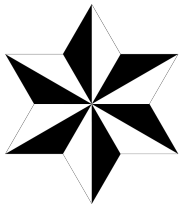

University of Brighton

Department of Computer Science

CI615 - Object-Oriented Design & Architecture

Design Patterns & Architecture Design and Documentation

Max Sherman

Supervisor: Ali Hamie

A report submitted in partial fulfilment of the requirements of
the University of Brighton for the degree of
Bachelor of Science in *Computer Science*

16/01/2025

Contents

Chapter 1

Design Patterns

1.1 Introduction

This chapter identifies and discusses five design patterns that can be applied to the architecture and design of the ABC Travel Agency web portal. Each pattern focuses on resolving a particular design challenge, with supporting UML diagrams to showcase its role in the system. Furthermore, the advantages and disadvantages of each design pattern are also discussed in the context of this system.

1.2 Factory pattern

The Factory pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to determine which class to instantiate (Gamma et al., 1995). In the context of this project, it can be used to create instances of travel services, such as flights, hotels, and car rentals, without specifying the exact class of the object to be created. The separation of object creation from the application logic enables the platform to support new services or service providers with minimal changes to existing code or logic (Larman, 2005). As a result, the factory design pattern promotes scalability and increases flexibility in the system design. It is particularly beneficial in scenarios where the application must adapt to new types of services or service providers dynamically, as it encapsulates the object creation process (Buschmann et al., 1996). Figure 1.1 shows the Factory pattern creating instances of travel services.

- **Advantages:** Promotes scalability and flexibility, as new services and service providers can be integrated with minimal changes to the existing codebase.
- **Disadvantages:** Increases code complexity, as it requires additional classes for implementation.

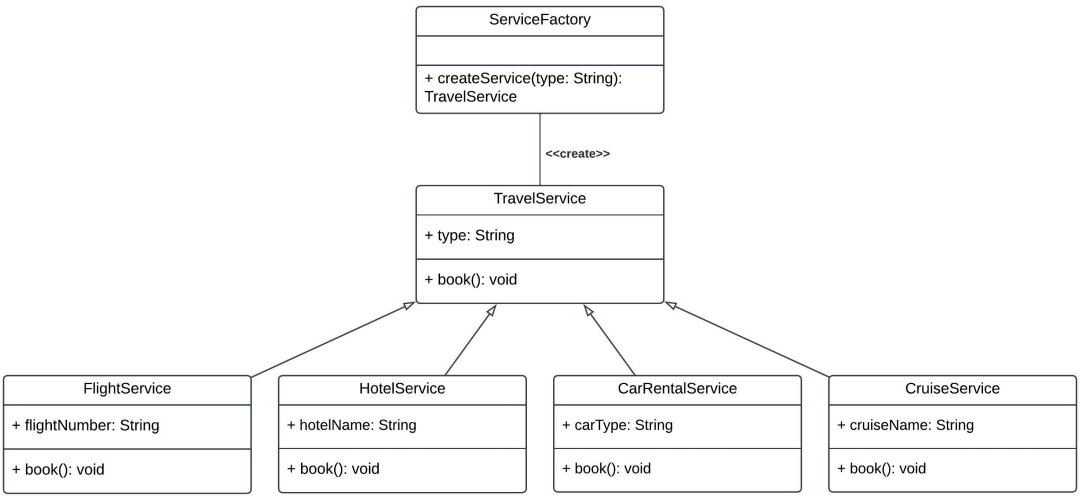

Figure 1.1: The Factory design pattern

1.3 Model-View-Controller (MVC) pattern

The Model-View-Controller (MVC) design pattern is an architectural design pattern that separates an interactive application into three components: Model, View, and Controller. This separation of concerns enhances flexibility and maintainability (Gamma et al., 1995, Buschmann et al., 1996).

Model: contains the core data and logic. It is independent of specific user inputs or user interface (UI) implementations (Taylor et al., 2010). The Model notifies the View whenever the state of the Model changes, which enables dynamic updates without direct coupling. In the context of the case study, the Model manages crucial data such as user profiles, bookings, and available travel services.

View: Responsible for visualising the data from the Model to the user through a UI. After retrieving the necessary data from the Model, the View converts the data into a user-friendly format, such as displaying search results, user account details, and booking confirmations. Multiple Views can be connected to a single Model, facilitating different representations including interfaces for mobile or desktop devices (Buschmann et al., 1996).

Controller: Handles user inputs, such as search queries, booking selections, and payment information, and modifies the state of the Model or updates the View accordingly. The Controller aids flexible user interactions, promoting both modularity and reusability (Buschmann et al., 1996).

The MVC pattern is essential for organising the web portal's structure, as it helps keep code modular and organised. Furthermore, it enables independent modification of components, which allows components to be updated or replaced without altering other critical components. Figure 1.2 presents the MVC Pattern showing interactions between the Model, View, and Controller components.

- **Advantages:** Promotes modularity by separating concerns between the logic, user interface, and user input handling. The separation of concerns also simplifies testing and debugging. Additionally, MVC supports multiple views which enhances flexibility and responsiveness (Gamma et al., 1995).

- **Disadvantages:** Increases complexity and can be excessive for small-scale applications. Managing the data flow between components can become cumbersome and Controllers can become dependant on specific Models, limiting the reusability (Buschmann et al., 1996).

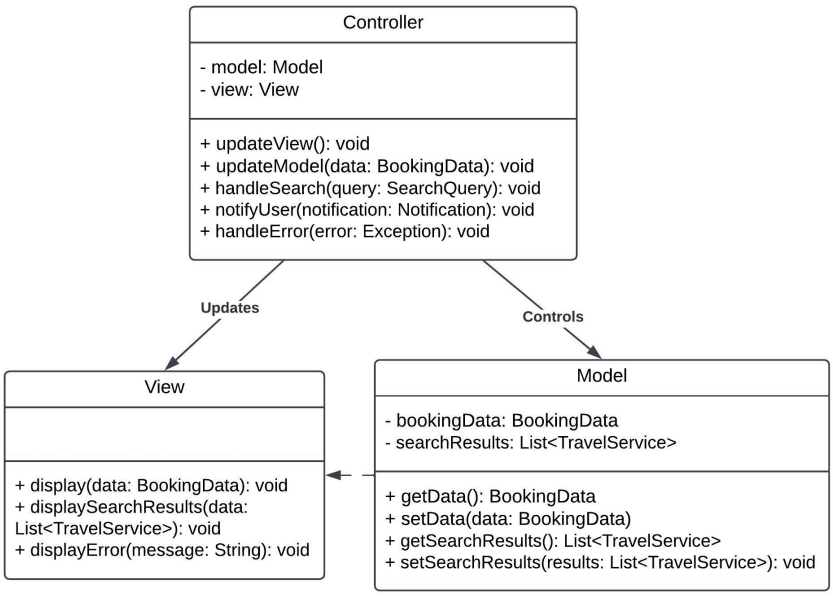

Figure 1.2: The Model-View-Controller design pattern

1.4 Observer pattern

The Observer pattern is a behavioural design pattern that establishes a one-to-many dependency between objects. When the subject changes its state, all dependent observers are notified and updated automatically. This design increases flexibility and modularity by encouraging loose coupling between components (Gamma et al., 1995).

Subject: Maintains a list of observers and provides methods for attaching, detaching, and notifying them. For the web portal, the notification service acts as the subject, sending updates on relevant offers.

Observer: Defines an interface for receiving updates from the subject. In this project, registered users act as observers, receiving notifications on personalised offers or bookings.

ConcreteSubject and ConcreteObserver: When the ConcreteSubject's state changes, such as when a new offer becomes available, it notifies all observers and stores the information. ConcreteObservers then update their state and process notifications accordingly, ensuring consistency across the system.

The Observer pattern is a well-suited choice for implementing the notification feature in the travel booking web portal. It allows the platform to notify registered users of relevant special offers based on their preferences without directly coupling the notification service to individual user profiles. Figure 1.3 depicts the Observer pattern where users (observers) receive updates from the notification service (subject)

- **Advantages:** Promotes modularity and loose coupling between the subject and its observers. Enables adding and removing observers dynamically without modifying the

Subject. New types of notifications can be added easily, supporting the system's modifiability (Gamma et al., 1995).

- **Disadvantages:** Can lead to performance overhead if there are too many observers, such as during promotional events with a large number of simultaneously notified users, or if the notification logic is too complex. The loose coupling between the subject and observers can make debugging their communication challenging.

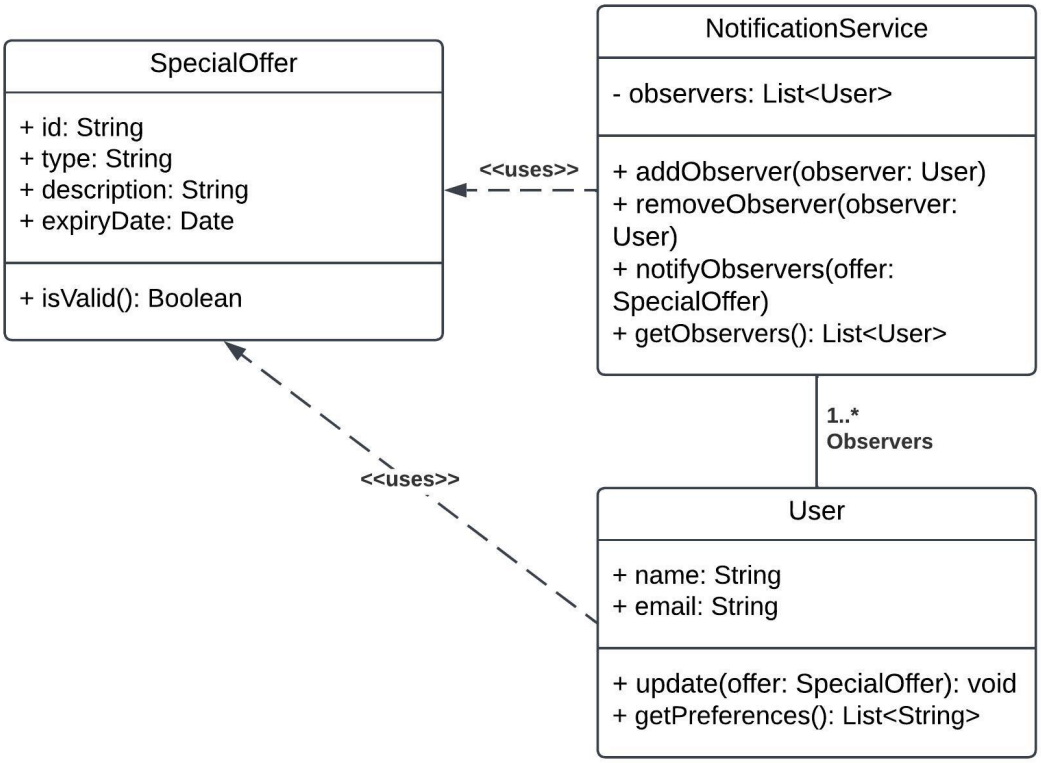

Figure 1.3: The Observer design pattern

1.5 Singleton pattern

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. This design pattern is particularly useful for managing shared resources in a controlled manner (Gamma et al., 1995).

Implementation: The Singleton pattern is implemented by making the constructor of the class private to prevent direct instantiation. Instead, a static method called `getInstance` creates and returns the single instance of the class if it does not already exist. If the instance has already been created previously, the method will only return the instance. This method ensures that the instance is created only when needed, a technique known as lazy initialisation.

In the context of the travel booking web portal, the Singleton pattern can be applied to manage the database connection and payment gateway instance. This approach eliminates the need to create and manage multiple connections and ensures consistent interactions with critical resources. Figure 1.4 shows the Singleton Pattern applied to the database connection.

- **Advantages:** Reduces memory usage and improves performance by avoiding duplicate instances. Furthermore, the Singleton pattern provides a global access point to shared resources. It simplifies resource management and ensures consistency across the system (Gamma et al., 1995).
- **Disadvantages:** Can hinder unit testing as it introduces a global state. It may lead to tight coupling, which can compromise the system's modifiability and scalability.

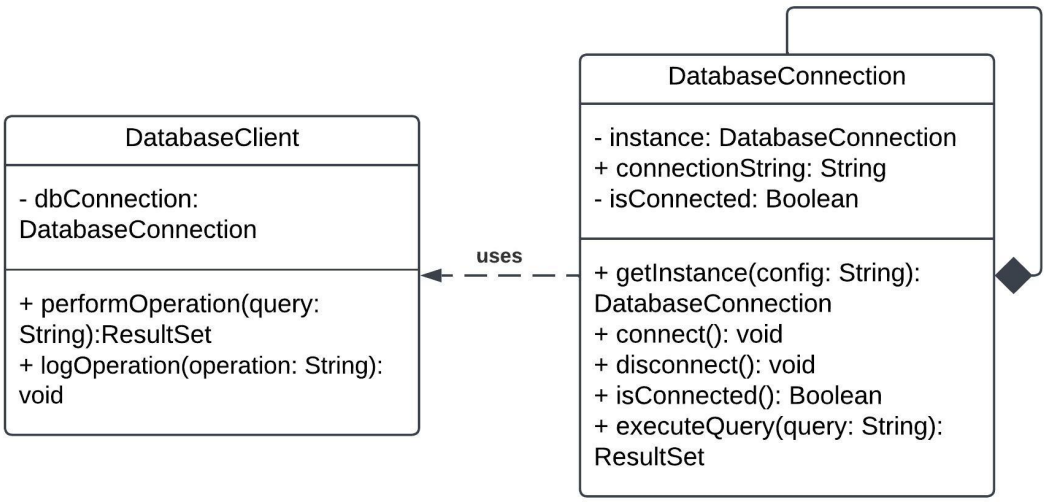

Figure 1.4: The Singleton design pattern

1.6 Strategy pattern

The Strategy Pattern is a behavioural design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the algorithm to vary independently from the clients that use it, enhancing flexibility and maintainability (Gamma et al., 1995). The pattern is implemented by defining a common interface for all strategies. Each concrete strategy implements this interface, encapsulating a specific algorithm. The Context class maintains a reference to a Strategy object and assigns the algorithmic work to this instance. In the travel booking web portal, the Strategy pattern can be used to implement different payment methods, such as credit card, PayPal, or other payment options. As a result, the appropriate payment method can be dynamically selected and applied at runtime. Figure 1.5 shows the Strategy Pattern for the payment process, with separate classes for each payment type.

- **Advantages:** Promotes the open-closed principle (open for extension, closed for modification) and thus scalability, allowing new strategies to be added without modifying existing code. Improves maintainability by replacing complex conditional logic with polymorphism. Enhances flexibility by allowing dynamic selection of algorithms (Gamma et al., 1995).
- **Disadvantages:** Increases the number of classes, which may add complexity to the system. This can impact performance and security, both of which are critical for the

payment processing module. Clients must understand the differences between strategies to choose the most appropriate one.

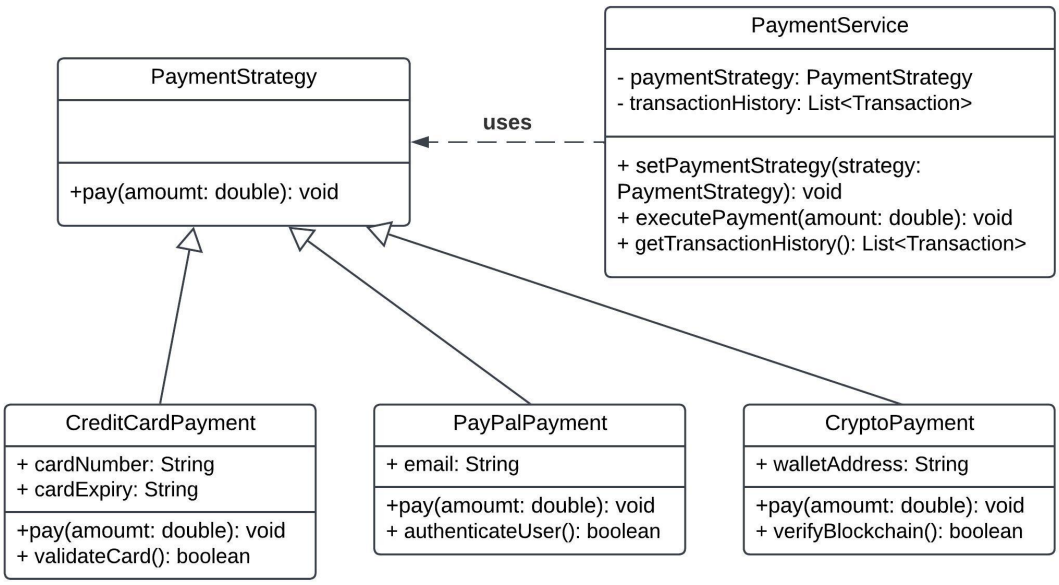

Figure 1.5: The Strategy design pattern

1.7 Summary of design patterns

This chapter covered five design patterns: Factory, MVC, Observer, Singleton, and Strategy. Each of these design patterns address specific challenges in the design of the ABC Travel Agency web portal. The Factory pattern promotes scalability by simplifying object creation for travel services. The MVC pattern enhances modularity by separating concerns between logic, user interfaces, and user interactions. The Observer pattern implements the notification system, enabling dynamic updates to users with loose coupling. The Singleton pattern ensures efficient resource management by maintaining single instances for critical components. Finally, the Strategy pattern provides flexibility in managing multiple payment methods. Together, these patterns contribute to a secure, scalable, modifiable and well-performing system design.

Chapter 2

Software Architecture Design

This chapter covers the software architecture design for the ABC Travel Agency web portal. The architecture aims to satisfy the functional and quality requirements specified for the system, including security, scalability, modifiability, performance, and availability. The Software Architecture Document (SAD) includes descriptions of key architectural requirements, architectural elements, architecture and design diagrams, and architectural styles.

2.1 Architectural Requirements

The architectural requirements for the ABC Travel Agency web portal cover both functional and non-functional aspects. The requirements ensure the system meets user expectations and business needs. This section details the use cases and quality attribute scenarios, which form the foundation of the software architecture.

2.1.1 Use Cases

Use cases describe the functional requirements of the system by detailing the interaction between users and the system in order to achieve specific goals. Each use case represents an Elementary Business Process (EBP) that leaves the system in a consistent state (Larman, 2005). The system includes the following use cases:

- **UC1: User registration and profile management.**
  - **Actors:** Unregistered users, registered users.
  - **Description:** Users can create a new account, manage personal information, and set travel preferences.
  - **Preconditions:** The user accesses the registration or profile management interface.
  - **Postconditions:** A new account is created or existing profile information is updated successfully.
- **UC2: Search for travel services.**
  - **Actors:** Unregistered users, registered users.
  - **Description:** Users can search for flights, hotels, or car rentals individually or in combination.
  - **Preconditions:** The user has access to the web portal and provides valid search criteria.

  - **Postconditions:** Relevant travel service results are displayed to the user.

- **UC3: Book travel services.**
  - **Actors:** Registered users.
  - **Description:** Users can select and book travel services.
  - **Preconditions:** The user is registered, logged in, and has selected a travel service.
  - **Postconditions:** The selected service is booked, and the user receives a booking confirmation.

- **UC4: Process payments.**
  - **Actors:** Registered users, external payment gateway.
  - **Description:** Users can securely complete payment transactions to book travel services.
  - **Preconditions:** The user has selected a service to book and provided payment details.
  - **Postconditions:** The payment is processed, and the user receives a payment confirmation.

- **UC5: Receive notifications.**
  - **Actors:** Registered users.
  - **Description:** Users receive emails about special offers based on their travel preferences.
  - **Preconditions:** The user has opted to receive email notifications and set their travel preferences.
  - **Postconditions:** Relevant emails containing special offers are sent to the user.

- **UC6: Mobile accessibility.**
  - **Actors:** Unregistered users, registered users.
  - **Description:** Users can access the portal through mobile devices such as iPhones and Android smartphones.
  - **Preconditions:** User has a mobile device with internet access.
  - **Postconditions:** The portal adapts the interface to the mobile device.

The use case diagram shown in figure 2.1 provides a visual representation of the interactions between the system's actors and its use cases.

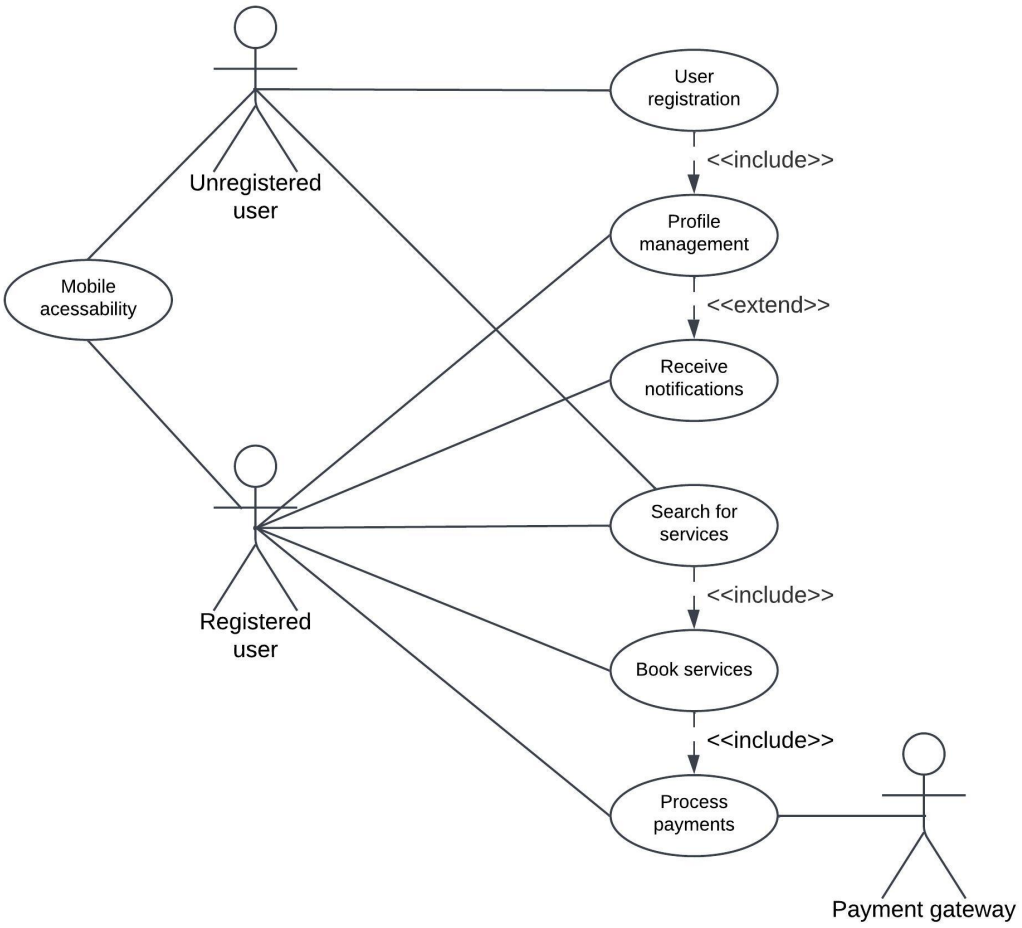

Figure 2.1: Use Case Diagram for ABC Travel Agency

2.1.2 Quality Attribute Scenarios

Quality Attribute Scenarios (QAS) define the non-functional requirements of the system, ensuring it meets security, scalability, modifiability, performance, and availability expectations. Each scenario is structured into six key components: Source, stimulus, environment, artifact, response, and response measure (Bass et al., 2013). The QAS provide a structured method for defining and measuring quality attributes. As a result, they offer precise guidance for architectural decisions and system design (Gorton, 2011). Table 2.1, 2.2, 2.3, 2.4, and 2.5 provide detailed examples of QAS that cover each quality attribute (Bass et al., 2013).

Source	A user attempting to access sensitive payment information.
Stimulus	Access request with invalid credentials.
Artifact	Sensitive payment details.
Environment	Normal operational conditions.
Response	The system blocks access and logs the attempt.
Response Measure	The response is completed in under one second.

Table 2.1: Quality Attribute Scenario: Security

Source	Users searching for travel services during periods of high demand.
Stimulus	A significant increase in simultaneous search requests.
Artifact	The search functionality.
Environment	Normal operation during periods of high demand.
Response	The system processes all simultaneous search requests effectively.
Response Measure	An average response time of 2 seconds or less.

Table 2.2: Quality Attribute Scenario: Scalability

Source	An administrator updating the list of travel service providers.
Stimulus	A request to add or remove a service provider.
Artifact	Service provider configuration and integration modules.
Environment	Normal administrative maintenance conditions.
Response	The system utilises the updated provider list without impacting existing features.
Response Measure	Updates can be completed within 3 hours of receiving the request and without disruptions to ongoing user operations.

Table 2.3: Quality Attribute Scenario: Modifiability

Source	A user completing a payment transaction.
Stimulus	Submission of payment details.
Artifact	Payment gateway system.
Environment	Normal operational conditions.
Response	The payment is validated and processed.
Response Measure	An average response time of 2 seconds or less.

Table 2.4: Quality Attribute Scenario: Performance

Source	A user trying to book services during a system maintenance period.
Stimulus	A booking request when the system is undergoing maintenance.
Artifact	Booking service and database.
Environment	Maintenance mode with limited system functionality.
Response	The system informs the user of the maintenance period and requests the user to try again later.
Response Measure	The notification is displayed within 1 second of the booking attempt.

Table 2.5: Quality Attribute Scenario: Availability

2.1.3 Architectural Requirements Traceability Matric

To ensure traceability between the functional and non-functional requirements, Table 2.6 provides a Requirement Traceability Matrix. This table maps the UCs to the corresponding

QAS, demonstrating how the system's functional requirements align with and support the architectural quality objectives.

Use Case	Description	Quality Attribute Scenarios
UC1	User registration and profile management.	Security (see Table 2.1), Modifiability (see Table 2.3).
UC2	Search for travel services.	Scalability (see Table 2.2), Performance (see Table 2.4).
UC3	Book travel services.	Performance (see Table 2.4), Availability (see Table 2.5).
UC4	Process payments.	Security (see Table 2.1), Performance (see Table 2.4).
UC5	Receive notifications.	Modifiability (see Table 2.3), Availability (see Table 2.5).
UC6	Mobile accessibility.	Scalability (see Table 2.2).

Table 2.6: Requirement Traceability Matrix: Mapping Use Cases to Quality Attribute Scenarios

2.2 Architectural Elements

This section identifies the key architectural components, connectors, and interfaces of the ABC Travel Agency web portal. Each element includes a description of their role and responsibilities. The elements are aligned with the system's functional and non-functional requirements.

2.2.1 Components

- **UI Module:**
  - **Description:** Manages the user interface, allowing users to interact with the system, either via desktop or mobile platforms.
  - **Responsibilities:** Handles user input, displays search results, and facilitates booking procedures, such as selecting flights or providing payment details.
  - **Dependencies:** Relies on the Business Logic Layer for operations.
- **Business Logic Layer:**
  - **Description:** Manages the core functionalities, such as search, booking, and payment handling.
  - **Responsibilities:** Processes user requests, validates input data, applies business rules, and interacts with external APIs.
  - **Dependencies:** Relies on the Data Layer, Authentication Service, and external APIs.
- **Data Layer:**
  - **Description:** Manages database access, including retrieving and saving data.
  - **Responsibilities:** Facilitates data storage and retrieval. Stores data such as user profiles, bookings, and system configurations.

  - **Dependencies:** Relies on a relational database management system.

- **Authentication Service:**
  - **Description:** Manages user registration and login authentication.
  - **Responsibilities:** Verifies user credentials.
  - **Dependencies:** Relies on external authentication APIs, such as OAuth 2.0.

- **Notification Service:**
  - **Description:** Sends email notifications for booking confirmations and special offers.
  - **Responsibilities:** Creates and sends email messages based on triggers from the Business Logic Layer.
  - **Dependencies:** Relies on external email services.

- **Payment Service:**
  - **Description:** Manages secure payment transactions.
  - **Responsibilities:** Handles payment processing through external gateways.
  - **Dependencies:** Relies on external payment processors.

2.2.2 Connectors

- **RESTful API:**
  - Connects the UI Module to the Business Logic Layer.
  - Ensures stateless communication using HTTP methods.

- **Database Connector:**
  - Allows the Data Layer to interact with the database.
  - Supports SQL queries for saving and retrieving data.

- **External API Connectors:**
  - Interacts with airline, hotel, car rental, payment, and email services.
  - Utilises standard API protocols such as JSON over HTTPS.

2.2.3 Interfaces

- **UI:**
  - Responsive front-end built using HTML, CSS, and JavaScript frameworks. Adapts to user devices accordingly (desktop or mobile).
  - Communicates with the back-end through RESTful APIs.

- **Authentication API:**
  - Supports functionalities such as user login, registration, and password recovery.
  - Secured using OAuth 2.0 and HTTPS.

- **External Service APIs:**
  - Implements standard interfaces for airline, hotel, and car rental bookings.
  - Provides real-time availability and pricing.
- **Payment API:**
  - Enables secure payment processing.
  - Supports multiple payment methods (credit card, PayPal, etc.).
- **Notification API:**
  - Facilitates sending or scheduling notifications.
  - Integrates with external email services.

2.3 Architecture and Design Diagrams

To visualise the architectural structure, the following UML diagrams are provided:

- **Package Diagram:** Figure 2.2 shows logical grouping of system components.
- **Component Diagram:** Figure 2.3 depicts individual components and their relationships.
- **Deployment Diagram:** Figure 2.4 illustrates deployment across physical nodes, such as the web server, database server, and mobile interface.
- **Sequence Diagram:** Figrue 2.5 Represents typical interactions, such as user login, search, booking, and payment processes.

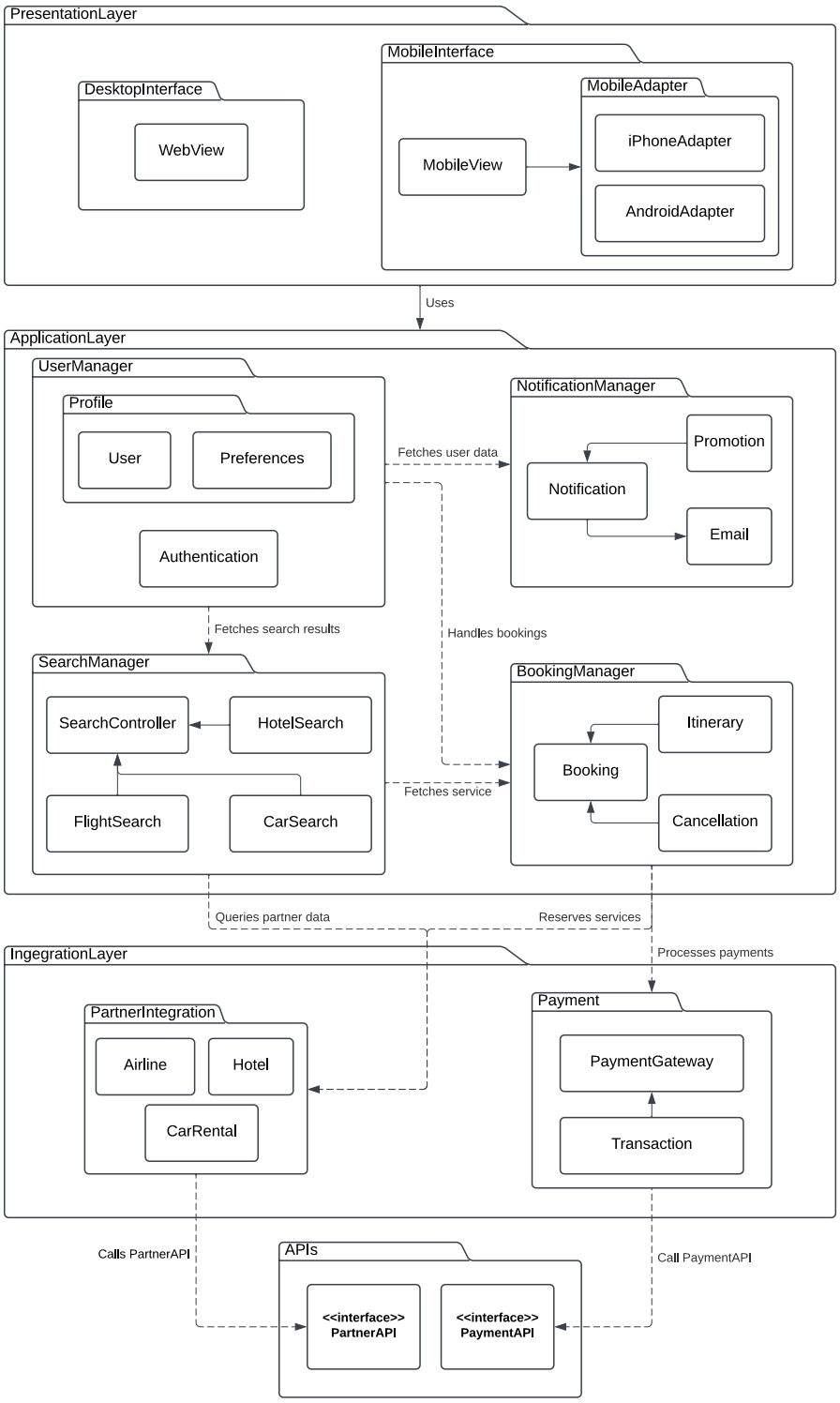

Figure 2.2: The package diagram

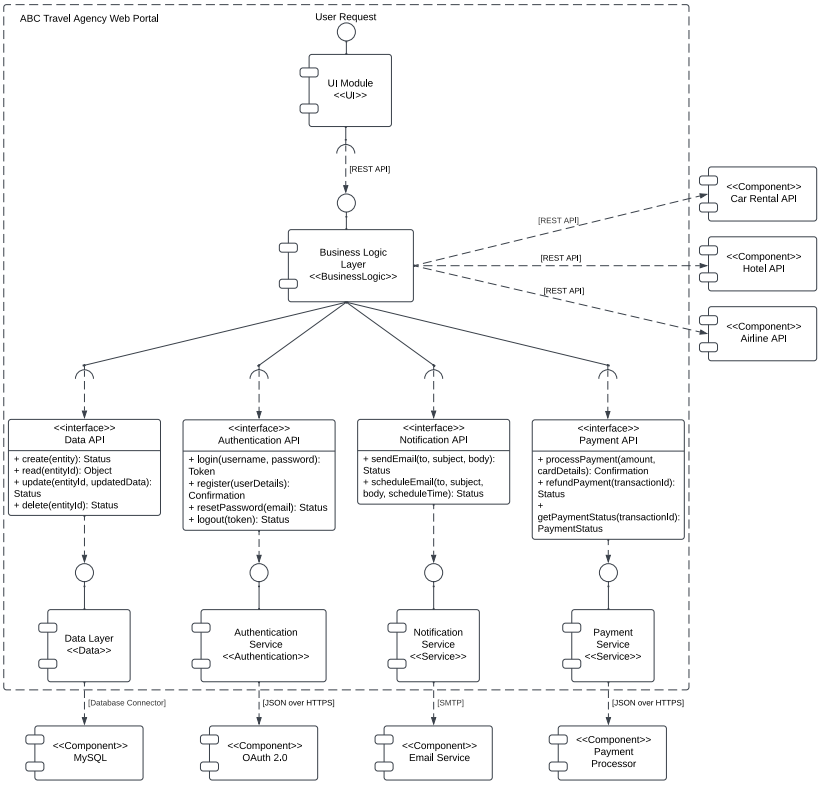

Figure 2.3: The component diagram

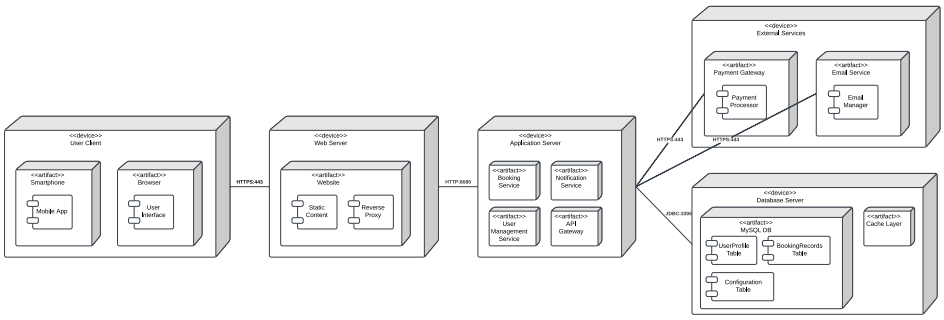

Figure 2.4: The deployment diagram

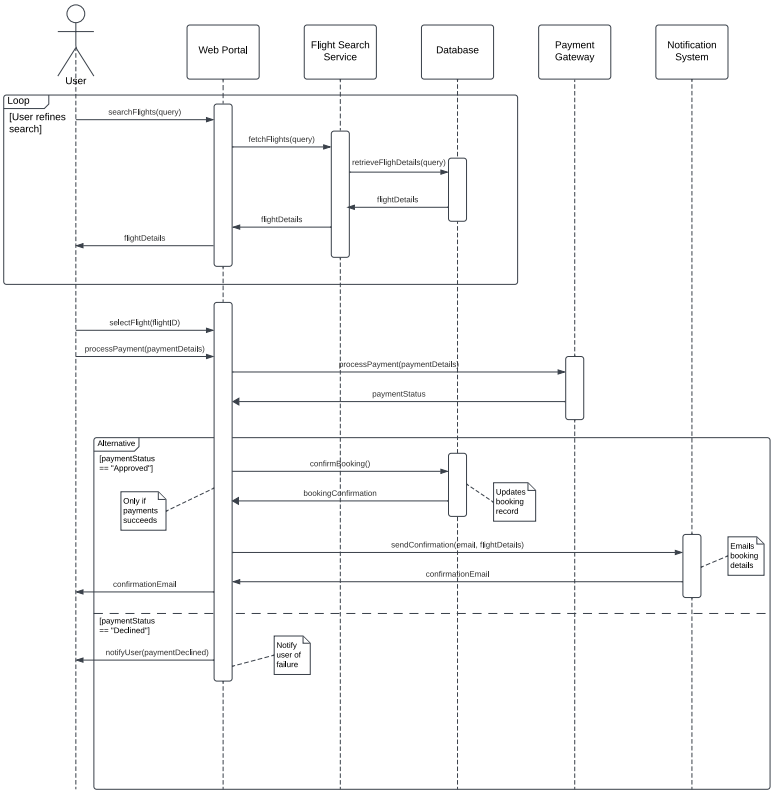

Figure 2.5: The sequence diagram

2.4 Architecture Solution Using Architectural Styles

This section describes the architectural styles used for designing the software architecture of the ABC Travel Agency web portal. The proposed architecture provides all the necessary quality attributes, as defined in Section 2.1.2.

2.4.1 Architectural Styles and Patterns

The ABC Travel Agency web portal employs a combination of architectural styles to address various system requirements.

- **Layered Architecture:** The layered architecture separates the system into distinct layers: Presentation, Business Logic, and Data Access. Each layer communicates with other layers through defined interfaces, which allows for a clear separation of concerns. For instance, the Presentation Layer interacts with the user through a responsive web

interface, while the Business Logic Layer processes the requests and applies the rules determined by the business. The Data Layer manages the storage and retrieval of relevant data. This style promotes modifiability since changes in one layer do not directly affect other layers (Bass et al., 2013). On the other hand, performance degradation is a potential drawback due to the additional overhead of communications between layers.

- **Micro-services Architecture:** The micro-services architecture breaks the system down into loosely coupled, independent services. For instance, accessing service providers, processing payments, and sending notifications are partitioned into separate micro-services. This approach improves scalability since the services can improve independently. Furthermore, it enhances robustness as the failure of one service will not impact other services (Gorton, 2011). However, micro-service management and coordination can become complex rapidly, which can require orchestration or service discovery tools.

- **Client-Server Architecture:** The web portal benefits from the client-server architecture because it separates the client-side from the server-side. Clients, such as web browsers and mobile devices, send requests to the server. Servers, such as web and database servers, process these requests and return the appropriate responses. This architecture improves scalability because a server can handle multiple clients simultaneously. Moreover, the main logic is provided by the server, which streamlines client-side updates. However, dependence on a centralised server leads to a single point of failure. This can be avoided by implementing load balancing and failover mechanisms (Garlan and Shaw, 1994)

- **Model-View-Controller (MVC) Pattern:** The MVC pattern is applied to the design of the user interface, separating it into three components: the Model, View and Controller. Like the layered architecture, it promotes the separation of concerns. In the context of the travel portal, the Model represents data, the View is responsible for presenting the data and the Controller handles user interactions. The pattern enhances maintainability and testability since changes can be limited to a single component. However, ensuring proper coordination between the components can lead to additional complexity (Buschmann et al., 1996, Gamma et al., 1995). For a detailed description of the MVC pattern, including its advantages and disadvantages, see Section 1.3.

2.4.2 Architectural Views

To provide a comprehensive understanding of the system's architecture, the following architectural views are linked to UML diagrams (Kruchten, 1995):

- **Logical View:** Represents the major system components and their interactions. This view highlights the layered structure and identifies key components such as the Presentation Layer, Business Logic Layer, and Data Layer (see Figures 2.2 and 2.6).

- **Development View:** focuses on the organisation of the system's codebase into modules, packages, or components, illustrating how the software is structured to support maintainability, scalability, and collaboration among developers (see Figures 2.2 and 2.3)

- **Process View:** Illustrates concurrency and runtime behavior, including key processes like booking and notification sending (see Figures 2.5 and 2.7).

- **Deployment View:** Maps software components to physical infrastructure. This view addresses performance and scalability considerations (see Figures 2.3 and 2.4)

- **Use Case/Sequence Diagram:** See Section 2.1.1 for the Use Cases and Figure 2.5 for the sequence diagram.

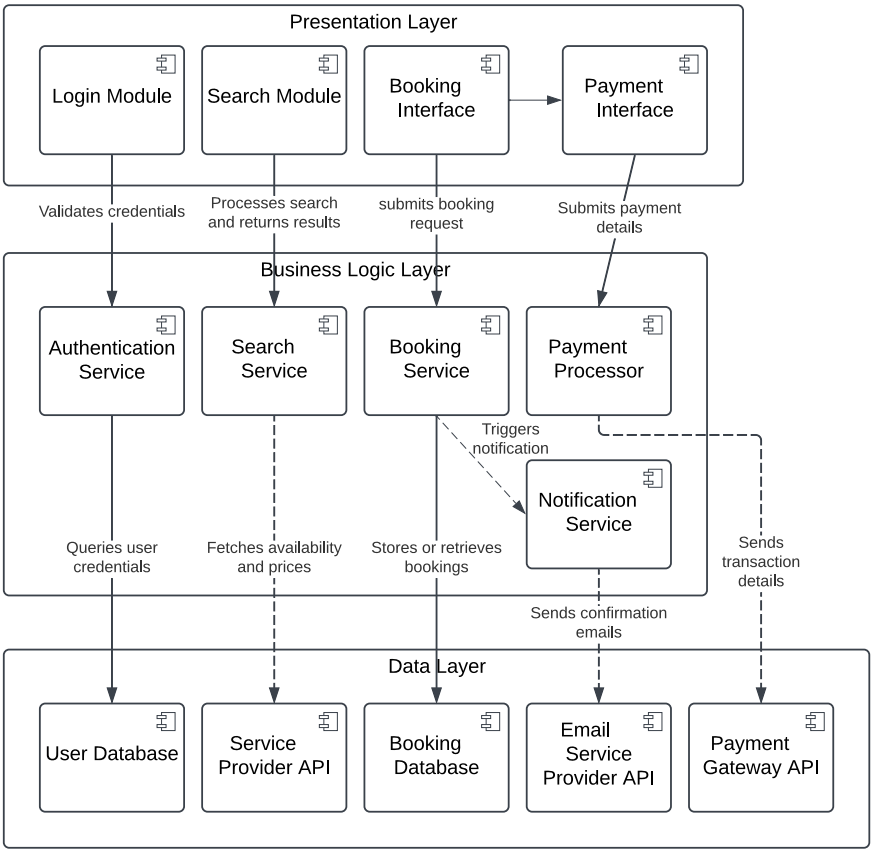

Figure 2.6: Logical View of the ABC Travel Agency Portal

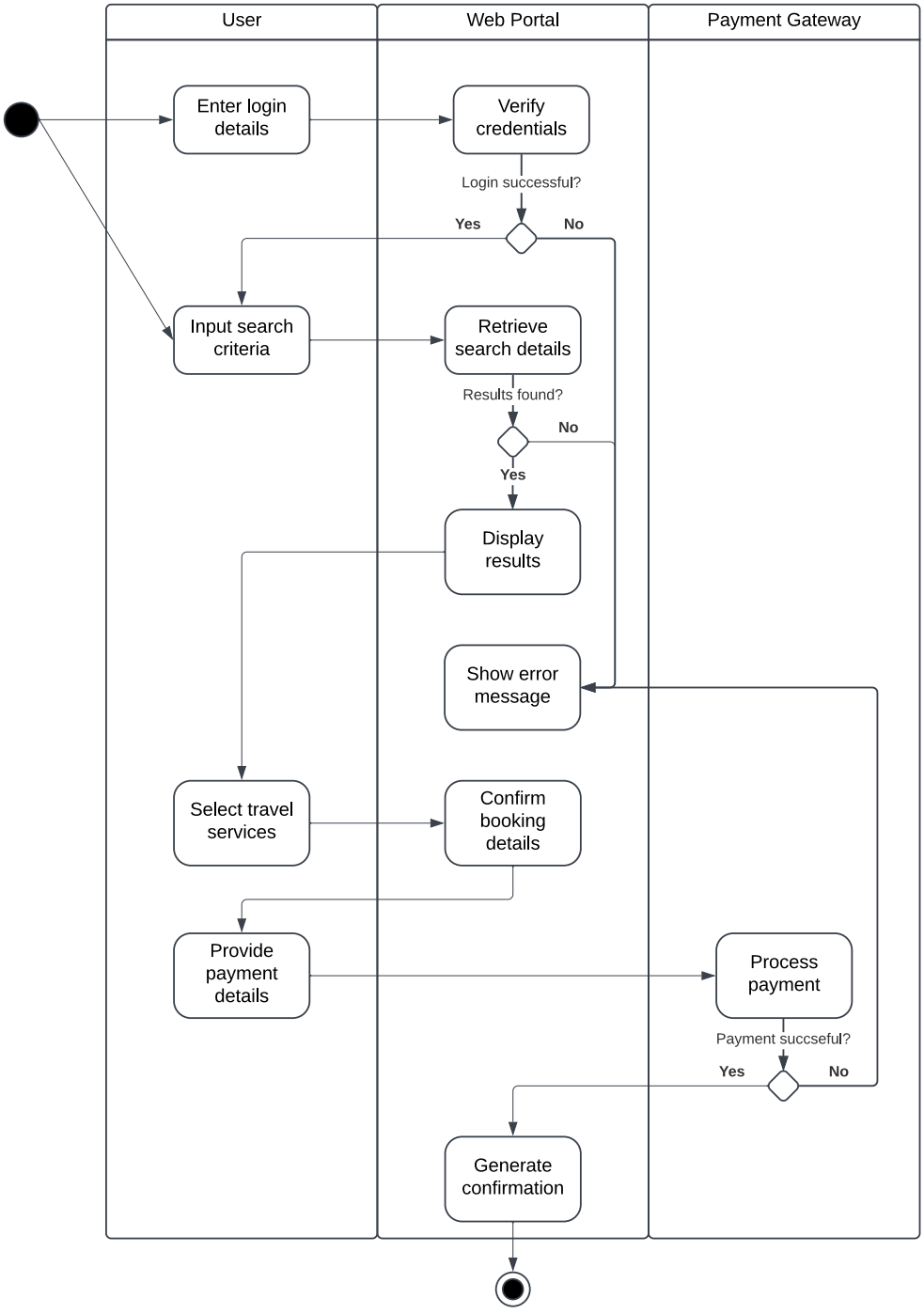

Figure 2.7: Logical View of the ABC Travel Agency Portal

2.4.3 Justification

The architectural styles covered in the previous section address the following quality attributes:

- **Scalability:** Micro-services can scale independently and in proportion to their specific demand.
- **Modifiability:** The layered, client-server, and MVC architectures allow changes to only affect a single component, which simplifies maintenance and updates.
- **Performance:** Both the client-server and micro-services architectures benefit from load balancing, which is responsible for distributing traffic across multiple servers. Not only does this remove the single point of failure and reduce bottlenecks, but it also ensures that the clients receive responses quickly and efficiently.
- **Availability:** Failover mechanisms are in place for the client-server and micro-services architectures to ensure that the system remains fully operational during server failures.

Chapter 3

Reflection Report

This reflection report evaluates the feasibility of the proposed architecture for the ABC Travel Agency web portal. The architecture carefully considers security, scalability, modifiability, performance, and availability, whilst addressing the functional and non-functional requirements of the system.

3.1 Feasibility of the Architecture

The chosen architecture effectively combines layered architecture, micro-services, client-server architecture and the MVC pattern. The layered architecture and MVC pattern distinctly separate concerns, facilitating maintenance and updates. The micro-services promote a modular approach which enhances the system's scalability (Gorton, 2011). The client-server architecture optimises the use of resources and manages interactions between users and the system (Garlan and Shaw, 1994). The architecture also leverages existing web development technologies, such as RESTful APIs and relational databases. By using well-established and proven frameworks, the development team can effectively create a system that addresses all key quality attributes. Collectively, these architectural styles balance complexity with flexibility, showcasing high overall feasibility.

3.2 Rationale for Chosen Architecture

The decisions made during the design of the architecture were supported by their contributions towards quality attributes. The layered architecture enhances modifiability and maintainability by encompassing changes to specific layers (Bass et al., 2013). This approach is especially advantageous for web applications since they often require diverse functionalities, such as user management, search features, and booking capabilities.

A key rationale for the chosen architecture is the separation of concerns, which is applied not only to the UI module but to the whole architecture of the portal. This decision was driven by the necessity of scalability and ease of maintenance, which is prominent for an application that is expected to increase in size. Each component can be enhanced or replaced independently, without impacting other components or layers. For additional justification, see Section 2.4.3.

3.3 Connections to Relevant Technologies and Software

The chosen architecture aligns with web application industry standards and best practices. Back-end frameworks such as Django and Spring Boot effectively support layered and MVC architectures and facilitate RESTful service development. Front-end frameworks such as React or Vue.js effectively integrate with the back-end, ensuring a responsive user interface for various devices.

For the database, using a relation database management system (RDBMS) such a MySQL or PostgreSQL offer a consistent approach to storing and accessing potentially sensitive information, including booking and payment data. Alternatively, cloud platforms such as AWS or Azure offer scalable and distributed solutions, including load balancing and server redundancy, further contributing towards the scalability and availability quality attributes.

3.4 Past Implementations and Personal Experience

Based on previous projects with similar designs, the layered approach is an effective approach to designing a large-scale application. In my personal experience, the performance of applications with a layered design remains high whilst simplifying development, maintenance, and testing. Similarly, the MVC pattern, whilst just as effective, is more suitable for small-scale applications or individual modules.

Additionally, I have noticed that the micro-services approach promotes loose coupling with external services. Reducing dependencies on external services allows services to be updated or replaced if, for instance, they are outdated or do not meet new requirements. This improves scalability and overall robustness, particularly for components that handle high traffic. Overall, in my option, these design choices are essential for developing and managing a robust and efficient large-scale system.

3.5 Alternative Solutions

Alternative architectural styles such as a monolithic or full micro-service architecture were considered during the design of the architecture. However, considering their impact on important quality attributes, they were deemed unsuitable for this case study. A monolithic approach would impede scalability and modifiability, impacting the system's ability to adapt to future demands. Unlike the monolithic approach, a fully micro-services-based approach is scalable; however, as mentioned in Section 2.4, the approach would introduce significant complexity, more specifically in managing the data and communication between services Gorton (2011). Given the scope and context of this case study, the hybrid solution was chosen to effectively balance maintainability and flexibility.

3.6 Conclusion

The architectural design of the ABC Travel Agency web portal effectively aligns with the functional and non-functional requirements detailed in Section 2.1. By incorporating various architectural styles and patterns, the chosen architecture provides a scalable, modifiable, and reliable solution that supports current demands and enables future improvements.

References

Bass, L., Clements, P. and Kazman, R. (2013), *Software Architecture in Practice*, 3rd edn, Addison-Wesley, Upper Saddle River, NJ, USA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996), *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, Chichester, England.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, USA.

Garlan, D. and Shaw, M. (1994), An introduction to software architecture, Technical Report CMU-CS-94-166, Carnegie Mellon University, Pittsburgh, PA. Also published in Advances in Software Engineering and Knowledge Engineering, Volume I, edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993. Also appears as CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.

Gorton, I. (2011), *Essential Software Architecture*, 2nd edn, Springer, Berlin, Germany.

Kruchten, P. (1995), 'Architectural blueprints—the '4+1' view model of software architecture', *IEEE Software* **12**(6), 42–50.

Larman, C. (2005), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd edn, Prentice Hall, Upper Saddle River, NJ, USA.

Taylor, R., Medvidovic, N. and Dashofy, E. (2010), *Software Architecture: Foundations, Theory, and Practice*, Wiley, Hoboken, NJ, USA.
URL: *https://dl.acm.org/doi/abs/10.1145/1810295.1810435*