.

6.1. Budget

Cost estimation is a crucial aspect of project management, allowing us to determine the financial resources required for successful development. This includes expenses related to development tools, cloud services, human resources, and other essential costs necessary to complete the project.

By accurately estimating the budget, we can ensure that sufficient resources are allocated to each phase, reducing the risk of overspending or underfunding critical components. In this section, we will outline the estimated costs associated with various aspects of the project, factoring in both direct and indirect expenses.

6.1.1. Personnel Costs per Activity

Personnel costs per activity (PCA) refer to the allocation of resources for human efforts involved in the project tasks outlined in the Gantt diagram from the previous chapter. This includes time spent on design, development, and project management.

Since the developer is handling multiple roles—such as design, coding, and documentation—the personnel costs will be based on a standard hourly rate for these tasks. Hourly rates are sourced from Indeed [19] to ensure the budget estimate is as realistic as possible, with the salary figures based on the market rates in Barcelona.

Table 4 lists the roles that would be required to complete the project in a real-world scenario, along with the hourly rates both before and after including Social Security taxes. To calculate the rate with taxes, we multiplied the base salary by a factor of 1.35.

Role	Hourly Rate (Excl. Taxes)	Hourly Rate (Incl. Taxes)
Project Manager (PM)	24,13 €	32.58 €
Software Architect (AC)	31,69 €	42.78
Software Analyst (AN)	18,92 €	25.54
Web Designer (WD)	11,76 €	15.88
***Front-end* Developer (FD)**	23,12 €	31.21
***Back-end* Developer (BD)**	25,13 €	33.93
Tester (TS)	15,25 €	20.59

Table 4. Hourly Rates for Each Project Role

Using the hourly rate for each role, we can estimate the cost per task and determine the overall estimated cost for the project. Table 5 displays the detailed breakdown of personnel costs.

Code	Task	Hours							Total Hours	Cost (EUR)
		PM	AC	AN	WD	FD	BD	TS		
	Project Management									
PM1	Contextualization and scope	24							24	781,92
PM2	Time planning	24							24	781,92
PM3	Budget management and sustainability	24							24	781,92
PM4	Project management final report	18							18	586,44
PM5	System's specification		18	12					30	1.076,52
	Development									
I	***Inception***									
I1	Architecture design		24	6					30	1.179,96
I2	Set up Jira	6							6	195,48
I3	Set up Github repository	3							3	97,74
I4	User interface mock-ups				24				24	381,12
S1	***Sprint 1***									
DV1	Restaurant Management					6	18		24	798
DV2	Search restaurants and view dishes					18	6		24	765,36
DV3	Admin Dashboard					6	12		18	594,42
DV4	CI/CD implementation						12		12	407,16
PM5	*Sprint 1* retrospective	2							2	65,16
DC1	*Sprint 1* documentation	6							6	195,48
S2	***Sprint 2***									
DV5	Register and authentication					3	9		12	399
DV6	Place an order					6	6		12	390,84
DV7	Assign order to courier					3	15		18	602,58

DV8	Update order status					12	18		30	985,26
PM6	*Sprint 2* retrospective	2							2	65,16
DC2	*Sprint 2* documentation	6							6	195,48
S3	***Sprint 3***									
DV9	Order tracking					3	9		12	399
DV10	Notifications					6	6		12	390,84
DV11	Profile management					9	3		12	382,68
DV12	View order history					3	3		6	195,42
DV13	View spending insights					3	3		6	195,42
DV14	Loyalty points system					3	3		6	195,42
PM7	*Sprint 3* retrospective	2							2	65,16
DC3	*Sprint 3* documentation	6							6	195,48
S4	***Sprint 4***									
DV16	Ratings					6	12		18	594,42
DV17	PIN					6	6		12	390,84
DV18	Final touches		3	3		12	6	6	30	906,60
PM8	*Sprint 4* retrospective	2							2	65,16
DC4	*Sprint 4* documentation	6							6	195,48
	Project Documentation and Defense Preparation									
DC5	Final report	30							30	977,40
PM9	Defense prep	30							30	977,40
TOTAL		**191**	**45**	**21**	**24**	**108**	**150**	**6**	**545**	**17.453,64**

Table 5. Personnel costs per task

6.1.2. Generic Costs

This section of the budget (GC) includes all costs that are calculated at a global level and not directly associated with any specific task or activity. Instead, they relate to the project as a whole.

Hardware

Given that hardware will be used beyond the project's timeframe, we will amortize its cost over a standard useful life of four years.

To calculate the amortized cost of the hardware during the course of the project, we consider the number of working days in a year in Catalonia, which for 2024 are 251 days [20], and assume a daily work dedication of 6.5 hours. The total duration of the project is 545 hours, which will be factored into the amortization formula.

The formula to determine the portion of the hardware cost attributed to the project is as follows:

$$\frac{Cost\ (EUR)}{Useful\ life\ (years) \cdot Annual\ working\ days \cdot Daily\ dedication\ (hours)} \cdot Use\ in\ the\ project\ (hours)$$

Table 6 will demonstrate the application of the formula for each of the hardware resources we will utilize throughout the project.

Hardware	Cost (EUR)	Amortization (EUR)
Mac Studio (Computer [21])	2.428,85	202,84
Logitech MX Keys S (keyboard [22])	119	9,94
Logitech MX Master 3S (mouse [23])	82,64	7,15
LG UltraGear 27GS75Q-B (monitor 1 [24])	219,99	18,37
LG 27MK600M-W (monitor 2 [25])	180,99	15,11
iPhone 13 (smartphone [26])	859	71,74
Total	**3890,47**	**325,15**

Table 6. Amortization costs for the hardware resources

Software

In our case, software will not be an issue, as we will rely exclusively on free software tools throughout the project.

The only potential cost could arise from cloud services; however, by utilizing Amazon Web Services (AWS) Free Tier, we can meet our requirements without incurring additional fees. While the Free Tier does come with certain limitations compared to paid tiers, it provides sufficient resources to cover the scope and scale of this project.

Indirect costs
Indirect costs, such as electricity, internet, and workspace expenses, will not be factored into the budget since the work will be conducted from home, where these costs do not need to be accounted for.

6.1.3. Other Costs

There are additional costs that must be considered, which do not fall under the categories of direct or indirect expenses that we have already covered in the generic costs section. These costs include contingencies and incidental expenses, which are necessary to ensure we are prepared for unexpected situations or predictable challenges we have previously discussed in earlier chapters.

Contingencies
Contingency costs are allocated to address unforeseen events or challenges that may arise during the project's execution. These funds provide a safety net, ensuring that the project can continue smoothly in the face of unexpected delays, additional resource requirements, or other unforeseen circumstances. By setting aside a contingency budget, we can mitigate risks and maintain project momentum without compromising on quality or deadlines.

Taking this into consideration, we have included a 15% contingency margin to our actual expenses (PCA + GC = 17.778,79€), bringing the total to **20.445,61€.**

Incidental costs
Incidental costs refer to smaller, often unpredictable expenses that may arise during the project. As we have previously discussed these potential events, we will concentrate on presenting the costs associated with addressing them in Table 7. By estimating the additional hours required, the personnel involved, and the probability of each event occurring, we can calculate the total incidental costs.

Incident	Additional Hours					Estimated Cost (EUR)	Risk (%)	Cost (EUR)
	PM	AC	AN	FD	BD			
Time limitation	12					390,96	75%	293,22
Application Complexity		12	6			666,60	75%	499,95
Real-time Tracking and Notifications				6	12	594,42	35%	208,05
Single Developer	6					195,48	75%	146,61

Unforeseen Technical Challenges	6				12	602,64	50%	301,32
Cloud Services	6					195,48	15%	29,32
Total								**1.478,47**

Table 7. Incidental costs.

6.1.4. Total Cost

With the calculations of Personnel Costs per Activity (PCA), Generic Costs (GC), the contingency margin, and the incidental costs now complete, we can consolidate these figures to provide a comprehensive overview of the project's financial requirements in Table 8.

Activity	Cost (EUR)
Personnel Costs per Activity (PCA)	17.453,64
Generic Costs (GC)	325,15
Contingency Margin	2,666,82
Incidental Costs	1.478,47
Total	**21.924,08**

Table 8. Total cost

6.1.5. Management Control

After calculating the project budget, it is crucial to monitor deviations from the initial predictions, as large projects often encounter obstacles that prevent budget and time estimations from being fully met, both expected and unforeseen. Therefore, we must establish a model to control potential budget deviations. Each time a task is completed, we need to compute the deviations for all associated costs. We will only consider the PCA deviations, as the other costs are fixed and non-variable.

To calculate the deviations and maintain control over the time and resources spent throughout the project, we will use the following formulas. Results close to zero indicate greater accuracy in the initial estimation, validating effective project management.

- **Total deviation of a task:** Track the actual hours worked per each role and calculate the deviation from initial estimates.

$$(Actual\ hours\ per\ role\ -\ Estimated\ hours\ per\ role)\ \cdot\ Hourly\ rate\ per\ role$$

- **Total deviation of costs:** The sum of all the deviations across each task, where n represents the total number of tasks. This provides a clear overview of how much the actual costs have differed from the estimated budget.

$$\sum_{i=1}^{n} (Total\ deviation\ of\ task\ i)$$

6.2. Sustainability

To analyze the sustainability of our project, we will use the matrix provided by the university (Table 9).

	PPP	Lifecycle	Risks
Environmental	Design Consumption	Ecological Footprint	Environmental Risks
Economic	Invoice	Viability Plan	Economic Risks
Social	Personal Impact	Social Impact	Social Risks

Table 9. Sustainability Matrix of the Bachelor's Thesis (TFG). [27]

6.2.1. Self-assessment

After completing the EDINSOST2-ODS survey [28], I have assessed my sustainability knowledge and concluded that I have a solid grasp of ethical principles and understand the role of professional responsibility in ensuring my engineering decisions are socially and environmentally sound. I recognize the importance of considering long-term impacts and how they influence sustainable practices.

Regarding environmental issues, I am aware of challenges such as pollution and climate change, but I realize there is much more to learn about efficiently managing resources and promoting renewable energy. My understanding in this area feels somewhat superficial, and I see it as a key opportunity for growth.

On the social side, I value inclusivity and fairness in engineering solutions, though I am still figuring out how to systematically integrate these principles into real-world projects.

Lastly, while I am familiar with the Sustainable Development Goals (SDGs) and understand the importance of aligning projects with these goals, I recognize that economic sustainability is an area I need to explore further, particularly in balancing environmental concerns with financial viability.

6.2.2. Economic Dimension

The application will be free to use, with payments only made to restaurants for orders. Couriers will earn a commission for each delivery they complete.

Regarding the resources consumed during the project's realization (Economic Cell/PPP), the estimated costs (*6.1. Budget*) include materials, human resources, and other necessary expenses. These costs represent what would be invoiced to a potential client and have been determined through a detailed time plan for the project, ensuring realistic and efficient budgeting.

For the project's viability during its lifecycle (Economic Cell/Life Use), we assume that potential revenue streams will come from commissions earned from restaurants and delivery fees. While no detailed financial viability report has been created, this context allows us to consider these sources as the basis for ensuring the application's sustainability and potential profitability in the long term.

Potential risks (Economic Cell/Risks) that could delay or prevent the project from achieving viability have also been considered. These include higher-than-expected development costs or longer implementation times. Risk mitigation strategies involve incorporating contingency margins into the budget and monitoring progress.

Additionally, to promote mindful consumption, the app features a spending tracker that allows users to monitor their restaurant-related expenses. This encourages greater awareness of spending habits, fostering a more conscious approach to ordering food.

6.2.3. Environmental Dimension

The design and implementation of this software have prioritized the environmental dimension by ensuring that resources are utilized efficiently and without waste. Like most modern web applications, we will leverage cloud services, which rely on electricity to power servers. However, we are committed to minimizing unnecessary usage.

To address the environmental impact during the project's realization (Environmental Cell/PPP), we are not directly evaluating or monitoring potential energy consumption (kWh) and CO2 emissions associated with development devices. However, the project design focuses on minimizing unnecessary resource usage to reduce its overall environmental footprint.

Regarding the environmental footprint throughout the application's lifecycle (Environmental Cell/Life Use), the project's design will focus on minimizing energy consumption during regular operations. By using AWS, we can configure services to allocate resources dynamically based on actual needs, and Kubernetes will enable efficient resource scaling, reducing wasteful energy use.

Finally, potential risks (Environmental Cell/Risks) that could lead to a higher environmental impact than anticipated will be considered. These include unexpected increases in user demand leading to higher energy consumption, misconfigurations in resource allocation, or reliance on data centers that do not prioritize renewable energy sources.

6.2.4. Social Dimension

The development of this bachelor thesis has had a meaningful impact on my personal and professional growth (Social Cell/PPP). Through the process, I have deepened my understanding of software architecture and development, fulfilling long-held aspirations. This journey has also influenced my immediate environment, as my dedication and learning experiences have inspired discussions and reflections among my peers and family about the value of continuous learning and technological advancement.

For its lifecycle impact on society (Social Cell/Life Use), the application aims to support individuals facing time constraints due to work, childcare, or education. By providing a convenient solution for ordering meals, the app offers users more flexibility to focus on other responsibilities and improve their quality of life. It also creates opportunities for couriers and restaurants, potentially boosting local economies and providing jobs in the food delivery sector.

Regarding potential risks (Social Cell/Risks), unintended consequences could arise, such as over-reliance on food delivery services impacting users' health or small, local restaurants struggling to compete with larger chains.

7. Specification

This section provides a comprehensive overview of the system's design and functionality through various models and diagrams. This includes detailed use case diagrams that illustrate the interactions between different actors and the system, specifying the core functionalities and user interactions. Additionally, the section encompasses the conceptual model, which outlines the high-level structure and relationships within the system, and the behavior model, which describes the dynamic aspects of the system's operations.

By detailing these elements, we aim to define the system's architecture and behavior, ensuring a clear understanding of how the application will function and meet its requirements.

7.1. Use Case Diagrams

This section provides a detailed specification of the system's functional requirements through various use case diagrams. Each diagram illustrates the interactions between different actors and the system, defining the core functionalities and processes integral to the application.

The use case diagrams are categorized by key functional areas, including user registration and authentication, restaurant management, menu browsing and ordering, delivery management, ratings, and the admin dashboard. These diagrams serve as a blueprint for understanding how the system operates, outlining the essential features and interactions necessary to meet user needs and operational goals.

Users: User Registration and Authentication

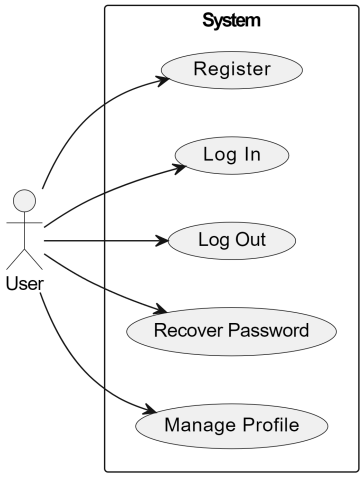


Figure 4. User Registration and Authentication Use Case Diagram

Restaurants: Restaurant Management

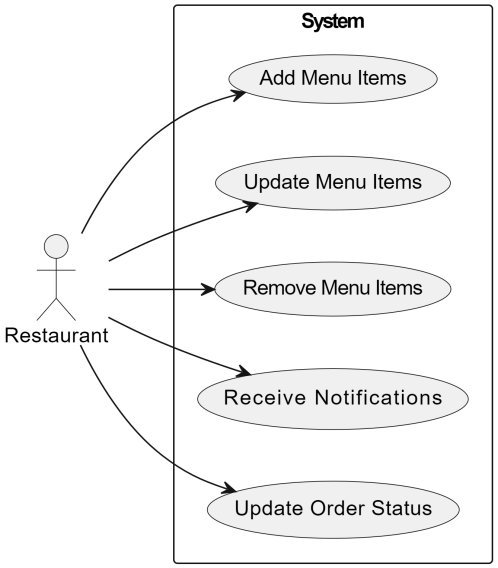


Figure 5. Restaurant Management Use Case Diagram

Customers: Menu Browsing and Ordering

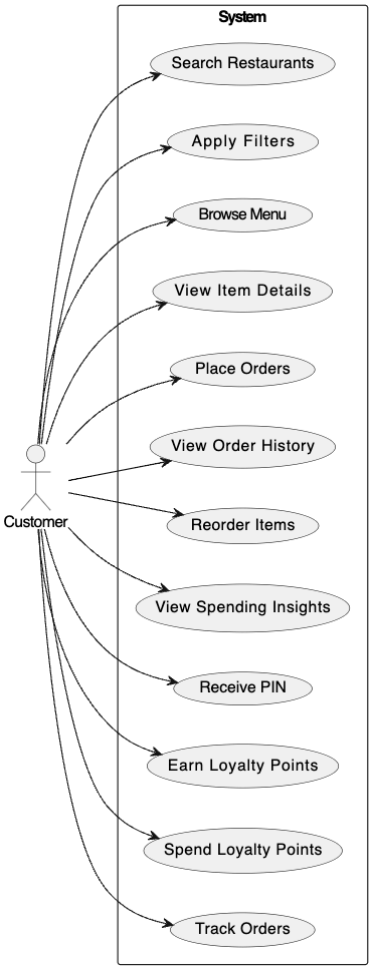


Figure 6. Menu Browsing and Ordering Use Case Diagram

Couriers: Delivery Management

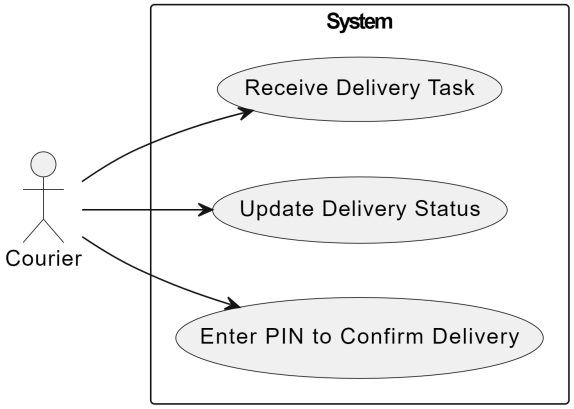


Figure 7. Delivery Management Use Case Diagram

Customers and Restaurants: Ratings

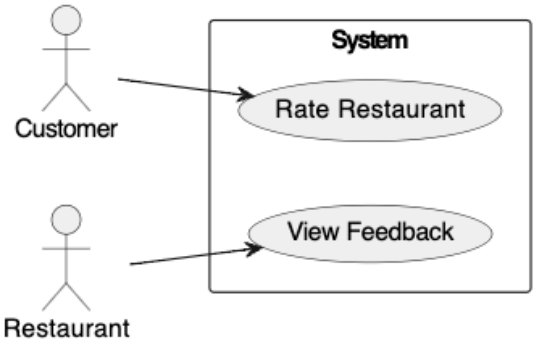


Figure 8. Ratings Use Case Diagram

Administrators: Admin Dashboard

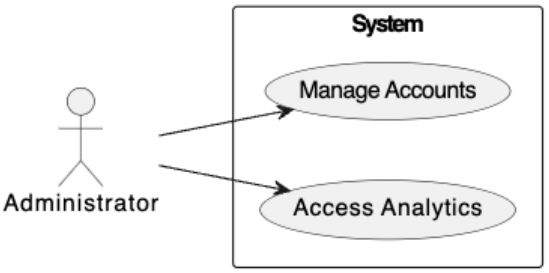


Figure 9. Admin Dashboard Use Case Diagram

7.2. Use Cases Descriptions

Now, we provide detailed descriptions of the core use cases identified. Each use case represents a specific interaction between users and the system. The descriptions cover the core functionalities of the application, detailing how various actors—such as customers, restaurants, couriers, and administrators—engage with the system.

The remaining use cases, which cover additional functionalities, can be found in Annex A.

Users: User Registration and Authentication

Use case	Register
Actor	User
Preconditions	● The user must not already have an account. ● The registration page must be accessible.
Trigger	The user decides to create a new account on the platform.
Principal Use Scenario	1. The user indicates they want to create an account. 2. The user provides necessary details such as email, password, and other optional information (phone number, etc.). 3. The system validates the provided information (e.g., email format). 4. If valid, the system creates a new account and sends a confirmation email. 5. The system confirms the successful registration.
Extensions	4a. If the email is already in use, the system informs the user and prompts them to log in or use a different email.

Use case	Log In
Actor	User
Preconditions	● The user must be registered and have a verified account.
Trigger	The user decides to log in to the platform.
Principal Use Scenario	1. The user enters their credentials (email and password or social media account) to log in to the application. 2. The system validates the credentials. 3. If the credentials are correct, the user is granted access to their account.
Extensions	1a. If the user forgets their password, they can request a

	password reset through an email link. 3a. If the credentials are incorrect, the system displays an error message and allows the user to retry.

Use case	Manage Profile
Actor	User
Preconditions	● The user must be logged in.
Trigger	The user wants to view or update their account details.
Principal Use Scenario	1. The user indicates that they want to update their profile details. 2. The system displays the user's current profile information (e.g., name, email, phone number, address). 3. The user modifies the details they want to update (e.g., edit delivery address). 4. The system validates any updates. 5. If valid, the system updates the profile information. 6. The system confirms the changes.
Extensions	5a. If the user enters invalid or incomplete information, the system prompts them to correct it.

Restaurants: Restaurant Management

Use case	Add Menu Dishes
Actor	Restaurant Manager
Preconditions	● The restaurant must have an account, and the user must be logged in.
Trigger	The restaurant manager decides to add a new dish to the restaurant's menu.
Principal Use Scenario	1. The restaurant manager indicates that they want to add a new menu dish. 2. The system prompts the manager to enter the dish details (e.g., name, description, category, price, allergens). 3. The manager enters the details and confirms the addition. 4. The system validates the input (e.g., no fields left empty, valid price format).

	5. If valid, the system saves the new menu dish and updates the restaurant's menu.
Extensions	5a. If any mandatory fields are missing, the system prompts the manager to fill in the required details. 5b. If there are errors in the price or format, the system asks for corrections before saving.

Use case	Update Order Status
Actor	Restaurant Manager
Preconditions	● The restaurant must have an active account, and the user must be logged in.
Trigger	The restaurant manager updates the status of an order as it progresses (e.g., from "preparing" to "ready for delivery").
Principal Use Scenario	1. The restaurant manager indicates that they want to update the order status. 2. The manager updates the status. 3. The system confirms the update and notifies the customer and courier if necessary.

Customers: Menu Browsing and Ordering

Use case	Place Orders
Actor	Customer
Preconditions	● The customer has selected dishes from a restaurant's menu and added them to their cart.
Trigger	The customer decides to complete the order.
Principal Use Scenario	1. The customer reviews the dishes in the cart. 2. The customer confirms the order. 3. An external service processes the payment and confirms the order with the restaurant. 4. The system displays a confirmation message with the order details.
Extensions	3a. If the payment fails, the system prompts the customer to retry.

Use case	View Spending Insights
Actor	Customer
Preconditions	● The customer has previously placed orders.
Trigger	The user wants to see his/her spending insights.
Principal Use Scenario	1. The customer initiates a request to view their spending insights. 2. The system retrieves and processes the customer's historical order and spending data. 3. The system provides an overview of the customer's spending patterns, including relevant insights such as total amount spent, and spending trends over time.
Extensions	2a. If no spending data is available, the system informs the customer.

Use case	Receive PIN
Actor	Customer
Preconditions	● The customer has placed an order, the restaurant has confirmed it and a courier is assigned to it.
Trigger	The system generates a unique PIN for order delivery verification.
Principal Use Scenario	1. After the order is assigned to a courier, the system generates a unique PIN for the order. 2. The customer receives the PIN via email.

Use case	Spend Loyalty Points
Actor	Customer
Preconditions	● The customer must have a sufficient number of loyalty points available in their account.
Trigger	The customer opts to redeem loyalty points during a transaction.
Principal Use Scenario	1. The user initiates a purchase. 2. The system verifies the user's available loyalty points. 3. The system calculates the applicable discount based on the redeemed points. 4. The system applies the loyalty points to the order, adjusting the total cost.

	5. The user completes the transaction with the updated total.

Use case	Track Orders
Actor	Customer
Preconditions	● The customer has placed an order that is currently in process.
Trigger	The customer wants to check the status of their order.
Principal Use Scenario	1. The customer indicates that they want to track their order . 2. The system retrieves the real-time status of the order. 3. The customer sees updates such as “preparing” or “out for delivery”.

Couriers: Delivery Management

Use case	Receive Delivery Task
Actor	Courier
Preconditions	● The courier must be logged into the system and available to accept delivery tasks. ● There must be pending delivery tasks in the system.
Trigger	A new order is ready for delivery.
Principal Use Scenario	1. The courier wants to view available delivery tasks. 2. The system presents a list of deliveries that are currently waiting to be picked up 3. The courier selects a delivery. 4. The courier accepts the task in the system. 5. The system confirms the acceptance, updates the delivery queue and the order status.

Use case	Enter PIN to Confirm Delivery
Actor	Courier
Preconditions	● The courier must have arrived at the delivery location. ● The customer must have a unique PIN for the order.
Trigger	The courier is ready to deliver the order and requires

	confirmation from the customer.
Principal Use Scenario	1. The courier asks the customer for the delivery PIN. 2. The customer provides the PIN. 3. The courier enters the PIN into the system. 4. The system verifies the PIN. 5. The system confirms the delivery and marks the task as completed. 6. The system displays a success message and notifies the customer and restaurant that the delivery is complete.
Extensions	5a. If the PIN is incorrect, the system prompts the courier to re-enter the PIN.

Customers and Restaurants: Ratings

Use case	Rate Restaurant
Actor	Customer
Preconditions	● The customer must be logged into the system. ● The customer must have completed an order and received their delivery.
Trigger	The customer decides to leave a rating for the restaurant after an order.
Principal Use Scenario	1. The customer indicates that they want to rate the restaurant where they ordered from. 2. The customer provides a rating. 3. The system submits the rating. 4. The system displays a confirmation message to the customer.

Use case	View Feedback
Actor	Restaurant Manager
Preconditions	● The restaurant manager must be logged into the system.
Trigger	The restaurant manager wants to view feedback from customers regarding their service.
Principal Use Scenario	1. The restaurant manager indicates that they want to view their customers' feedback. 2. The system displays ratings from customers about

	the restaurant's orders.

Administrators: Admin Dashboard

Use case	Manage Accounts
Actor	Administrator
Preconditions	● The administrator is logged into the system with proper permissions.
Trigger	The administrator needs to manage user, restaurant, or courier accounts.
Principal Use Scenario	1. The administrator indicates that they want to manage user accounts. 2. The administrator searches for a user, restaurant, or courier account. 3. The system displays account details. 4. The administrator edits, disables, or deletes the account. 5. The system confirms and updates the changes.

Use case	Access Analytics
Actor	Administrator
Preconditions	● The administrator is logged into the system with appropriate permissions.
Trigger	The administrator wants to access system data and performance metrics.
Principal Use Scenario	1. The administrator indicates that they want to access the app analytics. 2. The system presents various data reports (order volumes, user activity, etc.). 3. The administrator selects a specific metric to explore further. 4. The system displays detailed statistics and graphical representations.

7.3. Conceptual Model

When designing an information system, conceptual modeling captures and organizes the essential knowledge the system needs to function. [29] It defines key entities, relationships, and rules within the problem domain, independent of technical details.

7.3.1. Conceptual Schema

A conceptual schema is the representation of the significant concepts (objects) within a problem domain [30]. It primarily shows:

1. **Object classes**: define the main entities in the domain.
2. **Associations between object classes**: represent the relationships between different entities.
3. **Attributes of the object classes**: define the properties of each entity.
4. **Integrity constraints:** include both graphical and textual rules ensuring the consistency and validity of the data.

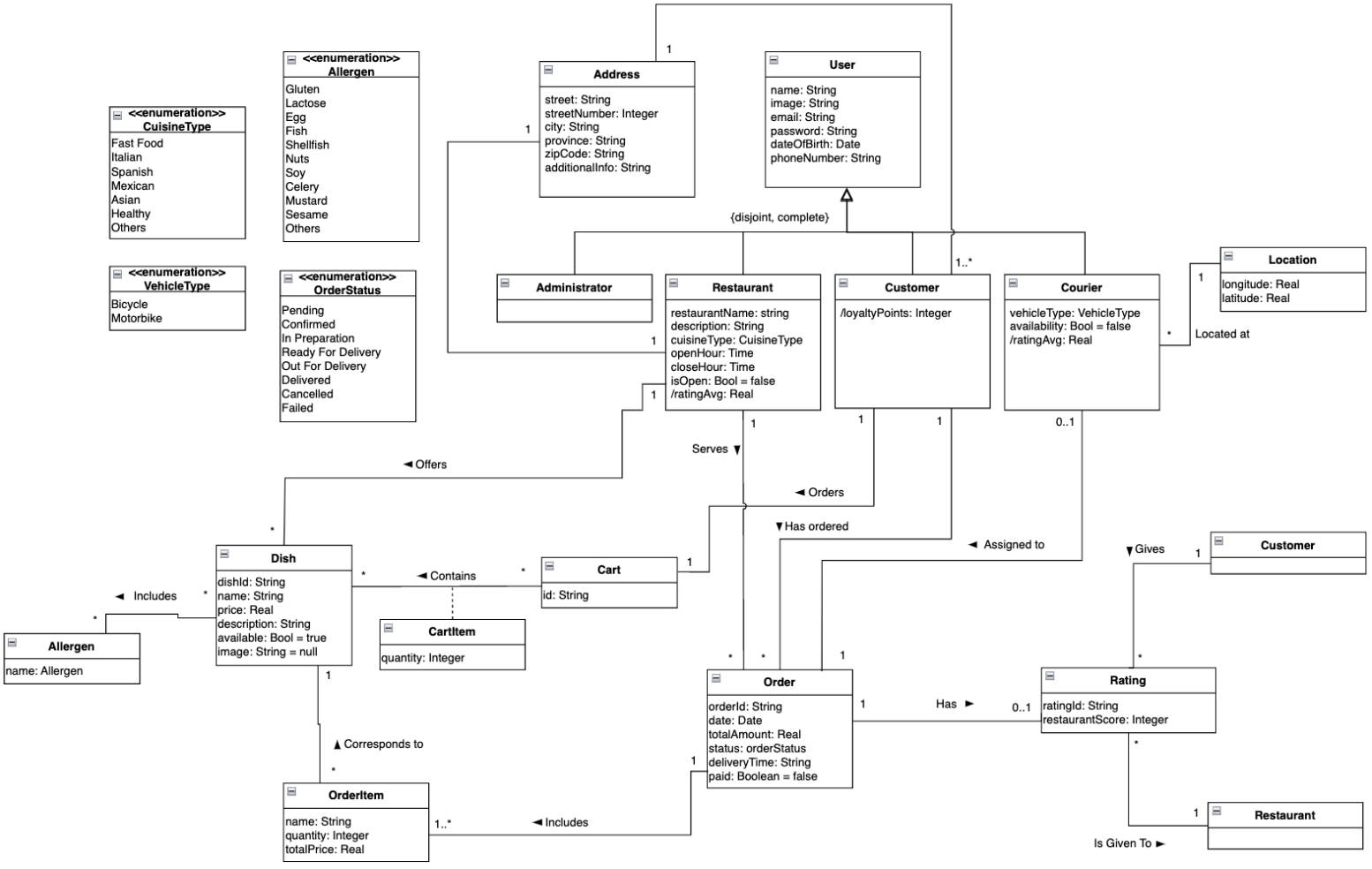


Figure 10. Conceptual Schema.

In Figure 10 we can see the conceptual schema of our application.

7.3.2. Integrity Constraints

Integrity constraints are critical rules applied to the data within the system to maintain its accuracy, consistency, and validity. These constraints ensure that the data adheres to specified requirements, preventing invalid entries and maintaining the integrity of relationships among entities.

External Keys: (**User**, email), (**Location**, courierEmail) (**Menu**, menuId), (**Dish**, name), (**Ingredient**, name), (**Allergen**, name), (**Order**, orderId), (**Rating**, ratingId), (**Review**, ratingId)

RT1: The *ratingAvg* of a Restaurant and Courier represents the average score they have received across all ratings they have received.

RT2: A Customer's *loyaltyPoints* are calculated based on the orders they have placed and their respective *totalAmount* values.

RT3: The Restaurant receiving a Rating must be the one associated with the dishes in the ordered cart.

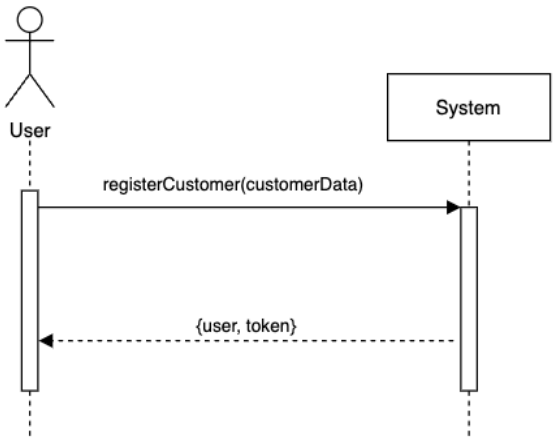
RT4: *restaurantScore* must be a number between 1 and 5.

RT5: All the Real or Integer attributes must have a value greater than zero.

7.4. Behavioral Model

The behavioral model focuses on the dynamic aspects of the system, illustrating how the system behaves in response to various events and interactions. Due to the project's scope, this section only provides an overview of the core events and functionalities that define the application's essential operations [31]. The remaining diagrams can be found in Annex B.

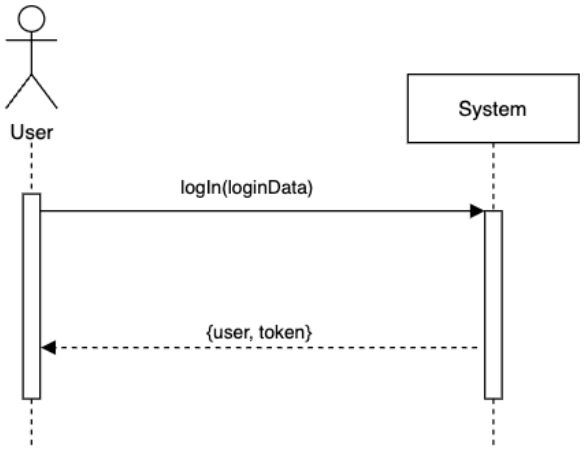
Register Customer



customerData	name: String, lastName: String, email: String, password: String, phoneNumber: String, address: String, image: String, datoOfBirth: Date
Precondition	-
Postcondition	The system creates a user with the data received
Body	User registered and authentication token

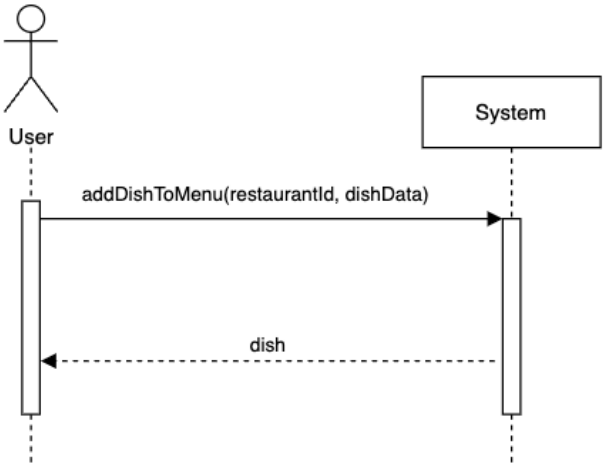
This sequence diagram would be the same for registering restaurants or couriers; the only difference is the data in the parameters, which corresponds to the attributes defined in the conceptual schema.

Log In



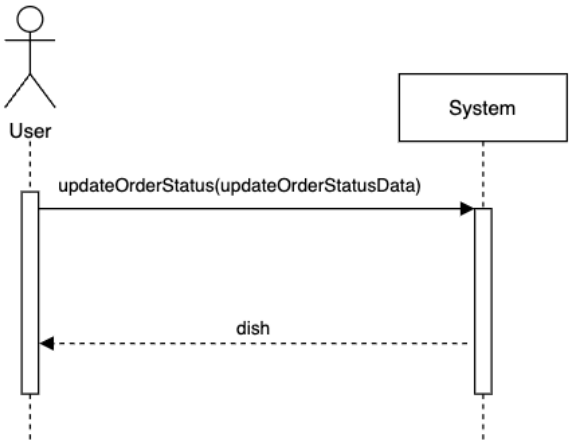
loginData	email: String, password: String
Precondition	-
Postcondition	The user is logged in
Body	User registered and authentication token

Add Menu Dishes



dishData	name: String, price: Real, description: String, image: String, allergens: Array[String]
Other Parameters	restaurantId: String
Precondition	The *restaurantId* corresponds to a restaurant registered. Allergens are valid (enumeration)
Postcondition	A dish with the provided details is added to the menu of the restaurant corresponding to the restaurantId
Body	Dish created.

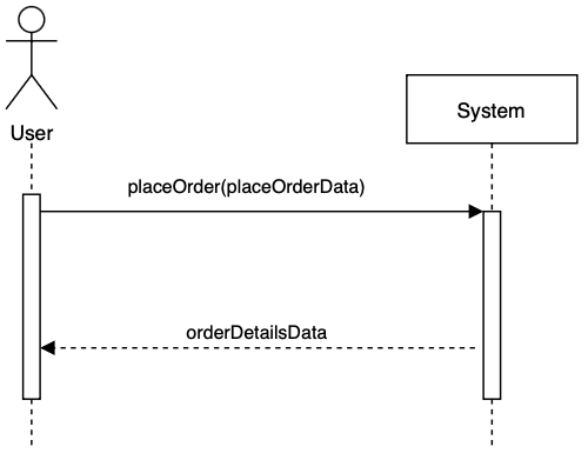
Update Order Status



updateOrderStatusData	orderId: String, newOrderStatus: orderStatus, courierId: String {optional}

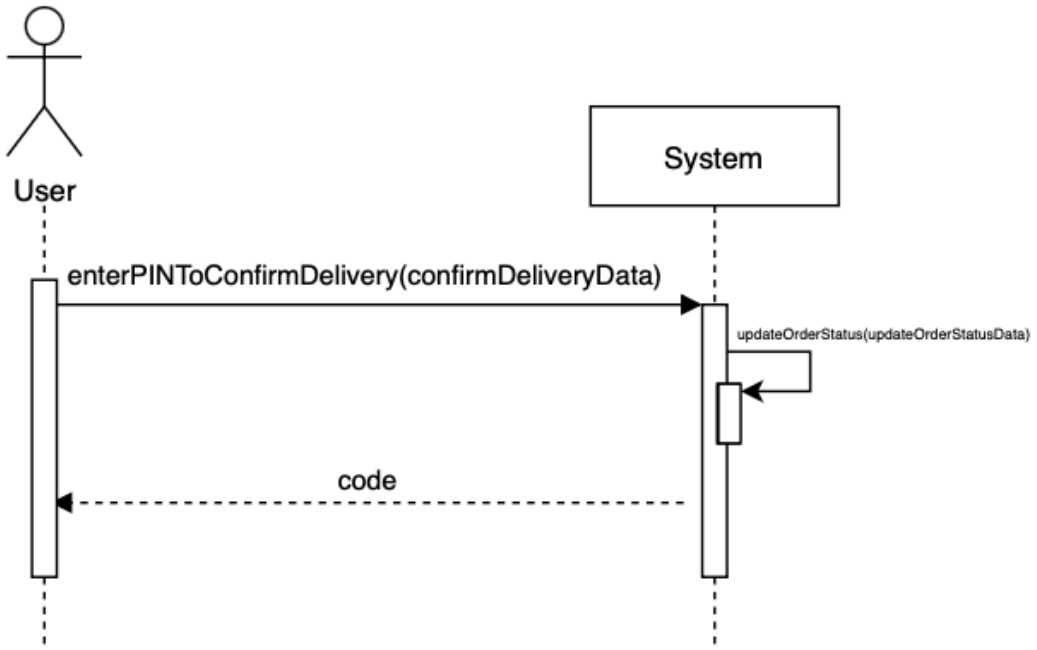
Precondition	The *orderId* corresponds to an order that exists in the system and is currently in process.
Postcondition	The status of the order has been updated. Depending on the new status a courier can be assigned to the order as well.
Body	Dish updated.

Place Orders



placeOrderData	customerId: String, restaurantId: String, restaurantName: string, dishes: Array[String]
Precondition	The *customerId* exists in the system, and the dishes are available. The *restaurantId* exists in the system. *totalAmount* is a positive number that represents the sum of the dish prices.
Postcondition	An order has been created and assigned to a customer and a restaurant.
Body	*orderDetailsData* contains all the details of the order that has been created.

Enter PIN to Confirm Delivery



confirmDeliveryData	orderId, pinCode
Precondition	*orderId* exists in the system and corresponds to an order with status "*Out for Delivery*"
Postcondition	The order status changes to "*Delivered*"
Body	Code indicating whether the operation has succeeded or not.

8. Design

The Design chapter of this thesis outlines the architectural and technical choices we have adopted. By leveraging a microservices architecture, our system will decouple core functionalities — such as order management, user authentication, and payment processing — into independent services. This allows each service to be developed, deployed, and scaled separately.

This approach has been chosen to ensure that all our non-functional requirements are met while implementing the functional ones. In this chapter, we detail the design considerations for each component, the interactions between services, and the best practices employed to achieve our goals.

8.1. Why are we using a microservices architecture?

According to *Designing Data-Intensive Applications* [32], this application is classified as *data-intensive* due to its primary challenge of managing large volumes of data. While designing an application for millions of users from the start is often considered premature optimization, this project aims to demonstrate how focusing on data management and robust software design can ensure strong performance, even under heavy system load.

Next, we will introduce the architecture we plan to use, highlighting its trade-offs in relation to the non-functional requirements we aim to prioritize. Based on *Building Microservices* [33] by Sam Newman, a key resource for this and the following chapters, we will explore the advantages of the microservices architecture.

Reliability
Microservices architecture enables us to manage the failure of individual services dynamically based on specific circumstances, thereby preserving the overall system functionality undisturbed. It is needless to say that loose coupling is a must to achieve this.

When a service fails, the impact is isolated to that particular instance. This isolation allows for more granular fault tolerance strategies, such as retry logic, circuit breakers, or fallback mechanisms.

Scalability
We have the flexibility to scale only the services that require additional resources, rather than the entire system. This allows a more efficient use of computing resources. Services that experience higher demand can be allocated more instances, while less critical or less frequently used services can operate on resources suited to their needs.

This selective scaling not only optimizes performance but also reduces infrastructure costs, which are an important consideration in this type of architecture.

Maintainability

Smaller teams working on more focused codebases tend to experience higher productivity. Each service should be designed to a size manageable by a 2-pizza team, that is, a team small enough to be fed by two pizzas. This separation makes the codebase easier to understand and maintain for the people working on it. As a result, developers can work more efficiently, since they can focus on a smaller scope without the need to navigate through a large, monolithic structure, and without being slowed down by dependencies on other parts of the system.

On the other hand, since different technologies can be used for each microservice based on specific needs, it is crucial to strike a balance between the breadth and complexity of the technologies employed and the costs associated with a diverse technology stack. Utilizing a wide array of technologies can offer significant advantages but may also introduce complexity and increase the learning curve for the development team. The time required to understand and integrate new technologies can lead to longer development cycles and it can delay the delivery of features to clients.

Disclaimer: We will use the terms “microservice “and “service” interchangeably from now on. Any distinction between the two will be explicitly stated when necessary.

8.2. Architecture

Figure 11 illustrates the technologies involved in the architecture. There are two distinct areas to address: the front-end, which encompasses what the user sees and interacts with, and the back-end, where the data requested and sent by the front-end is processed. We will adopt a **client-server model**, with the front-end serving as the client and the back-end as the server. The following chapters will examine the details and logic of both the front-end and back-end.

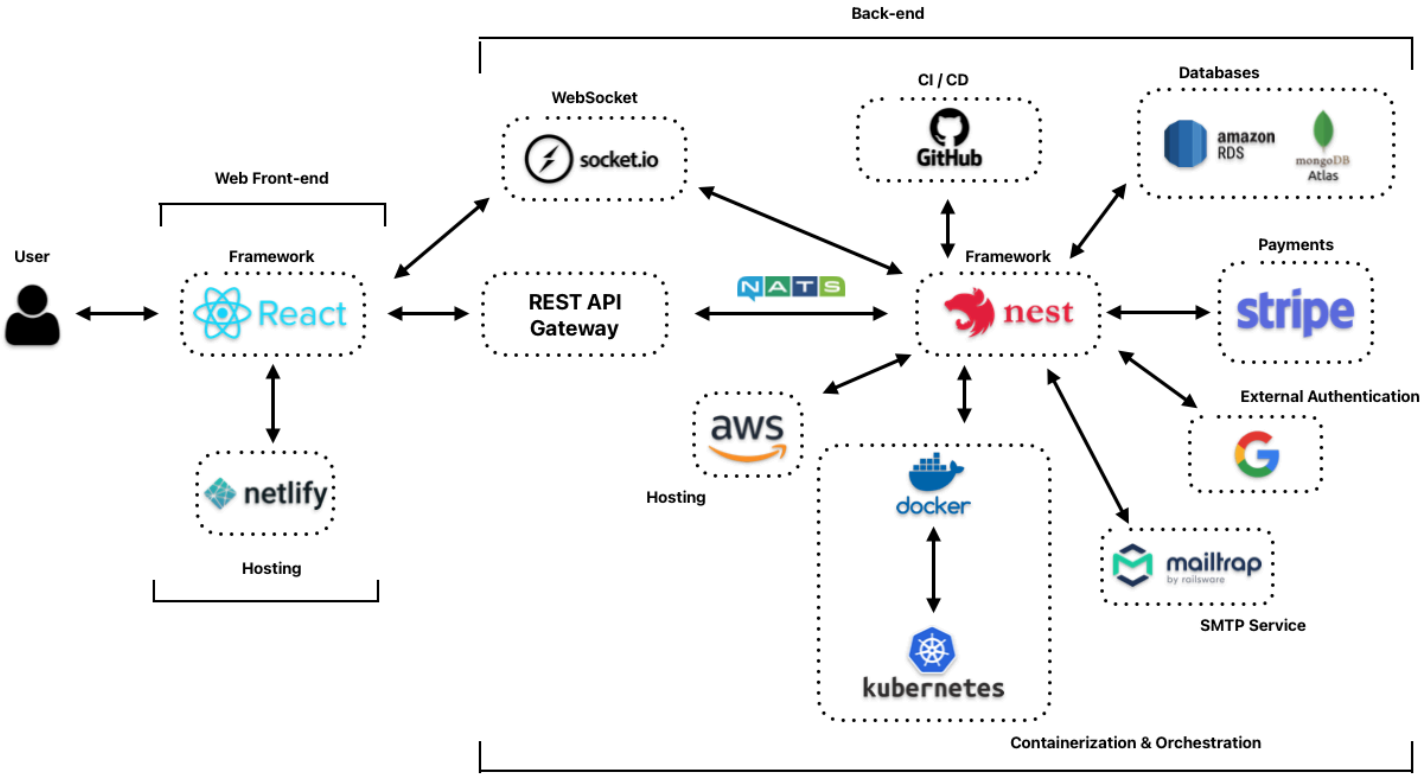


Figure 11. Technical Architecture Overview

On the front-end, *React* delivers an interactive user interface, enabling users to engage with the system. When users initiate requests, a REST API gateway facilitates data retrieval by querying the appropriate back-end microservices (**API Gateway Pattern**). Real-time updates are supported through multiple *WebSocket* channels, powered by *Socket.io*.

For inter-service communication, we use *NATS* as the message broker. *NestJS* handles back-end operations, processing requests and executing the necessary business logic. Data storage is managed by *MongoDB* and a *RDS* (PostgreSQL), ensuring reliable access to the application data. Payment processing is securely handled by *Stripe*, which simplifies transactions and eliminates the need for us to manage sensitive financial information.

The system is containerized using *Docker*, ensuring consistent deployments across environments. *Kubernetes* orchestrates these containers, providing scalability and reliability. *AWS* serves as the cloud infrastructure, offering robust computing resources (*EC2*) and image

storage via *S3*. *GitHub Actions* automates our *CI/CD* pipeline, enabling efficient code deployment and updates.

To enhance the user experience, Google authentication is integrated via *Firebase*, offering a faster and more convenient registration and login process. Lastly, *Mailtrap* is used as an SMTP service to send email notifications to our customers.

8.2.1. Front-end

For the front-end, the application will adopt a **Model-View-Controller (MVC)**-inspired approach to ensure a clear separation of concerns and enhance maintainability; a simple diagram is provided in Figure 12. With React as the front-end framework, the **View** is represented by React components, which render the user interface and handle interactions with the user.

The **Controller** role is managed within the React ecosystem by using event handlers, state management (via hooks or libraries like Redux), and API calls to process user inputs and update the interface dynamically. React's declarative nature simplifies this process, making the user interface more predictable and easier to manage.

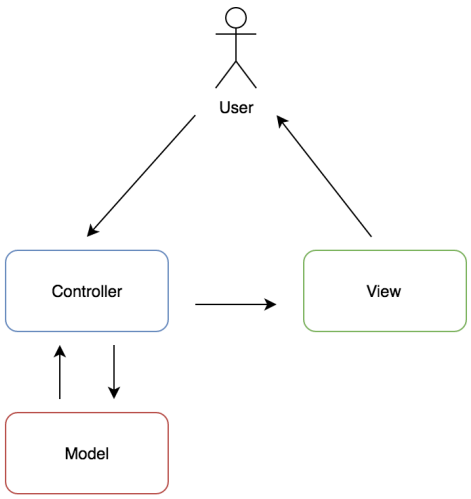


Figure 12. MVC Pattern. [34]

The **Model** resides in the back end, acting as the single source of truth for application data and business logic. It provides APIs for the front-end to fetch, update, or modify data as required. This separation ensures that the React front-end remains focused on delivering a good user experience, while data management, storage, and processing tasks are delegated to the back-end services.

8.2.2. Back-end

For the back-end side, as mentioned before, we will employ a microservices architecture, enabling us to develop and deploy numerous microservices, each tailored to handle specific functionalities within the overall system. We will use these elements to draw the diagrams.

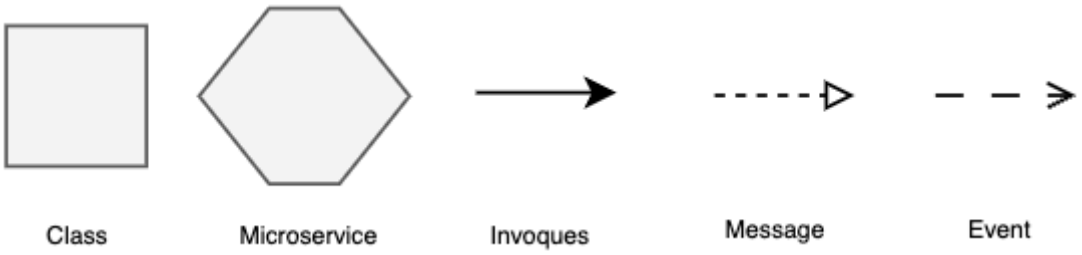


Figure 13. Description of the elements used in the diagrams.

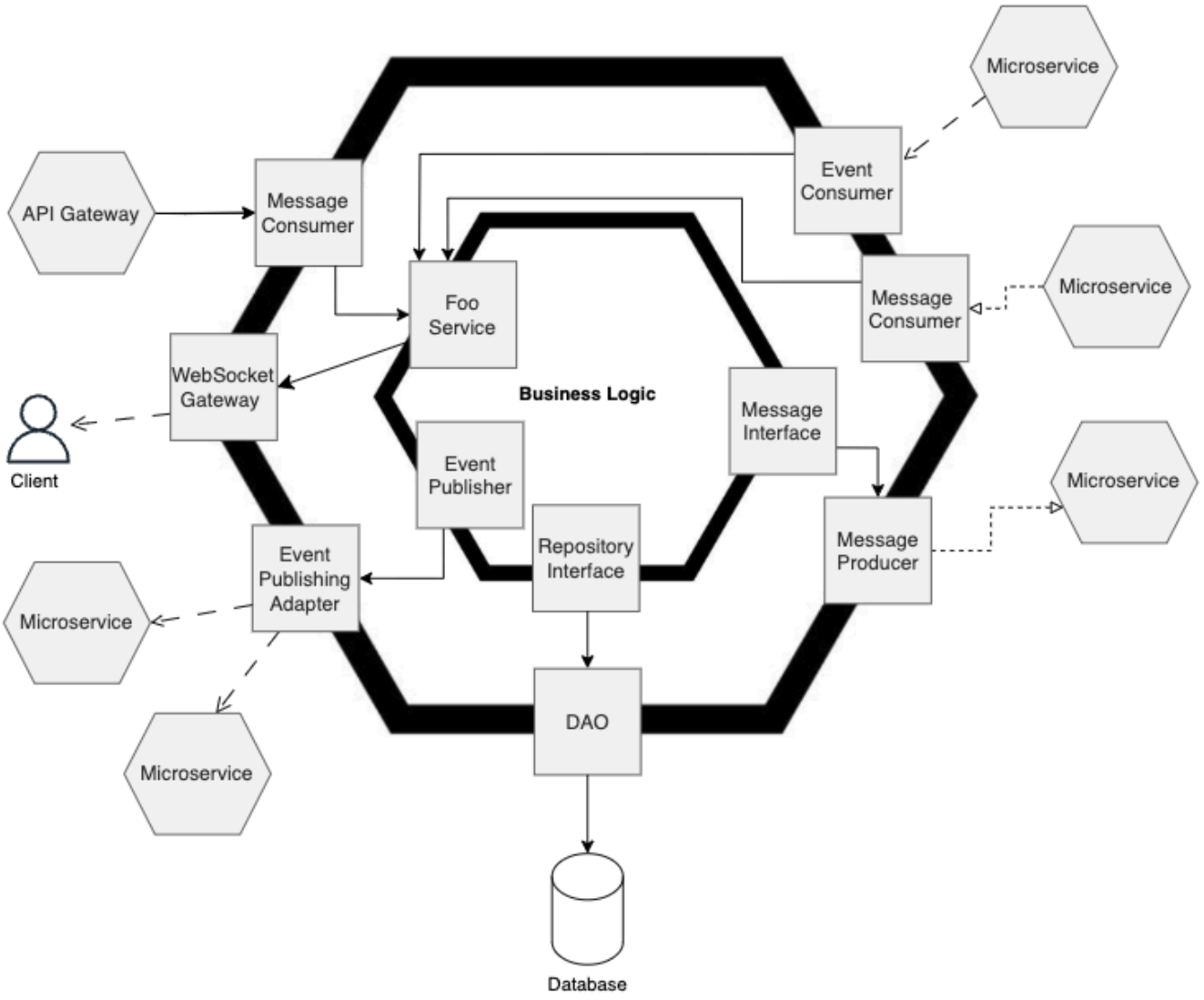


Figure 14. Standard Microservice Logical Architecture. [35]

Each microservice will be implemented using a **hexagonal architecture** (Figure 14), also known as the **ports and adapters pattern**. This architectural design separates the core business logic from external concerns, improving maintainability, scalability, and testability.

In a hexagonal architecture, each microservice is structured with distinct ports and adapters. **Ports** define the entry points into the core business logic, specifying operations in an abstract, technology-agnostic manner. **Adapters**, in turn, implement the interfaces defined by the ports, acting as bridges between the core logic and external systems such as user interfaces, third-party APIs, or messaging systems. This separation ensures that changes to external systems or technologies do not affect the business logic.

For data persistence, **Data Access Objects (DAOs)** will encapsulate database interaction logic. DAOs provide an abstract interface to the data layer, shielding the application from the specifics of database operations. This pattern allows each microservice to manage its database interactions independently, ensuring clean and reusable code.

To enable efficient and decoupled communication between microservices, we will adopt **event sourcing** and **message-based communication** mechanisms. These approaches propagate state changes and domain events asynchronously, enhancing fault tolerance and system responsiveness. Inter-service communication will also benefit from WebSocket-enabled gateways where real-time updates are required. **WebSockets** ensure that events or state changes are pushed immediately to connected clients or downstream systems, enabling low-latency, real-time interactions.

For managing complex workflows involving multiple services, we implement **saga patterns** for distributed transaction management.

- **Orchestrated sagas** rely on a central coordinator to control the flow of events, ensuring consistency and compensating for failures in a predictable manner.
- **Choreographed sagas**, in contrast, allow services to communicate and decide their actions autonomously by consuming and reacting to events, promoting scalability and loose coupling.

8.2.2.1. Logical Architecture

To enhance this architecture, we have adopted **Domain-Driven Design** (DDD) principles, which allow us to separate business logic based on relevant business terms and concerns. This approach not only aids in the identification of distinct microservices but also ensures a clear separation of concerns, thereby fostering independence among services and significantly reducing or eliminating coupling.

To further illustrate the application of DDD principles, Figure 15 presents the conceptual schema with the identified aggregates. These aggregates represent cohesive groupings of related business entities, delimited based on the boundaries defined by the core business logic.

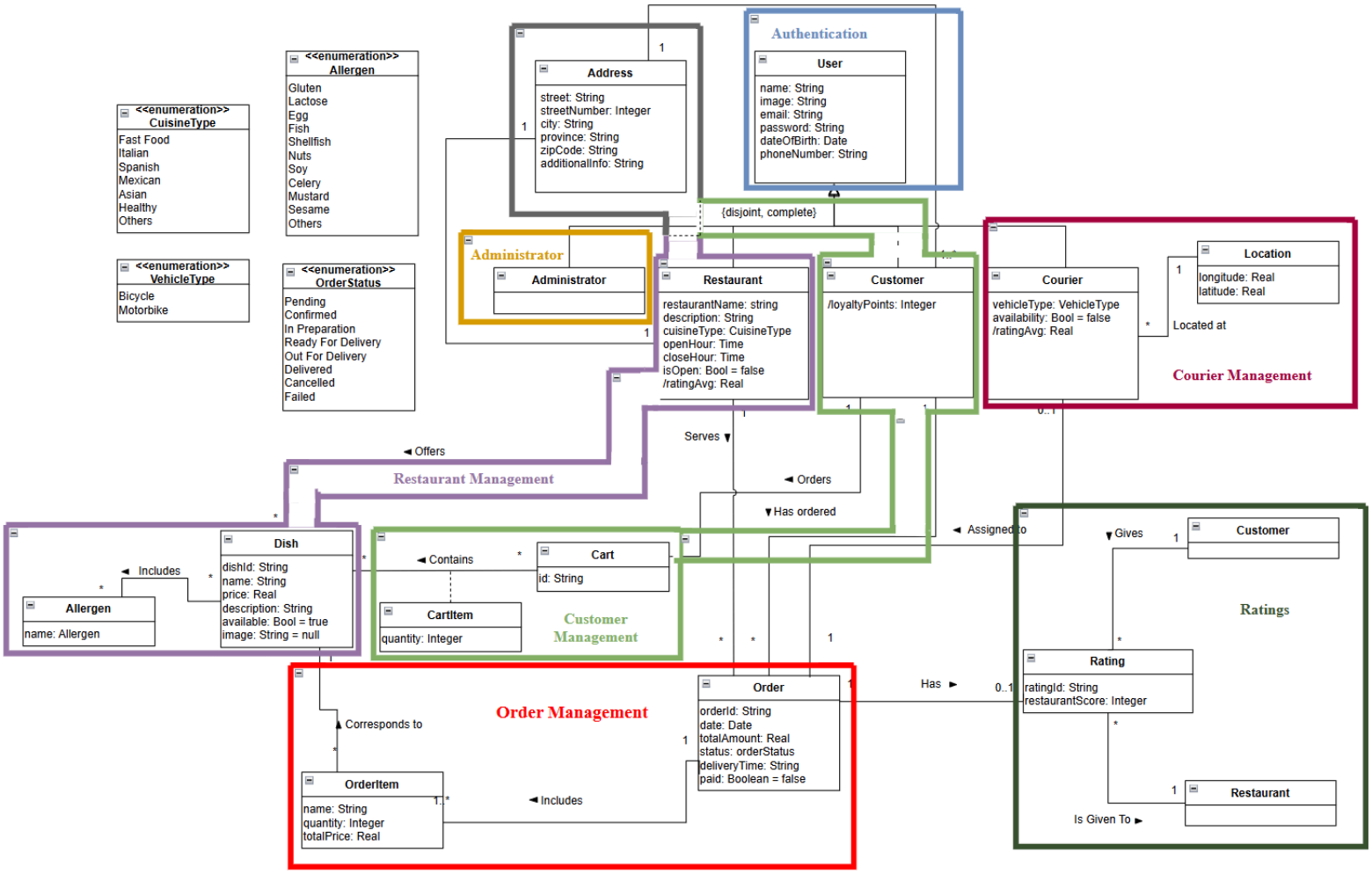


Figure 15. Aggregates based on the conceptual schema.

Now, let's list the aggregates that correspond to the food delivery business domain, with each aggregate mapping to a microservice.

Authentication

The Authentication service is responsible for managing all aspects of user authentication. This includes user registration, login, and logout processes, as well as handling password resets and securely generating authentication tokens. Figure 16 illustrates the logical architecture of the Authentication Microservice.

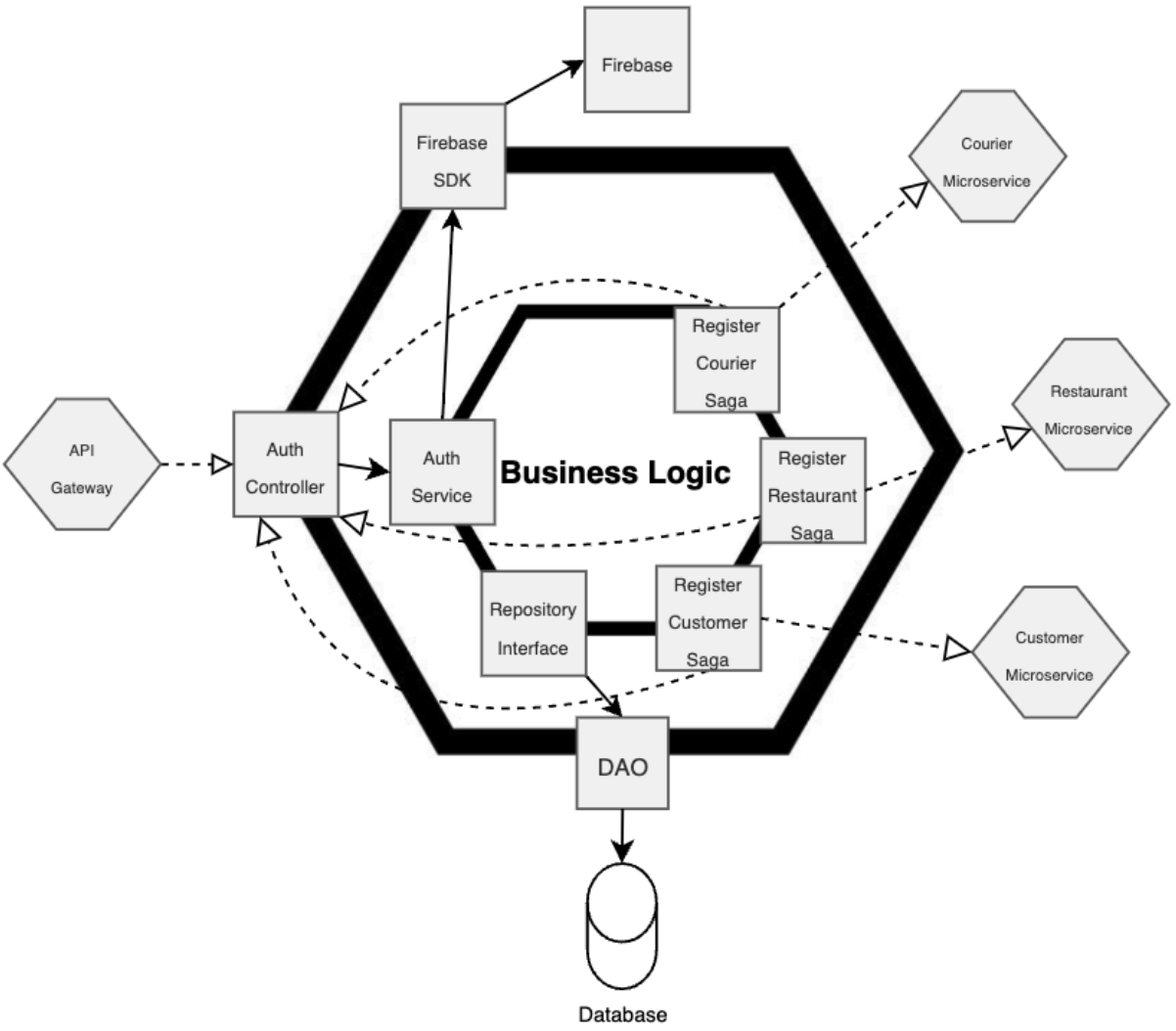


Figure 16. Authentication Microservice Logical Architecture.

Payment Processing

The Payment Processing service handles all financial transactions within the system. It processes payments for orders and loyalty points during checkout. To securely process payments, we will integrate Stripe, delegating all payment-related tasks to Stripe's robust platform, which handles card transactions. It follows a similar logic as the Authentication service; however, in this case, it will interact with Stripe as an external actor. In addition, it emits events as the order payment status changes. Figure 17 illustrates the logical architecture of the Payments Microservice.

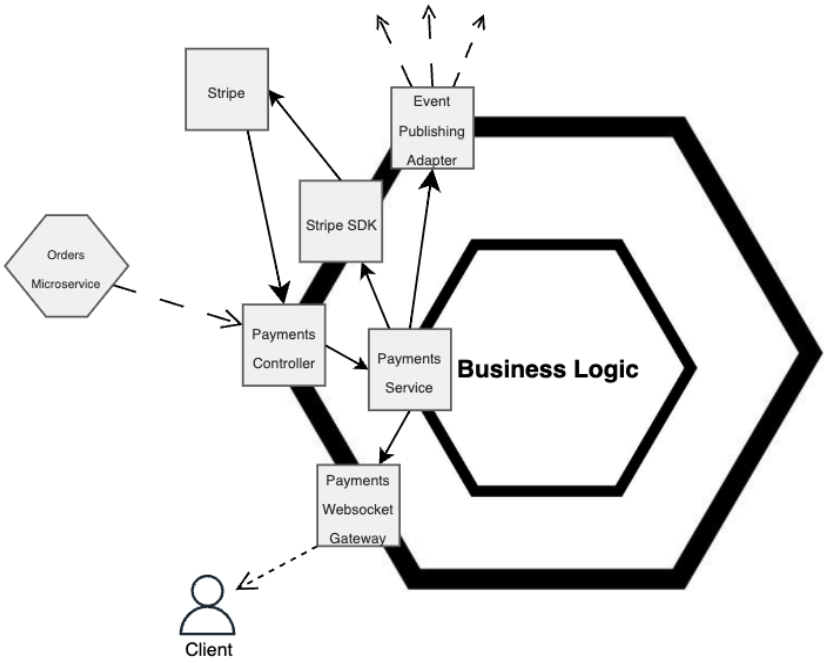


Figure 17. Payments Microservice Logical Architecture.

Customer Management

The Customer Management service is responsible for managing all customer-related data and interactions. This includes creating and updating customer profiles, managing personal information and addresses, handling loyalty program participation, and maintaining a history of past orders. Figure 18 illustrates the logical architecture of the Customer Microservice. It also listens to events from the Orders Microservice to act accordingly.

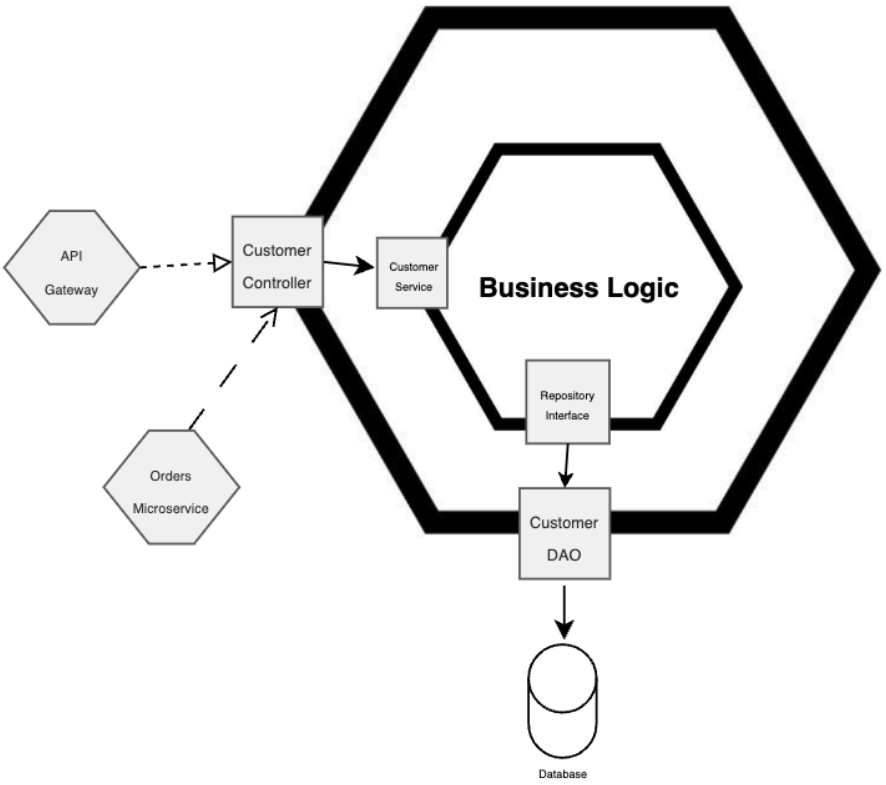


Figure 18. Customer Microservice Logical Architecture.

Restaurant Management
The Restaurant Management service oversees the creation, configuration, and maintenance of restaurant profiles within the platform. This includes managing restaurant details such as menus, pricing, available items and opening hours. This service will follow a similar logic as Customer Management service.

Courier Management
The Courier Management service is responsible for managing delivery drivers (couriers) and their interactions with the system. It includes onboarding couriers, tracking their availability and assigning deliveries. The service also handles courier ratings. This service will follow a similar logic as Customer Management service and it receives events from the Orders Microservice to update its status regarding its order assignment.

Delivery Management
The Delivery Management service is responsible for managing the logistics of order deliveries. This microservice listens to events generated by the Orders Microservice and processes them accordingly. It replicates these events through its WebSocket server, ensuring that couriers, customers, and restaurants connected via the front-end receive accurate instructions and real-time updates.

User Management (Administrator)
The User Management (Administrator) service is responsible for overseeing administrative tasks, including managing access control within the platform.

Order Management
The Order Management service oversees the entire lifecycle of customer orders. Its responsibilities include order creation, modification, cancellation, and status tracking. It produces events to notify other microservices subscribed to order status changes. Additionally, it listens to events from the Payments Microservice to update order statuses based on payment outcomes.

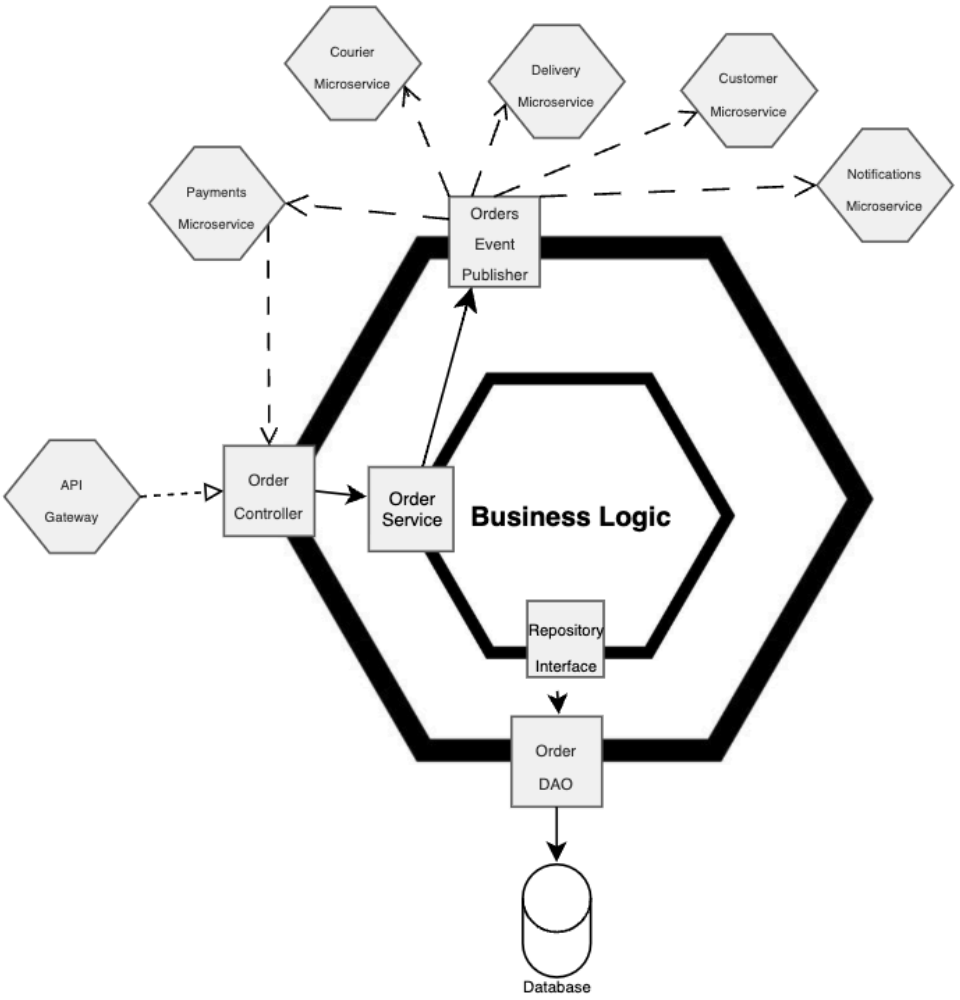


Figure 19. Order Microservice Logical Architecture.

Ratings

The Ratings service enables customers to rate and provide feedback on restaurants and delivery experiences. It will interact with the Restaurant and Orders Microservice.

Notifications

The Notifications service is responsible for sending real-time alerts and updates to customers. This includes order confirmations and delivery status updates. The service ensures that notifications are delivered via email. It subscribes to the events channels and acts accordingly.

Analytics and Reporting

The Analytics and Reporting service collects, processes, and analyzes data to generate actionable insights. It offers reports on customer behavior, sales trends, restaurant performance, delivery efficiency, and financial metrics. It subscribes to the events channels and saves the data.

8.3. Front-end Design

The front-end design focuses on creating an intuitive, responsive, and user-friendly interface for the application. This section outlines the navigation structure, core screen designs, internal architecture, and component functionality.

8.3.1. Navigational Map

The navigational map provides a clear overview of the application's structure, illustrating the relationships between its screens and user flows. We will provide a map for each role, as each has distinct workflows, although they share some functionalities, such as registration, login, and profile management.

8.3.1.1. Customer

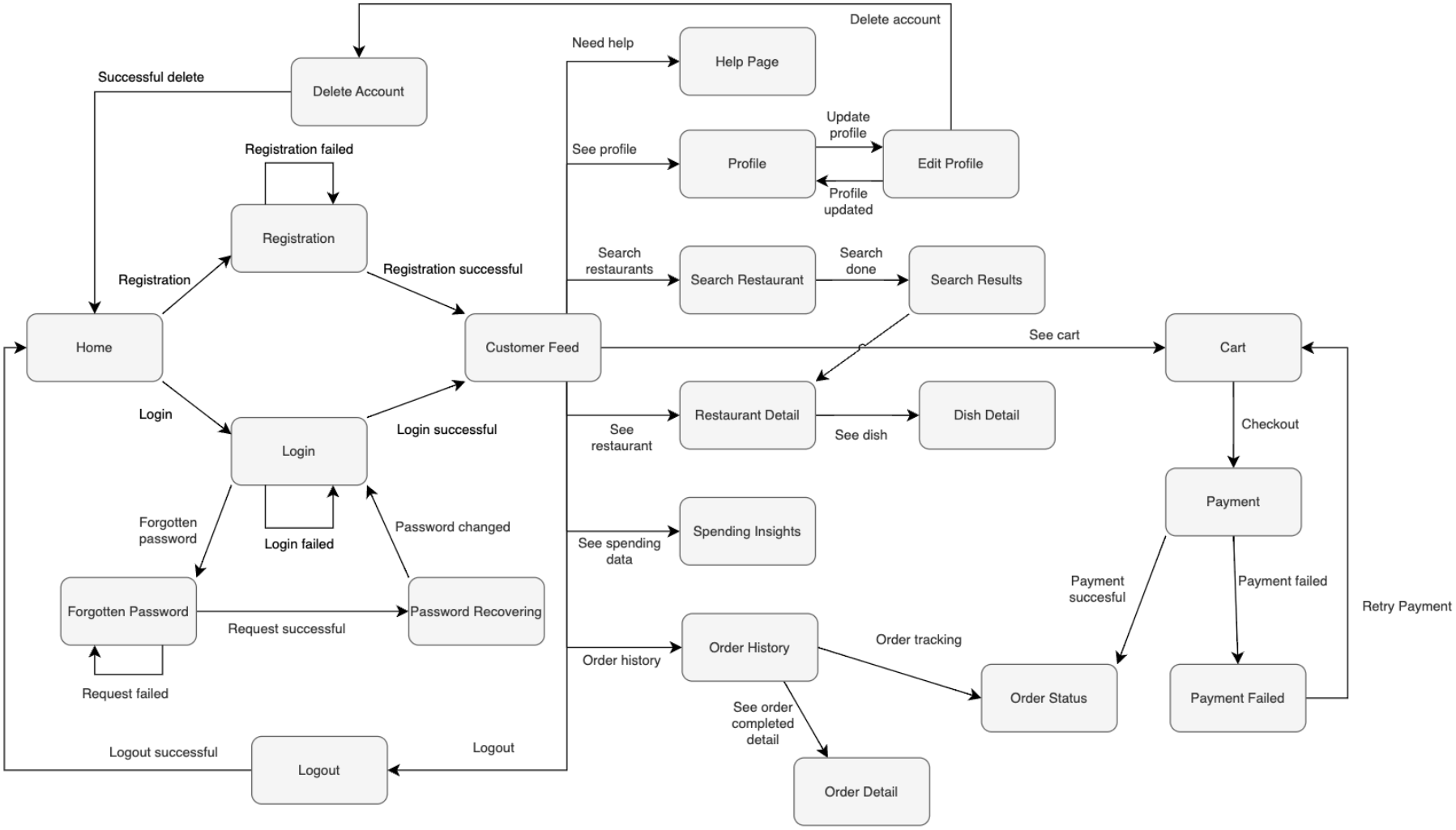


Figure 20. Customer navigational map.

8.3.1.2. Restaurant

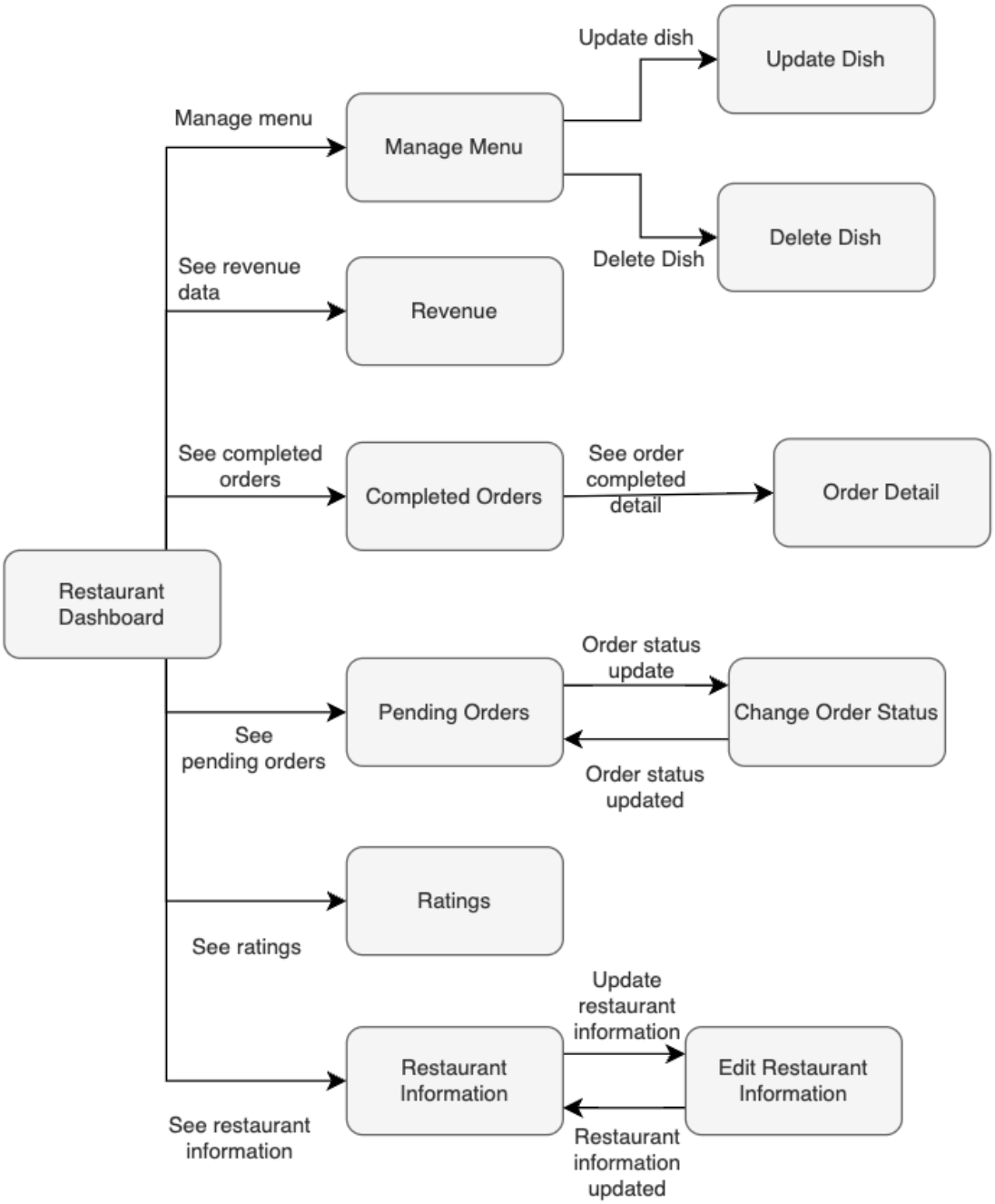


Figure 21. Restaurant navigational map.

We are omitting registration, login and profile management screens as they are the same as the customer's.

8.3.1.3. Courier

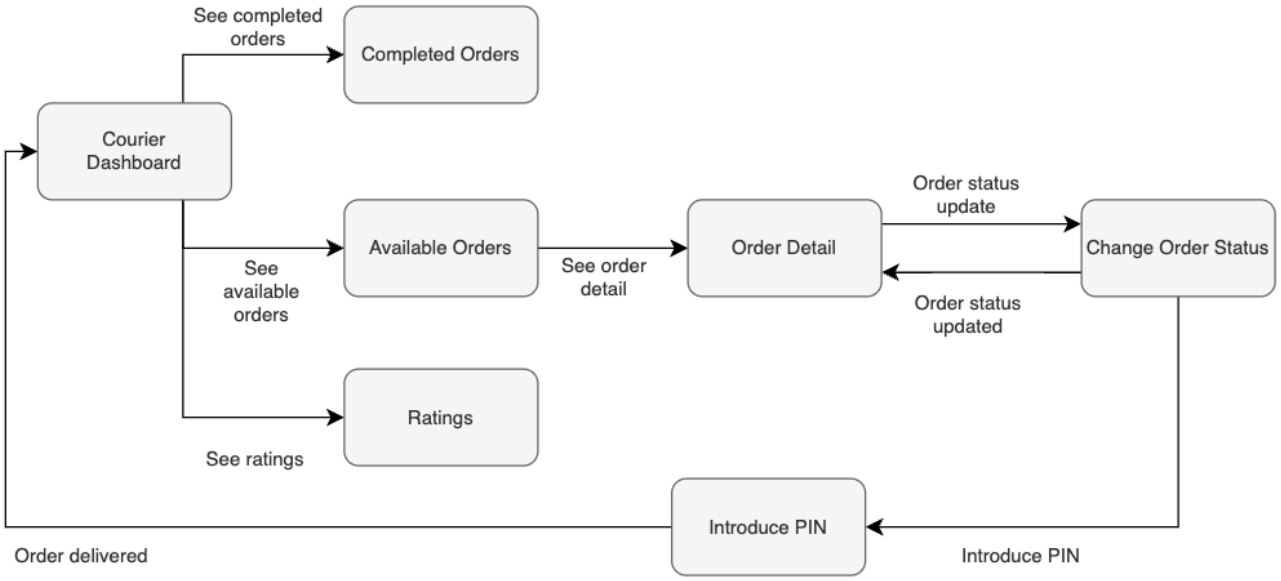


Figure 22. Courier navigational map.

We are omitting registration, login and profile management screens as they are the same as the customer's.

8.3.2. Core Screen Designs

This section presents the key screens that form the foundation of the application's user interface. The mockups shown here may differ from the final version of the web application after implementation. Additionally, only selected core screens are showcased in this section.

The header is a universal element present on all screens, adapting to different user roles by displaying relevant icons—for example, customers see a cart and a search bar, while other roles only have the menu icon. The menu icon, always visible, opens a sidebar for quick access to features and options. Between the header and footer lies the main content area, which dynamically changes based on the screen's purpose. The footer, consistent across all roles, ensures uniformity and includes a set of internal links, helping users navigate to specific parts of the web application. Additionally, the footer includes links to Food To Door's social media accounts. However, in this context, these links are non-functional as the enterprise is fictitious.

If the user is not logged in, the header displays a login button, directing them to the login page, and a register button, which leads to the customer registration page.

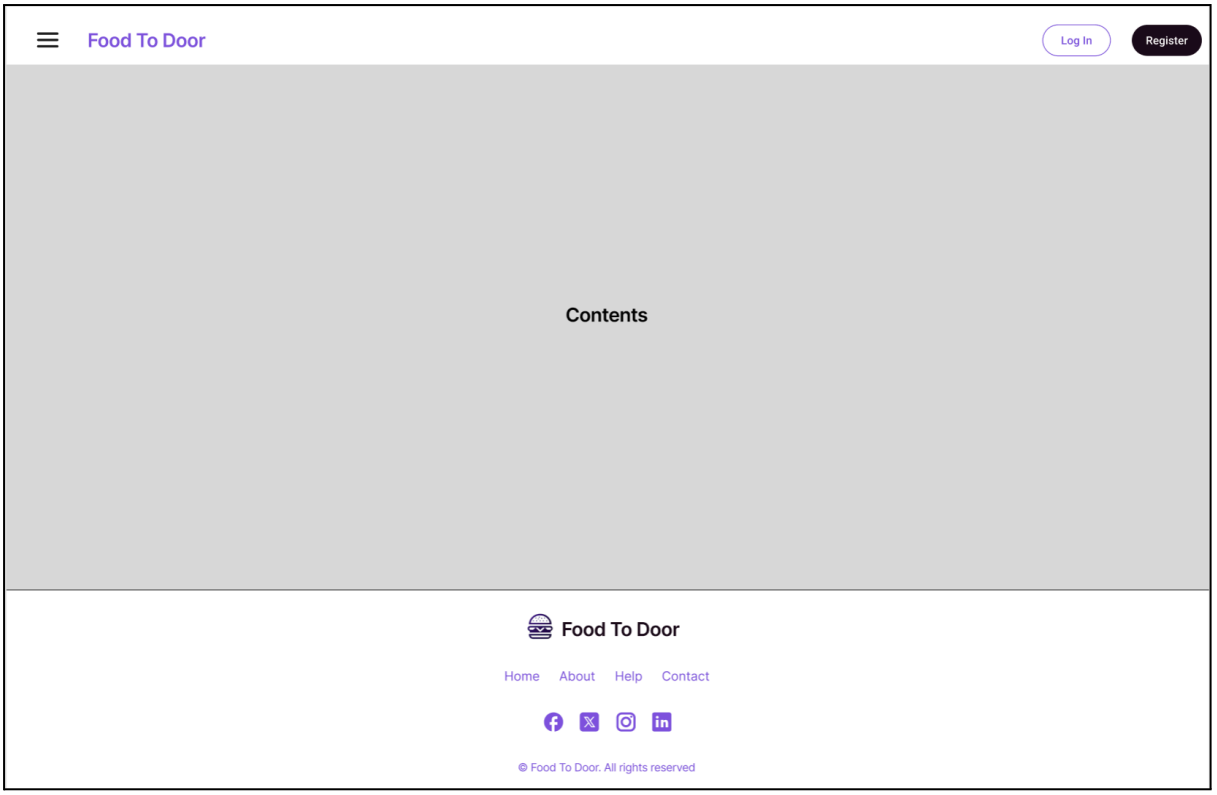


Figure 23. Web's core layout.

When the menu icon is clicked, a sidebar slides out from the left side of the screen, displaying options tailored to the user's role or login status. If the user is not logged in, the sidebar provides the options to register as a restaurant or courier, along with a link to the help page. The help page includes frequently asked questions (FAQs) to assist users in navigating and understanding the platform.

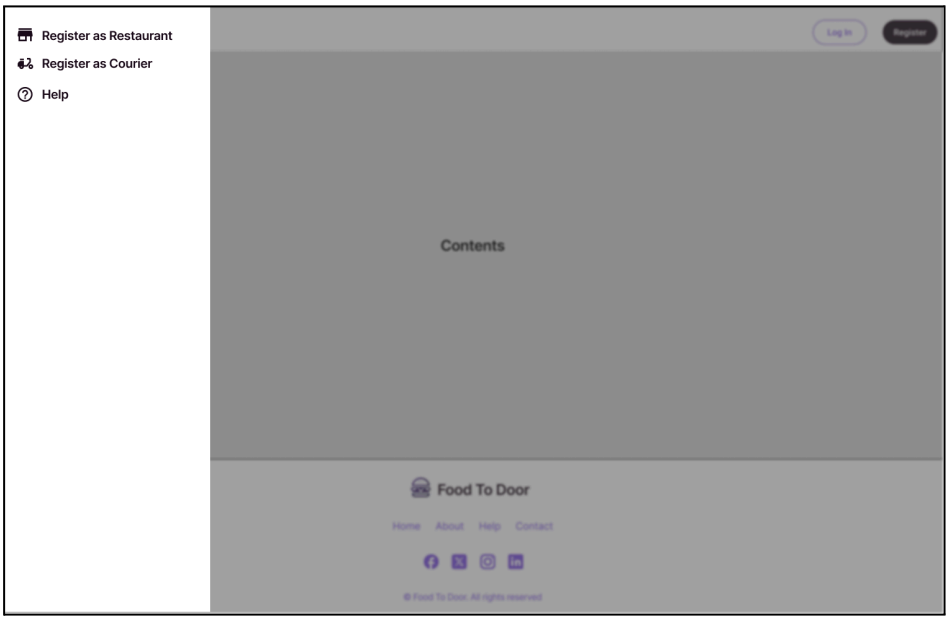


Figure 24. Sidebar for not signed-In users.

The *Login* page provides a form for entering an email and password for native login. Alternatively, users can choose to log in with Google, offering a quick authentication option. If it's the first time logging in with Google, the platform redirects the user to the customer registration page, pre-filling certain fields with information retrieved from their Google account.

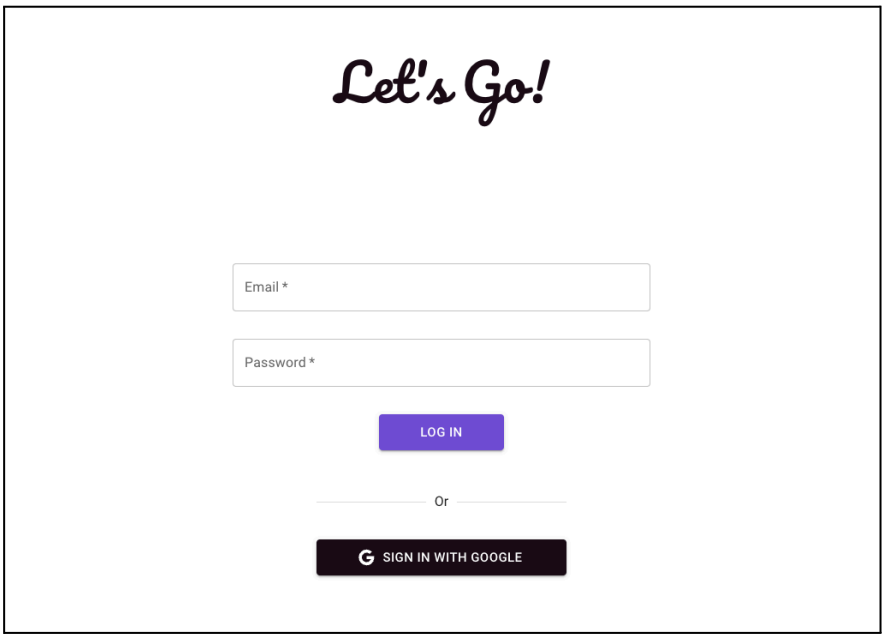


Figure 25. Login page.

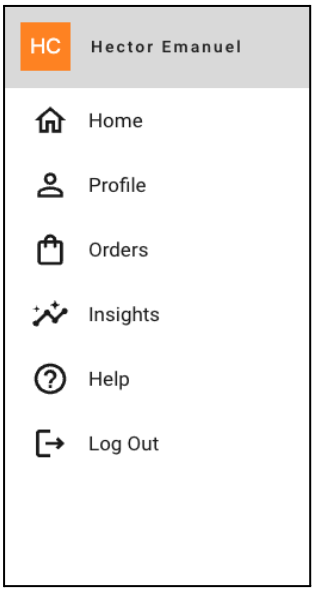


Figure 26. Customer sidebar.

As mentioned earlier, the sidebar dynamically adjusts its options based on the user's role upon login. In this mockup, the sidebar displays the options available to a customer, which include features like seeing the orders associated with the customer, managing profile, and accessing help. If the logged-in user is a restaurant or courier, the sidebar simplifies to only show "Dashboard," "Help," and "Log out" options.

The *Customer Feed* page serves as the "home" screen, providing users with a visually engaging and personalized experience. At the top of the page, a carousel displays vibrant images of various restaurants, designed to capture attention and promote exploration. Below the carousel, a section showcases featured restaurants, selected based on their high ratings and order volumes.

Each restaurant card in this section provides key information, including an image, the restaurant's name, its average rating, and the total number of ratings received. This allows users to quickly identify popular and well-reviewed options.

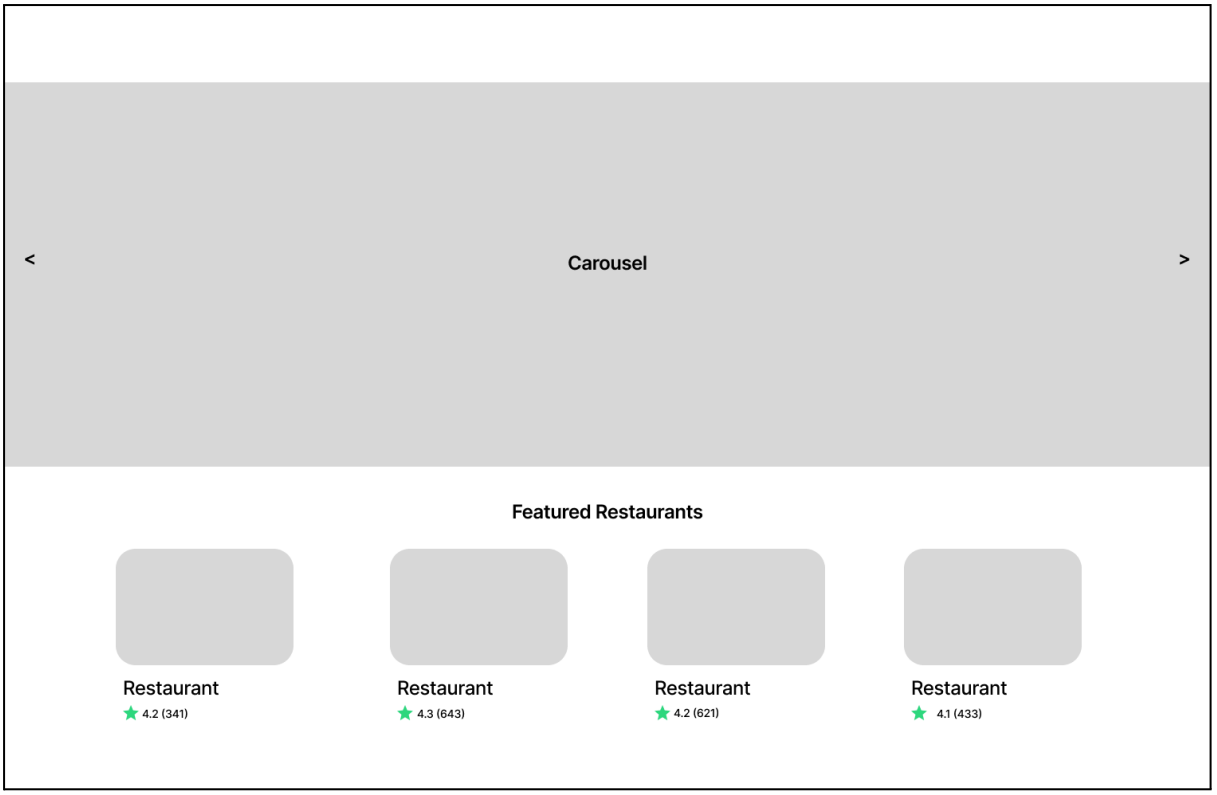


Figure 27. Customer Feed page.

The *Restaurant Detail* page provides a comprehensive overview of a restaurant and its offerings. At the top of the screen, a banner displays the restaurant's name, average rating, estimated delivery time, and courier expenses. Additionally, a search bar enables users to efficiently explore and search for specific dishes or products within the restaurant's menu.

Below the banner, the Featured Dishes section highlights the restaurant's promoted items. Restaurants have the ability to manage all menu sections and designate dishes for inclusion in the featured section.

Following the featured section, the menu is organized into clear categories such as Main Courses, Desserts, and more. Each category follows the same layout, so for simplicity, the mockup displays only one example.

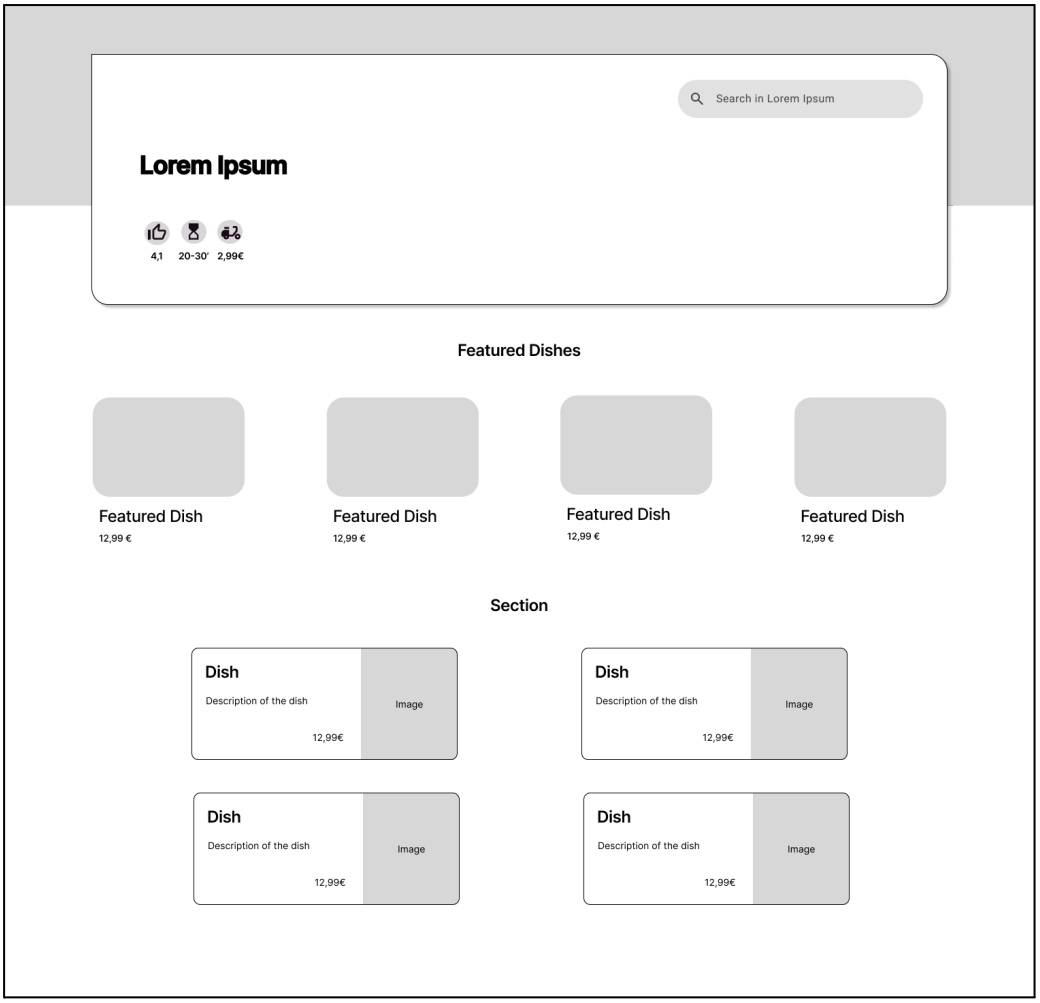


Figure 28. Restaurant Detail page.

For the *Cart* page, the empty cart page is displayed when no items have been added. It includes a button that redirects users back to the feed page, encouraging them to explore restaurants and start building their order.

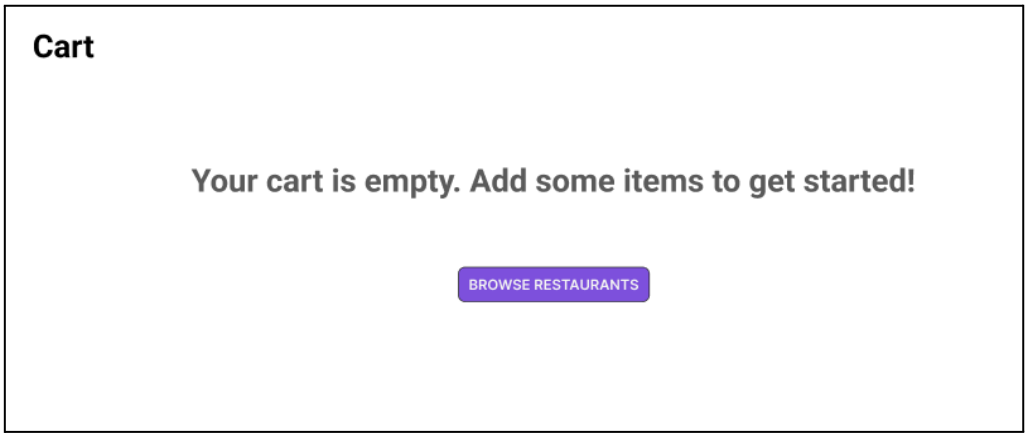


Figure 29. Cart page with an empty cart.

A non-empty cart displays a detailed list of the selected dishes. Each item includes the dish name, an image, the price, and a quantity modifier. The modifier allows users to adjust the quantity of a dish, and when the quantity is set to 1, the button changes to offer the option to delete the item from the cart. At the bottom, a checkout button redirects users to the Stripe payment page for secure payment processing.

A key rule ensures that the cart only contains items from a single restaurant. If a user attempts to add items from another restaurant, they must first clear their current cart to proceed.

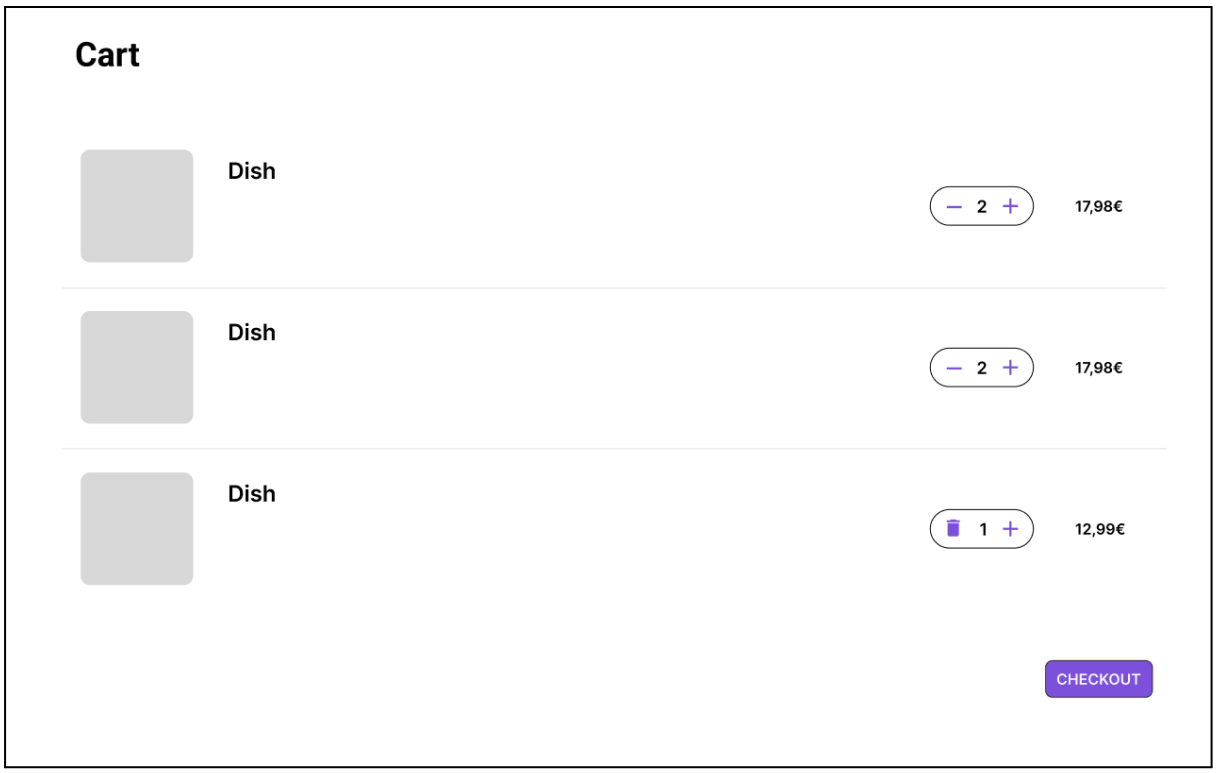


Figure 30. Cart page with a non-empty cart.

The *Order Status* page is designed to provide users with comprehensive information and updates regarding their orders. Users can access this page immediately after completing a successful payment or through the "See Orders" page. The order status is displayed in real-time, ensuring users are informed about their order's progress, from preparation to delivery. The page also provides an estimated delivery time, offering customers a clear idea of when their order will arrive.

A static map is included to visually display the locations of the restaurant and the customer at the moment of order placement. While live tracking is not implemented to avoid added complexity and development time, the map serves as a useful visual reference for users.

If the order has not yet been confirmed by the restaurant, users are given the option to cancel it directly from this page. Below the map, the page displays the total order amount and the name of the restaurant.

For customer support, the page includes an email address and phone number for inquiries. These are placeholder details since the application is fictitious, but they simulate real-world functionality. Finally, a button at the bottom allows users to navigate back to the "See Orders" page, making it easy to manage multiple orders and track their statuses.

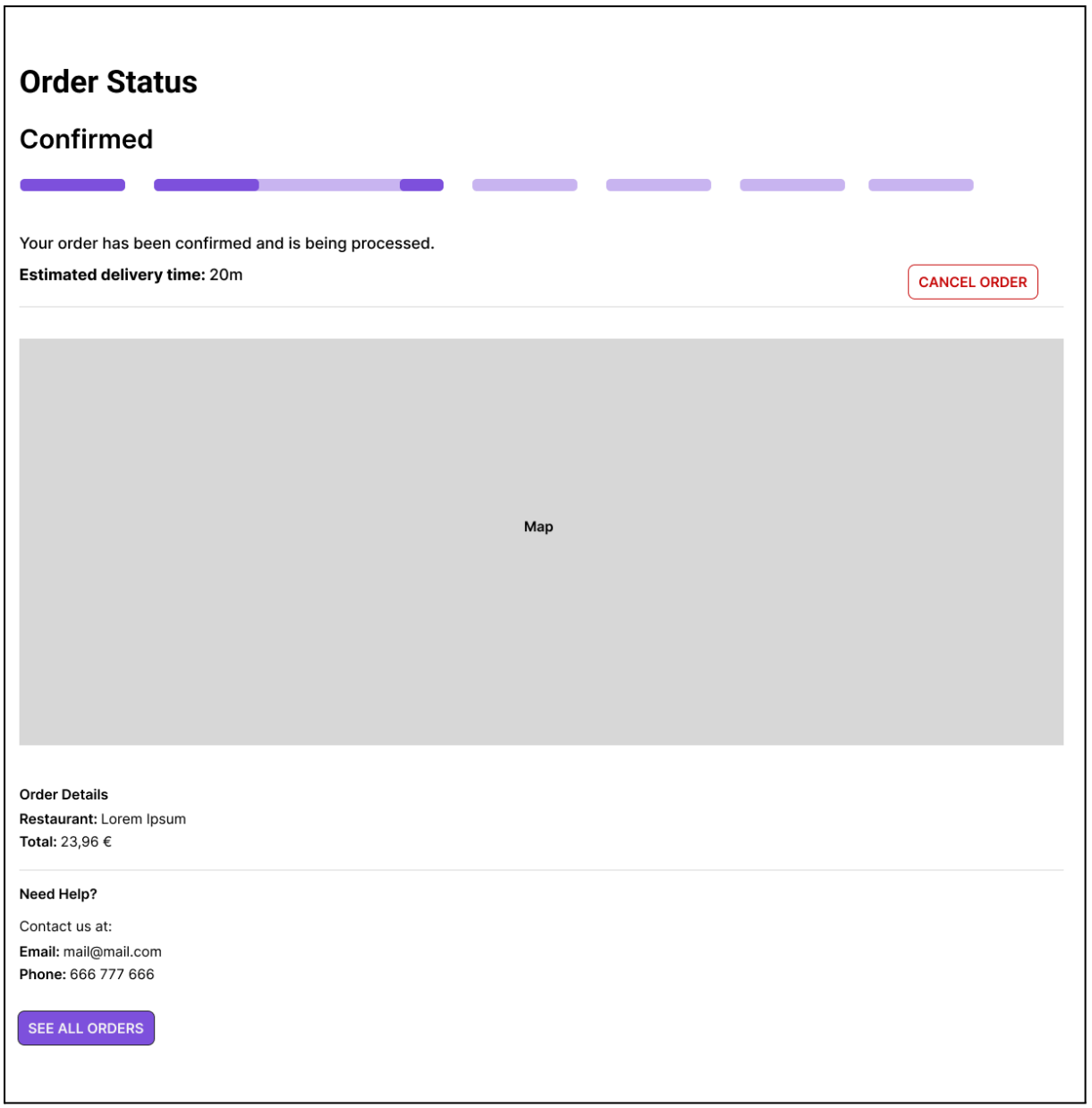


Figure 31. Order Status page.

The *Insights* page provides users with a comprehensive overview of their ordering habits through visual and statistical data. At the top, it features a graph that users can filter by different time periods, such as daily, weekly, or monthly views. This interactive graph allows users to track trends in their order history over time.

Below the graph, several key statistics are displayed to give users deeper insights into their behavior:

- **Total Orders**: The cumulative number of orders placed by the user.
- **Average Expense per Order**: A calculated figure showing how much, on average, the user spends on each order.
- **Most Active Day**: The day of the week or month when the user is most likely to place orders.
- **Favorite Restaurant**: The restaurant the user has ordered from the most, highlighting their preferred choice.

This page is designed to help users better understand their spending and ordering patterns, providing valuable information for budgeting or discovering personal preferences. The combination of visual data and statistical facts ensures an engaging and insightful user experience.

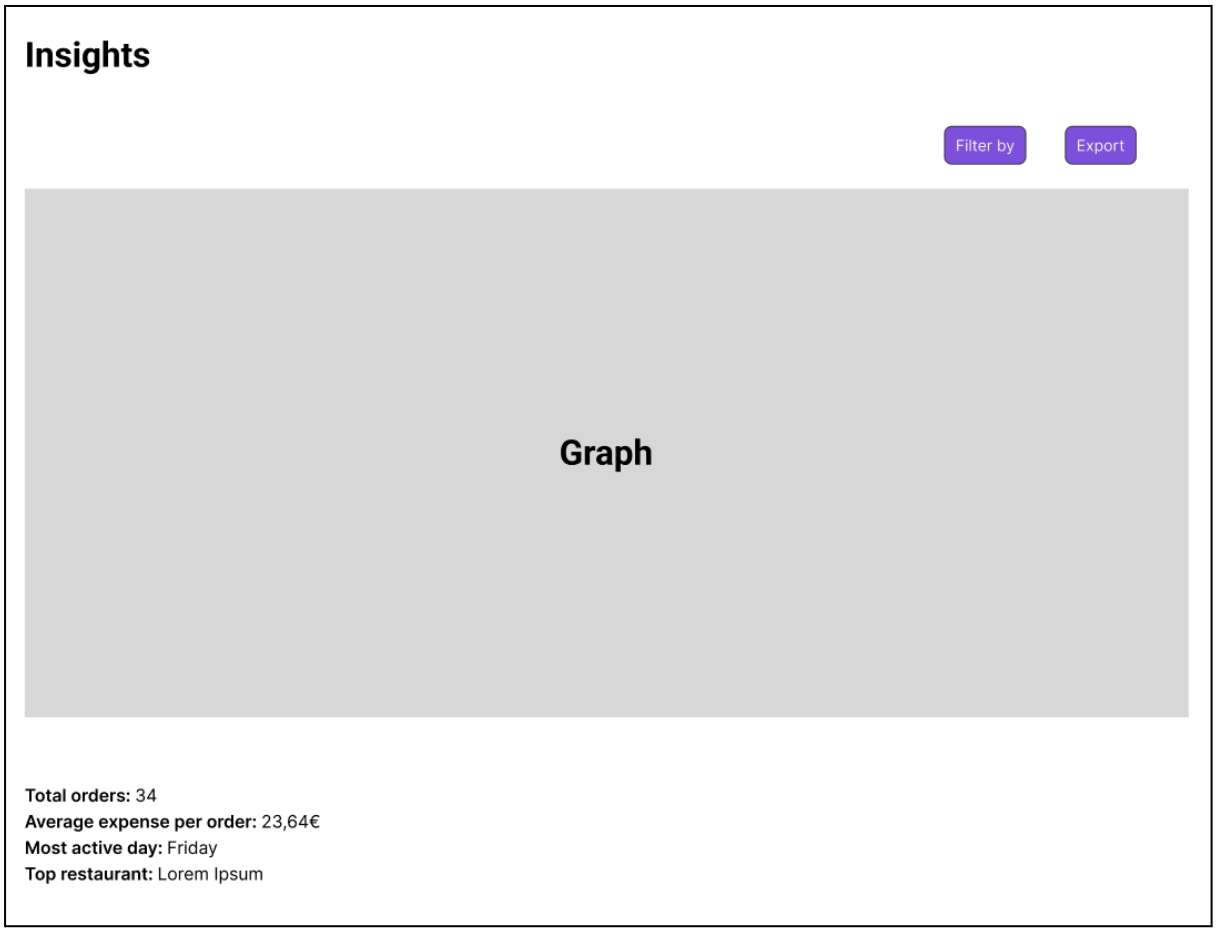


Figure 32. Insights page.

The *See Orders* page provides users with a detailed list of all their orders, arranged in descending order by date. This ensures that recent orders or those currently in progress are always displayed at the top, making it easy for users to track their most relevant activities.

Each order entry includes:

- **Restaurant Name**: For clear identification of where the order was placed.
- **Date of Placement**: To provide a quick reference for when the order was made.
- **Current Status**: To keep users informed about the order's progress at a glance.

Users can click on any order in the list to be redirected to the Order Status Page for that specific order. This feature allows them to view real-time updates and additional details about their chosen order.

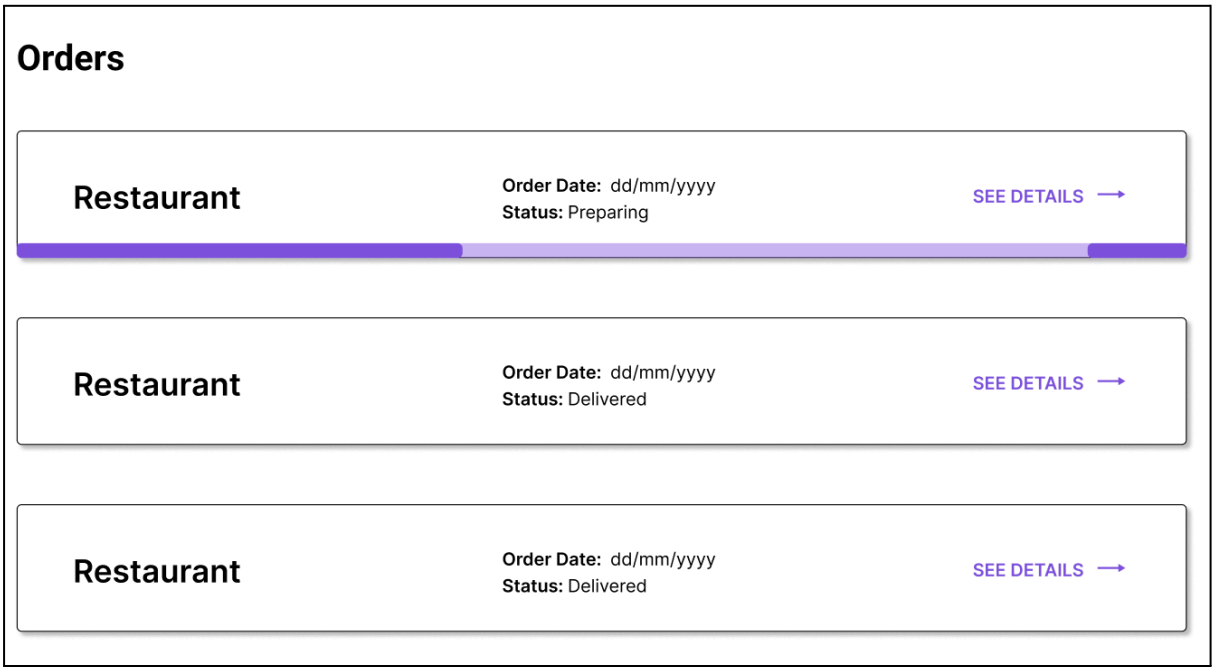


Figure 33. Orders page.

The *Order Status Update Dialog* is a pop-up that appears when a restaurant or courier attempts to update the status of an order. This dialog ensures clarity and confirmation before any status changes are made, reducing errors and maintaining workflow accuracy.

The dialog displays the following information:

- **Order Details**: Key information such as the order ID, and relevant details to identify the order quickly.
- **Current Status**: The order's current state to provide context for the next action.

It includes two buttons:

1. **Update Status**: Progresses the order to the next stage in the workflow (e.g., from "Preparing" to "Ready for Pickup").

2. **Cancel and Close**: Closes the dialog without making any changes to the order.

This dialog ensures that users can make informed decisions about order updates while maintaining an intuitive and straightforward interface.

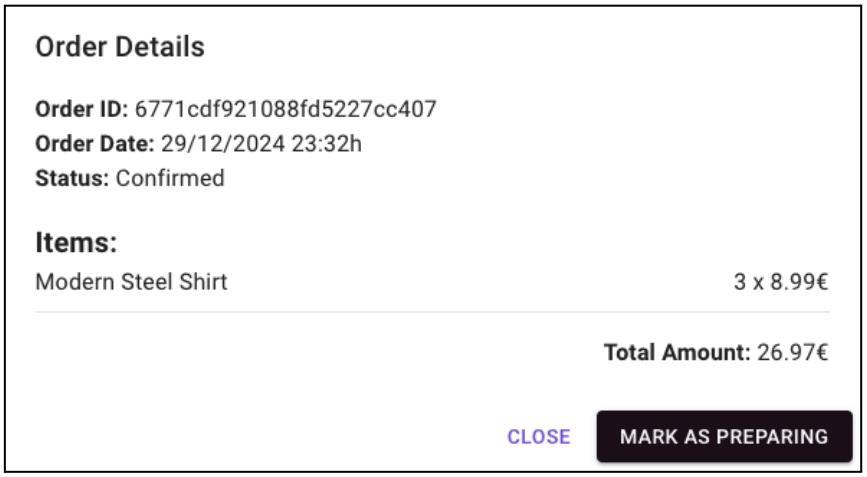


Figure 34. Order Status Update Dialog.

The *Dashboard* page acts as the central hub for restaurants and couriers, providing tailored functionalities to meet their specific needs. For restaurants, the page may display cards that allow quick access to pending orders, helping them manage orders awaiting acceptance or those currently in progress. It may also include tools for managing the restaurant's menu, offering a straightforward way to update or adjust items.

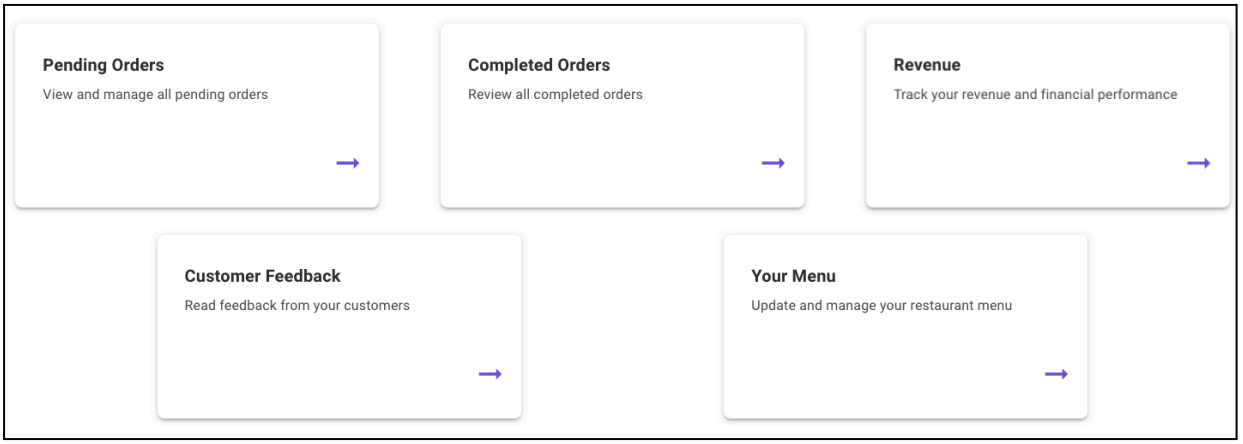


Figure 35. Dashboard page (restaurant).

For couriers, the dashboard focuses on operational efficiency by presenting options such as viewing available orders ready for pickup and delivery.

8.3.3. Internal Front-End Design

This section focuses on the internal design of the CartPage component, specifically the functionality for fetching the customer's cart from the backend through the customerAPI. To maintain clarity and brevity, we have chosen to illustrate only this operation, as extending the analysis to cover all frontend functionalities would make the section excessively complex and lengthy.

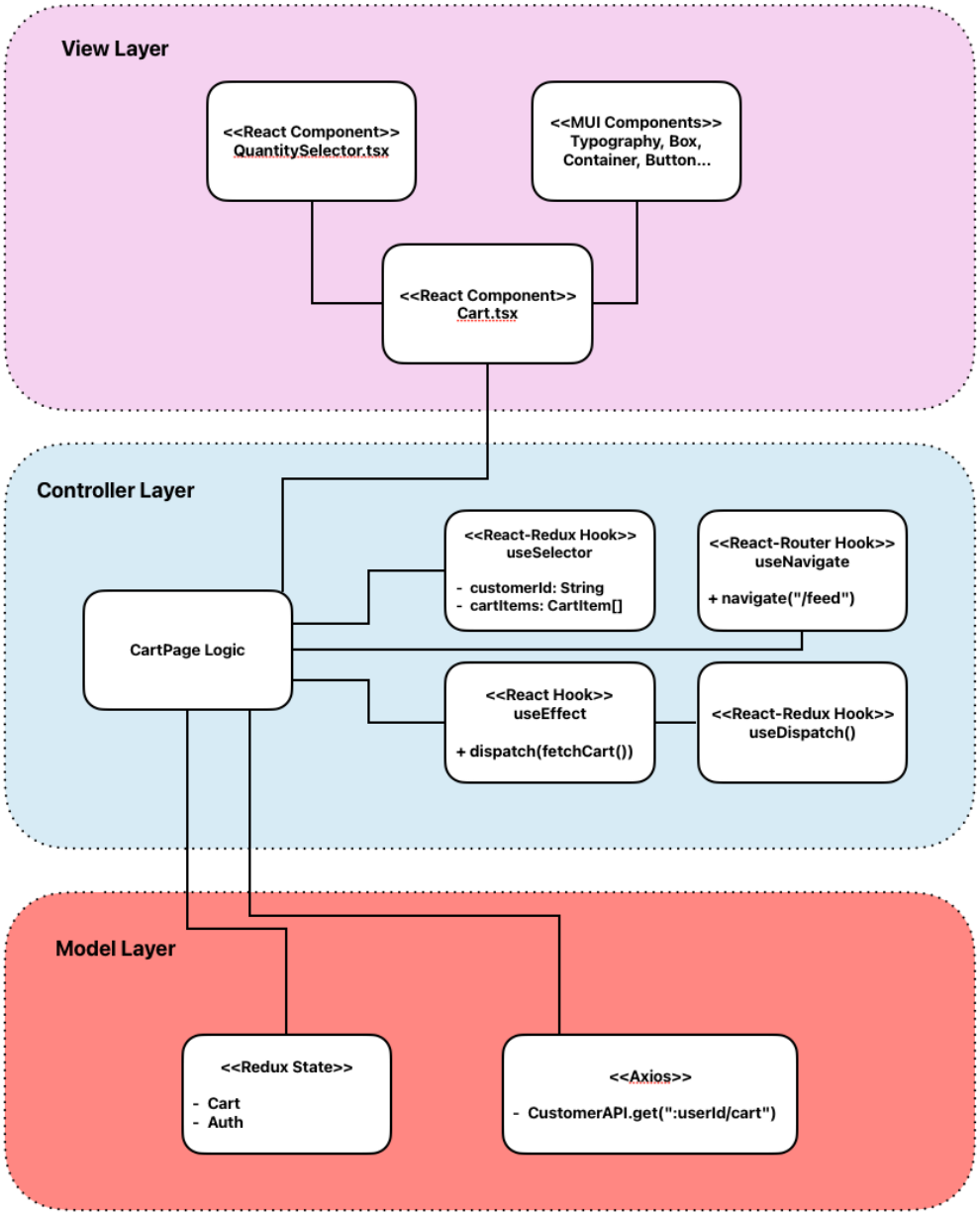


Figure 36. MVC Diagram for Fetching the Customer's Cart.

The design is presented using a layered architecture diagram, organizing the components, logic, and data flow into **Model-View-Controller** (MVC) layers. Additionally, a sequence

diagram is included to detail the step-by-step process of retrieving cart data, showcasing the interaction between frontend components, *Redux* state management, and the backend API.

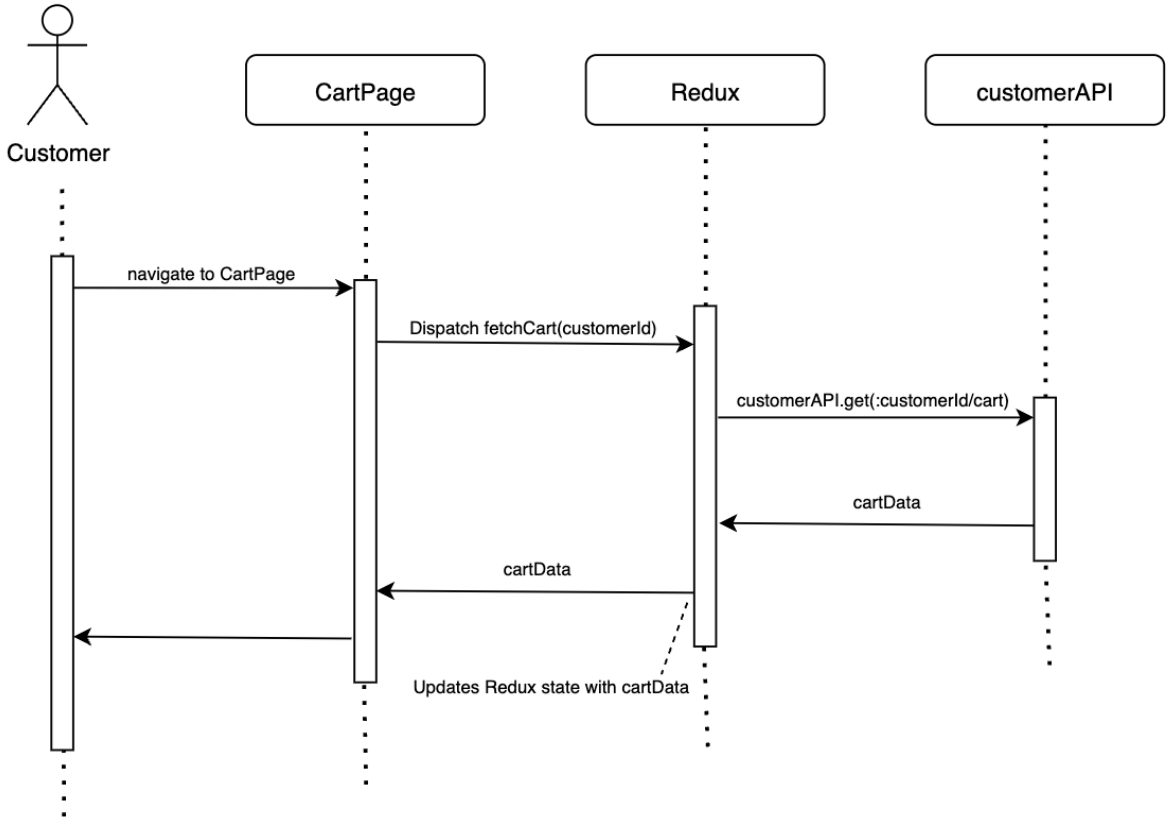


Figure 37. Sequence Diagram for Fetching the Customer's Cart.

Figure 37 illustrates the process of fetching the customer's cart in the *CartPage* component. The sequence begins when the user navigates to the CartPage. Upon loading, the CartPage component dispatches the fetchCart action, passing the userId obtained from the authentication state.

In this context, *dispatch* refers to triggering an action in *Redux* [36], a state management library used to manage the application's global state. Redux allows the application to store and update shared data, such as the cart and user information, in a centralized and predictable way. The dispatch function sends an action to Redux, which then processes it through reducers or middleware (like *fetchCart*) to update the state or perform side effects like API calls.

The *fetchCart* action calls the backend API to retrieve the customer's cart data. Once the data is returned, Redux updates the global state with the new information. The CartPage component then reacts to these state changes and renders the cart items for the user.

8.4. Back-end Design

This chapter focuses on the architecture and functionality of the back-end, which powers the application's core operations. It outlines the services, database design, and communication flows between components, highlighting how the back-end supports data processing, storage, and interaction with the front-end.

8.4.1. Back-End Components and Operations

This section provides an overview of the operations handled by the back-end services and their interaction with the front-end. To ensure clarity and avoid redundancy, only the Orders Microservice is described, as it exemplifies the structure and behavior shared across the microservices architecture.

The application follows the API Gateway pattern [37], where all requests from the front-end pass through a central gateway. The API Gateway acts as a single entry point, routing requests to the appropriate microservice based on the endpoint prefix. Each microservice is accessed via an endpoint structured as /api/{microservice_name}. For instance, requests related to orders are routed through /api/orders, which connects the API Gateway to the Orders Microservice. This pattern simplifies communication, centralizes control, and ensures a clear separation of concerns between microservices.

The following endpoints facilitate order management and communication between the front-end and back-end systems:

- **POST** /api/orders
  Description: Creates a new order.
  The POST method is used to add new data to the system, such as creating a new order.

- **GET** /api/orders
  Description: Retrieves all orders with optional pagination.
  The GET method fetches data from the system, providing details without modifying it.

- **GET** /api/orders/pending/:restaurantId
  Description: Retrieves all pending orders for a specific restaurant.

- **GET** /api/orders/pending
  Description: Retrieves all pending orders with optional pagination.

- **GET** /api/orders/customers/:customerId
  Description: Retrieves all orders for a specific customer.

- **GET** /api/orders/:id

Description: Retrieves details of a specific order.

- **PATCH** /api/orders/:id/status
  Description: Updates the status of a specific order.
  The PATCH method [38] is used to partially update existing data, unlike PUT, which replaces the entire resource. For example, a PATCH request to update an order's status modifies only that specific field, leaving other data intact. In contrast, a PUT request would require sending the complete resource with all fields, even those unchanged.

- **PATCH** /api/orders/:id/courier/accept
  Description: Assigns a courier to a specific order.

- **POST** /api/orders/verify/:id
  Description: Verifies the PIN for a specific order.

- **DELETE** /api/orders/:id
  Description: Deletes a specific order.
  The DELETE method removes data from the system, such as deleting an order.

On the other hand, Figure 38 illustrates the architecture and class diagram of the Orders Microservice, highlighting its key components and their relationships. Additionally, Figure 39 presents the class diagram of the Payments Microservice. These two diagrams are provided because, in the next section, we will refer to them to explain and construct the sequence diagram for the "place order" execution flow.

The *Domain* layer includes the *OrdersController*, which listens to the *NATS* server for incoming messages and events related to orders. It delegates business logic to the *OrdersService*, which manages core operations like order creation, status updates, and courier assignments.

The *Data Management* layer interacts with the database through *Prisma* [39], represented by the *OrderDelegate*. The OrderDelegate is an automatically generated API provided by Prisma that serves as the primary interface for interacting with the Order model in the database. It allows the service to perform CRUD operations such as creating, updating, retrieving, and deleting orders in a consistent and type-safe manner. By abstracting database interactions, the OrderDelegate simplifies the implementation of complex queries and ensures data integrity while leveraging Prisma's ORM capabilities.

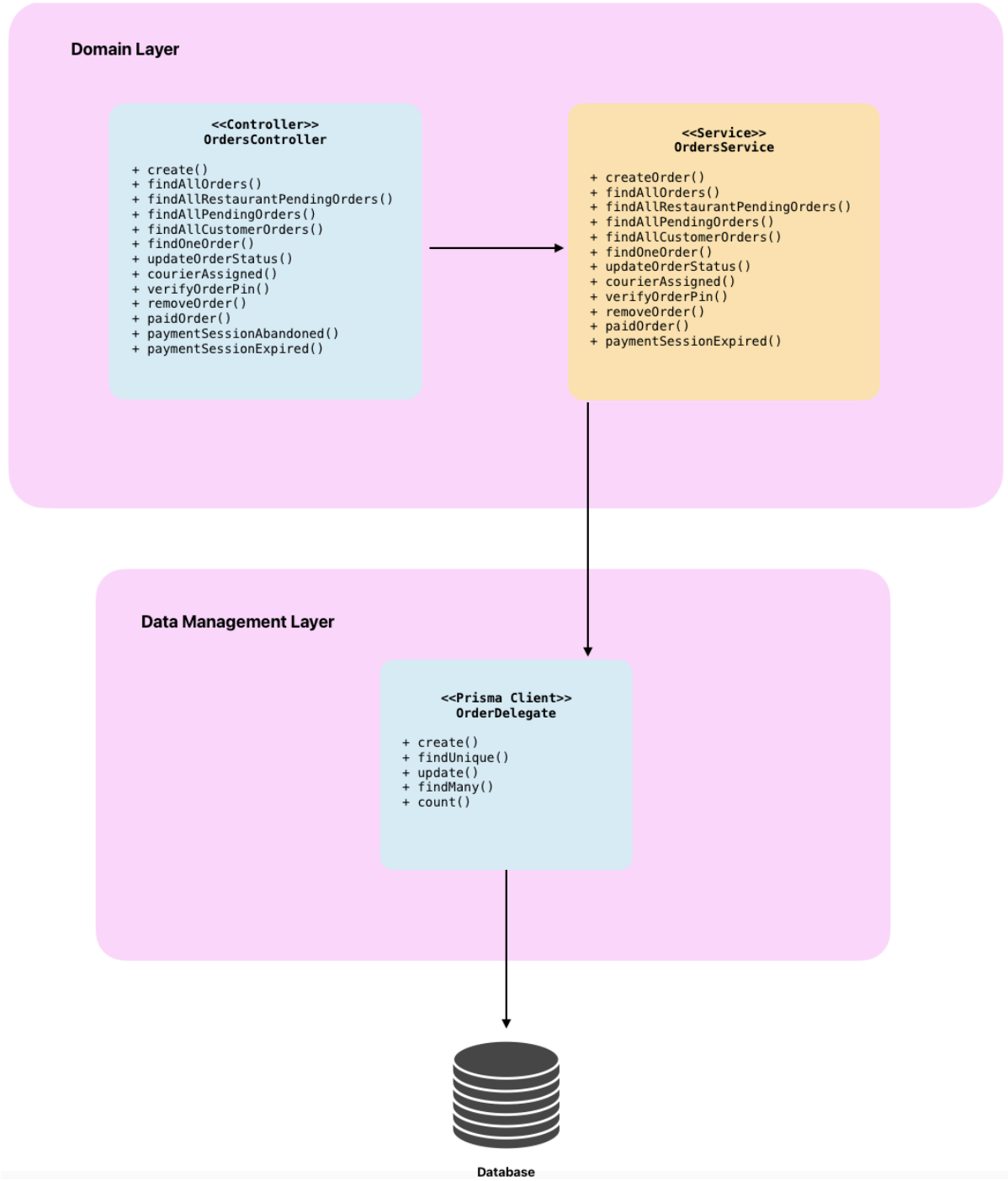


Figure 38. Diagram of Classes of the Orders Microservice.

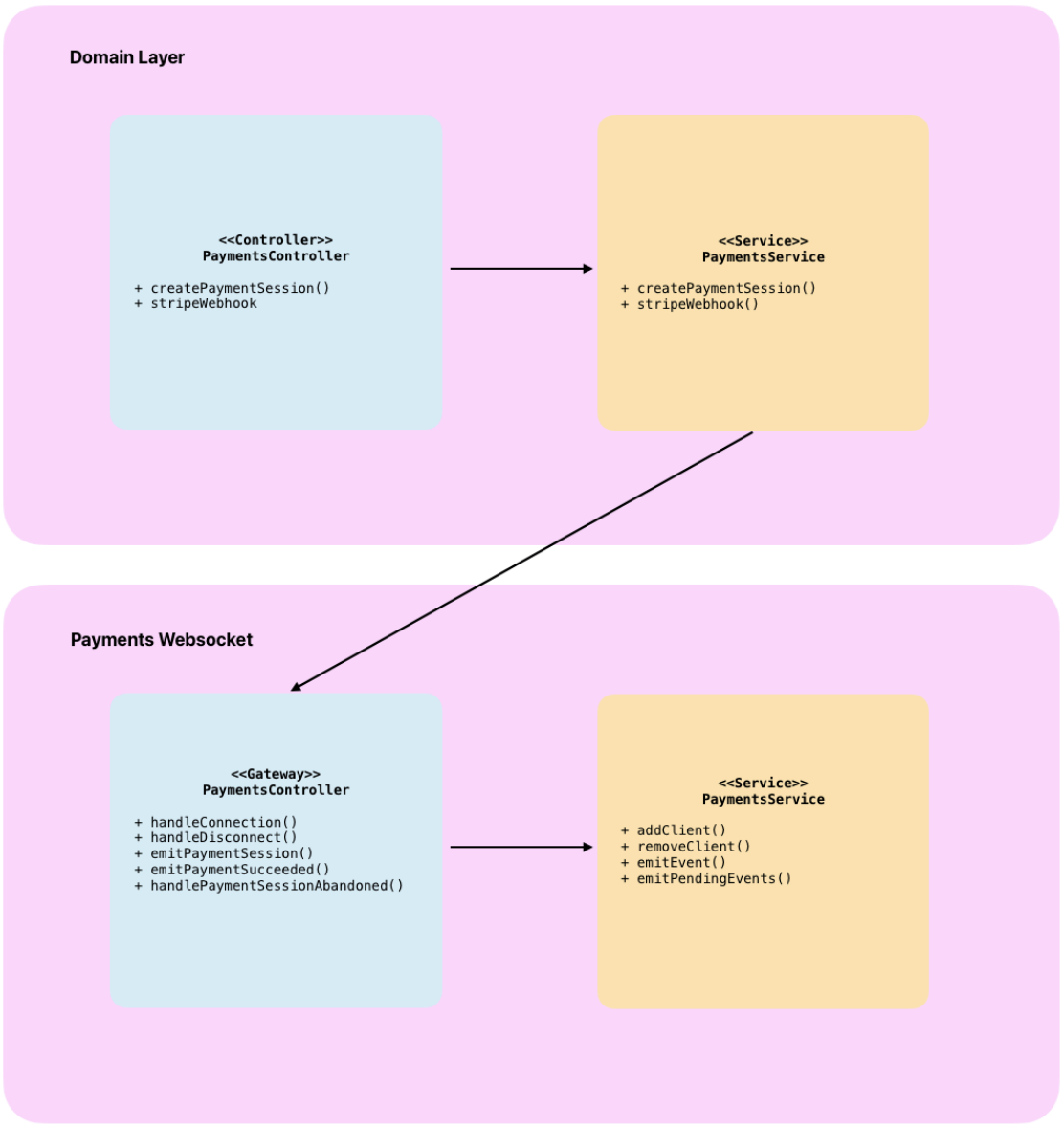


Figure 39. Diagram of Classes of the Payments Microservice.

8.4.2. Sequence Diagrams

This chapter presents the sequence diagram for the *place order* operation, showcasing core back-end interactions. It serves as a representative example of system workflows and microservice communication.

Figure 40 outlines the high-level interaction flow for the *Orders Microservice* logic when a customer initiates an order. The process begins with the client sending a POST request to the API Gateway containing the *createOrderDto* [40], which encapsulates the necessary order details. The API Gateway processes the request and forwards a createOrder message, along with the createOrderDto, to NATS, the messaging system.

NATS ensures message delivery and relays the request to the Orders Microservice, where the core order creation logic is implemented. After processing the request and creating the order,

the Orders Microservice returns an orderId through NATS, signifying that the order was successfully created. Finally, the API Gateway retrieves the orderId from NATS and returns it to the client as a response.

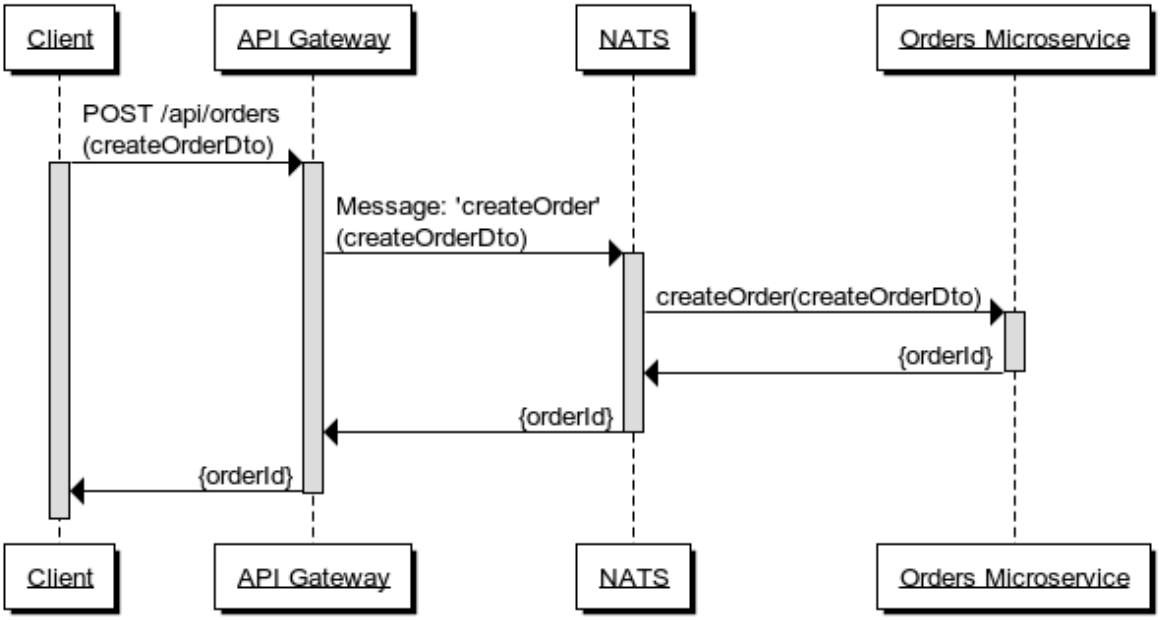


Figure 40. High-level order creation process overview .

Building on the previous diagram, Figure 41 provides a deeper exploration of how the Orders Microservice processes the createOrder request. Once NATS delivers the createOrder(createOrderDto) message to the OrdersController, the controller passes the request to the OrdersService, which handles the main business logic for creating the order.

The OrdersService uses Prisma to interact with the database to insert the order details and after successfully storing the order, the OrdersService emits an '*order_created*' event through NATS, containing the orderData and the orderId. This event ensures that other microservices or components can react to the creation of the order. The orderId is then sent back to the OrdersController, which forwards it to NATS, completing the response flow.

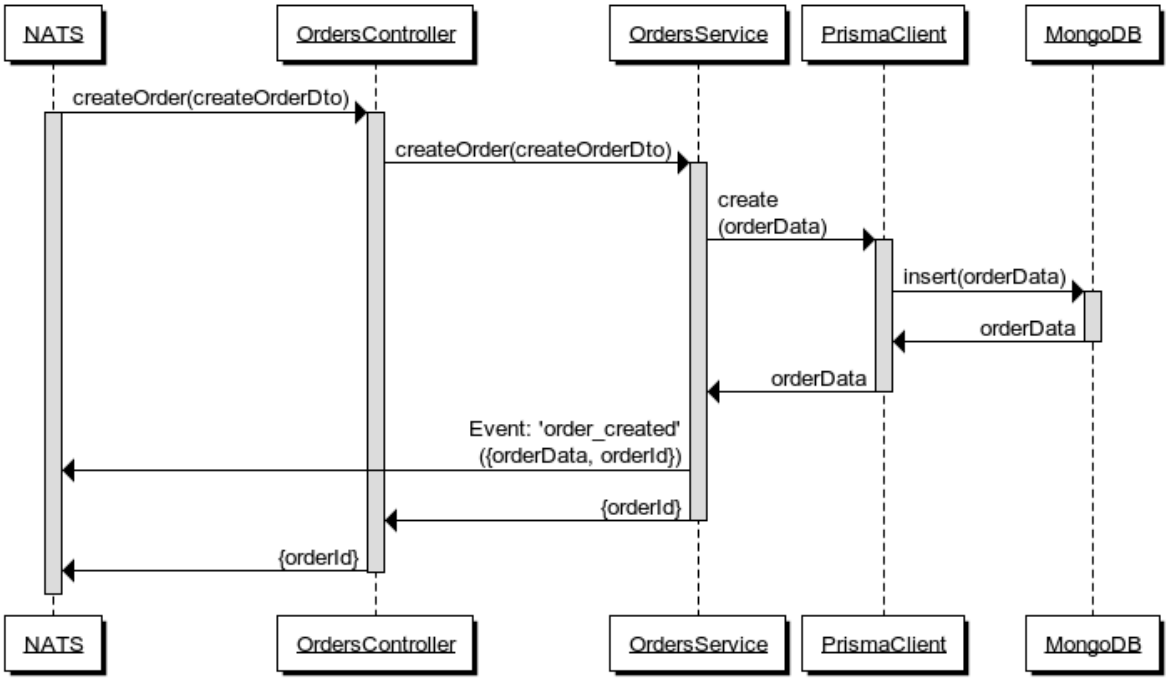


Figure 41: Detailed order creation process within Orders Microservice.

Now we will focus on the steps triggered after the order is created. The Orders Service emits an '*order_created*' event to NATS, containing the order data and orderId. After listening to this event, the Payments Controller initiates the payment session creation workflow.

The Payments Controller delegates the task to the Payments Service, which communicates with the Stripe SDK [41] to create a payment session using the provided payment data. Once the payment session is successfully created, the Stripe SDK returns the session details to the Payments Service.

The Payments Service then emits the payment session details, including the orderId and the URL of the created payment session via the Payments WebSocket, enabling real-time communication with the client. Finally, the Payments Service relays the payment session back to the Payments Controller, which forwards the session details to NATS, even though no component is subscribed to this response.

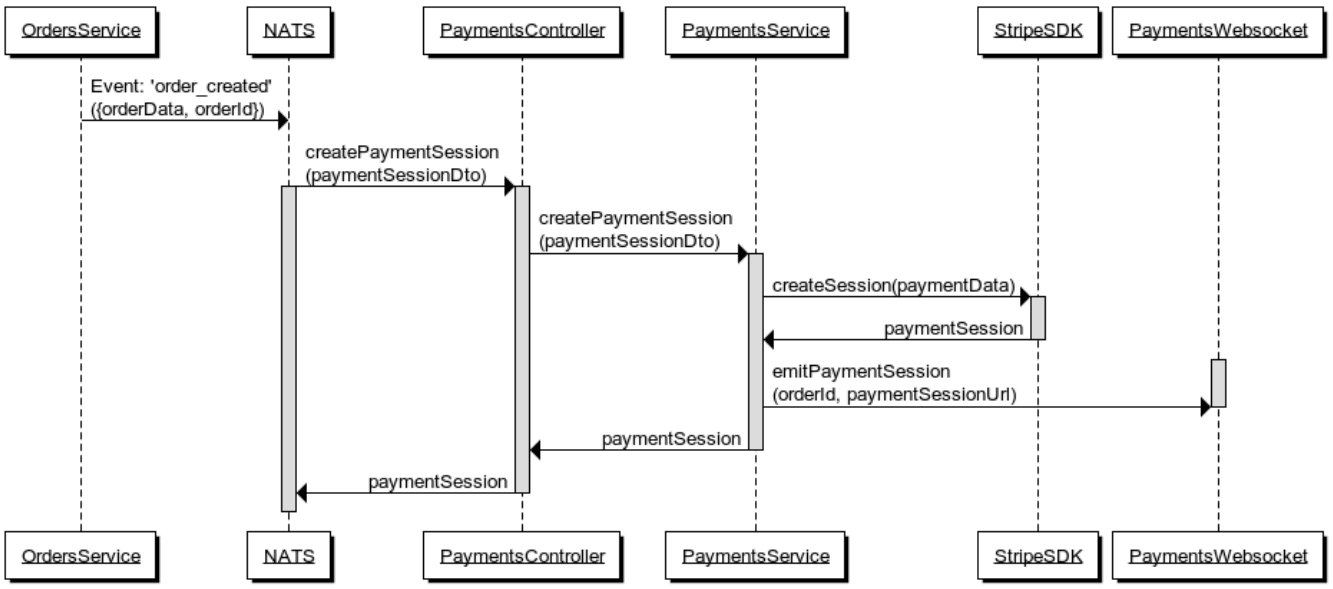


Figure 42: Payment session creation process following order creation.

Figure 43 details the process of notifying the client about payment success in real-time. After the Payments Service emits the payment session details via the Payments WebSocket Gateway [42], the gateway forwards the event to the Payments WebSocket Service by invoking emitEvent, passing the orderId and the event name '*payment_success*'.

The Payments WebSocket Service then uses the Socket associated with the specific orderId to emit a '*payment_success*' event, including the orderId as part of the payload. This event is received directly by the client, ensuring immediate notification of the successful payment.

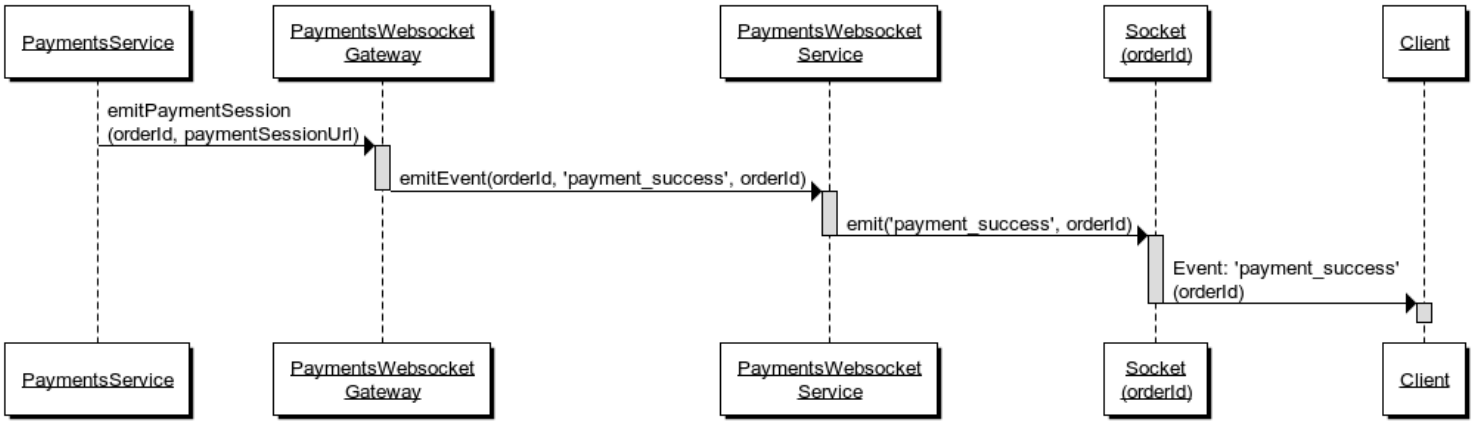


Figure 43: Real-time payment success notification to the client.

Lastly, Figure 44 illustrates the processing of Stripe webhook events. The flow begins when Stripe sends a POST request to the Payments Controller at the webhook endpoint. The

Payments Controller forwards the request to the Payments Service, which uses the Stripe SDK to construct and verify the incoming. This verification is performed using secrets defined in the .env file and the Stripe signature included in the request's headers.

Depending on the event type, the following actions are performed:

- If the event is '*charge.succeeded*', the Payments Service emits a '*payment_succeeded*' event through the Payments WebSocket for real-time client updates and publishes the same event via NATS for further processing.

- If the event is '*checkout.session.expired*', the Payments Service emits a '*payment_session_expired*' event via NATS to handle expired sessions.

Finally, the Payments Service sends a response back to the Payments Controller, which relays the response to Stripe, confirming the successful processing of the webhook event.

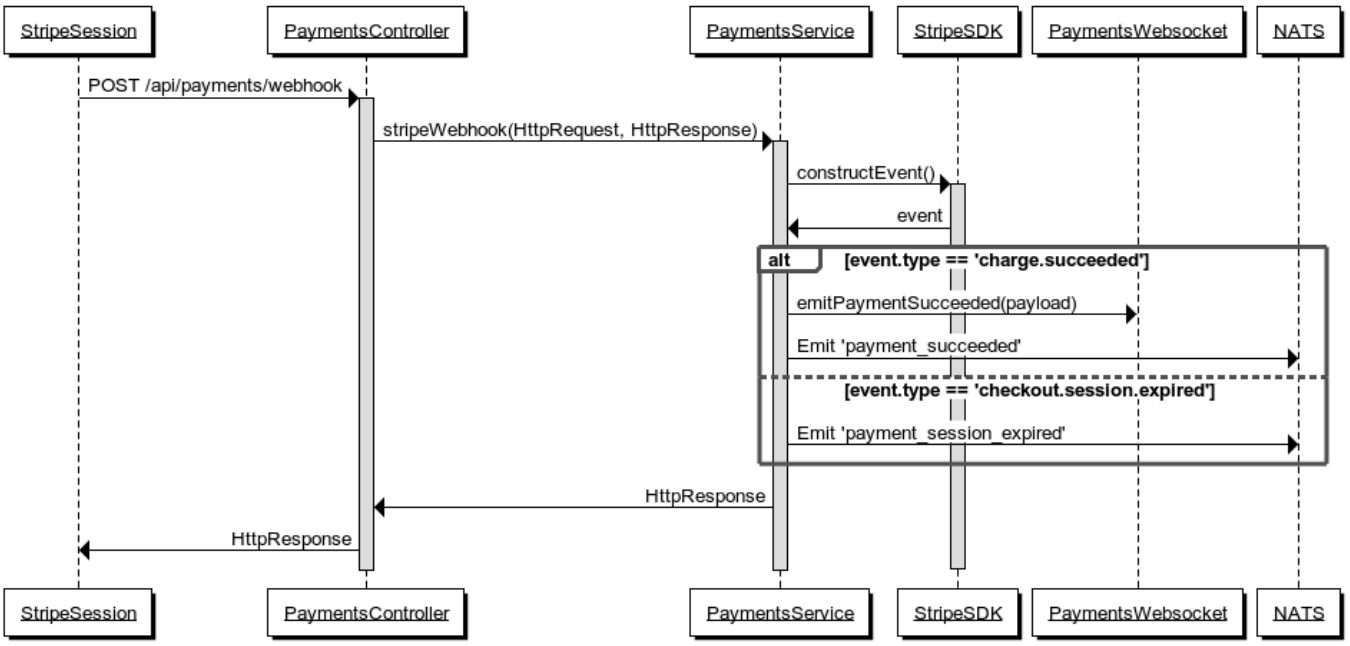


Figure 44: Handling Stripe payment webhook events.

8.4.3. Database Design

This section outlines the database design, focusing on the structure and relationships of the entities that store the application's data. The design ensures efficient support for back-end operations while maintaining data consistency and integrity.

The **Orders Microservice** employs MongoDB due to its flexibility in handling dynamic, schema-less data, making it ideal for storing orders, which often vary in structure depending on their contents. Additionally, MongoDB is advantageous for addressing scalability concerns related to order storage. In contrast, other microservices use PostgreSQL for its

robust relational capabilities, ensuring strict data consistency and complex relationship management where required.

To simplify and standardize database interactions, the system uses Prisma as an ORM. Prisma's modern query capabilities and type safety integrate very well with both MongoDB and PostgreSQL, allowing efficient, unified data handling across the microservices.

While MongoDB is a schema-less database, the Orders Microservice leverages Prisma to define and manage a structured schema at the application level. Prisma allows me to define a blueprint for how data like orders, their items, and receipts should look. This blueprint helps ensure that the data is consistent and easier to work with in the code.

The Orders Microservice documents in MongoDB are structured like this:

Order Document

```
{
  "_id": "ObjectId",
  "date": "ISODate",
  "totalAmount": "Number",
  "status": "String",
  "deliveryTime": "ISODate",
  "stripeChargeId": "String",
  "restaurantId": "UUID",
  "restaurantName": "String",
  "customerId": "UUID",
  "pin_code": "String",
  "courierId": "UUID",
  "items": ["ObjectId"],
  "orderReceipt": "ObjectId"
}
```

OrderItem

```
{
  "_id": "ObjectId",
  "dishId": "UUID",
  "name": "String",
  "quantity": "Number",
  "orderId": "ObjectId"
}
```

OrderReceipt

```
{
  "_id": "ObjectId",
  "orderId": "ObjectId",
  "receiptUrl": "String",
  "createdAt": "ISODate",
  "updatedAt": "ISODate"
}
```

On another hand, the **Customers Microservice** utilizes PostgreSQL as its database, leveraging its robust relational capabilities to manage structured data and enforce strict constraints. The following schema, also defined using Prisma ORM, captures the structure and relationships of customers, their carts, addresses, and cart items.

Customer (id, email, loyaltyPoints, cart, address, createdAt, updatedAt)

CHECK {id} is UNIQUE
CHECK {email} is UNIQUE
{cart} references Cart
{address} references Address

Cart (id, customer, customerId, items, updatedAt)

CHECK {id} is UNIQUE
CHECK {customerId} is UNIQUE
{customer} references Customer
{items} references CartItem

CartItem (id, quantity, cart, cartId, dishId)

CHECK {id} is UNIQUE
{cart} references Cart

Address (id, street, streetNumber, city, province, zipCode, additionalInfo, customer, customerId, createdAt, updatedAt)

CHECK {id} is UNIQUE
{customer} references Customer

The schemas for the **Customers, Restaurant, and Courier Microservices**, along with the **Auth Microservice**, are closely linked, sharing a common foundational data model for users. Each user has a unique ID (*UID*) that corresponds to its role-specific entity in the respective microservice, ensuring integration and correlation across the system. Additionally, the email field serves as a shared helper attribute across microservices, facilitating easy cross-referencing.

The **Auth Microservice** defines universal attributes for all user roles:

User (id, email, name, password, image, dateOfBirth, phoneNumber, role, createdAt, updatedAt)

CHECK {id} is UNIQUE
CHECK {email} is UNIQUE
CHECK {phoneNumber} is UNIQUE

Role (Enum) -> CUSTOMER, RESTAURANT, COURIER, ADMIN

Unless otherwise specified, all fields are considered NOT NULL by default.

9. Implementation

This chapter details the step-by-step development process of the application, organized by sprints.

9.1. Front-end Implementation

At its core, React is used to build the user interface due to its component-based architecture, which simplifies the creation of reusable and modular components. This approach allows for rapid development and better management of complex UI requirements. To enhance reliability and reduce potential errors, the application integrates TypeScript, providing strong typing and robust development tools that improve the overall maintainability of the codebase.

The development environment is powered by Vite [43], a build tool selected for its exceptional speed and efficiency. Vite's hot module replacement (*HMR*) [44] feature significantly enhances developer productivity by enabling instantaneous feedback on changes, while its optimized production builds reduce loading times and improve performance for end-users.

Material UI [45] is employed as the design framework, offering a robust set of pre-built components and customizable theming options. Its adherence to Google's Material Design principles ensures a polished and professional user interface, while also reducing the time required to implement complex designs. Material UI's flexibility allows the application to maintain a cohesive visual identity while adapting to specific design needs.

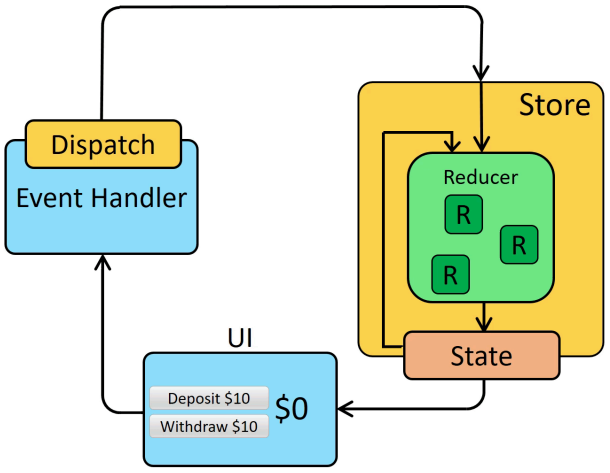


Figure 45: Redux data flow [46].

To handle state management, Redux is employed to maintain a centralized and predictable application state. This is particularly important for managing complex states, such as user authentication and active orders. Redux ensures that state updates are consistent, traceable, and easy to debug, which is crucial for large-scale applications.

Navigation within the application is facilitated by React Router [47], allowing for transitions between pages without reloading. This ensures that the application maintains the efficiency of a single-page application (SPA), offering users a smooth and responsive experience. React

Router's ability to handle dynamic routes and nested navigation makes it an excellent choice for applications with diverse and evolving routing requirements.

9.2. Backend Implementation

The backend development process is structured around building a robust, scalable, and secure architecture for the application. Each microservice is implemented using **NestJS** with TypeScript, leveraging its modular design and dependency injection to ensure maintainability and efficiency. This section outlines the technologies and strategies employed to meet the functional and non-functional requirements.

To manage authentication, **JWT** (JSON Web Tokens) are used to securely transmit user information, with custom *Guards* enforcing access to protected routes. Passwords are encrypted using *bcrypt*, employing a *salt factor* of 10, which determines the number of rounds used to add complexity to the hash, making it more resistant to brute-force attacks. Together, these measures provide a robust framework for authentication and authorization, ensuring the security of sensitive user data.

In NestJS, a *Guard* is a feature used to manage and control access to specific parts of an application, typically for authentication and authorization purposes. Guards are middleware-like mechanisms that run before the route handler is executed. They determine whether a request should be processed or rejected based on certain conditions.

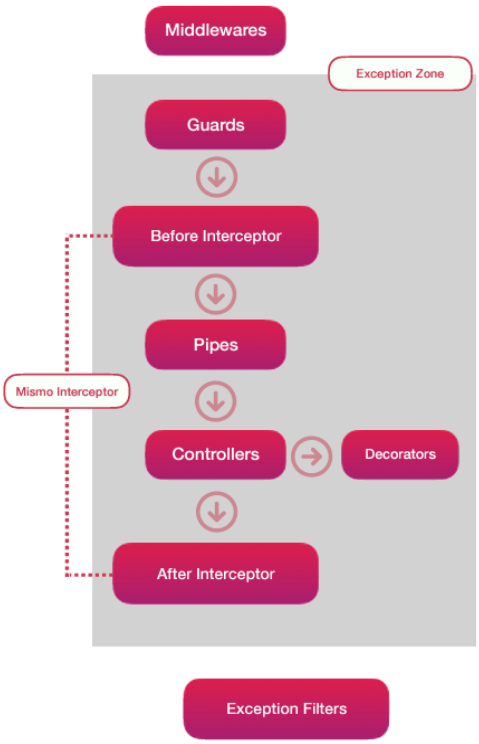


Figure 46: NestJS Request lifecycle [48].

Data persistence is handled using Prisma ORM, which streamlines database interactions across different services. Prisma integrates seamlessly with both MongoDB (used in the Orders Microservice) and PostgreSQL (used in other microservices), providing type safety and efficient query management. This ensures consistency and predictability in handling data across the system.

Real-time communication is facilitated by *Socket.IO*, enabling instant updates for clients, such as payment status or order tracking. For payment processing, the backend integrates with Stripe, which provides a secure and efficient way to handle transactions and webhooks.

NATS serves as the message broker for inter-service communication. This ensures asynchronous, reliable message passing between microservices, decoupling their operations and promoting scalability.

Finally, the backend environment is fully containerized using **Docker**, including all microservices, the NATS server, and the PostgreSQL database (to work in a local environment). This setup enables consistent local development and simplifies deployment. All containers are orchestrated using *Docker Compose*, allowing them to run simultaneously with minimal configuration effort.

9.2. Development Process

This section outlines the step-by-step development of the application, organized into four sprints. Each sprint represents a focused iteration where specific features were implemented for both the frontend and backend. Following the Agile methodology, the development process was incremental, enabling continuous refinement and adaptation to evolving requirements.

9.2.1. Sprint 1

Sprint 1 marked the beginning of the development phase and was centered on laying a strong foundation for the project. Since it had been several months since my last experience with NestJS and microservices (in a practical manner), I dedicated time to revisiting these technologies. This included reviewing documentation, consulting microservices books, experimenting with small projects, and completing tutorials to regain familiarity with the framework and its ecosystem. Additionally, as React was practically new to me, I spent a significant portion of the sprint exploring its core concepts. I followed tutorials on platforms like *Udemy* [49] and experimented with simple components, and I gradually built confidence in using React in conjunction with TypeScript. These preparatory steps were necessary to ensure that I could handle the more complex tasks ahead effectively.

Once I was comfortable with the core tools, I began working on the backend, and I thought that I first needed to implement some primary microservices methods and functions before I could start with the frontend. I began by developing the REST API Gateway, Auth, Customer, Restaurant, and Courier microservices. For each microservice, I set up its core operations, integrated the necessary configurations, and established connections to their respective databases.

```typescript
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(@Inject(NATS_SERVICE) private readonly client: ClientProxy) {}
  private readonly LOGGER = new Logger('AuthGuard');
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromCookie(request);

    if (!token || token === '') {
      throw new UnauthorizedException();
    }
    try {
      const { user, newToken } = await firstValueFrom(
        this.client.send('authVerifyUser', token),
      );
      request['user'] = user;

      request['token'] = newToken;

      if (!this.isAllowed(user.role, request.url)) {
        throw new UnauthorizedException(
          'Access Denied: You do not have permission for this endpoint',
        );
      }
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

  private extractTokenFromCookie(request: Request): string | undefined {
    return request.cookies?.authToken;
  }

  private isAllowed(role: string, url: string): boolean {
    // Definir las reglas de acceso basadas en el rol y la URL
    const accessRules = {
      CUSTOMER: ['/api/customer/*'], // Accesible solo a CUSTOMER
      RESTAURANT: ['/api/restaurant/*'], // Accesible solo a RESTAURANT
      ADMIN: ['/api/admin/*'], // Accesible a ADMIN y RESTAURANT
      COURIER: ['/api/courier/*'], // Accesible solo a COURIER
    };

    // Verificar si la URL solicitada pertenece a /api/auth/*
    if (/^\/api\/auth\//.test(url)) {
      return true; // Cualquier usuario puede acceder a las rutas de autenticación
    }

    // Verificar si el rol tiene acceso a la URL solicitada
    const allowedUrls = accessRules[role] || [];

    // Verificar si la URL solicitada se encuentra dentro de las permitidas para el rol
    return allowedUrls.some((pattern) => new RegExp(pattern).test(url));
  }
}
```

Figure 47: Custom Guard for Authentication.

Additionally, I configured the *docker-compose.yml* file to streamline the local development environment. This configuration included containerizing all the microservices, the NATS messaging server, and the PostgreSQL database instances.

Each service was defined as a separate container, with appropriate environment variables and network settings to enable communication between services. I also added volume configurations to persist data and ensure it remained consistent across container restarts. By

using Docker Compose, I was able to orchestrate the entire environment, allowing all containers to be built, started, and managed with a single command. This setup significantly simplified the development process by ensuring consistency, reducing manual configuration, and providing a fully integrated environment for testing and debugging the microservices.

```yaml
services:
    nats-server:
        image: nats:latest
        networks:
            - tfg-network
        ports:
            - 8222:8222

    api-gateway:
        build: ./api-gateway
        ports:
            - ${CLIENT_GATEWAY_PORT}:${CLIENT_GATEWAY_PORT}
        volumes:
            - ./api-gateway/src:/usr/src/app/src
        command: npm run start:dev
        networks:
            - tfg-network
        environment:
            - PORT=3000
            - NATS_SERVERS=nats://nats-server:4222

    auth-ms:
        build: ./auth-ms
        volumes:
            - ./auth-ms/src:/usr/src/app/src
        command: npm run start:dev
        networks:
            - tfg-network
        environment:
            - PORT=3001
            - NATS_SERVERS=nats://nats-server:4222
            - JWT_SECRET=${JWT_SECRET}
            - LEADER_DATABASE_URL=${LEADER_DATABASE_URL}${AUTH_DB}?schema=public
            - FIREBASE_TYPE=${FIREBASE_TYPE}
            - FIREBASE_PROJECT_ID=${FIREBASE_PROJECT_ID}
            - FIREBASE_PRIVATE_KEY_ID=${FIREBASE_PRIVATE_KEY_ID}
            - FIREBASE_PRIVATE_KEY=${FIREBASE_PRIVATE_KEY}
            - FIREBASE_CLIENT_EMAIL=${FIREBASE_CLIENT_EMAIL}
            - FIREBASE_CLIENT_ID=${FIREBASE_CLIENT_ID}
            - FIREBASE_AUTH_URI=${FIREBASE_AUTH_URI}
            - FIREBASE_TOKEN_URI=${FIREBASE_TOKEN_URI}
            - FIREBASE_AUTH_PROVIDER_X509_CERT_URL=${FIREBASE_AUTH_PROVIDER_X509_CERT_URL}
            - FIREBASE_CLIENT_X509_CERT_URL=${FIREBASE_CLIENT_X509_CERT_URL}
            - FIREBASE_UNIVERSE_DOMAIN=${FIREBASE_UNIVERSE_DOMAIN}

    restaurants-ms:
        build: ./restaurant-ms
        volumes:
            - ./restaurant-ms/src:/usr/src/app/src
        command: npm run start:dev
        networks:
            - tfg-network
        environment:
            - PORT=3001
            - NATS_SERVERS=nats://nats-server:4222
            - LEADER_DATABASE_URL=${LEADER_DATABASE_URL}${RESTAURANT_DB}?schema=public

    customers-ms:
        build: ./customer-ms
        volumes:
            - ./customer-ms/src:/usr/src/app/src
        command: npm run start:dev
        networks:
            - tfg-network
        environment:
            - PORT=3002
            - NATS_SERVERS=nats://nats-server:4222
            - LEADER_DATABASE_URL=${LEADER_DATABASE_URL}${CUSTOMER_DB}?schema=public
networks:
  tfg-network:
      external: true
```

Figure 48: docker-compose.yml.

A major technical challenge during this phase was configuring a master-slave database replication strategy for the PostgreSQL databases. This strategy was chosen to ensure high availability, data redundancy, and improved reliability. The setup involved researching best practices, troubleshooting configuration issues, and testing the replication to verify its functionality. While this task took longer than initially anticipated, the effort paid off by creating a robust and scalable backend infrastructure that could handle higher demands.

A **master-slave** strategy in database management is a replication setup where one database instance (the master) handles all write operations, and one or more secondary instances (the slaves) replicate the data from the master and handle read operations. This approach improves data redundancy by ensuring a backup exists in case of failure, enhances performance by distributing read operations, and supports scalability by reducing the load on the master database.

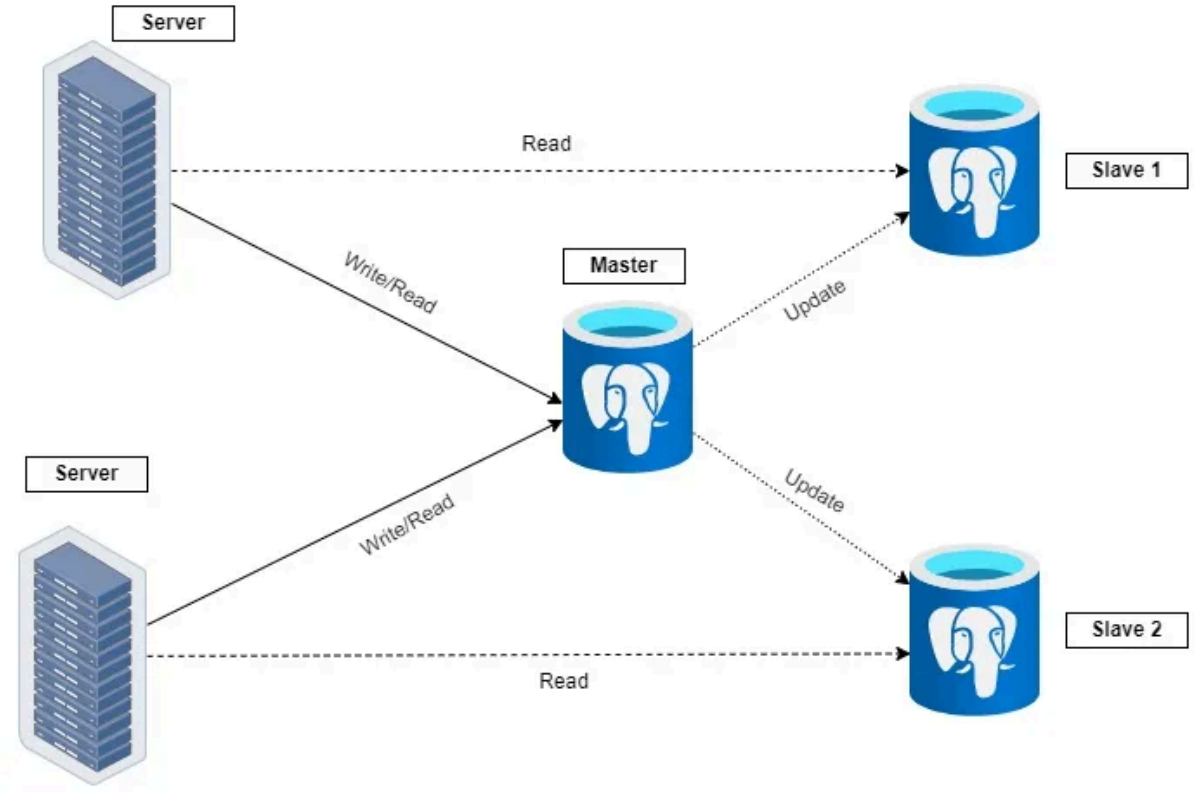


Figure 49: Master-Slave Strategy [50].

After completing the foundational backend work, I shifted focus to the frontend. I began by setting up the project environment, which included installing Material UI to streamline the creation of a polished and responsive user interface. Establishing the core file structure was another step, ensuring that the project would be easy to navigate and maintain as it grew. With the setup complete, I started designing and implementing the initial user interfaces and core layouts, laying the groundwork for the more complex frontend features to be tackled in Sprint 2. This initial work, described further in the *Core Screens Design* section, ensured that

the frontend development was aligned with the application's overall design and functionality goals.

Sprint 1 was a phase that balanced preparation, learning, and implementation. The time invested in relearning NestJS, exploring React, and overcoming the complexities of database replication contributed to a solid foundation for the project. Despite the challenges, such as the unexpected time required for database configuration, the sprint ended successfully, with tangible progress made in both backend and frontend development.

9.2.2. Sprint 2

Sprint 2 focused on integrating features in the frontend such as authentication, user interaction, and the initial steps of the order placement process. This sprint required work across both the frontend and backend, involving complex integrations and careful coordination to ensure smooth system performance.

A significant portion of this sprint was dedicated to implementing the login and registration system, which introduced both **native authentication** and **Google authentication** via Firebase. On the backend, I integrated the **Firebase SDK** to verify Google authentication tokens. This integration required creating a mechanism to differentiate between Google-authenticated users and those using native credentials, as the token retrieved from the request's cookies varies depending on the authentication provider.

For users logging in with Google for the first time, I implemented a flow to register their details in the database, ensuring that the system could handle both user types. This required designing backend logic to manage token verification, user differentiation, and registration processes without compromising security. On the frontend, I developed login and registration interfaces, ensuring they provided a good user experience regardless of the authentication method. This integration was technically challenging but essential for creating a flexible and secure system that supports diverse user authentication options.

Figure 50 illustrates the authentication flow, which begins in the frontend, proceeds through the backend, and concludes by setting an "authToken" cookie in the user's browser. This cookie is used to authenticate subsequent requests to the backend.

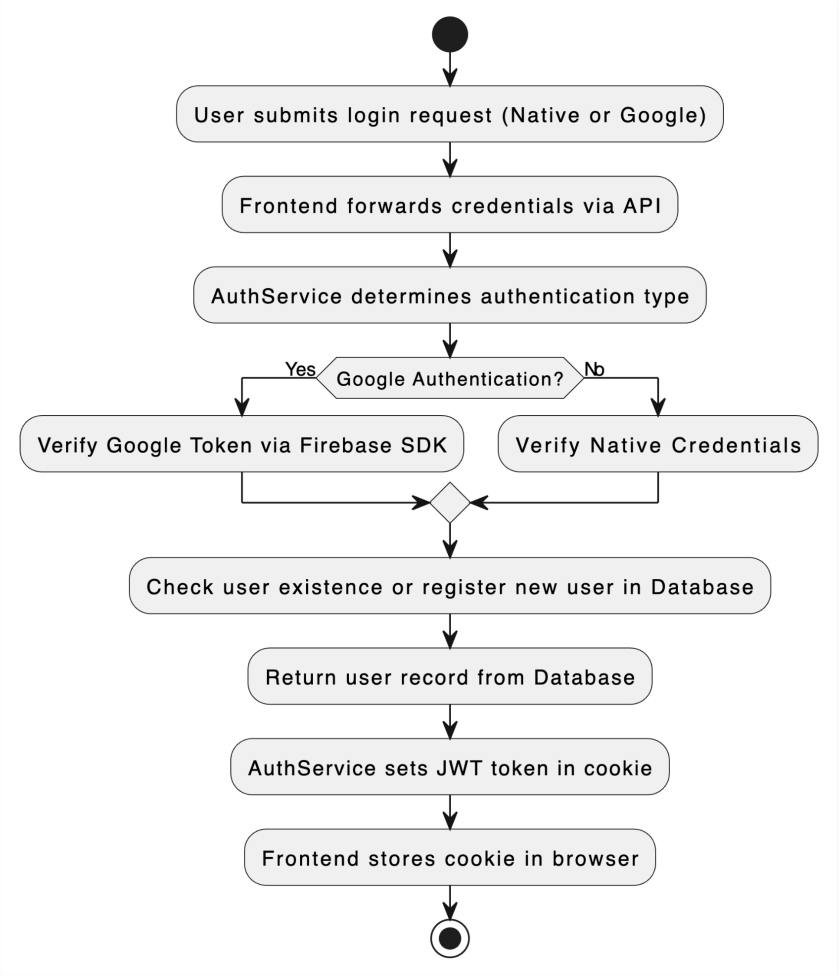


Figure 50: User Authentication Flow (Google and Native Login).

Beyond authentication, I began working on the feed page for customers, which serves as the primary interface for browsing restaurants and accessing their and accessing their specific restaurant pages. One of the core functionalities implemented was the ability to add and remove items from the cart dynamically. To achieve this, I integrated Redux to manage the cart state efficiently. This eliminated the need for reloading the page whenever items were added or removed, resulting in a better user experience. Redux allowed the frontend to maintain a centralized and consistent state, enabling real-time updates to the cart and improving overall usability. Backend requests to fetch the cart are only triggered when the page is reloaded, optimizing performance and reducing unnecessary API calls.

As part of the “place an order” execution flow, I began implementing the logic for syncing cart operations with the backend. This included preparing the system to handle subsequent order processing steps in sprint 3. The work done during this sprint ensured that the cart management system could handle scenarios such as quantity adjustments, item deletions, and state persistence.

Toward the end of the sprint, I initiated the development of the payment service by integrating Stripe. On the backend, I started configuring the interaction with Stripe’s API to

enable the creation of payment sessions. This setup involved designing endpoints and event handlers to manage payment requests and preparing for webhook handling. Although the full payment flow was not completed in this sprint, this foundational work was essential to ensuring that the order placement process could be finalized efficiently in Sprint 3.

```
/**
 * Creates a payment session with Stripe for the given order.
 *
 * @param paymentSessionDto - The payment session data.
 * @returns The created Stripe session.
 */
async createPaymentSession(paymentSessionDto: PaymentSessionDto) {
  const { items, orderId } = paymentSessionDto;
  try {
    const lineItems = items.map((item) => {
      return {
        price_data: {
          currency: 'eur',
          product_data: {
            name: item.name,
          },
          unit_amount: Math.round(item.price * 100),
        },
        quantity: item.quantity,
      };
    });

    const session = await this.stripe.checkout.sessions.create({
      payment_intent_data: {
        metadata: {
          orderId: orderId,
        },
      },
      line_items: lineItems,
      mode: 'payment',
      success_url: `${envs.stripeSuccessUrl}?orderId=${orderId}`,
      cancel_url: `${envs.stripeCancelUrl}?orderId=${orderId}`,
      expires_at: Math.floor(Date.now() / 1000) + 1800,
    });

    this.paymentsGateway.emitPaymentSession(orderId, session.url);

    return session;
  } catch (error) {
    console.error(
      `Error creating payment session for order ${orderId}:`,
      error,
    );

    // Emit an error to the client's room
    this.paymentsGateway.server.to(orderId).emit('payment_session_error', {
      orderId,
      error: 'Failed to create payment session. Please try again.',
    });
  }
}
```

Figure 51: Creation of the Stripe Payment Session.

Sprint 2 was an important phase in the project's development, requiring significant effort to implement and integrate complex features across multiple components. This sprint not only addressed immediate functionality needs but also set the stage for completing the "place an order" flow.

9.2.3. Sprint 3

Sprint 3 marked a significant milestone in the project as it focused on finalizing the "place an order" flow while introducing key features for dynamic updates and notifications. This sprint emphasized building a seamless experience for all stakeholders: customers, restaurants, and couriers. It involved integrating real-time functionalities, designing intuitive interfaces, and developing backend endpoints to support these interactions.

On the frontend, I implemented user interfaces to display the Order Status in real-time for customers. These interfaces enable users to track the progress of their orders, from confirmation to delivery. To achieve this functionality, I integrated *Socket.IO* to establish **real-time communication** between the backend and the client. Events such as "*order confirmed*", "*order prepared*", "*out for delivery*", and "*delivered*" are emitted from the backend and processed on the frontend to dynamically update the user interface without requiring manual page refreshes. This feature significantly enhanced the user experience by providing timely updates and fostering a sense of trust and reliability.

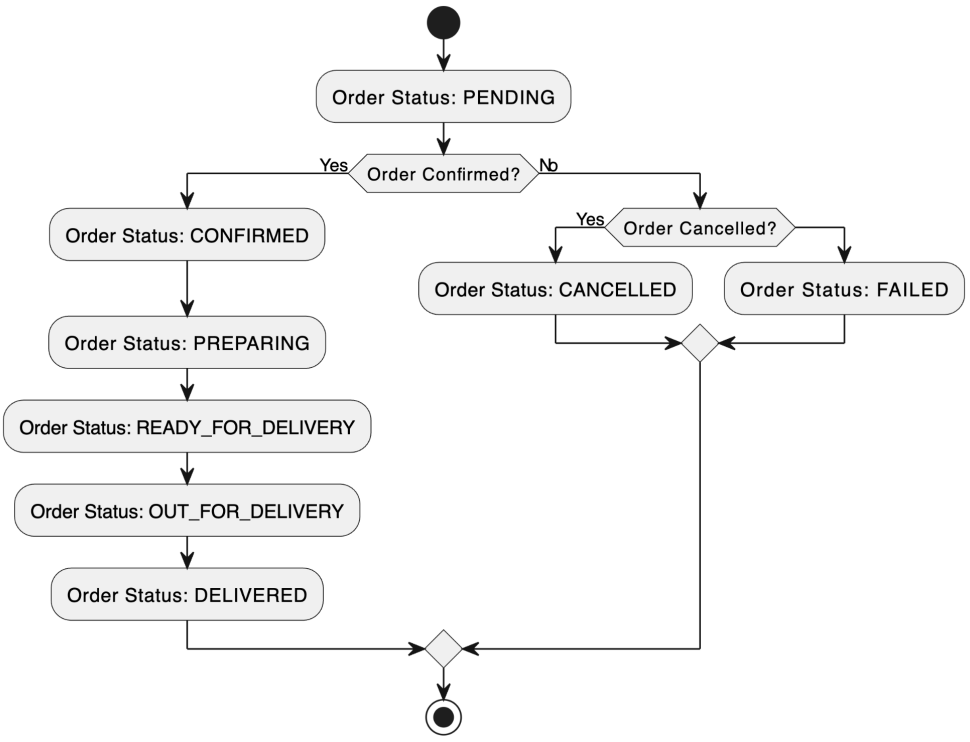


Figure 52: Order Status Flow.

In addition to the customer functionalities, I developed dedicated interfaces for Restaurants and Couriers. Restaurants can view incoming orders and update their statuses (e.g., from "confirmed" to "preparing"), while couriers can mark when they pick up an order or when it is delivered. Figure 52 illustrates the lifecycle of an order through its statuses.

On the backend, I spent quite a time implementing the necessary logic to manage order updates. This included endpoints for restaurants to change order statuses and for couriers to update delivery milestones. Each update triggers corresponding real-time events and notifications, ensuring that all stakeholders remain informed at every step.

The sprint also introduced a notifications system to keep customers informed via email. Utilizing *Mailtrap* as the email notification tool, I implemented a microservice that listens to certain events and sends notifications at key stages of the order lifecycle. Figure 53 displays the controller of this notifications microservice, which manages the logic for processing these events and triggering the appropriate email notifications:

- A confirmation email when the order is placed successfully.
- An email containing a PIN code for delivery verification, sent when the courier picks up the order.
- A final email to notify the customer when the order has been delivered.

```
@Controller()
export class NotificationsController {
  constructor(private readonly notificationsService: NotificationsService) {}

  @EventPattern('order_paid')
  paidOrder(@Payload() paidOrder) {
    this.notificationsService.sendConfirmationMail(paidOrder);
  }

  @EventPattern('courier_assigned')
  courierAssigned(@Payload() orderData) {
    this.notificationsService.sendOrderIncomingEmail(orderData);
  }

  @EventPattern('order_delivered')
  order_delivered(@Payload() orderData) {
    this.notificationsService.sendOrderDeliveredEmail(orderData);
  }
}
```

Figure 53: Notifications Microservice Controller.

By the end of Sprint 3, the “place an order” flow was fully functional, delivering a polished experience for customers, restaurants, and couriers. The combination of real-time updates, notifications, and dedicated interfaces ensured that the system was both user-friendly and operationally efficient.

9.2.4. Sprint 4

Sprint 4 focused on implementing secondary but valuable functionalities, refining the user experience, and conducting aesthetic reviews of the frontend. This sprint marked the final stage of the project implementation.

Key features introduced in this sprint include the **loyalty points system**, which rewards customers for their purchases. This system was designed to incentivize repeated use of the platform by allowing customers to accumulate points based on their spending, which can be redeemed to reduce order delivery fees. Additionally, the customer spending insights page was implemented, offering users a detailed overview of their spending habits, including trends and order summaries. This feature provides an added value to customers by fostering transparency and promoting more mindful consumption.

For both customers and restaurants, I implemented an **order history** page, enabling users to view past orders along with key details such as items purchased, dates, and order statuses. Furthermore, a **ratings** system was introduced, allowing customers to rate their experiences with restaurants.

The sprint also included **profile management**, allowing users to update their personal information, such as the profile image.

In parallel, a review of the frontend's aesthetics was conducted to identify and implement possible enhancements. While improvements were made to refine the design and provide a better user interface, some changes were not done due to time constraints.

Despite not being able to implement all desired features, this sprint reflects the dedication and growth throughout the project. The work completed in Sprint 4 not only enhanced the application's functionality but also reinforced the lessons learned during the development process, making it a rewarding conclusion to the project.

10. Testing

This chapter details the testing processes undertaken to ensure the system meets the defined requirements and functions as intended.

10.1. Testing Functional Requirements

This section evaluates the system's ability to perform its intended operations as defined in the functional requirements. The focus is on validating the core features, such as user authentication, order placement, real-time updates, and notifications, ensuring they function correctly and reliably.

User Registration and Authentication

User Login

Scenario	Result	Observations
The user attempts to log in with valid credentials.	Success	The user is redirected to the application's main screen.
The user attempts to log in with invalid credentials.	Success	The user is notified that the credentials are invalid and is prompted to try again.

User Registration

Scenario	Result	Observations
The user registers filling the form with the required data	Success	The user account is successfully created, and they are redirected to the main application screen.
The user registers using Google authentication and fills the empty inputs of the form	Success	The Google account is validated, and the user is registered if it is their first login.

Profile Management

Scenario	Result	Observations
The user updates their profile information (e.g., name, email).	Success	Profile updates are saved and displayed immediately across the system.
The user uploads a new profile picture.	Success	The profile picture is updated, and the changes are reflected across the application.

Restaurant Management

Menu Management

Scenario	Result	Observations
A restaurant adds a new dish to the menu.	Success	The dish appears immediately in the system and is available for customer orders.
A restaurant updates an existing dish (e.g., price or availability).	Success	Changes are saved and updated across the application.
A restaurant removes a dish from the menu.	Success	The dish is no longer visible to customers and cannot be ordered.

Order Notifications

Scenario	Result	Observations
A customer places a new order.	Success	The restaurant receives a real-time notification about the new order.
Multiple customers place orders simultaneously.	Success	The restaurant receives notifications for all orders without delays or issues.

Order Status Updates

Scenario	Result	Observations
The restaurant updates an order status to “Preparing.”	Success	Customers are notified in real time about the updated status.
The restaurant updates an order status to “Ready for Delivery.”	Success	Couriers and customers are notified, and the system reflects the updated status.

Menu Browsing and Ordering

Browsing and Searching

Scenario	Result	Observations
A customer browses a restaurant menu and views dish details, including prices.	Success	Menu and dish details load correctly.
A customer searches for restaurants or menu dishes by keywords or categories.	Success	Search results are accurate.
A customer applies filters to refine search results (e.g., cuisine type).	Success	Filters work as expected, narrowing down results based on the selected criteria.

Placing Orders

Scenario	Result	Observations
A customer checks out the cart items and places an order.	Success	Orders are successfully processed with the correct address and payment details recorded.
A customer views their order history and reorders a previously purchased dish.	Success	Order history loads correctly, and reordering functionality works without errors.

Spending Insights and Loyalty Points

Scenario	Result	Observations
A customer views information about their spending (e.g., total	Success	Spending insights load correctly and provide

spend over time).		accurate data.
A customer earns loyalty points after making a purchase.	Success	Points are awarded correctly and displayed in the customer's account.
A customer uses loyalty points to reduce delivery fees for an order.	Success	Loyalty points are applied correctly, and the reduced delivery fee is reflected in the final price.

Real-Time Updates and PIN Verification

Scenario	Result	Observations
A customer receives a unique PIN when their order is assigned to a courier.	Success	PINs are generated and sent to the customer via email.
A customer tracks their order in real-time and sees updates (e.g., "Preparing," "Out for Delivery").	Success	Order status updates appear in real time, and the tracking feature works without delays or errors.

Delivery Management

Updating Delivery Status

Scenario	Result	Observations
A courier updates the status of an order to "Picked Up."	Success	Status updates are reflected in real time for both the customer and the restaurant.
A courier updates the status of an order to "Delivered."	Success	The system accurately reflects the final delivery status in the order history and notifications.

PIN Verification

Scenario	Result	Observations
A courier enters the PIN provided by the customer to confirm delivery.	Success	The system verifies the PIN, marks the order as delivered, and updates the customer's order history.

Ratings

Customer Ratings

Scenario	Result	Observations
A customer rates a restaurant after completing an order.	Success	The rating is recorded accurately and displayed in the restaurant's feedback.

Restaurant Feedback

Scenario	Result	Observations
A restaurant views customer feedback and ratings.	Success	Feedback loads correctly, providing restaurants with insights to improve service quality.

Admin Dashboard

User and Profile Management

Scenario	Result	Observations
An administrator manages user accounts, restaurant profiles, and delivery personnel.	Success	Changes are saved accurately, and updates are reflected across the system.

System Analytics

Scenario	Result	Observations
An administrator views system analytics, including order volumes and user activity.	Success	Analytics data loads correctly, providing useful insights for system performance and user behavior.

10.2. Testing Non-Functional Requirements

This section focuses on evaluating the non-functional aspects of the system to ensure it meets the required performance, reliability, scalability, and usability standards. Non-functional requirements are critical for determining the overall quality of the application and its ability to handle real-world scenarios effectively.

NFR 1 - Reliability

Acceptance criteria	The system must achieve 99% uptime.
Result	Achieved
Observations	The system achieved 99.9% uptime, meeting the acceptance criteria. This was ensured by using services with high SLA guarantees: AWS EC2 and RDS guarantees at least 99% uptime Monthly [51], while Netlify provided 100% of availability the last 90 days [52], ensuring reliability across the backend and frontend infrastructure.

NFR 2 - Scalability

Acceptance criteria	The system should support 200 requests, ensuring that the average response time remains under 2s.
Result	Achieved
Observations	With Apache JMeter I simulated 200 requests, the average response time was 1.07s

NFR 3 - Maintainability

Acceptance criteria	Code must follow clean coding practices and modular architecture, with clear documentation provided.
Result	Achieved
Observations	The code follows clean coding practices, modular architecture, and includes clear documentation.

NFR 4 - Appearance

Acceptance criteria	It should receive positive feedback (more than 3.5 on average in a 1-5 scale) from the test users (10 test users). The design must be responsive, ensuring a seamless experience across various devices and screen sizes.
Result	Achieved
Observations	Feedback was collected from 10 test users, who evaluated the application on a scale of 1 to 5. The average rating achieved was 4,05. The calculation was based on the following individual scores: ● 4,5 × 4 ● 4 × 3 ● 3.75 × 2 ● 3 × 1

NFR 5 - Ease of Use

Acceptance criteria	The system should receive an average rating of more than 3.5 (on a 1-5 scale) from test users (10 test users) in usability testing. The application must have an intuitive interface with clear navigation and accessible features.
Result	Achieved
Observations	Feedback was collected from 10 test users, who evaluated the application on a scale of 1 to 5. The average rating achieved was 4,85. The calculation was based on the following individual scores: ● 5 × 7 ● 4,5 × 3

11. Laws and Regulations

Developing a food delivery web application involves compliance with several laws and regulations to ensure the project operates within legal and ethical boundaries.

General Data Protection Regulation (GDPR) [53]: As the application processes and stores user data, including personal information such as names and addresses, GDPR compliance is critical. Key requirements include:

- **Data Minimization (Article 5)**: Collecting only the necessary data for the service.
    - **Consent (Article 7):** Obtaining explicit consent before collecting or processing personal data.
    - **Right to Access and Deletion (Articles 15 and 17)**: Allowing users to view and delete their data upon request.
    - **Data Security (Article 32)**: Implementing measures to protect user data against breaches.

The application is designed to comply with GDPR by implementing features such as explicit user consent mechanisms, clear privacy policies, secure data storage with encryption and separation of data across databases, and tools for users to access or delete their information.

- **Consumer Protection Laws [54]**: Consumer protection regulations ensure fair treatment of users in the marketplace. Key considerations include:
    - **Transparency**: Displaying clear and accurate pricing, delivery fees, and service terms.
    - **Refund Policies**: Ensuring users can request refunds in cases of service failure.

The application includes detailed pricing information and a dedicated section for displaying allergen information for each menu item. Refund policies will not be implemented due to time constraints, as adding this functionality would exceed the project's scope.

- **Food Information Law (Allergens) [55]:** Providing mandatory disclosure of allergens in menu items.

The application includes a dedicated section for displaying allergen information for each menu item.

12. Project Management Results

This chapter presents an analysis of the project's management outcomes, comparing the initial planning with the final results. It examines how the original timelines, objectives, and methodologies were adapted throughout the development process to address challenges.

12.1. Methodology

The Scrum framework was effectively implemented but required some adjustments due to obstacles and delays in the development process. While regular meetings were planned, some were omitted when progress was insufficient to justify discussions. Despite these interruptions, consistent communication between the Scrum Master (Cristina, the director) and the Development Team (Emanuel, the student) ensured that the project stayed on track and overcame challenges.

The tools selected for project management and development remained consistent throughout. Jira was used to organize tasks and monitor progress, and Git was employed for version control. However, due to time constraints and the prioritization of completing functionalities, GitFlow was not used as intended. The focus shifted toward delivering results, bypassing the formal branching and merging strategies GitFlow states.

Debugging relied on logs generated by the backend server, which proved sufficient for identifying and addressing issues during development.

The methodology had to adapt to the realities of time constraints and development challenges, requiring adjustments such as skipping meetings and simplifying version control practices. Despite these adaptations, the tools and communication processes in place ensured the project stayed on track. Ultimately, the necessary work was completed, demonstrating the importance of flexibility in achieving the project's goals.

12.2. Time Planning

This section provides an analysis of the time allocated to each phase of the project compared to the actual time spent. Table 10 provides a detailed breakdown of the estimated and actual hours for tasks such as project management, development phases, documentation and defense preparation. By examining the time differences, the table highlights areas where additional effort was required and phases where time was saved.

Code	Task	Estimated Time	Actual Time	Time Difference
	Project Management	**120**	**120**	-
	Development	**359**	**377**	+18
I	***Inception***	**63**	**75**	+12
S1	***Sprint 1***	**86**	**86**	-
S2	***Sprint 2***	**80**	**92**	+12
S3	***Sprint 3***	**62**	**68**	+6
S4	***Sprint 4***	**68**	**56**	-12
	Project Documentation and Defense Preparation	**60**	**72**	+12
TOTAL		**539**	569	+30

Table 10: Estimated and actual hours of the project.

12.3. Budget

Code	Task	Estimated Costs (€)	Actual Costs (€)	Costs Difference (€)
	Project Management	**4.008,72**	**4.008,72**	-
	Development	**11.490,12**	**11.979**	+488,88
I	***Inception***	**1.854,3**	**2.367,66**	+513,36
S1	***Sprint 1***	**2.825,58**	**2.825,58**	-
S2	***Sprint 2***	**2.638,32**	**2.833,74**	+195,42
S3	***Sprint 3***	**2.019,42**	**2.206,68**	+187,26
S4	***Sprint 4***	**2.152,50**	**1.745,34**	-407,16
	Project Documentation and Defense Preparation	**1.954,80**	**2.345,76**	+390,96
TOTAL		**17.453,64**	**18.333,48**	+879,84

Table 11: Estimated and actual costs of the project.

Now, we'll provide an overview of the main project phases where time differences (and consequently, costs) were observed. Each explanation highlights the reasons behind the deviations from the initial estimates:

1. **Inception:** This phase required additional time primarily due to the **Design Chapter**. The effort involved in specifying the system architecture, diagrams, and models exceeded initial estimates. Refining these elements was critical to ensure a solid foundation for the project's implementation.

2. **Sprint 2:** The extra time was due to the complexities of implementing Google authentication via Firebase, including token validation and backend integration. Additional hours were also spent developing real-time cart management and synchronizing data between the frontend and backend.

3. **Sprint 3:**This phase required more time to complete real-time order tracking with **Socket.IO** and the notifications system. Integrating these features and ensuring their smooth operation added complexity to the implementation process.

4. **Sprint 4:** Sprint 4 took less time than estimated because several functionalities, such as the loyalty points system and profile management, were straightforward to implement. Some planned aesthetic improvements to the frontend were also deferred, further reducing the time required.

5. **Project Documentation and Defense Preparation**: This phase required 12 additional hours compared to the estimate. The additional time was spent refining the documentation and condensing the main report to improve clarity and readability.

13. Conclusions

This chapter reflects on the overall journey of the project, summarizing the key achievements, challenges, and lessons learned.

13.1. Project Final Conclusions

The completion of this project marks a significant achievement, successfully fulfilling the primary goal of developing a robust, feature-rich food delivery application using microservices. The implemented functionalities, such as real-time order tracking, reflect a careful balance of design, innovation, and practicality. Despite the challenges faced, including delays and adjustments to initial plans, the project demonstrates a clear alignment with its objectives and a strong application of software engineering principles.

Throughout the process, the integration of key technologies such as NestJS, React, Socket.IO, Docker, AWS and Kubernetes showcased the ability to build scalable and distributed systems.

The use of an Agile methodology provided a framework to iteratively develop and refine the system, adapting to challenges and prioritizing critical functionalities. While not all planned features were implemented, the decisions made throughout ensured a functional and polished application that met the project's key requirements.

This project also represents a substantial learning experience, reinforcing skills in database management, system integration, distributed architectures, and real-time communication. Moreover, it highlights the importance of flexibility in project management, as well as the ability to make pragmatic decisions when faced with time constraints.

The resulting application demonstrates the effective application of theoretical knowledge to practical implementation. While there are areas that can be further refined, this project provides a solid foundation for future work and has contributed meaningfully to my development as a software engineer.

13.2. Knowledge Integration

Throughout the project, I have applied knowledge acquired from various subjects I have previously studied. This integration of knowledge has been crucial in shaping the development and execution of the project.

ER (Requirements Engineering): Requirements Engineering subject has helped particularly in the areas of stakeholders, problem definition, justification, scope, and specification.

GEP (Project Management): The project management principles from GEP have been essential for structuring the project and ensuring it is done properly. Time planning, scope definition and budget management are very important for keeping the project on track.

IES (Introduction to Software Engineering): In the Introduction to Software Engineering course I learned how to do the conceptual and behavioral models, which have been applied to define the system's architecture and user interactions. These models helped to visualize the structure and behavior of the system before implementation, making it easier to understand how components would interact.

PES (Software Engineering Project): The Software Engineering Project course provided valuable insights into work methodology, CI/CD processes, software architecture, and project implementation. This subject has been quite beneficial as it gave me the skills to effectively manage and execute a project. Since I am working alone on this project, the knowledge gained from this course has been crucial in guiding my approach to development, ensuring that I follow best practices and manage the project's workflow efficiently.

AS (Software Architecture): The Software Architecture course introduced me to essential concepts such as hexagonal architecture and microservices. These principles have been directly applied to the design of the project. Building on this foundation, I have expanded my knowledge through independent research, reading books and technical articles, watching videos, and completing additional courses on my own.

BD, DBD, and CBDE (Databases, Database Design, and Specialized Database Concepts): These subjects have been very important in building a strong foundation for understanding and working with databases. They have provided me with the skills to design, manage, and optimize database systems effectively. Additionally, they introduced me to advanced strategies, such as the master-slave replication setup implemented in this project, and expanded my knowledge of NoSQL databases like MongoDB, which was used for the Orders Microservice.

PRO1, PRO2, and EDA (Programming 1 and 2, and Algorithms and Data Structures): These subjects have been very helpful in developing a strong foundation in programming and problem-solving. They provided me with the skills to write efficient and well-structured code, as well as a solid understanding of key programming concepts. Additionally, they introduced me to algorithms and data structures, which are fundamental for optimizing system performance and ensuring scalability.

13.3. Technical Competences

This section explores the technical competences addressed throughout the project. Each competence is analyzed in terms of the depth at which it was worked on and the specific challenges it helped to overcome.

CES1.1: Desarrollar, mantener y evaluar sistemas y servicios software complejos y/o críticos. [En profunditat]

This competence was addressed through the design and development of a microservices architecture for the food delivery application. The system's complexity required a modular approach, with independent services like Orders, Customers, Restaurants, and Notifications. Each microservice had its own database and logic, necessitating careful consideration of scalability, fault tolerance, and maintainability.

The use of NestJS allowed for a structured and consistent approach to building these services. Key challenges, such as ensuring robust inter-service communication, were resolved using NATS as a message broker and design patterns such as choreographed and orchestrated sagas. This design enabled asynchronous messaging and event-driven communication, essential for critical operations like order updates and notifications.

CES1.2: Dar solución a problemas de integración en función de las estrategias, de los estándares y de las tecnologías disponibles. [Bastante]

This competence was addressed by overcoming integration challenges across services and external tools. One notable example was integrating Firebase for Google authentication. This required designing a workflow to validate Google tokens, differentiate users based on authentication methods, and register first-time Google users seamlessly.

Additionally, integrating Stripe for payments posed its own challenges, such as handling payment sessions, verifying transactions, and ensuring secure communication between the backend and Stripe's API. Webhook processing was implemented to react to payment events like *charge.succeeded* or *checkout.session.expired*, aligning with Stripe's standards.

The selection of integration strategies was driven by the need for scalability, industry standards, and compatibility with the chosen technologies.

CES1.4: Desarrollar, mantener y evaluar servicios y aplicaciones distribuidas con soporte de red. [En profunditat]

This competence was addressed by developing and deploying a distributed system architecture that operates over the internet. The frontend was deployed using **Netlify**, a platform optimized for hosting static websites and single-page applications, ensuring fast delivery and high availability. The backend was deployed on an **AWS EC2** instance with **Kubernetes**, which provided container orchestration for the microservices, enabling scalability, fault tolerance, and efficient resource utilization. Additionally, the PostgreSQL databases were deployed on **Amazon RDS**.

Socket.IO was a key technology integrated into the system to provide real-time communication, enabling features such as live order status updates. This ensured seamless interaction between the backend and clients, providing a responsive and engaging user experience over the network.

CES1.5: Especificar, diseñar, implementar y evaluar bases de datos. [Un poco]

Although it is not the primary focus of the thesis, this competence was addressed through the design and implementation of databases for the microservices. MongoDB was chosen for the Orders Microservice due to its flexibility in handling dynamic, unstructured data, ideal for the varied nature of order details.

For relational data, PostgreSQL was used in other microservices, such as Customers and Restaurants. Implementing a master-slave replication strategy provided fault tolerance and scalability, ensuring data availability even under high demand.

CES2.1: Definir y gestionar los requisitos de un sistema software. [Un poco]

This competence was addressed at a basic level by defining and managing the system's requirements, with the **Specification** chapter serving as a foundation for this process. The use case diagrams and their descriptions provided a clear definition of core functionalities, ensuring the development process remained aligned with the system's intended functionality.

The conceptual model offered a structured representation of the system's entities and relationships, supporting accurate implementation and maintaining data consistency. Additionally, the behavioral model captured the dynamic interactions within the system, specifying how its components would respond and evolve in different scenarios.

Beyond functional requirements, non-functional requirements such as scalability and reliability were also considered. These requirements guided critical architectural decisions, including the adoption of a microservices architecture, the integration of real-time communication, and the implementation of database replication strategies.

13.4. Future Work

While the project has met its primary objectives, several potential enhancements could improve its functionality, maintainability, and user experience:

- **Enhance Security:** Strengthen backend and frontend security by implementing stricter authentication mechanisms, securing API endpoints, and protecting against vulnerabilities like XSS.

- **Improve the User Interface (UI):** Refine the design to make it more visually appealing and user-friendly, incorporating feedback to ensure better responsiveness and usability.

- **Refactor the Codebase:** Reorganize and optimize the code for better maintainability, modularity, and adherence to design patterns, making it easier to extend and document.

- **Enhance Observability and Metrics:** Integrate monitoring tools to track performance and errors in real-time, creating dashboards for efficient debugging and system management.

- **Optimize Deployment on AWS**: Improve infrastructure by optimizing Kubernetes configurations, automating CI/CD pipelines further, and ensuring high availability for all services.

- **Implement Real-Time Courier Tracking**: Add GPS-based tracking for couriers to provide customers with precise updates on delivery progress and improve navigation for couriers.

- **Enhance Maps Functionality**: Improve map displays by adding route details, traffic information, and interactive features to benefit both users and couriers.

- **Develop a Mobile App:** Create a mobile application to provide couriers with real-time navigation and delivery management, while offering customers a better user experience optimized for mobile devices.

- **Develop a recommendation system:** This feature would analyze user preferences, past orders, and behavioral patterns to suggest dishes, restaurants, or promotional offers tailored to each customer.

- **Enhance the loyalty program points system:** Expand the current system to include more customizable and engaging reward options, such as tiered membership levels (e.g., silver, gold, platinum) that offer increasing benefits. Additionally, integrate gamification elements like progress tracking, badges, or challenges to make the loyalty program more interactive and rewarding, fostering greater customer retention and engagement.

These enhancements would take the application to the next level, addressing current limitations and aligning it with higher standards of performance and usability.

References

[1] Ordering in: The rapid evolution of food delivery | McKinsey. [Accessed 2 September 2024]. Source: www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/ordering-in-the-rapid-evolution-of-food-delivery

[2] Uber Eats Revenue and Usage Statistics (2024) - Business of Apps. Online. [Accessed 2 September 2024]. Source: https://www.businessofapps.com/data/uber-eats-statistics/

[3] SOLUTIONS, JPLoft. The Growth of the Food Delivery Industry: Statistics and Industry Analysis. Medium. 3 May 2024. [Accessed 18 September 2024]. Source: https://medium.com/@jploft/the-growth-of-the-food-delivery-industry-statistics-and-industry-analysis-f16235ff366d

[4] Gluck, Adam. Introducing Domain-Oriented Microservice Architecture. Uber Blog. 23 July 2020. [Accessed 23 September 2024]. Source: https://www.uber.com/en-IN/blog/microservice-architecture/

[5] Uber Eats | Comida a Domicilio y Para Llevar | Haz un Pedido Online de Restaurantes Cercanos. Online. [Accessed 2 September 2024]. Source: https://www.ubereats.com/es

[6] Pide a domicilio online con Glovo en España: comida, la compra del súper, productos de parafarmacia. Online. [Accessed 2 September 2024]. Source: https://glovoapp.com/es/es/

[7] Comida a domicilio y para llevar | Just Eat. Online. [Accessed 2 September 2024]. Source: https://www.just-eat.es/

[8] Tristancho, Camilo. Top 15 Project Management Methodologies: An Overview. ProjectManager. Online. 10 September 2024. [Accessed 20 September 2024]. Source: https://www.projectmanager.com/blog/project-management-methodology

[9] Scrum Guide | Scrum Guides. Online. [Accessed 20 September 2024]. Source: https://scrumguides.org/scrum-guide.html

[10] Why IT Infrastructure Should Use Agile Sprint Cycles (SCRUM) - Agdiwo. Online. [Accessed 20 September 2024]. Source: https://www.agdiwo.com/en/agile-infrastructure/

[11] What Are The Phases Of Scrum? Online. [Accessed 20 September 2024]. Source: https://www.workamajig.com/blog/scrum-methodology-guide/scrum-phases

[12] Agile scrum roles and responsibilities. Online. [Accessed 20 September 2024]. Source: https://www.atlassian.com/agile/scrum/roles

[13] The Scrum Team | Scrum Alliance. Online. [Accessed 21 September 2024]. Source: https://resources.scrumalliance.org/Article/scrum-team

[14] Maldonado, Sergio Humberto Guzmán. Conociendo GitFlow. Medium. 4 November 2022. [Accessed 21 September 2024]. Source: https://medium.com/@sergiohumberto27/conociendo-gitflow-a588716fbc28

[15] Facultat d'Informàtica de Barcelona. Normativa del treball de final de grau en Enginyeria Informàtica de la Facultat d'Informàtica de Barcelona. 29 January 2020.

[Accessed 30 September 2024]. Source: https://www.fib.upc.edu/sites/fib/files/documents/estudis/normativa-tfg-mencio-addicional-gei.pdf
[16] Gantt Charts Explained: A Practical Guide for Project Managers. [Accessed 1 October 2024]. Source: https://www.teamgantt.com/what-is-a-gantt-chart
[17] Firebase Cloud Messaging | Send notifications across platforms. Firebase. Online. [Accessed 2 October 2024]. Source: https://firebase.google.com/products/cloud-messaging?hl=es-419
[18] Capa gratuita de AWS | Cloud computing gratis |AWS. Online. [Accessed 2 October 2024]. Source: https://aws.amazon.com/es/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all
[19] Sueldos | Indeed.com. [Accessed 6 October 2024]. Source: https://es.indeed.com/career/salaries?from=gnav-title-webapp/salaries
[20] España | Cataluña | ¿Cuántos días laborables en el año 2024? [Accessed 7 October 2024]. Source: https://www.dias-laborables.es/cuantos_dias_laborables_en_ano_2024_Catalu%C3%B1a.htm
[21] Mac Studio. Apple (ES). [Accessed 7 October 2024]. Source: https://www.apple.com/es/shop/buy-mac/mac-studio/cpu-12-núcleos-gpu-30-núcleos-neural-engine-16-núcleos-32-gb-memoria-512gb
[22] Teclado inalámbrico | Logitech MX Keys S. MediaMarkt. [Accessed 7 October 2024]. Source: https://www.mediamarkt.es/es/product/_teclado-inalambrico-logitech-mx-keys-s-bluetooth-usb-teclas-programables-carga-rapida-retroiluminacion-multidispositivo-windows mac-negro-1554219.html
[23] Ratón inalámbrico | Logitech MX Master 3S. [Accessed 7 October 2024]. Source: https://www.mediamarkt.es/es/product/_raton-logitech-mx-master-3s-inalambrico-8000-ppp-botones-personalizables-carga-rapida-clic-silencioso-multidispositivo-multisistema-negro-1534055.html
[24] Monitor LG UltraGear 27GS75Q-B 27". PcComponentes. [Accessed 7 October 2024]. Source: https://www.pccomponentes.com/monitor-lg-ultragear-27gs75q-b-27-led-ips-qhd-200hz-freesync
[25] Monitor | LG 27MK600M-W 27". MediaMarkt. [Accessed 7 October 2024]. Source: https://www.mediamarkt.es/es/product/_monitor-lg-27mk600m-w-27-full-hd-ips-led-5-ms-radeon-freesync%E2%84%A2-negro-y-plata-1467282.html
[26] Apple iPhone 13. MediaMarkt. [Accessed 7 October 2024]. Source: https://www.mediamarkt.es/es/product/_apple-iphone-13-medianoche-128-gb-5g-61-oled-super-retina-xdr-chip-a15-bionic-ios-1518038.html
[27] Matriz de sostenibilidad. FIB. [Accessed 12 January 2025]. Source: https://www.fib.upc.edu/sites/fib/files/documents/estudis/tfg-informe-sostenibilitat-2018.pdf

[28] Qüestionari Genèric Estudiants d'Enginyeria. Google Docs. [Accessed 9 October 2024]. Source: https://docs.google.com/forms/d/e/1FAIpQLSfVgBxcxZfh7pB_OVRUNGQmRpFDFlhAskukNcpQBowLRF4-sA/viewform
[29] OLIVÉ RAMON, A. (2007). Conceptual modeling of information systems. Berlin: Springer. ISBN: 9783540393894.
[30] Especificació en UML: Esquema conceptual de les dades. Presentation from the course Software Engineering Introduction (IES) in the Bachelor's degree in Informatics Engineering, taught at the FIB (Barcelona School of Informatics) of the UPC (Polytechnic University of Catalonia).
[31] UML - Behavioral Diagram vs Structural Diagram. Online. [Accessed 21 November 2024]. Available from: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram/
[32] Kleppmann, Martin (2017). Designing Data-Intensive Applications. O'Reilly Media. ISBN: 978-1-449-37332-0.
[33] Newman, Sam (2022). Building Microservices. O'Reilly Media. ISBN: 978-1-492-03402-5.
[34] DEUTSCH, Daniel. Understanding MVC Architecture with React. Online. 2 December fi2020. [Accessed 21 November 2024]. Available from: https://medium.com/createdd-notes/understanding-mvc-architecture-with-react-6cd38e91fefd
[35] Richardson, Chris. Microservices patterns: with examples in Java. Manning Publications, 2019. ISBN: 978-1-61729-454-9.
[36] Redux - A JS library for predictable and maintainable global state management | Redux. Online. [Accessed 24 December 2024]. Available from: https://redux.js.org/
[37] OZKAYA, Mehmet. API Gateway Pattern. Design Microservices Architecture with Patterns & Principles. Online. 9 March 2023. [Accessed 24 December 2024]. Available from:https://medium.com/design-microservices-architecture-with-patterns/api-gateway-pattern-8ed0ddfce9df
[38] PATCH - HTTP | MDN. Online. 19 December 2024. [Accessed 24 December 2024]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH
[39] Prisma Client API | Prisma Documentation. Online. [Accessed 24 December 2024]. Available from: https://www.prisma.io/docs/orm/reference/prisma-client-reference
[40] Data Transfer Object (DTO). Online. [Accessed 25 December 2024]. Available from: https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/dto
[41] Stripe | Get Started. Online. [Accessed 25 December 2024]. Available from: https://docs.stripe.com/get-started
[42] Documentation | NestJS - A progressive Node.js framework. Online. [Accessed 25 December 2024]. Available from: https://docs.nestjs.com
[43] Vite. Online. [Accessed 30 December 2024]. Available from: https://vite.dev
[44] Hot Module Replacement. webpack. Online. [Accessed 30 December 2024]. Available from: https://webpack.js.org/concepts/hot-module-replacement/

[45] MUI: The React component library you always wanted. Online. [Accessed 30 December 2024]. Available from: https://mui.com/

[46] Redux Fundamentals, Part 2: Concepts and Data Flow | Redux. Online. 29 May 2024. [Accessed 30 December 2024]. Available from: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow

[47] React Router Home. Online. [Accessed 30 December 2024]. Available from: https://reactrouter.com/home

[48] DevTalles | +Talento. DevTalles. Hoja de Atajos NestJS. Online. [Accessed 1 January 2025]. Available from: https://cursos.devtalles.com/pages/mas-talento

[49] Cursos en línea: aprende de todo y a tu propio ritmo. Udemy. Online. [Accessed 1 January 2025]. Available from: https://www.udemy.com/

[50] SHARMA, supriya. Unveiling the Power and Pitfalls of Master-Slave Architecture. Medium. Online. 24 May 2023. [Accessed 1 January 2025]. Available from: https://medium.com/@cpsupriya31/understanding-master-slave-architecture-uses-and-challenges-2acc907de7c4

[51] Acuerdos de nivel de servicios (SLA) de AWS. *Amazon Web Services, Inc.* Online. [Accessed 3 January 2025]. Available from: https://aws.amazon.com/es/legal/service-level-agreements/

[52] Netlify Status. Online. [Accessed 3 January 2025]. Available from: https://www.netlifystatus.com/

[53] Reglamento general de protección de datos. Online. [Accessed 9 December 2024]. Available from: https://gdprinfo.eu/es

[54] Los derechos de la persona consumidora en el mundo digital. Sus derechos avanzan | Ministerio de Derechos Sociales, Consumo y Agenda 2030. Online. [Accessed 9 December 2024]. Available from: https://www.dsca.gob.es/es/consumo/informacion-persona-consumidora/material-divulgacion/dia-mundial-derechos-personas-consumidoras/derechos-persona-consumidora-mundo-digital

[55] Aesan - Agencia Española de Seguridad Alimentaria y Nutrición. Online. [Accessed 9 December 2024]. Available from: https://www.aesan.gob.es/AECOSAN/web/seguridad_alimentaria/subdetalle/futura_legislacion.ht

Annexes

This section includes additional material that complements the main document. Each annex provides further details to support and expand on the content covered in the core sections.

1. Annex A: Additional Use Case Descriptions

This annex contains the descriptions of use cases not included in the main body of the document. These use cases cover secondary functionalities that provide additional insights into the application's behavior and interactions.

Users: User Registration and Authentication

Use case	Log Out
Actor	User
Preconditions	● The user must be logged in.
Trigger	The user decides to log out of their account.
Principal Use Scenario	1. The user indicates they want to log out. 2. The system ends the user's session.

Use case	Recover Password
Actor	User
Preconditions	● The user must have a registered account.
Trigger	The user forgets their password and requests to reset it.
Principal Use Scenario	1. The user indicates that they have forgotten their password. 2. The system prompts the user to enter their registered email address. 3. The user provides the email address. 4. The system sends a password recovery link to the provided email. 5. The user checks their email and clicks the recovery link or enters the code. 6. The system prompts the user to enter and confirm a new password. 7. The user enters the new password, and the system updates the password. 8. The system confirms the password reset and allows the user to log in with the new password.

Extensions	4a. If the provided email is not registered, the system informs the user.

Restaurants: Restaurant Management

Use case	Update Menu Dishes
Actor	Restaurant Manager
Preconditions	● The restaurant must have an active account, and the user must be logged in.
Trigger	The restaurant manager decides to update the details of an existing menu dish.
Principal Use Scenario	1. The restaurant manager indicates that they want to update a menu dish. 2. The system shows the current details of the dish. 3. The manager edits the details (e.g., name, description, price). 4. The system validates the new data and checks for conflicts. 5. If valid, the system updates the dish and saves the changes. 6. The system confirms the update and displays a success message.
Extensions	5a. If the manager inputs invalid data, the system prompts for corrections before saving.

Use case	Remove Menu Dishes
Actor	Restaurant Manager
Preconditions	● The restaurant must have an active account, and the user must be logged in.
Trigger	The restaurant manager decides to remove a menu dish from the restaurant's menu.
Principal Use Scenario	1. The restaurant manager indicates that they want to remove a menu dish. 2. The system prompts the manager to confirm the removal. 3. The manager confirms the removal. 4. The system removes the dish from the menu and updates the display.

	5. The system displays a success message confirming the removal.

Use case	Receive Notifications
Actor	Restaurant Manager
Preconditions	● The restaurant must have an active account, and the user must be logged in.
Trigger	A new order is placed, or the order status changes.
Principal Use Scenario	1. The system detects a new order or an update in the order status (e.g., order confirmed, preparing, delivered). 2. The system sends a real-time notification to the restaurant manager through the web application.

Customers: Menu Browsing and Ordering

Use case	Search Restaurants
Actor	Customer
Preconditions	● The customer is logged into the system.
Trigger	The customer initiates a search to find restaurants.
Principal Use Scenario	1. The customer enters keywords (e.g., restaurant name). 2. The system displays a list of matching restaurants. 3. The customer selects a restaurant from the list.
Extensions	2a. If no restaurants match the search, the system displays a "no results found" message.

Use case	Apply Filters
Actor	Customer
Preconditions	● The customer has initiated a restaurant search.
Trigger	The customer chooses to apply filters to narrow down search results.
Principal Use Scenario	1. The customer selects filters such as cuisine type.

	2. The system filters the restaurant list based on the selected criteria. 3. The customer views the filtered restaurant list.
Extensions	1a. The customer may reset the filters.

Use case	Browse Menu
Actor	Customer
Preconditions	● The customer has selected a restaurant.
Trigger	The customer requests to view the menu of a selected restaurant.
Principal Use Scenario	1. The customer clicks on a restaurant to view its menu. 2. The system retrieves and displays the restaurant's available menu dishes. 3. The customer scrolls through the list of menu dishes.

Use case	View Dish Details
Actor	Customer
Preconditions	● The customer is browsing a restaurant's menu.
Trigger	The customer selects a specific dish from the menu to view more details.
Principal Use Scenario	1. The customer clicks on a menu dish. 2. The system displays the dish details, including description, price, and allergens. 3. The customer views the detailed information.

Use case	View Order History
Actor	Customer
Preconditions	-
Trigger	The customer wants to view their past orders.
Principal Use Scenario	1. The customer indicates that they want to view their order history. 2. The system retrieves the customer’s past orders. 3. The customer views details of their past orders, including dates, restaurant names, and order

	amounts.
Extensions	3a. If no previous orders exist, the system displays a message indicating no history is available.

Use case	Reorder Dishes
Actor	Customer
Preconditions	● The customer has viewed their order history and selected a past order.
Trigger	The customer decides to reorder a previous dish.
Principal Use Scenario	1. The system displays the previous order details. 2. The customer selects the option to reorder. 3. The system adds the same dishes to the cart. 4. The customer proceeds with the regular order placement process.
Extensions	3a. If a dish is no longer available, the system informs the customer.

Use case	Earn Loyalty Points
Actor	Customer
Preconditions	-
Trigger	The customer completes an order.
Principal Use Scenario	1. The customer places an order. 2. The system calculates the appropriate loyalty points based on the order value. 3. The system updates the customer's account with the newly earned points.

Couriers: Delivery Management

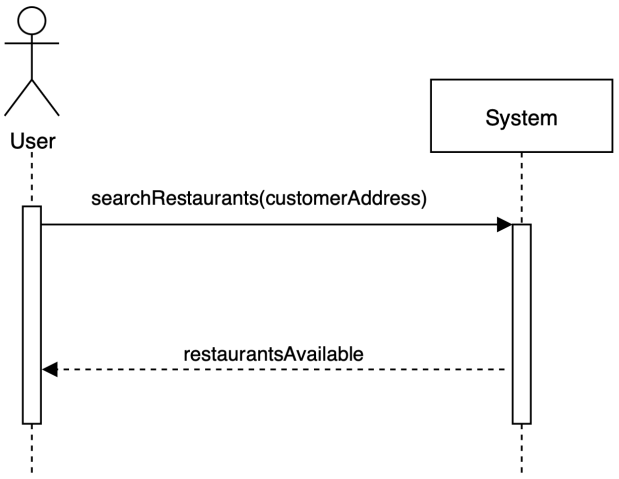
Use case	Update Delivery Status
Actor	Courier
Preconditions	● The courier must be assigned a delivery task. ● The courier must have access to the system via the

	app.
Trigger	The courier updates the delivery status during the delivery process.
Principal Use Scenario	1. The courier picks up the order from the restaurant. 2. The courier updates the status to "Picked Up." 3. The system confirms the status change and notifies the customer. 4. The courier arrives at the delivery address. 5. The courier introduces the PIN the consumer gives him. (*Use Case: Enter PIN to Confirm Delivery*) 6. The system updates the status to "Delivered". 7. The system confirms the delivery and marks the task as completed.

2. Annex B: Additional Behavioral Model Diagrams

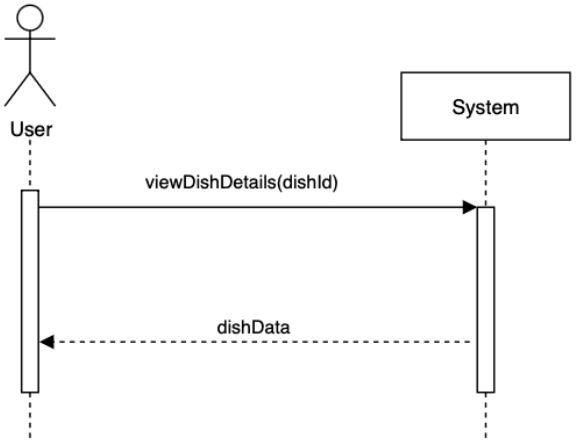
This annex includes additional diagrams related to the behavioral model of the system. These diagrams illustrate dynamic interactions and responses not covered in the main sections, providing a more comprehensive view of the application's functionality.

Search Restaurants



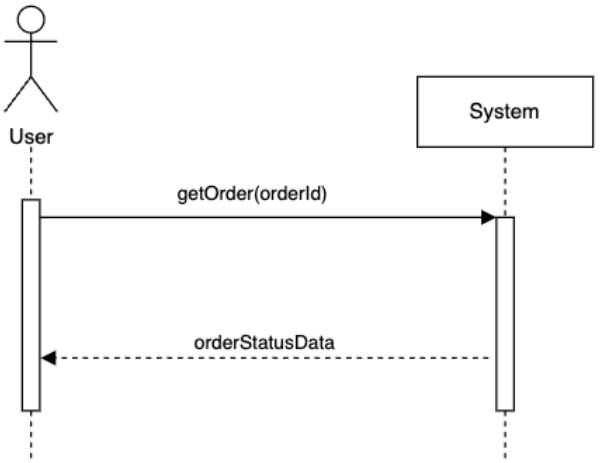
Parameters	customerAddress: String
Precondition	-
Postcondition	-
Body	The restaurants that are available and within the search scope.

View Dish Details



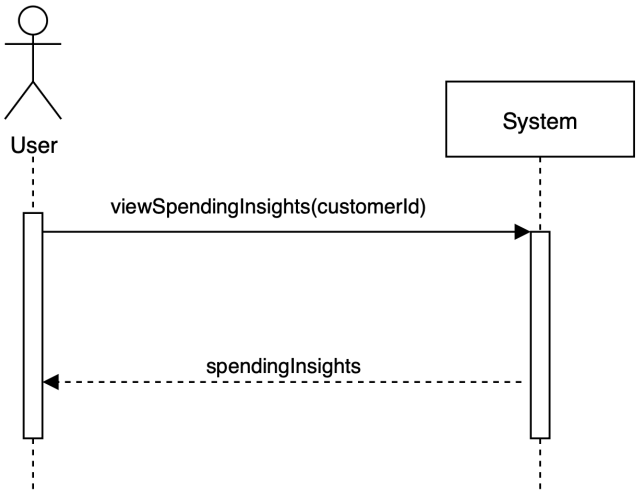
Parameters	dishId: String
Precondition	The *dishId* corresponds to a dish that exists in the system.
Postcondition	-
Body	*dishData* (see *Add Menu Dishes*) contains the details of the dish that is requested to be viewed.

Get Order



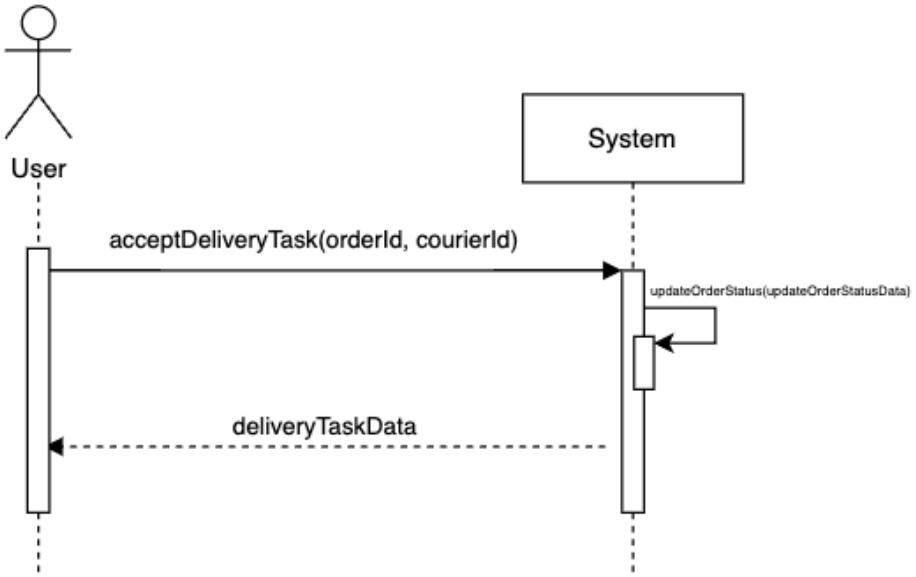
Parameters	orderId: String
Precondition	*orderId* exists in the system
Postcondition	-
Body	*orderStatusData* contains the order status.

View Spending Insights



Parameters	customerId: String
Precondition	*customerId* exists in the system.
Postcondition	-
Body	The spending data insights of the customer are returned.

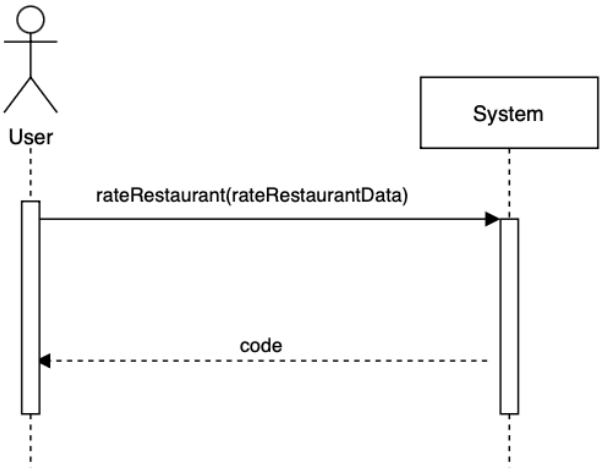
Accept Delivery Task



Parameters	orderId: String, courierId: String

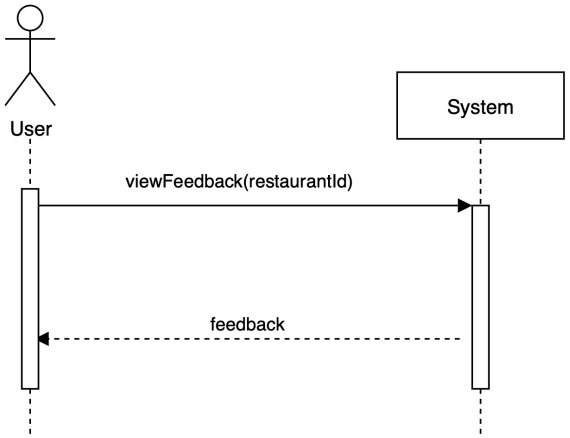
Precondition	*orderId* exists in the system, and the associated order has a status of "*Ready for Delivery*".
Postcondition	The order receives a courierId, the courier is assigned to the order and a PIN code is sent to the customer.
Body	*deliveryTaskData* contains the restaurant address to pick up the order and the order ID.

Rate Restaurant



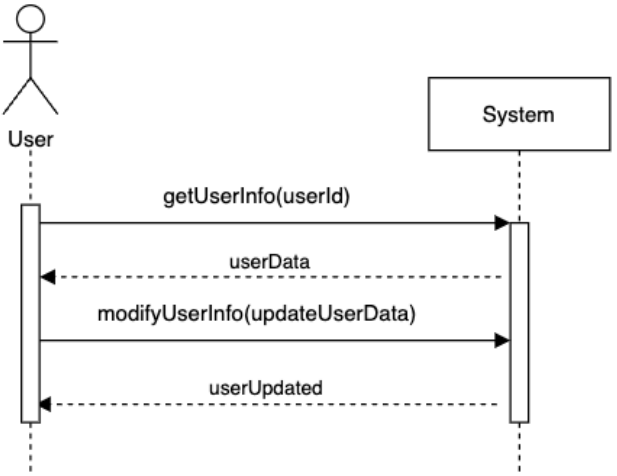
rateRestaurantData	orderId: String, score: Integer [1..5]
Precondition	*orderId* exists in the system and corresponds to a "*Delivered*" order.
Postcondition	The rating is associated with the restaurant. The restaurant's *ratingAvg* is recalculated.
Body	Code indicating whether the operation has succeeded or not.

View feedback



Parameters	restaurantId: String
Precondition	*restaurantId* exists in the system
Postcondition	-
Body	The ratings associated with the restaurant are returned.

Manage Accounts



Method	getUserInfo(userId)
Parameters	userId: String
Precondition	*userId* exists in the system
Postcondition	-
Body	*userData* contains all the user's personal

	data.

Method	modifyUserInfo(updateUserData)
updateUserData	Contains the attributes to be modified.
Precondition	-
Postcondition	The user’s data has been modified.
Body	The user that has been updated