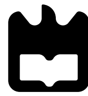
Universidade de Aveiro
2023

Paulo Sérgio Maravilha Gasalho

Sistema de Gestão de Múltiplos Destinos baseado em micro-serviços

Micro-service-based Multi Destination Management System



Universidade de Aveiro
2023

Paulo Sérgio Maravilha Gasalho

Sistema de Gestão de Múltiplos Destinos baseado em micro-serviços

Micro-service-based Multi Destination Management System

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Osvaldo Rocha Pacheco, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e da Doutora Zélia Maria de Jesus Breda, Professora auxiliar do Departamento de Economia, Gestão, Engenharia Industrial e Turismo da Universidade de Aveiro.

o júri / the jury

presidente / president	Professor Doutor Carlos Manuel Azevedo Costa Professor Associado com Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro
vogais / examiners committee	Professor Doutor Miguel Angel Guevara López Professor Adjunto do Instituto Politécnico de Setúbal - Escola Superior de Tecnologia de Setúbal
	Professor Doutor Osvaldo Manuel da Rocha Pacheco Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Agradeço desde já aos meus orientadores, ao Professor Doutor Osvaldo Pacheco e à Professora Doutora Zélia Breda pela paciencia e pelo incansável apoio prestado ao longo de todo este processo.
Aos meus pais e avós que tanto me apoiaram ao longo do curso e que viveram os meus sucessos e insucessos com tanta intensidade, acreditando sempre em mim.
A Carolina, por me ter apoiado e ajudado nas fases mais criticas da realização desta dissertação.
A todos os amigos que fiz ao longo do curso e que levo agora para a vida.
A todos, o meu muito obrigado.

Palavras Chave

API, Aplicações WEB, Arquitetura Baseada em Microsserviços, Bases de Dados PostgresSQL, Destinos Turísticos, Sistema de Informação, Sistemas de Gestão de Destinos

Resumo

Atualmente, os destinos turísticos recorrem a Sistemas de Gestão de Destinos como forma de divulgarem junto da procura turística as ofertas turísticas que têm disponíveis nas suas regiões. No entanto, o que ocorre é que a adoção destes sistemas de informação é ainda muito baixa, o que se deve essencialmente aos elevados custos da sua implementação, pois atualmente é necessário criar um sistema deste tipo completamente de raiz. Esta dissertação teve assim como principal objectivo provar que é possível desenvolver uma plataforma que permita criar sistemas de gestão de destinos de forma muito mais rápida e económica, através da construção de uma infraestrutura partilhada por vários destinos, onde seria possível reutilizar ao máximo os componentes de software de um SGD. O resultado foi a criação de um plataforma composta por um módulo de gestão, onde para além de ser possível criar e gerir SGD, permite ainda que os gestores de produtos turísticos anunciem os seus serviços. Além disso foi também criado um portal demonstrador que simula a oferta de um destino turistico, com o intuito de validar o sistema criado. Em termos tecnológicos, é de destacar o facto de que o Backend assenta numa arquitetura baseada em microserviços que permite uma rápida escalabilidade, extensibilidade e eficiência energética que são objetivos chave para o sucesso deste projeto.

Keywords

API, Architecture Based on Microservices, Destination Management Systems, Information Systems, PostgreSQL Databases, Touristic Destinations, WEB Applications

Abstract

Nowadays, tourist destinations use Destination Management Systems as a way of publicizing tourist demand and the tourist offers they have available in their regions. However, the adoption of these information systems is still very low, essentially due to the high costs of implementing them, as it is currently necessary to create a system of this type completely from scratch. The main aim of this dissertation was, therefore, to prove that it is possible to develop a platform that allows destination management systems to be created much more quickly and economically by building an infrastructure shared by several destinations, where it would be possible to reuse the software components of a DMS as much as possible. The result was the creation of a platform comprising a management module, where, as well as being able to create and manage DMSs, it also allows tourism product managers to advertise their services. A demonstrator portal was also created to simulate the offer of a tourist destination to validate the system created. In technological terms, it is worth highlighting that the Backend is based on an architecture based on microservices, which allows for rapid scalability, extensibility, and energy efficiency, which are critical objectives for the success of this project.

Contents

Contents i

List of Figures iii

List of Tables v

Acronyms vii

1 Introduction 1

1.1 Motivation 1

1.2 Goals 2

1.3 Document Structure 2

2 State of Art 5

2.1 Introduction 5

2.2 Software Architectures 5

2.3 Web Development Architectures 9

2.4 Database Server 10

2.4.1 Relational Database 10

2.5 Application Server 12

2.5.1 Spring Boot 12

2.6 Web Server 12

2.6.1 React 12

2.7 Related Projects 13

2.7.1 Visit Alentejo 13

2.8 Conclusion 17

3 System Requirements and Architecture 19

3.1 Introduction 19
3.2 System Requirements 20
3.2.1 Requirements Gathering Process 20
3.2.2 Actors 21
3.2.3 Functional Requirements 22
3.2.4 Non Functional Requirements 26
3.2.5 General Data Protection Regulation 26
3.3 System Architecture 27
3.3.1 Physical and Technological model 27
3.3.2 Domain model 28
3.4 Conclusion 30

4 Implementation 31
4.1 Introduction 31
4.2 API Gateway and Discovery Server 31
4.2.1 API Gateway 31
4.2.2 Discovery Server 33
4.3 Authentication 34
4.4 Business Microservices 36
4.5 Frontend/Backend connection 44
4.6 Conclusion 45

5 Results and Discussion 47
5.1 Introduction 47
5.2 Portal and Management System: Prototypes 47
5.2.1 Portal 47
5.2.2 Management 51
5.3 System Performance 55
5.3.1 Load Test 56
5.4 Conclusion 59

6 Conclusions and Future Work 61
6.1 Final Considerations 61
6.2 Future work 61

References 63

List of Figures

2.1 Layered Architecture. Figure from [8] 6
2.2 Event-Driven Architecture. Figure from [8] 7
2.3 MicroKernel Architecture. Figure from [8] 8
2.4 MicroServices Architecture. Figure from [8] 9
2.5 3-tier Client-Server Architecture. Figure from [12] 10
2.6 Visit Alentejo Homepage 14
2.7 Visit Alentejo - Where to sleep 15
2.8 Visit Alentejo - Hotel Detail 16
2.9 Visit Alentejo - Events 17

3.1 Multi-Destination Management System - Proposed Solution 20
3.2 Management Module - Use Case Diagram 23
3.3 Portal Module - Use Case Diagram 25
3.4 Physical and Technological Model 28
3.5 Domain Model 29

4.1 Spring Cloud Gateway Dependency 32
4.2 Api Gateway Routes 32
4.3 Cross-Origin Resource Sharing (CORS) Configurations 33
4.4 Eureka Server Dependency 33
4.5 Eureka Server Configurations 33
4.6 Eureka Client Dependency 34
4.7 Eureka Client Configurations 34
4.8 Authentication Dependencies 34
4.9 Accomodation Microservice Dependencies 37
4.10 Accomodation Microservice Packages 38

5.1 Aveiro Region Portal - Main Page 48

5.2 Aveiro Region Portal - More Detail about a Touristic Product 49
5.3 Aveiro Region Portal - Do a Reservation of a Touristic Product 49
5.4 Aveiro Region Portal - Do a Rating of a Touristic Product 50
5.5 Aveiro Region Portal - My Ratings Page 50
5.6 Aveiro Region Portal - My Reservations Page 51
5.7 Multi-Destination Management System - Main Page 52
5.8 Multi-Destination Management System - My Touristic Product Reservations 52
5.9 Multi-Destination Management System - Activities/Events List 53
5.10 Multi-Destination Management System - Create Restaurant/Bar Advert 53
5.11 Multi-Destination Management System - Locations List 54
5.12 Multi-Destination Management System - Create Location 54
5.13 Multi-Destination Management System - Destinations List 55
5.14 Multi-Destination Management System - Create Destination 55
5.15 JMeter - Test Plan 57
5.16 JMeter - Thread Properties 57

List of Tables

4.1 Authorization Representational State Transfer (REST) API endpoints 36
4.2 Restaurant-Bar Service REST API endpoints 39
4.3 Accomodation Service REST API endpoints 40
4.4 Non Touristic Service REST API endpoints 40
4.5 Activities-Events Service REST API endpoints 41
4.6 Rating Service REST API endpoints 42
4.7 Reservation Service REST API endpoints 42
4.8 Destination/Location Service - Destination REST API endpoints 43
4.9 Destination/Location Service - Location REST API endpoints 43
4.10 Portal - Needed Endpoints By Microservice 44
4.11 Management - Needed Endpoints By Microservice 45

5.1 JMeter - Portal Modules Test Results 58
5.2 JMeter - Management Module Test Results 58

Acronyms

API Application Programming Interface
CORS Cross-Origin Resource Sharing
CRUD Create, Read, Update and Delete
CSS Cascading Style Sheets
DMOs Destination Management Organizations
DMS Destination Management System
DTO Data Transfer Object
HTTP Hypertext Transfer Protocol
HTML HyperText Markup Language
JDBC Java Database Conectivity
JSON Javascript Object Notation
JWT JSON Web Token
MVC Model-View-Controler
ORDBMS Object-Relational Database Management System
RBAC Role Based Access Control
REST Representational State Transfer
RDBMS Relational Database Management System
SQL Structured Query Language
UI User Interface

Chapter 1

Introduction

1.1 Motivation

Today, tourism plays a vital role in the economy of several countries and is even their most significant source of income. However, this activity is not neglected even by governments with lower tourist demand [1]. Therefore, it is in the general interest that the tourist attractions of a given region or place are well publicized to reach as many people as possible.

Thus, those responsible for the management of a tourist destination, which, according to Bunghez [1], consists of a "place or geographical space where a visitor or tourist stops for a night or a period, or the end of a tourist's vacation, whether traveling for tourist or business reasons" are beginning to take an interest in online platforms where they provide tourist demand with the possibility of planning their trips and promoting their tourism products. [2]

These platforms consist of the Destination Management System (DMS), which, according to the literature, can be described as a collection of computerized information about a destination, accessible in an interactive way [3], and which act as a link between the Destination Management Organizations (DMOs) and the tourist companies located in the destination, allowing the DMOs to coordinate their operations and manage their relationships [4]. These are usually driven by the DMOs and can be considered synonymous with the technological infrastructure of an DMOs [5], which can be private, public, or public-private sector.

However, the adoption of these systems by destinations is still very low, which is essentially due to the high implementation costs and the delay in development resulting from the technological difficulties that have to be overcome [6], which makes it hard for many small destinations with little financial capacity, to join this type of innovation. This is because currently, a destination that wants to implement a solution of this type needs to create at least two modules: a portal module that visitors use to the destination and that consists of the means of dissemination intended by the destination and a management module where those who manage the destination announce the tourist activities that will appear on the portal. In addition, it is necessary to create all the software components from scratch for both modules, from the database to the backend and finally to the Frontend, and also to host this entire system on a physical server or in its cloud with all the costs that this entails.

1.2 Goals

This dissertation has several objectives, the main one being to prove that it is possible to develop a platform that allows DMS to be created much more quickly and economically, building an infrastructure shared by the various destinations. It should, therefore, be possible to reuse the software components for both the portal module and the management module as much as possible, making it possible to create a new DMS for a given destination by creating only a customized front end for the destination portal, with all the remaining components being generic and usable in multiple destinations.

In addition, it is also necessary to ensure that this solution makes it possible to add new functionalities or support new tourism products, thus making it possible to satisfy the needs of as many tourist destinations as possible without breaking compatibility with existing functionalities and assessing whether the basic functionalities that will be implemented initially satisfy the requirements of the system's main stakeholders.

As stated above, one of the main objectives is to reduce the costs of creating and maintaining a DMS, so this solution could prove particularly interesting for low-density territories, which often cannot access this type of innovation due to budget constraints and will therefore find it easier to adopt a system of this type.

However, it must also be ensured that a Multi-Destination Management System of this type performs similarly to a system made from scratch for a given destination. From the point of view of a visitor to a given region, it is seen as an isolated system for each destination.

From the point of view of software architecture, it is also the aim of this dissertation to prove that an architecture based on microservices makes it possible to provide a fast, stable, and efficient response to the objectives mentioned above.

Finally, to fulfill these objectives, a prototype of a Multi-Destination Management System was developed with a management module shared between the various destinations and two portal modules representing two different tourist destinations. This prototype was validated from a performance point of view using load tests.

1.3 Document Structure

This document is, therefore, structured as follows.

Chapter 2 presents some aspects of software engineering, such as the most commonly used software architectures and how the web works, demonstrating its typical structure and presenting each layer. It then shows one of the DMS analyzed to make the prototype for this dissertation.

Chapter 3 mentions the requirements needed to develop the system and also presents the architecture created to achieve the objectives above.

Chapter 4 details the implementation behind this solution, namely some relevant aspects of the system's backend and how the Frontend consumes the endpoints provided by Application Programming Interface (API).

Chapter 5 demonstrates the implemented solution and validates its performance using system load tests.

This document ends in Chapter 6, where some final considerations are made about the solution, and some points for future improvement are highlighted.

Chapter 2

State of Art

2.1 Introduction

This chapter will explore some relevant aspects of the literature, both from a software engineer's point of view and from the point of view of the destination management technologies that tourism currently uses, to design a truly innovative system that fulfills the objectives set for this dissertation.

From the point of view of software engineering, some aspects relevant to the development of a software solution will be studied, such as software architecture, and some of the most widely used architectures will be presented, as well as their strengths and weaknesses. In addition to this, following the typical structure of a web application, it will also be explained what a relational database consists of. Three of the best-known Relational Database Management System (RDBMS) will be presented, and a performance comparison will be given. It will detail what the Backend of an application consists of, three well-known frameworks will be presented, and the advantages and disadvantages of each framework will be compared. Finally, it will describe what the Frontend of an application consists of, giving and comparing three frameworks widely used for this purpose.

From the point of view of the destination management technologies that the tourism sector currently uses, three DMS used by three different tourist destinations will be presented, showing the main functionalities of each one and the possible flaws each has.

2.2 Software Architectures

A key point in the design and construction of complex software systems is their architecture. Good architecture helps ensure that the system meets essential requirements in performance, reliability, portability, scalability, and interoperability. Bad architecture can have dire consequences [7].

Thus, it is important to study various types of architectural patterns, such as the following:

Layered Architecture. A layered architecture pattern is a software architecture pattern that organizes an application into several hierarchical layers, each having precise obligations.

Companies traditionally use this pattern as it allows a division of the software to be produced by teams.

The layers of a layered framework typically consist of a presentation layer, a business logic layer, and a records layer, as shown in Figure 2.1. The presentation layer is responsible for processing the data entered by the user and displaying the results to the person. The business logic layer contains the business rules of the software and manages data processing and auditing. The persistence layer is responsible for storing and retrieving the information.

One of the advantages of this architecture is its modularity since changes in one layer no longer affect the other layers, which facilitates installation, verification, and scalability.

However, this type of layered architecture consists of a monolithic architecture and, therefore, has more difficulty scaling than other solutions. Even though it can be scaled by splitting the layers into separate physical deployments and/or creating particular application instances in multiple virtual machines, it becomes costly and inefficient because all the application functionality has to be scaled. Also, the layered architecture is not very fault tolerant - a fatal failure anywhere in the application brings down the entire functionality of the application [8].

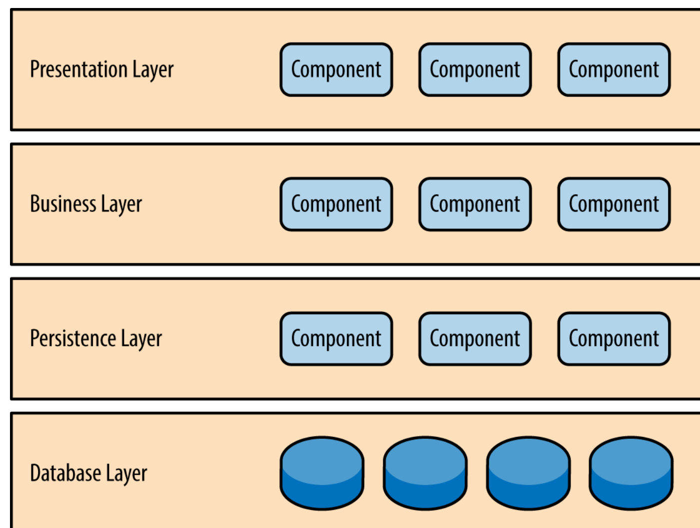


Figure 2.1: Layered Architecture. Figure from [8]

Event-Driven Architecture. An event-driven architecture (EDA) is a software architecture pattern in which applications talk to each other through events asynchronously and decoupled, reducing dependencies and enabling greater flexibility and scalability. This pattern is particularly beneficial in complicated systems, including financial or healthcare industries, where information is constantly changing, and real-time processing is required.

In this architecture, the publish-subscribe topology is used, where the events are sent to all the subscribed components, which receive and process the events independently, as seen in Figure 2.2.

One of the advantages of this architecture is its performance since event-driven systems can handle large amounts of data and demanding processing requirements.

However, this architecture can still introduce some complexity, as event control and processing require additional infrastructure and coordination. Furthermore, the asynchronous nature of the architecture can make it difficult to guarantee the consistency of information and the integrity of transactions. Thus, this type of architecture is not recommended if its processing is request-based, where it is common for the user to request data from a database or perform basic Create, Read, Update and Delete (CRUD) operations on system entities [8].

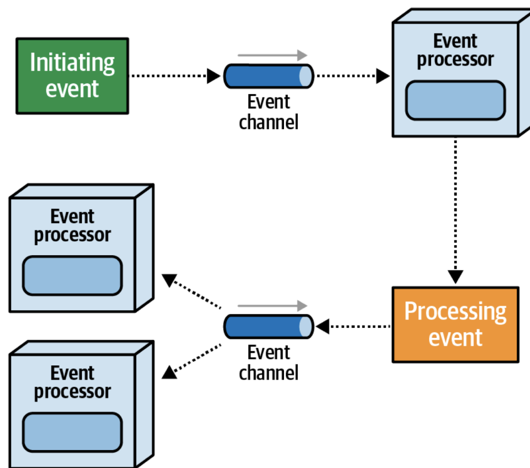


Figure 2.2: Event-Driven Architecture. Figure from [8]

MicroKernel Architecture. A MicroKernel architecture pattern is a software architecture pattern in which a piece of software is divided into a central system, called the core, and a set of optional plug-in modules that increase its functionality.

In this architecture, the core provides the software's primary infrastructure and functionality. In contrast, the plug-in modules provide additional capabilities and capacities, as shown in Figure 2.3. Plug-in modules can thus be developed independently, speeding up development times for improvements and validation.

However, although this architecture has some flexibility since plug-in modules can be added or removed as desired, all requests have to go through the core, which causes some bottlenecks at this point in the software, making this architecture not very elastic and scalable. In addition, it is also not fault-tolerant due to the kernel because a failure in the kernel affects the entire system [8].

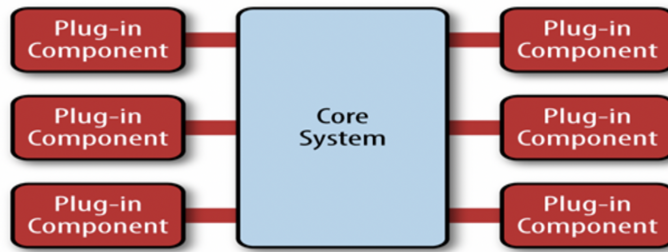


Figure 2.3: MicroKernel Architecture. Figure from [8]

Microservices Architecture. Microservices architecture is a software architecture pattern in which an application is composed of a set of small independent services that can by themselves answer requests or communicate with each other through Hypertext Transfer Protocol (HTTP) requests. These services are, however, designed to implement particular business domains and can be developed, tested, and implemented independently, allowing for greater flexibility and scalability [9].

Furthermore, this type of architecture allows microservices to be developed using a variety of programming languages and technologies, allowing programmers to choose the most appropriate tools for each service.

One of the most significant advantages of this architecture is its scalability, as microservices can be easily scaled up or down as required to deal with varying levels of traffic and processing requirements. In addition, the modular nature of the architecture allows for easier maintenance and updates, as each service can be updated independently without affecting the rest of the application.

However, this type of pattern can introduce some complexity in development as services must be designed and managed carefully to ensure they are correctly integrated and compatible, and also, in the case where they are distributed, it can become more difficult to ensure data consistency and maintain transactional integrity. Although these difficulties exist, it is possible to minimize them by choosing a development framework suitable for such an architecture, with good documentation and a high adoption by the community [8].

In an architecture of this type, the number of services often scales quickly, and therefore, it becomes very complex for the clients to manage which service to request at any given time. Thus, API Gateways are typically used in this type of architecture, which are nothing more than a particular service that works as the only gateway to the application, encapsulating internal aspects of the implementation and also supporting essential functionalities such as permissions verification, Load balancing, caching of requests and system monitoring.

The main difficulty in implementing API Gateway lies in its ability to handle high simultaneous requests and performance requirements, which can be overcome by using more efficient load balancing algorithms and in the limit by using multiple API Gateways if the application needs it [10].

Another point that is a common characteristic in this type of architecture is the use of a separate database for each of the microservices, as can be seen in 2.4. However, this does not necessarily have to be the case, and there are even implementations with a single database for all microservices or no single database, with some databases being shared with several services. Here, the responsibility lies with the software architect, who will evaluate the application's needs [8].

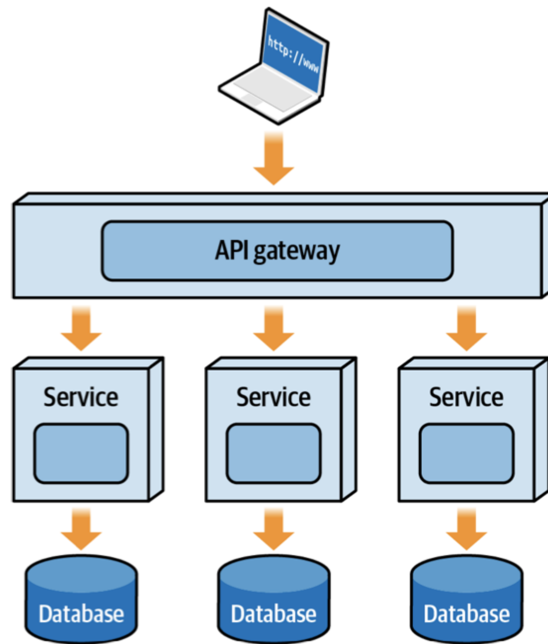


Figure 2.4: MicroServices Architecture. Figure from [8]

2.3 Web Development Architectures

As seen above, there are multiple architectures to guide software architects in software development. Still, depending on the type of software to be implemented, some are more suitable than others. Web applications use an architectural pattern called the client-server pattern. In this pattern, there can be applications with multiple layers, but in web applications, the most commonly used are three layers of communication:

- The web server, also called the presentation layer, is responsible for presenting data to the client, receiving their requests, and returning responses. This layer usually runs in a web browser and can be developed using HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and Javascript. This layer is often called the application's Frontend.
- The application server, also known as the logic layer, is the heart of the application and is responsible for implementing the application's business logic used to process user requests, often cross-referenced with information from the database server to return a result to the user. This layer is typically implemented using languages such as Java, Python, or PHP and communicates with the web server layer via API calls. This layer is often called the application's Backend.

- The database server is the application's data layer and is where the information processed is stored and managed. There are various types of relational database systems, which will be discussed in the next section [11].

This architecture can be visualized in detail in the Figure 2.5.

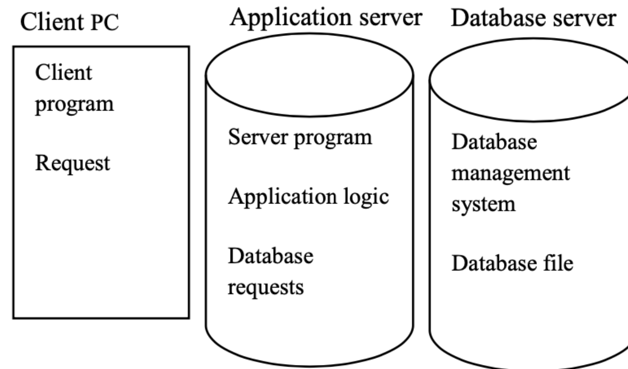


Figure 2.5: 3-tier Client-Server Architecture. Figure from [12]

To ensure communication between layers, there are various communication protocols, but the most widely used is the HTTP [12].

It is also important to note that computer science has developed various tools that simplify the implementation of the various layers that this standard has, namely the application's frontend and backend layers, which are called frameworks.

2.4 Database Server

2.4.1 Relational Database

A relational database is a type of database where data has well-defined relationships with each other. In this model, the information is stored in tables, which represent the entities of the information system, and the columns of these tables represent the attributes of these entities. In each row of these tables, there is a unique identifier (primary key), and there may also be identifiers for other tables, thus establishing the relationship (foreign keys) in addition to the data itself.

Relational databases are often considered transactional and, therefore, characterized by the implementation of the properties Atomicity, Consistency, Isolation, and Durability (ACID), being mostly chosen by programmers looking for these characteristics:

- Atomicity - all changes to the data associated with a transaction are performed as if it were a single operation.
- Consistency - data is kept consistent at the beginning and end of a transaction.
- Isolation - The effect of a transaction is kept invisible to others until it is confirmed to avoid confusion.
- Durability - upon successful transaction completion, changes to the data are persisted and not canceled, even if the system fails [13].

As we have seen, a relational database deals only with data organization based on a relational model without performing management tasks. For this, there are RDBMS. An RDBMS consists of software that allows the management, consultation, storage, and retrieval of information in a database of this type. A common feature of all RDBMS is the use of the Structured Query Language (SQL). This language is the standard used to interact with relational database management systems, allowing the database administrator to insert, update, or delete rows of data efficiently [14].

Some examples of DBMSs widely used today are PostgreSQL, MySQL, and Microsoft SQL Server, which will be detailed below:

Microsoft SQL Server

Microsoft SQL Server is a RDBMS developed and licensed by Microsoft, created in 1989. Some of the main features of this database management system are support for complex data types, transaction control, advanced indexing, data warehousing, high availability, and various security options [15]. The SQL Server product is primarily a paid solution with Standard and Enterprise versions. There is also a free Express version to be used for small applications [16].

Postgres

PostgreSQL is an Object-Relational Database Management System (ORDBMS), a database management system that manages an object-relational database. An object-relational database turns out to be an extension of relational databases with some features of object-oriented programming languages. So this type of database turns out to integrate better with these. PostgreSQL dates back to 1973 and has been developed and popularized by the open-source community since 1996. Currently, this database management system is used by, among others: Apple, Cisco, Instagram, Spotify, and Skype, among others.

PostgreSQL is ACID (atomicity, consistency, isolation, durability) compliant, is transactional, and includes support for common B-tree indexes and hashes. It has updatable and materialized views, triggers, and foreign keys. It also supports functions (including some NoSQL) and stored procedures [17].

MySQL

MySQL is an open-source RDBMS created in 1995 and currently being maintained by Oracle. This system implements a mechanism that significantly improves query execution speed by storing the last queries and their results in a cache. In addition, this system supports a wide range of data types, including numeric, date/time, character, Javascript Object Notation (JSON), boolean, and enumerated, supports various indexes, such as B-tree, hash, R-tree, and inverted indexes, and provides different encrypted options for access control, offering reliable security. However, although this system does not yet have features as extensive as PostgreSQL, it is still considered an excellent option for various types of Web applications [18].

Performance Comparison

Let's analyze a study conducted at the Lublin University of Technology about the performance of the above databases. We can conclude that PostgreSQL is the database management system with the best performance compared to Microsoft SQL Server and MySQL for a large number of records (more than 1000 records), having a lower performance for a lower number of records (up to 1000 records) to MySQL. This confirms the popularity that MySQL has for simple applications with little scalability and the notoriety that PostgreSQL has for more demanding applications with a more significant need to scale at the persistence level. Combined with this is the fact that PostgreSQL is based on an open-source license, which is a significant point for controlling development costs [16].

2.5 Application Server

2.5.1 Spring Boot

Spring Boot is one of the most widely used backend frameworks in the world, based on another popular Java framework called Spring. Spring Boot's mission is to simplify the creation of web applications by relying on the principles of convention rather than configuration, which allows developers to focus more on business logic and less on server configuration. In addition, this framework is optimized for microservice-based architecture, which allows the programmer to easily implement issues relating to communication between services, discovery of microservices, exposing them to the client, and security of all services [19].

In terms of Spring Boot's strengths, the following stand out:

- Rapid Application Development allows developers to produce production-level applications quickly
- Automatic configuration allows programmers to spend less time since the framework automatically performs configurations such as accessing the database via Java Database Conectivity (JDBC) or recognizing classes in the services.
- Starter POMs which consist of dependency packages that are essential for specific tasks in a microservice, such as the spring-boot-starter-web dependency, which allows you to raise an Model-View-Controler (MVC)-based REST API without the need for additional dependencies.
- Actuator makes it possible to control and manage the state of applications in areas such as health, metrics, or shutdown.
- Integration with other well-known libraries and frameworks such as Hibernate, Spring Cloud, or Spring Data [19].

2.6 Web Server

2.6.1 React

ReactJS is an open-source JavaScript library created by Facebook to build single-page User Interface (UI) that allows you to use data from the backend and update it without reloading the entire web page [20].

Some of the most important points that the ReactJS library implements are:

- Unidirectional information flows from parent components to child components, which simplifies the development and debugging of these applications.
- Virtual DOM to efficiently update and render components whenever changes occur in the application.
- Optimized performance thanks to features such as splitting large components into smaller ones or lazy loading.
- Declarative, since it allows developers to define how they want the interface to look, React performs the operations behind it to realize this [20].

Since ReactJS is only a JavaScript library, it is necessary to create or use a framework already started by a third party that has dependencies that allow routing, data fetching, and HTML generation and that already contains a basic structure that is ready to work with [21].

A prevalent example of these frameworks is Create React App, which Facebook officially supports, and with just one npm command, you can start developing a web application with ReactJS libraries [22].

2.7 Related Projects

In terms of projects related to the aim of this dissertation, according to the research carried out, it was impossible to find any system that, on a single platform, makes it possible to create and manage multiple tourist destinations, sharing resources.

However, it was possible to find some DMS implemented from scratch, which made it possible to gather functional requirements for the Multi-Destination Management System implemented. Thus, DMS were explored at the national level, such as Terras de Trás os Montes and VisitAlentejo, and at the European level, VisitBerlin, and Visit.Brussels.

However, in order not to make this dissertation too extensive, only one system will be described in the following section, which was chosen because it was the most complete and easy to use. VisitAlentejo will be presented next.

2.7.1 Visit Alentejo

The VisitAlentejo website is a DMS run by Turismo do Alentejo and the Alentejo Regional Tourism Promotion Agency, which aims to publicize and promote tourism in the Alentejo region. Its main functions are to advertise tourist attractions such as places to sleep, where to eat, what to do in the region and events, as well as offer the chance to take a virtual tour of some of the region's tourist attractions and allow potential visitors to plan their trip, as can be seen on the main page of the DMS in Figure 2.6.

Figure 2.6: Visit Alentejo Homepage

Suppose you analyze the system's functionalities in more detail. In that case, you'll notice that the "Where to sleep", "Where to eat" and "What to do" pages are similar, all with a list of tourist products by location and the possibility of applying various filters or carrying out a direct search for a particular tourist product. An example of these pages can be found in Figure 2.7 with the "Where to sleep" page.

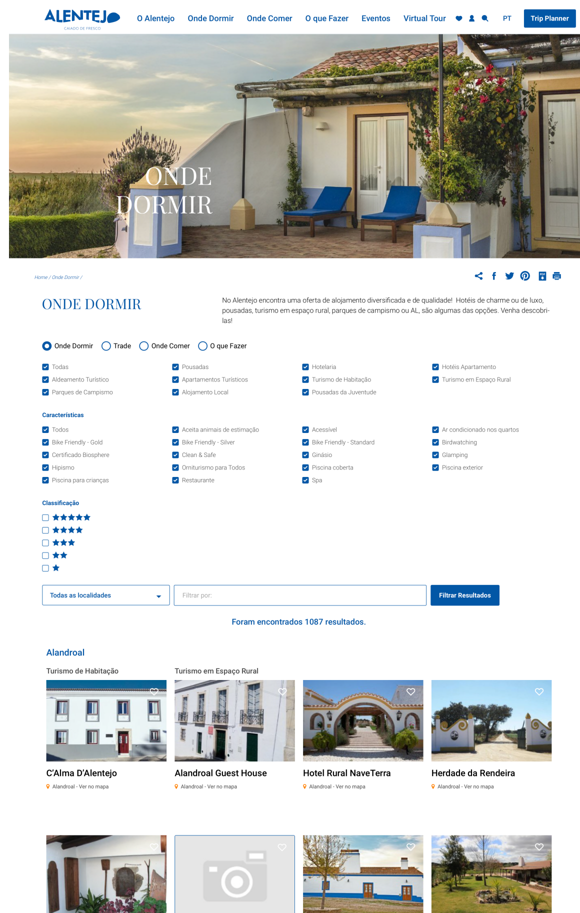


Figure 2.7: Visit Alentejo - Where to sleep

After clicking on an advert, it is possible to obtain more information about a tourism product. Although they vary according to the type of tourism product, there are some common characteristics such as the name, address, contact details, features of the tourism product, and photographs, as can be seen in Figure 2.8, with information about residential tourism.

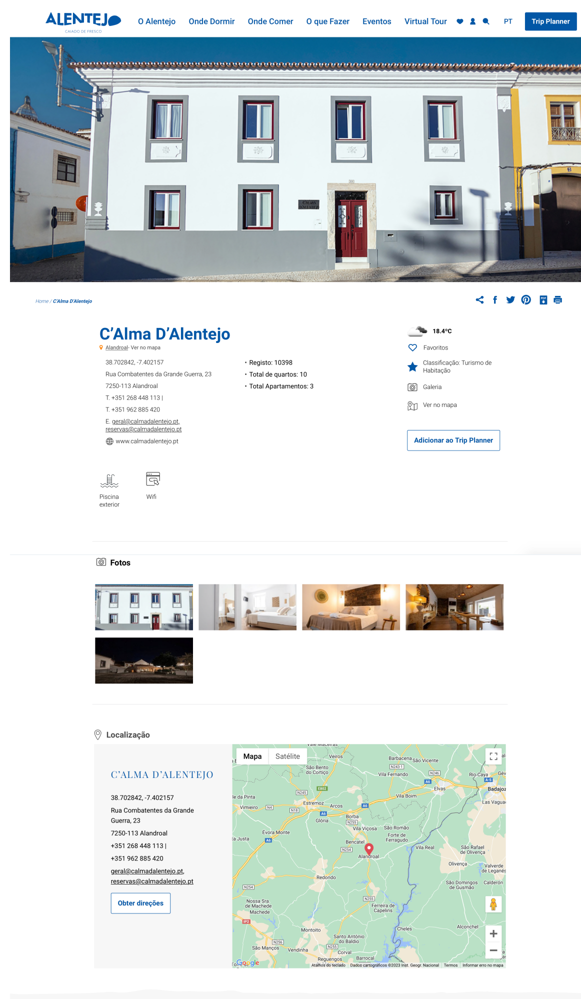


Figure 2.8: Visit Alentejo - Hotel Detail

As previously mentioned, VisitAlentejo also has a list of tourist events in the region, showing the events taking place and those yet to take place, as Figure 2.9 shows.

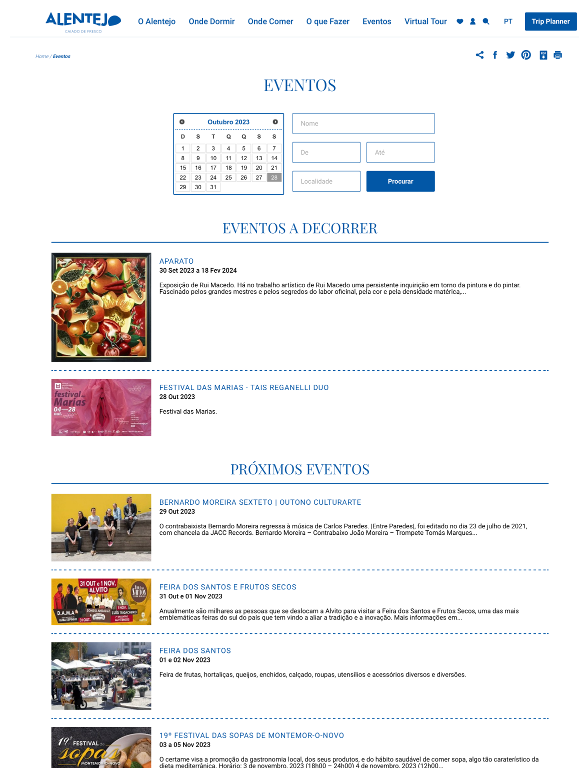


Figure 2.9: Visit Alentejo - Events

To summarise, in terms of VisitAlentejo's strengths, the following should be highlighted:

- The system's main functionalities (Where to sleep, Where to eat, What to do, and Events)
- The possibility of making a virtual visit
- Features presented in each tourism product
- Filters offered to the user in each of the listings

In terms of absences, it is important to mention:

- Inability to book directly through DMS
- Absence of a system of ratings and opinions from other users
- Lack of useful or emergency contacts in each location

2.8 Conclusion

In this chapter, it was possible to understand how the process of choosing a suitable software architecture for creating an information system works and, more specifically, to know how a typical web architecture works. It was also possible to analyze three of the best-known database management systems in terms of performance and thus decide which is most suitable for this dissertation. In addition, more in-depth research was carried out into both the backend framework and the frontend framework used.

Finally, four existing DMS in production were analyzed, and the one considered the most complete and advantageous of the four was examined in greater detail.

Chapter 3

System Requirements and Architecture

3.1 Introduction

As we have seen previously, the scope of this dissertation is much more than developing a DMS for a particular destination, but instead creating a prototype system that demonstrates that it is possible to manage multiple destinations sharing the same resources and infrastructure as much as possible.

This type of solution introduces several advantages over the development of DMS from scratch, such as :

- Reuse of Resources, since the services developed by several destinations can be used without having to implement everything again
- Scalability of the solution, which also allows new functionality to benefit multiple destinations at the same time quickly
- Energy Efficiency, since the various destinations will share the same computational resources, resources can be managed by favoring the destination that needs more at any given time.
- Cost control, because once several destinations reuse the same infrastructure, it is possible to develop DMS at controlled costs, which favors both small and large destinations.

To create the project's Backend, which can be seen in Figure 3.1, the literature was analyzed to find the best software architecture solution. The conclusion was that an Architecture based on Microservices could respond more successfully to scalability, ease of maintenance, replaceability, fault tolerance, and reliability than a monolithic architecture typically used to create DMS from scratch. This meant that the architecture chosen for the Backend of this project was one based on microservices [23].

On the Frontend, there need to be two different types of web applications with which the system users can interact, as we can see in Figure 3.1 and which will be:

- Portal modules used by tourists interested in a particular tourist destination where they can obtain information about restaurants, hotels, tourist activities, and useful contacts about the destination and assign ratings or book some of these tourist products. These

modules will be independent between destinations, i.e., each destination will have its portal module.

- Management Module that will be used by those who exploit these tourism products, which can advertise and publicize their business, and also by system administrators that would act as moderators of the content that would then be available to multiple portals. In this case, there will only be one management module shared by all the portal modules of the Multi-Destination Management System.

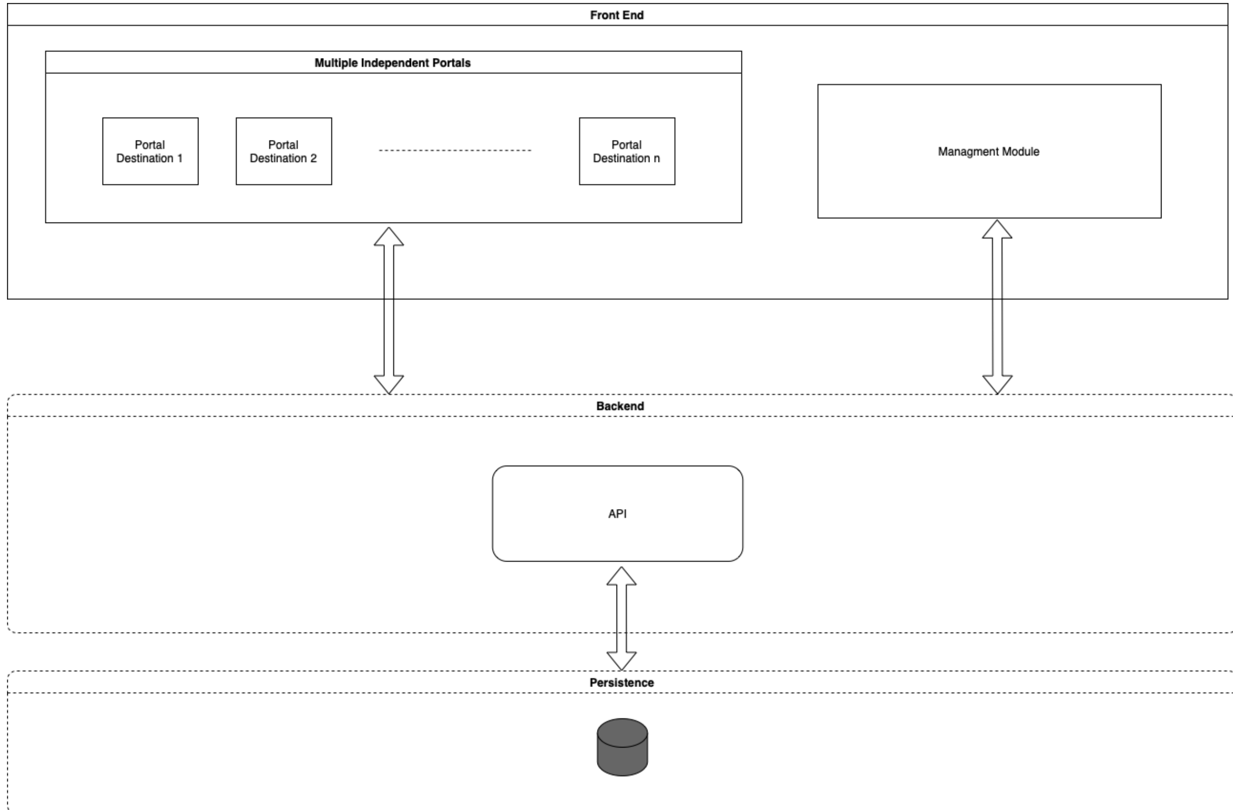


Figure 3.1: Multi-Destination Management System - Proposed Solution

In this chapter, the issues presented here will be more detailed, namely the requirements necessary for the development of the system shown and also the software architecture found to solve the proposed problem successfully.

3.2 System Requirements

This section presents the system requirements specification due to the first phase of system development. The following subsections start with a description of the requirements-gathering process, followed by a description of the context, the actor identification, the use case diagrams, and the non-functional requirements of the solution.

3.2.1 Requirements Gathering Process

In the requirements gathering phase, it was necessary to collect the most pressing needs on the part of tourism agents and compile them most generically and comprehensively possible

for all destinations.

To realize this goal, a literature review was first carried out to understand the primary needs of the DMOs and also what the gaps and limitations of existing DMS were. In addition, an analysis was made of several existing online systems, and the standard functionalities between them were recorded. In the second phase, several meetings were held with an expert in the tourism field to assess whether the points considered essential in the first phase made sense for most destinations. Finally, the requirements raised were divided by the various services, always making them as specific and independent of each other as possible.

Thus, and in a generic way, the following basic requirements were raised:

- Currently, a tourist destination wishing to have a DMS in its region has to carry it out completely from scratch, which requires a very high development cost and a longer time to get into production. Thus, and as we saw earlier, one of the requirements that the Multi-Destination Management System must meet is to be reusable by multiple destinations, being only necessary to create a portal for the destination and taking advantage of the management module and the entire backend that is shared by various destinations.
- Even if the same backend powers the portal modules and uses the same services, it is necessary that the user seems completely united and independent DMS, making it possible to customize the DMS portal according to the preferences of each destination.
- The primary stakeholders in a DMS were defined as the restaurant and nightlife industry, hotels and local accommodations, Adventure, Nature, Beach, Cultural, and Health tourism, and it was also considered essential to inform users of important contacts such as fire brigades, police, and hospitals/health centers. However, it should be possible to add new tourism stakeholders if necessary or to specialize some existing services further by creating new, more specific ones.
- The management of the content published on the platform by the tourism product managers should be simple through a management module with a user-friendly interface available to the platform administrators in charge of this task, which will thus enrich the DMS implemented by the software administrators.
- The system should be fed by any owner/manager of a tourism product, such as a restaurant owner, who should be able to quickly register on the platform and advertise his business, which, after approval by a platform administrator, will be visible to any DMS user.

These generic requirements will be presented in more detail in the following subsections.

3.2.2 Actors

The target users of all the portal modules implemented will be tourists who use the portals of each destination as a source of information about a destination they want to visit or have visited. On the management module side, the primary users will be all tourism product managers who wish to advertise and publicize their business on the destination portal. In addition, the platform administrators will also be users of the management module to perform

management, control, and regulation tasks of the tourism products published by the tourism product managers. They are also responsible for maintaining, testing, and improving the functionalities and performance of the system implemented.

Thus, we can define the actors of the implemented system as represented in the following list:

- Tourists/Visitants

  Tourists/Visitants will be the most numerous users of the platform. They will consult all tourism products available at a given destination; they should be able to access all information on a given product, make a reservation, and, at the end of the experience, rate it.
- Touristic Product Managers

  Touristic product managers will generally be the owners of tourism products of a destination with an interest in advertising their business in the DMS of the destination. They should have access to the management module, be able to publish their tourism products, see the ratings given by users, and manage the reservations made on the platform. They should also have access to the portal module, just like the tourists.
- Platform Administrators

  Platform Administrators will be responsible for creating new supported destinations on the platform, approving or rejecting the advertisements that the tourism product managers make, checking that no unauthorized content is published, and requesting corrections to the advertisements. They will also be responsible for fixing bugs reported by the various destinations on the platform, implementing new services to meet the needs of new destinations, and making optimizations, among other issues related to the system code. In addition, they should also accumulate the accesses that the touristic managers have in the management module and the accesses that the tourists have in the portals.

3.2.3 Functional Requirements

After having made a generic collection of the requirements that the Multi-Destination Management System must meet and identified the main stakeholders in this type of information system, the functional requirements will now be defined.

However, it should be taken into account that the system was developed as a proof of concept, and therefore, some functional requirements, such as the reservation of tourism products and the description of tourism products, were kept simple; if the system moves into production in the future a complete specification will be required.

Thus, it will be explained what the system will do in detail and the functionalities that each actor has at his disposal, divided into the two modules that will be available. It will also be used from now on the concept of listings, which will work as a way to agglomerate the advertisements that managers of tourism products can create, namely advertisements of the restaurant and nightlife industry, hotels and accommodation, and Adventure, Nature, Beach, Cultural and Health tourism.

Management Module

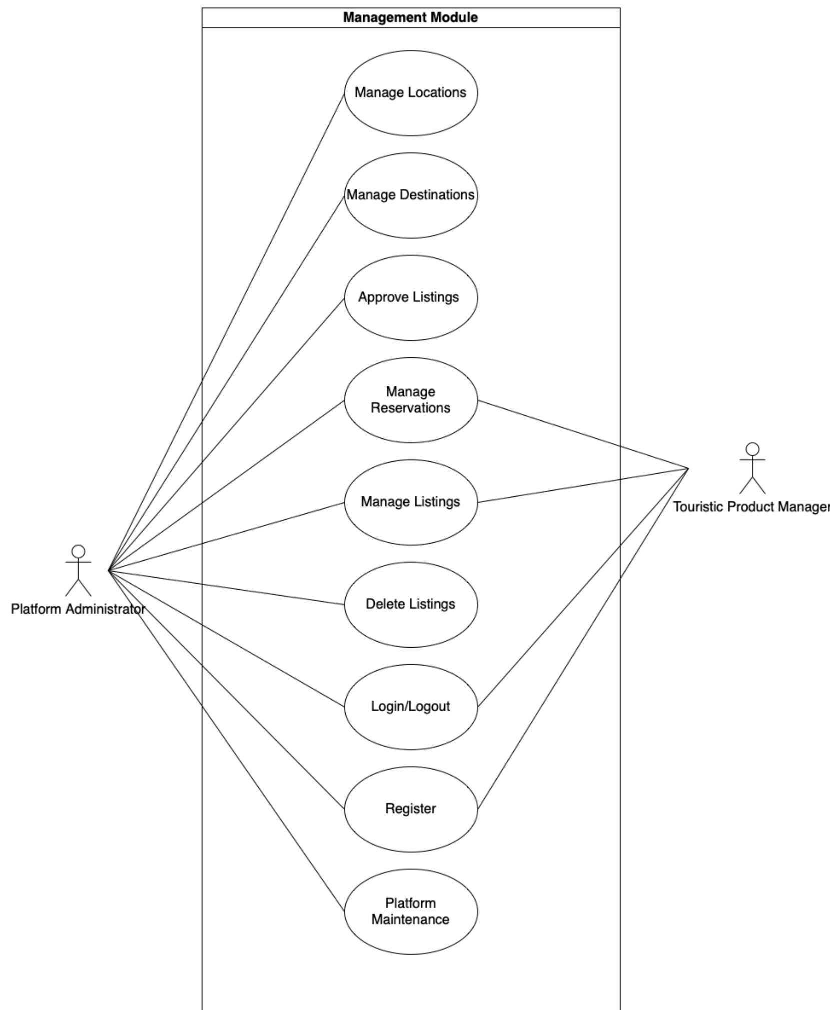


Figure 3.2: Management Module - Use Case Diagram

Figure 3.2 presents the use cases model of the management module, with the interactions that the actors using this system perform on it.

In the following list, the use cases of the presented model will be detailed:

- Manage Locations

  The platform administrator must be able to add new locations to the system and edit or delete existing locations. The use case allows the tour manager actor to perform all CRUD operations on locations.
- Manage Destinations

  The platform administrator must be able to create new destinations, which will act as a grouping of several locations. The destination concept will be a filter to make the DMS for a particular destination. The use case allows the actor to add, edit, and delete destinations, as well as associate locations with destinations.
- Manage Listings

The tourism product manager/owner must be able to create and edit listings in the various tourism products supported by the platform (i.e., hotels and accommodation, restaurants, and bars, useful contacts, activities/events), as well as view their pending and approved listings. The platform administrator, in this use case, can view the listings of all managers.

- Manage reservations

  The manager/owner of a tourism product should be able to view the reservations made by tourists on his tourist products. The platform administrator, in this use case, can view the reservations of all the managers.

- Approve Listings

  The platform administrator can approve/reprove the listings created by the tourism product managers/owners. This use case allows the platform administrator to review and approve new listings made by the tourism product managers/owners before publication in the system.

- Delete listings

  The platform administrator can delete listings created by tourism product managers/owners. This use case allows the platform administrator to delete old or obsolete listings made by the tourism product managers/owners.

- Login/Logout

  All the actors of the management module must be able to login into the platform to access the management module use cases since all of them require authentication and authorization. Likewise, they must be able to logout when they no longer need to carry out this type of tasks.

- Platform Maintenance

  The platform administrator must be able to perform maintenance tasks and improve the platform or even add new services and features. This actor will have access to all the use cases of the system and access to the source code and system databases.

Portal Module

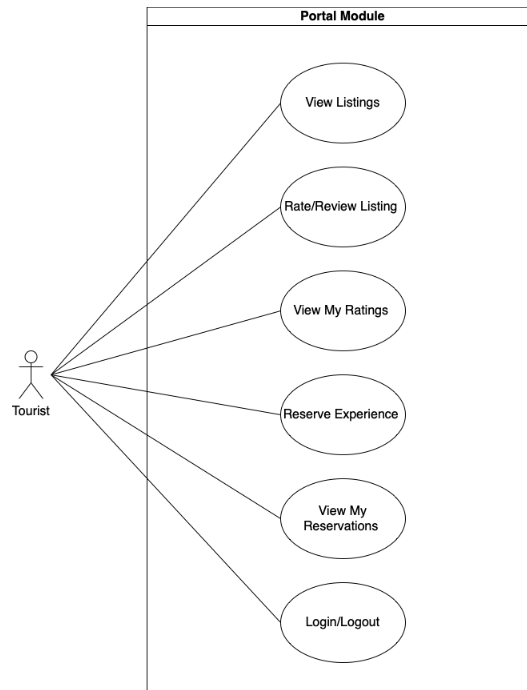


Figure 3.3: Portal Module - Use Case Diagram

Figure 3.3 presents the use cases model of the portal module, with the interactions that the actors using this system perform on it.

The following list details the use cases of the presented model:

- View listings

  Tourists should be able to view listings of all tourism products. It should also be possible to see the description of each tourism product. This use case should be the only one in the system where no authentication is required.
- Reserve Experience

  Tourists should be able to select a listing and book an experience (e.g., reserve a hotel room or reserve a table in a restaurant).
- Rate/Review Experience

  Tourists should be able to evaluate and provide feedback on an experience after enjoying it, which can help other tourists make informed decisions.
- View my ratings

  Tourists should be able to access a page with all the ratings they have given to the experiences they visited.
- View my reservations

  Tourists should be able to access a page with all the reservations they have made on the different tourism products of the platform.
- Login/Logout

  Tourists should be able to log in to the platform as a way of being able to access the portal module use cases, as all of them (except for viewing listings) require authentication

and authorization. Likewise, they must be able to logout when they no longer need to carry out this type of tasks.

3.2.4 Non Functional Requirements

For the Multi-Destination Management System to become a competitive and innovative solution compared to made-from-scratch DMS, the following non-functional requirements have been taken into account:

- Performance and Scalability

  The system should be highly responsive to user interactions with the UI, ensuring acceptable response times when requests are sent to the server. The system database should be horizontally and vertically scalable, if necessary, to handle large amounts of data without compromising performance as business needs increase.
- Reusability

  One of the innovations that this product should bring is the possibility to reuse the vast majority of its components to create new solutions based on it. It should be possible to instantiate new DMS by creating only portal modules for each unique destination interested in using the system.
- Usability

  It is necessary to guarantee that the platform is user-friendly and does not have a high learning curve, leading the actors of both modules to execute the implemented functionalities without many steps and without being subject to errors.
- Extensibility

  The platform should be easily extensible in functionalities to support new needs of new destinations interested in having an implementation based on it. This extension cannot influence the functionalities already implemented, neither at the performance level nor at the stability level of the system in general.
- Maintainability

  After the implementation in the final client, the platform must be easy to maintain by the intervenients and not demand too much effort from the platform administrators.

3.2.5 General Data Protection Regulation

The Multi-Destination Management System designed in this work takes into account compliance with the European Union's General Data Protection Regulation. [24]

In the act of registration, there is a need to collect data from the users of the system for the realization of functionalities that require authorization/authentication. Therefore, all users, before registering, have at their disposal a consent with which they must agree. In this consent, it is explained who has access to the personal data, which is the tourist destination where they are registering, and the entity responsible for the Management of the Multi-Destination Management System, which ultimately has visibility over the data of all destinations.

In addition, the data requested from users are the minimum necessary for the proper functioning of the solution; no data that is not necessary for its use is requested, and the

legitimate holder of the data has the right to request the deletion of their data from the platform at any time.

3.3 System Architecture

In this section, a software architecture that implements the system requirements will be presented and defended, starting with the physical and technological model of the solution and then the domain model of the various microservices of the system.

3.3.1 Physical and Technological model

Figure 3.4 shows the system's physical and technological architecture, describing how the various components will interact. The diagram has, therefore, been grouped into the frontend side, the backend side, and the persistence side.

On the Frontend side, as we saw earlier, the system's users can interact with two different types of web applications: the portal modules, which are used by all the players in the system and each destination has its own, and the management module, which is used by the tourism product managers and platform administrators and is unique and common to all destinations. These applications were developed using the Create React App framework and, as web applications, only require a computer or mobile device with Internet access to access them. These web applications make requests and receive responses from the Backend, presenting them to users.

The Backend is responsible for providing the Frontend with a set of endpoints available on a single IP address and a single port belonging to the Gateway. Communication will thus take place via the HTTP protocol, using the GET, POST, PUT, and DELETE verbs, depending on the request made. Subsequently, the discovery service will indicate to the API Gateway the route of the respective microservice that responds to it, depending on the endpoint called. On the microservice side, the request will go through the respective business logic and, if necessary, can exchange information with the database. In terms of software architecture, the backend is based on microservices. This option is because, compared to other architectures studied, it is the one that best meets the non-functional requirements raised in the previous subsection. It is an architecture widely used today, and the learning curve is more accessible, making development faster and more efficient. From a technological point of view, the programming language chosen was Java, as it is a very well-documented programming language with excellent support for microservice architectures and is the language I feel most comfortable developing software in; the development framework chosen was SpringBoot, as it is a technology that is well adapted and prepared for an architecture based on microservices, as it has well-constructed and accessible documentation and a wide range of support dependencies to aid development.

On the persistence side, a PostgreSQL database management system was chosen because it offers a better quality/cost ratio for a large volume of data. At this early stage, it was decided to use a single database for all the services; since this project is still just a research

project, it didn't make sense to use a database for each service, as this would make the project more demanding to run locally and wouldn't bring significant advantages.

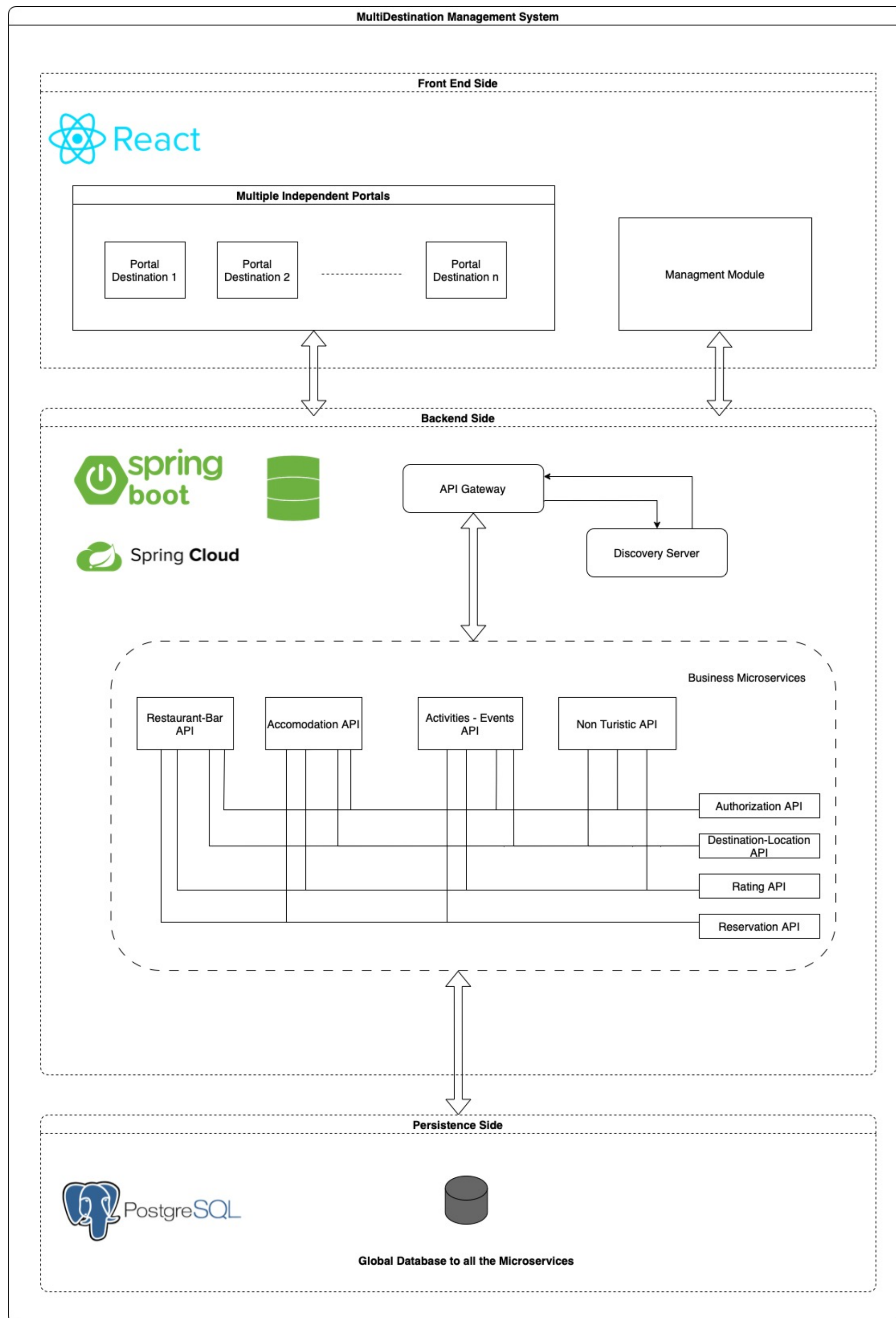


Figure 3.4: Physical and Technological Model

3.3.2 Domain model

The domain model shown in Figure 3.5 describes the various entities, attributes, and relationships that exist in the Backend of the Multi-Destination Management System. Although the architecture of the system backend is based on microservices, there is a need for intra-services communication. Therefore, in this model, many relationships are not between entities

of the same microservice. However, this diagram aims to explain these interactions, abstracting the type of architecture used.

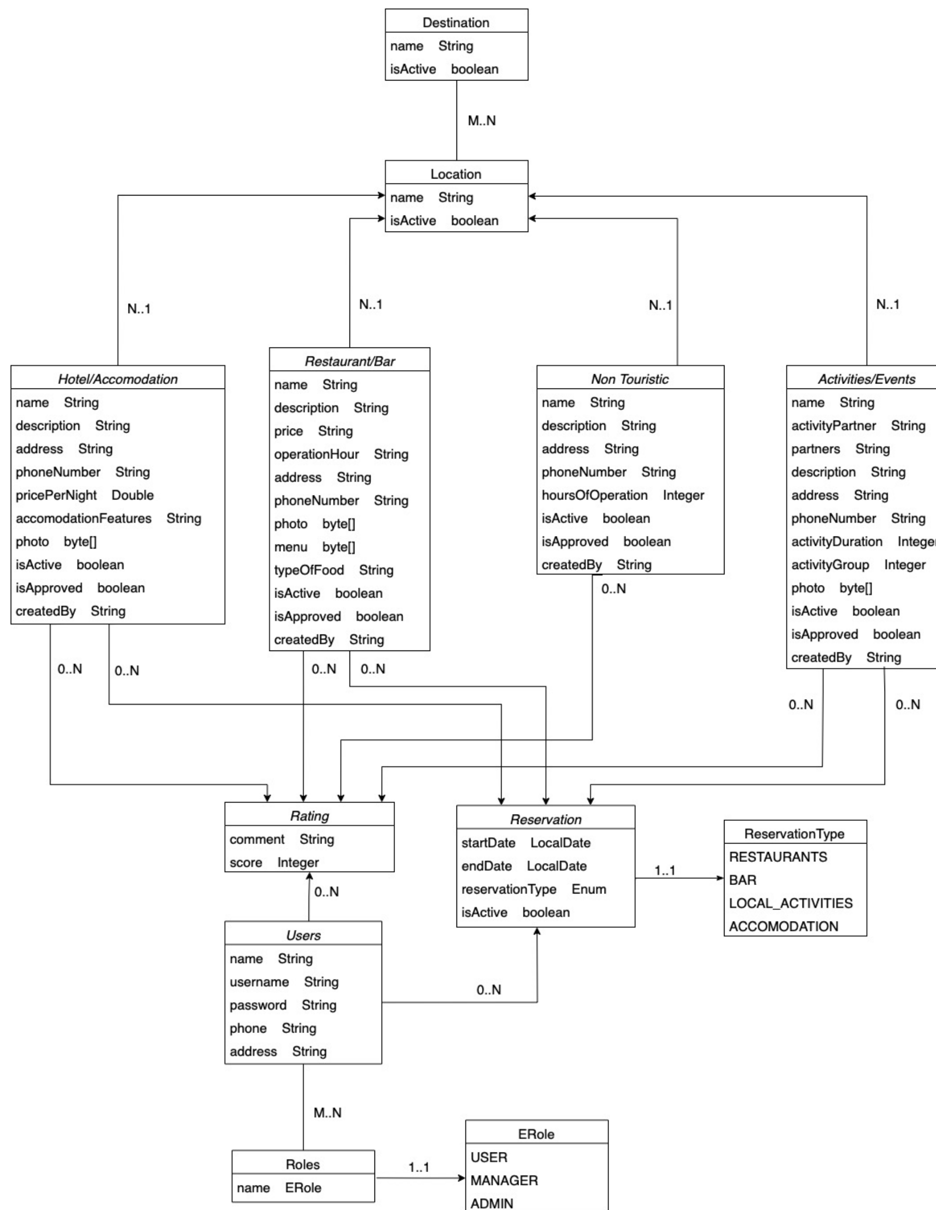


Figure 3.5: Domain Model

Looking at the diagram from top to bottom, we have the destination entity. This entity defines each of the destinations that the system can have and a destination is composed of several locations, and a location can belong to several destinations; that is, we are facing a relationship of many to many. Let us think about the following example: Aveiro can be considered a location and constitute by itself a DMS. However, Aveiro can belong to another destination, for example, the Aveiro region, and form another DMS. Thus, we have two different systems for the same locality.

Next, we have the entities that correspond to the various stakeholders that were considered most relevant during the collection of requirements, the hotels/accommodation, the restaurants

and bars, the tourist activities (here we include the various types of activities that do not fit into the previous ones) and the non-tourist activities (the useful services and contacts of a place). These entities will hereafter be referred to as business entities. The attributes defined in each of the business entities result from the specificities of each of them that were raised during the requirements survey. Thus, there is a 1 to many relationship with the location entity since each business entity has a location, and each location has multiple business entities.

The business entities establish a 0-to-many relationship with the classification entity since each business entity can have none or many classifications of its services, and it also has a 0-to-many relationship with the reservation entity, except for the non-tourist entity that there is no point in allowing reservation since it is a purely informative module for the tourist.

Finally, we have the user entity, which has a 0-to-many relationship with the reservation entity and the rating entity since it may have no rating or reservation or many in the various business entities. Besides, users can have only one role in the system, which can be USER, MANAGER, or ADMIN; however, these roles can belong to several users, thus establishing a 1-to-many relationship.

3.4 Conclusion

The main focus of this chapter was to gather all the requirements needed to develop an innovative system for managing multiple tourist destinations. Thus, the stakeholders in a system of this type, the main actors, and the functional and non-functional requirements the system must meet were analyzed. In addition, an architecture that successfully responds to the proposed problem was presented, along with the technologies used to develop the system. Finally, the system's domain model was introduced, including the entities necessary for proper functioning.

Chapter 4

Implementation

4.1 Introduction

In this chapter, the most critical and essential details for implementing the architecture proposed in section 3.2 of the previous chapter will be explained. The first thing that will be presented is the core of the entire Multi-Destination Management System, which is the system's Backend.

In the Backend, it will be explained in detail the implementation used in the API Gateway, the Discovery service, and how they communicate with each other. It will also explain all the authentication and authorization mechanism that is used by all microservices of the project, along with the essential endpoints that are exposed for this purpose. Finally, implementation details will be presented across the various Business Microservices, along with the endpoints they make available to the outside.

In the Frontend, we won't go into too much technical detail since this academic project is just a proof of concept. However, it is crucial to present the type of endpoints that the various Portal modules and the Management module must consume.

4.2 API Gateway and Discovery Server

This section will present the implementation performed for the API Gateway, the Discovery service, and how both services communicate.

4.2.1 API Gateway

The API Gateway used in this system is the Gateway created under the Spring Cloud project [25]. To configure and use this implementation, it was first necessary to create a new microservice in the project, then add the Maven dependency with the artifact id *spring-cloud-starter-gateway* in the pom.xml of the microservice, as we can see in Figure 4.1 and create some settings in the application.yml file used to start the microservice.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Figure 4.1: Spring Cloud Gateway Dependency

The configurations that were necessary to introduce in the application.yml were essential to indicate to the API Gateway where the routes for each of the microservices of the project, where it was stated a URI, the predicates that must be followed for the construction of the URL and some filter by route, as it is possible to see in the example that the Figure 4.2 demonstrates, and some CORS configurations standard to all microservices of the system were also added, namely the allowed HTTP operations, the allowed headers and the allowed origins by the Gateway as we can see at the Figure 4.3.

```
cloud:
  gateway:
    routes:
      ## Authorization
      - id: authorization
        uri: lb://authorization
        predicates:
          - Path=/authorization/**
        filters:
          - StripPrefix=1
      ## activities-events
      - id: activities-events
        uri: lb://activities-events
        predicates:
          - Path=/activities-events/**
        filters:
          - StripPrefix=1
      ## hotel-accomodation
      - id: hotel-accomodation
        uri: lb://hotel-accomodation
        predicates:
          - Path=/hotel-accomodation/**
        filters:
          - StripPrefix=1
```

Figure 4.2: Api Gateway Routes

```
globalcors:
  cors-configurations:
    '[/**]':
      allowedOrigins: '*'
      allowed-methods:
        - GET
        - POST
        - PUT
        - PATCH
        - DELETE
      allowed-headers: '*'
default-filters:
  - DedupeResponseHeader=Access-Control-Allow-Origin, RETAIN_FIRST
```

Figure 4.3: CORS Configurations

It is also imperative to mention that in each of the routes for the microservices, the URIs are not with a direct IP/port or DNS pair for the service but a load balancing address since this API Gateway will use a Discovery service to point to each of the microservices as will be explained in the next section.

4.2.2 Discovery Server

The discovery service used in this system uses the implementation of the Spring Cloud project, just like the API Gateway, named Netflix Eureka Server. This technology assumes creating a server that will work as Eureka Server, where all microservices must register (Eureka Client) so that the API Gateway knows where they are.

So, looking at the implementation of Eureka Server, it was necessary to create a new microservice in the project. In this microservice, it was required to add the maven dependency with the artifact id *spring-cloud-starter-netflix-eureka-server* in pom.xml as shown in figure 4.4, add the annotation "@EnableEurekaServer" in the main Java file of the microservice and add some settings in the application.properties file. The necessary configurations for the proper functioning of the Discovery Server were to add a port to the service, in this case 8761, and add the properties "register-with-eureka" and "fetch-registry" to false, as shown in figure 4.5, which if they were set to true would cause that when starting the server, the embedded client would try to register itself on the Eureka server and also try to fetch the registration, which is not yet available. [26]

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Figure 4.4: Eureka Server Dependency

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Figure 4.5: Eureka Server Configurations

Looking now at the implementation of the various Eureka Clients, it was necessary to add in all the business microservices the maven dependency with the artifact id *spring-*

cloud-starter-netflix-eureka-client in pom.xml, as shown in figure 4.6, add the annotation "@EnableEurekaClient" in the main Java file of each of the microservices and add some settings in the application.properties file. The necessary settings to add to the clients were the defaultZone, which corresponds to the URL where the discovery service is, and add the property "preferIpAddress" to True, which means that the registration of the instance on the Eureka server will be done through the IP address, as shown in the Figure 4.7.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Figure 4.6: Eureka Client Dependency

```
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
eureka.instance.preferIpAddress=true
```

Figure 4.7: Eureka Client Configurations

After these settings, the Discovery Server already knows where the various microservices are and can provide the API Gateway the addresses of Load Balancing where they are, so it is possible to request the URL of the API Gateway. The whole system takes care of forwarding it to the microservice destination.

4.3 Authentication

Like most applications, the Multi-Destination Management System needs to recognize a user's identity to filter and protect the data they can access, depending on their permissions.

It was, therefore, necessary to create a new microservice called "authorization server" and add a security dependency integrated into the Spring Cloud project, which has the artifact id *spring-cloud-starter-oauth2*, as shown in Figure 4.8. This dependency was chosen because it is highly integrated with the API Gateway, the Discovery service, and the microservices implemented since they all use dependencies from the Spring Cloud project. [27]

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

Figure 4.8: Authentication Dependencies

Then, it was necessary to create the user entity, which stores the personal information required for the proper functioning of the system, such as user name, password, phone number, address, and name, and also create the role entity, which is also associated with the user entity.

In subsection 3.1.2, the system actors were presented, and it was concluded that the system actors would be the tourists, the tourism product managers, the tourism technicians, and the development team. It was also supposed that all these actors have access to the portal module and may or may not have access to the platform management module with varying

permissions. Thus, the solution used a Role Based Access Control (RBAC) where three roles were defined: USER, MANAGER, and ADMIN.

The Tourist, as an actor who only has access to the Portal module, has the role USER; the Tourism Product Managers, as they have access to the Portal module and access with restrictions to the management module, have the role MANAGER, and finally, the tourism technicians and the development team, as they have access to the Portal module and to the management module without restrictions, have the role ADMIN.

To carry out this separation of roles, two registration endpoints were created:/registration, which is used by the portal modules and assigns the role of USER to those who register through it, and /registration-manager, which is used by the management module and gives the role of MANAGER to those who register through it. In the case of the ADMIN role, as it is a role that only a very restricted group of users will have access to, it will be assigned through the database to the users who need it by someone responsible for administering the entire Multi-Destination Management System.

After registration, it is necessary for users to log into the modules, and for this, the native endpoint of the Spring OAuth2 dependency will be used, the /oauth/token, which returns an authorization token that will be used in all authenticated requests from the Multi-Destination Management System. In table 4.1, it can be understood how the authorization service endpoints can be used and their functionality for the system.

In addition, it was also necessary to perform some specific configurations involving the extension of abstract classes and the implementation of some Spring Security and Spring OAuth 2 interfaces. These configurations allowed the following:

- Add support for RBAC
- Perform validation of endpoints that need to pass authentication from those that are free to access
- Add some CORS settings namely the allowed REST methods, allowed sources, and allowed headers
- Settings for generation and validation of generated JSON Web Token (JWT) Tokens

It was also necessary to add in all the other microservices of the system (except the API Gateway and the Discovery Service), the dependencies previously mentioned, and the configurations of the endpoints that need to pass through authorization that is specific for each microservice.

Path (/authorization)	HTTP Method	Authorization	Query Params	Body	Function to the system
/oauth/token	POST	Basic Username: web Password: 123	username:{username} password:{password} grant_type:password manager: y/n	N/A	It logs into the system, obtaining the token that will be used in all authenticated requests. To login in the management module it is necessary to pass manager=y. To login in the portal modules it is necessary to pass manager= n.

/registration	POST	No Auth	N/A	{
				"name": {name},
				"username": {username},
				"password": {password},
				"phone": {phonenumber},
				"address": {address}
				}
					Allows users to register in the portal modules, being assigned the role USER.

/registration-manager	POST	No Auth	N/A	{
				"name": {name},
				"username": {username},
				"password": {password},
				"phone": {phonenumber},
				"address": {address}
				}
					Allows users to register in the management module, being assigned the role MANAGER.

Table 4.1: Authorization REST API endpoints

4.4 Business Microservices

As seen in Figure 3.4 from the previous chapter, the system's business logic has been divided into various autonomous and independent microservices that respond to the specific needs of those considered the system's main stakeholders. These microservices have been given the name business microservices.

Thus, based on the functional collection of requirements presented in Chapter 3, the following business microservices were implemented:

- Restaurant - Bar
- Accommodation
- Non-Touristic
- Activities - Events

In addition to the microservices that directly implement the logic of the stakeholders in the system, we also have three other business microservices that are shared by all the microservices presented above:

- Destination - Location
- Rating
- Reservation

The business microservices follow the MVC software design pattern. The choice of this software design pattern has to do with the fact that it is an architecture very well supported by Spring Boot and is widely used to develop REST API in architectures based on Microservices. [28]

All these microservices have a very similar implementation, with only the entities, business logic, and exposed endpoints varying. Therefore, to make this dissertation less exhaustive, only the process of creating a business microservice will be presented in detail, but all the endpoints of the other microservices will be shown later. So, as an example, for the Accommodation microservice, after its creation, it was necessary to add some maven dependencies to the

microservice's pom.xml. The dependencies used for the microservice to work properly were those shown in Figure 4.9:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-oauth2</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-ui</artifactId>
        <version>1.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Figure 4.9: Accomodation Microservice Dependencies

Starting with the dependency at the top of the figure, we have the one identified with the artefact id *spring-boot-starter-web*, which includes all the libraries and configurations needed to start a web application. We have the dependency with the artefact id *spring-cloud-starter-oauth2*, which, being part of the Spring Cloud project, includes support for OAuth 2.0 authentication, which is used in this project. We also have the *spring-boot-starter-data-jpa* dependency, which consists of a package with various dependencies that simplify the process of integrating relational databases with Spring Boot.

In addition to these, and still, within the dependencies created by the Spring Framework, we have the dependency identified with the artefact id *spring-cloud-starter-config*, which consists of a package of other dependencies, in this case, used to manage configuration files external to the application which are loaded dynamically when it starts up [29], and the dependency identified with the artefact id *spring-cloud-netflix-client* which, as explained above, is required for all services (except eureka server) so that this discovery service can locate all the microservices in the system.

In terms of dependencies outside the Spring Framework, we used the dependency with the artifact id *springdoc-openapi-ui*, which allows us to generate endpoint documentation on a

Swagger page automatically; we also used the dependency with the artifact id *lombok*, which helps a lot to reduce the repetitive processes typical of the Java language by automatically generating getters/setters and constructors and, finally, we used the dependency with the artifact id *postgresql*, which is essential for ensuring connectivity with PostgreSQL databases.

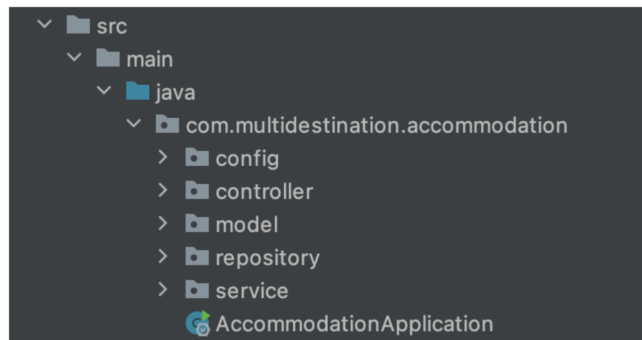


Figure 4.10: Accomodation Microservice Packages

In terms of package structure, all the system's microservices are divided into the layers recommended by the MVC software architecture standard, as can be seen in figure 4.10.

So, starting at the top, we have the "Config" package, which contains two classes responsible for the configurations needed for the microservice to work correctly. In this case, the configurations that had to be added implement security rules such as the endpoints that must undergo authentication and authorization for those exposed for anonymous use and the JWT token's public key configurations. On the side of the "Controller" package, there is a class that manages inputs and outputs with the system's users. For this purpose, the REST protocol was chosen, as it is the most widely developed and used standard in WEB communication today. So, a set of GET, POST, PUT, and DELETE endpoints were implemented according to the system's needs. As far as security is concerned, it was ensured that the endpoints implemented in the controller comply with the principle of minimum permissions, i.e., a user should only have access to the resources strictly necessary for the actions they intend to perform. Therefore, the endpoints are protected by Roles. Next is the "Model" package, which contains the classes that implement the microservice's entities and models. These classes include all the information that the microservice needs, whether it's mapping the database to a Java object (via the entities), a Java object used to transfer data within the microservice Data Transfer Object (DTO) or a Java object used to define the attributes that the request from a REST endpoint should receive. The "Repository" package contains all the classes that query the database. These queries can be made in the form of a native SQL query or generated automatically via Spring Data JPA, and the result of these queries can be mapped directly to one of the entities found in the "Model" package. Finally, the "Service" package contains all the classes responsible for implementing the microservice's business logic, i.e., these service classes bridge the gap between the repositories and the data obtained from them and stored in the models with the necessary business processing so that it is finally possible to deliver this information to the clients via the controllers.

These aspects are common to the system's various business microservices. However, there

Path (/api/restaurants-and-bars)	HTTP Method	Authorization	Query Params	Body	Function to the system

""	POST	Bearer Token	N/A	{
				"name": String
				"description": String
				"price": String
				"operationHour": String
				"address": String
				"phoneNumber": String
				"photo": byte[]
				"menu": byte[]
				"typeOfFood": String
				"location": Location
				}
					Allows users with Administrator or Manager roles to create restaurants/bars adverts for the system.
					Once created, the advert is pending validation by an administrator user.

""	PUT	Bearer Token	N/A	{
				"name": String
				"description": String
				"price": String
				"operationHour": String
				"address": String
				"phoneNumber": String
				"photo": byte[]
				"menu": byte[]
				"typeOfFood": String
				"location": Location
				}
					For users with the Administrator role, you can change the information for all the restaurants/bars in the database, and for users with the Manager role for all the restaurants/bars they manage.
					
					After the update, the announcement is pending validation by an administrator user.

""	GET	Bearer Token	N/A	N/A	For users with the Admin role, you can get all the restaurants/bars in the database, and for users with the Manager role, all the restaurants/bars they manage.
/destination/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain all restaurants/bars in a given destination without any authentication.
/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain specific information about a restaurants/bars without any authentication.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows administrator users to delete any restaurants/bars from the system.
/approve/{id}	POST	Bearer Token	id: Long	N/A	Allows administrator users to approve restaurants/bars adverts, which are automatically visible on the respective portals.

Table 4.2: Restaurant-Bar Service REST API endpoints

are several differences in the business rules implemented between the multiple microservices since the needs that each microservice has to meet are entirely different.

The following tables show the endpoints implemented in all the microservices, with full details of how to use them and the functionality they implement in the system:

Path (/api/hotel-and-accomodations)	HTTP Method	Authorization	Query Params	Body	Function to the system
""	POST	Bearer Token	N/A	{ "name": String "description": String "address": String "phoneNumber": String "pricePerNight": Double "accomodationFeatures": String "photo": byte[] "location": Location }	Allows users with Administrator or Manager roles to create hotel/accommodation adverts for the system. Once created, the advert is pending validation by an administrator user.
""	PUT	Bearer Token	N/A	{ "name": String "description": String "address": String "phoneNumber": String "pricePerNight": Double "accomodationFeatures": String "photo": byte[] "location": Location }	For users with the Administrator role, you can change the information for all the hotels/accommodations in the database, and for users with the Manager role for all the hotels/accommodations they manage. After the update, the announcement is pending validation by an administrator user.
""	GET	Bearer Token	N/A	N/A	For users with the Admin role, you can get all the hotels/accommodations in the database, and for users with the Manager role, all the hotels/accommodations they manage.
/destination/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain all hotels/accommodation in a given destination without any authentication.
/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain specific information about a hotel/accommodation without any authentication.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows administrator users to delete any hotel/accommodation from the system.
/approve/{id}	POST	Bearer Token	id: Long	N/A	Allows administrator users to approve hotel/accommodation adverts, which are automatically visible on the respective portals.

Table 4.3: Accomodation Service REST API endpoints

Path (/api/non-tourists)	HTTP Method	Authorization	Query Params	Body	Function to the system
""	POST	Bearer Token	N/A	{ "name": String "description": String "address": String "phoneNumber": String "hoursOfOperation": String "location": Location }	Allows users with Administrator role to create non touristic adverts for the system. Once created, the advert is pending validation by an administrator user.
""	PUT	Bearer Token	N/A	{ "name": String "description": String "address": String "phoneNumber": String "hoursOfOperation": String "location": Location }	Allows users with the Administrator role to update information for all non touristic adverts of the system After the update, the announcement is pending validation by an administrator user.
""	GET	Bearer Token	N/A	N/A	Allows users with the Admin/Manager role, to get all the non-touristic adverts in the system.
/destination/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain all non-touristic adverts in a given destination without any authentication.
/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain specific information about a non-touristic advert without any authentication.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows users with Admin role to delete any non-touristic adverts from the system.
/approve/{id}	POST	Bearer Token	id: Long	N/A	Allows users with Admin role to approve non-touristic adverts, which are automatically visible on the respective portals.

Table 4.4: Non Touristic Service REST API endpoints

Path (/api/activities-events)	HTTP Method	Authorization	Query Params	Body	Function to the system

""	POST	Bearer Token	N/A	{
				"name": String
				"activityPartner": String
				"partners":String
				"description": String
				"address":String
				"phoneNumber":String
				"activityDuration": Integer
				"activityGroup": Integer
				"photo": byte[]
				"location": Location
				}
					Allows users with Administrator or Manager roles to create activities/events adverts for the system.
					Once created, the advert is pending validation by an administrator user.

""	PUT	Bearer Token	N/A	{
				"name": String
				"activityPartner": String
				"partners":String
				"description": String
				"address":String
				"phoneNumber":String
				"activityDuration": Integer
				"activityGroup": Integer
				"photo": byte[]
				"location": Location
				}
					For users with the Administrator role, you can change the information for all the activities/events in the database, and for users with the Manager role for all the activities/events they manage.
					
					After the update, the announcement is pending validation by an administrator user.

""	GET	Bearer Token	N/A	N/A	For users with the Admin role, you can get all the activities/events in the database, and for users with the Manager role, all the activities/events they manage.
/destination/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain all activities/events in a given destination without any authentication.
/{id}	GET	No Auth	id: Long	N/A	Allows users to obtain specific information about a activities/events without any authentication.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows administrator users to delete any activities/events from the system.
/approve/{id}	POST	Bearer Token	id: Long	N/A	Allows administrator users to approve activities/events adverts, which are automatically visible on the respective portals.

Table 4.5: Activities-Events Service REST API endpoints

Path (/api/ratings)	HTTP Method	Authorization	Query Params	Body	Function to the system
""	POST	Bearer Token	N/A	{ "comment": String "score": Integer "touristicProductId": Long "ratingType": String }	Allows all authenticated users to create ratings/criticisms for the various types of tourist product adverts in the system. They can then assign a rating from 1 to 5 and write a comment about the advert.
""	PUT	Bearer Token	N/A	{ "comment": String "score": Integer "touristicProductId": Long "ratingType": String }	Allows all authenticated users to update the ratings/criticisms that they have made on the different tourist product adverts in the system.
/ratings-by-product-id /{touristicProductId}	GET	No Auth	touristicProductId: Long	N/A	Allows all users to obtain the ratings/criticisms of a tourist product they are viewing without any authentication.
/myratings	GET	Bearer Token	N/A	N/A	Allows users to obtain all the ratings/criticisms they have made of the tourist products they have visited.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows users to delete their ratings/reviews on the various tourist products in the system.

Table 4.6: Rating Service REST API endpoints

Path (/api/reservations)	HTTP Method	Authorization	Query Params	Body	Function to the system
""	POST	Bearer Token	N/A	{ "touristicProductId": Long "startDate": LocalDate "endDate": LocalDate "reservationType": String }	Allows all authenticated users to create reservations for the various types of tourist product adverts in the system.
""	PUT	Bearer Token	N/A	{ "touristicProductId": Long "startDate": LocalDate "endDate": LocalDate "reservationType": String }	Allows all authenticated users to update the reservations that they have made on the different tourist product adverts in the system.
/myreservations	GET	Bearer Token	N/A	N/A	Allows users to obtain all the reservations they have made in the tourist products they want to visit.
/reservations-by-username /{username}	GET	Bearer Token	username: String	N/A	Allows users with admin permissions to obtain the reservations that any user of the system has made on all tourist products .
/reservations-of-manager	GET	Bearer Token	N/A	N/A	Allows users with admin or manager permissions to obtain the reservations made by users in the products they have advertised.
/{id}	GET	Bearer Token	id: Long	N/A	Allows users to obtain specific information about a reservation that they made.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows users to delete their reservations on the various tourist products in the system.

Table 4.7: Reservation Service REST API endpoints

Path (/api/destination)	HTTP Method	Authorization	Query Params	Body	Function to the system

""	POST	Bearer Token	N/A	{
				"id": Long
				"name": String
				"isActive": boolean
				"locations": Set<Location>
				}
					Allows users with admin permissions to create new destinations to the system

""	PUT	Bearer Token	N/A	{
				"id": Long
				"name": String
				"isActive": boolean
				"locations": Set<Location>
				}
					Allows users with admin permissions to update information of the destinations of the system

""	GET	Bearer Token	N/A	N/A	Allows users with admin permissions to obtain all the destinations of the system
/{id}	GET	Bearer Token	id: Long	N/A	Allows users with admin permissions to obtain specific information about the destinations.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows users with admin permissions to delete the destinations of the system.

Table 4.8: Destination/Location Service - Destination REST API endpoints

Path (/api/location)	HTTP Method	Authorization	Query Params	Body	Function to the system

""	POST	Bearer Token	N/A	{
				"id": Long
				"name": String
				}
					Allows users with admin permissions to create new locations to the system

""	PUT	Bearer Token	N/A	{
				"id": Long
				"name": String
				}
					Allows users with admin permissions to update information of the locations of the system

""	GET	Bearer Token	N/A	N/A	Allows users with admin permissions to obtain all the locations of the system
/{id}	GET	Bearer Token	id: Long	N/A	Allows users with admin permissions to obtain specific information about the locations.
/{id}	DELETE	Bearer Token	id: Long	N/A	Allows users with admin permissions to delete the locations of the system.

Table 4.9: Destination/Location Service - Location REST API endpoints

4.5 Frontend/Backend connection

As seen above, the Multi-Destination Management System uses a client-server architecture in which a single server serves multiple clients (portal modules and the shared management module). In this chapter, the server component of the system has been presented so far, and this subsection will show how the clients communicate with the server.

The Multi-Destination Management System server exposes the various endpoints implemented in the business microservices via the gateway (port 8090). Thus, the multiple clients make HTTP requests to the gateway, sending information to the microservices or, conversely, collecting data from them.

In the case of the portal modules, the endpoints used are mainly of the GET type. Their request must provide a unique ID as a query parameter, which will be used to identify and differentiate the destination by the server and return the information advertised about the tourist destination. In addition to GET endpoints, portals also use POST endpoints to allow the user to categorize or book tourism products. Table 4.10 shows the endpoints consumed by the portals per microservice.

Business Microservice	Endpoint	HTTP Method
Restaurant-Bar	/restaurant-bar/api/restaurants-and-bars/destination/{destinationId}	GET
Accommodation	/hotel-accomodation/api/hotel-and-accomodations/destination/{destinationId}	GET
Activities-Events	/activities-events/api/activities-events/destination/{destinationId}	GET
NonTourist	/non-tourist/api/non-tourists/destination/{destinationId}	GET
Rating	/rating/api/ratings	POST
	/rating/api/ratings/ratings-by-product-id/{product_id}	GET
	/rating/api/ratings/myratings	GET
	/reservation/api/reservations	POST
2-3 multiplecolor[HTML]EFEFEFReservation	/reservation/api/reservations/myreservations	GET

Table 4.10: Portal - Needed Endpoints By Microservice

The management module functions as a CRUD, i.e., it implements the creation, reading, updating, and deletion functionalities for the system's various microservices. However, depending on each user's permissions, some endpoints may not be authorized for all users with access to this platform. Table 4.11 shows the endpoints the management module uses to implement its functionality and the permissions required to be authorized to use them.

Business Microservice	Endpoint	HTTP Method
Restaurant-Bar	/restaurant-bar/api/restaurants-and-bars	GET
	/restaurant-bar/api/restaurants-and-bars	POST
	/restaurant-bar/api/restaurants-and-bars	PUT
	/restaurant-bar/api/restaurants-and-bars/{id}	GET
	/restaurant-bar/api/restaurants-and-bars/{id}	DELETE
	/restaurant-bar/api/restaurants-and-bars/approve/{id}	POST
Accommodation	/hotel-accomodation/api/hotel-and-accomodations	GET
	/hotel-accomodation/api/hotel-and-accomodations	POST
	/hotel-accomodation/api/hotel-and-accomodations	PUT
	/hotel-accomodation/api/hotel-and-accomodations/{id}	GET
	/hotel-accomodation/api/hotel-and-accomodations/{id}	DELETE
	/hotel-accomodation/api/hotel-and-accomodations/approve/{id}	POST
Activities-Events	/activities-events/api/activities-events	GET
	/activities-events/api/activities-events	POST
	/activities-events/api/activities-events	PUT
	/activities-events/api/activities-events/{id}	GET
	/activities-events/api/activities-events/{id}	DELETE
	/activities-events/api/activities-events/approve/{id}	POST
NonTourist	/non-tourist/api/non-tourists	GET
	/non-tourist/api/non-tourists	POST
	/non-tourist/api/non-tourists	PUT
	/non-tourist/api/non-tourists/{id}	GET
	/non-tourist/api/non-tourists/{id}	DELETE
	/non-tourist/api/non-tourists/approve/{id}	POST
Destination-Location	/location/api/destination	GET
	/location/api/destination	POST
	/location/api/destination	PUT
	/location/api/destination/{id}	GET
	/location/api/location	GET
	/location/api/location	POST
	/location/api/location	PUT
	/location/api/location/{id}	GET
Reservation	/reservation/api/reservations/reservations-of-manager	GET
	/reservation/api/reservations/reservations-by-username	GET

Table 4.11: Management - Needed Endpoints By Microservice

4.6 Conclusion

This chapter presents the most critical aspects of implementing the system. The focus was on some essential microservices for implementing an architecture based on microservices, namely the API Gateway and the Discovery Server. Afterward, we explained how we implemented a user authentication and authorization mechanism for the system. In addition, the essential details in implementing the microservices that contain the business logic of the system's stakeholders and the endpoints that each of these services implements were also presented. As a way of making it possible to create one or more frontends for the portal modules or the management module, it was also explained by microservice which endpoints would need to be consumed to guarantee the correct functioning of the solution.

Chapter 5

Results and Discussion

5.1 Introduction

This chapter will demonstrate and analyze the results obtained from this work. Since a functional prototype was developed with two portals simulating two tourist destinations and a management platform, a brief presentation of the prototype will be made. However, specific details of the front-end created will not be analyzed since the aesthetic and usability component of the solution developed is outside the scope of this dissertation. The results of the system's performance tests will also be presented and discussed, in particular, the load tests carried out using the JMeter tool.

5.2 Portal and Management System: Prototypes

To better present and validate the implemented solution, two prototypes were created: one of a portal for a destination, in this case, the Aveiro region, and a management platform that can be used by various managers of tourism products in various destinations. All the functionalities that were implemented in the Backend of the solution are functional in the two front-end prototypes.

5.2.1 Portal

To demonstrate how the implemented system could be used, two prototype portals were created for two tourist destinations, the Terras de Trás os Montes and the Aveiro region. However, since the functionalities of the portals are similar, only the Aveiro region portal will be extensively presented. In Figure 5.1, we can see the home page of the destination Portal. Here, various Hotels/Accommodations, Restaurants/Bars, Activities/Events, and Useful Contacts for the destination are advertised. It's also worth noting that in all the adverts you can find more information about the tourism product and make a reservation or rating (except for useful contacts).

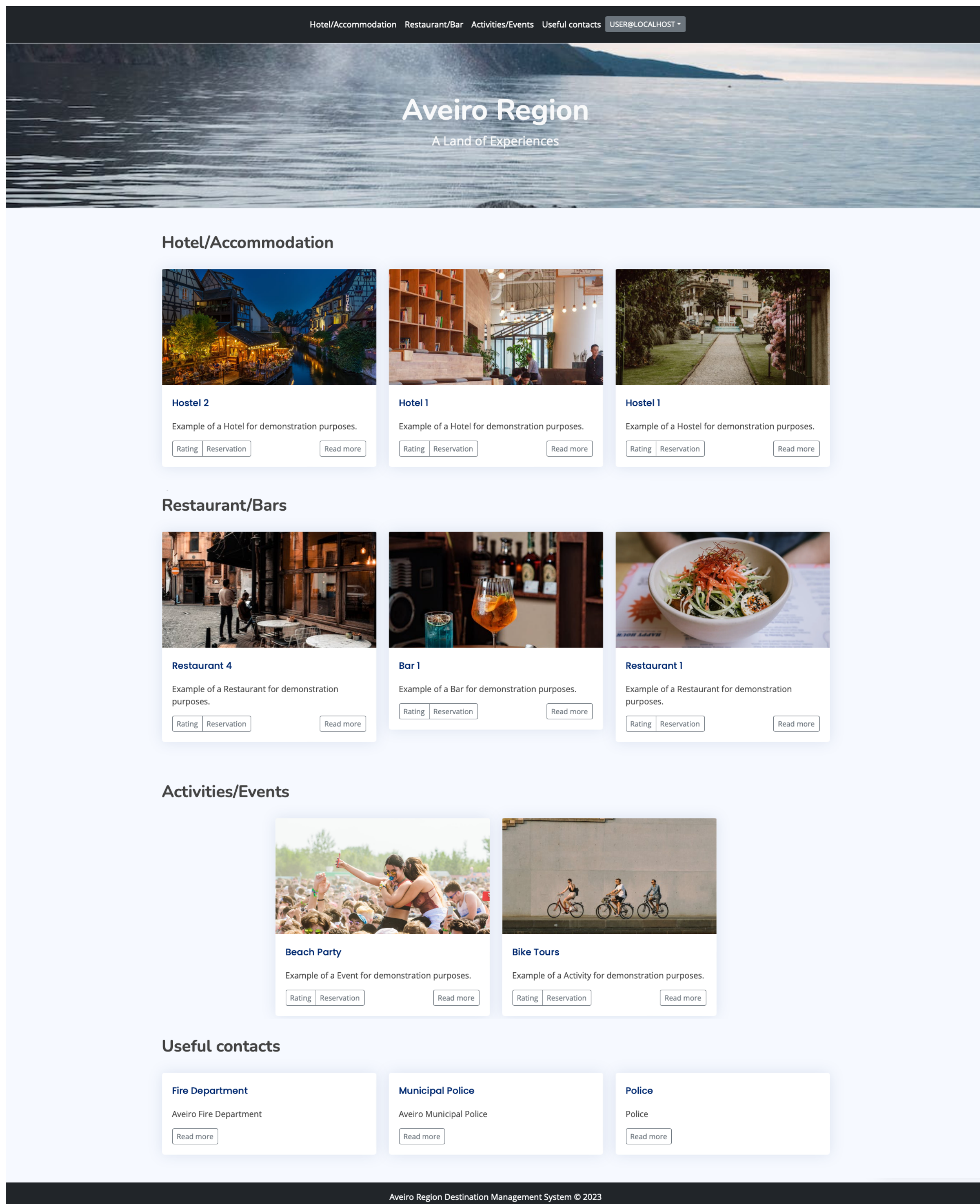


Figure 5.1: Aveiro Region Portal - Main Page

As soon as the user selects the more information option available on all DMS adverts, they have access to information about the tourism product, which, as explained above, varies depending on the type of advert they are viewing. They also have access to the comments and ratings that other users have made, as shown in Figure 5.2.

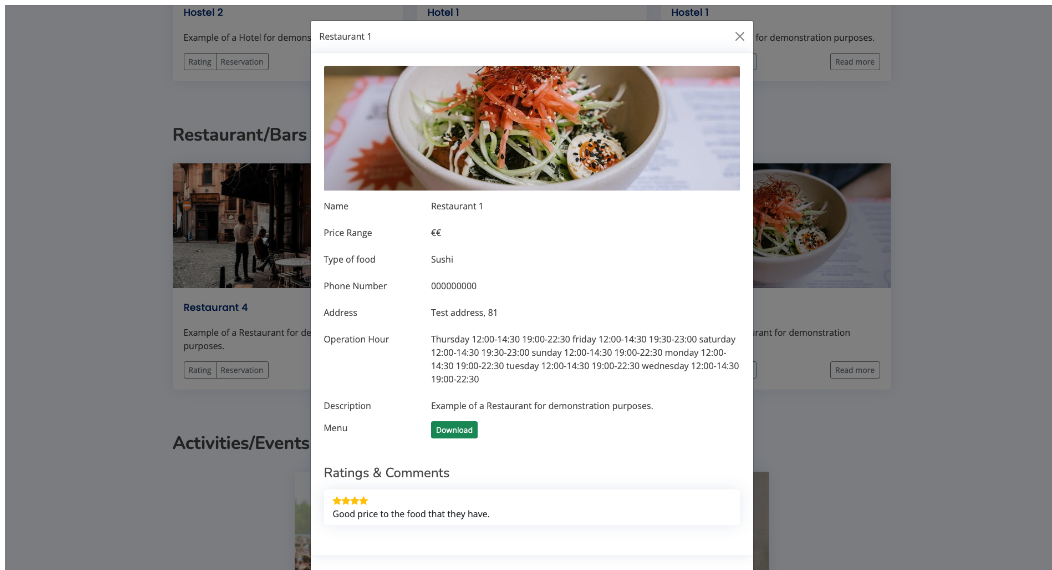


Figure 5.2: Aveiro Region Portal - More Detail about a Touristic Product

Suppose the user is interested in a particular tourism product. In that case, they can make a reservation by selecting the "Reservation" button in the respective advert and being directed to a pop-up where they can choose the dates they want to book, as seen in Figure 5.3.

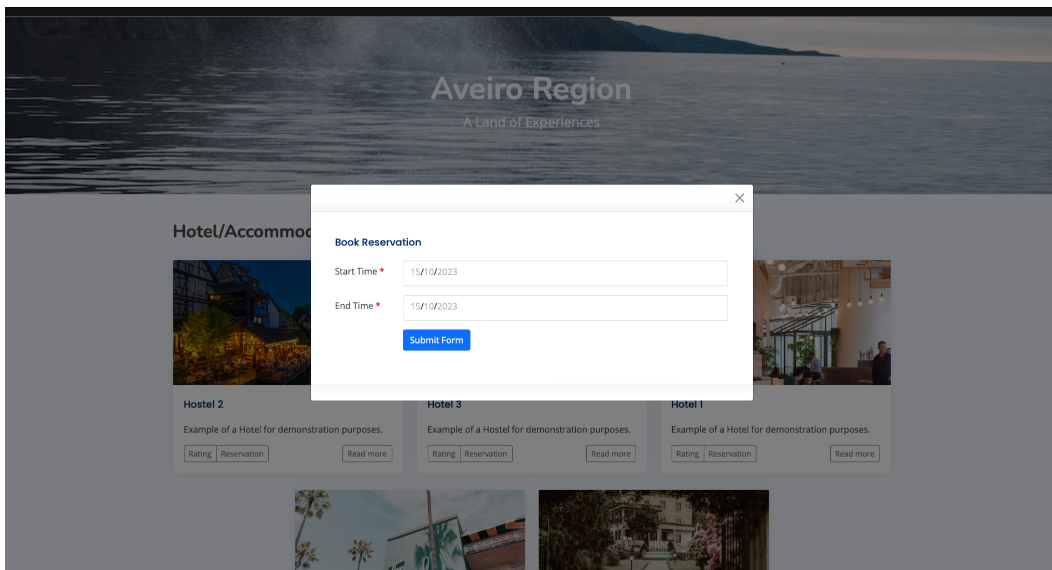


Figure 5.3: Aveiro Region Portal - Do a Reservation of a Touristic Product

After enjoying the tourism product, the user can rate the experience by clicking on the rating button in the respective advert, being directed to a pop-up where they can assign a rating from 1 to 5 in the form of stars and write a short comment, as seen in Figure 5.4.

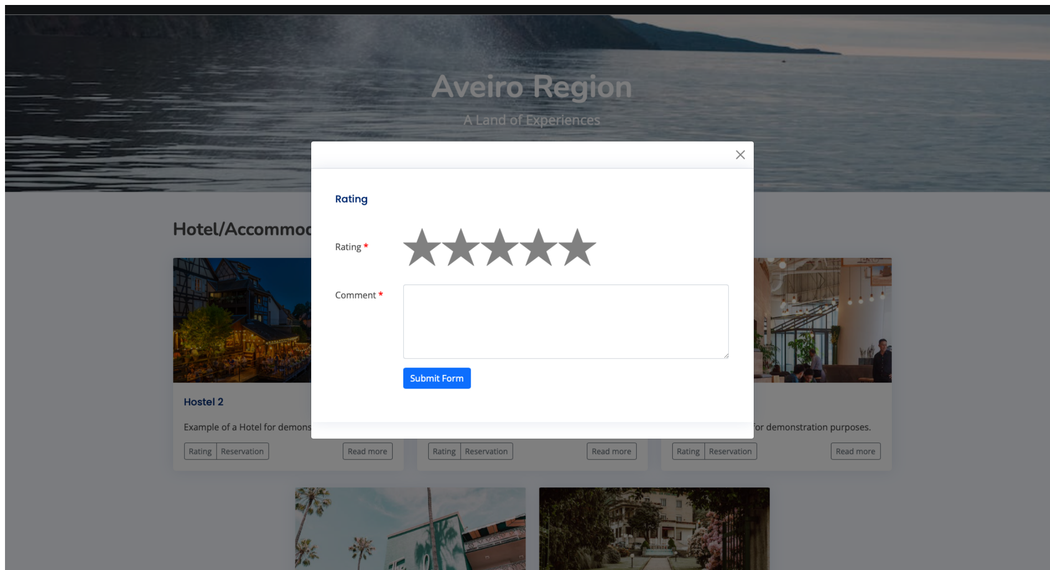


Figure 5.4: Aveiro Region Portal - Do a Rating of a Touristic Product

If they wish, users can also consult the ratings given to the tourist products they have visited, as shown in Figure 5.5.

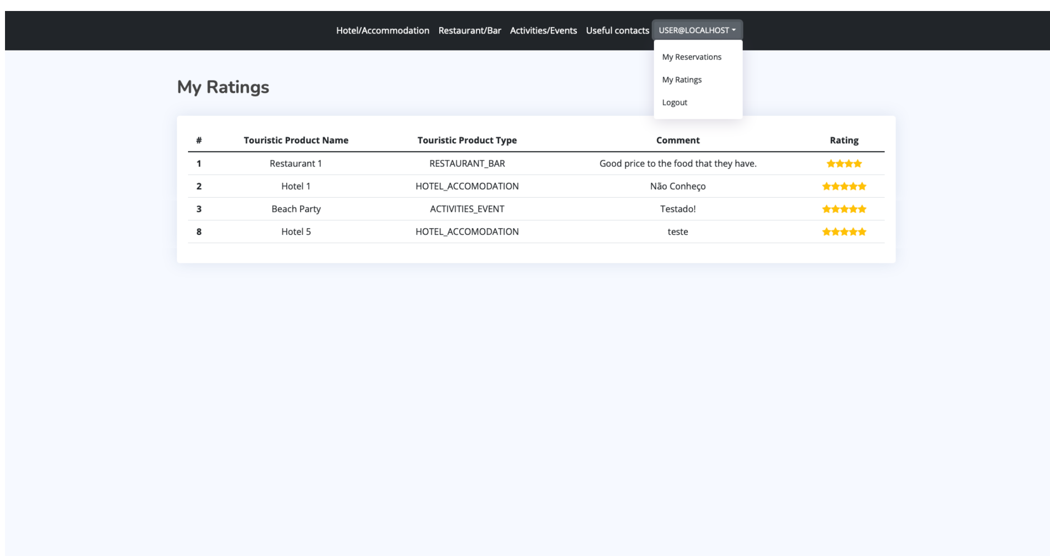


Figure 5.5: Aveiro Region Portal - My Ratings Page

In addition, they can also check the reservations they have pending and have made in the past, as shown in Figure 5.6.

Figure 5.6: Aveiro Region Portal - My Reservations Page

However, it's important to remember that the tasks "do a reservation", "do a rating", "view my reservations" or "view my ratings" require the user to be registered and logged in to the destination's Portal.

5.2.2 Management

As explained above, one of the aims of the system is to create a management module that can be made available to tourism product managers to advertise their businesses, which, after approval by a system administrator, are automatically displayed on one or more tourist destination portals. A prototype of a management module was therefore created that demonstrates how it would be possible to support the flows that tourism product managers can make in the system and also how administrators control the entire platform.

Figure 5.7 shows the main page of the management platform, which has a sidebar on the left-hand side with the various functionalities of a tourism product manager, which will be presented in more detail in the following figures.

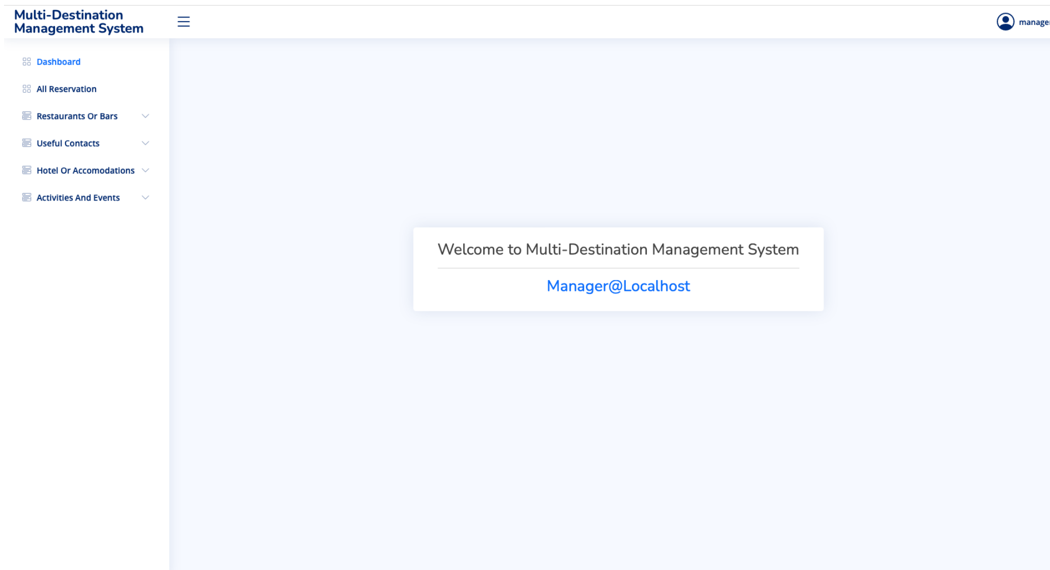


Figure 5.7: Multi-Destination Management System - Main Page

In the "All Reservation" option, you can find all the reservations made for the manager's tourism products, with information on the advert, the username of the person who made the reservation, and the start and end dates of the reservation, as can be seen in Figure 5.8.

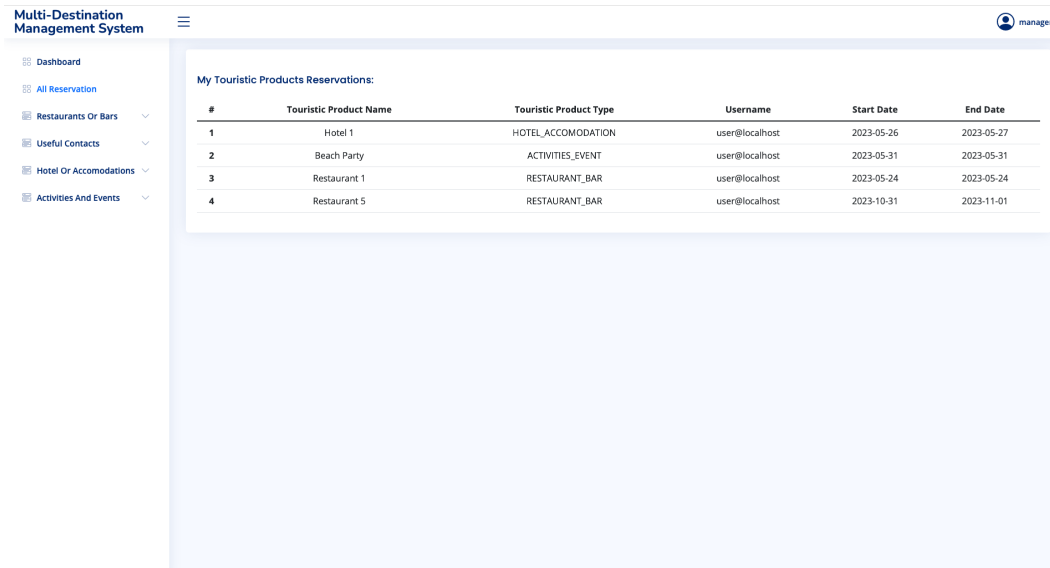


Figure 5.8: Multi-Destination Management System - My Touristic Product Reservations

In each tourist product (Restaurants/Bars, Hotels/Accommodations or Activities/Events), you can find a list of the tourist products that the manager has advertised or, if you are an admin user, with all the advertisements that all the managers have advertised.

In this list, managers can find details of their ads, their approval status, and an option to edit them. If they are admins, they can also approve, reject ads, and delete ads from the system. Figure 5.9 shows the list of system activities/events from the point of view of a system administrator.

Figure 5.9: Multi-Destination Management System - Activities/Events List

In addition, both admin and manager users can add new ads to the system, but only admin users can add useful contacts to the system. Figure 5.10 shows an example of a page for creating an advertisement, in this case, for a restaurant/bar in the system.

Figure 5.10: Multi-Destination Management System - Create Restaurant/Bar Advert

The administrators, as explained in Chapter 3, have the possibility and responsibility of adding locations to the system so that the tourism product managers of those locations can publish their businesses. Thus, they have access to a list of all the locations that are already available, and they have access to a page for adding new locations, as shown in Figure 5.11 and Figure 5.12, respectively.

Figure 5.11: Multi-Destination Management System - Locations List

Figure 5.12: Multi-Destination Management System - Create Location

Later, after creating locations, administrators can create tourist destinations on the platform, associating one or more locations with each Destination. This can be seen in the example in Figure 5.13, where there is a list of all the tourist destinations the Multi-Destination Management System supports, along with the locations each Destination has. It is also possible to see how creating a tourist destination works in Figure 5.14. Once the Destination has been created, the system administrator will have access to a destination identifier, which he will use as a filter in the endpoints presented in Chapter 4 as necessary for creating a new portal.

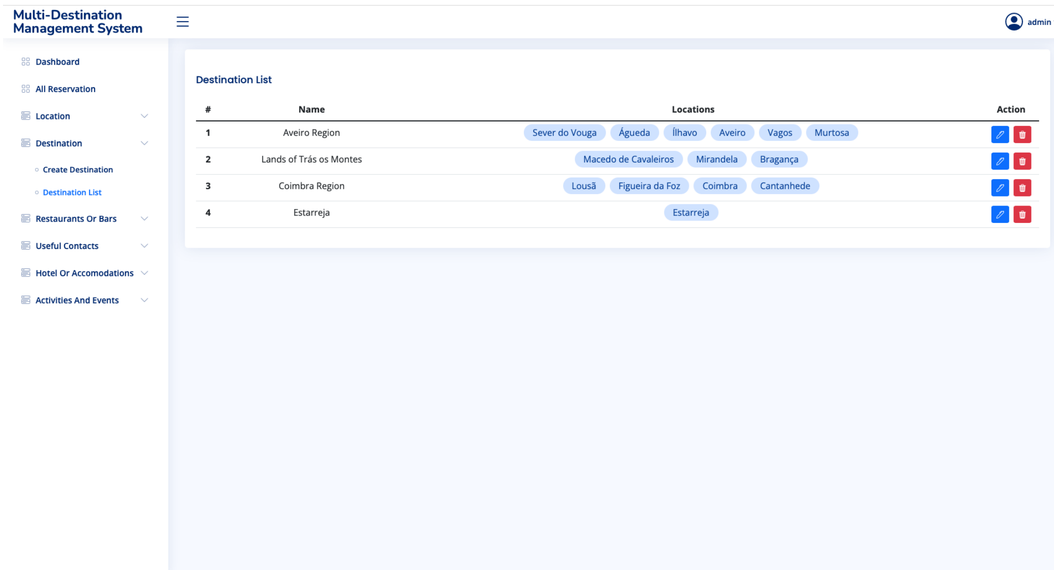


Figure 5.13: Multi-Destination Management System - Destinations List

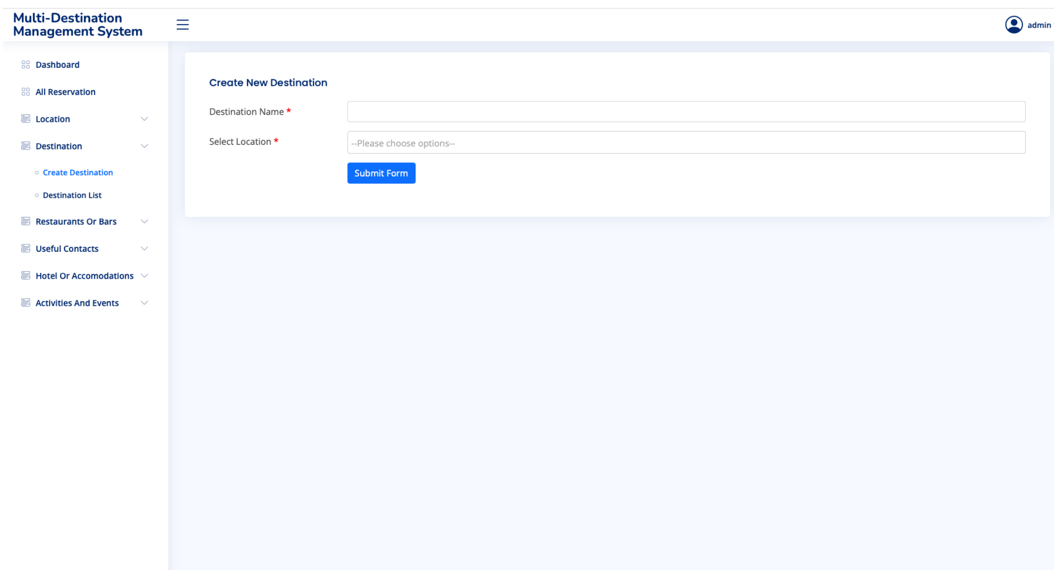


Figure 5.14: Multi-Destination Management System - Create Destination

5.3 System Performance

To ensure that the performance of a Multi-Destination Management System can be the same or similar to that of a single DMS, it was necessary to carry out some performance tests on the system. Various types of performance tests can be carried out on software, but the most common are:

- General performance test: the system under test is in a normal hardware and software environment without any pressure.
- Stability test: the system is put into continuous operation to see if it remains stable.
- Load test: As with the stability test, the system is put into continuous operation, but it will be close to the pressure limit it can withstand. This type of test thus provides a

method for testing whether the system operates stably in a critical state. It is often used to evaluate the system's capacity or to assess where its performance can be improved.

- Stress test: In this test, the system is subjected to increasing pressure until it collapses to test the maximum pressure it can withstand. In this way, it is possible to determine at what load condition the system's performance is in a state of failure to obtain its performance limit [30].

So, to understand the limits that a Multi-Destination Management System can have, it was decided to evaluate the performance through load tests, which allowed us to conclude possible improvements in each of the business microservices and also to understand the performance that the system has with multiple destinations sharing the same infrastructure.

5.3.1 Load Test

Conditions

To analyze the results obtained in the system load tests credibly, it is important to know under what conditions the test was carried out, what tool was used, and what parameters were used in the trial.

About the conditions under which the test was carried out, it was carried out locally on the machine where the system was developed since it is a prototype and is, therefore, not deployed on any server. These conditions mean that the performance of the machine itself can have an impact on the values obtained in the tests, in particular, the minimum, average, and maximum response times for requests. Still, it also allows conclusions to be drawn about the system's performance.

The software used for the load tests is Apache JMeter. This tool is an open-source application based on Java that allows you to carry out load tests, functional tests, or unit tests on an application. Using JMeter, it is possible to simulate multiple simultaneous users making requests to a web application and evaluate the performance of the software using metrics such as response time, transfer rate, and number of failures. This solution also has a GUI to create test plans and export the results as web dashboards [31].

Parameters

In terms of test parameters, it was necessary to choose the endpoints most representative of the typical interaction flow that users will carry out and also to determine appropriate load values for a performance test of this type, bearing in mind that the system is running on a personal computer with all the limitations that this brings.

Figure 5.15 shows how the test plan was structured. Two thread groups were created, one for the portal modules and the other for the management module.

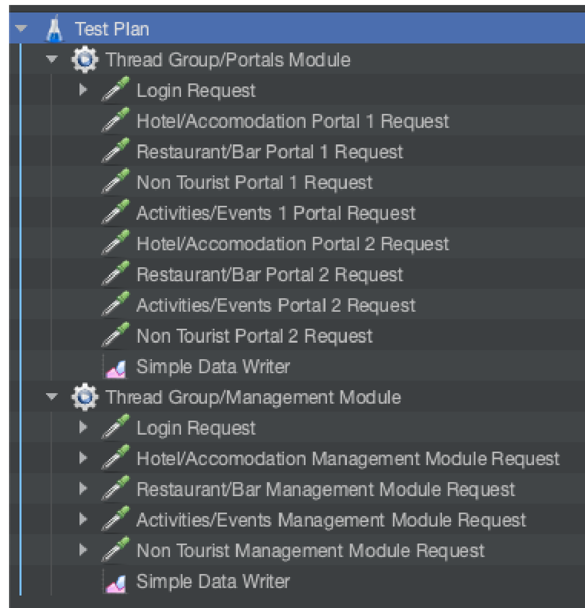


Figure 5.15: JMeter - Test Plan

In the portal modules thread group, the login endpoint used in these modules and the GET-type REST requests that are initially called to populate a portal were added, and these requests were all duplicated to simulate two portals from two different destinations. In the management module's thread group, the login endpoint used was also added, and the typical GET-type REST requests that a tourism product manager calls in their normal flow of interaction with the system were added.

In terms of relevant parameters for each thread group, we have the number of threads, which consists of the number of concurrent users that will be simulated in the test, the ramp-up period, and the loop count, which JMeter uses to determine how long it will take to operate all the selected threads, and the duration of the test. [32] In the two thread groups created, the parameters used are visible in Figure 5.16.

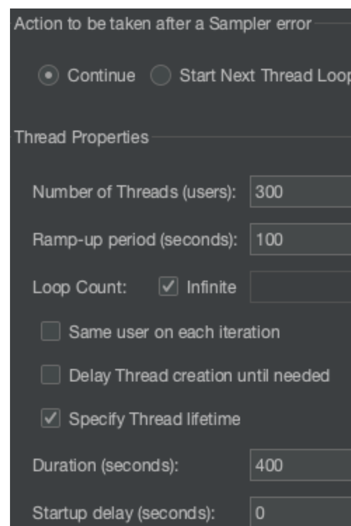


Figure 5.16: JMeter - Thread Properties

Results

After carrying out the tests, various relevant data on the performance of the Multi-Destination Management System were returned. Still, the number of executions, the percentage of requests that returned an error, the average, minimum, and maximum response times, and the size of the packets received by JMeter already allow us to draw reasonable conclusions.

Portals Load Test Statistics							
Requests	Executions			Response Times(ms)			Network (KB/sec)
Label	Samples	Fail	Error %	Average	Min	Max	Received
Activities/Events Portal 1 Request	694	0	0.00%	147.23	63	1168	160.00
Activities/Events Portal 2 Request	506	0	0.00%	297.53	140	1891	263.60
Hotel/Accomodation Portal 1 Request	804	0	0.00%	556.88	118	3612	434.66
Hotel/Accomodation Portal 2 Request	694	0	0.00%	567.82	228	3599	621.06
Login Request	805	0	0.00%	328.42	234	1376	2.55
Non Tourist Portal 1 Request	694	0	0.00%	33.91	12	895	1.74
Non Tourist Portal 2 Request	505	0	0.00%	27.45	13	110	1.64
Restaurant/Bar Portal 1 Request	803	429	53.39%	74185.20	1732	98454	1749.53
Restaurant/Bar Portal 2 Request	694	388	55.91%	83612.61	7638	100025	1808.48

Table 5.1: JMeter - Portal Modules Test Results

Management Load Test Statistics							
Requests	Executions			Response Times(ms)			Network (KB/sec)
Label	Samples	Fail	Error %	Average	Min	Max	Received
Activities/Events Management Request	1161	0	0.00%	168.76	79	1460	354.02
Hotel/Accomodation Management Request	1460	0	0.00%	670.39	187	3746	1351.09
Login Request	1461	0	0.00%	325.53	241	1361	4.68
Non Tourist Management Request	1161	0	0.00%	26.35	11	155	5.27
Restaurant/Bar Management Request	1460	829	56.81%	80576.60	7895	104600	8999.72

Table 5.2: JMeter - Management Module Test Results

Analysing Table 5.1 and Table 5.2, it is possible to conclude that the response times for requests made on the portals are shorter than those made in the management module.

In addition, it can be seen that the restaurant/bar service has excessively high response times for requests and packet sizes, which causes a high percentage of requests to be returned in error, as the server is unable to cope with a large load of requests with a large packet size.

On the other hand, it can be concluded that the other microservices in the system have very reasonable response times, given the load applied to the system and the conditions under which the test was carried out.

Discussion

Taking into account the results of the system tests, it is possible to conclude that the size of the restaurant and bar microservice responses needs to be optimized. The reason identified for the size of the responses in this microservice being so large is that in each advert, the menu and photo are stored in the database in the form of an array of bytes and returned directly in response to the request made to the microservice. One way around this problem could be to host the images/documents in a service external to the system, store the URL in the database, and return it in the request responses. It is also possible to see that with a considerable load applied to the system, the remaining services are pretty stable and resilient, with good response times to requests. However, to draw better conclusions about the system load, the test must be run after the project has been deployed in a real environment.

5.4 Conclusion

This chapter presented the results obtained in this dissertation, namely through the creation of a prototype portal for a tourist destination and the creation of a management module that supports multiple tourist destinations, where some of the main functionalities of the implemented prototypes were discussed from the point of view of the system's users. In addition, load tests were carried out on the system using the JMeter software, which highlighted some of the strengths of the implementation and some points for future improvement but also proved that using an infrastructure shared by multiple destinations does not mean sacrificing the system's performance in general.

Chapter 6

Conclusions and Future Work

6.1 Final Considerations

The main aim of this dissertation was to prove that it is possible to develop a platform that allows DMS to be created much more quickly and economically by building an infrastructure shared by several destinations, where it would be possible to reuse the software components of a DMS as much as possible.

To achieve this goal, a Backend based on microservices was created, where each microservice consisted of one of the most abundant types of business in the world of tourism, as well as support microservices such as Reservations and Rating. In addition, to demonstrate that this backend was functional and that the functionalities of each service were suitable for each business, a Frontend for a management module was created to illustrate how the tourist offer could advertise its products and services on the system, and two portals for two different tourist destinations were created to demonstrate the possibility of more than one tourist destination using the same infrastructure.

Finally, the portals were created, and the management module was also used to carry out load tests on the system to prove that even if the infrastructure feeds multiple tourist destinations, it can be just as performant as a system implemented from scratch and unique to each tourist destination.

The Multi-Destination Management System therefore proves to be a viable alternative for destinations with low financial capacity, which can thus access a system of this type without having to give up optimum performance, stability, and functionalities.

6.2 Future work

This work has created the basis for a platform that allows multiple DMS to be made based on the same infrastructure and has been designed in such a way that it can be improved and new functionalities introduced quickly. In terms of future steps, the most essential thing in the first phase would be to optimize the way files and images are stored in the system's database, as this would guarantee a faster response time to requests. A compelling improvement that could make it possible to commercialize the system would be to create a more beautiful and

dynamic front end for the destination portals, which could promote the destinations and use the microservices implemented.

Finally, it would also be interesting to improve the operation of the booking microservice so that it would allow users to know which days are already booked and make it possible to book by hours and minutes and not just by whole days.

References

[1] C. L. Bunghez, “The importance of tourism to a destination’s economy,” *Journal of Eastern Europe Research in Business & Economics*, vol. 2016, 2016.

[2] J. Estevao, M. J. Carneiro, and L. Teixeira, “Destination management systems’ adoption and management model: Proposal of a framework,” *Journal of Organizational Computing and Electronic Commerce*, vol. 30, no. 2, 2020.

[3] D. Buhalis, “Information and telecommunications technologies as a strategic tool for small and medium tourism enterprises in the contemporary business environment,” pp. 254–275, 1994.

[4] C. Petti and G. Solazzo, “Architectural scenarios supporting e-business models for a dms,” in *Information and communication technologies in tourism 2007*, Springer, 2007.

[5] WTO, *E-business for tourism-practical guidelines for destinations and businesses*, 2001.

[6] C. A. Martins, “Determinantes da implementação de uma estratégia de negócio eletrónico por parte das organizações de gestão de destinos turısticos,” Ph.D. dissertation, Universidade de Aveiro (Portugal), 2018.

[7] D. Garlan, “Software architecture: A roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 91–101.

[8] “Software Architecture Patterns Understanding Common Architecture Patterns and When to Use them,” Microsoft. [Online]. Available: `https://get.oreilly.com/rs/107-FMS-070/images/Software-Architecture-Patterns.pdf`.

[9] N. Dmitry and S.-S. Manfred, “On micro-services architecture,” *International Journal of Open Information Technologies*, vol. 2, 2014.

[10] J. Zhao, S. Jing, and L. Jiang, “Management of api gateway based on micro-service architecture,” in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1087, 2018, p. 032 032.

[11] *What is a three-tier architecture?* [Online]. Available: `https://www.ibm.com/topics/three-tier-architecture`.

[12] S. Sulyman, “Client-server model,” *IOSR Journal of Computer Engineering*, vol. 16, pp. 57–71, Jan. 2014. DOI: `10.9790/0661-16195771`.

[13] D. Kunda and H. Phiri, “A comparative study of nosql and relational database,” *Zambia ICT Journal*, vol. 1, no. 1, pp. 1–4, 2017.

[14] “What is a relational database?” IBM. [Online]. Available: `https://www.ibm.com/topics/relational-databases`.

[15] M. Ilić, L. Kopanja, D. Zlatković, M. Trajković, and D. Ćurguz, “Microsoft sql server and oracle: Comparative performance analysis,” in *The 7th International conference Knowledge management and informatics*, 2021.

[16] R. Wodyk and M. Skublewska-Paszkowska, “Performance comparison of relational databases sql server, mysql and postgresql using a web application and the laravel framework,” *Journal of Computer Sciences Institute*, 2020.

[17] “PostgreSQL Vs MySQL: Different Databases For Different Use Cases,” Panoply. [Online]. Available: https://blog.panoply.io/postgresql-vs.-mysql.

[18] “PostgreSQL vs MySQL: The Critical Differences,” Integrate.IO. [Online]. Available: https://www.integrate.io/blog/postgresql-vs-mysql-which-one-is-better-for-your-use-case/.

[19] M. Mythily, A. Samson Arun Raj, and I. Thanakumar Joseph, “An analysis of the significance of spring boot in the market,” in *2022 International Conference on Inventive Computation Technologies (ICICT)*, 2022. DOI: 10.1109/ICICT54344.2022.9850910.

[20] P. Rawat and A. N. Mahajan, “Reactjs: A modern web development framework,” *International Journal of Innovative Science and Research Technology*, 2020.

[21] “Start a new react project,” Meta OpenSource, 2023. [Online]. Available: https://react.dev/learn/start-a-new-react-project.

[22] “Getting started with React,” Mozilla, 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started.

[23] L. De Lauretis, “From monolithic architecture to microservices architecture,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2019, pp. 93–96.

[24] *10 medidas para preparar a aplicação do regulamento europeu de proteção de dados*. [Online]. Available: https://www.sg.pcm.gov.pt/media/33598/06.pdf.

[25] *Spring Cloud Gateway*. [Online]. Available: https://spring.io/projects/spring-cloud-gateway.

[26] *Spring Cloud Netflix*. [Online]. Available: https://spring.io/projects/spring-cloud-netflix.

[27] *An Intro to Spring Cloud Security*. [Online]. Available: https://www.baeldung.com/spring-cloud-security.

[28] G. Mak, “Spring mvc framework,” in *Spring Recipes: A Problem-Solution Approach*, Springer, 2008.

[29] *Spring cloud config*. [Online]. Available: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/#_quick_start.

[30] J. Wang and J. Wu, “Research on performance automation testing technology based on jmeter,” in *2019 International Conference on Robots Intelligent System (ICRIS)*, 2019, pp. 55–58. DOI: 10.1109/ICRIS.2019.00023.

[31] V. Tiwari, S. Upadhyay, J. K. Goswami, and S. Agrawal, “Analytical evaluation of web performance testing tools: Apache jmeter and soapui,” in *2023 IEEE 12th International Conference on Communication Systems and Network Technologies (CSNT)*, 2023, pp. 519–523. DOI: 10.1109/CSNT57126.2023.10134699.

[32] G. Mahajan, D. V. Attar, and S. Kalamkar, “Generation of jmeter scripts for performance testing of moodle server,” in *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, 2022, pp. 2277–2281. DOI: 10.1109/ICAC3N56670.2022.10074284.