Software Design Document for project Airline Management System

Finn Krappitz (5123009)
Tom Knoblach (5123034)
Jasmin Wander (5123060)
Joshua Pfennig (5123097)
Supervised by: Prof. Dr. Peter Braun

February 12, 2025

Contents

1 Introduction

1.1 Domain Summary

We developed a comprehensive Airline Management System that efficiently handles flights, ticketing for both customers and employees, and management of airline operations. The system supports multiple airlines as subsidiaries, each with its own

team of employees, enabling decentralized management while maintaining overall control. Additionally, it provides well-defined ports for seamless communication with databases and external APIs, facilitating integration and data exchange with other systems. For payment validation, the system integrates with Stripe, ensuring secure and efficient processing of transactions. Authentication with THWS username and password, as well as token-based authentication, was prepared. However, due to the lack of proper error handling support in the GraphQL client, it was not utilized and commented out.

1.2 Use case description

All entities in the system provide the necessary CRUD operations, ensuring efficient management of key resources. When a ticket is created, an email is automatically sent to the client's provided email address, including a detailed invoice. This ensures clients have instant access to all relevant booking information. Initializing a new client or employee guarantees that the creation is only successful if a valid german phone number and email address is provided.

Upgrading a seating class is validated by checking the availability of enough seats in the desired class. If there are not enough seats available, the customer is informed that the upgrade is not possible, maintaining transparency and preventing overbookings. Similarly, when upgrading luggage weight, the system performs availability checks and increases the price by a higher amount than if the luggage weight were set correctly at the time of the initial booking. This pricing model ensures that upgrades are accurately reflected in the final cost. As a simplification, only the weight and not the size of the luggage is taken into consideration.

Cancellations can be processed by either the client or the airline, affecting all connected tickets. In both scenarios, an email confirmation is sent, ensuring that clients are notified of the cancellation.

2 System Architecture

Our Airline Management System uses a hexagonal architecture, separating the core business logic from the external systems, making the application more maintainable and testable.

2.1 Domain Layer

The domain layer contains the core business logic and domain models. It is independent of any external systems or frameworks. In our project, this includes entities like *DomainAirline*, *DomainAirplane*, *DomainAirport*, *DomainClient*, *DomainEmployee*, *DomainFlight*, and *DomainTicket*. Common programming paradigms like inheritance have been used for the generalization of *DomainEmployee* and *DomainClient* with the *DomainPerson* class. This layer also includes the services and use cases that implement the business logic, defining the rules and operations that govern how the business works. For example, the class *TicketService* is responsible for ensuring that overbooking does not occur. With the help of the asychronical encryption algorithm AES, the payment api key and email server password have been encoded, ensuring secure storage of those credentials. In addition, the domain layer

defines ports, which are interfaces for interacting with external systems like an API and a database. They decouple the core business logic from the implementation details wihtin the adapters that are outside of the domain layer.

The database ports provide a corresponding repository class for every entity that extends a generic CRUD template, promoting consistency and reducing redundancy in database operations. The API operations additionaly provide all the use cases accesible for the end user. Nonetheless, since it happens to be an employee api and not one for an administrator, the delete options are not provided for most of the models to restrict the access.

2.2 Infrastructure Layer

The infrastructure layer provides a persistence and GraphQL API component, also providing the necessary adapters implementing the domains ports and mappers to communicate with the domain.

Firstly, the persistence component within the infrastructure layer handles all interactions with the database. The database is modeled as a file-based H2 database. Hibernate has been used as the ORM-framework to create the tables including joining multiple tables like *Employee* and *Flight*. Furthermore, their necessary CRUD operations for persisting the changes in the database are implemented and utilize the template pattern [1]. This approach offered several advantages, including improved code reusability and the ability to centralize common database operations in a template, reducing code duplication. It also helped standardize the process for interacting with the database, making it easier for team members to understand and maintain. Furthermore, employing the singleton pattern [2] ensures that only a single instance of *EntityManagerFactory*, responsible for loading the database configuration, is created throughout the entire application.

The API layer utilizes GraphQL to facilitate communication between the external world like the GraphQL client and the core domain. It explicitly defines models and input-models, which are specified in the GraphQL schema file. This schema file serves as the contract between the client and the server, defining the available queries, mutations, and the shape of the data. Each query and mutation is mapped to a corresponding resolver that interacts with the domain layer with the help of the provided adapters to fetch or modify data as necessary. These resolvers are organized in one file per model, to maintain modularity and clarity. Further details can be found in section 3.

2.3 Challenges during implementation

Adopting the hexagonal architecture required a significant learning curve for the team, particularly in understanding how to decouple the domain layer from the infrastructure layer and implement ports and adapters correctly. Misunderstandings about the responsibilities of adapters and separating concerns between layers led to occasional misplacement of logic, requiring debugging and refactoring throughout the project. Especially challenging was accepting a certain degree of code duplication—thus violating the DRY principle—to maintain minimal coupling between layers, e.g. providing entities in all layers and components.

3 API Technology

We chose GraphQL as our API technology due to its ability to handle the high level of coupling within our entities efficiently.

3.1 Advantages

Traditional REST APIs can lead to underfetching, requiring multiple requests to retrieve related data, such as fetching a ticket along with its pilot and associated client. GraphQL resolves this by allowing the client to request exactly the data it needs in a single query. This ability to minimize both underfetching and overfetching made GraphQL a clear choice for managing the complex relationships in our domain. Another benefit is the schema-driven development approach which guarantees consistency and clarity in API interactions, making the system easier to maintain and enhancing developer onboarding.

3.2 Challenges and limitations

However, we also encountered limitations and trade-offs. Debugging the system was particularly challenging due to intricacies in the implementation. For instance, the getters and setters for properties had to follow strict naming conventions to function correctly with GraphQL. This was especially unusual for boolean properties, where a typical convention like isOperable needed to be implemented as getIsOperable. Deviating from these conventions caused subtle data-mapping issues that were difficult to identify and resolve.
Additionaly, we added custom scalar types for *void* and *Long* to meet our specific requirements. While GraphQL's extensibility made these customizations possible, the absence of standard support for such common data types added to the implementation complexity.
Finally, custom error handling, especially for authentication and detailed error reporting, is less straightforward with GraphQL, as we discovered during our project since the error message is always reduced to *"Internal Server error occured"*.
Despite these challenges, GraphQL's strengths in handling complex data relationships made it the right choice for our system.

4 Implementation Details

For authentication, a typical HTTP request is sent to the THWS authentication server, which validates the username and password and returns a token if the credentials are correct. This token is then intended to be sent back to the client for subsequent requests. However, due to the lack of custom GraphQL error handling and the inherently stateless nature of GraphQL, the token could not be stored on the client side for future requests.

As a result, to simplify the implementation, the token is stored on the server, limiting the system to a single client being able to communicate at a time. This approach minimizes complexity but sacrifices the flexibility of client-side storage. Every time a request is made, the use cases check whether the stored token is still

valid, ensuring the user's authentication status is verified before any business logic is processed.

Regarding the implementation of the adapters, it is important to highlight that the API models are deliberately simplified compared to the domain models. For instance, when data is transferred from the domain layer to the API, only the relevant information needed by the client is extracted and presented. This ensures that the internal structure of the domain, which may include complex relationships or additional details unnecessary for the client, remains hidden from external consumers. Additionally, the relationships between entities are no longer modeled as fully linked objects in the API; instead, they are represented by IDs. This approach, which can be seen as a form of denormalization, reduces complexity and improves performance by simplifying how related data is represented and transferred. Similarly, when data is received from the API, it is transformed into the domain's expected format. This process may involve tasks such as parsing date formats and assigning default values to attributes that are not provided by the API, which can be seen in the *APIAirplane* and *APIFlight* class for instance.

Eventually, the Spring Framework improves the usability of custom GraphQL scalars, such as *void* and *Long*, by making them globally available across the entire application. Through Spring's configuration management, these custom types are registered as beans, ensuring they can be easily injected and used throughout the application. This approach promotes a more organized and maintainable structure by centralizing the scalar type definitions, making them simpler to manage and update as the application evolves.

5 Testing Strategy

The testing strategy for this project is centered around ensuring the reliability, correctness, and robustness of the system. The approach incorporates unit tests and integration tests to validate both individual components and their interactions within the system.

5.1 Unit Tests

Unit tests included CRUD tests on different levels of the application. For example, persistence tests validated the correct insertion and deletion of data in the database, ensuring that entities like clients were correctly mapped to the underlying storage. Additionally, business logic tests were created to verify core functionalities, such as the correct handling of ticket seating class upgrades depending on seat availability. Futhermore, edge cases were covered, such as handling invalid inputs or null values, to test the system's robustness as it can be seen in *PaymentTest* with valid and invalid cardnumbers provided by the Stripe-API.

Regarding the implementation, common initialization logic was placed in the *@BeforeEach* method to ensure consistency and avoid redundant code. For example, unique identifiers were generated for test isolation. Clean-up operations were included in *@AfterEach* to delete any data created during the test execution, maintaining a pristine database state.

5.2 Integration Tests

We implemented API integration tests to validate the interaction between different system components and ensure that the API functions as expected. Those include simulated real-world scenarios as booking a flight. It then reads the object from the database to verify that the data was correctly persisted. Different fields have been validated such as personal details of the client as well as ticket information including the seating class. The automatic setup and shutdown of the GraphQL API server is managed by Docker, but it waits for the server to start up before running the integration tests.

5.3 Reflection on Effectiveness of Testing Strategy

The testing strategy proved to be effective in ensuring the reliability of both the API and core business logic. We found out that implementing unit tests early in the development process, ideally before or immediately after each functionality, would have been very helpful as it allows to catch issues early, especially with data mappings and core functions like seat assignments. Waiting too long to test can lead to difficulties in debugging, particularly when it comes to integration tests, where issues with business logic may only become apparent much later.

API integration tests provided confidence in the system's ability to handle real-world scenarios, ensuring smooth communication between the graphql client and our backend system.

Due to the restriction of only one user being allowed to access the database at a time, no concurrency issues had to be tested.

Overall, the strategy ensured a solid foundation for the application, but continuous refinement and additional test coverage may be necessary as the system evolves.

6 Learning Outcomes and Reflection

In addition to the mentioned outcomes of missing custom GraphQL error handling, early testing in the development process and advantages of the template pattern, we also learned that thorough planning at the beginning is crucial to the project's success. In particular, when dealing with multiple layers, any changes to models or structures later in the project can be time-consuming and complex. Therefore, having a clear roadmap in place from the start can help avoid such challenges.

Despite the large scale of the project and the constant challenges that arose during development, we managed to implement all requirements without many restrictions. Although the difficulties were not reduced, we were able to find solutions through continuous adjustments and teamwork. A useful tool was our project board in GitHub, which helped us maintain an overview and visualize progress.

Additionally, our team recognized that strong communication and collaboration were essential for maintaining a smooth workflow. However, the learning curve associated with the hexagonal architecture meant that we needed extra time in the early stages to fully understand and implement it effectively.

One challenge we faced was the need to deepen our knowledge of Hibernate. While we had some basic understanding, it became clear that we needed to en-

hance our skills especially regarding joining entities, which led to delays towards the end of the project. However, this gap in knowledge provided a valuable learning opportunity for the team.

An important learning experience came from implementing Stripe for payment validation. Since the system did not have a frontend, we had to approach Stripe integration differently from typical implementations. This required us to focus on handling backend logic for secure and efficient payment processing, which provided valuable insights into how to adapt standard payment systems to non-traditional setups.

7 Appendices

This article was drafted and refined using GPT-4 based on an outline containing related information. The GPT-4 output was reviewed, revised, and enhanced with additional content. It was then edited for improved readability and active tense, partially using Grammarly.

8 References

[1] Musch, Olaf. *Design Patterns with Java: An Introduction*. Springer Vieweg Wiesbaden, 1. Auflage, pp. 33-39. 2023. DOI: 10.1007/978-3-658-39829-3. ISBN: 978-3-658-39828-6.

[2] Sarcar, Vaskaran. Java Design Patterns: A Tour of 23 Gang of Four Design Patterns in Java. pp. 17-21. Apress, 2016. DOI: 10.1007/978-1-4842-1802-0.