Politecnico di Torino

Politecnico di Torino

Master of Science (M.Sc.) in Computer Engineering

Academic Year 2025/2026

Graduation Session March 2026

Design and Implementation of a Cloud-Native Reactive Architecture

The Smart Travel Platform Case Study

Supervisor

Prof. Giovanni Malnati

Candidate

Federico Palazzi

Abstract

The rapid evolution of enterprise web applications has driven a paradigm shift from monolithic architectures to cloud-native *microservices* solutions. This transition is motivated by the need for greater scalability, resilience, and agility in the software development life cycle. This thesis presents the design, development and deployment of "*Smart Travel*", a comprehensive e-commerce platform dedicated to the sale of travel packages. The project serves as a proof of concept (PoC) developed within an industrial internship context, aiming to analyze the complexities and benefits of a modern and distributed software architecture.

The primary objective of this work is not merely the implementation of functional requirements (such as catalog management and order processing), but rather the critical evaluation of competing technologies and the automation of the release process. A central element of this research is the comparative analysis between two leading Java ecosystems: the established *Spring Boot* framework and the cloud-native *Quarkus* framework. Specifically, the project adopts a fully reactive approach for both of them, utilizing *Spring WebFlux* and *Quarkus Reactive* capabilities.

From an architectural perspective, the Smart Travel system is designed to ensure loose coupling and high availability. The backend adopts a microservices pattern supported by *RabbitMQ*, a message broker that facilitates asynchronous *event-driven* communication among services. Data persistence is managed through *MongoDB*, a NoSQL database chosen for its schema flexibility, which is particularly advantageous for handling diverse and evolving travel package structures compared to rigid relational models. On the client side, the user interface is delivered via a *Single Page Application (SPA)* developed with *Angular* and leveraging *RxJS* for reactive state management, ensuring a responsive and fluid user experience.

A significant portion of this thesis is dedicated to the "industrialization" of the software, bridging the gap between development and operations (*DevOps*). The project moves beyond local execution by implementing a robust *Continuous Integration and Continuous Delivery (CI/CD)* pipeline using *Jenkins*. This pipeline automates the entire delivery chain, including code checkout, compilation, container image creation and pushing to a private registry. The final deployment target is *OKD* (the community distribution of *Kubernetes* that powers *Red Hat OpenShift*). The thesis details the orchestration of containerized applications within the OKD cluster, managing critical aspects such as service discovery, route exposure and configuration management via `ConfigMaps` and `Secrets`.

In conclusion, the Smart Travel project demonstrates the viability and robustness of a heterogeneous, fully reactive microservices architecture. It successfully integrates different modern technologies into a cohesive ecosystem, proving that the adoption

of cloud-native principles, while introducing complexity in orchestration and consistency management, offers superior long-term benefits in terms of maintainability and scalability. The work provides a practical blueprint for modernizing legacy Java applications, offering concrete evidence of how the synergy between Reactive frameworks, Event-Driven patterns, and automated DevOps practices defines the state of the art in contemporary software engineering.

Acknowledgements

Writing this thesis and completing this academic journey has been one of the most challenging yet profoundly rewarding experiences of my life. While this document bears only my name, it is the culmination of the support, guidance, and encouragement of many people to whom I owe my deepest gratitude.

First, I would like to thank my thesis supervisor, Prof. Giovanni Malnati, for their guidance, patience, and expertise throughout this project. I am also deeply grateful to the team at the Lutech company, particularly for providing the infrastructure and collaborative environment that made this research possible.

To my family: thank you for your unconditional love and unwavering support.

To my parents, thank you for always believing in me, for the sacrifices you have made to give me this opportunity, and for being a constant source of encouragement even during the most stressful phases of my studies. I truly could not have reached this milestone without you.

To my university mates: thank you for the shared struggles, the late-night debugging sessions, and the countless coffees that kept us going. Navigating the complexities of exams, projects, and deadlines was made infinitely easier — and much more fun — because we did it together. The friendship we built is something I will always carry with me.

To my friends: thank you for keeping me grounded. Your ability to pull me away from the computer screen, listen to my frustrations, and remind me of the world outside of university has been essential to my sanity and well-being.

Finally, a note of gratitude to myself — for the perseverance, the long nights, and the refusal to give up in those moments when everything seems to fall apart. This achievement belongs to all of us.

Thank you.

Scrivere questa tesi e completare questo percorso accademico è stata una delle esperienze più impegnative ma anche più profondamente gratificanti della mia vita. Sebbene questo documento porti solo il mio nome, è il culmine del supporto, della guida e dell'incoraggiamento di molte persone a cui devo la mia più profonda gratitudine.

Innanzitutto, vorrei ringraziare il mio relatore, Prof. Giovanni Malnati, per la sua guida, la sua pazienza e la sua competenza durante tutto questo progetto. Sono anche profondamente grato al team dell'azienda Lutech, in particolare per aver fornito l'infrastruttura e l'ambiente collaborativo che hanno reso possibile questa ricerca.

Alla mia famiglia: grazie per il vostro amore incondizionato e il vostro incrollabile supporto.

Ai miei genitori, grazie per aver sempre creduto in me, per i sacrifici che avete fatto per darmi questa opportunità e per essere stati una costante fonte di incoraggiamento anche durante le fasi più stressanti dei miei studi. Non avrei davvero potuto raggiungere questo traguardo senza di voi.

Ai miei compagni di università: grazie per le fatiche condivise, le sessioni notturne di debugging e gli innumerevoli caffè che ci hanno dato la forza di andare avanti. Affrontare la complessità di esami, progetti e scadenze è stato infinitamente più facile — e molto più divertente — perché lo abbiamo fatto insieme. L'amicizia che abbiamo costruito è qualcosa che porterò sempre con me.

Ai miei amici: grazie per avermi tenuto con i piedi per terra. La vostra capacità di allontanarmi dallo schermo del computer, di ascoltare le mie frustrazioni e di ricordarmi il mondo al di fuori dell'università è stata essenziale per il mio equilibrio e il mio benessere.

Infine, una nota di gratitudine a me stesso — per la perseveranza, le lunghe notti e il rifiuto di arrendermi in quei momenti in cui tutto sembrava crollare. Questo traguardo appartiene a tutti noi.

Grazie.

Table of Contents

List of Figures

Glossary

ACID

Atomicity, Consistency, Isolation, Durability

AJAX

Asynchronous JavaScript and XML

API

Application Programming Interface

BFF

Backend-For-Frontend

BSON

Binary JSON

CDC

Change Data Capture

CDI

Contexts and Depencency Injection

CI/CD

Continuous Integration / Continuous Delivery

DBMS

Database Management System

DI

Dependency Injection

DOM

Document Object Model

DX

Developer Experience

EDA

Event Driven Architecture

JAX-RS

Java API for RESTful Web Services

JSON

JavaScript Object Notation

JWT

JSON Web Token

MPA

Multi-Page Application

NFR

Non-Functional Requirements

NoSQL

Not Only SQL

ORM

Object-Relational Mapping

PoC

Proof of Concept

POJO

Plain Old Java Object

Pub/Sub

Publish/Subscribe

RBAC

Role Based Access Control

RDBMS

Relational Database Management System

RxJS

Reactive Extensions for JavaScript

SDL

Schema Definition Language

SEO

Search Engine Optimization

SPA

Single Page Application

UI

User Interface

URL

Uniform Resource Locator

Chapter 1

Introduction

1.1 Context

The rapid evolution of enterprise web applications over the last decade has driven a significant paradigm shift in software engineering, moving away from traditional monolithic architectures towards cloud-native microservices solutions.

Historically, monolithic applications were the industry standard and they integrated user interface, business logic, and data access layers into a single deployable unit. They offered simplicity during the early stages of development, straightforward local testing, and centralized debugging. However, as applications grow in size, complexity, and user base, the inherent limitations of the monolithic approach become increasingly apparent [1].

A monolithic design relies on tightly coupled components. Consequently, a minor update, a new feature addition, or a simple bug fix requires the entire massive codebase to be recompiled, tested, and redeployed. This rigid structure often leads to development bottlenecks and significantly increases the risk of system-wide failures during deployments. Furthermore, scaling a monolith is inherently inefficient; if a single specific component (such as the payment processing module) experiences a surge in traffic, the entire monolithic application must be scaled horizontally, leading to unnecessary resource consumption and inflated operational costs.

To overcome these scalability and maintainability challenges, the industry has widely adopted the microservices architecture [1]. This modern paradigm decomposes a large application into a suite of small, autonomous, and loosely coupled services. Each microservice is organized around a specific business capability, encapsulates its own data, and communicates with other services through lightweight, well-defined application programming interfaces (APIs) or asynchronous message brokers. This decentralized approach offers superior modularity, improved fault isolation - the

failure of one service does not inevitably crash the entire system - and the agility to deploy and scale individual services independently based on their specific demand. However, the transition to microservices is not without its own set of complexities. It introduces challenges such as increased operational overhead, the need for sophisticated monitoring and logging solutions, and the complexity of managing distributed transactions and data consistency across services. Despite these challenges, the microservices architecture has become the de facto standard for building scalable, resilient, and maintainable enterprise applications in the cloud era.

1.2 Thesis Goal

The primary objective of this thesis is to explore, apply, and critically evaluate modern cloud-native technologies and microservices architectures. To achieve this, the "*Smart Travel*" platform was designed and developed as a proof of concept (PoC). Rather than simply focusing on the implementation of functional business requirements, this project serves primarily as a practical testbed to analyze the complexities, challenges, and benefits of building a modern, distributed software system. The project was conducted during an internship at a software company, situated directly within the industry's broader context of architectural modernization.

1.2.1 Project description

The Smart Travel application was designed as a comprehensive e-commerce platform dedicated to the sale of travel packages. The system manages the entire customer journey, from the public consultation of a catalog containing flights, accommodations, and organized activities, to the execution of the purchasing process, order history management, and the automated dispatch of email notifications.
A central technical goal of this research is the critical evaluation and comparative analysis of two leading Java ecosystems used to build the platform's microservices: the established *Spring Boot* framework and the newer cloud-native *Quarkus* framework. Specifically, the project adopts a fully reactive programming paradigm for both frameworks, utilizing *Spring WebFlux* and *Quarkus Reactive* capabilities. This enables non-blocking, asynchronous execution, allowing the system to handle a massive number of concurrent requests with a highly optimized `thread pool`.
Furthermore, this thesis aims to explore the "industrialization" of software through the adoption of *DevOps* practices. The objective is to move beyond local execution by implementing a robust *Continuous Integration and Continuous Delivery* (CI/CD) pipeline. By automating the entire delivery chain - from code checkout and compilation to container image creation and pushing to a private registry - the project demonstrates how to bridge the gap between development and operations. Finally, the thesis details the deployment and orchestration of these containerized

microservices within an *OKD* cluster (the community distribution of *Red Hat OpenShift*), managing critical operational aspects such as service discovery, route exposure, and configuration management.

1.2.2 Expected Results

The execution of this thesis work is expected to produce both concrete technical artifacts and valuable analytical insights. The expected results can be categorized into three main areas:

1. From a software engineering perspective, the project aims to deliver a fully functional PoC of the e-commerce platform. This includes the successful implementation of the frontend interface and the backend microservices, validating the practical application of asynchronous communication via RabbitMQ and flexible data management using MongoDB. The resulting system is expected to demonstrate the resilience and scalability inherent in a properly designed cloud-native architecture.

2. From an analytical standpoint, the research provides a qualitative comparative evaluation between Spring Boot and Quarkus within a reactive programming context. Given the scope of the internship, the comparison focuses heavily on the *Developer Experience* (DX) rather than quantitative runtime metrics. The expected outcome is a clear assessment of the differences in syntax, configuration management, boilerplate code reduction, and the overall ease of implementing reactive paradigms in both frameworks. This comparison will offer valuable insights into the practical development lifecycle, ecosystem maturity, and the learning curve associated with adopting Quarkus compared to the traditional Spring environment.

3. From an operational perspective, the expected result is the successful demonstration of a fully automated DevOps lifecycle. By delivering a working CI/CD pipeline via Jenkins that culminates in a live deployment on an OKD cluster, the project will prove the viability and necessity of infrastructure automation. This includes the ability to manage containerized applications at scale, handle configuration and secrets management effectively, and ensure seamless service discovery and routing in a production-like environment.

1.3 Thesis structure

The thesis is structured as follows:

- **Chapter 2: Background and Technologies** - Provides an overview of the core concepts and technologies underpinning the project, including microservices and event-driven architectures, imperative and reactive programming paradigms, backend and frontend frameworks (Spring Boot, Quarkus, Angular), NoSQL databases (MongoDB), and API communication models (REST vs. GraphQL).
- **Chapter 3: System Architecture** - Details the architectural decisions, technology stack, and design patterns used in the Smart Travel project. It covers the microservices decomposition, the event-driven communication via message brokers, and the frontend design.
- **Chapter 4: Application Design and Implementation** - Offers a detailed description of the development process. This includes data modeling, API contract definitions, and explanations of key software components, accompanied by code snippets comparing the Spring Boot and Quarkus implementations.
- **Chapter 5: Infrastructure Automation and Cloud Deployment** - Describes the different steps of the CI/CD pipeline, Jenkins automation configurations, and the final orchestration of containerized applications within the OKD cluster environment.
- **Chapter 6: Conclusion** - Summarizes the project findings, evaluates the results of the technology comparison, discusses the limitations encountered during development, and outlines future work directions for the platform.

Chapter 2

Background and Technologies

This chapter provides a comprehensive overview of the core concepts and technologies underpinning the project. It begins by exploring the architectural shift from traditional monolithic systems to decentralized microservices, then moving to the main differences between imperative and reactive programming models, highlighting how the latter addresses the high-concurrency demands of modern cloud infrastructure.
Furthermore, it examines the specific technological stack selected for the application, including the frontend framework, backend environments, and NoSQL database management. It details the different communication paradigms adopted — spanning from synchronous REST and GraphQL APIs to asynchronous Event-Driven Architectures — to allow the system's isolated components to seamlessly interact with each other.
Finally, it introduces the core principles of DevOps and Continuous Integration/Continuous Deployment pipelines, establishing the necessity of automated infrastructure for managing and deploying modern microservices at scale.

2.1 Microservices Architecture

The evolution of cloud computing has fundamentally altered how enterprise applications are designed, developed, and maintained. While monolithic architectures served as the industry standard for decades, they present significant scaling and maintenance limitations as systems grow in complexity. To address these bottlenecks, the software industry has increasingly adopted the microservices architecture, a paradigm that decomposes large, rigid applications into a suite of smaller, independent services [2].

2.1.1 Principles of Microservices

Unlike a monolithic application that shares a single massive codebase and a centralized database, a microservices architecture is built upon the principle of decentralization and loose coupling. Each microservice is designed around a specific business capability or "bounded context" [3]. For example, in an e-commerce platform like Smart Travel, the catalog management and the order processing capabilities are handled by distinct autonomous services.
The foundational principles of this architecture include:

- **Autonomy and Independent Deployment:** Each service is a self-contained unit. It encapsulates its own code, dependencies, and typically its own data store. This allows teams to update, deploy, and scale a single service without needing to recompile or redeploy the entire system.

- **Lightweight Communication:** Because microservices operate as independent processes, they must communicate over a network to fulfill complex business transactions. This is achieved using well-defined *Application Programming Interfaces* (APIs), typically relying on synchronous protocols like HTTP or asynchronous message delivery mechanisms.

- **Decentralized Data Management:** Rather than relying on a single shared database schema, each microservice manages its own database, choosing the DBMS technology best suited to its specific data access patterns.

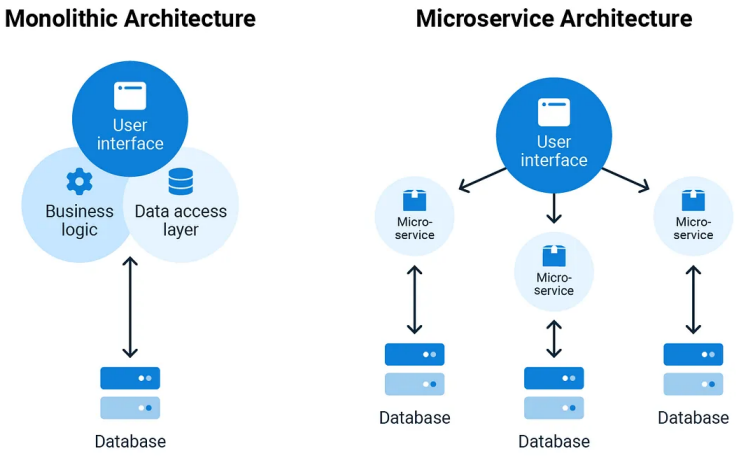

Figure 2.1: Monolithic vs. Microservices architecture. *Source: [4]*

2.1.2 Benefits

The shift from monolithic to microservices architecture provides several advantages [5]:

- **Targeted Scalability:** In a monolithic system, an unexpected spike in traffic for one specific feature (e.g., the payment gateway) requires scaling the entire application, leading to resource wastage. Microservices allow infrastructure teams to scale only the specific services experiencing high demand, optimizing cloud resource consumption and costs.
- **Fault Isolation and Resilience:** Because microservices are loosely coupled, a critical bug, memory leak, or crash in one service does not inherently cascade and bring down the entire platform. The application can handle partial failures gracefully, degrading functionality while keeping core operations online.
- **Technological Freedom:** Teams are not locked into a single technology stack. Since microservices communicate via standardized network protocols, developers can select the most appropriate programming language, framework, and database for each specific task. This flexibility allowed the Smart Travel project to implement and compare different Java frameworks (Spring Boot and Quarkus) within the same ecosystem.
- **Agility and CI/CD:** Smaller codebases are easier to understand, test, and deploy. This structure naturally complements DevOps practices, enabling CI/CD pipelines to push frequent updates to production.

2.1.3 Challenges and Limitations

Despite its significant advantages, adopting a microservices architecture introduces a high degree of distributed system complexity. Some of the key challenges include:

- **Operational Overhead and Monitoring:** Managing dozens of independently running services is inherently more complex than managing a single monolith. It requires sophisticated container orchestration platforms (such as Kubernetes or OKD) and centralized observability tools for distributed tracing, log aggregation, and metric monitoring to identify where a failure occurred across the network.
- **Network Latency and Congestion:** Replacing in-memory function calls with remote network requests introduces latency. If a single user action requires a long chain of inter-service API calls, the accumulated network overhead can severely degrade system performance.

- **Data Consistency:** Because each service maintains its own database, achieving data consistency across the system becomes a major challenge. Traditional `ACID` transactions that span multiple tables are no longer feasible. Instead, developers must rely on complex distributed transaction patterns and embrace *eventual consistency*.

2.2 Imperative vs. Reactive paradigm

The choice of programming paradigm fundamentally dictates how an application manages system resources, handles concurrency, and scales under load. In the context of modern Java microservices, understanding the underlying thread management of both approaches is crucial for evaluating their suitability for high-concurrency, I/O-bound applications.

2.2.1 The Imperative Model

Traditional applications use blocking I/O and an imperative (sequential) execution model. In a web application context - such as a traditional Spring MVC application - this paradigm relies on the *Thread-per-Request* concurrency model [6].
When a new HTTP request arrives, the server allocates a dedicated thread from a pre-configured thread pool to handle the entire lifecycle of that request. If the application needs to interact with an external resource, such as querying a database or calling a third-party API, the thread initiates the I/O operation and then completely halts execution (blocks) until it receives a response.
While this model is straightforward to write, debug, and reason about, it is highly inefficient for I/O-bound microservices. During the waiting period, the blocked thread consumes valuable memory and CPU cycles via context switching, doing absolutely no useful work. Under sudden spikes in traffic, the thread pool can quickly become exhausted. Once all threads are blocked waiting for slow external services, the application becomes unresponsive to new user requests, severely limiting scalability.

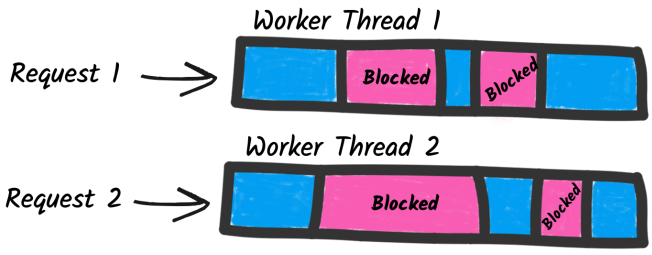

Figure 2.2: Thread-per-Request model in an imperative application. *Source: [7]*

2.2.2 The Reactive Model

The reactive model was designed to build non-blocking applications that are highly responsive, resilient, and resource-efficient. Instead of the Thread-per-Request model, reactive frameworks utilize an *Event Loop* architecture. In this model, a very small number of threads known as `event loop threads` continuously loop to process incoming requests.
When an I/O operation is required, the thread does not block. Instead, it delegates the task to a non-blocking I/O handler, registers a callback, and immediately returns to the event queue to process the next incoming HTTP request. Once the external database or API finishes its processing, it pushes an event back to the event loop, which then resumes the original request and sends the response to the client [7].

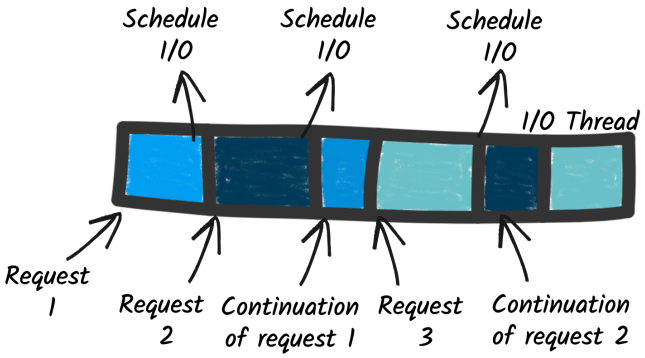

Figure 2.3: Reactive model in a non-blocking application. *Source: [7]*

The key advantages of the reactive model include:

- **Resource Efficiency:** By eliminating blocked threads, the system can handle thousands of concurrent connections using a fraction of the memory and CPU.
- **Data as Streams:** Data is not pulled sequentially but pushed asynchronously. Frameworks like Spring WebFlux and Quarkus Mutiny provide rich APIs to compose and transform these data streams without blocking the main execution flow.

The primary advantage of the reactive concurrency model is its resource efficiency and scalability. By ensuring that threads are never blocked waiting for external I/O operations, the system can sustain thousands of concurrent connections using a fraction of the memory and CPU required by a traditional imperative system. This architectural shift makes the reactive model particularly well-suited for I/O-bound

microservices, allowing them to remain highly responsive and maintain stable performance even under massive, sudden surges in network traffic.

However, this model introduces significant complexities regarding development and operational debugging. The most critical is the inclusion of a blocking operation — such as a legacy JDBC database query, a synchronous HTTP call, or a heavy CPU-bound computation — directly on the event loop. Because the system relies on a small pool of threads, blocking even a single one can immediately bottleneck the application. Furthermore, the non-linear execution flow fragments traditional stack traces. When an exception occurs, the stack trace often only reflects the point of failure within the asynchronous callback, losing the historical context of the original user request and making root-cause analysis considerably more difficult than in the linear imperative model.

2.3 Event-Driven Architectures

As microservices ecosystems expand, the method by which these independent services communicate becomes a critical architectural decision. While traditional web architectures rely heavily on synchronous, point-to-point interactions, the inherent limitations of this approach at scale have led to the widespread adoption of *Event-Driven Architectures* (EDA) [8].

In a tightly coupled microservices environment, relying solely on direct HTTP/REST calls can quickly result in a fragile "distributed monolith." As the number of services grows, point-to-point communication creates a complex, tangled web of dependencies. If a single core service experiences downtime or network latency, the effects can cascade throughout the dependency chain, potentially bringing down the entire platform. Furthermore, this approach introduces strict *temporal coupling*: both the calling service and the receiving service must be online, available, and responsive at the exact same moment for a transaction to succeed.

To mitigate these risks, EDA introduces a fundamental paradigm shift. Instead of services directly commanding one another to perform actions, they broadcast "events" — immutable records indicating that a significant change in state or a specific business action has already occurred (for example, a `PaymentProcessed` or `UserRegistered` event). Other autonomous services within the ecosystem can then asynchronously listen for and react to these events at their own pace. This decoupling transforms the architecture from a rigid, command-driven model into a highly reactive system, vastly improving overall resilience and allowing individual components to scale independently based on event volume.

2.3.1 Synchronous vs. Asynchronous Communication

In a synchronous communication model, typically implemented via HTTP and REST APIs, a service sends a request to another service and blocks its own execution until it receives a direct response. While conceptually simple and easy to trace, this tightly coupled approach creates cascading bottlenecks. If the receiving service is slow, overwhelmed, or offline, the calling service is forced to wait, consuming resources and potentially causing system-wide performance degradation [9].
On the other hand, asynchronous communication decouples the sender from the receiver. In an Event-Driven Architecture, services communicate by producing and consuming *events* - lightweight messages that represent a significant change in state or a specific occurrence within the business domain (e.g., "user registered" or "order placed"). The service producing the event simply broadcasts it and immediately resumes its own operations, entirely unaware of who will consume the event or when it will be processed [8].

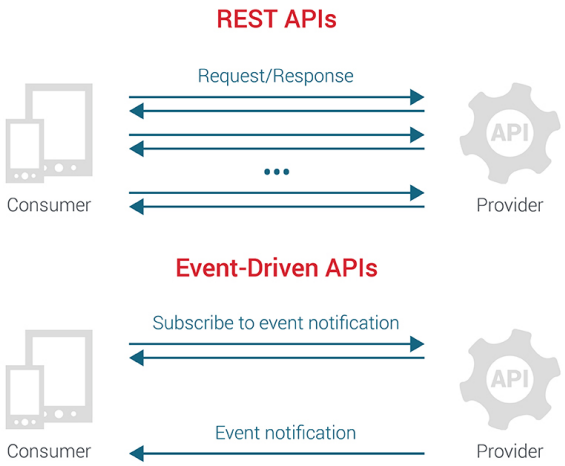

Figure 2.4: Synchronous vs. Asynchronous communication. *Source: [10]*

2.3.2 The Role of Message Brokers

To facilitate this asynchronous exchange without requiring services to know each other's network locations or IP addresses, EDA relies on a centralized intermediary known as a message broker [11].
This architecture generally relies on three primary components:

- **Producers (Publishers):** The microservices that generate events when a specific business action occurs. Their only responsibility is to format the event data and safely push it to the broker.

- **Message Brokers:** The middleware infrastructure that receives, stores, and routes the events. The broker ensures reliable message delivery, managing queues or topics so that events are safely held even if the intended downstream recipients are currently offline or busy.

- **Consumers (Subscribers):** The microservices that continuously listen to the broker for specific types of events. Upon receiving an event, the consumer processes the data and executes its own independent business logic.

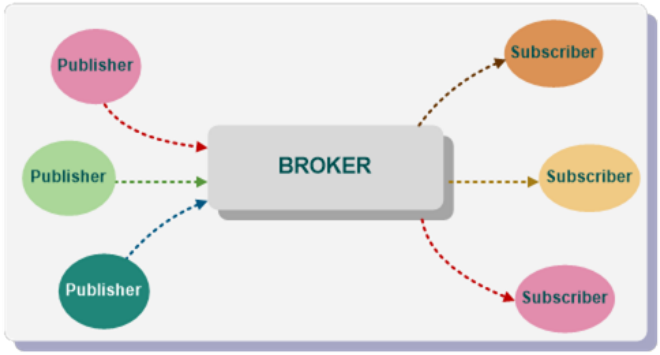

Figure 2.5: Event-Driven Architecture

This publish/subscribe (Pub/Sub) model allows multiple different consumers to react to a single event simultaneously and independently.

2.3.3 Benefits and Challenges

Implementing an Event-Driven Architecture provides different advantages. The decoupling of services enables exceptional scalability, as individual components can be scaled up or down based on their specific event processing loads. Furthermore, it significantly enhances overall system resilience: if a consumer service crashes, the message broker simply retains the pending events in a queue until the service is restored, guaranteeing no data loss during the outage [9].
However, this paradigm introduces unique complexities regarding distributed data. Because communication is fundamentally asynchronous, achieving immediate transactional data consistency across the entire platform is impossible. Instead, developers must design applications around the concept of *eventual consistency*, where the system state reconciles over time. Additionally, tracking the flow of a single user action across multiple asynchronously triggered microservices requires sophisticated distributed tracing mechanisms and centralized logging [11].

2.4 Backend Technologies

Java has been the cornerstone of enterprise backend development for decades, favored for its robustness, mature ecosystem, and platform independence. However, as the industry transitioned from monolithic architectures running on heavy application servers to containerized microservices orchestrated by Kubernetes, traditional Java frameworks faced significant challenges. They often suffered from slow startup times and high memory footprints, characteristics misaligned with the rapid scaling and high-density deployment needs of cloud-native environments [12].
To address these challenges, the Java ecosystem has evolved significantly. During the development of this thesis' project, two distinct frameworks were selected to build the backend microservices: the established Spring Boot and the modern cloud-native Quarkus.

2.4.1 Spring Boot: The Industry Standard

Introduced by Pivotal Software (now VMware Tanzu) in 2014, Spring Boot revolutionized Java enterprise development. It was designed to simplify the bootstrapping and development of new Spring applications by adhering to the *convention over configuration* principle. Before Spring Boot, developers spent hours configuring XML files and managing complex application server deployments. Spring Boot eliminated this friction by providing auto-configuration, opinionated dependency management, and embedded web servers (such as Tomcat or Netty), allowing developers to package an entire microservice into a single, easily deployable `fat JAR` [13].
Today, Spring Boot is the de facto standard for building Java microservices. It can count on an enormous ecosystem, with out-of-the-box integrations for almost every conceivable database, message broker, and cloud provider. While its traditional roots lie in the synchronous, imperative programming model (*Spring MVC*), the framework has actively modernized. With the introduction of *Spring WebFlux*, it now fully supports the reactive, event-loop programming paradigm discussed in Section 2.2, enabling high concurrency without the overhead of thread-per-request blocking.

2.4.2 Quarkus: The Cloud-Native Framework

While Spring Boot adapted to the cloud era, Quarkus was born entirely within it. Created by Red Hat and released in 2019, Quarkus describes itself as a "Kubernetes Native Java framework tailored for GraalVM and HotSpot". Its primary mission is to optimize Java for containerized and serverless environments, where fast startup

times (measured in milliseconds) and minimal memory consumption are critical for reducing cloud infrastructure costs and enabling rapid auto-scaling [12].
Quarkus achieves this extreme efficiency by fundamentally changing the Java build process. Traditional frameworks like Spring heavily rely on reflection, dynamic class loading, and classpath scanning at runtime (processes that consume significant memory and slow down the boot time). Quarkus, instead, utilizes a *compile-time boot* technique. It shifts the vast majority of framework initialization and configuration parsing to the build phase (*Ahead-Of-Time* processing).
Furthermore, Quarkus is built from the ground up on a reactive core powered by Eclipse Vert.x [14]. This unified architecture allows developers to seamlessly blend standard imperative code and non-blocking reactive code within the exact same application, dynamically optimizing the underlying execution thread model without requiring the developer to switch between entirely different API stacks.

2.4.3 Programming Models and Data Access Paradigms

At their core, both Spring Boot and Quarkus serve the exact same fundamental purpose: they are comprehensive Java frameworks designed to build robust, production-ready backend web applications and microservices. They abstract away the complex infrastructure required for enterprise software — such as exposing HTTP endpoints, managing database connection pools, and orchestrating dependency injection—allowing developers to focus strictly on domain-specific business logic.
However, while their end goals are identical, they take fundamentally different approaches to their internal programming models and data access paradigms.
From an API and dependency injection perspective, Spring Boot relies heavily on its own proprietary, highly mature ecosystem. Developers utilize Spring-specific annotations (such as `@RestController`, `@Autowired`, and `@Service`) to wire the application together. Conversely, Quarkus was designed to leverage existing industry standards. It relies on MicroProfile and Jakarta EE specifications, utilizing CDI (Contexts and Dependency Injection) with `@Inject` for bean management and JAX-RS (Java API for RESTful Web Services) with annotations like `@Path` and `@GET` for defining network endpoints.
The most striking architectural difference, however, lies in how the two frameworks handle data access and domain modeling:

- **Spring Data and the Repository Pattern:** Spring Boot strictly encourages the *Repository Pattern*. In this model, the domain entity (the class representing a database document or row) is kept as a pure POJO (Plain Old Java Object). Data access logic is strictly separated into a distinct interface (e.g., `ReactiveMongoRepository`), which Spring automatically implements at

runtime. This provides a clean separation of concerns but requires creating multiple files and interfaces even for simple CRUD operations [15].

- **Quarkus Panache and the Active Record Pattern:** To drastically reduce boilerplate code, Quarkus introduces a layer called *Panache* (available for both Hibernate ORM and MongoDB). While Panache supports the traditional Repository pattern, its standout feature is its implementation of the *Active Record Pattern*. In this paradigm, the domain entity itself extends a base class (such as `PanacheMongoEntity`) and directly inherits static data access methods. This allows developers to query or save data directly through the entity class (e.g., `Flight.findById(id)` or `flight.persist()`). This tight coupling of data and behavior significantly accelerates development for straightforward business domains, though it can become difficult to manage in highly complex relational schemas [16].

2.4.4 The Rationale for Comparison

The decision to utilize both Spring Boot and Quarkus within the Smart Travel platform was driven by the desire to evaluate the state of the art in Java microservices. Implementing identical business logic - such as reactive database queries and asynchronous RabbitMQ event publishing/consuming - using both frameworks provides a valuable lens through which to assess their respective developer experiences, syntactic clarity, and ease of configuration. While Spring Boot offers the safety and familiarity of a massive, mature ecosystem, Quarkus promises a leaner, faster, and easier approach to cloud-native development [17].

2.5 Frontend Technologies

Just as backend architectures have evolved to handle greater scale and complexity, frontend development has undergone a massive transformation. The traditional web paradigm, where the server renders and delivers completely new HTML pages for every user interaction, has largely been superseded by the *Single Page Application* (SPA) model. This shift was driven by the need for more dynamic, responsive, and app-like user experiences that could not be achieved with full-page reloads. To implement this modern frontend architecture, the *Angular* framework was selected for the project, leveraging its powerful component-based structure and native support for reactive programming with the *RxJS* library.

2.5.1 Single Page Application Architecture

Historically, web applications were built as Multi-Page Applications (MPAs). In an MPA, every time a user clicks a link or submits a form, the browser discards the current page, sends a request to the server, waits for the server to process the logic, and then downloads an entirely new HTML document to render. While this approach is straightforward and naturally optimized for *Search Engine Optimization* (SEO), it is highly inefficient for complex interactive applications. The constant full-page reloads lead to latency, wasted bandwidth, and a disjointed user experience [18].
A Single Page Application flips this paradigm. As the name suggests, an SPA loads only a single foundational HTML web page upon the user's initial visit. This initial payload includes all the necessary static assets - such as HTML scaffolding, CSS stylesheets, and the core JavaScript application bundle.
Once the application is loaded into the browser, the page is never reloaded again during the user's session. Instead, when the user interacts with the application (e.g., searching for a flight or filtering hotel results), JavaScript intercepts the action and dynamically rewrites the current DOM (*Document Object Model*). If new data is required, the SPA makes asynchronous requests (via `AJAX` or the `Fetch API`) to backend APIs. The server responds with lightweight JSON data rather than heavy HTML markup. The frontend then processes this data and injects it directly into the existing page structure [18].
This architecture provides several significant advantages:

- **Fluid User Experience:** By eliminating the disruptive "white flash" of page reloads, SPAs provide a continuous, fluid experience that closely mimics a native desktop or mobile application.

- **High Performance and Speed:** After the initial load, subsequent interactions are nearly instantaneous because only raw data is transmitted over the network, drastically reducing bandwidth consumption and server load.

- **Clear Separation of Concerns:** The SPA architecture perfectly complements a microservices backend. The frontend becomes a completely independent application that only cares about consuming data via standardized APIs, allowing frontend and backend teams to develop, deploy, and scale their codebases independently.

Despite its benefits, the SPA model introduces challenges, particularly regarding the initial loading time - since the browser must download a large JavaScript bundle upfront - and the complexity of managing client-side routing and browser history. To address these complexities, robust frontend frameworks are required.

2.5.2 The Angular Framework

To implement the SPA architecture for the Smart Travel platform, Angular was selected. Maintained by Google, Angular is a comprehensive, enterprise-grade framework specifically designed for building scalable single-page applications [19]. Unlike lightweight libraries that focus solely on rendering views (such as React), Angular provides a highly opinionated, all-encompassing toolset. It enforces a strict architectural pattern, requiring developers to write code using TypeScript — a statically typed superset of JavaScript that introduces interfaces, classes, and advanced object-oriented features. This ensures greater code reliability, earlier error detection during compilation, and improved maintainability in large-scale projects.
At its core, the Angular framework is built upon several fundamental concepts that work together to create a cohesive architecture:

- **Modules:** Angular applications are inherently modular. An `NgModule` is a container that groups related components, directives, pipes, and services into cohesive blocks of functionality. Every application has at least one root module (typically `AppModule`) that bootstraps the application. Enterprise applications often utilize numerous "feature modules" that allow for *lazy-loading* — loading specific parts of the application only when the user navigates to them, thereby significantly reducing the initial JavaScript payload.

- **Components and Templates:** The UI is divided into encapsulated, reusable building blocks called components. Each component strictly separates its concerns: a TypeScript class manages the state and behavior, an HTML template defines the view, and a CSS/SCSS file scopes styling exclusively to that component. The connection between the class and the template is managed through powerful *data binding* mechanisms (such as string interpolation, property binding, and event binding), which automatically synchronize the UI with the underlying application state.

- **Directives:** While components define custom DOM elements, directives are classes that add additional behavior to elements in the Angular application. They are primarily divided into *structural directives*, which alter the layout of the DOM by adding, replacing, or removing elements (such as `*ngIf` for conditional rendering and `*ngFor` for list iteration), and *attribute directives*, which alter the appearance or behavior of an existing element (such as dynamically applying CSS classes with `ngClass`).

- **Services and Dependency Injection (DI):** To ensure components remain lean and focused solely on presenting data, business logic and data fetching are delegated to *Services*. Angular features a robust, hierarchical *Dependency*

Injection (DI) system. Instead of a component manually instantiating a class to fetch data, it declares a dependency in its constructor. The DI framework then automatically provides a singleton instance of the requested service—such as an HTTP client used to query the catalog microservice. This pattern promotes loosely coupled, highly modular, and easily testable code [19].

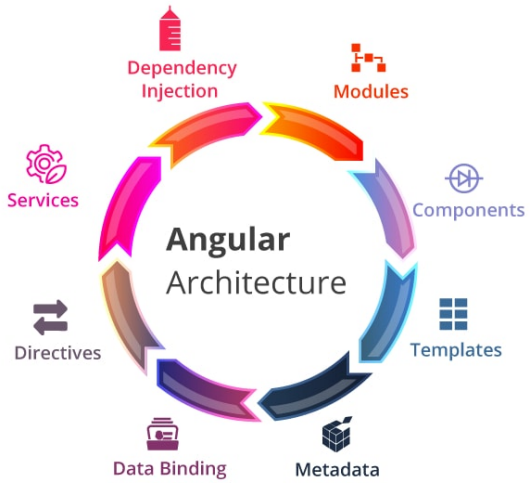

Figure 2.6: Angular Architecture. *Source: [20]*

2.5.3 Frontend Reactive Programming

Because SPAs heavily rely on asynchronous events — such as user inputs, delayed API responses, and routing changes — managing this complex flow of data is a major challenge. To handle this, Angular natively integrates the RxJS library.
RxJS brings the reactive programming paradigm (discussed in Section 2.2) directly into the browser. It introduces the concept of *Observables*, which represent continuous streams of asynchronous data or events over time [21].
Instead of relying on traditional JavaScript `Promises`, which can only handle a single future value and cannot be cancelled, Observables can emit multiple values over time and can be closed or cancelled at any moment. For example, when performing a search query, the Angular HTTP client returns an Observable. Using RxJS operators (like `map`, `filter`, or `debounceTime`), the frontend can gracefully transform the incoming data, ignore rapid repeated clicks from the user, and seamlessly update the UI state without resorting to convoluted, heavily nested callback functions [21]. By leveraging RxJS, the frontend achieves the same non-blocking, event-driven resilience as the reactive backend microservices.

2.6 Database Technologies

As modern applications demand greater flexibility, performance, and the ability to handle massive volumes of unstructured or semi-structured data, traditional *Relational Database Management Systems* (RDBMS) often face significant limitations. The rigid, tabular schema of SQL databases requires careful upfront planning and costly migrations when data requirements change. Furthermore, scaling a relational database is typically restricted to vertical scaling (adding more CPU or RAM to a single server), which eventually hits a hardware ceiling [22]. To address these challenges, *NoSQL* databases emerged as a powerful alternative, offering distributed architectures and highly adaptable data models.

2.6.1 NoSQL Concepts and Classifications

NoSQL databases depart from the fixed table-and-row model, prioritizing horizontal scalability, high availability, and agility over strict ACID compliance. Instead of forcing data into rigid relational structures, NoSQL systems allow data to be stored in formats that more closely align with the objects used in modern application code [23].

NoSQL databases are generally categorized into four primary models:

- **Key-Value Stores:** The simplest NoSQL model, where data is stored as a collection of unique keys and associated values (e.g., Redis). They are highly optimized for fast lookups and caching.

- **Wide-Column Stores:** Data is organized into tables, rows, and dynamic columns, allowing each row to have different columns. They excel at handling massive, predictable query workloads (e.g., Cassandra).

- **Graph Databases:** Designed to store and traverse relationships between entities using nodes and edges (e.g., Neo4j). They are ideal for social networks, fraud detection, or recommendation engines.

- **Document Databases:** Data is stored in flexible, JSON-like documents. Each document contains pairs of fields and values, which can include nested arrays and objects. This is the category to which MongoDB belongs [22].

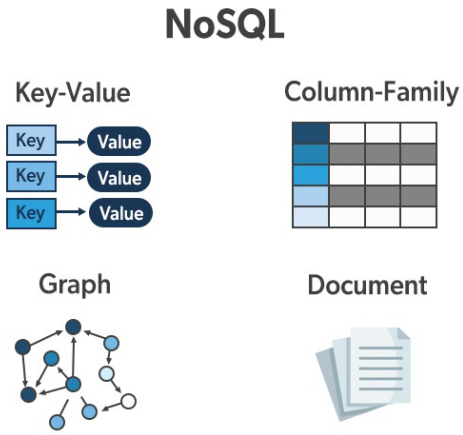

Figure 2.7: NoSQL Database Types

2.6.2 MongoDB Architecture: Collections vs. Tables

The Smart Travel application leverages MongoDB as its primary data store. MongoDB is the leading NoSQL document database, natively designed to handle modern application data with superior flexibility and scalability. Its fundamental departure from traditional relational databases lies in how it structures and stores records.
In a traditional RDBMS, data is structured into *tables* consisting of *columns* (attributes) and *rows* (records). Every row in a table must adhere strictly to the predefined columns. Representing complex, hierarchical data — such as a travel order containing multiple passengers, nested payment details, and a list of booked activities — requires "normalizing" the data. This means splitting the information across multiple distinct tables and reassembling it at query time using computationally expensive `JOIN` operations.
On the other hand, MongoDB organizes data into *Collections* (the conceptual equivalent of tables) and *Documents* (the equivalent of rows).

- **Documents:** A document is a self-contained unit of data. Under the hood, MongoDB stores these documents in a binary format called `BSON` (Binary JSON), which extends standard JSON with additional data types like dates, decimals, and raw binary data. Unlike a flat SQL row, a BSON document can store rich, hierarchical data structures, including nested sub-documents and arrays. This means an entire "Order" can be retrieved in a single, highly efficient database read operation. Furthermore, this document-oriented architecture natively maps to the object-oriented structures used in Java or TypeScript, largely eliminating the need for complex *Object-Relational Mapping* (ORM) layers [23].

- **Collections:** Documents are grouped together into Collections. However, unlike SQL tables, collections enforce a *flexible schema* (or dynamic schema). In a relational database, adding a new feature — such as introducing a "discount_code" or "dietary_requirements" field to only a subset of travel packages — requires an expensive `ALTER TABLE` migration that can temporarily lock the database and force null values upon all existing rows. In MongoDB, documents within the same collection do not need to have identical structures. A new field can simply be appended to a new document without affecting existing records.

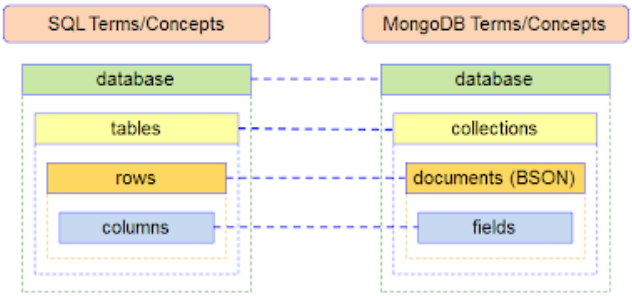

Figure 2.8: SQL vs. MongoDB concepts. *Source: [24]*

This schema flexibility significantly accelerates the development lifecycle and perfectly accommodates the diverse, constantly evolving attributes of different travel products within the Smart Travel catalog.

2.6.3 Data Processing and Advanced Features

MongoDB provides a highly capable and expressive query ecosystem to manipulate and retrieve data efficiently:

- **Aggregation Framework:** To perform complex data processing operations — such as filtering, sorting, grouping, and calculating totals — MongoDB utilizes the Aggregation Pipeline. Modeled on the concept of data processing pipelines in Unix, documents enter a multi-stage pipeline where each stage transforms the data as it passes through. Stages like `$match` (filtering), `$group` (aggregating data), and `$project` (reshaping the output document) allow developers to perform advanced analytics directly within the database engine without needing to transfer massive datasets to the application layer [25].

- **Indexing Capabilities:** Despite its flexible schema, MongoDB relies heavily on indexes to ensure high query performance. Like traditional RDBMS, it uses `B-tree` structures to prevent the database from performing full collection scans. It supports secondary indexes, compound indexes, and highly specialized index types such as geospatial indexes (crucial for location-based travel searches) and text-search indexes.
- **ACID Transactions:** Historically, a major criticism of NoSQL databases was their lack of transactional guarantees across multiple records. However, since version 4.0, MongoDB has introduced full support for multi-document ACID transactions across replica sets and sharded clusters. This allows developers to execute complex, multi-step operations (such as processing a payment and simultaneously updating an order status) with the guarantee that either all changes are applied successfully, or none are, bridging the reliability gap between relational and document databases [26].

2.6.4 Scalability and High Availability

Beyond its data model, MongoDB's architecture is built for distributed environments:

- **Replica Sets:** To ensure high availability, MongoDB uses Replica Sets — a cluster of nodes (typically one Primary and multiple Secondaries) that maintain identical copies of the data. If the Primary node fails, the system automatically elects a new Primary, providing seamless failover and redundancy [23].
- **Sharding:** When the data volume exceeds the capacity of a single machine, MongoDB handles horizontal scaling through Sharding. The data is partitioned across multiple servers based on a designated "Shard Key," allowing the database to manage massive datasets without degrading query performance.

While MongoDB offers immense freedom, it still requires careful data modeling. Developers must strategically choose between *embedding* related data (storing it within the same document for faster read operations) and *referencing* it (linking separate documents via IDs to avoid duplicating large amounts of data). This decision is crucial for optimizing query performance and ensuring data integrity, especially in a complex domain like travel where entities such as flights, hotels, and user profiles have intricate relationships.

2.7 API Paradigms: REST vs. GraphQL

In a decoupled architecture where the frontend is separated from the backend microservices, the API serves as the fundamental communication bridge. The

design of these APIs dictates how efficiently data is transferred over the network. For the "Smart Travel" platform, evaluating the right API paradigm was essential for optimizing the interaction between the Angular frontend and the Java backend. The two dominant paradigms in modern web development are *REST* and *GraphQL*.

2.7.1 REST (Representational State Transfer)

REST has been the undisputed standard for web APIs for over a decade. Introduced by Roy Fielding in 2000, it is an architectural style rather than a strict protocol. RESTful APIs are built around the concept of "resources" (e.g., users, flights, orders), which are identified by standard URLs. Clients interact with these resources using standard HTTP methods such as `GET` (to retrieve data), `POST` (to create data), `PUT` (to update), and `DELETE` to remove data [27].

Core Principles of REST

To be considered truly RESTful, an application's API must adhere to several guiding architectural constraints:

- **Client-Server Separation:** The user interface concerns (the client) are strictly separated from the data storage and business logic concerns (the server). This decoupling allows the frontend (e.g., the Angular SPA) and the backend microservices to evolve and scale independently.
- **Statelessness:** Each client request to the server must contain all the information necessary to understand and process it. The server does not store any client context or session state between requests. This stateless nature makes REST highly scalable, as any server replica within a cluster can handle any incoming request without needing to synchronize session data.
- **Cacheability:** Responses must explicitly define themselves as cacheable or non-cacheable. If a response is cacheable, the client or a network intermediary can reuse the data for equivalent later requests, significantly improving network efficiency and reducing server load.
- **Layered System:** A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary (like a load balancer, an API Gateway, or a proxy). This enables load balancing and shared caches without altering the client's behavior.
- **Uniform Interface:** This is the fundamental feature that distinguishes a REST API. It simplifies the architecture by enforcing a standard, uniform way to interact with the system, relying heavily on standard HTTP specifications.

Resources and Representations

In REST, the primary abstraction of information is a *resource*. A resource can be any concept, entity, or collection within the business domain — such as a user, a flight, or an order. Resources are uniquely identified by standard URLs, often referred to as endpoints [27]. For example, the endpoint `/flights` might represent the collection of all flight resources, while `/flights/123` would represent a specific flight with the ID of `123`.
It is important to note that when a client requests a resource, the server does not send the actual internal database record; instead, it sends a *representation* of the resource's state at that specific moment. In modern web development, this representation is almost exclusively formatted in lightweight JSON objects.
Clients manipulate these resources using standard HTTP verbs, which map directly to traditional database CRUD (Create, Read, Update, Delete) operations:

- `GET` (Read): Retrieves a representation of the resource. It is a safe and idempotent operation, meaning it does not alter the state of the server.
- `POST` (Create): Submits a new payload to the server to create a new subordinate resource.
- `PUT` (Update): Updates an entire existing resource with a new representation, or creates it if it does not exist.
- `DELETE` (Delete): Removes the specified resource from the server.

Limitations of REST in Modern SPAs

While REST's statelessness and uniform interface make it the default choice for general-purpose microservice-to-microservice communication, it has revealed significant limitations regarding data fetching as frontend applications have grown more complex:

- **Over-fetching:** A REST endpoint typically returns a fixed data structure. If the Smart Travel frontend only needs the names and prices of available flights to display a simple list, a `GET /flights` request might still return massive JSON objects containing seating charts, airline details, and baggage rules that the client must download over the network but ultimately discard [28].
- **Under-fetching (the N+1 problem):** If a view requires data from multiple related resources—such as a user's profile and their past order history—the client is often forced to make multiple sequential HTTP requests to different endpoints (e.g., `GET /users/123` followed by `GET /users/123/orders`). This significantly increases network latency and degrades the overall user experience.

2.7.2 GraphQL: The Client-Driven Alternative

To solve the inefficiencies of over-fetching and under-fetching, Facebook developed GraphQL, open-sourcing it in 2015. Unlike REST, which exposes multiple URLs for different resources, a GraphQL API typically exposes only a single endpoint (e.g., `/graphql`) [29].
Instead of the server dictating the structure of the response, GraphQL transfers control to the client. It is a strictly typed query language that allows the frontend to explicitly ask for exactly the data it needs, and nothing more. The client sends a specific query document in the body of an HTTP `POST` request, and the server responds with a JSON object that exactly mirrors the shape of the requested query. For example, a single GraphQL query can simultaneously fetch a user's name, their last three orders, and the specific hotel names associated with those orders. The backend GraphQL engine acts as an orchestrator, resolving the disparate data points from various underlying databases or microservices and assembling them into a single, highly optimized response [28].

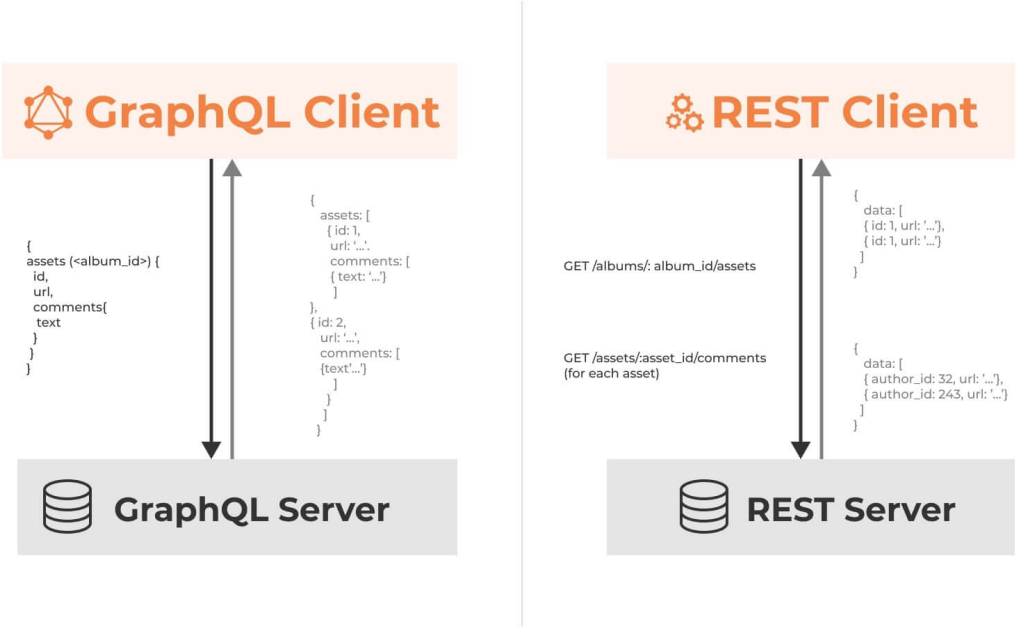

Figure 2.9: REST vs. GraphQL

Schema and Type System

At the core of every GraphQL server is a strictly defined contract known as the *Schema*, written in the *Schema Definition Language* (SDL). The schema defines every piece of data that can be queried or modified, explicitly declaring the types of objects, their fields, and the relationships between them [29].

At the foundational level of this type system are the primitive data types, known in GraphQL as *Scalars*. Scalars represent the leaves of any query — they are the concrete data values that resolve to a single point and cannot be queried any further. GraphQL provides five default scalar types out of the box:

- `Int`: A signed 32-bit integer.
- `Float`: A signed double-precision floating-point value.
- `String`: A UTF-8 character sequence.
- `Boolean`: A standard `true` or `false` value.
- `ID`: A unique identifier, typically used to refetch an object or as a key for a cache. While it is serialized as a String, defining a field as an `ID` signifies that it is not intended to be human-readable.

Beyond these defaults, the GraphQL specification allows developers to define *Custom Scalars*. This is particularly helpful for platforms like Smart Travel, where domain-specific data types such as "Flight", "Accommodation", or "Activity" can be represented as custom scalars with their own validation rules and serialization logic.
Because GraphQL is strongly typed, the server validates every incoming client request against this schema before attempting to execute it. This guarantees that the client only asks for data that actually exists and provides a massive boost to the developer experience, as frontend tooling can automatically generate code and provide real-time auto-completion based on the schema's exact structure.

Core Operations: Queries, Mutations, and Subscriptions

While REST relies on HTTP methods to define the intent of an operation, GraphQL handles all requests via `HTTP POST` and defines intent using three distinct operation types:

- **Query:** The equivalent of a REST `GET` request. Queries are used by the client to fetch data without causing any side effects on the server. A major feature of a Query is that the structure of the returned JSON data perfectly mirrors the nested structure of the requested query. For example, a single Query can simultaneously fetch a user's name, their last three orders, and the specific hotel names associated with those orders, entirely avoiding the N+1 request problem.
- **Mutation:** The equivalent of REST `POST`, `PUT`, and `DELETE` requests. Mutations are used to write data to the server — such as creating a new user,

booking a flight, or cancelling an order. In GraphQL, a Mutation not only performs the intended write operation but can also return the newly modified object in the exact same request, allowing the frontend application to instantly update its local state.

- **Subscription:** While Queries and Mutations follow a standard request-response cycle, Subscriptions establish a persistent, long-lasting connection between the client and the server, typically using WebSockets. Subscriptions are used to push real-time updates from the server to the client whenever a specific event occurs (e.g., pushing a notification when a flight's price suddenly drops).

By relying on a single endpoint, a strongly typed schema, and flexible operations, GraphQL provides a highly efficient and declarative data-fetching paradigm perfectly suited for modern Single Page Applications.

2.7.3 Choosing the Right Paradigm

While GraphQL offers unparalleled efficiency for frontend data fetching, it introduces significant complexity on the backend. Developing a GraphQL server requires defining a rigorous schema, implementing complex resolver functions, and carefully managing database queries to prevent malicious or overly deep queries from exhausting server resources. Furthermore, traditional HTTP caching mechanisms are much harder to implement with GraphQL than with REST [29].
In the context of modern platforms like Smart Travel, the two paradigms are often not mutually exclusive. REST remains highly effective for server-to-server microservice communication and simple CRUD (Create, Read, Update, Delete) operations, while GraphQL is increasingly favored as an API Gateway layer to serve perfectly tailored data payloads to frontend applications.

2.8 DevOps and CI/CD Pipelines

The architectural transition from monolithic applications to distributed microservices fundamentally changes not only how software is written but also how it is delivered. Managing, testing, and deploying dozens of independent services manually is highly prone to human error and operationally unsustainable. To address this bottleneck, the software industry has widely adopted DevOps practices and automated deployment pipelines [30].

2.8.1 The DevOps Philosophy

Traditionally, software development and IT operations functioned in isolated silos. Developers wrote the code and pass it to operations teams, who were then responsible for deploying and maintaining it. This often led to misaligned incentives: developers wanted to push new features rapidly, while operations teams prioritized system stability, naturally resisting frequent changes [31].
DevOps is a cultural and technical movement that merges these two disciplines (Development and Operations). It promotes a collaborative environment where cross-functional teams take full ownership of the entire application lifecycle — from initial planning and writing code to deploying it into production and monitoring its live performance. The core objective of DevOps is to shorten the development lifecycle, increase deployment frequency, and achieve highly dependable releases [30].

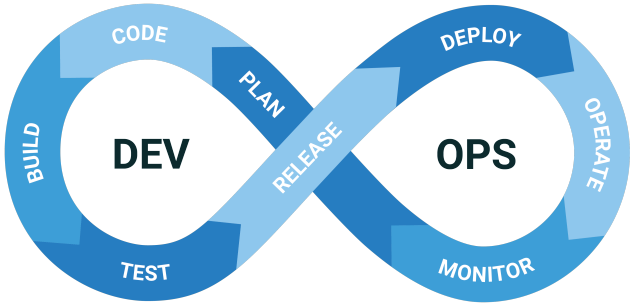

Figure 2.10: DevOps

2.8.2 Continuous Integration and Continuous Deployment

The technical engine that drives the DevOps philosophy is the CI/CD pipeline. A pipeline is a sequence of automated steps that source code must pass through before it reaches the end user. It enforces strict quality gates and removes manual intervention from the software release process [32].

- **Continuous Integration (CI):** In a microservices environment, multiple developers frequently commit code changes to a shared version control repository (such as Git). CI is the practice of automatically triggering a build process every time a new commit is pushed. The CI server automatically compiles the code and runs a suite of unit and integration tests. This rapid feedback loop ensures that new code seamlessly integrates with the existing codebase and that bugs are caught immediately, rather than during a massive release at the end of the month.

- **Continuous Delivery and Deployment (CD):** Once the code passes the CI phase, the CD phase automates the release process. In *Continuous Delivery*, the built artifact (such as a Docker container image) is automatically prepared and pushed to a staging environment, awaiting a manual trigger for final production release. In *Continuous Deployment*, even this manual trigger is removed: every change that passes all automated tests is deployed directly to production without human intervention [33].

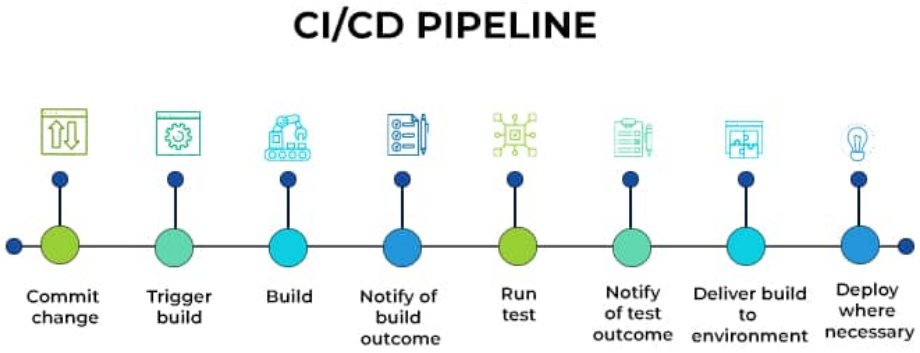

Figure 2.11: CI/CD pipeline

2.8.3 The Necessity of Automation at Scale

Implementing robust CI/CD pipelines is not merely a convenience for cloud-native applications, but a structural necessity. Because a platform like Smart Travel is composed of multiple independent services (e.g., Users, Travel Catalog, Orders, Notification) built with varying technologies (Angular, Spring Boot, Quarkus), each service requires its own distinct lifecycle and pipeline.
Automation guarantees repeatability. By defining the infrastructure and deployment steps as code, a pipeline ensures that a microservice will be compiled, tested, packaged into a container, and deployed in the exact same manner every single time. This eliminates configuration drift, drastically reduces the downtime associated with manual deployments, and provides the agility required to roll back seamlessly if a critical defect is detected in production [32].

Chapter 3

System Architecture

This chapter begins by establishing the business context and user roles of the Smart Travel domain, followed by an analysis of its functional and non-functional requirements. It then explores the high-level architecture, detailing the decomposition of the monolithic domain into bounded, autonomous microservices (Users, Travel Catalog, Orders, and Notifications).
It proceeds to examine the data modeling strategy, highlighting how MongoDB's flexible schema accommodates the diverse structures of travel products.
Finally, the chapter outlines the communication patterns employed to orchestrate these services, contrasting synchronous API gateways with the asynchronous event-driven mechanisms powered by RabbitMQ and Debezium.

3.1 Application Domain Overview

Before diving into the technical requirements and the architectural topology of the system, it is essential to establish the business context and the primary objectives of the application.
Smart Travel was conceived as a full-stack e-commerce application dedicated to the tourism sector. Its primary business goal is to provide a centralized hub where users can search, compose, and purchase complete travel experiences — spanning flights, accommodations, local activities, and travel packages. From a software engineering perspective, the platform serves as a PoC to validate the implementation of cloud-native microservices, reactive programming paradigms, and event-driven communication.
To achieve these goals, the application was designed around a specific set of core business requirements and user-facing functionalities:

- **Public Catalog Exploration:** The platform provides public access to the travel catalog, allowing users to explore available offerings using advanced

search and filtering mechanisms to find specific flights, hotels, or activities.

- **Flexible Purchasing and Cart Management:** The system supports a highly flexible shopping experience. Users can purchase individual travel services standalone or aggregate them into complete and customized packages, with the ability to review and modify their cart selections prior to checkout.

- **End-to-End Order Management:** The platform handles the complete purchasing lifecycle, providing authenticated customers with a dedicated personal area to track and monitor their past and upcoming orders.

- **External Service Integration:** To facilitate real-world e-commerce capabilities, the application securely integrates with external third-party services, such as online payment gateways (e.g., PayPal) to process financial transactions.

- **Automated Notifications:** To ensure a responsive user experience, the system implements an automated notification pipeline, dispatching immediate email confirmations upon the successful completion of an order.

3.1.1 User Roles and Objectives

To satisfy diverse business needs, the platform was designed to support multiple distinct user profiles, each with specific permissions, workflows, and end goals. The system identifies four primary actors:

- **Guest (Unregistered User):** Guests can freely navigate the platform to explore the catalog of travel packages and search for flights, accommodation and activities on specific dates. However, they are restricted from executing financial transactions or composing custom packages until they authenticate.

- **Customer (Registered User):** Once authenticated, a Customer aims to plan and purchase their travel itinerary. Beyond the capabilities of a Guest, Customers can compose personalized travel packages by aggregating distinct catalog components, proceed to secure checkout and manage a personal dashboard containing their order history.

- **Travel Agent:** A business-oriented user responsible for enriching the platform's catalog. The Travel Agent's primary goal is to curate and compose new travel packages. They have access to dedicated interfaces where they can bundle flights, hotels, and activities into fixed "Agency Packages," manage their publication status (draft, published, or archived), and view historical data on the packages they have created.

- **Administrator:** The Administrator possess the highest level of authorization, granting them access to the user management system and the ability to oversee all travel packages (including those created by Travel Agents). The Administrator's role is to maintain the integrity of the platform by managing users' accounts and permissions.

3.2 Requirements Analysis

Before defining the technical architecture, a requirements analysis was conducted to establish the scope and expected behavior of the application. The requirements were divided into functional, detailing what the system must do, and non-functional, dictating how the system should perform under various conditions.

3.2.1 Functional Requirements

The functional requirements were categorized by business domain and mapped to specific user roles (Guest, Customer, Travel Agent, and Administrator). The following table outlines the comprehensive list of functional features expected from the platform.

Table 3.1: Smart Travel Functional Requirements

ID	Description
FR1	**Authentication and Authorization**
FR1.1	User login and logout
FR1.2	Account creation (Sign up)
FR1.3	Account management (Edit details, Delete account)
FR2	**Flights**
FR2.1	Search flights by destination and date range
FR2.2	Filter search results
FR2.3	View flight details (e.g., layovers, airlines)
FR2.4	Flight payment processing (Summary, Total calculation)
FR3	**Accommodations**
FR3.1	Search accommodations by destination and date range
FR3.2	Filter search results
FR3.3	View accommodation details (e.g., available rooms, amenities)
FR3.4	Accommodation payment processing (Summary, Total calculation)
FR4	**Activities**

Continued on next page

Table 3.1 – continued from previous page

ID	Description
FR4.1	Search activities by destination and date range
FR4.2	Filter search results
FR4.3	View activity details (e.g., location, schedule)
FR4.4	Activity payment processing (Summary, Total calculation)
FR5	**Travel Packages (Agency Created)**
FR5.1	Search existing agency packages by destination and date range
FR5.2	Filter search results
FR5.3	View package details (Flights, Accommodation, Activities)
FR5.4	Package payment processing
FR5.5	View recommended travel packages on the homepage
FR6	**Custom Travel Packages (Customer Created)**
FR6.1	Search individual components to build a custom package
FR6.2	Select components and compose the custom package
FR6.3	Custom package payment processing
FR7	**Customer Personal Area**
FR7.1	View order history and specific order details
FR7.2	View and manage account details
FR8	**Notifications**
FR8.1	Send an automated summary email to the customer for every purchase
FR9	**Package Management (Travel Agent)**
FR9.1	Add a new travel package (Draft / Unpublished)
FR9.2	Publish a previously created draft package
FR9.3	Archive a published travel package
FR9.4	Delete an unpublished travel package
FR9.5	View history of created travel packages
FR10	**User Management (Administrator)**
FR10.1	View all user accounts
FR10.2	Edit user accounts details
FR10.3	Delete user accounts

3.2.2 Non-Functional Requirements

While the functional requirements dictate the platform's features, the non-functional requirements (NFR) establish the systemic qualities necessary for the application to succeed in a cloud-native production environment. Based on the project goals, the following constraints were prioritized:

- **Scalability:** The system must gracefully handle varying loads. By adopting a microservices architecture, individual domains (such as the Travel Catalog service during heavy search traffic) must be able to scale horizontally and independently without requiring the entire platform to be duplicated.

- **Performance and Responsiveness:** To provide a fluid user experience, the frontend SPA must minimize full-page reloads. On the backend, services should leverage reactive programming (via Spring WebFlux and Quarkus Reactive) to ensure high throughput and minimize blocking operations, keeping API response times consistently low even under concurrent load.

- **Resilience and Fault Tolerance:** The failure of a single component must not trigger a system-wide outage. The architecture must employ asynchronous communication (Event-Driven messaging via RabbitMQ) to decouple critical write operations — such as processing an order and sending a confirmation email — ensuring that temporary downstream unavailability does not result in data loss.

- **Maintainability and Automation:** The codebase must be highly modular, allowing separate development teams to work on the frontend, catalog, and order services simultaneously. Furthermore, the deployment lifecycle must be entirely automated via CI/CD pipelines to ensure consistent, repeatable, and testable releases into the target OKD cluster.

3.3 High-Level Architecture

To satisfy the scalability, maintainability, and functional requirements described in the previous section, the application was designed utilizing a distributed microservices architecture. The system is composed of several independent backend services, a centralized API gateway, an event broker, and a dedicated frontend application.

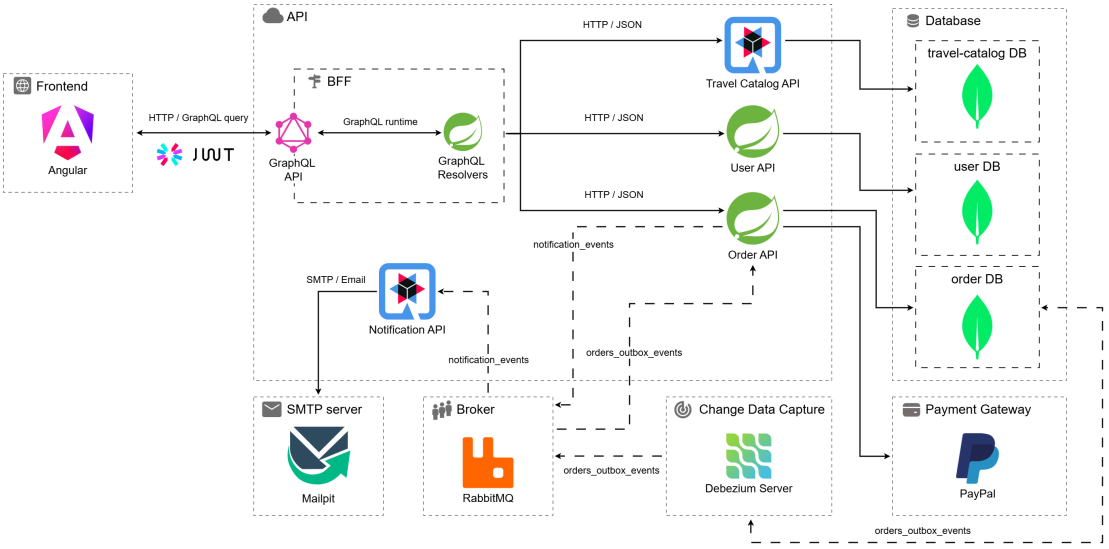

Figure 3.1: Smart Travel High-Level Architecture

3.3.1 The Backend-For-Frontend Pattern

Instead of allowing the Angular frontend to communicate directly with the underlying microservices, the architecture implements the *Backend-For-Frontend* (BFF) pattern.

In a distributed microservices ecosystem, a frontend client often needs to fetch data from multiple independent services to render a single view. Allowing the client to communicate directly with these backend services introduces several critical issues: it creates lots of network traffic with high latency, tightly couples the frontend to the backend schema, and exposes internal network architectures to the public internet [34].

While a traditional API Gateway solves the exposure problem by acting as a single entry point, a single, general-purpose gateway serving all types of clients (e.g., web browsers, mobile apps, third-party consumers) quickly becomes a bloated, monolithic bottleneck [35].

Introduced to solve this exact problem, the BFF pattern advocates for creating a dedicated backend service tailored specifically for the needs of a single frontend interface. In this way, the BFF can implement client-specific logic, such as aggregating data from multiple microservices into a single response, transforming data formats, and enforcing security policies without affecting other clients or services.

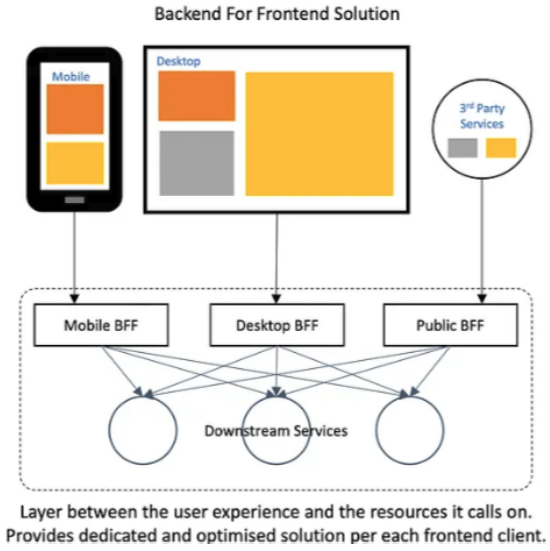

Figure 3.2: Backend-For-Frontend pattern. *Source: [36]*

Implementation in Smart Travel

For the Smart Travel platform, the BFF is tailored to the requirements of the Angular SPA. It fulfills two primary architectural responsibilities:

- **Data Aggregation and GraphQL Translation:** When the frontend sends a GraphQL query, the BFF acts as an orchestrator. Its internal resolvers parse the query, execute the necessary downstream HTTP/JSON requests to the Travel Catalog, User, and Order microservices, aggregate the responses in memory and return a single JSON payload to the client. This translation layer completely eliminates the N+1 request problem and prevents both over-fetching and under-fetching on the client side.

- **Centralized Security Boundary:** The BFF acts as the strict security perimeter for the platform. It handles all authentication and authorization checks using **Spring Security** and **JSON Web Tokens (JWT)**. Because the BFF validates the JWTs and enforces *Role-Based Access Control* (RBAC) before routing any request downstream, the individual microservices do not need to implement complex or redundant security checks. They operate within a trusted internal network, safely assuming that any request forwarded by the BFF has already been authenticated.

3.3.2 The Microservices Ecosystem

The core business logic is divided into four autonomous microservices. Adhering to the "Database-per-Service" pattern, each data-persisting microservice manages its own isolated MongoDB instance hosted on MongoDB Atlas, ensuring true loose coupling. To evaluate the state-of-the-art Java frameworks, these services were developed using a mix of Spring Boot and Quarkus:

- **Travel Catalog API (Quarkus):** Manages the core product domain. It handles searches, filtering, and retrieval of flights, accommodations, activities, and travel packages. It connects to the dedicated `travel-catalog DB`.
- **User API (Spring Boot):** Manages user profiles, account details and system roles. It connects to the `user DB`.
- **Order API (Spring Boot):** Orchestrates the checkout process, order history, and payment validation. It connects to the `order DB`.
- **Notification API (Quarkus):** A purely asynchronous, event-driven service responsible for dispatching communications to the user which does not require a database. Instead, it listens to the message broker and interacts with Mailpit, a fake SMTP server, to send emails.

3.3.3 Event-Driven Payment and Notification Flow

One of the most complex architectural flows in the Smart Travel platform is the payment process, which leverages the *Outbox Pattern* alongside *Change Data Capture* (CDC) to guarantee data consistency across distributed services.
While the granular implementation of this flow will be detailed in subsequent chapters, the high-level orchestration follows these strict steps:

1. **Checkout Initiation:** The frontend requests an order creation via the BFF. The Order API creates a `PENDING` order in MongoDB, retrieves a payment URL from PayPal, and the user is redirected to authorize the transaction.
2. **Payment Authorization:** After the user authorizes the payment, they are redirected back to the application. The Order API updates the order status to `PAID` and, crucially, writes an event to a database outbox collection within the exact same transaction.
3. **Change Data Capture (CDC):** A **Debezium Server** continuously monitors the MongoDB change streams. Upon detecting the new outbox insert, Debezium captures the change and publishes it to the `orders_outbox_events` queue in **RabbitMQ**.

4. **Asynchronous Capture:** The Order API, acting as a consumer, reads the outbox event from RabbitMQ. It officially captures the funds via PayPal, updates the database status to `COMPLETED`, and publishes a new event to the `notification_events` queue.

5. **Notification Dispatch:** The Notification API consumes the event from the `notification_events` queue and triggers an SMTP server (Mailpit) to send a finalized order confirmation email to the customer.

This sophisticated flow ensures that no data is lost even if a service temporarily fails. If the Notification API is down, RabbitMQ simply holds the event in the queue until the service recovers, perfectly demonstrating the resilience of Event-Driven Architecture discussed in Section 2.3.

3.4 Data Modeling

For this project, MongoDB was selected as the underlying persistence layer across all services.
By utilizing MongoDB's document model, the application heavily leverages data denormalization and embedding. Instead of relying on foreign keys and computationally expensive joins across separate domains, objects frequently embed "summaries" or "snapshots" of related data. This guarantees that each microservice remains fully autonomous and resilient to network partitions.

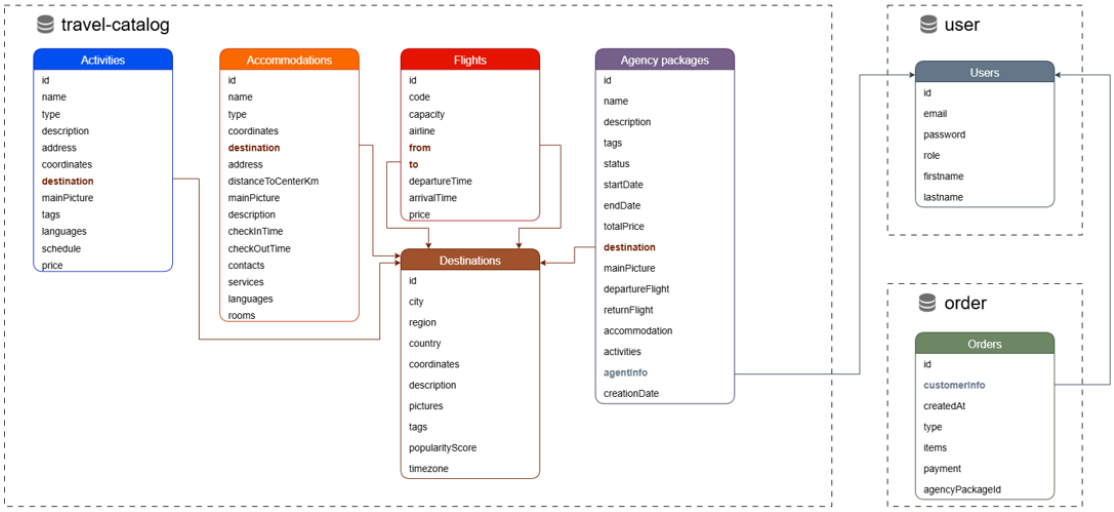

Figure 3.3: Smart Travel Database Schema

3.4.1 Travel Catalog Database

The `travel-catalog` database represents the most complex domain of the platform, responsible for storing the diverse inventory of travel products. Managed by the Quarkus Travel Catalog microservice, it is divided into five primary collections:

- **Destinations:** The foundational geographical entities where each document stores precise location data (city, region, country, coordinates) alongside descriptive metadata (pictures, tags and a popularity score).

- **Flights:** The `flights` collection contains documents representing individual flight segments. They define the airline, capacity, departure/arrival timestamps, pricing, and strictly embed `FlightDestination` objects for the origin and destination to avoid referencing the external Destinations collection during rapid searches.

- **Accommodations:** The `accommodations` collection stores highly detailed hierarchical documents. Because of MongoDB's flexible schema, an accommodation document embeds not only basic scalar fields (address, distance to center, check-in times) but also complex nested arrays representing the available `rooms`, provided `services` and `policies`. It also embeds a `DestinationSummary` to map its location.

- **Activities:** Similar to accommodations, the `activities` collection stores scheduled local experiences (e.g., guided tours, museums), embedding pricing, schedules, and location summaries.

- **Agency Packages:** The `agency_packages` collection represents curated, pre-assembled bundles created by travel agents. A package document embeds the `FlightOrder`, `AccommodationOrder`, and a set of `ActivityOrder` objects that comprise the package. It also embeds a `UserSummary` (`agentInfo`) representing the agent who created it.

3.4.2 User Database

The `user` database is deliberately kept lightweight and strictly focused on identity and access management. Managed by the Spring Boot User API microservice, it consists of a single `users` collection.
The `User` document schema stores the primary authentication credentials (`email`, `password`), personal details (`firstname`, `lastname`), and the user `role` (`CUSTOMER`, `TRAVEL_AGENT`, `ADMIN`). This field is heavily utilized by the BFF to generate JWTs and enforce (RBAC) across the entire platform.

3.4.3 Order Database

The `order` database encapsulates the transactional history and payment states of the platform. Managed by the Spring Boot Order API, its design perfectly illustrates the concept of *Eventual Consistency and Data Snapshotting* in microservices.
When an order is created and stored in the `orders` collection, it categorizes the transaction via an `OrderType` (`SINGLE`, `AGENCY`, or `CUSTOM`) and embeds the exact state of the payment (`PaymentStatus`, timestamps, and hidden provider tokens).
The `Order` document does not simply store a list of MongoDB `ObjectId` references pointing back to the flights or accommodations in the Catalog database. Doing so would tightly couple the two microservices and risk data corruption: if a hotel later changed its name or raised its price in the Catalog, historic orders would incorrectly reflect the new data.
Instead, the `Order` document embeds a massive `OrderItems` object containing concrete representations of the purchased goods (e.g., `FlightOrder`, `ActivityOrder`). These embedded objects take a hard, immutable *snapshot* of the product at the exact moment of purchase. For example, a `FlightOrder` duplicates the airline name, flight code, departure times, and the precise `Price` paid. This denormalization guarantees that historical order receipts remain completely independent from future catalog modifications, ensuring absolute transactional integrity.

3.5 Communication Patterns

In a distributed microservices architecture, the network replaces the in-memory method calls of a monolithic application. Consequently, the patterns chosen for inter-service communication fundamentally dictate the platform's latency, reliability, and scalability. To balance the need for immediate data retrieval with the necessity for fault-tolerant background processing, the Smart Travel platform implements a hybrid communication strategy, utilizing both synchronous and asynchronous paradigms.

3.5.1 Synchronous Communication

Synchronous communication is utilized for operations where the client (or calling service) requires an immediate response to proceed. In this model, the thread remains blocked (or asynchronously suspended in a reactive context) until the HTTP response is received. The platform employs two distinct synchronous patterns depending on the network boundary:

- **Frontend to BFF (GraphQL):** The external boundary between the Angular SPA and the BFF is governed by GraphQL over HTTP. As discussed in Section

2.7, GraphQL was explicitly chosen for its strictly typed query language and to eliminate over-fetching and ensures the UI receives exactly the data it needs to render. By exposing a single endpoint, the BFF allows the frontend to request complex, deeply nested datasets — such as an agency package, its included flight details, and the associated user reviews — in a single network round-trip.

- **BFF to Microservices (REST):** The internal boundary between the BFF and the downstream microservices (Travel Catalog, User, Order) relies on standard RESTful APIs. Because these services reside within the same secure, high-speed local network or OKD cluster, the overhead of multiple HTTP `GET` or `POST` requests is negligible. REST provides a universally understood, stateless, and highly cacheable standard that seamlessly bridges the Spring Boot and Quarkus ecosystems. The BFF acts as the REST client, orchestrating concurrent HTTP calls to the various domain APIs and stitching their JSON responses together before returning the GraphQL payload to the frontend.

3.5.2 Asynchronous Communication

While synchronous APIs are excellent for reading data, they introduce dangerous temporal coupling when writing data or executing multi-step business transactions. If the Order API synchronously called the Notification API to send an email, a failure in the mail server would cause the entire checkout process to crash. To prevent this, Smart Travel delegates critical cross-service workflows to an asynchronous, Event-Driven Architecture (EDA).

- **Message Brokering with RabbitMQ:** RabbitMQ is deployed as the central message broker. Instead of direct service-to-service commands, microservices publish "events" (messages representing state changes) to specific queues (e.g., `notification_events`). Consumer services listen to these queues and process the messages at their own pace. This decouples the producer from the consumer, ensuring that if the Notification API is temporarily offline, RabbitMQ will safely persist the messages until the service recovers.
- **Change Data Capture with Debezium:** To safely publish events to RabbitMQ without encountering the "dual-write problem" (the risk of successfully saving an order to MongoDB but failing to publish the event to the broker due to a network glitch), the platform implements the *Transactional Outbox Pattern* powered by Debezium. Debezium is a CDC platform that connects directly to MongoDB's replica set oplog (operations log). When the Order API writes an event to the database, Debezium detects the stream change at the database level and reliably forwards it to RabbitMQ. This guarantees at-least-once delivery of all critical business events.

- **SMTP Emulation with Mailpit:** As the final step in the asynchronous notification pipeline, the Quarkus Notification API consumes events from RabbitMQ and formats customer emails. To safely test this flow during development without spamming real email addresses or requiring paid external SMTP providers, the architecture incorporates **Mailpit**. Mailpit acts as a localized, fake SMTP server that catches all outbound emails and provides a web interface to inspect them, validating the end-to-end asynchronous workflow.

Chapter 4

Application Design and Implementation

With the architectural foundation and data models established previously, this chapter focuses on the concrete technical implementation of the Smart Travel platform. It details the transition from conceptual design to executable code, highlighting how the chosen frameworks, libraries, and paradigms were applied to solve specific challenges.

The chapter is divided into five primary areas. First, it dives into the backend microservices, contrasting the reactive data-access implementations of Spring Boot and Quarkus. Second, it explores the implementation of the (BFF) layer, detailing how a centralized GraphQL API was constructed to aggregate downstream data. Third, it examines the frontend development, focusing on how Angular and RxJS manage the complex asynchronous state of the SPA.

Fourth, the chapter details the end-to-end authentication and authorization flow, bridging the Angular frontend's token management with the Spring Security perimeter at the BFF. Finally, it provides a deep dive into the implementation of the platform's core business workflows, demonstrating how these isolated layers collaborate to execute the catalog search mechanisms, the secure payment integration utilizing the Transactional Outbox pattern, and the asynchronous event-driven notification pipeline.

4.1 Backend Microservices Development

The backend of the Smart Travel platform was deliberately constructed using a heterogeneous technology stack. By developing the Orders microservice in Spring Boot and the Travel Catalog microservice in Quarkus, the project serves as a practical comparative study of the two leading Java enterprise frameworks.

Crucially, both stacks were implemented using a strictly non-blocking, reactive programming paradigm to maximize concurrency. However, their internal approaches to dependency injection, reactive streams, and database access differ significantly.

4.1.1 Implementing Reactive Services with Spring Boot

Let's take as example the Order API which was developed utilizing Spring Boot and Spring WebFlux. It strictly adheres to the traditional three-tier architecture (Controller, Service, Repository), promoting a high degree of separation of concerns. At the presentation layer, endpoints are defined using proprietary Spring annotations. The `OrderController` handles incoming HTTP requests and returns asynchronous data streams using Project Reactor's `Mono` (for 0-1 elements) or `Flux` (for 0-N elements) [37].

```
1  @RestController
2  @RequestMapping("/api/orders")
3  public class OrderController {
4    private final OrderService orderService;
5
6    @GetMapping("{id}")
7    public Mono<ResponseEntity<OrderResDTO>> getOrderById(
       @PathVariable String id) {
8      return orderService.getOrderById(id).map(ResponseEntity::ok);
9    }
10 }
```

Listing 4.1: Spring Boot WebFlux Controller Snippet

At the data access layer, Spring enforces the **Repository Pattern**. The domain entity (`Order`) is kept as a pure data object, while a separate interface extends `ReactiveMongoRepository`. For standard operations, Spring automatically generates the implementation at runtime. However, a major strength of Spring Data is its ability to map custom BSON queries directly to interface methods using the `@Query` annotation.
As shown in the `OrderRepository`, custom queries can be effortlessly defined to check for specific nested fields (like `customerInfo.userId`) and arrays of statuses without requiring manual database driver configurations:

```java
@Repository
public interface OrderRepository extends ReactiveMongoRepository<
    Order, String> {

  @Query(value = "{ 'customerInfo.userId': ?0, 'payment.status': {
    $in: ['PENDING', 'PAID']} }", exists = true)
  Mono<Boolean> hasPendingPayment(ObjectId userId);

  @Query("{ 'customerInfo.userId': ?0, 'payment.status': { $in: ['
   PENDING', 'PAID'] } }")
  Mono<Long> countPendingOrPaidOrders(ObjectId userId);

  Mono<Order> findByPayment_Token(String paymentToken);
}
```

Listing 4.2: Spring Boot Reactive Mongo Repository

For highly complex, dynamic queries — such as the advanced filtering required by the `getOrders` endpoint — the service layer bypasses the repository interface entirely and leverages the `ReactiveMongoTemplate`. This allows the programmatic construction of `Criteria` objects. A fundamental characteristic of this reactive paradigm is deferred execution: defining operations via `mongoTemplate.find()` or `mongoTemplate.count()` does not immediately trigger a database query. Instead, these methods merely assemble a data-processing pipeline. The actual execution against the database is strictly delayed until a consumer — in this architecture, the underlying Spring WebFlux framework handling the HTTP response — explicitly subscribes to the returned publisher. To maximize efficiency, Project Reactor's `Mono.zip()` is utilized to subscribe to both the content fetch and the document count `Monos` simultaneously, executing them concurrently without blocking the underlying execution thread [38].

```java
@Service
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {
  private final OrderMapper orderMapper;
  private final ReactiveMongoTemplate mongoTemplate;

  @Override
  public Mono<PagedResDTO<OrderResDTO>> getOrders(int page, int
   size, String sort, String order, OrderFilter filters) {
    Sort.Direction direction = order.equalsIgnoreCase("desc") ?
        Sort.Direction.DESC :
        Sort.Direction.ASC;
    Sort sortBy = Sort.by(direction, sort);
    long skip = (long) page * size;

    // Filter query
```

```
16      Query query = new Query().with(sortBy).skip(skip).limit(size);
17      Criteria criteria = new Criteria();
18      List<Criteria> criteriaList = new ArrayList<>();
19
20      // Compose criteria based on provided filters
21      if (filters.getOrderId() != null && !filters.getOrderId().
        isBlank()) {
22        try {
23          criteriaList.add(Criteria.where("id").is(
24              new ObjectId(filters.getOrderId())
25          ));
26        } catch (IllegalArgumentException e) {
27          // Invalid ObjectId for orderId, ignore this filter
28        }
29      }
30      // Other filters...
31
32      Mono<List<Order>> contentMono = mongoTemplate.find(query,
        Order.class).collectList();
33
34      // Total items count query
35      Query countQuery = new Query();
36      if (!criteriaList.isEmpty()) countQuery.addCriteria(criteria);
37      Mono<Long> totalCountMono = mongoTemplate.count(countQuery,
        Order.class);
38
39      return Mono.zip(contentMono, totalCountMono)
40          .map(tuple -> {
41            List<Order> orders = tuple.getT1();
42            long totalElements = tuple.getT2();
43            int totalPages = (int) Math.ceil(
44              (double) totalElements / size
45            );
46            int elementsInPage = orders.size();
47
48            return PagedResDTO.<OrderResDTO>builder()
49                .content(orders.stream().map(orderMapper::toDto)
50                  .toList())
51                .totalElements(totalElements)
52                .totalPages(totalPages)
53                .currentPage(page)
54                .elementsInPage(elementsInPage)
55                .build();
56          });
57  }
58 }
```

Listing 4.3: Spring Boot ReactiveMongoTemplate Dynamic Query

4.1.2 Implementing Cloud-Native Services with Quarkus

In stark contrast to Spring Boot's proprietary ecosystem, the Travel Catalog API was developed using Quarkus, which leverages standard Jakarta EE and MicroProfile specifications.

Spring Boot, while open-source and massively popular, is fundamentally a proprietary ecosystem. It relies heavily on its own specific paradigms, interfaces, and annotations — such as `@RestController` for routing and `@Autowired` for dependency injection. Applications built with Spring are tightly coupled to the Spring framework and cannot be trivially migrated to other Java environments.

Conversely, Quarkus was designed from the ground up to implement open, vendor-neutral industry standards, primarily **Jakarta EE** (the modern, community-driven successor to Java EE, hosted by the Eclipse Foundation) and **MicroProfile** (a set of specifications specifically optimized for cloud-native microservices).

Instead of forcing developers to learn a new proprietary API, Quarkus acts as an ultra-fast runtime for these established standards. For example, at the presentation layer, the `FlightController` utilizes standard JAX-RS (Java API for RESTful Web Services) annotations (e.g., `@Path`, `@GET`, `@RestPath`) instead of Spring's Web annotations. For component management, it uses CDI (Contexts and Dependency Injection).

While Quarkus strictly adheres to these standards for routing and injection, it modernizes the execution model. The reactive streams are managed by SmallRye Mutiny, substituting Reactor's `Mono`/`Flux` with `Uni`/`Multi`. Mutiny is designed with an explicitly event-driven API, focusing on reacting to events (e.g., `.onItem()`, `.onFailure()`) rather than data manipulation operators.

Furthermore, the most profound architectural departure in Quarkus lies in its data access layer. Quarkus eliminates the traditional Repository interface by implementing the **Active Record Pattern** via MongoDB Panache. This paradigm drastically reduces boilerplate code. Unlike traditional Java beans, Panache encourages the use of public fields, eliminating the need for verbose getter and setter generation. Furthermore, business logic tightly coupled to the entity is kept directly within the model class.

As an example, let's consider the `Flight` domain entity which directly extends `ReactivePanacheMongoEntity`:

```
1 @Builder
2 @NoArgsConstructor
3 @AllArgsConstructor
4 @MongoEntity(collection = "flights")
5 public class Flight extends ReactivePanacheMongoEntity {
6
7   public String code;
8   public Integer capacity;
9   public String airline;
```

```
10    public String airlineLogo;
11    public FlightDestination from;
12    public FlightDestination to;
13    public Instant departureTime;
14    public Instant arrivalTime;
15    public Price price;
16 }
```

Listing 4.4: Quarkus Panache Active Record Entity

Because the entity itself handles data access, there is no need to inject a separate repository into the Service layer. Database operations are invoked directly on the `Flight` class itself. For instance, creating a new flight or fetching one by its ID is accomplished natively:

```
1  // Inside FlightService.java
2  public Uni<FlightResDTO> getFlightById(ObjectId id) {
3    return Flight.<Flight>findById(id)
4        .onItem().ifNull().failWith(() -> new FailureException(
5            ResponseEnum.NOT_FOUND, "Flight not found")
6        )
7        .map(flightMapper::toDto);
8  }
9
10 public Uni<FlightResDTO> addFlight(@Valid FlightReqDTO
       flightReqDTO) {
11   Flight flight = flightMapper.toEntity(flightReqDTO);
12   return flight.<Flight>persist().map(flightMapper::toDto);
13 }
```

Listing 4.5: Quarkus Mutiny and Panache Service Logic

For complex dynamic queries, Panache provides a `ReactivePanacheQuery` object that natively integrates with BSON `Filters`, providing a highly expressive alternative to Spring's `Criteria` builder, while maintaining the same non-blocking execution profile.

```
1    // Inside FlightService.java
2    public Uni<PagedResDTO<FlightResDTO>> getFlights(int page, int
       size, String sort, String order, String timezone, @Valid
       FlightFilter filters) {
3      Bson sortExpr = order.equalsIgnoreCase("desc")
4          ? Sorts.descending(sort)
5          : Sorts.ascending(sort);
6
7      Bson filterQuery = Filters.and(
8          filters.getCode() != null ?
9              Filters.regex("code", Pattern.compile(filters.getCode
       (), Pattern.CASE_INSENSITIVE)) :
10             Filters.exists("code"),
```

```
        filters.getAirline() != null ?
            Filters.regex("airline",
                Pattern.compile(filters.getAirline(), Pattern.
    CASE_INSENSITIVE)) :
            Filters.exists("airline"),
        // Other filters...
    );

    ReactivePanacheQuery<Flight> query = Flight
        .find(filterQuery, sortExpr)
        .page(Page.of(page, size));

    return Uni.combine().all().unis(query.list(), query.count())
        .asTuple()
        .map(tuple -> {
          List<Flight> flights = tuple.getItem1();
          Long totalElements = tuple.getItem2();

          int totalPages = (int) Math.ceil(
            (double) totalElements / size
        );
          int elementsInPage = flights.size();

          return PagedResDTO.<FlightResDTO>builder()
              .content(flights.stream().map(flightMapper::toDto)
                .toList())
              .totalElements(totalElements)
              .totalPages(totalPages)
              .currentPage(page)
              .elementsInPage(elementsInPage)
              .build();
        });
  }
```

Listing 4.6: Quarkus Panache Reactive Query with Dynamic Filters

4.2 The Backend-For-Frontend Layer

In a distributed architecture, forcing a frontend application to orchestrate HTTP calls across dozens of distinct microservices introduces significant network latency and tightly couples the client to the backend topology. To resolve this, the Smart Travel platform implements a *Backend-For-Frontend* (BFF) built with Spring Boot. The BFF serves as a unified API Gateway, aggregating downstream data through a centralized GraphQL interface while simultaneously acting as the platform's primary security perimeter.

4.2.1 GraphQL Schema and Data Fetchers

Unlike REST, which exposes fixed data structures across multiple endpoints, GraphQL exposes a single endpoint that allows the client to explicitly request only the data it needs. The foundation of this contract is the schema definition.
As shown in the `agencyPackage.graphql` example, the schema defines strictly typed queries, mutations, and complex domain entities. Because GraphQL supports nested types, a single query for an `AgencyPackage` can aggregate the package metadata, its associated `DestinationSummary`, and the nested `FlightOrder` arrays, perfectly modeling the hierarchical nature of the travel domain.

```
extend type Query {
    getAgencyPackages(page: Int, size: Int, sort: String, order:
   String, filters: AgencyPackageFilter): AgencyPackagePage!
}

type AgencyPackage {
    id: ID!
    name: String!
    status: String!
    totalPrice: Price!
    destination: DestinationSummary!
    departureFlight: FlightOrder!
    accommodation: AccommodationOrder!
    activities: [ActivityOrder!]
    agentInfo: UserSummary!
}
```

Listing 4.7: GraphQL Schema Definition Snippet

To resolve these queries, the `AgencyPackageController` maps directly to the schema definitions using `@QueryMapping` and `@MutationMapping` annotations from the Spring for GraphQL library.

4.2.2 Context-Aware Resolving and Security

A critical responsibility of the BFF is enforcing security before traffic reaches the internal microservices network. Because the BFF integrates with Spring Security, the GraphQL resolvers are inherently context-aware.
Using the `@AuthenticationPrincipal` annotation, the controller intercepts the reactive security context to determine the user's role on the fly. This enables sophisticated, business-level filtering directly at the gateway. For example, if an unauthenticated Guest or a Customer queries the catalog, the resolver dynamically overrides the query filters to ensure they only retrieve packages with a `PUBLISHED` status, securely hiding drafts created by Travel Agents. Furthermore, destructive operations are strictly protected using `@PreAuthorize`.

```java
@Controller
@RequiredArgsConstructor
public class AgencyPackageController {

  private final AgencyPackageService agencyPackageService;

  @QueryMapping("getAgencyPackages")
  public Mono<PagedResDTO<AgencyPackageResDTO>> agencyPackages(
      @Argument Integer page, @Argument Integer size, @Argument AgencyPackageFilter filters,
      @AuthenticationPrincipal Mono<CustomUserDetails> userMono
  ) {
    final AgencyPackageFilter inputFilters = filters != null ? filters : new AgencyPackageFilter();

    return userMono
        .map(Optional::of)
        .defaultIfEmpty(Optional.empty())
        .flatMap(user -> {
          // Prevent showing private/draft packages to unauthorized users
          if (user.isEmpty() || user.get().hasRole(UserRoleEnum.CUSTOMER)) {
            inputFilters.setStatus(PackageStatus.PUBLISHED);
            inputFilters.setAuthorId(null);
          }
          return agencyPackageService.getAgencyPackages(page, size, sort, order, inputFilters);
        });
  }

  @MutationMapping("deleteAgencyPackage")
  @PreAuthorize("hasAnyRole('ADMIN', 'AGENT')")
  public Mono<OkResDTO> deleteAgencyPackage(@Argument String id) {
    return agencyPackageService.deleteAgencyPackage(id);
  }
}
```

Listing 4.8: Context-Aware GraphQL Controller

4.2.3 Inter-Service Communication via WebClient

Once the GraphQL query is authorized and parsed, the BFF must fetch the underlying data from the isolated microservices. To achieve this without stalling the event loop, the service layer utilizes Spring's reactive `WebClient`, a non-blocking HTTP client. In the `AgencyPackageServiceImpl`, incoming GraphQL arguments are mapped to standard REST query parameters. The WebClient then dispatches

an asynchronous GET request to the downstream Quarkus Travel Catalog API. Moreover, it incorporates reactive error handling using `.onStatus()`. If the downstream microservice returns an HTTP error (e.g., a 404 Not Found or 400 Bad Request mapped to a `ProblemDetail` standard), the WebClient intercepts it and translates it into a `GraphqlFailureException`, ensuring that internal REST errors are elegantly propagated back to the Angular frontend as standardized GraphQL error objects.

```
@Service
public class AgencyPackageServiceImpl implements
    AgencyPackageService {

  private final WebClient webClient;

  @Override
  public Mono<AgencyPackageResDTO> getAgencyPackageById(String id)
    {
    return webClient.get()
        .uri("/{id}", id)
        .retrieve()
        .onStatus(HttpStatusCode::isError, response ->
            response.bodyToMono(ProblemDetail.class).flatMap(
                    problem -> Mono.error(
                        new GraphqlFailureException(problem)
                    )
            )
        )
        .bodyToMono(AgencyPackageResDTO.class);
  }
}
```

Listing 4.9: Reactive Inter-Service Communication via WebClient

4.3 Frontend Development

While the backend microservices handle the complex domain logic and data persistence, the frontend application is responsible for delivering a fluid, highly interactive user experience. The Smart Travel client was developed as a (SPA) using the modern Angular framework.

By leveraging Angular's opinionated architecture, the frontend is strictly organized into modules, services, and components. Furthermore, the frontend relies heavily on Apollo Client for GraphQL communications and RxJS for asynchronous state management. This ensures that the user interface remains responsive and synchronized with the backend data streams without requiring full-page reloads.

4.3.1 Component-Based UI and Lazy Loading

The UI is built using a strict component-based architecture. To accelerate development and maintain a cohesive, accessible design language, the application integrates **PrimeNG**, a comprehensive suite of pre-built Angular UI components. These standardized elements (such as breadcrumbs, accordions, data tables, and dynamic dialogs) are seamlessly composed alongside custom, domain-specific building blocks (e.g., `FlightTabComponent`, `FlightViewComponent`) to construct complex views. For example, the flight search page (Figure 4.1) leverages PrimeNG structural components wrapped within isolated Angular components. This encapsulation ensures that styles and business logic do not bleed across the application, maximizing reusability and maintainability.

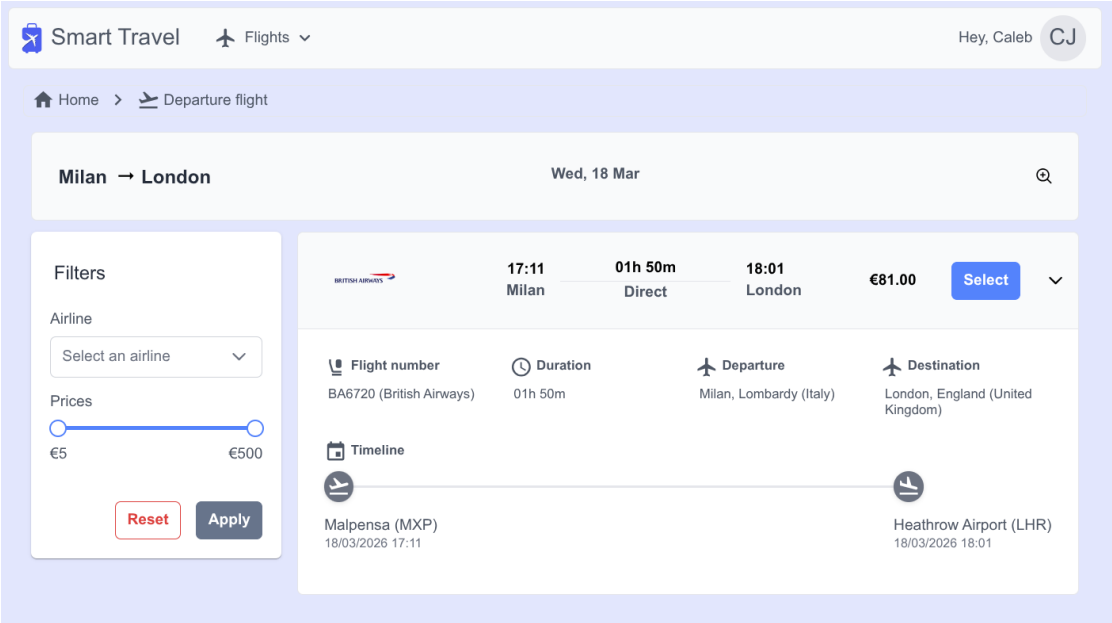

Figure 4.1: Smart Travel Flight Search UI

To guarantee fast initial load times, the application implements *Route-Level Lazy Loading*. Instead of compiling the entire application into a single, massive JavaScript bundle, the `app.routes.ts` configuration dynamically loads feature components only when the user navigates to their respective paths using the `loadComponent` syntax. Additionally, access to these routes is strictly governed by Angular *Guards* (e.g., `isCustomerGuard`), which actively interact with the authentication layer to prevent unauthorized access to sensitive views like the Checkout page.

```
export const routes: Routes = [
  {
    path: 'flights',
    loadComponent: () => import('./features/flight/flight.
   component').then((m) => m.FlightComponent),
    canActivate: [isUnauthenticatedOrCustomerGuard],
  },
  {
    path: 'checkout',
    loadComponent: () => import('./features/checkout/checkout.
   component').then((m) => m.CheckoutComponent),
    canActivate: [
        redirectLoginGuard, isCustomerGuard, IsOrderPresentGuard
    ],
  }
];
```

Listing 4.10: Angular Route Configuration with Lazy Loading and Guards

4.3.2 Data Fetching with Apollo GraphQL

To consume the data aggregated by the Spring Boot BFF, the Angular application utilizes the Apollo Client. Instead of relying on the standard `HttpClient` to make generic REST calls, services are strictly typed to the GraphQL schema.
As demonstrated in the `FlightService`, GraphQL queries (`gql`) are dynamically constructed based on user input, pagination parameters, and the browser's local timezone. Crucially, the Apollo `watchQuery` method does not return a standard JavaScript `Promise`. Instead, it returns an RxJS `Observable` (`valueChanges`). This allows the service to leverage powerful functional operators like `map()` to transform the raw GraphQL response into a standardized `QueryResult` interface before it ever reaches the UI components.

```
@Injectable({ providedIn: 'root' })
export class FlightService {
  constructor(private apollo: Apollo) {}

  getFlightDetails(id: string): Observable<QueryResult<Flight>> {
    const query = gql`
      {
        getFlightById(id: "${id}") {
          id code capacity airline
          price { value currency }
          # ... nested fields
        }
      }`;

```

```
15      return this.apollo.watchQuery<{ getFlightById: Flight }>({
        query })
16        .valueChanges.pipe(
17          map(({ data, loading, errors }) => ({
18            loading,
19            error: errors && errors[0] ? errors[0].message : null,
20            data: data?.getFlightById ?? null,
21          }))
22        );
23    }
24  }
```

Listing 4.11: Apollo GraphQL Query via Angular Service

4.3.3 Reactive State Management with RxJS

Managing state — such as tracking which step of the checkout process a user is on or reacting to changes in the URL search parameters — is notoriously difficult in complex SPAs. To solve this, the application embraces a fully reactive paradigm using RxJS.

In the `FlightComponent`, standard variables are replaced with `BehaviorSubjects` and `Observables`. When the user modifies the URL search filters, the component does not manually trigger a data reload. Instead, it subscribes to the `ActivatedRoute.queryParamMap` Observable. Any change to the URL is automatically piped and mapped into a new state object.

```
1   export class FlightComponent implements OnInit {
2     private route = inject(ActivatedRoute);
3     stepSubject = new BehaviorSubject<PackageStep>(PackageStep.
      FLIGHT_DEPARTURE);
4     step$ = this.stepSubject.asObservable();
5
6     flightParams$!: Observable<{ searchParams: FlightQueryParams,
      filterParams: FlightFilter }>;
7
8     ngOnInit() {
9       // Automatically extract query params on every route change
10      this.flightParams$ = this.route.queryParamMap.pipe(
11        map((params) => ({
12          searchParams: this.extractSearchParams(params),
13          filterParams: this.extractFilterParams(params),
14        }))
15      );
16    }
17  }
```

Listing 4.12: Reactive Component State Management

To bridge this reactive TypeScript logic with the HTML view, the application leverages Angular's modern control flow syntax (`@if`) combined with the `async` pipe. The `async` pipe automatically subscribes to the `step$` Observable when the component renders, extracts the current value (`as currentStep`), and automatically unsubscribes when the component is destroyed, completely eliminating the risk of memory leaks.

```
@if (step$ | async; as currentStep) {
  <p-breadcrumb
    [model]="currentStep === step.FLIGHT_DEPARTURE ? items.slice(0, 2) : items"
    (onItemClick)="handleChangeStep($event.item['index'])" />
}
<flight-view [step$]="step$" [params$]="flightParams$">
  <flight-tab [params]="(flightParams$ | async)?.searchParams" />
</flight-view>
```

Listing 4.13: Angular Modern Control Flow and Async Pipe

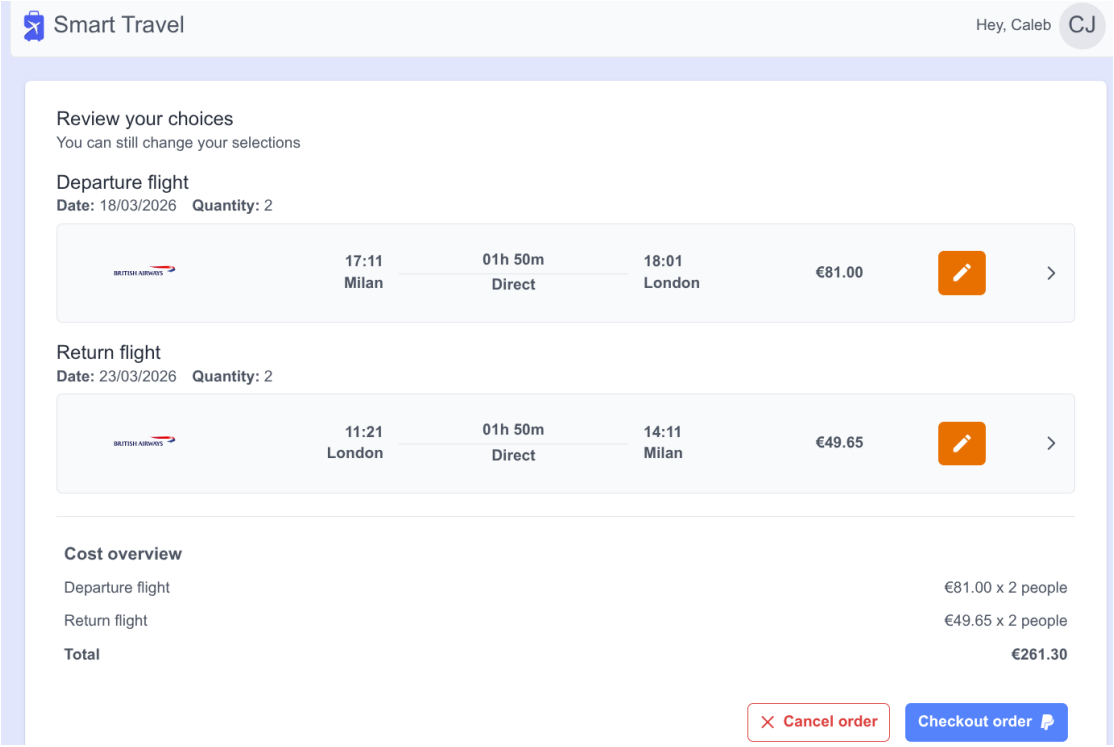

Figure 4.2: The Checkout interface, displaying the state aggregated through the reactive workflow prior to payment gateway redirection.

4.4 End-to-End Authentication and Security

Securing a distributed microservices platform requires a robust, stateless authentication mechanism. The Smart Travel application implements a *JSON Web Token* (JWT) architecture, establishing a secure chain of trust from the Angular SPA to the Spring Boot BFF. Because the platform relies on strict *Role-Based Access Control* (RBAC) to differentiate between Guests, Customers, Travel Agents, and Administrators, this security context must be propagated and validated without blocking the reactive event loops.

4.4.1 The Reactive Spring Security Perimeter

The BFF acts as the primary security perimeter for the entire application. Because the BFF is built on Spring WebFlux, traditional blocking Servlet filters (such as `OncePerRequestFilter`) cannot be used. Instead, the application configures a reactive `SecurityWebFilterChain`.
As demonstrated in the `SecurityConfig`, the application explicitly disables stateful sessions and CSRF protection (which is unnecessary for stateless JWTs). Furthermore, because GraphQL typically returns a `200 OK` HTTP status even when errors occur, standard Spring Security 401 Unauthorized responses can break GraphQL clients. To prevent this, a custom `ServerAuthenticationEntryPoint` was implemented to intercept unauthorized access attempts and format them into a strictly compliant GraphQL error JSON structure.

```
1  @Configuration
2  @EnableWebFluxSecurity
3  @EnableReactiveMethodSecurity
4  class SecurityConfig {
5
6    @Bean
7    SecurityWebFilterChain springSecurityFilterChain(
8        ServerHttpSecurity http,
9        ReactiveAuthenticationManager authenticationManager,
10       ServerAuthenticationConverter authenticationConverter,
11       ServerAuthenticationEntryPoint authenticationEntryPoint) {
12
13     var authenticationWebFilter = new AuthenticationWebFilter(
     authenticationManager);
14     authenticationWebFilter.setServerAuthenticationConverter(
     authenticationConverter);
15
16     return http.authorizeExchange(exchanges -> exchanges
17             .pathMatchers(HttpMethod.OPTIONS).permitAll()
18             .pathMatchers(HttpMethod.POST, "/graphql").permitAll()
19             .anyExchange().authenticated())
```

```
20          .addFilterAt(authenticationWebFilter,
21              SecurityWebFiltersOrder.AUTHENTICATION)
22          .exceptionHandling(exceptionHandlingSpec ->
23              exceptionHandlingSpec.authenticationEntryPoint(
    authenticationEntryPoint)
24          )
25          .csrf(ServerHttpSecurity.CsrfSpec::disable)
26          .build();
27    }
28 }
```

Listing 4.14: Reactive Spring Security Configuration

When a request enters the filter chain, the `JwtServerAuthenticationConverter` reactively intercepts the `Authorization` HTTP header. If a `Bearer` token is present, it extracts the payload and converts it into a custom `JwtToken` object, allowing the `ReactiveAuthenticationManager` to validate the cryptographic signature and populate the `ReactiveSecurityContext` with the user's `CustomUserDetails` (containing their ID, email, and roles).

4.4.2 Role-Based Access Control (RBAC)

With the reactive security context established at the gateway, the application can securely enforce RBAC at the GraphQL resolver level using the `@PreAuthorize` annotation. The `AuthController` relies on this mechanism to ensure that only properly authenticated users can query sensitive profile information.

```
1  @Controller
2  @RequiredArgsConstructor
3  public class AuthController {
4    private final AuthService authService;
5    private final UserService userService;
6
7    @MutationMapping("login")
8    public Mono<LoginResDTO> login(@Argument LoginReqDTO loginReq) {
9      return authService.login(loginReq.getEmail(), loginReq.
     getPassword());
10   }
11
12   @QueryMapping("profile")
13   @PreAuthorize("isAuthenticated()")
14   public Mono<UserNoPwdResDTO> profile(@AuthenticationPrincipal
     CustomUserDetails user) {
15     return userService.getUserById(user.getId());
16   }
17 }
```

Listing 4.15: GraphQL Authentication Controller

4.4.3 Frontend Security State Management

On the Angular client side, security is managed reactively via the `AuthService`. When the user successfully authenticates via the `login` GraphQL mutation, the BFF returns a short-lived Access Token and a long-lived Refresh Token. These tokens are persisted in the browser's local storage via the `TokenService`.
To ensure the UI instantly reflects the user's authentication state across all active components, the `AuthService` encapsulates the user data within an RxJS `BehaviorSubject`.

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  private userSubject = new BehaviorSubject<User | null>(null);
  user$: Observable<User | null>;

  // Expose roles for route guards and UI rendering
  isCustomer(): boolean {
    return this.isLoggedIn() && this.getUser()!.role === UserRole.CUSTOMER;
  }

  login(email: string, password: string): Observable<QueryResult<LoginRes>> {
    return this.apollo.mutate<{ login: LoginRes }>({
        mutation: LOGIN_MUTATION,
        variables: { loginReq: { email, password } },
      })
      .pipe(
        tap(({ data }) => {
          if (data) {
            this.tokenService.setAccessToken(data.accessToken);
            this.tokenService.setRefreshToken(data.refreshToken);
            this.userSubject.next(data.user); // Reactively update the UI state
          }
        })
      );
  }
}
```

Listing 4.16: Angular Reactive Authentication Service

On application startup, the frontend automatically queries the `profile` endpoint. If the Access Token has expired, the client invokes the `refresh` mutation to acquire new credentials. Furthermore, Angular Route Guards (e.g., `IsUnauthenticatedGuard`) observe the `user$` stream, actively preventing unauthorized users from accessing protected views (like the Checkout page or the Travel Agent Dashboard) without requiring complex, synchronous permission checks.

4.5 Core Business Workflow

The checkout and payment process represents the most complex distributed workflow within the Smart Travel platform. It perfectly illustrates the integration of external APIs (PayPal), synchronous user redirects, and asynchronous event-driven backend processes.
Executing financial transactions across distributed microservices introduces the risk of partial failures (e.g., the "dual-write" problem). If a service successfully updates the database but fails to publish the subsequent event to the message broker due to network latency, the system enters an inconsistent state. To guarantee absolute data consistency and at-least-once message delivery, the platform implements the **Transactional Outbox Pattern** combined with *Change Data Capture* (CDC) via Debezium and RabbitMQ.
The workflow is architected into three distinct chronological phases.

4.5.1 Phase 1: Order Creation and Outbox Pattern

The flow begins synchronously when a user confirms their cart. The Angular frontend dispatches an order creation request through the BFF to the Order API. The Order API initiates a server-to-server call to PayPal using the PayPal SDK, requesting the creation of a new payment session. It explicitly provides a `returnUrl` (where PayPal should redirect the user upon success) and a `cancelUrl`. Upon receiving the payment token from PayPal, the Order API persists the `Order` entity in the MongoDB database with an initial status of `PENDING`. Finally, it returns the generated PayPal checkout URL to the frontend, which performs a hard redirect to the external gateway.
After the user successfully authorizes the payment on the external PayPal interface, they are redirected back to the BFF, which subsequently issues a `capture` request to the Order API.
At this juncture, the Order API must perform two critical actions: update the order status to `PAID` and trigger the background processes to officially capture the funds. To avoid the dual-write problem, the service employs the Transactional Outbox pattern.
Instead of directly communicating with RabbitMQ, the Order API writes a `PayPalOutboxEvent` document into a dedicated `orders_outbox_events` MongoDB collection. Because both the order update and the outbox insertion occur within the same reactive database transaction, they are guaranteed to either succeed or fail together.

```java
@Document(collection = "orders_outbox_events")
public class PayPalOutboxEvent {
  @Id
  private String id;
  private String orderId;
  private String paypalToken;
  private String payerId;
  @JsonDeserialize(using = MongoTimestampDeserializer.class)
  private Instant createdAt;
}
```

Listing 4.17: Transactional Outbox Event Model

Notably, the Order API implements a highly optimized variant of this pattern: log-tailing deletion. As seen in the service layer, immediately after the `outboxEvent` is saved, it is deleted. This keeps the outbox collection completely empty, saving storage space, while still forcing MongoDB to append the insert operation to its internal replica set oplog (Operations Log).

```java
// Inside OrderServiceImpl.java
@Transactional
public Mono<Void> captureOrder(@Valid CaptureOrderReqDTO
    captureReq) {
     return getOrderEntityByToken(captureReq.getToken())
          .flatMap(order -> {
              // Update order status
              order.getPayment().setStatus(PaymentStatus.PAID);

              var outboxEvent = PayPalOutboxEvent.builder()
                  .orderId(order.getId())
                  .paypalToken(captureReq.getToken())
                  .payerId(captureReq.getPayerId())
                  .build();

              // Atomically save both, then immediately
              // delete the outbox record
              return orderRepository.save(order)
                  .then(payPalOutboxRepository.save(outboxEvent))
                  .flatMap(payPalOutboxRepository::delete);
          }).then();
}
```

Listing 4.18: Transactional Outbox Implementation

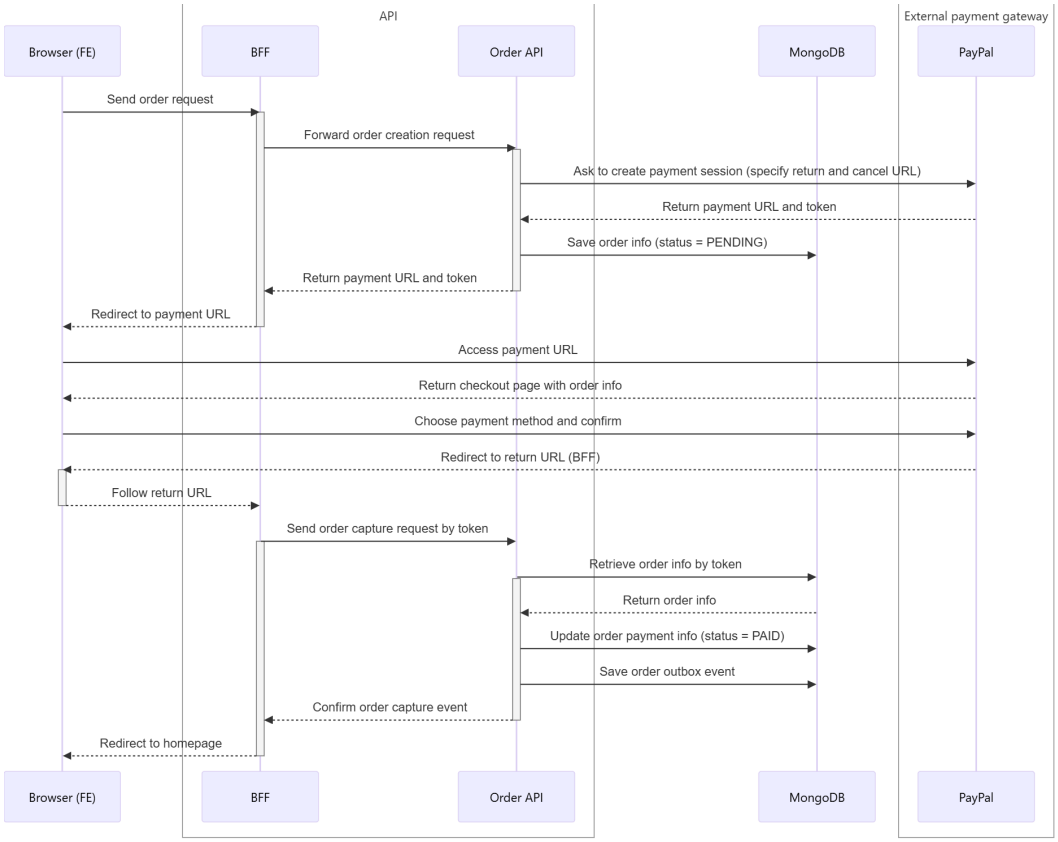

Figure 4.3: Phase 1: Order Creation and external payment gateway redirect.

4.5.2 Phase 2: Asynchronous Capture via Debezium and RabbitMQ

A standalone Debezium service acts as a CDC connector, continuously checking the MongoDB oplog. The instant it detects the ‘insert‘ operation for the `PayPalOutboxEvent`, it captures the payload and reliably publishes it to a RabbitMQ queue.
Simultaneously, the Order API acts as an asynchronous consumer using Project Reactor's `Receiver`. The `PayPalOutboxListener` strictly filters incoming Debezium payloads, ensuring it only reacts to "create" operations (`"op": "c"`). Once validated, it contacts PayPal to definitively capture the authorized funds. If successful, it updates the database status to `CONFIRMED`, dispatches a new event to the `notification_events` queue, and acknowledges (`delivery.ack()`) the message. If a transient error occurs (e.g., PayPal API timeout), the message is negatively acknowledged (`delivery.nack(true)`), safely returning it to the queue for automatic retry processing.

```java
// Inside PayPalOutboxListener.java

@PostConstruct
public void listenAndForward() {
  receiver.consumeManualAck("orders_outbox_events")
    .flatMap(delivery -> {
        // Extract Debezium payload and check operation type
        JsonNode payload = root.path("payload");
        if (!"c".equals(payload.path("op").asText())) {
          delivery.ack();
          return Mono.empty();
        }
        // Execute PayPal Capture
        return Mono.just(payPalService
          .captureOrder(token, payerId)).flatMap(order -> {
            if (order.getStatus().equals(OrderStatus.COMPLETED))
                // Publish notification event and Ack
                return sender.send(Mono.just(notificationEvent))
                    .doOnSuccess(ignored -> delivery.ack());
            else
                // Schedule for retry
                return Mono.fromRunnable(
                  () -> delivery.nack(true)
                );
          });
    }).subscribe();
}
```

Listing 4.19: Reactive RabbitMQ Outbox Listener

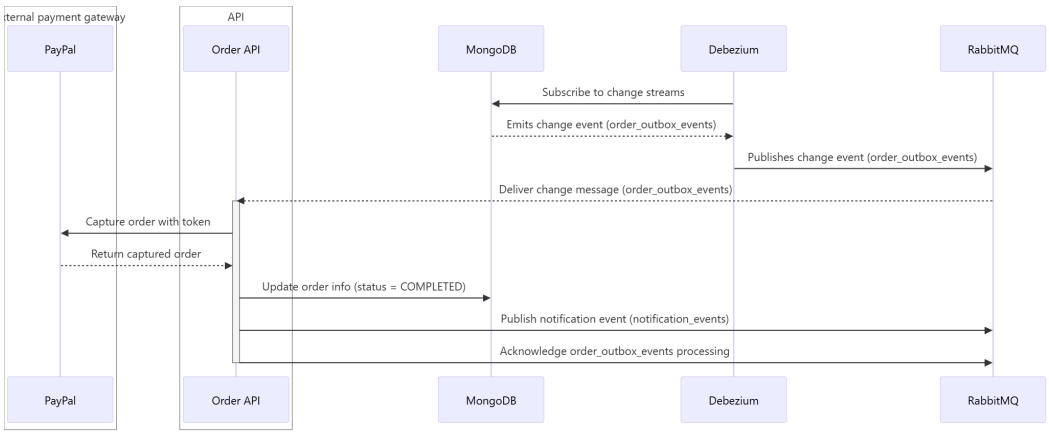

Figure 4.4: Phase 2: The Transactional Outbox pattern, CDC via Debezium, and asynchronous payment capture.

4.5.3 Phase 3: Event-Driven Email Notification

During the final step of the workflow, the Notification API Quarkus microservice listens to the `notification_events` queue using MicroProfile Reactive Messaging. When the `OrderListener` receives the serialized `OrderNotificationDTO`, it dynamically compiles an HTML email template utilizing Quarkus Mailer. During development, these messages are routed to Mailpit, a local SMTP testing server. Because this entire phase is asynchronous and decoupled, any failure in the SMTP server will not impact the user's checkout experience or the financial integrity of the order; RabbitMQ will simply retain the notification event until the mail server recovers and the consumer successfully processes it.

```java
// Inside OrderListener.java

@Incoming("notification_events")
public Uni<Void> consume(Message<byte[]> msg) {
  OrderNotificationDTO order = deserialize(msg.getPayload());
  String customerEmail = order.customerInfo().getEmail();

  return MailingResource.Templates.orderConfirmation(order)
      .to(customerEmail)
      .subject("Order confirmation #" + order.id())
      .send()
      .onItem().transformToUni(x ->
          Uni.createFrom().completionStage(msg.ack()))
      .onFailure().call(e ->
          Uni.createFrom().completionStage(() -> msg.nack(e)));
}
```

Listing 4.20: Quarkus Reactive Messaging Notification Consumer

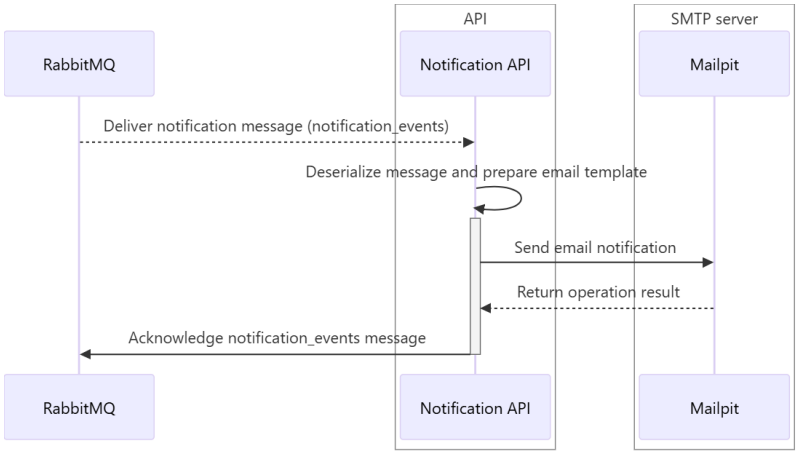

Figure 4.5: Phase 3: Decoupled event consumption and SMTP dispatch.

Chapter 5

Infrastructure Automation and Cloud Deployment

While Chapter 4 detailed the software engineering and implementation of the Smart Travel application, writing source code represents only a fraction of the modern software delivery lifecycle. In a distributed environment comprising multiple independent backend services, external API gateways, message brokers, and isolated databases, manual testing, configuration, and deployment are no longer viable.
This chapter details the transition of the Smart Travel platform from an isolated codebase to a live, scalable, and resilient cloud-native application. It first explores the foundational DevOps culture that drives modern software delivery. Subsequently, it details the implementation of a Continuous Integration and Continuous Delivery (CI/CD) pipeline using Jenkins. Finally, it provides a comprehensive overview of how the platform's workloads are containerized, configured, and orchestrated within an OKD cluster.

5.1 The DevOps Culture

Historically, the software engineering industry operated with rigid silos separating Development (Dev) teams, responsible for writing new features, and IT Operations (Ops) teams, responsible for maintaining system stability. This paradigm inherently created a conflict of interest: developers were incentivized to push changes rapidly to meet business demands, while operations teams were incentivized to block or delay changes to prevent production outages [39].
The advent of *DevOps* represents a cultural and technical movement designed to resolve this conflict by breaking down these organizational silos. At its core, DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications at high velocity [40]. It relies heavily

on a model of shared responsibility, often summarized by the philosophy "you build it, you run it." Rather than handing off unverified code to a separate department, developers take accountability for the entire lifecycle of their software, from the initial commit to production monitoring [39].

In the context of the Smart Travel platform, adopting a DevOps mindset was not merely a philosophical choice, but a strict technical necessity. Deploying a single monolithic application manually over a secure file transfer protocol is tedious but feasible. Conversely, a microservices architecture decomposes an application into suites of independently deployable services [41]. Manually compiling, containerizing, networking, and deploying the Angular frontend, the API Gateway, the distinct Spring and Quarkus domain microservices, Debezium, RabbitMQ, and multiple isolated MongoDB databases is an operational impossibility.

Consequently, the distributed architecture mandates that every code change must be automatically built, comprehensively tested, and seamlessly deployed. The combination of microservices and increased release frequency leads to significantly more deployments, which can only be managed safely through rigorous automation [40].

5.2 CI/CD Pipeline Implementation

To implement the DevOps philosophies discussed in the previous section, the Smart Travel platform relies on a (CI/CD) pipeline orchestrated by Jenkins.

The underlying infrastructure — including Jenkins and the target OKD cluster — was provisioned on an Amazon Web Services (AWS) EC2 instance by the company's IT Operations team. To ensure strict security perimeters, this environment has been entirely isolated from the public internet and only accessible via a secure SSH connection through a bastion host. Operating within this predefined infrastructure, the development focus shifted entirely to automating the software delivery lifecycle: writing declarative deployment pipelines, defining container specifications, and orchestrating the workload.

As illustrated in Figure 5.1, the monolithic application concept is entirely abandoned at the CI/CD level. Instead, every individual microservice and infrastructure component (e.g., frontend, bff, travel catalog, orders, notification) features its own independent Jenkins pipeline, allowing them to be built, versioned, and deployed completely autonomously.

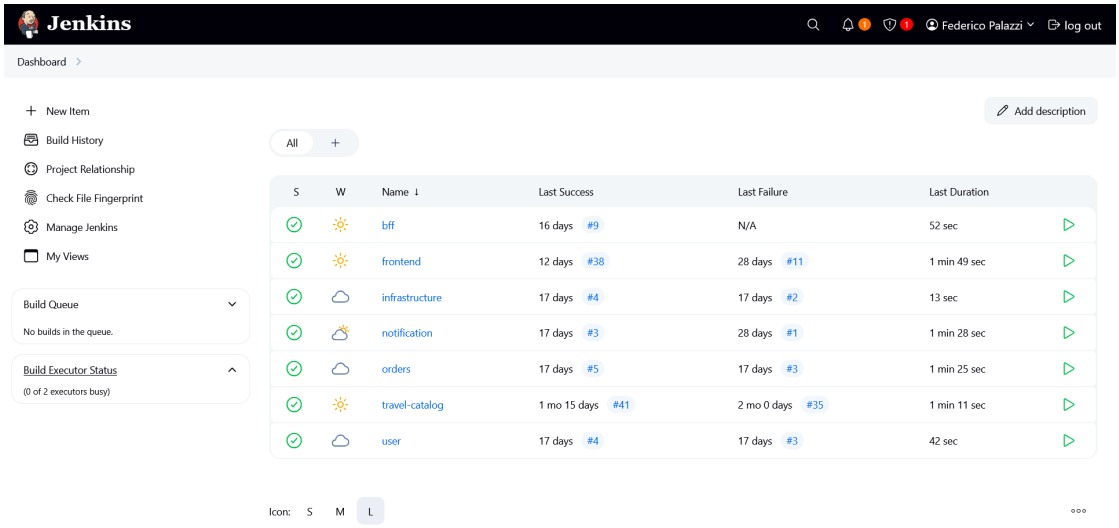

Figure 5.1: The Jenkins Dashboard displaying the independent deployment pipelines for each microservice.

5.2.1 Declarative Pipelines and Isolated Build Agents

The deployment logic for each microservice is codified using Jenkins Declarative Pipelines (`Jenkinsfile`) stored directly alongside the application source code. A critical architectural choice in these pipelines is the use of isolated, ephemeral build agents via Docker.

Rather than installing Java, Maven, or Node.js directly onto the Jenkins host server — which often leads to version conflicts and "it works on my machine" anomalies — the pipelines instantiate specific Docker containers (for example, `maven:3.9.6-eclipse-temurin-21` for the backend and `node:20-alpine` for the frontend) to execute the build steps. This guarantees absolute reproducibility across all environments.

```
stage('Build App') {
    agent {
        docker {
            image 'node:20-alpine'
            args '-u root'
        }
    }
    steps {
        script {
            // Dynamically extract version for Docker tagging
            def version = sh(
              script: "node -p require('./package.json').version",
              returnStdout: true
```

```
14                ).trim()
15                env.IMAGE_TAG = version
16            }
17            sh '''
18              npm ci
19              npm run build
20            '''
21        }
22    }
```

Listing 5.1: Frontend Jenkinsfile: Isolated Node.js Build Agent

5.2.2 Dependency Resolution and Dynamic Tagging

For the Java microservices (such as the BFF and Notification API), the pipeline must resolve a complex dependency graph before compilation. Because the architecture relies on a *common* module for shared DTO and interfaces (ensuring strict API contracts between services), the Jenkins pipeline implements a `Build Common` stage. As shown in the Jenkins console output (Figure 5.2), this stage securely checks out the shared repository using Git credentials and installs it into the ephemeral Maven `.m2` cache, allowing the subsequent application build to resolve the dependencies successfully.

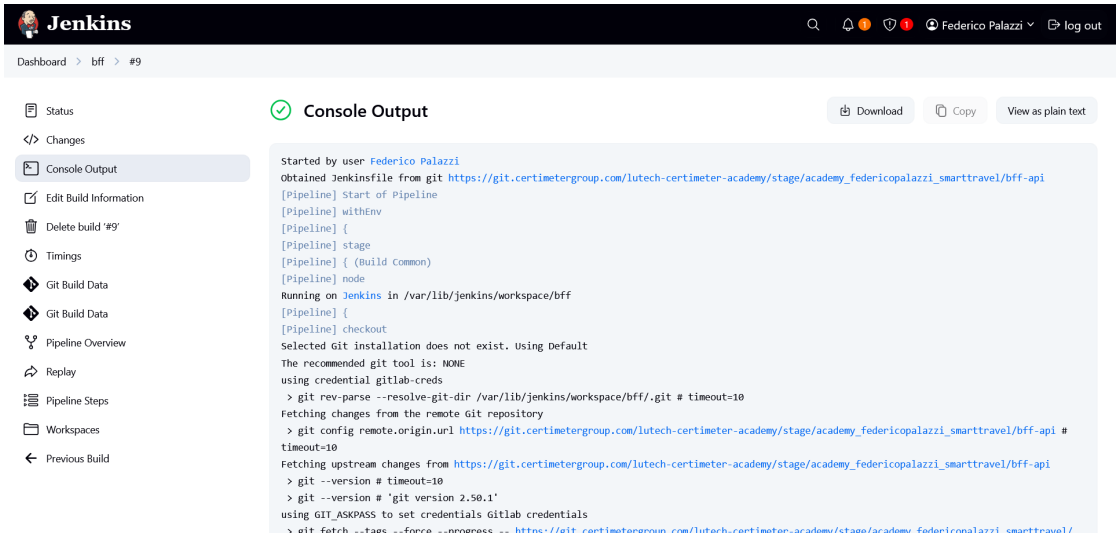

Figure 5.2: Console output of the BFF pipeline, demonstrating the successful execution of the `Build Common` declarative stage.

Furthermore, the pipelines eliminate hardcoded versioning. Utilizing Groovy scripting, Jenkins dynamically interrogates the project's metadata files (`pom.xml`

for Java, `package.json` for Angular) to extract the current semantic version. This version is directly injected into the pipeline's environment variables and used as the official tag for the Docker image, ensuring perfect alignment between the codebase version and the deployed container artifact.

5.2.3 Containerization Strategy

Once the application binaries are compiled, Jenkins triggers the containerization process. The Dockerfiles are strictly optimized for production.
For the backend services, lightweight JRE (Java Runtime Environment) base images are utilized (`eclipse-temurin:21-jdk`). Notably, the Quarkus Notification API pipeline takes advantage of Quarkus's highly optimized *uber-jar* packaging, avoiding the need for complex multi-stage Docker builds. For the Angular frontend, the compiled static assets are injected into an Nginx server image. The `Dockerfile` overrides the default Nginx configuration and adjusts directory permissions to allow execution under restricted security contexts.

```
1  FROM nginx:1.29
2
3  # Copy custom nginx config for SPA routing
4  COPY nginx.conf /etc/nginx/nginx.conf
5  COPY default.conf /etc/nginx/conf.d/default.conf
6
7  # Support running as arbitrary user for enhanced security
8  RUN chmod g+rwx /var/cache/nginx /var/run /var/log/nginx
9
10 # Copy built Angular app from Jenkins workspace
11 COPY dist/frontend/browser /usr/share/nginx/html
12
13 EXPOSE 8088
14 CMD ["nginx", "-g", "daemon off;"]
```

Listing 5.2: Angular Frontend Dockerfile

5.2.4 Artifact Registry Publishing

The final stage of the CI pipeline is publishing the immutable container image directly to the cluster's internal registry utilizing OKD *ImageStreams*. Unlike traditional standalone container registries, an ImageStream provides an abstraction layer over container images from within the OKD cluster. This native integration is highly advantageous, as it allows OKD deployment configurations to automatically detect when a new image tag is pushed and immediately trigger a rolling update of the microservice.

To maintain security during this process, Jenkins utilizes the `withCredentials` plugin to inject the necessary registry authentication tokens directly into the execution context without exposing them in the console logs.
The pipeline authenticates against the internal OKD registry, tags the newly built image with the dynamically extracted version and the designated cluster namespace, and pushes the artifact. Finally, it systematically cleans up the local Docker daemon cache to prevent storage exhaustion on the Jenkins build server. At this point, the artifact resides natively within the OKD environment, ready for container orchestration and workload distribution.

5.3 Container Orchestration with OKD

While Docker is excellent for packaging applications into immutable images, running a distributed microservices platform requires much more than simply starting containers. The system requires internal DNS resolution, load balancing, automated health checks, secure secret management, and self-healing capabilities (automatically restarting a crashed service). To achieve this, the industry relies on container orchestrators, specifically Kubernetes.
The Smart Travel platform is deployed on an OKD cluster. A cluster is essentially a unified pool of compute resources (virtual machines known as worker nodes) managed by a central control plane. OKD, as the upstream community edition of Red Hat OpenShift, wraps standard Kubernetes with enterprise-grade security constraints, native CI/CD pipelines, and enhanced developer dashboards.

5.3.1 OKD Cluster Overview and Image Management

The deployment lifecycle within OKD begins in the internal container registry. As discussed in Section 5.2, the Jenkins pipeline pushes built artifacts directly to OKD *ImageStreams* (Figure 5.3). An ImageStream is an OpenShift-specific resource that provides an abstraction layer over container images, allowing the cluster to centrally track image tags and metadata.
While OKD ImageStreams possess the capability to automatically trigger rollouts upon detecting a new image, the Smart Travel platform intentionally implements a Continuous *Delivery* model rather than automated Continuous *Deployment*. Therefore, Jenkins is strictly responsible for building, tagging, and securely delivering the immutable artifacts to the cluster's registry. The actual rollout of the new application version is executed deliberately. This architectural decision ensures that administrators maintain strict, manual control over when infrastructure changes are applied to the live environment.

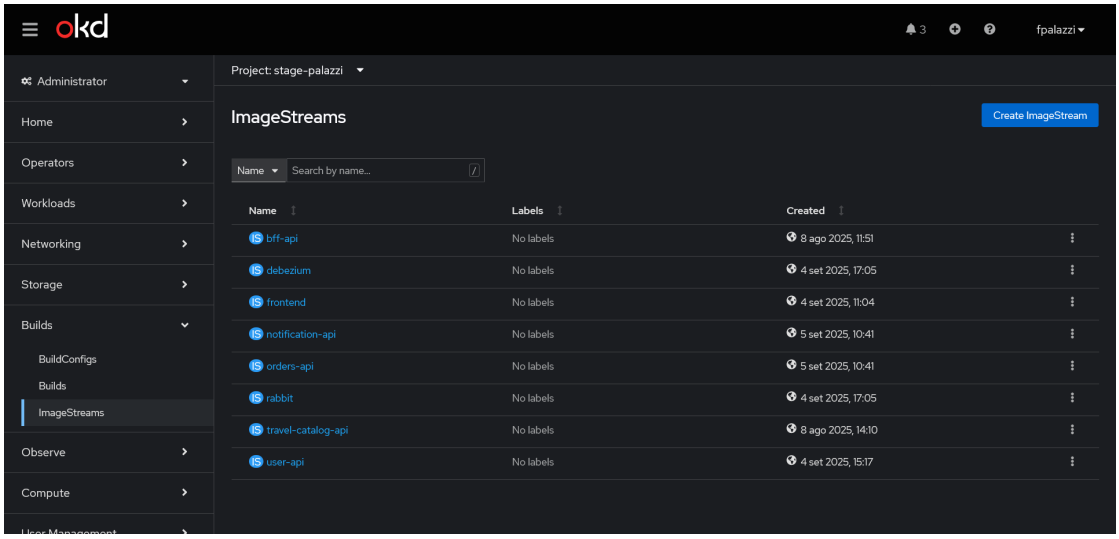

Figure 5.3: The OKD ImageStreams dashboard, demonstrating the internal tracking of the pushed container artifacts.

Once the services are deployed, the OKD Topology view (Figure 5.4) provides a visual representation of the application mesh, grouping interconnected pods, services, and routes.

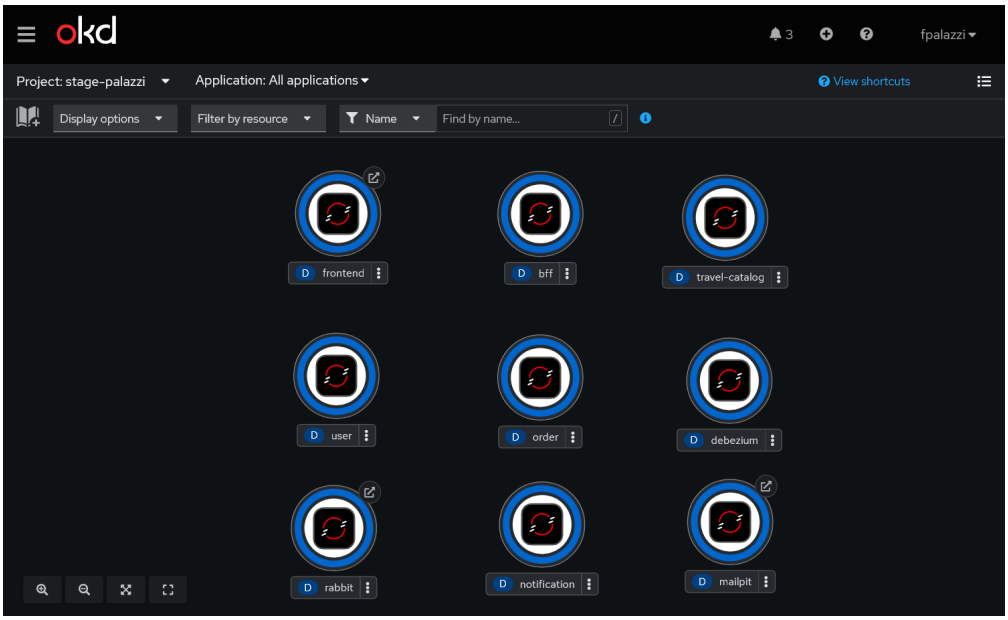

Figure 5.4: The visual topology of the Smart Travel cluster, illustrating the deployed microservices, message broker, and CDC components.

5.3.2 Workload Management: Pods and Deployments

In Kubernetes, the smallest deployable computing unit is a *Pod*, which encapsulates one or more containers. However, Pods are ephemeral; if a node crashes, the Pod dies. To ensure high availability, the platform utilizes `Deployment` manifests. A Deployment acts as a declarative supervisor, ensuring that the specified number of container replicas is constantly running.
Because OKD enforces strict *Security Context Constraints* (SCC), containers cannot run with root privileges by default. This required specific configurations in the deployment files. For example, the Angular frontend deployment explicitly disables privilege escalation and drops all root capabilities, ensuring the Nginx container executes securely.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: frontend
5  spec:
6    replicas: 1
7    template:
8      spec:
9        containers:
10       - name: frontend
11         image: image-registry.openshift-image-registry.svc:5000/
     stage-palazzi/frontend:0.0.14
12         ports:
13         - containerPort: 8088
14         securityContext: # Required by OKD strict security
     constraints
15           allowPrivilegeEscalation: false
16           capabilities:
17             drop:
18               - ALL
19           runAsNonRoot: true
```

Listing 5.3: Frontend Deployment Manifest Snippet

5.3.3 Configuration and Security: ConfigMaps and Secrets

A core tenet of cloud-native development is the strict separation of code and configuration. To inject environment variables (such as downstream service URLs or database credentials) into the containers without hardcoding them in the source code, the deployment utilizes `ConfigMap` and `Secret` resources.
The `app-config` ConfigMap defines the internal cluster URLs used by the BFF to communicate with the internal microservices. Conversely, sensitive data is stored as a base64-encoded `Secret`.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  # Internal Kubernetes DNS names
  TRAVEL_CATALOG_URL: "http://travel-catalog-service:8080"
  ORDER_URL: "http://order-service:8082"
  BROKER_HOST: "rabbit-service"
---
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  # Base64 encoded sensitive credentials
  PAYPAL_CLIENT_ID: QWJY...
  PAYPAL_SECRET: RUZY...
```

Listing 5.4: Cluster Configuration and Secret Resources

These resources are dynamically injected into the deployed Pods via the `envFrom` directive in the deployment manifest, ensuring the containers remain entirely environment-agnostic.

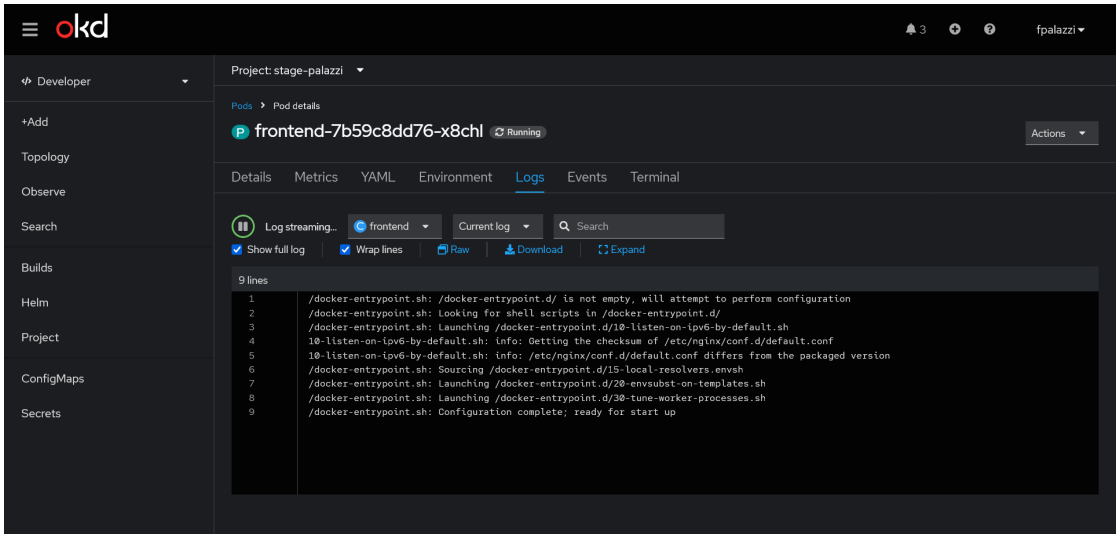

Figure 5.5: OKD Pod Logs view confirming the successful, non-root initialization of the Nginx frontend container.

5.3.4 Networking: Services, Routes, and Reverse Proxying

In a dynamic cluster, Pod IP addresses change constantly. To provide a stable networking interface, OKD uses the `Service` resource. A `Service` acts as an internal load balancer, creating a persistent internal IP and DNS record (e.g., `bff-service.stage-palazzi.svc.cluster.local`).
However, `Services` are only accessible *inside* the cluster. To expose an application to the public internet, we need to utilize a `Route` (an extended version of a Kubernetes Ingress), which provides edge termination for HTTPS and maps a public domain name to an internal service.
A critical architectural security decision was made regarding the BFF API: **it does not possess a public Route**. To minimize the public attack surface, the BFF is completely isolated within the cluster. Instead, the public-facing Angular Frontend (running on Nginx) acts as a secure Reverse Proxy.
The custom `nginx.conf` file is configured to intercept any incoming traffic targeting the `/graphql` or `/api/` paths. Using the internal OKD DNS resolver, Nginx securely forwards these requests directly to the hidden `bff-service`, preserving the original headers and passing through the authentication cookies while keeping the backend entirely shielded from direct external access.

```
server {
    listen       8088;
    server_name  _;
    root   /usr/share/nginx/html;
    index  index.html;

    # Proxy GraphQL API calls to internal BFF service
    location /graphql {
        resolver 172.30.0.10 valid=30s; # DNS resolver - cluster DNS service IP
        # Route to the internal Kubernetes Service (not publicly exposed)
        set $upstream http://bff-service.stage-palazzi.svc.cluster.local:8085/graphql;
        proxy_pass          $upstream;
        proxy_http_version 1.1;
        # Pass through all authentication headers securely
        proxy_set_header    Authorization $http_authorization;
        proxy_set_header    Host $http_host;
        proxy_pass_request_headers on;
    }
    # Serve Angular SPA for all other routes
    location / { try_files $uri $uri/ /index.html; }
}
```

Listing 5.5: Nginx Reverse Proxy Configuration for the Hidden BFF

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - port: 8088
      targetPort: 8088
  type: ClusterIP
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend-route
spec:
  host: smart-travel.apps.stage-soldi-okd.lutechstage.com
  to:
    kind: Service
    name: frontend-service
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
```

Listing 5.6: Frontend Service and Public Route Definition

Chapter 6

Conclusions

6.1 Project Summary

The objective of this thesis was to design, implement, and deploy *Smart Travel*, a comprehensive, cloud-native platform tailored for the modern tourism industry. Moving away from traditional, rigid monolithic architectures, the project successfully embraced a distributed microservices paradigm. By decoupling the application into independent, domain-specific bounded contexts — such as the Travel Catalog, Orders, User Management, and Notifications — the platform achieved the high scalability, fault tolerance, and independent deployability required by contemporary enterprise software.

Throughout the development lifecycle, strict adherence to modern software engineering principles was maintained. The architecture successfully integrated a heterogeneous technology stack, leveraging the reactive capabilities of both Spring Boot and Quarkus to handle highly concurrent workloads without blocking system threads. Furthermore, the frontend was developed as a dynamic Single Page Application using Angular and RxJS, providing a seamless and highly responsive user experience across complex workflows like custom package creation and order checkout.

6.2 Architectural and Technical Achievements

The implementation of the Smart Travel platform served as a rigorous practical application of advanced cloud-native design patterns. Several key architectural achievements define the success of this project:

- **Heterogeneous Reactive Microservices:** The project successfully demonstrated the viability of mixing enterprise Java frameworks. By utilizing Spring

WebFlux for the Orders API and Quarkus Panache for the Travel Catalog, the system capitalized on the respective strengths of each ecosystem while unifying them under a fully non-blocking, reactive execution model.

- **Optimized Client-Server Communication:** The introduction of a Spring for GraphQL BFF successfully mitigated the common microservice pitfalls of over-fetching and under-fetching. The BFF acted as a secure API Gateway, intelligently aggregating downstream REST data and enforcing Role-Based Access Control via stateless JWTs before traffic could reach the internal cluster network.

- **Distributed Data Consistency:** The implementation of the Transactional Outbox Pattern, combined with Change Data Capture via Debezium and RabbitMQ, resolved the critical "dual-write" problem during the financial checkout phase. This guaranteed absolute data consistency between the local MongoDB databases and the asynchronous notification services.

- **DevOps and Infrastructure Automation:** The transition from raw code to a live cloud environment was completely automated. Jenkins CI/CD pipelines successfully managed the dynamic versioning, isolated compilation, and containerization of the artifacts. Furthermore, the deployment onto an OKD cluster demonstrated advanced networking, utilizing Nginx as a reverse proxy to shield internal services from the public internet.

6.3 Challenges and Lessons Learned

Building a distributed, reactive system from the ground up presented significant engineering challenges.
The most profound challenge was the paradigm shift required to adopt *Reactive Programming*. Transitioning from traditional imperative Java (where operations execute sequentially on a single thread) to declarative reactive streams (using Project Reactor's `Mono`/`Flux` and Mutiny's `Uni`/`Multi`) required a fundamentally different mental model. Debugging asynchronous streams, managing backpressure, and ensuring that no blocking calls were accidentally introduced into the event loop demanded rigorous attention to detail.
Additionally, orchestrating the system within the strict security constraints of OKD proved challenging. Unlike standard Docker environments, OKD's Security Context Constraints (SCC) explicitly forbid containers from running as the root user. This required refactoring Dockerfiles and deployment manifests to ensure that services like Nginx possessed the appropriate internal directory permissions without escalating privileges.

6.4 Future Enhancements

While the Smart Travel platform currently possesses a robust and scalable foundation, the microservices architecture naturally lends itself to continuous iteration and enhancement. Several enhancements for future work have been identified to further improe both the technical architecture and the user experience:

- **Advanced Notification System:** Expanding the current asynchronous RabbitMQ notification architecture to support real-time push notifications (e.g., via WebSockets or Server-Sent Events) in addition to a broader range of email alerts, such as itinerary reminders and booking status changes.
- **User Reviews and Ratings:** To foster community trust and improve package quality, a dedicated review microservice could be introduced. This would allow verified customers to rate and review specific accommodations and activities after completing their trips.
- **Geolocation and Mapping Services:** Integrating external geolocation APIs (such as Google Maps or Mapbox) into the frontend to provide users with interactive visual representations of flight routes, hotel locations, and activity venues.
- **Internationalization (i18n) and Localization:** To support a broader demographic and global market reach, the platform could be enhanced with multi-language support and dynamic, real-time currency conversion, orchestrated directly through the BFF layer.
- **Artificial Intelligence Integration:** To further enhance the user experience, the platform could integrate an AI-driven recommendation engine. By analyzing a user's past order history and search queries, a dedicated microservice could suggest highly personalized travel packages and activities.

Bibliography

[1] Muskaan Goyal and Pranav Bhasin. *Moving from monolithic to microservices architecture for multi-agent systems*. 2025. URL: https://arxiv.org/pdf/2505.07838 (cit. on p. 1).

[2] Amazon Web Services (AWS). *What are Microservices?* 2025. URL: https://aws.amazon.com/microservices/ (cit. on p. 5).

[3] Microsoft Learn. *Microservices Architecture Style*. 2025. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (cit. on p. 6).

[4] CoderCo. *Monoliths vs Microservices: A Guide to Choosing the Right Architecture for Your Application*. 2023. URL: https://blog.coderco.io/p/monoliths-vs-microservices-a-guide (cit. on p. 6).

[5] Eyitayo Olaigbe. *Microservices Architecture: Building Resilient, Scalable Applications in the Cloud Era*. 2024. URL: https://www.tayoolaigbe.com/articles/microservices-architecture (cit. on p. 7).

[6] Ray Dhiraj. *Spring WebFlux Reactive REST API*. 2023. URL: https://www.devglan.com/spring-boot/spring-webflux-reactive-rest-api (cit. on p. 8).

[7] Quarkus Official Documentation. *Quarkus Reactive Architecture*. 2024. URL: https://quarkus.io/guides/quarkus-reactive-architecture (cit. on pp. 8, 9).

[8] IBM. *What Is Event-Driven Architecture?* 2024. URL: https://www.ibm.com/topics/event-driven-architecture (cit. on pp. 10, 11).

[9] Microsoft Learn. *Asynchronous message-based communication*. 2022. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication (cit. on pp. 11, 12).

[10] Lydia Defranchi. *What are event-driven APIs? Understanding event-driven vs REST API interactions*. 2025. URL: https://blog.axway.com/learning-center/apis/basics/event-driven-vs-rest-api-interactions (cit. on p. 11).

[11] Red Hat. *Event-driven architecture: Understanding the essential benefits*. 2021. URL: https://www.redhat.com/en/blog/event-driven-architecture-essentials (cit. on pp. 11, 12).

[12] Rich Sharples / Red Hat. *Bringing Java into the Kubernetes-native future with Quarkus*. 2020. URL: https://www.redhat.com/en/blog/bringing-java-kubernetes-native-future-quarkus (cit. on pp. 13, 14).

[13] Spring.io. *Microservices with Spring*. 2025. URL: https://spring.io/microservices/ (cit. on p. 13).

[14] Eclipse Vert.x. *Vert.x*. URL: https://vertx.io (cit. on p. 14).

[15] Spring.io. *Spring Data*. 2025. URL: https://spring.io/projects/spring-data (cit. on p. 15).

[16] Quarkus Official Documentation. *Simplified MongoDB with Panache*. 2025. URL: https://quarkus.io/guides/mongodb-panache (cit. on p. 15).

[17] Hitesh Umaletiya / Brilworks. *Spring Boot vs Quarkus vs Micronaut Performance and Use Case Guide*. 2025. URL: https://www.brilworks.com/blog/spring-boot-vs-quarkus-vs-micronaut/ (cit. on p. 15).

[18] Adobe Experience Cloud Team. *Single-page applications (SPAs) — what they are and how they work*. 2023. URL: https://business.adobe.com/blog/basics/learn-the-benefits-of-single-page-apps-spa (cit. on p. 16).

[19] Angular Team. *What is Angular? - Official Documentation*. 2025. URL: https://angular.dev/overview (cit. on pp. 17, 18).

[20] Besant Technologies. *What is Angular?* URL: https://www.besanttechnologies.com/what-is-angular (cit. on p. 18).

[21] ReactiveX. *ReactiveX: The Observer pattern done right*. 2025. URL: https://reactivex.io/ (cit. on p. 18).

[22] Zoumana Keita. *NoSQL Databases: What Every Data Scientist Needs to Know*. 2023. URL: https://www.datacamp.com/blog/nosql-databases-what-every-data-scientist-needs-to-know (cit. on p. 19).

[23] Roshni Verma. *MongoDB Architecture: A Comprehensive Guide*. 2024. URL: https://medium.com/@roshniverma021/mongodb-architecture-a-comprehensive-guide-c72799347b22 (cit. on pp. 19, 20, 22).

[24] Victoria Malaya. *SQL vs. NoSQL*. 2013. URL: https://sql-vs-nosql.blogspot.com/2013/11/indexes-comparison-mongodb-vs-mssqlserver.html (cit. on p. 21).

[25] MongoDB Inc. *Aggregation Operations*. 2025. URL: https://www.mongodb.com/docs/manual/aggregation/ (cit. on p. 21).

[26] MongoDB Inc. *Transactions*. 2025. URL: https://www.mongodb.com/docs/manual/core/transactions/ (cit. on p. 22).

[27] Amazon Web Services (AWS). *What is a RESTful API?* 2025. URL: https://aws.amazon.com/what-is/restful-api/ (cit. on pp. 23, 24).

[28] IBM. *GraphQL vs. REST: What's the Difference?* 2023. URL: https://www.ibm.com/topics/graphql-vs-rest (cit. on pp. 24, 25).

[29] Apollo GraphQL. *Introduction to GraphQL*. 2024. URL: https://www.apollographql.com/docs/intro/ (cit. on pp. 25, 27).

[30] Amazon Web Services (AWS). *What is DevOps?* 2025. URL: https://aws.amazon.com/devops/what-is-devops/ (cit. on pp. 27, 28).

[31] Atlassian. *DevOps: A complete guide*. 2024. URL: https://www.atlassian.com/devops (cit. on p. 28).

[32] Red Hat. *What is CI/CD?* 2023. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd (cit. on pp. 28, 29).

[33] GitLab. *What is CI/CD? Continuous integration and continuous delivery explained*. 2025. URL: https://about.gitlab.com/topics/ci-cd/ (cit. on p. 29).

[34] Microsoft Learn. *Backends for Frontends pattern*. 2024. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends (cit. on p. 35).

[35] Sam Newman. *Pattern: Backends For Frontends*. 2015. URL: https://samnewman.io/patterns/architectural/bff/ (cit. on p. 35).

[36] Michael Szczepanik. *Backend for frontend (BFF) pattern — why do you need to know it?* 2021. URL: https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0 (cit. on p. 36).

[37] Baeldung. *Difference Between Flux and Mono*. 2025. URL: https://www.baeldung.com/java-reactor-flux-vs-mono (cit. on p. 44).

[38] Project Reactor Team. *Project Reactor: A Reactive Programming Library for Java*. URL: https://projectreactor.io/ (cit. on p. 45).

[39] Atlassian. *DevOps Culture*. 2024. URL: https://www.atlassian.com/devops/what-is-devops/devops-culture (cit. on pp. 65, 66).

[40] Amazon Web Services. *What is DevOps?* 2024. URL: `https://aws.amazon.com/devops/what-is-devops/` (cit. on pp. 65, 66).

[41] James Lewis and Martin Fowler. *Microservices: a definition of this new architectural term.* 2014. URL: `https://martinfowler.com/articles/microservices.html` (cit. on p. 66).