Design Document for Gym Management System

Portfolio Assignment 04 - Backend Systems

Marvin Kraus, Manuel Stöth, Pascal Putz

February 15, 2025

1 Introduction

The Gym Management System is a distributed backend application designed to streamline the daily operations of a gym. The system focuses on handling core processes such as member registration, trainer management, course administration, and booking management, thereby simplifying everyday operations in a fitness center.

The primary use cases include:

- **Member Management:** Registering new gym members, updating member details, retrieving member profiles, and deleting member records.
- **Trainer Management:** Creating trainer profiles, updating their information, and filtering trainers based on name or expertise.
- **Course Management:** Handling course creation, updating course details, listing courses with pagination, and filtering courses by name or trainer.
- **Booking Management:** Facilitating course bookings by linking members to courses with an associated booking date.

The domain is centered around four key entities:

- **Member** (see *Member.java*) – stores member name and membership type.
- **Trainer** (see *Trainer.java*) – contains trainer name and expertise.
- **Course** (see *Course.java*) – includes course name, capacity, and the assigned trainer.
- **Booking** (see *Booking.java*) – links a member to a course on a specific booking date.

Additionally, the file *GymClientMethods.java* demonstrates practical use cases by showing how these entities interact via gRPC calls (e.g., creating and retrieving a member).

2 Software Architecture

Our solution is built using the **Hexagonal Architecture (Ports and Adapters)** approach, which clearly separates the core business logic from external systems. The architecture is divided into the following layers:

Domain Layer

The Domain Layer is the heart of the application and now contains both the core domain models and the business logic. It includes:

- JPA-annotated entities such as *Member.java*, *Trainer.java*, *Course.java*, and *Booking.java*.
- Business logic components (e.g., *MemberLogic*, *TrainerLogic*, *CourseLogic*, and *BookingLogic*), which have been refactored into the domain layer. These classes implement operations such as validation, entity creation, updating, deletion, and enforcing business invariants.

Application Layer

The Application Layer now contains the server-side implementation. This includes the gRPC service adapters such as:

- *MemberServiceImpl*
- *TrainerServiceImpl*
- *CourseServiceImpl*
- *BookingServiceImpl*

These classes handle incoming gRPC requests, map them to domain objects (by invoking the corresponding business logic in the Domain Layer), and convert the responses back to Protocol Buffer messages.

Infrastructure Layer

The Infrastructure Layer contains the adapters for persistence and external integrations:

- Repository interfaces and implementations (e.g., *MemberRepository* with *MemberRepositoryInterface*, *TrainerRepository*, *CourseRepository*, and *BookingRepository*) that handle CRUD operations using Hibernate.
- Configuration files (e.g., the Hibernate configuration and *HibernateUtil.java*) that set up the H2 database.
- **Containerization:** Instead of a separate Docker configuration, we utilize the Jib Maven Plugin to build and containerize the application.

Ports and Adapters

The business logic interfaces (ports) decouple the core functionality from the external adapters. The API adapters in the Application Layer and the repository adapters in the Infrastructure Layer serve as bridges that translate between different representations, ensuring that the domain remains isolated from external concerns.

Challenges in Implementation

A key challenge was mapping between gRPC messages and domain models while preserving a clear separation between external representations and internal logic. Implementing pagination and filtering in the repository layer also required careful design. Additionally, integrating the Jib Maven Plugin for containerization added complexity to the build process.

3 API Technology

We chose **gRPC** for our API for several reasons:

- **Performance:** gRPC uses HTTP/2, enabling low-latency communication with features such as multiplexing and streaming.
- **Contract-First Approach:** Protocol Buffers enforce a strict, versioned API contract, ensuring robust interfaces.
- **Strong Typing:** The auto-generated code from .proto files minimizes runtime errors and ensures compile-time safety.
- **Scalability:** gRPC supports bi-directional streaming, paving the way for real-time updates and future enhancements.

While gRPC has a steeper learning curve and its binary format is less intuitive for debugging compared to JSON, its advantages in performance and API contract quality make it the optimal choice for our distributed system.

4 Implementation Details

Mapping and Object Creation

All domain entities are constructed using the Builder Pattern, which simplifies the creation of immutable objects. For example, the *Member* class provides a static Builder for constructing member objects. Similar patterns are used in *Trainer*, *Course*, and *Booking*. In the API adapters, incoming gRPC requests are mapped to these domain objects, and responses are mapped back to Protocol Buffer messages.

Persistence and Infrastructure

- **ORM and Database:** Hibernate is used as the JPA provider to map our entities (and business logic) to an H2 database. The Hibernate configuration file registers all entities and manages session handling via *HibernateUtil.java*.
- **Repositories:** CRUD operations are abstracted into repository interfaces (e.g., *MemberRepositoryInterface*) and implemented in classes such as *MemberRepository*. This design ensures that changes in the persistence layer do not affect the domain logic.
- **Build and Deployment:** Maven is used as the primary build tool. It integrates plugins for Java compilation, Protocol Buffer processing, and containerization using the Jib Maven Plugin.

Frameworks and External Libraries

Our solution leverages:

- **gRPC** for API communication.
- **Hibernate** (via Jakarta Persistence) for ORM.
- **H2 Database** for development and testing.
- **Maven** for build and dependency management.
- **JUnit** for unit testing.
- **Jib Maven Plugin** for containerization (building Docker images).

Authentication and Security

Although a comprehensive authentication mechanism is not implemented in the current version, the system is designed to allow for the easy integration of an external authentication adapter (e.g., using Spring Security) in the future without impacting the core business logic.

5 Testing Strategy

Unit Testing

Unit tests are implemented for the domain logic classes (e.g., *MemberLogicTest*, *TrainerLogicTest*) using JUnit 5. Fake repository implementations (such as *FakeMemberRepository* and *FakeTrainerRepository*) simulate persistence operations, enabling isolated testing of:

- Entity creation, updating, and deletion.
- Pagination and filtering logic.
- Proper exception handling when entities are not found.

Integration Testing

Integration tests verify the end-to-end functionality of the system:

- gRPC endpoints are tested by establishing a channel to the running application and invoking service calls (as shown in *GymClientMethods.java*).
- Complete workflows (e.g., creating and retrieving a member) are validated, and tests are executed as part of the Maven build process (via *mvn verify*).
- The use of the Jib Maven Plugin ensures that containerization is consistent between development and production environments.

Reflection on Testing

The layered testing approach allowed us to identify issues early in development. Unit tests ensured the correctness of individual components, while integration tests confirmed that all system layers interact as intended. Future enhancements may include performance tests and expanded integration scenarios.

6 Learning Outcomes and Reflection

Key Learning Outcomes

- **Hexagonal Architecture:** Implementing a clear separation of concerns enhanced the maintainability and scalability of the system.
- **gRPC API Development:** We learned the benefits of a contract-first approach using Protocol Buffers, which provides strong typing and high performance.
- **ORM and Persistence:** Working with Hibernate and an H2 database improved our understanding of object-relational mapping and efficient transaction management.
- **Team Collaboration:** The modular design allowed parallel development and smoother integration, underlining the importance of clear interfaces and version control.

Reflections on Project Execution

- **Successes:** The strict separation between layers and the modular architecture contributed to a highly maintainable and extensible system. Automation via Maven and the Jib Maven Plugin streamlined build and deployment processes.
- **Areas for Improvement:** Future projects could benefit from a more robust authentication mechanism, improved logging, and the inclusion of UML diagrams to further clarify system design.

Acknowledgment

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.

References

[1] A. Cockburn, *Hexagonal Architecture*, [Online]. Available: `https://alistair.cockburn.us/hexagonal-architecture/` (2005).