



INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Design and Evaluation of a Cloud-Native Web API for Third-Party Integrations

Rafael Bruce Tomé Santos

Master in Computer Engineering

Supervisors:

Doctor Ana Maria de Almeida, Associate Professor,
Iscte – Instituto Universitário de Lisboa

Doctor Carlos Coutinho, Assistant Professor,
Iscte – Instituto Universitário de Lisboa

September, 2025



TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

Design and Evaluation of a Cloud-Native Web API for Third-Party Integrations

Rafael Bruce Tomé Santos

Master in Computer Engineering

Supervisors:

Doctor Ana Maria de Almeida, Associate Professor,
Iscte – Instituto Universitário de Lisboa

Doctor Carlos Coutinho, Assistant Professor,
Iscte – Instituto Universitário de Lisboa

September, 2025

Acknowledgment

Write here the acknowledgments and grants, if any...

Resumo

Escrever aqui o resumo em Português.

PALAVRAS CHAVE: *palavra_1*, *palavra_2*, ...

Abstract

Write your abstract here in English.

KEYWORDS: *word_1*, *word_2*, ...

Contents

Acknowledgment	i
Resumo	ii
Abstract	iii
List of Figures	vii
List of Tables	ix
List of Acronyms	x
Chapter 1. Introduction	1
1.1. Context	1
1.2. Motivation	1
1.3. Research Questions	2
1.4. Document Structure	2
Chapter 2. Literature Review	3
2.1. Freight Transportation Overview	3
2.2. Web Application Programming Interfaces (APIs) Overview	4
2.3. Web API Security: Authentication and Authorization	5
2.4. Web API Frameworks	6
2.5. Monolithic vs. Microservices Architecture	8
2.5.1. Monolithic Architecture	8
2.5.2. Microservices Architecture	9
2.6. Cloud Computing	10
2.7. Containerization	12
Chapter 3. Research Methodology	13
3.1. Design Science Research Methodology (DSRM)	13
3.2. PRISMA-S	14
3.2.1. PRISMA-S Overview	14
3.2.2. Information Sources	14
3.2.3. Search Strategies	14
3.2.4. Search Results	15
3.3. Further Research	16
Chapter 4. Design and Architecture	17

4.1.	Requirements	17
4.2.	High-Level Architecture	19
4.2.1.	Client	20
4.2.2.	API Gateway	20
4.2.3.	Microservice Environment	20
4.2.4.	Virtual Network (VNet)	21
4.2.5.	Data Persistence Layer	21
4.2.6.	Identity Provider	21
4.2.7.	Message Broker	21
4.2.8.	Secret Storage	22
4.2.9.	WebCargo and Cargofive APIs	22
4.2.10.	Code Repository	22
4.2.11.	Container Registry	22
4.3.	Runtime Communication	23
4.3.1.	Client-Server Communication	23
4.3.2.	Identity Provider Communication	24
4.3.3.	Database Communication	25
4.3.4.	Direct Inter-Service Communication	26
4.3.5.	Event-Driven Inter-Service Communication	26
4.3.6.	External Carrier API Communication	29
4.4.	API Specification	30
4.5.	Internal Architecture of the Microservices	31
4.5.1.	Middleware	32
4.5.2.	Data Models	32
4.5.3.	Controllers	32
4.5.4.	Services	33
4.5.5.	Repositories	33
4.6.	Deployment	33
4.7.	Technology Stack	35
Chapter 5.	Implementation	36
5.1.	Development Environment	36
5.2.	Microservices Implementation	37
5.2.1.	Project Structure	37
5.2.2.	Freight Rates and Quotes Obtainment	39
5.2.3.	Freight Rates Cache	43
5.2.4.	Quote History	43
5.2.5.	Location Search	45
5.2.6.	Role-Based Access Control (RBAC)	46
5.2.7.	User Data Deletion	47
5.3.	Cloud Components Configuration	47

5.3.1.	API Gateway Configuration	47
5.3.2.	Microservice Environment Configuration	50
5.3.3.	Data Persistence Layer Configuration	51
5.3.4.	Message Broker Configuration	51
5.3.5.	Secret Storage Configuration	51
5.4.	Continuous Integration/Continuous Delivery (CI/CD) Pipeline Configuration	52
Chapter 6.	Evaluation	53
6.1.	Testing Strategy	53
6.2.	Performance Evaluation	53
6.3.	Cost Analysis	53
6.4.	Requirements Fulfillment	53
Chapter 7.	Conclusion	54
7.1.	Dissertation Summary	54
7.2.	Answering the Research Questions	54
7.3.	Academic Contributions	54
7.4.	Limitations	54
7.5.	Future Work	54
References		55

List of Figures

Figure 2.1	Freight Forwarding Stakeholders, adapted from [9]	3
Figure 2.2	Web API frameworks used by professional developers, adapted from [21]	6
Figure 2.3	Example of a Monolithic Architecture	8
Figure 2.4	Example of a Microservices Architecture	10
Figure 2.5	Cloud service models	11
Figure 3.1	Design Science Research Methodology [9]	13
Figure 3.2	Literature Search Flow Diagram	15
Figure 4.1	High-Level Architecture	19
Figure 4.2	Client-Server Communication	23
Figure 4.3	Identity Provider Communication	24
Figure 4.4	Database Communication	25
Figure 4.5	Direct Inter-Service Communication	26
Figure 4.6	Event-Driven Inter-Service Communication: User Deleted	27
Figure 4.7	Event-Driven Inter-Service Communication: Cargofive Synchronization	28
Figure 4.8	External Carrier API Communication	29
Figure 4.9	API Specification	30
Figure 4.10	Internal Architecture of a Microservice	31
Figure 4.11	Deployment to the Cloud Platform	34
Figure 5.1	Project Structure	37
Figure 5.2	Service Addresses (Development Environment)	39
Figure 5.3	Service Addresses (Production Environment)	39
Figure 5.4	Air Quote Request DTO	40
Figure 5.5	Air Rate Request DTO	40
Figure 5.6	Air Rate DTO	41
Figure 5.7	Air Quote Response DTO	41
Figure 5.8	Air Quote Request in Postman	42
Figure 5.9	Air Quote Response in Postman	42

Figure 5.10	Quote History Request in Postman	43
Figure 5.11	Quote History Response in Postman	44
Figure 5.12	Location Search Request in Postman	45
Figure 5.13	Location Search Response in Postman	45
Figure 5.14	User Search Request in Postman	46
Figure 5.15	User Search Response in Postman (Access Forbidden)	46
Figure 5.16	User Search Response in Postman (Success)	46
Figure 5.17	API Gateway Settings	48
Figure 5.18	API Gateway Backends	48
Figure 5.19	API Gateway Cross-Origin Resource Sharing (CORS) Policy	49
Figure 5.20	API Gateway Authentication Policy	49
Figure 5.21	API Gateway Routing Policy	49
Figure 5.22	Container Apps Environment Configuration	50
Figure 5.23	Container App Scaling Rules	51
Figure 5.24	Key Vault Secrets	52

List of Tables

Table 4.1	Azure Service of each Cloud Platform Component	35
-----------	--	----

List of Acronyms

API: Application Programming Interface

CI/CD: Continuous Integration/Continuous Delivery

CORS: Cross-Origin Resource Sharing

CPU: Central Processing Unit

CRUD: Create Read Update Delete

DNS: Domain Name System

DSRM: Design Science Research Methodology

DTO: Data Transfer Object

HTTP: Hypertext Transfer Protocol

HTTPS: Hypertext Transfer Protocol Secure

IATA: International Air Transport Association

IaaS: Infrastructure as a Service

IDE: Integrated Development Environment

IT: Information Technology

IS: Information System

JSON: JavaScript Object Notation

JWT: JSON Web Token

LCL: Less than Container Loaded

OAuth: Open Authorization

OIDC: OpenID Connect

ORM: Object Relational Mapping

OS: Operating System

PaaS: Platform as a Service

PRISMA: Preferred Reporting Items for Systematic reviews and Meta-Analyses

RBAC: Role-Based Access Control

REST: Representational State Transfer

RPC: Remote Procedure Call

SaaS: Software as a Service

SOA: Service-Oriented Architecture

SPA: Single-Page Application

UI: User Interface

URL: Uniform Resource Locator

VNet: Virtual Network

VM: Virtual Machine

XML: eXtensible Markup Language

CHAPTER 1

Introduction

1.1. Context

Digital transformation is changing the way businesses communicate with a network of partners and third-party service providers. This shift is powered by Application Programming Interfaces (APIs), which enable software systems to communicate with each other smoothly [1]. However, this increased connectivity fuels the need to aggregate and integrate heterogeneous external services into a unified platform. Companies seeking to build platforms must often interact with several third-party APIs that vary widely in their design, data formats, communication protocols, and reliability. This inconsistency creates complexity, making it more difficult to build systems that are scalable, maintainable, and resilient [2].

The freight forwarding industry serves as a case study for this integration challenge. Many companies in this industry are undergoing digitalization to meet growing client demands and remain competitive [3]. Traditional freight forwarders have been lagging behind due to an overreliance on manual processes. This gives rise to digital freight forwarders that provide more advanced and automated services, integrating several carrier providers, often across multiple modes of transport [4].

Devlop¹, an Information Systems (ISs) company from Portugal that specializes in developing solutions for the transport and logistics sectors, proposed creating a web application to provide real-time freight quotes from multiple carriers to customers. To this effect, two external APIs would be integrated: WebCargo² (for air transport) and Cargofive³ (for sea transport). A traditional monolithic software architecture is not suitable for this as the application grows, since an issue in one integration could hinder the entire system. This establishes the need for a more robust and scalable approach.

This dissertation proposes a conceptual framework to address these integration and scalability challenges, which may be present in other projects with similar characteristics in any industry. To do so, it uses a proof-of-concept implementation to detail and evaluate its architecture.

1.2. Motivation

The dissertation aims to address the architectural challenges faced when integrating various third-party APIs by investigating a cloud-native microservices approach. The application proposed by Devlop faces these difficulties. The development of effective and

¹Devlop, <https://devlop.systems/>, (accessed 16 Aug. 2025)

²WebCargo, <https://www.webcargo.co/about/>, (accessed 19 Aug. 2025)

³Cargofive, <https://cargofive.com/>, (accessed 19 Aug. 2025)

scalable software is crucial to the success of modern freight forwarding digital platforms, especially in a time when choosing the right technology to address all user requirements is becoming more difficult [5].

Even though the theoretical benefits of microservices and cloud computing are well-documented in scientific literature, there is a lack of practical case studies. The few studies that go beyond theory tend to focus on specific and isolated architectural features rather than presenting a complete framework for building an entire application. This work moves the discussion from theoretical advantages to a complete implementation, providing a concrete case study to the field of software architecture.

The author of this dissertation is motivated by the possibility to learn about modern enterprise-level software architecture and contribute to both the academic and enterprise communities.

[Este último parágrafo é apropriado?]

1.3. Research Questions

Based on the considerations presented in Section 1.2, this research presents a conceptual framework for developing cloud-native web APIs that integrate and aggregate external services. This study is guided by this research question:

RESEARCH QUESTION 1. What is the ideal software architecture for complex web applications that need to integrate several external APIs?

The research question leads the following hypothesis, which will be studied in this paper:

HYPOTHESIS 1. A microservices architecture, orchestrated through an API Gateway and deployed to a feature-rich cloud platform, is the most suitable choice for complex web applications integrating multiple external APIs.

[Defini uma hipótese pois era requisito na lista de tarefas no Asana. Faz sentido oferecer uma hipótese? Na introdução dou a entender que o objetivo da dissertação é propor uma framework conceptual para x. Provar a hipótese definida acima deve ser o objectivo fundamental da dissertação?]

1.4. Document Structure

This document is divided into ??? chapters:

CHAPTER 2

Literature Review

2.1. Freight Transportation Overview

Freight transportation is defined as the flow and storage of goods throughout a supply chain, from an origin to a destination. To achieve this, several stakeholders contribute to the process [6, 7]. In this project, since the application to develop will be administered by a freight forwarding company, the most important ones are the following:

Shipper: The person or company that sells goods to be sent from one location to another. They are the initiators of the whole process, generating the demand for freight transportation [6].

Carrier: The party that provides the transportation service of goods, such as a shipping or airline company [6].

Freight forwarder: The intermediary entity between the shipper and the carriers. It organizes and coordinates the shipment of goods on behalf of its customer seller (the shipper). They are usually responsible for selecting and contracting with appropriate carriers, container consolidation, shipment tracking, transshipment, negotiating delivery terms and freight rates, among other activities [6, 8].

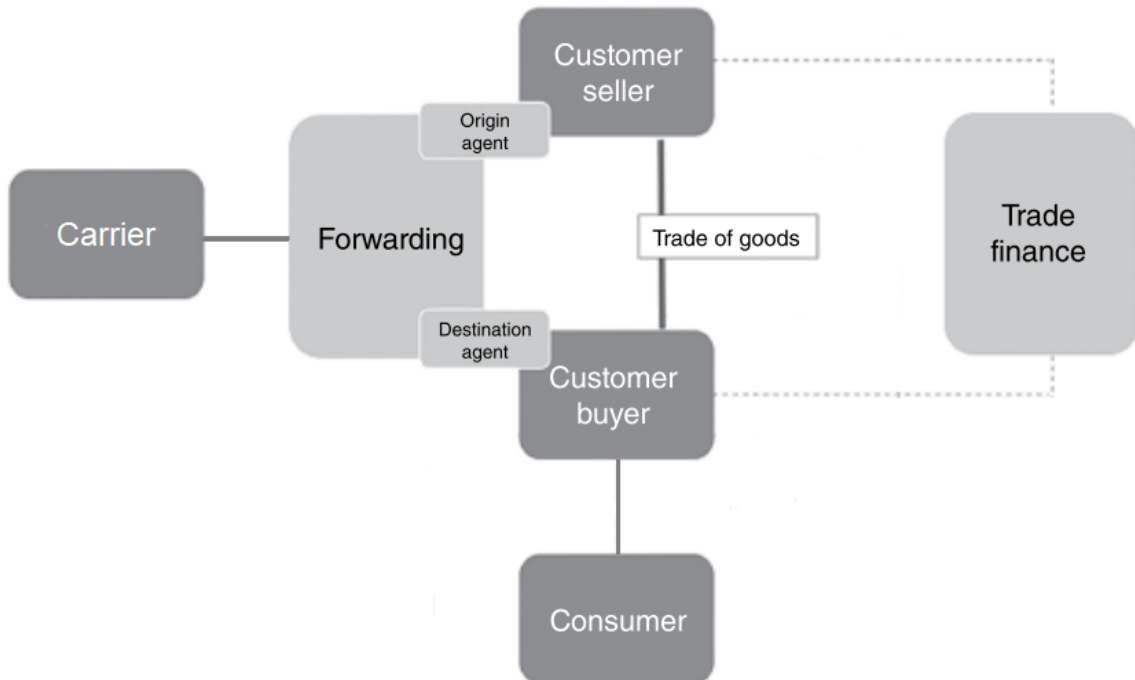


FIGURE 2.1. Freight Forwarding Stakeholders, adapted from [9]

Understanding freight rates is relevant for this project, since responding to quotation requests from shippers is one of its goals. The freight rate is the definite cost a carrier charges to transport a specific cargo from one location to another. This cost depends on factors like the cargo's weight, volume, mode of transport, and travel distance. A freight quote is a comprehensive estimate of the total cost to transport a particular cargo from an origin to a destination, which includes the freight rate and any additional fees [6, 10, 3, 11].

Various transport modes may be used, including air, sea, waterway, road, and rail. Multiple modes of transport may also be used throughout the journey, a practice known as multimodal transport. In 2019, 70% of the global freight (measured in tonne-kilometers) was done by ships, followed by road vehicles with 22%, trains with 7%, and air vehicles with less than 1% [12].

Although shippers have the option to work directly with carrier companies, many enterprises, especially small to medium-sized ones, resort to the services provided by freight forwarders. These enterprises tend to deliver Less than Container Loaded (LCL) cargo, which requires consolidation to achieve cost-efficient transportation. The expertise of freight forwarders and their wide network of carriers help shippers find the most appropriate offers and reduce freight rates, decreasing the total delivery price [8].

2.2. Web APIs Overview

An API is a set of rules and specifications that define how software components or applications should interact with each other. In this dissertation, web APIs are specifically defined as server-side (backend) web APIs, which act as an exposed interface for a web server. They allow client systems like web browsers and mobile applications to interact with the server's resources without needing to know its internal structure and implementation [13, 14, 15].

Essentially, a web API acts as an abstraction layer that facilitates communication through a request-response cycle. A client sends a request to a specific endpoint on a server, which contains details about the desired operation and any necessary data. The server processes the request and returns an Hypertext Transfer Protocol (HTTP) response. This response contains a status code indicating success or failure and, depending on the request, a body with the requested data, often formatted in JavaScript Object Notation (JSON) or eXtensible Markup Language (XML). This client-server relationship forms a complete web application that end-users can use [13, 14, 15].

Web APIs specifications come in different architectural styles, which define the syntax and structure of the client-server interactions. These include:

Representational State Transfer (REST): The most widely used architectural style for API development. It organizes its operations around resources, which may be any information that can be identified or named within the context of a web application, such as users, files, purchases, events, etc. REST APIs generally use standard HTTP methods to perform Create Read Update Delete (CRUD)

operations on these resources: "GET" to read, "POST" to create, "PUT" and "PATCH" to update, and "DELETE" to delete [14, 16, 17, 15].

GraphQL: This architecture allows clients to specify the exact structure of the data they require. Unlike REST, which often requires multiple requests to different endpoints to gather related data (under-fetching) or returns more data than needed (over-fetching), GraphQL exposes a single endpoint. This reduces the amount of data transferred over the network, at the cost of development complexity [14, 16].

Remote Procedure Call (RPC): RPC APIs are function-based, unlike REST and GraphQL APIs, which are centered on resources and data, respectively. Each endpoint these APIs expose are associated with a particular function on the server side. This architectural style can offer high performance with its deserialization method, as well as bidirectional and real-time streaming communication support. However, it can be harder for consumers of these APIs to find the desired endpoints, as it is less standardized [14, 16].

2.3. Web API Security: Authentication and Authorization

Web APIs expose potentially confidential operations and data, requiring robust security measures. These measures can be understood as two layers: authentication and authorization. Authentication is the process of identifying users and verifying if they are indeed who they claim to be. Authorization is processed after authentication and determines the actions that a specific user is allowed to perform [18, 19].

There are several security protocols and mechanisms to address authentication and authorization, including:

Open Authorization (OAuth) 2.0: The OAuth protocol is an authorization framework that allows users to access backend services without sharing their credentials. In a web application, when a user clicks the login link, the client directs the user to an authorization server where they authenticate. Once the user is authenticated, the authorization server redirects the user back to the client with an authorization code, which is exchanged for an access token. The client includes this token whenever it sends a request to the resource server. The server validates it and uses it to determine what the user is authorized to do [18, 19].

JSON Web Token (JWT): A JWT is a compact standard for securely transmitting information between parties as a JSON object. This information, known as claims, can be verified and trusted because it is digitally signed. JWTs can be used as the format for access tokens in OAuth 2.0 flows, and may contain additional information about the user [18, 19, 17].

OpenID Connect (OIDC): OIDC is an authentication layer built on top of the OAuth 2.0 framework that guarantees users have a single identity for multiple services or applications. While OAuth 2.0 provides delegated authorization, OIDC adds

authentication through ID tokens, allowing clients to verify the identity of the user based on the authentication performed by an authorization server [19].

API Keys: API keys are a simple authentication method where a unique string of characters, the API key, is assigned to a user or application. API keys do not distinguish between users, providing little flexibility of access control. [18].

Although other security mechanisms were found in literature, these were identified as the most relevant to the security requirements of modern and distributed web APIs [19]. API keys were mentioned despite their low granularity because they are the authentication method of the two external APIs mentioned in Section 1. All these mechanisms are significant to the architecture detailed in Chapter 4.

2.4. Web API Frameworks

Software frameworks are a set of reusable libraries and tools that create a structure for software development. By providing generic functionality, they reduce complexity and repetitiveness, enabling developers to write less code while achieving higher quality results [20].

In 2024, Stack Overflow conducted a survey to assess the popularity of several technologies and tools among software developers. In this survey, participants were asked "Which web frameworks and web technologies have you done extensive development work in over the past year, and which do you want to work in over the next year?". 38,132 professional developers responded. Figure 2.2 shows the results, including only the top seven backend frameworks [21].

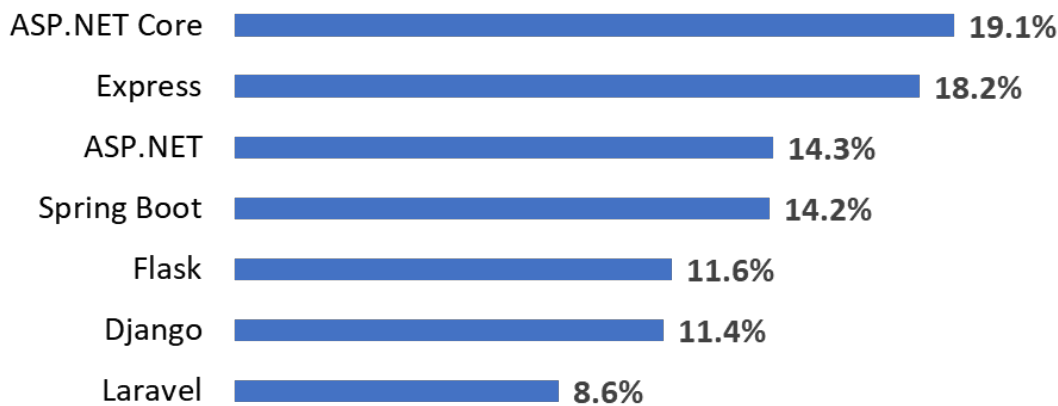


FIGURE 2.2. Web API frameworks used by professional developers, adapted from [21]

The web API frameworks present in Figure 2.2 are the following:

ASP.NET Core/ASP.NET: ASP.NET Core is a high-performance and feature-rich framework developed by Microsoft. It is the successor to ASP.NET. Compared to its predecessor, ASP.NET Core offers many additional features and capabilities like increased performance, cross-platform compatibility, and native support for client-side development [22, 23]. This framework uses the C# programming

language, demonstrating faster request processing compared to Django, Laravel, and frameworks based on Node.js. [24, 20, 23]. ASP.NET Core is suitable for computationally-intensive or enterprise-scale applications due to its high performance and scalability [25].

Express: A flexible and minimalist framework that uses the Node.js environment, allowing server-side JavaScript execution, which explains its high popularity [26]. Due to its event-driven single-threaded nature, it is more resource-efficient than Django and ASP.NET Core, for example [24, 26]. However, it is slower in processing CPU-intensive requests compared to ASP.NET Core and Spring Boot [24, 26, 27]. It also lacks some features natively present in other frameworks [28]. Still, Express is particularly effective for I/O-intensive applications, like chat and streaming applications [26, 29].

Spring Boot: An opinionated and feature-rich Java-based framework focused on ease of development. It reduces the need for configuration and generic code by providing starter dependencies, auto-configuration, default components, and more. It also offers real-time health status monitoring, metric reports, and traffic tracking. Spring Boot is especially suited for large-scale enterprise applications that use microservices [28, 29, 23].

Flask: A lightweight and minimalist Python microframework. It is quite flexible and facilitates quick development, requiring no specific tools or libraries. Its design supports projects that start small and can be extended easily and customized as needed. Consequently, Flask is best suited for building resource-efficient microservices and smaller applications that require a high level of control. [28, 29].

Django: A scalable and feature-rich Python framework. It has a comprehensive list of features, including an Object Relational Mapping (ORM) tool that simplifies database operations, and an automatically generated administrator interface [28, 29, 30]. One of its drawbacks is slower request processing compared to other frameworks like ASP.NET Core and Spring Boot [24, 27]. Despite that, it is suitable for both small-scale and large-scale applications [28, 29].

Laravel: A feature-rich framework built on the PHP programming language. Its toolset includes an ORM database tool, versatile database migrations, and built-in task queuing and scheduling. This framework prioritizes concise and well-organized code. Laravel is suitable for both small projects and large-scale applications [28, 20, 31].

2.5. Monolithic vs. Microservices Architecture

The architectural design of a web API is a decision that significantly impacts its development, deployment, scalability, and maintenance. When developing web APIs, two dominant patterns emerge: the traditional monolithic approach and the modern microservices approach. Several papers highlight their benefits and drawbacks [2, 32, 33, 34].

2.5.1. Monolithic Architecture

A monolithic architecture is a traditional model for software development in which the entire or most of the application is built as a single, tightly coupled unit. For a web API, this means that all endpoints and their underlying logic are present within a single codebase and deployed as one unit [32, 33, 34, 35, 36, 37].

The main advantages of this approach are due to its basic nature. It offers simplicity of development, especially at the beginning of a project, because there is no burden from managing a distributed system. This centralized structure can also lead to improved performance, since calls between different components are direct function calls rather than network requests. A monolith is also easier to test, as requests tend to undergo fewer jumps between software components [32, 33, 34, 35, 36, 23].

However, the monolithic model can be resource-inefficient since it is impossible to scale individual components independently. This architecture also imposes one particular technological framework on the entire application, reducing flexibility. In addition, because even a small change or issue can require the entire application to be recompiled and redeployed, it lowers resilience and deployment speed. All these issues only worsen as the application grows, leading to decreased scalability and maintainability, as well as possibly negating some of its advantages [32, 33, 34, 35, 36, 38].

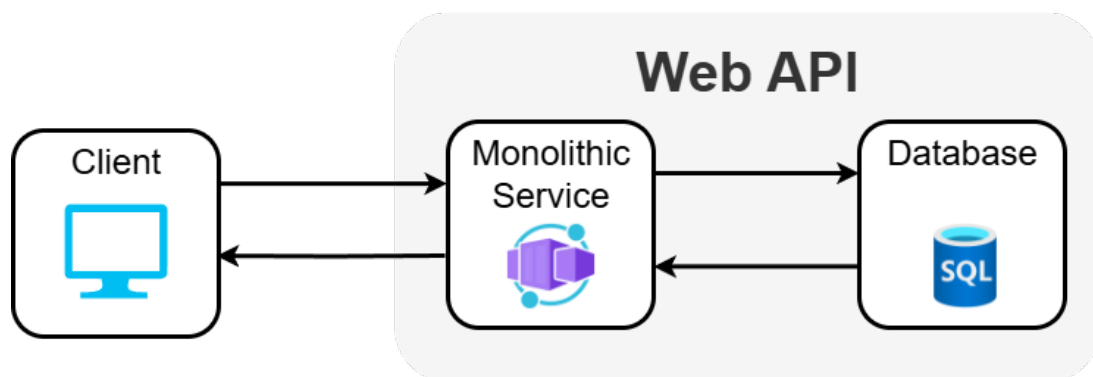


FIGURE 2.3. Example of a Monolithic Architecture

2.5.2. Microservices Architecture

A microservices architecture follows a Service-Oriented Architecture (SOA), structuring an application as a collection of small and loosely coupled services. Each service is self-contained, responsible for a specific business goal, and may communicate with other services [32, 33, 34, 39, 35, 36].

It offers superior scalability, as each service can be scaled independently based on its specific load, which is a highly efficient model for cloud-based deployments. This decoupling also allows a heterogeneous technology stack, for example, one service may use the ASP.NET Core framework while another uses the Spring Boot framework. Furthermore, this type of architecture is more resilient, as the failure of a single service does not necessarily compromise the entire application. Additionally, this architecture improves maintainability by breaking a large application down into smaller and more understandable codebases. Developers can work on individual services, allowing them to develop, deploy, and update their components autonomously [32, 33, 34, 39, 35, 36, 38].

However, the microservices pattern introduces some challenges. One drawback is the increased implementation effort, since developers must implement mechanisms for inter-service communication. This leads directly to communication and network complexity. Instead of simple function calls, services must communicate over a network, introducing considerations of latency, security, and fault tolerance [32, 33, 39, 35, 36, 38, 37].

To address some challenges inherent to the microservice architecture, the API Gateway pattern is often recommended. In this pattern, an API Gateway acts as an intermediary for all client requests, simplifying the client-side logic. Instead of clients needing to know the addresses of and communicate with multiple services, they make requests to the gateway only. The gateway then routes requests to the appropriate microservice, and can also aggregate responses from multiple services into a single response. It may handle concerns such as authentication and rate limiting as well [32, 33, 39, 36].

Regarding data management, multiple approaches may be applied when dealing with microservices, ranging from a single shared database for all services, to one database per service. Sharing a single database is simpler and facilitates data integrity, but is less scalable and flexible. In contrast, using one database per service offers greater scalability and flexibility, but increases complexity and adds the challenge of maintaining data integrity between services [32, 39, 36].

Figure 2.4 shows an example of a microservice architecture adhering to the API Gateway and database-per-service patterns.

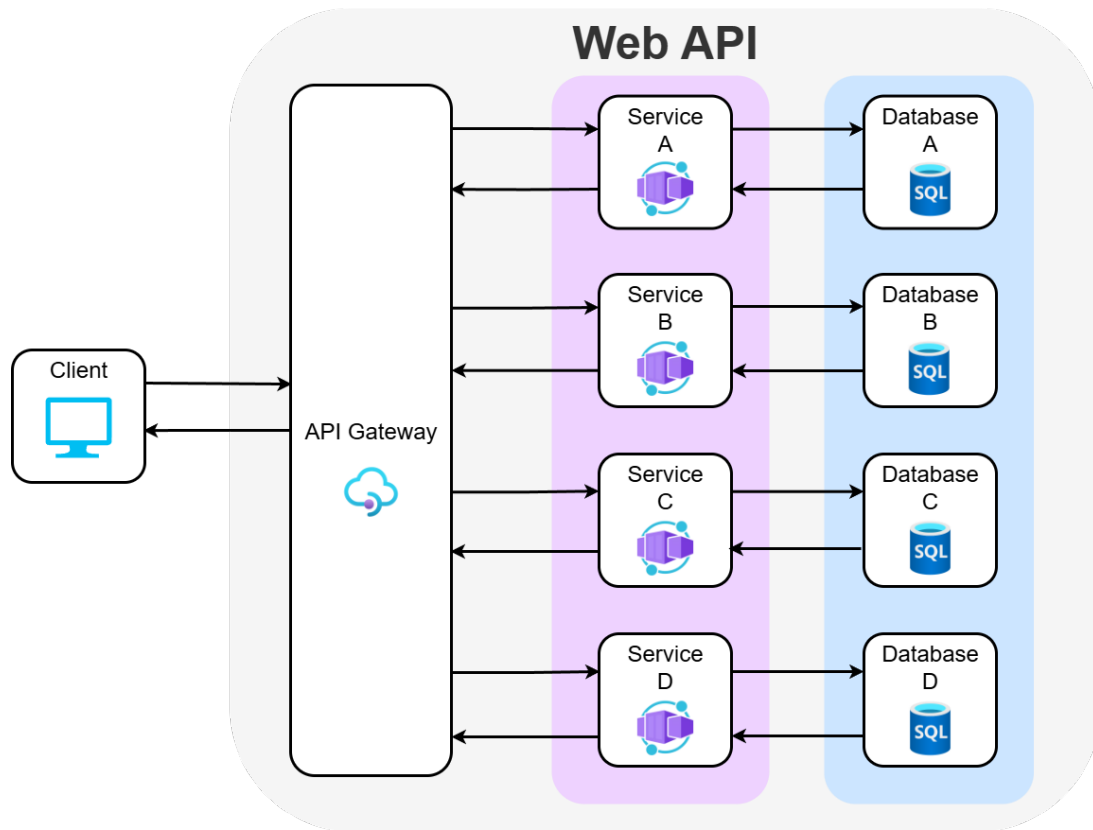


FIGURE 2.4. Example of a Microservices Architecture

2.6. Cloud Computing

Cloud computing is the provision of on-demand Information Technology (IT) services to customers over the Internet. These services may consist of software applications, databases, analytics, servers, and more. The resources are hosted on a remote server (the "cloud") located in a provider's data center, and accessible through a web browser [35, 40, 41, 42].

Cloud services can be divided into three distinct models:

Infrastructure as a Service (IaaS): Provides processing power, storage, and network capabilities through a Virtual Machine (VM). The customer is responsible for the Operating System (OS), runtimes, applications, and data. This model is quite flexible and is suitable for companies that want to improve the reliability and scalability of their infrastructure [35, 40, 43, 44].

Platform as a Service (PaaS): Offers an environment to build, deploy, and run applications without managing the associated hardware and software components. The customer can focus on developing applications and handling data, making this model useful for software engineers and developers [35, 40, 43, 44].

Software as a Service (SaaS): Provides a ready-to-use application fully managed by the cloud provider. This model is designed for end users of the application and organizations that wish to offer it to their employees [35, 40, 43, 44].

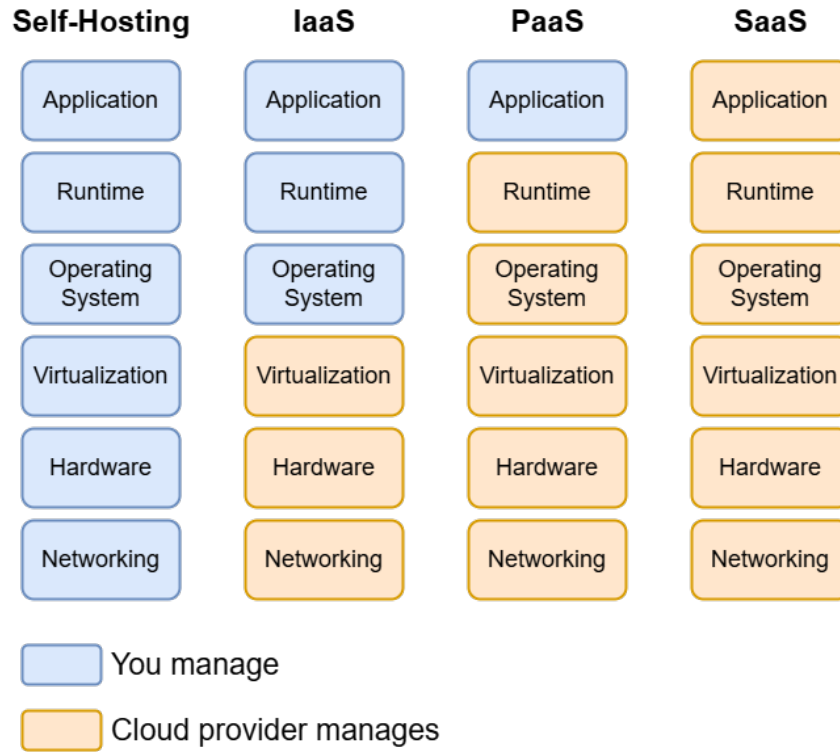


FIGURE 2.5. Cloud service models

Cloud computing has several advantages over self-hosting. It prevents expensive asset acquisitions and reduces maintenance costs because resources are only paid for when they are actually needed. Furthermore, it allows companies to scale their services as required, since cloud resources can be purchased in practically any quantity and readjusted fast. Cloud providers also have fallback data centers they can resort to in case of failure without affecting the user experience, improving reliability. Finally, by providing on-demand infrastructure, cloud computing speeds up the development and testing of new solutions while decreasing financial risk, promoting innovation [35, 40, 45, 41, 42].

However, cloud computing also has some disadvantages. Choosing a cloud-based service means that a third-party organization, possibly a foreign one, now manages sensitive data. This raises security issues, as well as legal considerations related to data protection regulations. Another drawback is low interoperability; once a company chooses a particular cloud provider, changing to another or working with multiple is difficult. In addition, the limited flexibility and customization of cloud computing can inhibit integration with certain systems [40, 45].

Overall, cloud computing is changing the way businesses deliver their software solutions. It is democratizing the impact of IT by giving companies of any size instant access to high-quality computing infrastructure. With this technology, software developers can develop and deploy web applications faster while reducing costs [35, 44, 41].

2.7. Containerization

Containerization is an approach to deploying software where an application, along with all its necessary dependencies (libraries, frameworks, configuration files, etc.), is packaged into a single unit called a container. Containerization is a virtualization technology that is popular in cloud and microservice environments due to its beneficial characteristics [35, 46, 47].

Containers are portable and self-sufficient packages, which ensures they run consistently across different environments. This consistency reduces deployment issues and makes it easier to set up new environments. Additionally, unlike regular VMs, containers virtualize their OS, meaning they only contain the application itself, with no OS overhead. Because of this, containers use less memory and Central Processing Unit (CPU) power, making them more efficient and scalable than VMs [35, 46, 47].

Docker is the leading container platform. It automates and standardizes the deployment of applications inside software containers by providing an additional layer of abstraction to the containerization process. To create a container, Docker uses a Dockerfile, which is a text-based script of instructions used to create a container image. This immutable image is a lightweight, standalone, executable package that contains everything needed to run an application. When the image is run, it becomes a container, which is an active instance of the image [35, 46, 47].

CHAPTER 3

Research Methodology

3.1. Design Science Research Methodology (DSRM)

This dissertation follows the DSRM, a six-step research approach often used to create IS-related artifacts such as models, methods, design theories, or, in this context, an instantiation [9].

This dissertation adheres to the "Design & Development Centered Initiation" entry point shown in Figure 3.1, as the first and second steps have already been presented in Chapter 1.

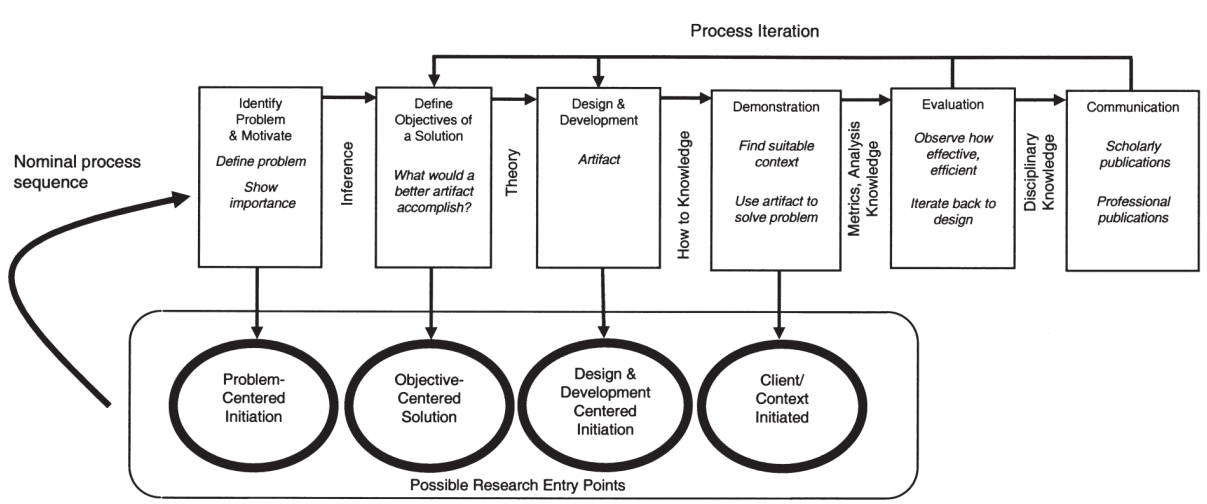


FIGURE 3.1. Design Science Research Methodology [9]

The following list briefly explains what each step entails and where it is located in this paper:

Problem Identification and Motivation: Section 1 highlights that despite APIs being crucial for business connectivity, integrating multiple external APIs creates significant challenges for scalability and reliability. A freight forwarding industry project is introduced as a case study. The motivation expressed in Section 1.2 is to offer a solution to these problems and contribute with a complete case study to the scientific literature.

Definition of Solution Objectives: The goals of the solution proposed in this dissertation are introduced in Section 1.2. In Section 4.1, all requirements are precisely listed and defined, constituting the solution objectives.

Design and Development: Chapter 4 details the architecture of the entire solution, while Chapter 5 reveals important parts of the implementation process.

Demonstration: Chapter 5 demonstrates a working artifact deployed on a remote cloud.

Evaluation: Chapter 6 evaluates the developed instantiation considering its reliability, performance, relative cost, and efficacy.

Communication: This dissertation itself serves as a mean of communication, intended for both the academic and business audiences. Chapter 7 summarizes the most relevant findings.

3.2. PRISMA-S

3.2.1. PRISMA-S Overview

Because DSRM does not define a method to search scientific literature, PRISMA-S was used as the basis for the literature search. Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA) is a complete set of guidelines used to improve the reporting of systematic literature reviews, promoting rigor. PRISMA-S (or PRISMA-Search) is a reporting guideline that extends PRISMA, contributing with a detailed checklist specifically for reporting literature search methods.

3.2.2. Information Sources

The B-on¹ platform was used for the PRISMA-S literature search. B-on is a library that aggregates multiple databases², including IEEE, Web of Science, and Elsevier. This agglomeration removes the need to search individual databases separately.

3.2.3. Search Strategies

The search was performed using the exact query shown below, where "TI" and "AB" stand for "title" and "abstract", respectively:

```
(TI ("web API*" OR "backend*" OR "web application*")  
OR AB ("web API*" OR "backend*" OR "web application*"))  
AND TI (Architecture OR Develop* OR Design)  
AND TI (Cloud* OR Microservice*)
```

The following search filters were applied iteratively:

- (1) The document must be from 2015 or later.
- (2) The document must be peer-reviewed.
- (3) The document must be written in English or Portuguese.

¹B-on, <https://www.b-on.pt/en/>, (accessed 16 Aug. 2025)

²B-on, "Collections", <https://www.b-on.pt/en/collections/>, (accessed 16 Aug. 2025)

3.2.4. Search Results

Once the search query was run and all aforementioned filters were applied, all filtered documents went through a process to determine whether they would be included in the final review. That process is shown in Figure 3.2.

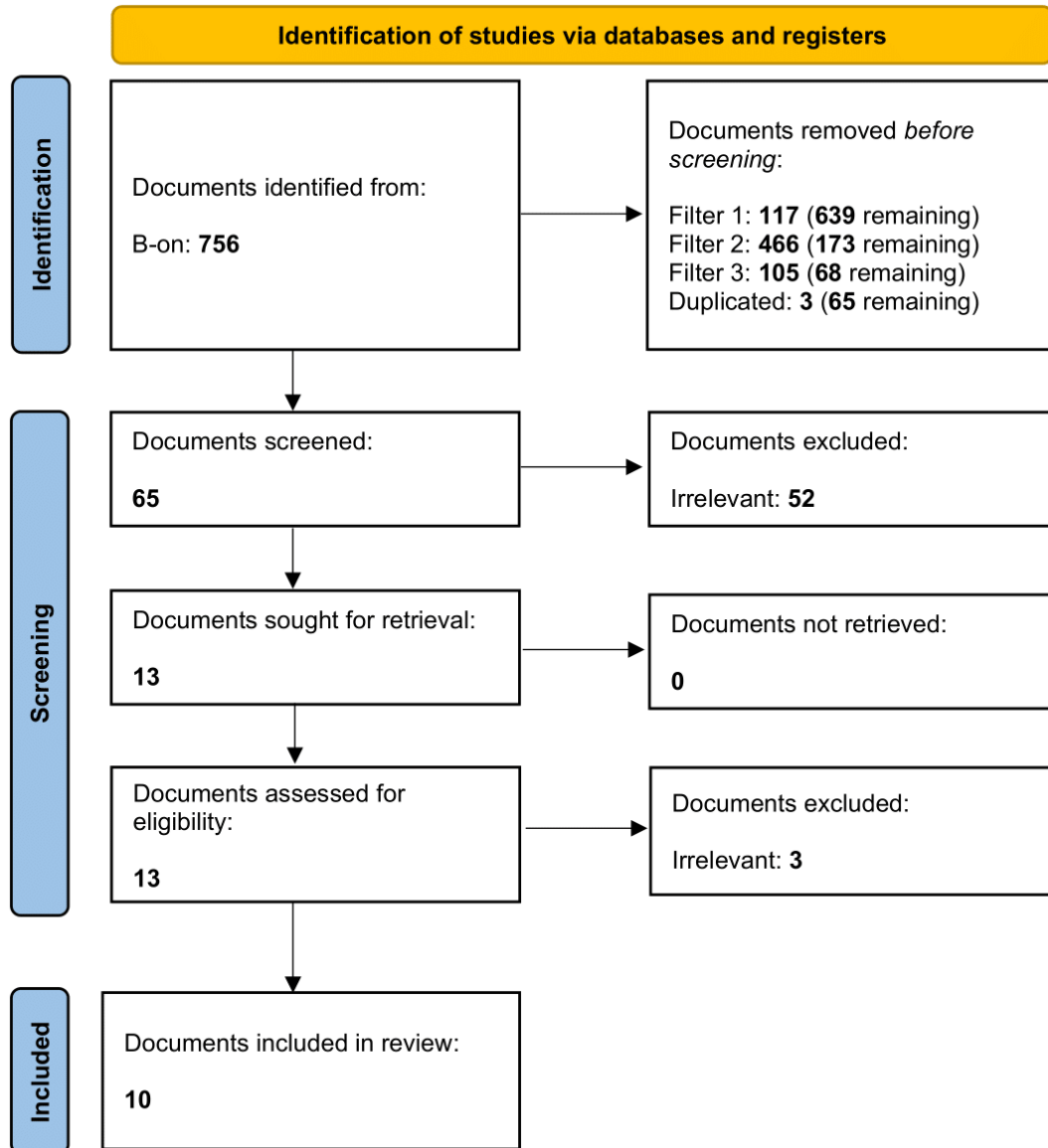


FIGURE 3.2. Literature Search Flow Diagram

The search query identified 756 documents. Upon filtering and deduplication, 65 documents remained. These documents went through a screening process that involved reading their titles and abstracts to determine their relevance. After screening, 13 documents were left. The eligibility of these documents was then evaluated by reading their full text to verify their relevance more rigorously. Only 10 documents were included in the final review, some of which did not present a complete architectural approach for a web APIs or web application. This outcome matches the lack of literature on this topic mentioned in 1.2.

3.3. Further Research

The 10 resulting documents gathered in Subsection 3.2.4 did not offer a comprehensive and varied view of web applications or web APIs. Therefore, additional literature searches were conducted based on topics present in those 10 documents, namely Monolithic and Microservice Architectures, Cloud Computing, Authentication and Authorization, Containerization, and Web API Frameworks. Furthermore, a literature search on freight forwarding was done to better understand the business domain of the web API that will be developed. To perform this further research, B-on, Google Scholar, and websites from authoritative sources were accessed.

CHAPTER 4

Design and Architecture

4.1. Requirements

The design and architecture of the web API are guided by a specific list of requirements. These include functional requirements, which establish the system's features, and non-functional requirements, which determine the attributes of the system and how it should operate. Every requirement is listed below:

REQUIREMENT 1. Provide freight quotes to users by querying external carrier APIs (WebCargo for air, Cargofive for sea) for freight rates. To achieve this, the system must expose an endpoint that accepts a payload containing the origin, destination, and other parameters, depending on the transport mode.

REQUIREMENT 2. Provide users with their quote search history via a GET request, which contains both the requests they made and the responses they received from the web API. The results must support pagination.

REQUIREMENT 3. Allow only users with the administrator role to access information about any user of the web API.

REQUIREMENT 4. Expose endpoints for querying supported airports and seaports. The query must have pagination and search term options.

REQUIREMENT 5. When a user deletes their account, also delete all associated data from the databases.

REQUIREMENT 6. Restrict the functionalities described in Requirements 1, 2, and 3 to authenticated users only.

REQUIREMENT 7. In case of a complete failure in one external carrier API integration, the rest of the system must remain operational.

REQUIREMENT 8. Support decoupled and automatic scaling of the web APIs's request processing capabilities based on real-time use of specific endpoints.

REQUIREMENT 9. Freight rates from external carrier APIs must be cached for 1 day to avoid reaching rate limits.

REQUIREMENT 10. Under a sustained load of 500 requests per second, the 95th percentile response time for GET requests that do not require external API calls must be less than 200 milliseconds.

REQUIREMENT 11. Enforce a single public entry point to access the web API's resources.

REQUIREMENT 12. Authentication must be done securely via the OAuth 2.0 and OIDC protocols, without storing user credentials within the system's databases.

Requirements 1 to 5 are functional, whereas Requirements 6 to 12 are non-functional. Requirement 1 was explicitly specified by the industry partner Devlop, representing the web API's main functionality. To expand its scope, others were added as reasonable requirements for a real-world, enterprise web API.

4.2. High-Level Architecture

The high-level architecture focuses on the main components of the system and concisely describes their roles. For the remainder of this document, the names of all major components will be capitalized. Figure 4.1 shows the high-level architecture of the web API.

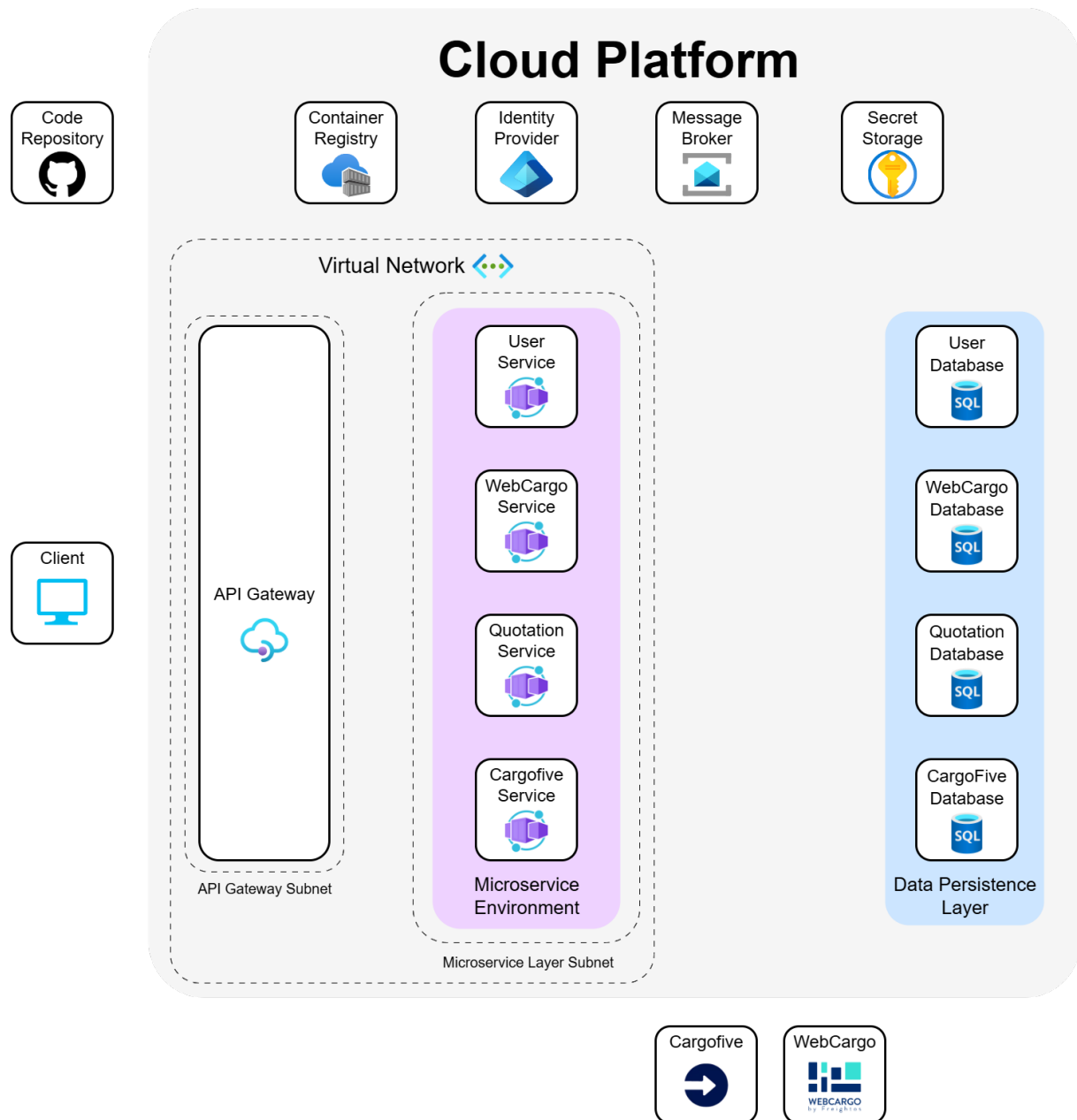


FIGURE 4.1. High-Level Architecture

4.2.1. Client

The Client is the user-facing application that serves as the primary interface to the system. It is a decoupled frontend from another project, implemented as a Single-Page Application (SPA). Its main responsibilities are to render the User Interface (UI), manage user sessions, and communicate with the API Gateway for data and functionality. For user authentication and account management, the Client directs users to a dedicated page on the Identity Provider, where they can log in, log out, delete their account, etc.

4.2.2. API Gateway

The API Gateway functions as the single entry point for all incoming Client requests, similarly to the pattern described in Subsection 2.5.2. This component routes requests to the appropriate microservice, allowing the Client to interact with the system without needing to know what microservices it contains. The API Gateway also enforces authentication by validating access tokens (JWTs).

4.2.3. Microservice Environment

The Microservice Environment holds the core of the system's business logic, implemented following a microservices architectural style described in Subsection 2.5.2. The microservices in this environment are designed around specific business goals, communicating with each other directly through endpoints exposed between them, or indirectly through the Message Broker component. In this web API, each microservice enforces its own authorization policies, restricting access to resources based on roles or more complex business rules. In addition, every microservice runs in a Docker container. There are four microservices in this environment, which are considered major components of the system:

User Service: This service provides endpoints to retrieve user-related information. To do so, it communicates with the Identity Provider. It also communicates with the Identity Provider to be informed about user deletions, which it propagates to the rest of the system through the Message Broker. Even though User Service does not store user data, it still needs an associated database to store data related to the Identity Provider. The purpose of this data will be explained in Chapter 5.

Quotation Service: This service orchestrates the core freight quotation business logic. It communicates with WebCargo Service and Cargofive Service to obtain freight rates and uses them to calculate and store freight quotes based on user input received through the API Gateway. Users can fill a freight quote request and obtain a response with a freight quote, as well as retrieve their quote history. Quotation Service also stores data relevant to the general quotation process, including locations (airports and seaports), special handling codes, and cargo container types. This general data is anonymously accessible through endpoints.

WebCargo Service: This service is responsible for all business logic related to the WebCargo external API, which is used to obtain up-to-date air freight rates. It extracts all relevant information present in the freight rates retrieved from the

WebCargo API and stores them in the corresponding database. Most importantly, it provides air freight rates to Quotation Service when requested.

Cargofive Service: Analogously to the WebCargo Service, this service is responsible for all business logic related to the Cargofive external API, which is used to obtain up-to-date sea freight rates. It extracts all relevant information present in the freight rates retrieved from the Cargofive API and stores them in the corresponding database. Most importantly, it provides sea freight rates to Quotation Service when requested.

4.2.4. Virtual Network (VNet)

The VNet provides network isolation for the system's components in the cloud platform. The Microservice Environment and the API Gateway reside in the VNet, each in its dedicated subnet. The main objective of these dedicated subnets is to implement network segmentation, enforcing network policies that apply to all inbound and outbound traffic. Specifically, the API Gateway subnet is configured to allow inbound traffic from the Internet, while the Microservice Environment subnet only accepts traffic from within the VNet. Furthermore, components within the VNet can securely access other components (the Message Broker, the Azure Key Vault, and the Data Persistence Layer) via private endpoints, which integrate them directly into the VNet's address space.

4.2.5. Data Persistence Layer

The Data Persistence Layer manages the persistent storage of the web API's data. This layer may consist of multiple database technologies, as each microservice owns its data and communicates with its corresponding database. Access to this layer is secured through a private endpoint within the VNet.

4.2.6. Identity Provider

The Identity Provider is an external service responsible for all user authentication and account management. It implements the OAuth 2.0 and OIDC protocols to authenticate users and issue JWTs to the Client. By delegating authentication and account management to an instance of a specialized Identity Provider, the system avoids storing user credentials and benefits from enterprise-level security features, such as multifactor authentication.

4.2.7. Message Broker

The Message Broker facilitates asynchronous communication between microservices and employs a publish-subscribe model. This component is useful when one service must send a notification or delegate a task but does not need to know the exact recipients nor receive a direct response. The Message Broker also implements mechanisms for message durability and delivery guarantees in case a microservice subscribes to a topic after messages have already been published. In this web API, the Message Broker is utilized when User Service needs to notify the system that a user has been deleted. In addition, it is used

when Cargofive Service needs to synchronize data with Quotation Service. The Message Broker is accessed through a private endpoint within the VNet.

4.2.8. Secret Storage

The Secret Storage is a centralized and secure repository for all application secrets. This contains API keys for external carrier APIs (WebCargo and Cargofive) and the secret value to access the Identity Provider instance as an administrator. The microservices access the Secret Storage at runtime by a private endpoint within the VNet.

4.2.9. WebCargo and Cargofive APIs

The WebCargo and Cargofive APIs are third-party dependencies that provide essential business functionality not built in the system. The system integrates with these APIs to retrieve up-to-date freight rates from many carriers, as explained in Subsection 4.2.3. WebCargo Service and Cargofive Service communicate with the WebCargo and Cargofive APIs respectively, using the API keys supplied by Devlop for authentication. Each of these external APIs has its own documentation and usage policies. Furthermore, the WebCargo API returns responses in XML format, while the Cargofive API uses JSON.

4.2.10. Code Repository

The Code Repository is the version control system that stores the source code of all microservices and Continuous Integration/Continuous Delivery (CI/CD) pipeline workflows. These workflows automate the build and deployment process and are triggered when the source code is updated. These workflows access the Container Registry and the Microservice Environment components through privately stored secrets.

4.2.11. Container Registry

The Container Registry is a private repository for storing and managing the versioned container images for each microservice. During the CI/CD process, new Docker images are built and pushed to the registry. Subsequently, these images are pulled from the registry to be run as Docker containers, where each container is an instance of a particular microservice within the Microservice Environment.

4.3. Runtime Communication

The architecture presented in Section 4.2 defines a set of components. These components must communicate with each other to achieve their objectives. This Section explains the web API's runtime communication, which refers to the exchange of data between components that typically occurs while the microservices are running.

4.3.1. Client-Server Communication

The client-server communication includes all requests sent by the Client to a server-side component: either the API Gateway or the Identity Provider. It also includes the API Gateway routing to the correct microservice, since successful Client requests are ultimately handled in the Microservice Environment. Figure 4.2 illustrates these interactions.

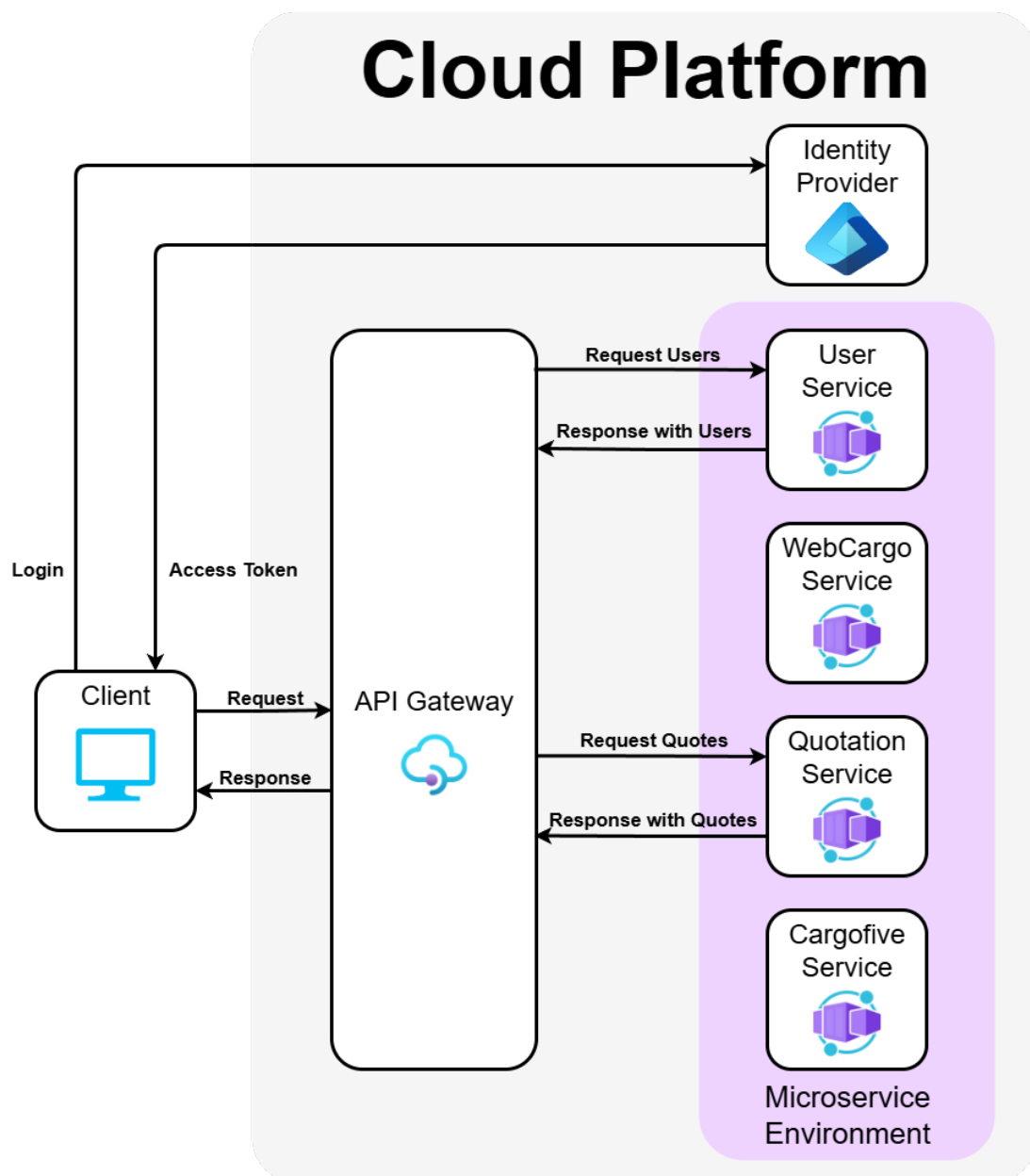


FIGURE 4.2. Client-Server Communication

The Client initiates authentication by redirecting users to the Identity Provider's login page. After a successful login, the Identity Provider redirects the user back to the Client with an authorization code. The Client then exchanges this code for an access token (JWT) by making a request to the Identity Provider's token endpoint, completing the authentication process.

The Client sends requests to the API Gateway to access the web API's resources. If the particular resource requires authentication, the API Gateway first validates the access token included in the request's authorization header. Then, if the token is valid or no authentication is needed, the API Gateway routes the request to the appropriate microservice in the Microservice Environment. The microservice processes the request and sends a response back to the API Gateway, which forwards it to the Client.

4.3.2. Identity Provider Communication

User Service exposes endpoints that allow the Client to get information about users. Since the Identity Provider is responsible for storing user accounts, User Service communicates with it to retrieve user data. This communication is shown in Figure 4.3. Additionally, User Service subscribes to user account deletion notifications from the Identity Provider.

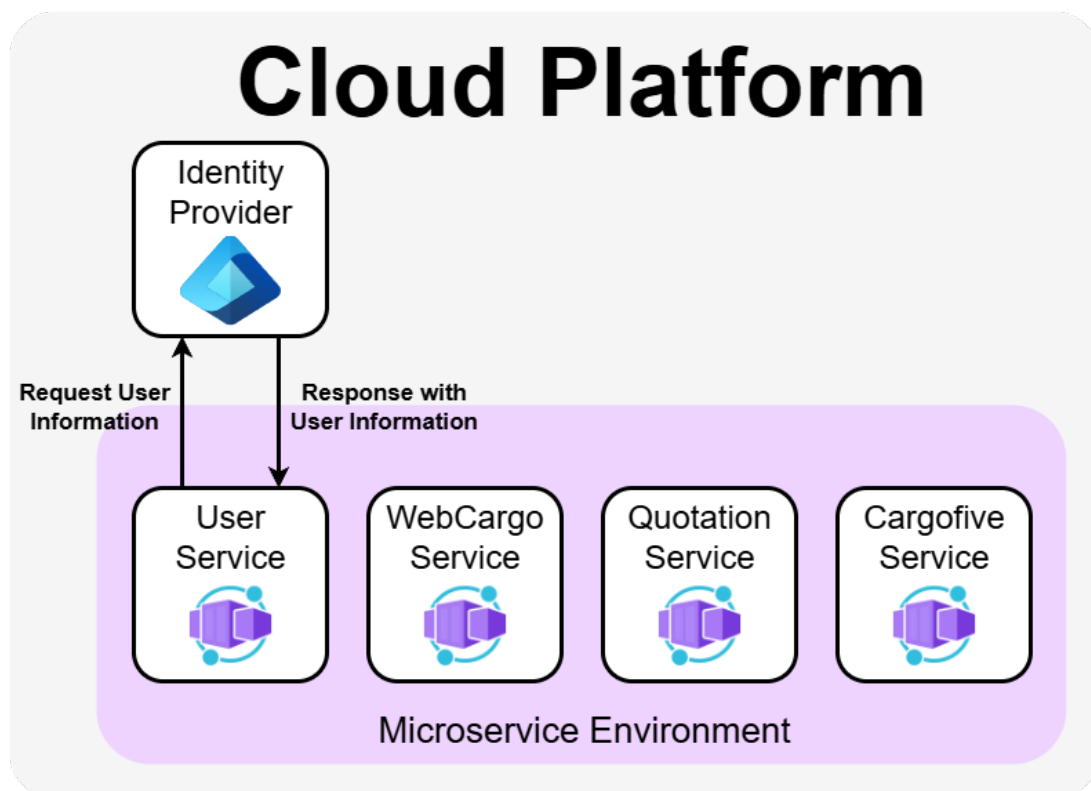


FIGURE 4.3. Identity Provider Communication

4.3.3. Database Communication

Each microservice establishes a connection to its correspondent database and performs the necessary queries to retrieve or modify data. Access to the Data Persistence Layer is secured through a private endpoint within the VNet. Figure 4.4 illustrates the communication between microservices and their databases.

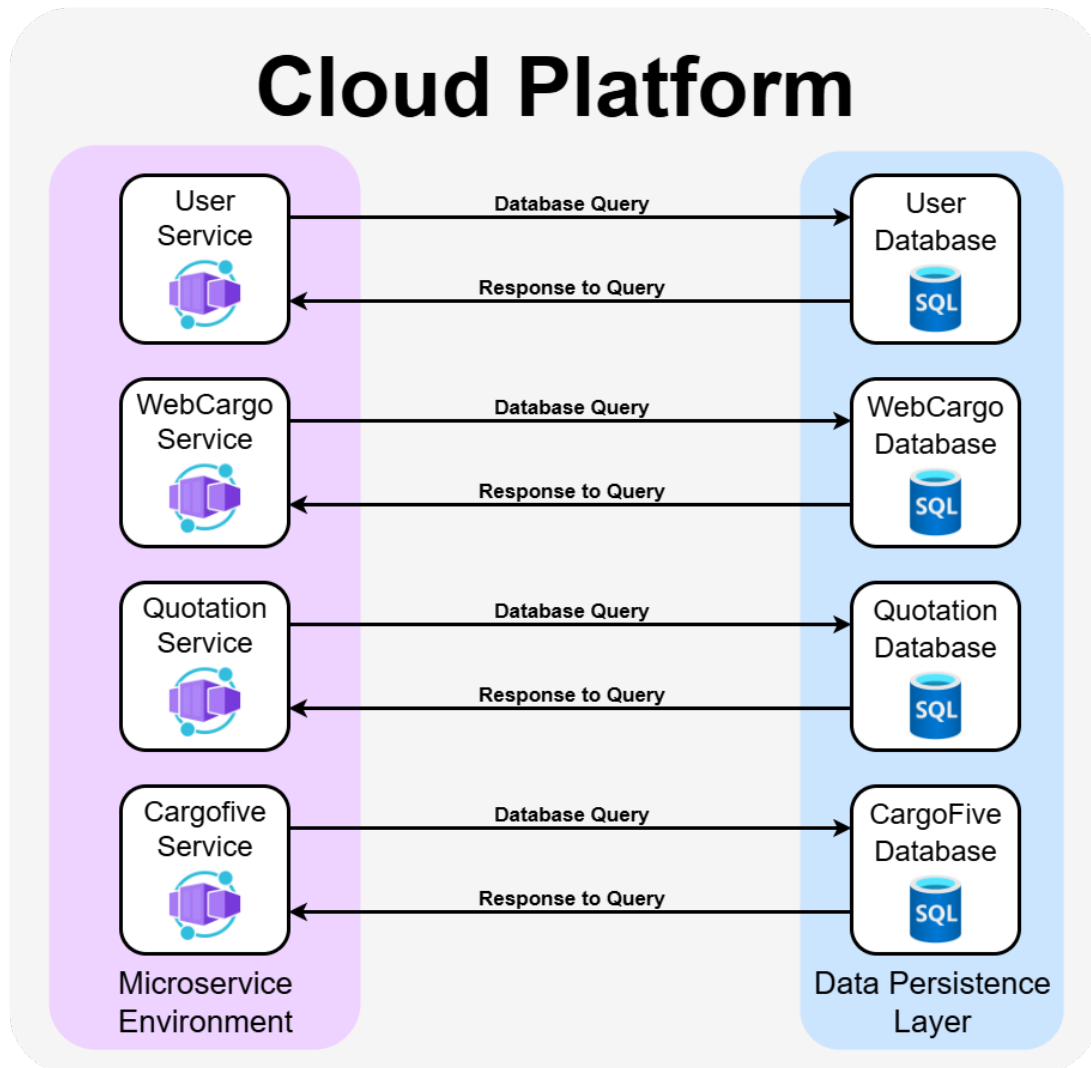


FIGURE 4.4. Database Communication

4.3.4. Direct Inter-Service Communication

Microservices can communicate with each other directly by making HTTP requests to each other's endpoints. Some of these endpoints may only be reachable to other services, not the Client. This allows for real-time data exchange and coordination between services.

The web API makes use of this type of communication when Quotation Service needs to obtain freight rates from WebCargo Service or Cargofive Service, as shown in Figure 4.5.

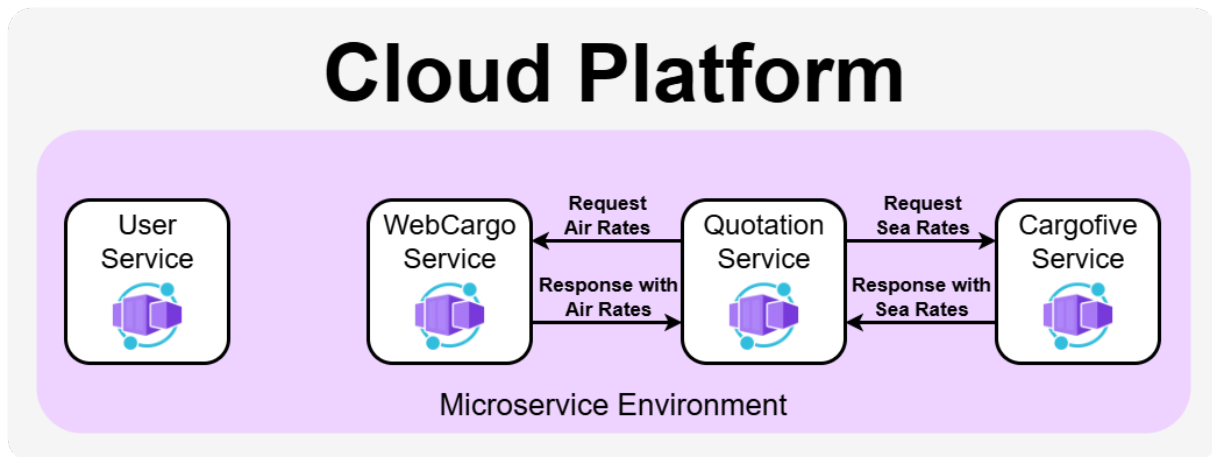


FIGURE 4.5. Direct Inter-Service Communication

4.3.5. Event-Driven Inter-Service Communication

Event-driven communication is applied in two scenarios using the Message Broker, as explained in Subsection 4.2.7.

The first scenario occurs when a user deletes their account through the Identity Provider. Firstly, User Service must be informed about the deletion to propagate it to the rest of the system. To achieve this, it can either poll the Identity Provider periodically or subscribe to its user deletion notifications. Once User Service is informed, it publishes a "UserDeleted" event to the Message Broker containing the identifier of the deleted user. Finally, Quotation Service receives this event and deletes the quotation search history of the deleted user from its database. The user deletion event process is shown in Figure 4.6.

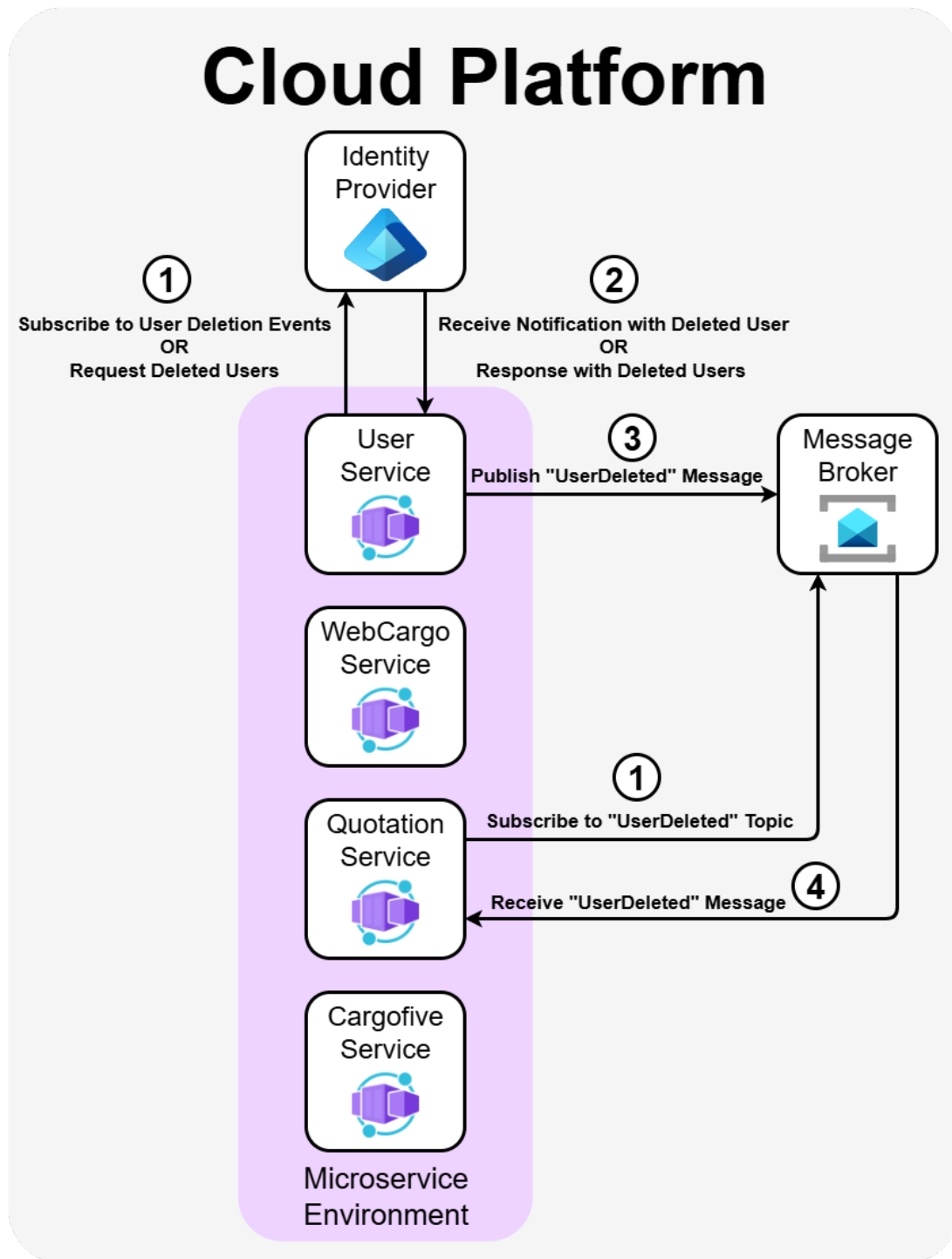


FIGURE 4.6. Event-Driven Inter-Service Communication: User Deleted

The second scenario occurs when Cargofive Service needs to synchronize data with Quotation Service. This synchronization is necessary because the Cargofive API uses its own identifiers for seaports, unlike the WebCargo API, which can identify airports through standardized codes. Cargofive Service fetches the list of supported seaports from the Cargofive API weekly. Cargofive Service publishes a "SeaportsUpdated" event to the Message Broker containing the relevant data. Quotation Service is subscribed to this event, so it is notified and updates its database if necessary. This process is shown in Figure 4.7.

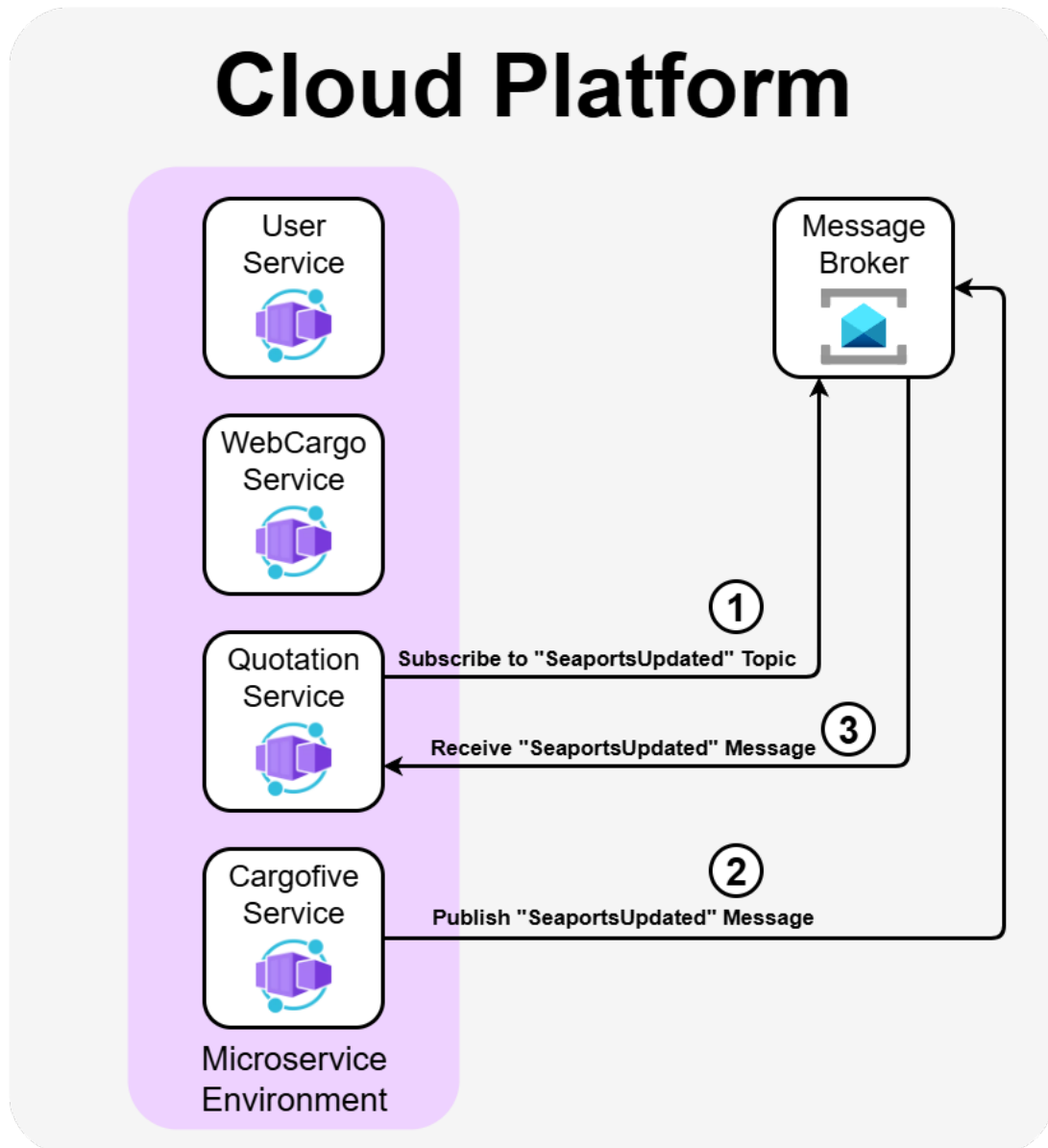


FIGURE 4.7. Event-Driven Inter-Service Communication: Cargofive Synchronization

4.3.6. External Carrier API Communication

WebCargo Service and Cargofive Service communicate with the WebCargo and Cargofive APIs, respectively. The two services use the APIs as clients to send requests and receive responses, as shown in Figure 4.8. The API keys required for authentication are retrieved from the Secret Storage at runtime and stored in memory.

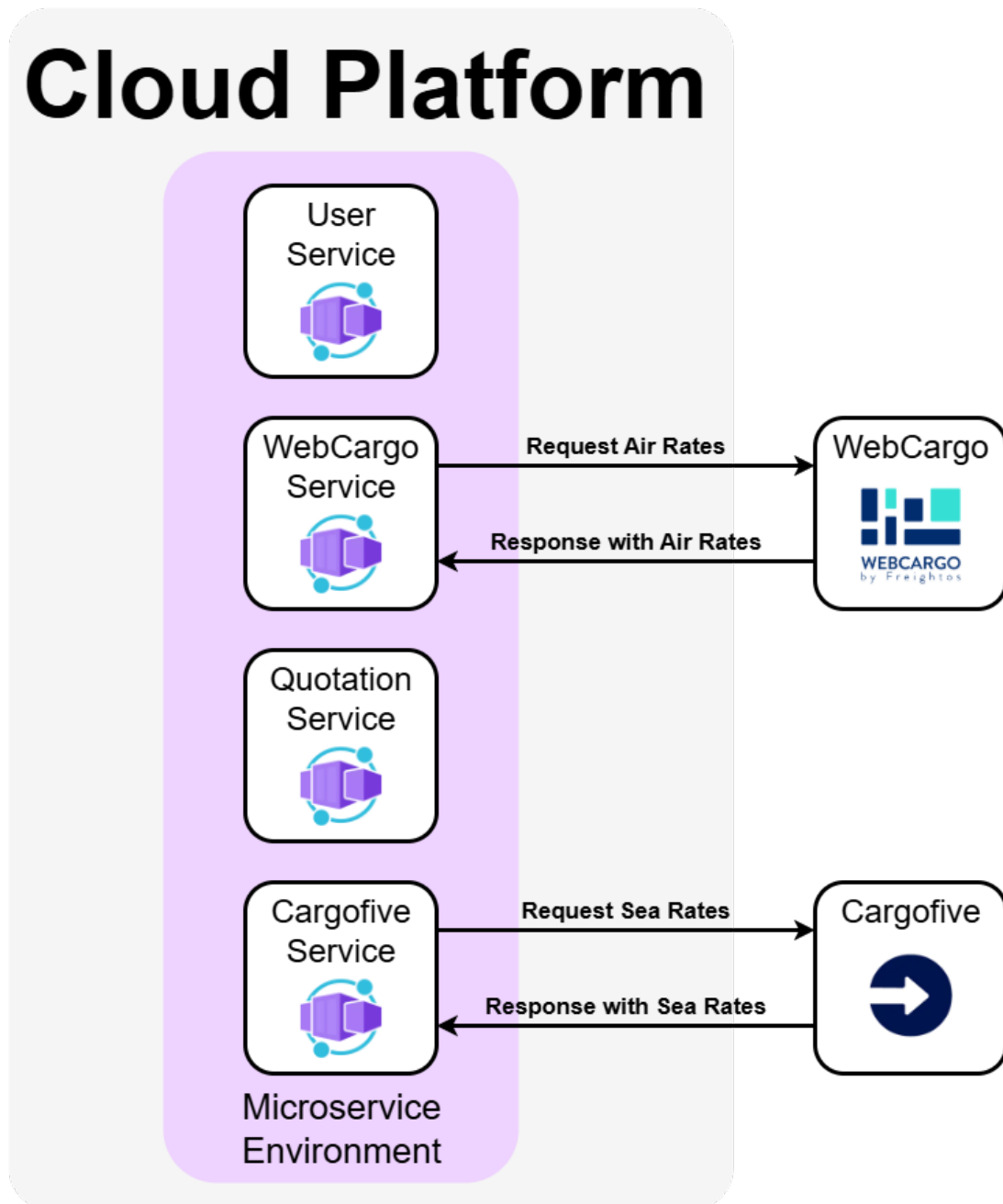


FIGURE 4.8. External Carrier API Communication

4.4. API Specification

The API specification is a document that lays out the endpoints, data schemas, and request-response formats of the web API, providing a clear notion of how to interact with it. Figure 4.9 shows the web API's specification, which adheres to the OpenAPI standard¹. This document contains all endpoints that the microservices expose to the Client through the API Gateway.

User		
GET	/users/me	▼
GET	/users/{id}	▼
GET	/users	▼
AirQuotation		
GET	/quotes/air/{id}	▼
DELETE	/quotes/air/{id}	▼
POST	/quotes/air	▼
GET	/quotes/air/history	▼
SeaQuotation		
GET	/quotes/sea/{id}	▼
DELETE	/quotes/sea/{id}	▼
POST	/quotes/sea	▼
GET	/quotes/sea/history	▼
Location		
GET	/locations/{id}	▼
GET	/locations	▼
SpecialHandlingCode		
GET	/special-handling-codes	▼

FIGURE 4.9. API Specification

¹OpenAPI Specification, <https://swagger.io/specification/>, (accessed 31 Aug. 2025)

4.5. Internal Architecture of the Microservices

In the web API, the source code of each microservice adheres to a similar architecture. Figure 4.10 shows the main modules of this architecture and their interactions, where all arrows inside the "Microservice" region represent function calls in the running program. Each module is designed to handle specific types of tasks and responsibilities, promoting code modularity and maintainability. The next subsections explain the purpose of each module.

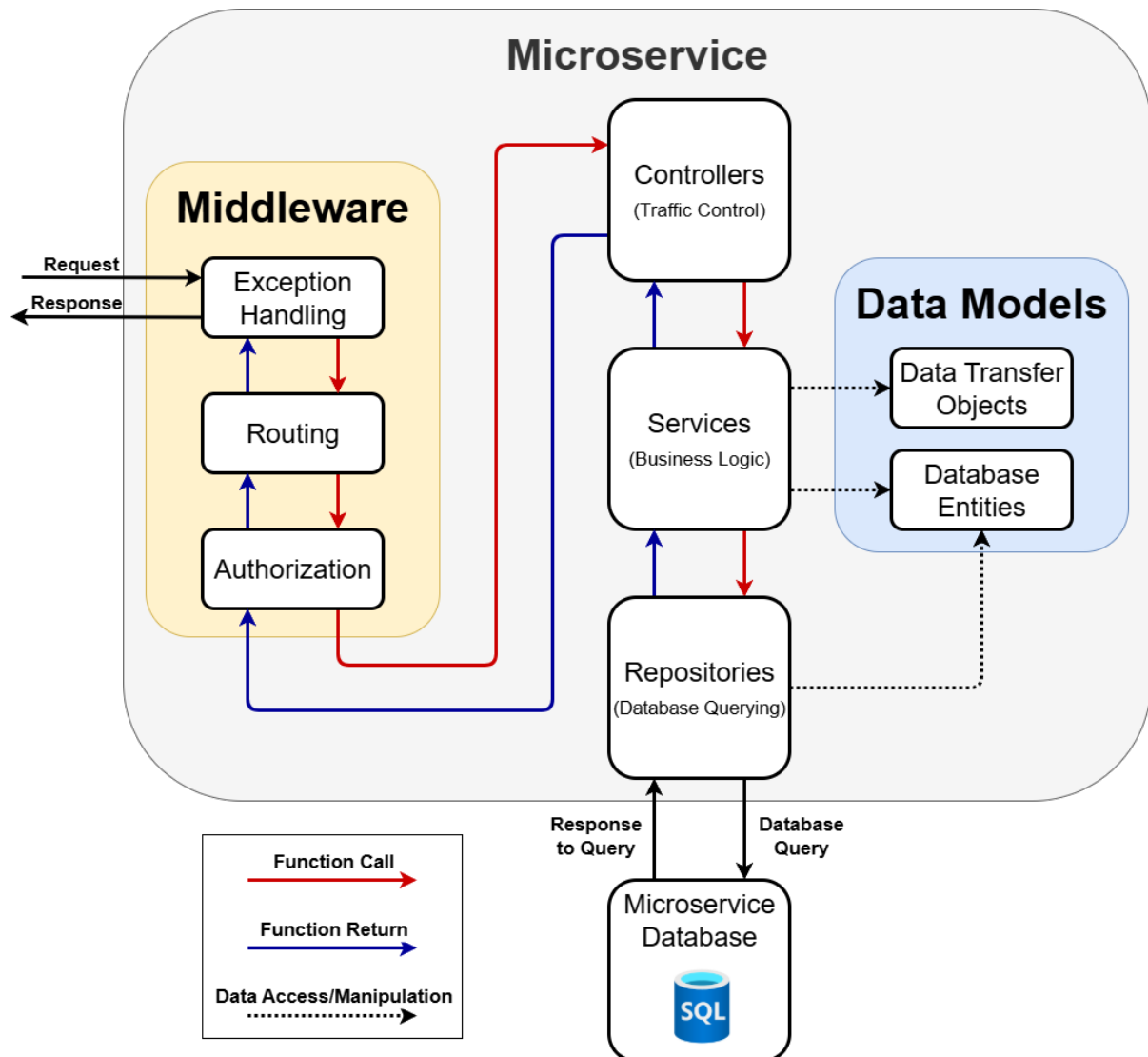


FIGURE 4.10. Internal Architecture of a Microservice

4.5.1. Middleware

The middleware pipeline directly interacts with incoming requests and outgoing responses in a specified order of middleware processes. Each process can inspect or change the request or response. In the web API being designed, three middleware processes are applied in this exact order:

Exception Handling: This process is invoked at the beginning of the request pipeline.

If an unhandled error occurs during the processing of a request, it catches the error and generates a response with a fitting error code and message.

Routing: This process is responsible for matching incoming requests to the appropriate endpoint based on the request's specified destination. The destination of the request depends on its Uniform Resource Locator (URL) and HTTP method.

Authorization: This process checks whether the authenticated user has the necessary permissions to access the requested resource. If the user lacks the required permissions, it blocks the request and generates a response with a "403 Forbidden" status code. For endpoints that allow anonymous access, this process has no effect. Authentication and authorization requirements are defined in the controller modules.

4.5.2. Data Models

Data models represent business data structures. In the web API's source code, they correspond to objects that are designed for data encapsulation but contain no business logic. There are two types of data models:

Data Transfer Objects (DTOs): These objects hold business data and may contain validation or transformation logic. Microservice components use DTOs for data interchange in many API interactions, including those with the Client, other microservices, and the external carrier APIs.

Database Entities: These data models represent database tables and are used to map the microservice's business data to the the correspondent database schema. An instance of an entity is equivalent to a row in the database table it represents. Entities allow the database schema to be programmatically defined in the microservice's source code.

4.5.3. Controllers

Controllers are responsible for handling requests routed by the middleware. Each controller defines one or more endpoints corresponding to a specific resource (location, user, freight quote, etc.) or functionality of the microservice. These modules control traffic by specifying each endpoint's authentication and authorization requirements, as well as their expected input and output parameters. Controllers interact with the necessary Service modules to fulfill the request, and initiate a response with relevant data to the Client.

4.5.4. Services

Service modules execute the core business logic of the microservice and coordinate interactions between other modules. They use DTOs to exchange data with external APIs, the Client, and other microservices. These modules can create database entity instances directly to then store them in the database by interacting with repository modules. Service modules also interact with repository modules to retrieve, modify, or delete database records.

4.5.5. Repositories

Repositories provide functions for database access and manipulation. They act as an abstraction layer between the service modules and the database, holding the logic required to interact with the microservice's database. Repositories use database entity instances to perform CRUD operations on behalf of service modules. Furthermore, repository modules are agnostic to specific database technologies.

4.6. Deployment

The web API's microservices are deployed to the cloud platform through a CI/CD pipeline. This pipeline automates the build, test, and deployment processes, ensuring that the relevant microservice instances running in the cloud are promptly updated. The CI/CD pipeline is triggered whenever a change is pushed to the main branch of the Code Repository. Figure 4.11 illustrates the two main steps of the deployment process.

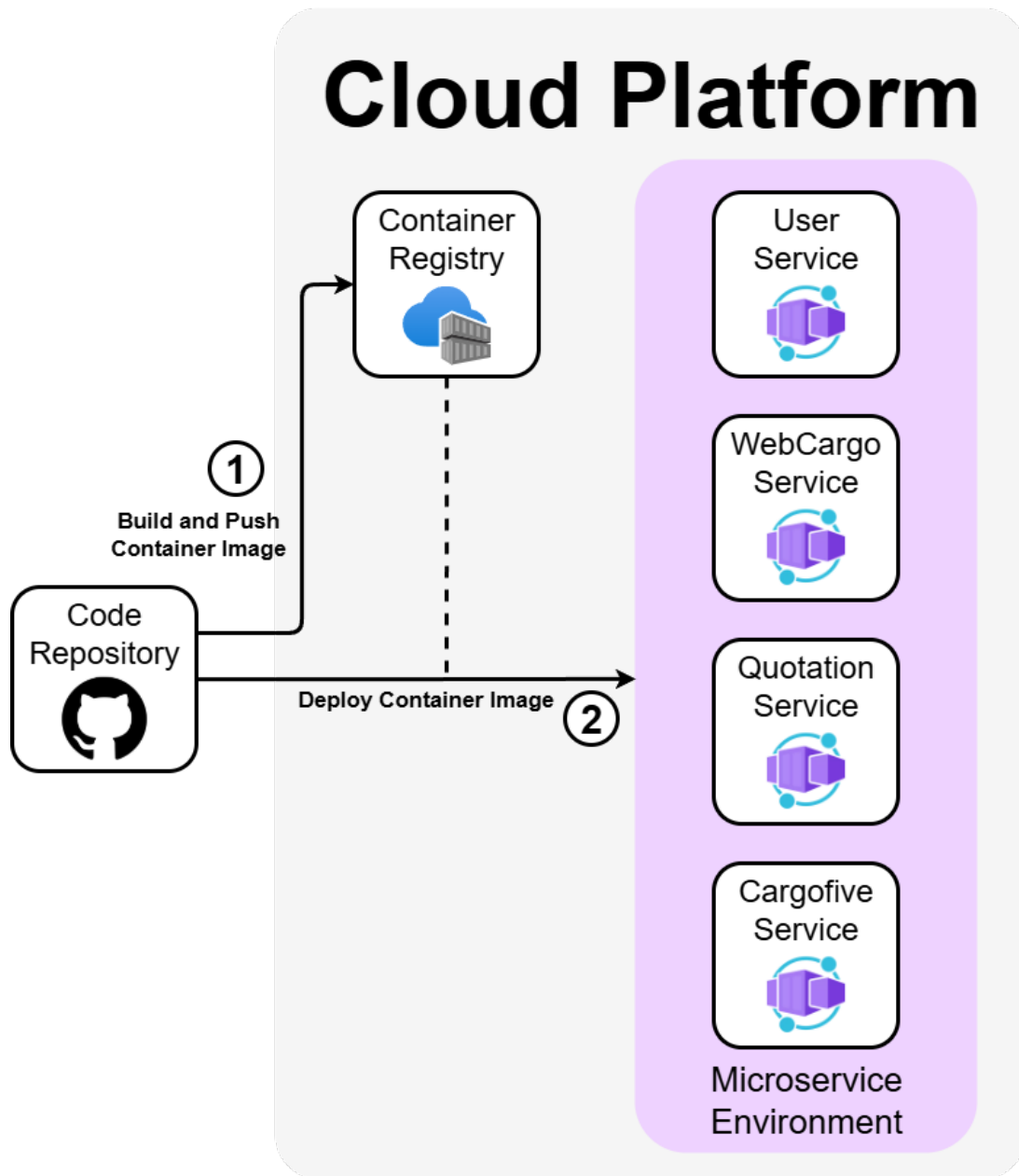


FIGURE 4.11. Deployment to the Cloud Platform

In the first stage, the source code is compiled with a production-ready configuration and built into a container image, which is then pushed to the Container Registry. In the second stage, the container image is deployed to the cloud platform's container orchestration service (the Microservice Environment), where it is run in a scalable and supervised environment. Both of these stages involve authenticating with the relevant cloud platform components using secrets stored in the Code Repository.

4.7. Technology Stack

Although the web API's architecture described throughout Chapter 4 is intentionally agnostic to a specific technology stack, particular technologies must be chosen for the implementation stage in Chapter 5.

The selected cloud platform is Microsoft Azure², which provides a wide range of PaaS services that facilitate the creation and management of the web APIs's components. Table 4.1 links each cloud platform component mentioned in Section 4.2 to the corresponding Azure service type.

Web API Component	Azure Service
API Gateway	API Management
Microservice Environment	Container Apps Environment
VNet	Virtual Network
Data Persistence Layer	SQL Server
Identity Provider	Microsoft Entra ID
Message Broker	Service Bus
Secret Storage	Key Vault
Container Registry	Container Registry

TABLE 4.1. Azure Service of each Cloud Platform Component

Every microservice in the Microservice Environment is built with the ASP.NET Core 8.0 framework, using C# as the programming language. This framework was chosen for its performance, extensive documentation, and integration with Azure services. Finally, the web API follows the REST architectural style for simplicity and standardization.

²Microsoft Azure, <https://azure.microsoft.com/>, (accessed 31 Aug. 2025)

CHAPTER 5

Implementation

5.1. Development Environment

During development, JetBrains Rider¹ Integrated Development Environment (IDE) was used as the main tool for writing, testing, and debugging the microservices' source code. This IDE provides a vast set of features, including code completion, refactoring tools, and integrated debugging support, which significantly increase developer productivity.

Docker Desktop² was installed to test the Docker containers and register them to the Container Registry.

All cloud platform components were configured and managed in the Azure Portal, which offers an intuitive graphical interface.

Postman³ was utilized to manually validate the web API's endpoints during development by acting as the Client component.

The complete source code for the API's microservices and their respective deployment workflows are available in a public GitHub repository⁴.

¹JetBrains Rider, <https://www.jetbrains.com/rider/>, (accessed 31 Aug. 2025)

²Docker Desktop, <https://www.docker.com/products/docker-desktop/>, (accessed 31 Aug. 2025)

³Postman, <https://www.postman.com/>, (accessed 31 Aug. 2025)

⁴<https://github.com/RafaelSantos777/FreightQuotationWebAPI>

5.2. Microservices Implementation

5.2.1. Project Structure

There are five projects in the .NET solution, one for each microservice and a shared project. Each microservice project contains its own implementation of the architecture described in Section 4.5. The shared project contains common code that can be useful to multiple microservices, such as constant values and utility functions. Figure 5.1 shows the project structure of the solution, with Quotation Service being chosen as the example.

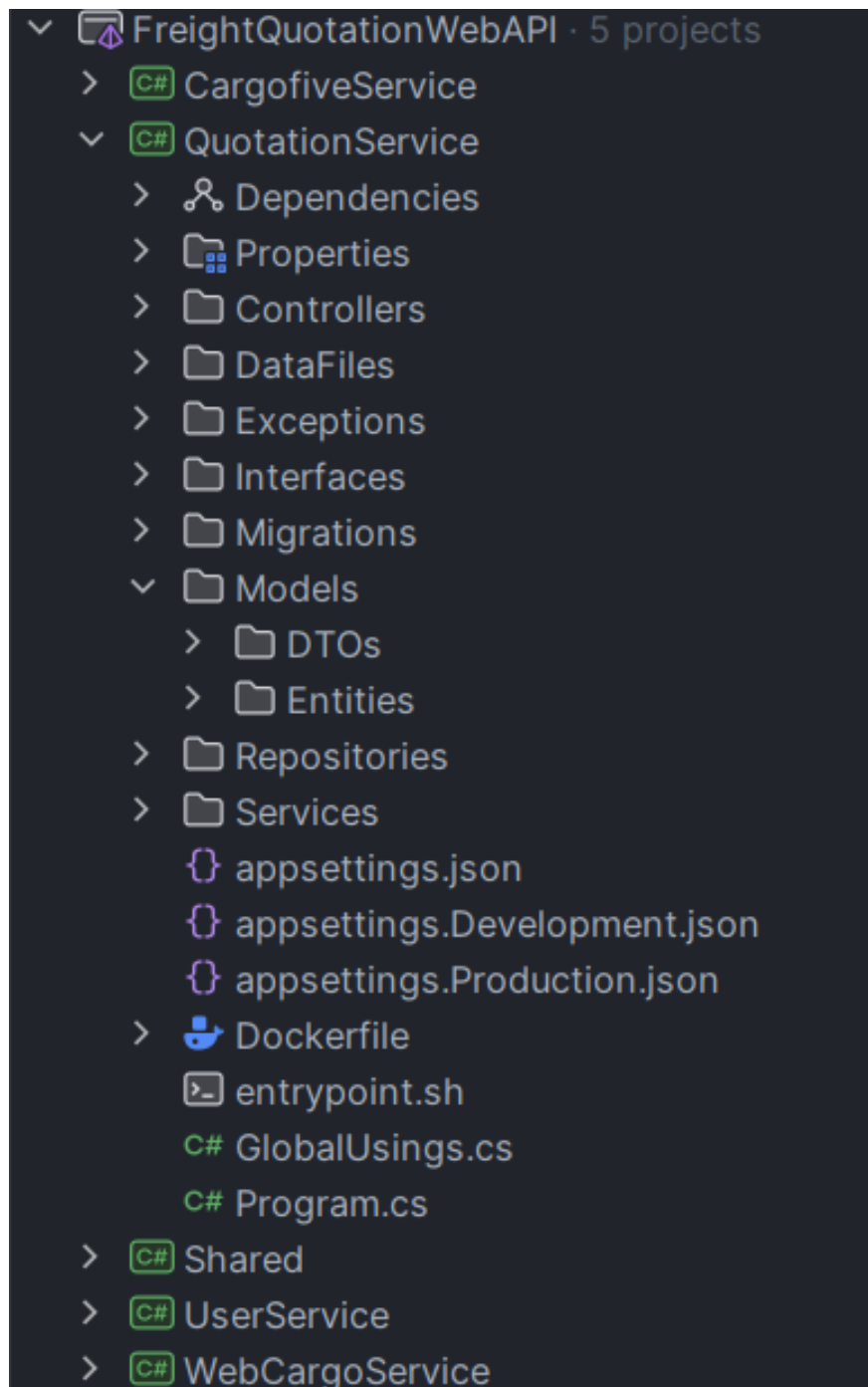


FIGURE 5.1. Project Structure

Besides the microservice components described in Section 4.5, each microservice also contains other important directories and files, including:

Dependencies: This directory contains all NuGet package dependencies required by the microservice.

DataFiles: This directory consists of miscellaneous data files. Quotation Service reads files in this directory to store initial data for locations and special handling codes, which are loaded into the database when the microservice starts for the first time.

Exceptions: This directory contains custom exception classes that represent specific error conditions in the microservice.

Interfaces: This directory contains interface definitions for Service and Repository modules. These interfaces define the contracts that implementing classes must adhere to, promoting loose coupling and easier testing.

Migrations: This directory holds the database migration files generated by .NET Entity Framework Core. These files define the changes to be applied to the database schema over time. When the database definition in the source code changes, a new migration file is created to represent the change. The database migration is applied before the microservice starts.

appsettings.json: This file contains configuration settings for the microservice, such as database connection strings and information related with the cloud platform components. There are also environment-specific versions of this file ("appsettings.Development.json" and "appsettings.Production.json") that specify additional settings when the microservice is running in a specific environment. These files do not contain secret values.

Dockerfile: This file defines the instructions to build a Docker image for the microservice. It specifies the base image, copies the source code, builds the project, and defines the entry point for running the microservice in a container.

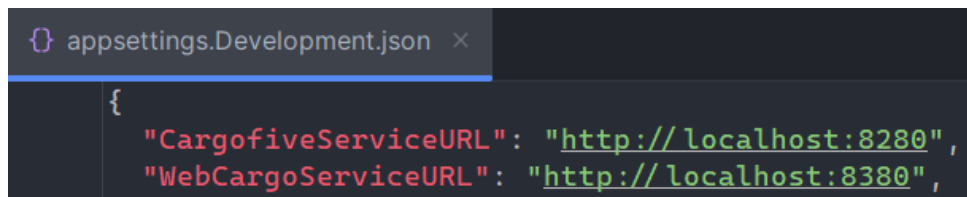
entrypoint.sh: This script is executed when the microservice container starts. The microservice container has two distinct behaviors based on the parameters passed to this script. If the first parameter is "migrate", the script applies any pending database migrations and then exits. Otherwise, the script runs the microservice normally by executing the code defined in Program.cs.

Program.cs: This C# file is responsible for configuring the web server, the middleware pipeline, and starting the web API.

5.2.2. Freight Rates and Quotes Obtainment

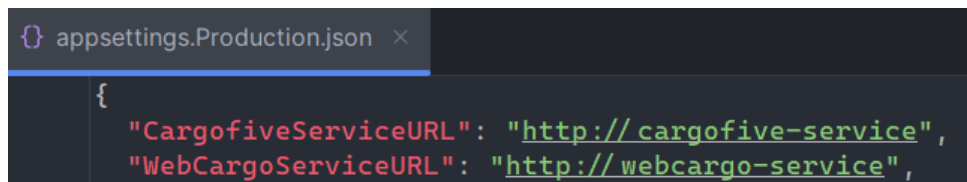
When user makes a quote request to the web API, it is routed to Quotation Service, which orchestrates the process by interacting with WebCargo Service and Cargofive Service.

Through configuration files, Quotation Service knows the addresses of WebCargo Service and Cargofive Service, which depend on whether the services are running in a local development environment or in an Azure production environment. Figures 5.2 and 5.3 show the addresses used in each environment. The addresses in a production environment are mapped to the full domain names of the services in the VNet through Domain Name System (DNS) records.



```
{
  "CargofiveServiceURL": "http://localhost:8280",
  "WebCargoServiceURL": "http://localhost:8380",
}
```

FIGURE 5.2. Service Addresses (Development Environment)



```
{
  "CargofiveServiceURL": "http://cargofive-service",
  "WebCargoServiceURL": "http://webcargo-service",
}
```

FIGURE 5.3. Service Addresses (Production Environment)

In this subsection, the process of obtaining an air freight quote is described. The process for sea freight quotes is analogous, with the only significant difference being that Cargofive Service is used instead of WebCargo Service.

When Quotation Service receives an air quote request from an end user, it is transformed into a DTO corresponding to a C# object shown in Figure 5.4. If the DTO is valid, Quotation Service sends a direct request to a WebCargo Service. This request comprises all the essential query parameters to search for air freight rates from the WebCargo API: the International Air Transport Association (IATA) codes of the origin and destination airports. Quotation Service stores locations in its database, so it can retrieve the IATA codes based on the location identifiers received in the air quote request. Quotation Service and WebCargo Service have an equivalent definition of the request DTO, shown in Figure 5.5.

```

namespace QuotationService.Models.DTOs.Internal;

public record AirQuoteRequestDTO {

    long OriginAirportId
    long DestinationAirportId
    double LengthCentimeters
    double WidthCentimeters
    double HeightCentimeters
    double WeightKilograms
    string CurrencyCode
    string? SpecialHandlingCode
}

```

FIGURE 5.4. Air Quote Request DTO

```

namespace WebCargoService.Models.DTOs.Internal;

public record RateRequestDTO {

    string OriginAirportIATACode
    string DestinationAirportIATACode
}

```

FIGURE 5.5. Air Rate Request DTO

When WebCargo Service receives the freight rate request, it checks if a valid cache entry exists for the given origin and destination. If it does, the cached freight rates are simply returned to Quotation Service as a list of DTOs shown in Figure 5.6.

If no valid cache entry exists, a request is made to the WebCargo API. Once the response is received, its body containing the freight rates is deserialized into a list of DTOs with a structure similar to the WebCargo API's response body. Then, the retrieved freight rates are stored in the cache and sent to Quotation Service as previously described.

```

namespace WebCargoService.Models.DTOs.Internal;

public record RateDTO {

    long UniqueCode
    string Airline
    string OriginAirportIATACode
    string DestinationAirportIATACode
    decimal VolumetricFactorMetric
    string ProductName
    string CurrencyCode
    DateOnly ValidFrom
    DateOnly? ValidTo
    string SpecialHandlingCodeString
    List<RateSurchargeDTO> Surcharges
    decimal MinimumBreakpointCost
    List<RateBreakpointDTO> Breakpoints

```

FIGURE 5.6. Air Rate DTO

Finally, Quotation Service works with the received freight rates to calculate freight quotes based on the user's input. The freight quote response is stored in the database and returned to the user as a DTO shown in Figure 5.7.

```

namespace QuotationService.Models.DTOs.Internal;

public record AirQuoteResponseDTO {

    long Id
    AirQuoteRequestDetailedDTO AirQuoteRequestDetailed
    List<AirQuoteDTO> AirQuotes

```

FIGURE 5.7. Air Quote Response DTO

The functionality described in this subsection can be tested in Postman. Figure 5.8 shows an example air quote request. Figure 5.9 shows a part of the corresponding response.

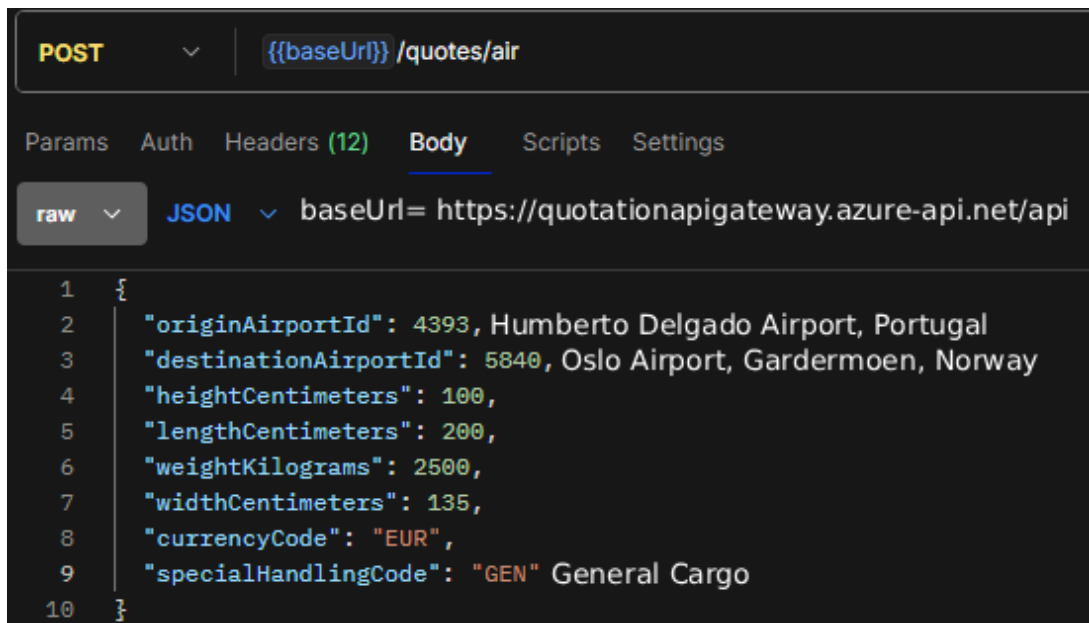


FIGURE 5.8. Air Quote Request in Postman

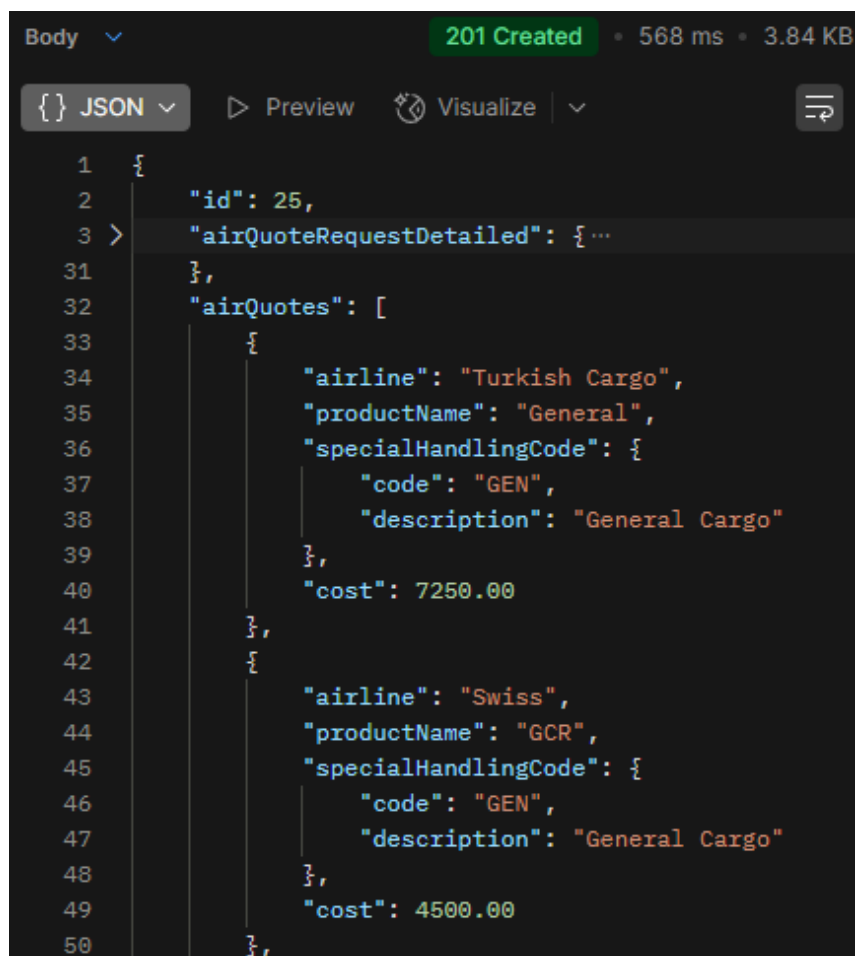


FIGURE 5.9. Air Quote Response in Postman

5.2.3. Freight Rates Cache

External carrier APIs impose rate limits on the number of requests that can be made within a specific time frame. To avoid exceeding these limits, freight rates obtained from the WebCargo and Cargofive APIs are cached for 1 day. The caching mechanism is implemented in WebCargo Service and Cargofive Service, respectively.

Each cache entry is based on a unique combination of request parameters that define what freight rates will be returned by the external carrier API. For both APIs, a combination of the origin and destination locations are sufficient to uniquely identify a cache entry. When a request for freight rates is received, the service first checks if a valid cache entry exists. If it does, the cached freight rates are returned. If not, a request is made to the external carrier API, and the retrieved freight rates are stored in the cache.

The main purpose of the caching mechanism is to reduce the number of requests to the external carrier APIs. Because of this, the databases of WebCargo Service and Cargofive Service store not only the rate history, but also the cached data, for simplicity. However, this implementation has some drawbacks, such as increased cached lookup times and potential database load. In future work, a dedicated caching alternative like Redis Cache⁵ would be useful for its increased performance and built-in cache expiration mechanisms.

5.2.4. Quote History

Whenever Quotation Service processes a freight quote request, it stores the request and response data in its database. This allows users to retrieve their quotation history through a dedicated endpoint.

The returned history is ordered from newest to oldest and paginated to limit the number of results in a single response. Pagination parameters are specified in the request query string. The parameter "page" indicates the page number, while "limit" indicates the number of results per page. If these parameters are not specified, default values are used (1 for page and 100 for limit).

Figure 5.10 shows an example request to the quote history endpoint in Postman. Figure 5.11 shows the corresponding response, which was collapsed for visibility.

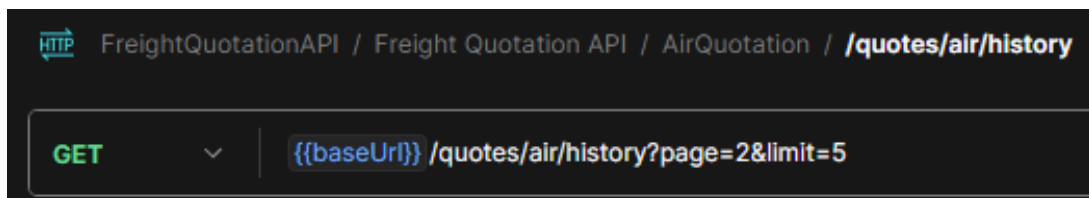


FIGURE 5.10. Quote History Request in Postman

⁵Redis Cache, <https://redis.io/docs/>, (accessed 12 Sep. 2025)


```
Body 200 OK • 108 ms • 31.21 KB
[{} JSON Preview Visualize
1  [
2    {
3      "id": 20,
4    > "airQuoteRequestDetailed": { ...
32    },
33  > "airQuotes": [ ...
133  ]
134  },
135  {
136    "id": 19,
137  > "airQuoteRequestDetailed": { ...
165  },
166  > "airQuotes": [ ...
176  ]
177  },
178  {
179    "id": 18,
180  > "airQuoteRequestDetailed": { ...
208  },
209  > "airQuotes": [ ...
300  ]
301  },
302  {
303    "id": 17,
304  > "airQuoteRequestDetailed": { ...
332  },
333  > "airQuotes": [ ...
586  ]
587  },
588  {
589    "id": 16,
590  > "airQuoteRequestDetailed": { ...
615  },
616  > "airQuotes": [ ...
1967  ]
1968  }
1969  ]
```

FIGURE 5.11. Quote History Response in Postman

5.2.5. Location Search

Quotation Service exposes an endpoint that allows users to search for locations by name. This functionality is useful when users need to find the identifier of a location to use in a freight quote request.

The location search endpoint accepts a query parameter for the location name and another for the location type. Furthermore, it supports pagination, similarly to the one described in Subsection 5.2.4. The endpoint returns a list of matching locations with their identifiers and other information.

Figure 5.12 shows an example request to the endpoint in Postman for airport locations whose name contains "barce". Figure 5.13 shows the resulting response.

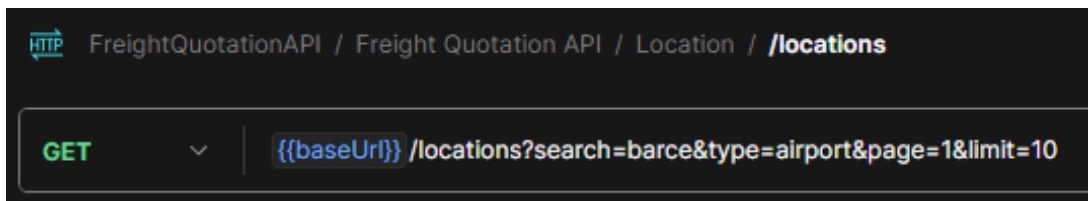


FIGURE 5.12. Location Search Request in Postman

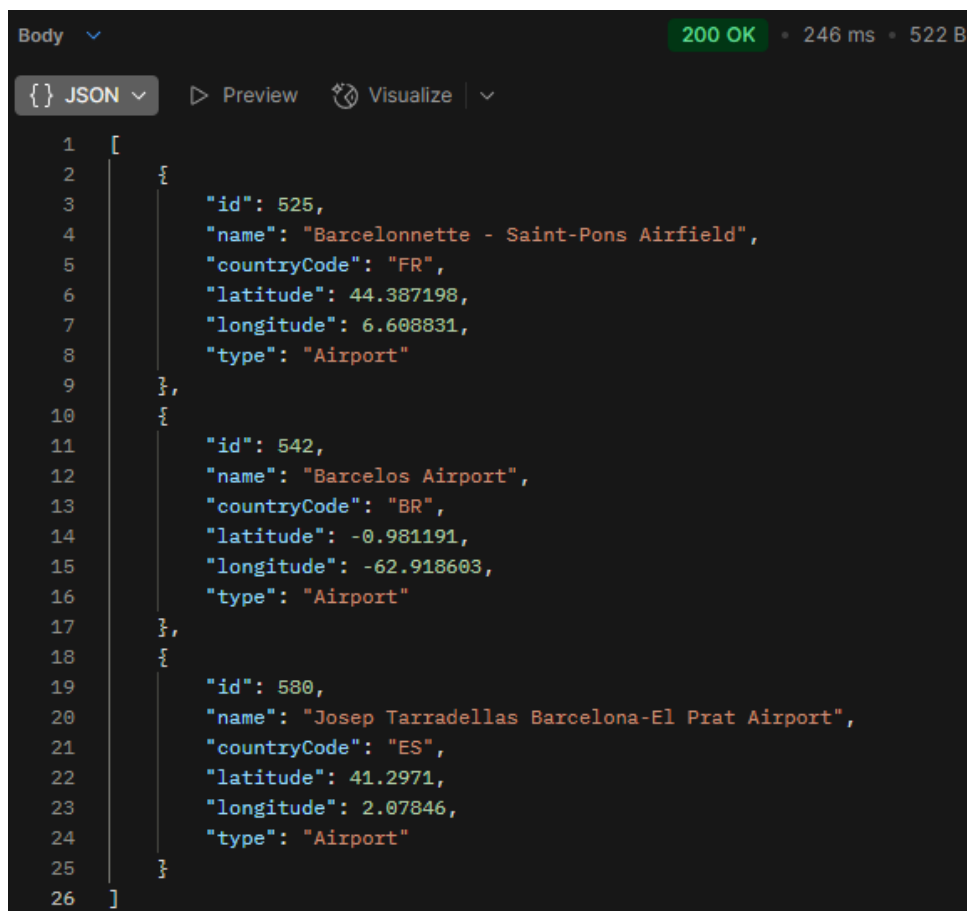


FIGURE 5.13. Location Search Response in Postman

5.2.6. Role-Based Access Control (RBAC)

The microservices rely on the authentication verification performed by the API Gateway. Therefore, they do not need to validate access tokens themselves. However, they must enforce authorization policies to restrict access to resources based on user roles or more complex business rules.

Only users with the "Administrator" role can access the endpoint to search for other users in the system. User roles can be found in the "roles" claim of the access token (JWT). User roles can be assigned manually through the Azure portal. Alternatively, User Service can assign roles by calling the appropriate Graph API⁶ endpoint, which is used to interact with Microsoft's cloud resources.

Figure 5.14 shows a search request in Postman for users whose name contains "mar". Figure 5.15 shows the "403 Forbidden" response received when the request is made with a JWT of a user without the "Administrator" role. Conversely, Figure 5.16 shows the successful response received when the request is made with a JWT of an administrator.

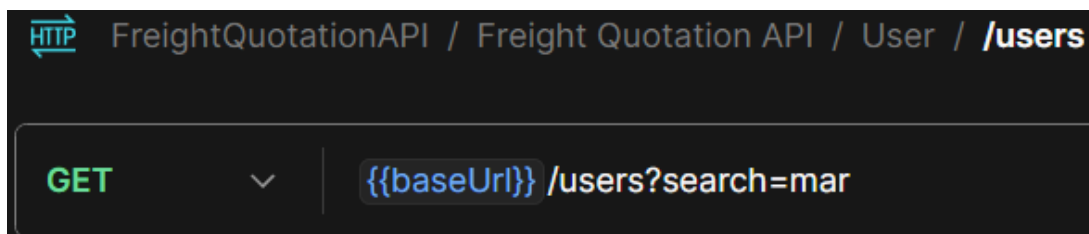


FIGURE 5.14. User Search Request in Postman

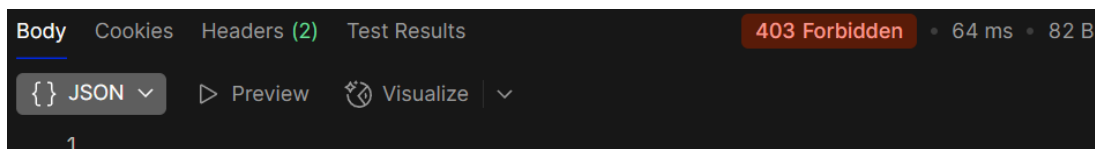


FIGURE 5.15. User Search Response in Postman (Access Forbidden)

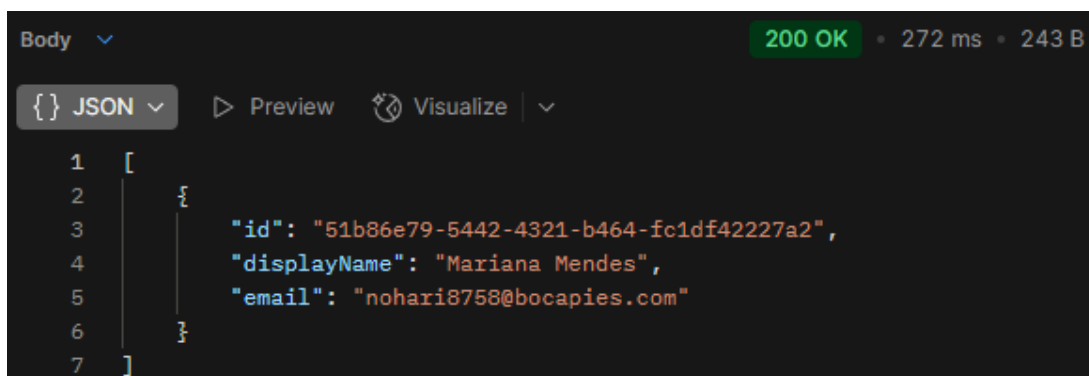


FIGURE 5.16. User Search Response in Postman (Success)

⁶Microsoft Learn, "Use the Microsoft Graph API", <https://learn.microsoft.com/en-us/graph/use-the-api>, (accessed 15 Sep. 2025)

5.2.7. User Data Deletion

When a user is deleted, all their associated data is also deleted from the web API's databases. Since Entra ID is the Identity Provider, it is responsible for managing user accounts. Therefore, User Service must be informed about user account deletions by communicating with Entra ID. In Subsection 4.3.5, two possible approaches to get this information were mentioned: subscribing to the Identity Provider's user deletion notifications, or polling the Identity Provider periodically. Entra ID supports both approaches through the Microsoft Graph API.

The first approach involves subscribing to Microsoft Graph API's change notifications for user deletions. This requires setting up a webhook endpoint in User Service to receive notifications, which Entra ID communicates with. However, the Graph API only supports this approach for workforce tenants, which are intended for employees of an organization. The implemented web API is intended for general customer users and thus utilizes a different tenant type. Therefore, this event-driven approach is not feasible for the web API.

The second approach involves polling the Microsoft Graph API periodically. Indiscriminate polling would be inefficient, as it would involve returning all users and checking for deletions. Instead, User Service uses delta queries⁷ to retrieve only the changes that occurred since the last query. This is achieved by storing a delta link provided by the Graph API after each query in the User Service's database. The delta link is then used in subsequent queries to fetch only the changes since the last query. User Service polls the Graph API every 15 minutes.

When User Service retrieves at least one deleted user through polling, it sends a batch of messages to a Service Bus (Message Broker) topic called "user_deleted". Each message contains the identifier of a deleted user. Any microservice may subscribe to this topic. In particular, Quotation Service has a subscription to this topic and processes each message by deleting the quote history of the corresponding user from its database.

5.3. Cloud Components Configuration

5.3.1. API Gateway Configuration

An Azure API Management instance is used to route incoming requests to the appropriate microservice, thus acting as the API Gateway. A single API is added to the API Management instance by importing the definition from the OpenAPI specification file mentioned in Section 4.4. This API is named "Freight Quotation API".

The Freight Quotation API has a "Settings" tab, shown in Figure 5.17, where general configurations are made. Secure HTTP access is enforced by allowing only Hypertext Transfer Protocol Secure (HTTPS) requests. Furthermore, the base public URL is defined ("https://quotationapigateway.azure-api.net/api"), which clients must use to access the web API.

⁷Microsoft Learn, "Use delta query to track changes in Microsoft Graph data", <https://learn.microsoft.com/en-us/graph/delta-query-overview>, (accessed 15 Sep. 2025)

Design
Settings
Test
Revisions (1)
Change log

General

*
Display name

Freight Quotation API

*
Name

freight-quotation-api

Description ⓘ

Web service URL

e.g. httpbin

URL scheme

☐ HTTP
☒ HTTPS
☐ HTTP(S)

API URL suffix

api

Base URL

https://quotationapigateway.azure-api.net/api

FIGURE 5.17. API Gateway Settings

Unlike the microservice components, the API Gateway accepts communication from the public internet, as it resides in a subnet that allows such traffic. The microservices, which reside in a different subnet, rely on the API Gateway to interact with the Client. Because of this, each microservice is added as a separate backend service to the API Gateway, as shown in Figure 5.18. The "Backend name" is used as a simple alias to identify the backend service. The "Run URL" is the internal URL of the microservice in the VNet.

Backend name	Description	Type	Runtime URL
quotation-service		Custom URL	https://quotation-service.greenground-aff007cf.spaincentral.azurecontainerapps.io
user-service		Custom URL	https://user-service.greenground-aff007cf.spaincentral.azurecontainerapps.io

FIGURE 5.18. API Gateway Backends

The API Gateway's behavior towards the Freight Quotation API is defined in the "Design" tab. In this tab, an XML configuration file defines multiple policies that are applied to incoming requests or outgoing responses. Considering the roles of the API Gateway mentioned in Subsection 4.2.2, the policies can be split into three parts: Cross-Origin Resource Sharing (CORS), authentication, and routing.

CORS policies are applied to all incoming requests to allow cross-origin requests from the Client. Since the Client application does not have a fixed domain name, the CORS policy allows requests from any origin. The API Gateway also accepts any HTTP method. However, if the Client had a fixed domain name, the CORS policy would be more restrictive. The CORS policy is shown in Figure 5.19.

```

<cors>
  <allowed-origins>
    <!--Allow any origin-->
    <origin>*</origin>
  </allowed-origins>
  <allowed-methods>
    <!--Allow any HTTP method-->
    <method>*</method>
  </allowed-methods>
</cors>

```

FIGURE 5.19. API Gateway CORS Policy

The next step is to enforce authentication by validating the access token (JWT) included in the "Authorization" header of incoming requests. The API Gateway uses the OpenID Connect metadata document of the Identity Provider to validate the token's signature and claims. If the token is invalid or missing, the request is rejected with a "401 Unauthorized" response. Figure 5.20 displays the authentication policy configuration with explanatory comments.

```

<choose>
  <!--If accessing "/locations" or "/special-handling-codes" endpoints, skip authentication check-->
  <when condition="@ (new [] { "/api/locations", "/api/special-handling-codes" }
    .Any(path => context.Request.OriginalUrl.Path.StartsWith(path)))" />
  <!--If accessing other endpoints, enforce authentication-->
  <otherwise>
    <!--JWT must be specified in the "Authorization" header of the request-->
    <validate-jwt header-name="Authorization" failed-validation-httpcode="401">
      <openid-config url="https://freightquotation.ciamlogin.com/e2b3262c-0ae0-456d-8c49-6cbe7422af1f" />
      <audiences>
        <!--"83b002e0-b385-4b11-8a7b-5dbf91bf18a4" is the identifier of the server-side web API-->
        <!--Used to verify if the JWT's intended use is for the web API-->
        <audience>83b002e0-b385-4b11-8a7b-5dbf91bf18a4</audience>
      </audiences>
    </validate-jwt>
  </otherwise>
</choose>

```

FIGURE 5.20. API Gateway Authentication Policy

Finally, the API Gateway routes incoming requests to the appropriate microservice based on the request's destination. The routing policy is shown in Figure 5.21. The policy uses the backend identifier of each backend service defined in Figure 5.18 to identify the target microservice.

```

<choose>
  <!--If accessing "/locations", "/special-handling-codes", or "/quotes" endpoints, route to Quotation Service-->
  <when condition="@ (new [] { "/api/locations", "/api/special-handling-codes", "/api/quotes" }
    .Any(path => context.Request.OriginalUrl.Path.StartsWith(path)))" />
  <set-backend-service backend-id="quotation-service" />
  </when>
  <!--If accessing "/users" endpoints, route to User Service-->
  <when condition="@ (new [] { "/api/users" }.Any(path => context.Request.OriginalUrl.Path.StartsWith(path)))" />
  <set-backend-service backend-id="user-service" />
  </when>
</choose>

```

FIGURE 5.21. API Gateway Routing Policy

5.3.2. Microservice Environment Configuration

The Azure Container Apps Environment hosts all microservices as Azure Container Apps. In addition, there is one Container App Job for each microservice, which is used to apply database migrations when a new deployment occurs. The Container Apps Environment resides within a dedicated subnet ("ContainerEnvironmentSubnet") in the VNet to ensure secure communication. This subnet adheres to default Azure subnet policies, which only allow traffic from other components within the VNet, or private endpoints. Figure 5.22 shows the overall configuration of the Container Apps Environment.

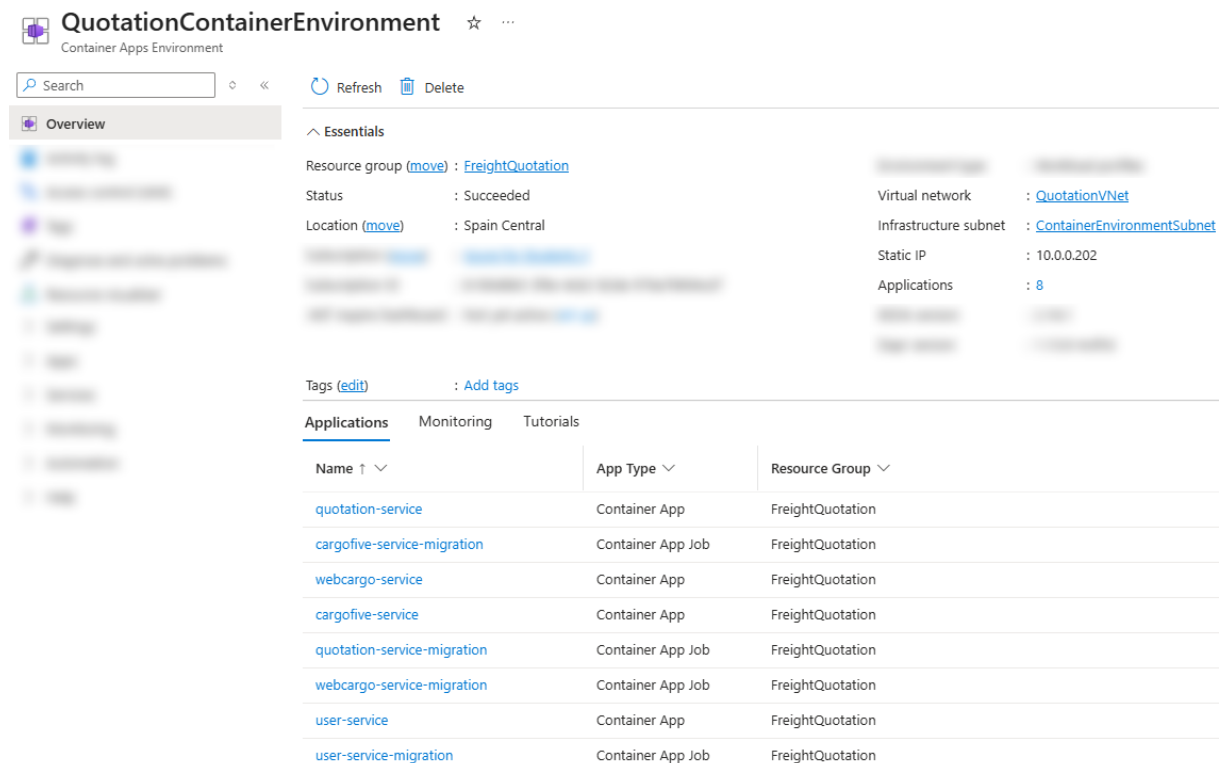


FIGURE 5.22. Container Apps Environment Configuration

Each microservice is deployed as a separate Container App. Each Container App runs a specific Docker image version stored in the Container Registry. When the CI/CD pipeline deploys a new version of a microservice, it updates the Container App to use the new Docker image, creating a revision of the Container App. Microservices run in Docker containers of that image.

Another useful feature is the ability to set scaling rules for each microservice, which may be based on HTTP traffic, CPU usage, or memory usage. Specifically, every Container App is configured to scale between 1 and 10 instances, based on the average number of concurrent HTTP requests being processed per instance. If this number exceeds 10, the Container App will automatically scale up to deal with the increased load. Conversely, if the average workload remains lower for 5 minutes, the Container App will scale down to conserve resources. Figure 5.23 shows the scaling configuration.

Scale

Scale rule settings

Control automatic scaling by setting the range of application replicas that'll be deployed in response to a trigger event. Use scale rules to determine the type of events that trigger scaling. [Learn more](#)

Min replicas ⓘ

1

Min: 0

Max replicas ⓘ

10

Max: 1000

Cooldown period ⓘ

300

Polling interval ⓘ

30

Current number of replicas ⓘ

1 [\(View Details\)](#)

Scale rules

Search

+ Add

Name ↑ ↓	Type ↑ ↓	Del... ↓
http-scaler	HTTP scaling	

FIGURE 5.23. Container App Scaling Rules

5.3.3. Data Persistence Layer Configuration

5.3.4. Message Broker Configuration

5.3.5. Secret Storage Configuration

For security reasons, the source code of each microservice does not contain any secret values. Instead, Azure Key Vault is used to securely store confidential information for production environments, which is made available to the microservices through a private endpoint. The Key Vault stores the API keys to authenticate with external carrier APIs, the Service Bus connection string, and the secret to access Entra ID's resources. Figure 5.24 enumerates the names of all secrets.





 Generate/Import  Refresh  Restore Backup  Manage deleted secrets		
Name	Type	Status
CargofiveAPI--APIKey		✓ Enabled
ConnectionStrings--ServiceBus		✓ Enabled
EntraID--ClientSecret		✓ Enabled
WebCargoAPI--APIKey1		✓ Enabled
WebCargoAPI--APIKey2		✓ Enabled

FIGURE 5.24. Key Vault Secrets

5.4. CI/CD Pipeline Configuration

CHAPTER 6

Evaluation

6.1. Testing Strategy

6.2. Performance Evaluation

6.3. Cost Analysis

6.4. Requirements Fulfillment

CHAPTER 7

Conclusion

7.1. Dissertation Summary

7.2. Answering the Research Questions

[Questões ou hipóteses? Neste caso seria apenas uma.]

7.3. Academic Contributions

[Definir as contribuições académicas. O que traz de novo? Que lacunas preenche?]

7.4. Limitations

7.5. Future Work

References

- [1] Naga Mallika Gunturu. “Enterprise API transformation: Driving towards API economy”. In: *arXiv preprint arXiv:2304.05322* (2023).
- [2] Alexis Huf and Frank Siqueira. “Composition of heterogeneous web services: A systematic review”. In: *Journal of Network and Computer Applications* 143 (2019), pp. 89–110.
- [3] Scott Wang and Johannes Kern. “Digitalization Solutions in the Competitive CEP Industry – Experiences from a Global Player in China”. In: *The Digital Transformation of Logistics: Demystifying Impacts of the Fourth Industrial Revolution*. Wiley-IEEE Press, 2021, pp. 97–112. DOI: 10.1002/9781119646495.ch7.
- [4] Mac Sullivan, Dennis Wong, and Zheyuan Tang. “The Evolution of Freight Forwarding Sales”. In: *The Digital Transformation of Logistics: Demystifying Impacts of the Fourth Industrial Revolution*. Wiley-IEEE Press, 2021, pp. 329–344. DOI: 10.1002/9781119646495.ch23.
- [5] Florinë MUSHICA and Agon MEMETI. “A COMPREHENSIVE ANALYSIS OF ARCHITECTURAL PATTERNS IN ASP.NET CORE WEB APPLICATION DEVELOPMENT”. In: *Journal of Natural Sciences and Mathematics of UT-JNSM* 9 (17-18 Oct. 2024), pp. 207–218. ISSN: 25454072. DOI: 10.62792/ut.jnsm.v9.i17-18.p2816. URL: <https://journals.unite.edu.mk/Abstract?AId=1209&DId=2816>.
- [6] Dongping Song. “A Literature Review, Container Shipping Supply Chain: Planning Problems and Research Opportunities”. In: *Logistics* 5 (2 June 2021). ISSN: 23056290. DOI: 10.3390/logistics5020041.
- [7] Ruben Huber. “The Digital Transformation of Freight Forwarders”. In: *The Digital Transformation of Logistics: Demystifying Impacts of the Fourth Industrial Revolution*. Wiley-IEEE Press, 2021, pp. 153–167. DOI: 10.1002/9781119646495.ch11.
- [8] Sheng Teng Huang, Emrah Bulut, and Okan Duru. “Service quality evaluation of international freight forwarders: an empirical research in East Asia”. In: *Journal of Shipping and Trade* 4 (1 Dec. 2019). DOI: 10.1186/s41072-019-0053-6.
- [9] Ken Peffers et al. “A design science research methodology for information systems research”. In: *Journal of Management Information Systems* 24 (Jan. 2007), pp. 45–77.
- [10] Mairon Freight. *Which Factors Influence Freight Rates?* (accessed: 13 Feb. 2025). URL: <https://mairon.co.uk/which-factors-influence-freight-rates/>.

- [11] WebCargo. *Shipping Quote*. <https://www.freightos.com/glossary/shipping-quote/>. (accessed: 19 Aug. 2025). 2023.
- [12] ITF. *ITF Transport Outlook 2023*. May 2023. DOI: 10.1787/b6cc9ad5-en. URL: https://www.oecd-ilibrary.org/transport/itf-transport-outlook-2023_b6cc9ad5-en.
- [13] Brenda Jin, Saurabh Sahni, and Amir Shevat. *Designing Web APIs BUILDING APIS THAT DEVELOPERS LOVE*. O'Reilly, 2018. URL: www.wowebook.org.
- [14] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. "An Analysis of Public REST Web Service APIs". In: *IEEE Transactions on Services Computing* 14 (4 July 2021), pp. 957–970. ISSN: 19391374. DOI: 10.1109/TSC.2018.2847344.
- [15] Ach Khozaimi et al. "DESIGN AND DEVELOPMENT OF BACKEND APPLICATION FOR THESIS MANAGEMENT SYSTEM USING MICROSERVICE ARCHITECTURE AND RESTFUL API". In: 11 (4 2022). ISSN: 2301-6914. DOI: 10.21107/kursor.v11i4.313.
- [16] Arnaud. Lauret. *Design of Everyday APIs*. Manning Publications Company, 2019, pp. 299–304. ISBN: 1617295108.
- [17] Hatma Suryotrisongko, Dedy Puji Jayanto, and Aris Tjahyanto. "Design and Development of Backend Application for Public Complaint Systems Using Microservice Spring Boot". In: *Procedia Computer Science*. Vol. 124. Elsevier B.V., 2017, pp. 736–743. DOI: 10.1016/j.procs.2017.12.212.
- [18] Ashish Gupta, Meenakshi Panda, and Anoop Gupta. "Advancing API Security: A Comprehensive Evaluation of Authentication Mechanisms and Their Implications for Cybersecurity". In: *International Journal of Global Innovations and Solutions (IJGIS)* (2024). <https://ijgis.pubpub.org/pub/7drcnfbc>.
- [19] Murilo Góes de Almeida and Edna Dias Canedo. "Authentication and Authorization in Microservices Architecture: A Systematic Literature Review". In: *Applied Sciences (Switzerland)* 12 (6 Mar. 2022). ISSN: 20763417. DOI: 10.3390/app12063023.
- [20] I. H. Madurapperuma, M. S. Shafana, and M. J. A. Sabani. "State-of-art frameworks for front-end and back-end web development". In: *2nd International Conference - 2022 (ICST2022) Proceedings on "Building Sustainable Future Through Technological Transformation"*. Faculty of Technology, South Eastern University of Sri Lanka, Sri Lanka, 2022, pp. 62–67.
- [21] Stack Overflow. *2024 Developer Survey: Technology*. <https://survey.stackoverflow.co/2024/technology>. Accessed: 10 Feb. 2025. 2024.
- [22] Microsoft. *ASP.NET Core Documentation*. (accessed: 11 Feb. 2025). URL: <https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0>.
- [23] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybylek. "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10 (2022), pp. 20357–20374. ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3152803.

- [24] Sakib Haque et al. “Performance Evaluation of Back-End Frameworks: A Comparative Study”. In: *IEEE International Conference on Program Comprehension*. Vol. 2022-March. IEEE Computer Society, 2022, pp. 36–47. ISBN: 9781450392983. DOI: 10.1145/nnnnnnnn.nnnnnnnn.
- [25] John Ciliberti. “ASP.NET Core MVC Fundamentals”. In: *ASP.NET Core Recipes: A Problem-Solution Approach*. Berkeley, CA: Apress, 2017, pp. 1–42. DOI: 10.1007/978-1-4842-0427-6_1. URL: https://doi.org/10.1007/978-1-4842-0427-6_1.
- [26] Oliver Karlsson. *A Performance comparison Between ASP.NET Core and Express.js for creating Web APIs*. 2021.
- [27] Dominik Choma, Kinga Chwaleba, and Mariusz Dzieńkowski. “THE EFFICIENCY AND RELIABILITY OF BACKEND TECHNOLOGIES: EXPRESS, DJANGO, AND SPRING BOOT”. In: *Informatyka, Automatyka, Pomiar w Gospodarce i Ochronie Srodowiska* 13 (4 2023), pp. 73–78. ISSN: 23916761. DOI: 10.35784/iapgoss.4279.
- [28] Mozilla Developer Network. *Server-side web frameworks*. (accessed: 7 Feb. 2025). URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Web_frameworks.
- [29] Oleh Zanevych. “ADVANCING WEB DEVELOPMENT: A COMPARATIVE ANALYSIS OF MODERN FRAMEWORKS FOR REST AND GRAPHQL BACK-END SERVICES”. In: *Grail of Science* (37 Mar. 2024), pp. 216–228. ISSN: 2710-3056. DOI: 10.36074/grail-of-science.15.03.2024.031.
- [30] Songtao Chen et al. “Django Web Development Framework: Powering the Modern Web”. In: *American Journal of Trade and Policy* (2017). ISSN: 2313-4755.
- [31] Luong Anh Tuan Nguyen et al. “Design and Implementation of Web Application Based on MVC Laravel Architecture”. In: *European Journal of Electrical Engineering and Computer Science* 6 (2022), pp. 23–29. DOI: 10.24018/ejece.2022.6.4.448. URL: <https://www.ejece.org/index.php/ejece/article/view/448>.
- [32] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. “Architectural Patterns for Microservices: A Systematic Mapping Study”. In: Mar. 2018. DOI: 10.5220/0006798302210232.
- [33] Kendrick Adrio et al. “Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway on Web-based application using GraphQL API”. In: *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE. 2023, pp. 654–660.
- [34] Vasileios Moysiadis et al. “A Cloud Computing web-based application for Smart Farming based on microservices architecture”. In: *2022 11th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE. 2022, pp. 1–5.
- [35] João Figueira and Carlos Coutinho. “Developing self-adaptive microservices”. In: *Procedia Computer Science*. Vol. 232. Elsevier B.V., 2024, pp. 264–273. DOI: 10.1016/j.procs.2024.01.026.

- [36] Kenan CEBECİ and Ömer KORÇAK. “Design of an Enterprise Level Architecture Based on Microservices”. In: *Bilişim Teknolojileri Dergisi* 13 (4 Oct. 2020), pp. 357–371. ISSN: 1307-9697. DOI: 10.17671/gazibtd.558392.
- [37] Chanwoo Yoo et al. “Migrating Monolithic Web Applications to Microservice Architectures Leveraging Use Case and Component Similarity”. In: *Journal of information and communication convergence engineering* 23 (2 June 2025), pp. 78–87. DOI: 10.56977/jicce.2025.23.2.78.
- [38] Mario Villamizar et al. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.
- [39] Samira Khalfaoui, Hafida Khalfaoui, and Abdellah Azmani. “Microservices-Driven Automation in Full-Stack Development: Bridging Efficiency and Innovation With FSMicroGenerator”. In: *IEEE Access* 13 (2025), pp. 125131–125156. ISSN: 21693536. DOI: 10.1109/ACCESS.2025.3589285.
- [40] Nordic Council of Ministers. *Nordic Public Sector Cloud Computing-a Discussion Paper*. Nordic Council of Ministers, 2012. DOI: 10.6027/TN2011-566.
- [41] Muhammad Uzair Nadeem et al. “Cost Analysis of Running Web Application in Cloud Monolith, Microservice and Serverless Architecture”. In: *Journal of Independent Studies and Research Computing* 22 (2 Dec. 2024). ISSN: 24120448. DOI: 10.31645/JISRC.24.22.2.7. URL: <https://jisrc.szabist.edu.pk/ojs/index.php/jisrc/article/view/218>.
- [42] Mario Villamizar et al. “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures”. In: *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*. Institute of Electrical and Electronics Engineers Inc., July 2016, pp. 179–182. ISBN: 9781509024520. DOI: 10.1109/CCGrid.2016.37.
- [43] Google. *PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different?* (accessed: 16 Feb. 2025). URL: <https://cloud.google.com/learn/paas-vs-iaas-vs-saas?hl=en>.
- [44] John Berry. “Logistics in the Cloud-Powered Workplace”. In: *The Digital Transformation of Logistics: Demystifying Impacts of the Fourth Industrial Revolution*. Wiley-IEEE Press, 2021, pp. 129–146. DOI: 10.1002/9781119646495.ch9.
- [45] Tanweer Alam. “Cloud Computing and its role in the Information Technology”. In: *IAIC Transactions on Sustainable Digital Innovation (ITSDI)* 1 (2 Feb. 2020), pp. 108–115. DOI: 10.34306/itsdi.v1i2.103.
- [46] Sanjay Hardikar, Pradeep Ahirwar, and Sameer Rajan. “Containerization: Cloud Computing based Inspiration Technology for Adoption through Docker and Kubernetes”. In: *Proceedings of the 2nd International Conference on Electronics and*

Sustainable Communication Systems, ICESC 2021. Institute of Electrical and Electronics Engineers Inc., Aug. 2021, pp. 1996–2003. ISBN: 9781665428675. DOI: 10.1109/ICESC51422.2021.9532917.

- [47] Amit M. Potdar et al. “Performance Evaluation of Docker Container and Virtual Machine”. In: *Procedia Computer Science*. Vol. 171. Elsevier B.V., 2020, pp. 1419–1428. DOI: 10.1016/j.procs.2020.04.152.