Software Design Document for Smart Home Management System with Energy Reports

Buse Okcu, Irmak Damla Özdemir
Supervised by: Prof. Dr. Peter Braun

February 15, 2025

1 Introduction

1.1 Purpose

The purpose of this Software Design Document is to detail the design and architecture of the Smart Home Management System with Energy Reports. This backend system is implemented using the Hexagonal Architecture, as taught in the Backend Systems course, ensuring a clear separation between domain logic and infrastructure components. The backend is developed with GraphQL, offering efficient and flexible data retrieval while supporting energy optimization features. This document outlines the system's functionalities, architectural decisions, and integration strategies.

1.2 Scope

The Smart Home Management System with Energy Reports is a backend application enabling users to manage and automate smart home devices via an API. The system aims to optimize energy consumption through energy reports.

1.3 Overview

The domain layer is organized into two primary sub-packages:

1.3.1 Models

Package *models* contains the core models representing the main entities of the system. The system includes the following models:

- Device: Represents devices in the system and it has status(on/off), consumption values (per hour or daily), its own energy reports, automation rules. It also keeps the last-turn-on-time of the device.
- User: User has its own characteristic attributes like e-mail and password. User can add devices to the rooms, automation rules to the devices and gets energy reports in a certain period of time.
- AutomationRule: Automation rules are defined through enum (TIME and CONSUMPTION), automate device actions in certain situations. Energy consumption of device can be limited through automation rules.
- EnergyReport:Energy reports are generated based on the usage duration of the devices.
- Room

1.3.2 Ports

Ports define the interfaces for services, enabling interaction between the domain and external layers. The system includes the following ports:

- AutomationRuleService
- DeviceService
- EnergyReportService
- RoomService
- UserService

This structure ensures a clear separation of concerns, adhering to the principle of Hexagonal Architecture, with focus on scalability and maintainability.

2 Software Architecture

2.1 Hexagonal Architecture

The Smart Home Management System follows the hexagonal architecture (also known as Ports and Adapters), ensuring modularity, maintainability, and testability. Input ports receive and process external requests, ensuring they align with the core logic, while output ports handle outgoing communication, transmitting responses to databases or external systems. This approach decouples the core business logic from the infrastructure, allowing seamless integration with various APIs, databases, and external services.[1].

At the core, the domain layer encapsulates all business rules related to home automation, device control, and energy reporting. The application layer orchestrates interactions between domain entities and external interfaces via well-defined ports. The infrastructure layer consists of adapters that facilitate communication with

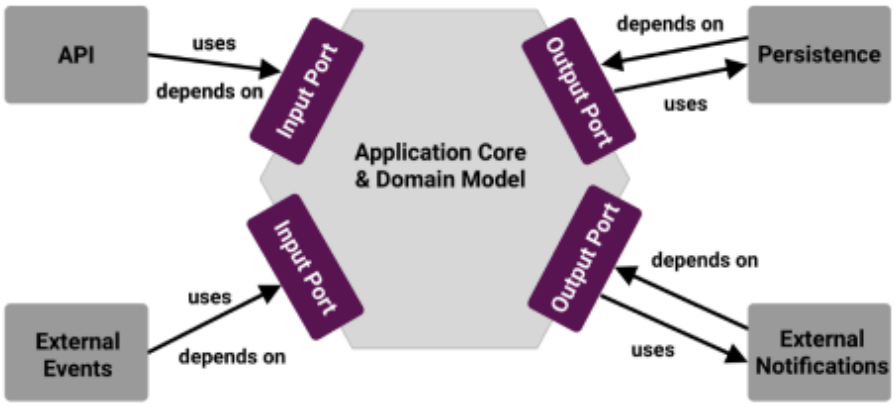

Figure 1: Hexagonal Architecture Diagram

databases, APIs, and external services, making the system flexible for future integrations.
The core domain logic is surrounded by ports (interfaces) that act as entry points and adapters that bridge communication between the business logic and infrastructure components (e.g., databases, APIs, external services).

- **Domain Layer:** Contains core business entities such as *Device, Room and AutomationRule*, each representing real-world smart home concepts. Business logic is encapsulated in these domain models, ensuring separation from infrastructure concerns.
- **Business Logic:** Business logic is placed inside the Application core, which is at the center of the hexagonal architecture. It is implemented in the core application services, which interact with the domain. Application core consists of two key components: Domain Model and Application services.
- **Infrastructure Layer:** Implements persistence and external communication. It allows storage, retrieval, and control of smart devices while abstracting dependencies such as the database. It consists of:
  - **API DTOs:** It is used to receive and send data via API endpoints. This avoids exposing domain models directly. DTO helps in validating API requests before passing them to the business logic.
  - **Adapters:** Implement business logic, call repositories for database operations, use mappers to convert between DTOs, domain models and entities.
  - **Entities:** JPA entity classes that define how objects are stored in the database. Each entity maps to a database table.
  - **Mappers:** Used to maintain a clean separation between domain models and entities.
  - **Repositories:** Spring Data JPA interfaces that interact with the database. Spring Data JPA automatically generates database queries, simplifying persistence operations.

2.2 Ports and Adapters Implementation

Ports and Adapters pattern is followed to abstract external dependencies.

- Ports(Interfaces): Defined in the domain layer for business logic operations.
- Adapters(Implementation): Implemented in the infrastructure layer to interact with the database.

Here is an example flow:

1. API Request → *DeviceController*
2. DTO Processing → *DeviceInput*
3. Business Logic Execution→ *DeviceServiceImpl*
4. Database Operation → *DeviceRepository*
5. Return Response → *DeviceMapper*

2.3 Challenges

Decoupling infrastructure dependencies from business logic required careful design of port interfaces and adapter implementations.

3 Explanation of API Technology

The Smart Home Management System with Energy Reports employs GraphQL as the primary API technology, implemented with Spring Boot. This choice was made to ensure efficient data retrieval, flexibility in querying, and optimized network performance. The system benefits from GraphQL's ability to handle complex relationships between entities, making it well-suited for managing smart home devices and energy reports.

- **Flexible Data Retrieval:** GraphQL allows clients to specify exactly which fields they require, eliminating over-fetching and under-fetching of data.
- **Optimized for Nested Data:** The system involves multiple interrelated entities *(User, Device, EnergyReport, AutomationRule)*, making GraphQL's ability to fetch related data in a single request a key advantage.
- **Efficient Query Execution:** GraphQL enables clients to send a single request to retrieve multiple entities, reducing the need for multiple API calls, which is particularly beneficial for real-time energy reporting and device automation.

3.1 Trade-offs

- A GraphQL query that requests detailed energy reports for multiple users simultaneously can lead to high computational overhead.
- GraphQL queries are designed for retrieving structured data, not ideal for real-time event-based triggers.
- GraphQL provides flexibility but can lead to high-cost queries that significantly impact API performance.

By leveraging GraphQL with Spring Boot, the Smart Home Management System achieves optimized data retrieval, reduced network overhead, and greater flexibility in API interactions. The ability to query multiple related entities in a single request, combined with schema-driven adaptability, ensures that the API remains scalable, maintainable, and efficient for smart home automation and energy reporting.

4 Implementation Details

4.1 Implementation Choices

The system leverages adapter mapping to ensure clear separation between the application, domain, and infrastructure layers. REST Controllers in the application layer process API requests and utilize service adapters to handle business logic. These adapters (*ServiceImpl* classes) bridge the gap between incoming DTOs (Data Transfer Objects) and the domain model.

MySQL is used as the primary database, running in a Docker container for portability and scalability. Database interactions are managed through Spring Data JPA with Hibernate as the ORM framework, ensuring efficient data persistence. MySQL Connector (JDBC) facilitates communication between the application and the database, with configurations defined in *application.properties*.

4.2 Framework Choice

Spring Boot's dependency injection (DI) and service layer structure naturally align with Hexagonal Architecture, ensuring a clear separation between domain, application, and infrastructure layers. The Ports and Adapters pattern maintains loose coupling, while Spring Data JPA simplifies CRUD operations for entities like *Device, AutomationRule, and EnergyReport*, offering transaction management and database abstraction. This approach enhances modularity and maintainability by leveraging Spring Boot's native support for service-oriented architecture.

5 Testing Strategy

Testing part consists of unit tests, integration tests, and persistence tests:

- **Unit Tests:** Use JUnit to verify individual component logic. They focus on domain logic validation: execution logic, turn on/off behavior, correct calculation of consumption, etc.

- **Integration Tests:** Use Spring Boot Test and MockMvc to perform API calls. They validate API functionality and system interactions. For example *BaseIntegrationTest* provides a MySQL TestContainer for database simulation. Controller tests, on the other hand, test REST API endpoints for CRUD operations.
- **Persistence Tests:** Validate repository operations for all entities and ensures database consistency and DTO-entity conversions.

This strategy ensures reliability by covering business logic, API endpoints, and database interactions. The Testcontainers approach enhances accuracy by testing against an actual MySQL instance.

6 Learning Outcomes and Reflection

Throughout this project, we implemented a hexagonal architecture to design a scalable and maintainable backend system. The adoption of this architecture provided valuable insights into structuring software applications in a way that minimizes dependencies between core business logic and external infrastructure components.[2]

To maintain flexibility and prevent direct coupling, we introduced adapters that translate API requests into device-specific instructions. This approach aligns with the Adapter Design Pattern, as described in Design Patterns: Elements of Reusable Object-Oriented Software, which defines an adapter as a mechanism that "converts the interface of a class into another interface clients expect."[2]

Additionally, understanding how to design adapters efficiently was a crucial learning point. For example, in handling communication with external devices, we explored various mapping strategies, including object transformation techniques, to ensure a seamless translation between our internal API.

On the other hand, as Framework, while Spring Boot is widely used for its powerful ecosystem and ease of setup, its complexity and hidden abstractions introduce challenges that make it prone to unexpected failures and difficult debugging. Given these concerns, in future projects, we would consider using another framework, for example Quarkus with JAX-RS, instead of Spring Boot.

We frequently used pair programming technique, which improved efficiency and helped resolve errors more effectively. Being a two-person team allowed us to gain deeper insights into backend APIs and Hexagonal Architecture through hands-on implementation.

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.

References

[1] Prof. Dr. Peter Braun. Backend systems - software architectures for backend systems(transcript). Lecture slides, THWS, 2025.

[2] Alistair Cockburn. Hexagonal architecture, 2005.