Hotel Management System - Design Document

Team Members:
Christian Model - 5123081
Hasan Chalabiiev - 5123092
Steffen Krutzsch - 5123146
Vincent Weis - 6823003

February 15, 2025

Abstract

This document describes the design and implementation details of our Hotel Management System. We implemented a backend system applying the Hexagonal Architecture principles, using gRPC for our API. We elaborate on the domain, software architecture, technology choices, implementation details, testing strategy, and our reflections on key learning outcomes.

1 Introduction

The Hotel Management System enhances hospitality operations by centralizing the management of four core entities: Hotels, Guests, Bookings, and Rooms. It supports CRUD (Create, Read, Update, Delete) operations and extends functionality with domain-specific workflows:

- **Hotels**: Filter and paginate listings, rate hotels, and find available rooms by date and type.
- **Guests**: Track booking history and update contact details.
- **Bookings**: Manage check-in/check-out status and cancellations.
- **Rooms**: Adjust pricing dynamically.

2 Software Architecture

In our application, we implement the concept of the *Hexagonal Architecture* . The core idea is that the application's *Domain* remains as independent as possible from technical details or external systems. All external access goes through clearly defined *Ports*, which are then implemented by corresponding *Adapters*. This makes it easier to swap out technologies such as database access, gRPC or REST interfaces, scheduling, and so on, without modifying the core business logic.

2.1 Hexagonal Structure Overview

Each component of our architecture maintains its own variant of the models (e.g., `Hotel`, `Booking`, `Guest`, `Room`) and uses dedicated mapper classes to translate data between these variants, ensuring a clean separation of concerns and decoupling the different layers.

- **Domain (Core / Business Logic):**
  Contains the core entities (e.g., `Hotel`, `Room`, `Booking`, `Guest`) and corresponding business services (e.g., `HotelService`, `RoomService`, `BookingService`, `GuestService`). This layer is solely responsible for the business logic, including how bookings are created, canceled, or updated. The domain layer is designed independently of any technical details or external systems.

- **API Component (Part of infrastructure):**
  The API layer exposes the application's functionality via gRPC service definitions. It implements classes (e.g., `*ServiceGrpcImpl`) that convert incoming gRPC messages into calls to the domain services, using its own set of models (the DTOs) to interact with external clients.

- **Persistence Component (Part of infrastructure):**
  This layer is responsible for database operations and leverages JPA/Hibernate (with H2 for local runs). It maintains its own variant of the models as persistence entities (e.g., `HotelEntity`, `RoomEntity`, `BookingEntity`, `GuestEntity`). This ensures that changes in the database layer do not directly affect the business logic.

2.2 Ports and Adapters

Ports:

- *Inbound Ports* are the domain's service interfaces or method signatures. For instance, the interface `GuestServicePort` is called from our gRPC stubs, which defines methods (createGuest, updateEMail, etc.) our domain service `GuestService` provides.

- *Outbound Ports* are the repository interfaces, e.g., `GuestRepository`, which the domain services use to manipulate data.

Adapters:

- *Primary Adapters* are the gRPC endpoints (like `GuestServiceGrpcImpl`), which convert gRPC messages into domain calls.

- *Secondary Adapters* are the database adapters (like `GuestDatabaseAdapter`), which implement `GuestRepository` and internally use `DAO` classes to perform CRUD.

Key Challenge: The hardest part was consistently enforcing the Hexagonal boundary — i.e., ensuring that the domain classes do not directly depend on infrastructure classes. We used interfaces for communication between the domain and the persistence component to maintain separation.

3 API Technology Choice

We chose **gRPC** for our implementation:

- **Advantages**:
  - gRPC provides a strongly typed interface using `.proto` files, which ensures consistency between clients and servers.
  - gRPC simplifies development by auto-generating client and server code in multiple programming languages, reducing boilerplate and potential errors.
  - gRPC uses an efficient binary protocol over HTTP/2, which ensures lower latency and higher throughput compared to traditional text-based protocols like REST.
  - The `.proto` files provide a well-defined schema for requests and responses, making the APIs easier to understand and maintain.
- **Trade-Offs**:
  - While gRPC excels in server-to-server communication, it is not as easily consumable by web clients due to limited browser support for native gRPC calls (e.g., gRPC-Web or REST translations may be required).
  - Modifying `.proto` files requires regeneration of stubs, which may complicate rapid iteration cycles.

For our use case in hotel management, gRPC was particularly attractive because of its strongly typed contracts and high performance, ensuring reliable communication between microservices. While we did not utilize advanced features like streaming, the benefits of efficient unary communication outweighed the trade-offs in terms of development complexity.

4 Implementation Details

This section provides technical insights into the choices made for the implementation of our Hotel Management System. Our decisions were driven by clarity, maintainability, and adherence to the principles of hexagonal architecture.

4.1 Authentication and Authorization

We *did not fully integrate* advanced authentication flows due to the project scope. In a real-world scenario, we would place JWT checks or an OAuth system in the gRPC interceptors.
We prepared a class `THWSAuthorizationUtils.java` (commented out in security package) to illustrate how we might validate tokens against an external service.

4.2 Adapter Mapping

To ensure a clean separation between the different layers of our application, we implemented dedicated mapper classes. These mappers facilitate the conversion between the various representations of our models: the domain objects, API Data Transfer Objects (DTOs), and persistence entities. This design ensures that each layer remains decoupled and can evolve independently without propagating changes to the core business logic.

4.3 Framework Choice

For our implementation, we selected **Quarkus** as our primary framework. The decision was based on several key advantages:

- **Developer Productivity:** Quarkus offers rapid startup times and live coding features, which significantly accelerate the development process.
- **Seamless Integration:** It provides native support for JPA/Hibernate and gRPC, allowing us to efficiently manage both the persistence and API layers while staying true to the hexagonal architecture.
- **Easy Component Integration** Quarkus makes it straightforward to integrate new components — such as the scheduled MailerJob we integrated to automate email sending — due to its modular design and flexible extension mechanisms.

The combination of these features made Quarkus the optimal choice for our project, as it supports both rapid development and scalable, maintainable production deployments.

Overall, our implementation choices reflect a strong commitment to modularity and extensibility, ensuring that each component of the system is independently testable and can be maintained with minimal coupling.

5 Testing Strategy

Our testing strategy was designed to ensure the reliability and correctness of each component of the system. We implemented both unit and integration tests as follows:

5.1 Unit Tests

We focused on testing all components in isolation. For instance, tests for domain models, services, API, and persistence (e.g., booking logic, gRPC request handling, and database operations) were implemented using frameworks like JUnit and Mockito. This allowed us to mock dependencies and verify that each component operates correctly without interference from external systems.

5.2 Integration Tests

Integration tests were used to verify that the different layers of the application (API, domain, persistence) work seamlessly together. We set up a both a automatic integrationtest (this is run via `mvn verify`) and an interactive CLI. This CLI automatically starts up when the docker container gets started because we implemented it as our main application in Quarkus. This two types of integration tests validate the proper functioning of JPA/Hibernate mappings and ensure that the gRPC endpoints correctly marshal and unmarshal data. These tests were executed in an environment that closely mirrors production, providing confidence that the system will behave as expected in real deployments.

Overall, this layered testing approach helped us identify and resolve issues early in the development process, ensuring a robust and reliable system.

As we already mentioned, all tests (UnitTests + integration tests) are executed via `mvn verify`. Please check out our README.md for further information.

6 Learning Outcomes and Reflection

This project has been an invaluable learning experience, offering insights into both technical and collaborative aspects of software development.

6.1 Key Learnings

We deepened our understanding of the hexagonal architecture, particularly the importance of decoupling the domain from technical concerns. This separation allowed us to create independent layers that could evolve without causing ripple effects throughout the system.

6.2 What Worked Well

Our team division by domain areas (e.g., Hotel, Booking, Service, Room) allowed each member to focus on a specific part of the system, ensuring dedicated attention to each implementation. This specialization proved effective in driving progress within individual components. In addition, we held regular Zoom meetings where we engaged in shared-IDE sessions. This collaborative coding environment facilitated real-time discussions and code reviews, ensuring that team members could quickly resolve issues and align on implementation details across different areas.

6.3 Areas for Improvement

While the domain-specific team division had its merits, it sometimes became challenging when dependencies on components from other areas emerged, which slowed down progress on individual parts. In future projects using hexagonal architecture, we would consider organizing the team by architectural layers (Domain, API, Persistence) instead of by service types. This approach could enable more independent development of each layer, potentially accelerating overall progress by reducing cross-dependencies.

Acknowledgments (AI Usage)

This article was drafted and refined using GPT-o3-mini and GPT-o1 based on an outline containing related information. The output was reviewed, revised, and enhanced with additional content.