Documentation for milu-backend

A Distributed Todo Application Implemented with Hexagonal Architecture

Michael Winsauer 5123128 Luis Weich 5123052

February 15, 2025

Abstract

This document describes the design, architecture, and implementation details of **milu-backend**, a distributed todo application built using a Hexagonal Architecture and REST API technology. We explain our design decisions, the integration of our business logic with external adapters, our API choices, and our testing strategy. Finally, we reflect on the lessons learned and propose improvements for future iterations.

1 Introduction

The **milu-backend** project is a distributed system designed for managing todo items in a collaborative environment. Our solution leverages Hexagonal Architecture to achieve a clear separation of concerns. This design ensures that the core business logic remains independent of external systems—such as persistence, authentication, or web APIs—thus making our application modular, testable, and scalable.

The domain of our application centers on three primary entities:

- **Accounts:** Representing users of the system, each account holds user-specific data and is associated with one or more boards.
- **Boards:** Serving as organizational units, boards group related todo items, allowing users to structure tasks by projects or categories.
- **Todos:** These are the individual tasks or action items that users create, update, delete, or track. Each todo has attributes such as a name, description, and state.

Key use cases include:

- **User Management:** Users can create, update, and delete their accounts. This allows personalized tracking and management of tasks.
- **Task Organization:** By creating boards, users can effectively group and organize their todos according to different projects or themes.
- **Task Management:** Within each board, users can perform full CRUD (Create, Read, Update, Delete) operations on todos. In addition, our API supports filtering by attributes (such as name and state) and pagination, which improves performance when handling large volumes of data.
- **Collaborative Work:** Multiple boards and todos can be linked to a single account, supporting collaboration and shared task tracking within teams.

In summary, **milu-backend** provides a robust and flexible platform for task management. By employing Hexagonal Architecture, the solution cleanly separates business logic from technical concerns, facilitating easier maintenance, testing, and future enhancements.

2 Software Architecture

Our solution is built on the principles of Hexagonal Architecture—also known as Ports and Adapters—which emphasizes a clean separation of concerns. This design enables us to keep our core business logic isolated from external systems, such as databases, web services, or user interfaces. The architecture is organized as follows:

- **Domain Layer:** This layer is the heart of our application. It contains the core business logic and the domain models (i.e., `Account`, `Board`, and `Todo`). The domain layer is deliberately kept free of any external dependencies, ensuring that the core functionality remains pure and easy to test. Changes in external technologies, such as different databases or communication protocols, do not affect this layer.
- **Application/Service Layer:** Serving as the intermediary, the application layer orchestrates the various use cases. It interacts with the domain layer through well-defined interfaces, known as ports, and coordinates operations by invoking the appropriate business logic. For example, when a client sends a request to create a new todo, the service layer validates the input, interacts with the domain models, and delegates data persistence to the outbound adapters.
- **Ports:** Ports are interfaces defined in the domain layer that specify the operations available both inbound (e.g., creating, retrieving, updating, or deleting an account) and outbound (e.g., saving an entity to a database). By defining these interfaces, we abstract the core business logic from the specific technologies used for communication or data storage. This ensures that the domain layer remains independent and that we can easily swap out implementations without affecting core functionalities.
- **Adapters:** Adapters are the concrete implementations of the ports and serve as bridges between the core domain and external systems:
  - **Inbound Adapters:** These include our REST resources (e.g., `AccountResource`, `BoardResource`, `TodoResource`). They handle HTTP requests, translate incoming data (typically JSON) into domain objects, and delegate actions to the service layer. They also transform domain responses back into HTTP responses, often incorporating hypermedia links for enhanced API discoverability.
  - **Outbound Adapters:** Our outbound adapters consist of the JPA repositories that handle data persistence. These adapters translate domain models into JPA entities and vice versa. Since JPA requires concrete classes, we implement mapping layers that convert between our domain representations (or JSON DTOs) and the persistence-specific implementations (e.g., `JpaAccount`, `JpaBoard`, `JpaTodo`). This mapping ensures that the domain layer remains agnostic of the underlying database technology.

The most challenging aspect of our implementation was enforcing a strict separation between our core business logic and the persistence details. To address this, we developed mapping layers that handle conversion between domain models, JSON representations, and JPA entities. This allows the core domain to remain unpolluted by persistence-specific code while still enabling the necessary data storage and retrieval operations through our outbound adapters.

Overall, this architectural approach results in a highly modular and maintainable system. It facilitates easier unit and integration testing by decoupling components and makes future technology changes smoother, as modifications to external systems do not ripple into the core business logic.

3 API Technology

We chose a RESTful API for our implementation. REST was selected because:

- It naturally supports CRUD operations.

- It is widely understood and allows for easy integration with front-end clients.
- We could implement hypermedia (HATEOAS) principles, enabling clients to discover available actions dynamically.

Our API leverages Dropwizard with Jersey to handle HTTP requests, providing a robust and production-ready framework for building RESTful services. Dropwizard offers rapid development capabilities along with built-in metrics, health checks, and configuration management, while Jersey simplifies the creation of resource classes and the mapping between HTTP requests and Java methods. This combination allows us to easily expose our core functionality as RESTful endpoints.

To enhance usability and performance, our API supports filtering, paging, and caching. For example, our list endpoints accept query parameters such as `name`, `limit`, and `offset` to allow clients to retrieve a subset of results. Filtering helps in narrowing down the dataset based on criteria (like searching for accounts or todos by name), while paging prevents the API from overwhelming the client with too much data at once by limiting the number of results returned in a single call. Additionally, we use HTTP Cache-Control headers to instruct clients and intermediate proxies to cache responses for a defined period, thereby reducing server load and improving response times.

Moreover, we have embraced the hypermedia principle by including contextual links within our responses. Each resource returned by our API is wrapped in a hypermedia container that provides links—such as `self`, `update`, and `delete`—which clients can use to navigate the API dynamically. These links are generated using the request context, ensuring that they are absolute and correct, and they help to make the API more discoverable and self-descriptive. This design approach not only adheres to REST best practices but also significantly improves the developer experience when interacting with our service.

4 Implementation Details

4.1 Adapters and Mapping

In our milu-backend project, we paid special attention to cleanly separating the core business logic from the external layers. To achieve this, we use dedicated adapters and mapping layers. In the REST layer, we utilize JSON DTOs—such as `JsonAccount`, `JsonBoard`, and `JsonTodo`—to communicate with clients. These classes are designed to provide only the necessary information for data transfer and to shield our domain from persistence-specific details.

On the persistence side, our JPA entities (e.g., `JpaAccount`, `JpaBoard`, and `JpaTodo`) implement our domain interfaces, but they include all the annotations and configurations required by Hibernate. To bridge the gap between these layers, we implemented mapping functions that convert between JSON DTOs, domain objects, and JPA entities. This mapping not only isolates our core domain logic from the technical intricacies of data persistence and JSON serialization, but it also provides a flexible mechanism for evolving the API without impacting the business logic.

4.2 Authentication

Authentication within our application is handled using JSON Web Tokens (JWT). We have implemented a dedicated `JwtHandler` that is responsible for encoding user data into tokens and verifying them on incoming requests. This approach ensures that only authenticated users can access sensitive endpoints, thereby securing our API. By integrating JWT authentication directly into our REST resources, we achieve a stateless authentication mechanism that is both scalable and easy to maintain.

4.3 Frameworks

We have chosen a set of mature, well-integrated frameworks to support our implementation:

- **Dropwizard:** This framework was selected for its rapid development capabilities and its production-ready features such as health checks, metrics, and configuration management. It provides a robust foundation for building our RESTful API.

- **Jersey:** As the backbone of our REST implementation, Jersey simplifies the creation of resource classes, HTTP request mapping, and JSON serialization/deserialization.
- **Guice:** Dependency injection is managed by Guice, which ensures that our components remain loosely coupled and easily testable. Guice's integration with Dropwizard allows us to cleanly manage our application's lifecycle and configuration.

Together, these frameworks allow us to build a modular, maintainable, and scalable system. They enable us to implement advanced features such as filtering, paging, caching, and hypermedia controls while keeping our core business logic pure and independent from technical concerns.

5 Testing Strategy

Our testing strategy is a cornerstone of our development process, ensuring that both individual components and the system as a whole function as expected. We employ a dual approach that encompasses unit tests.

5.1 Unit Testing

We place a strong emphasis on unit testing to validate our business logic in isolation. Using **JUnit** as our primary testing framework, we write tests for each method and class in our domain and service layers.

Our unit tests cover a wide range of scenarios, including:

- **Correct Paths:** Ensuring that methods return expected results when provided with valid input.
- **Edge Cases:** Testing boundaries and unusual inputs to confirm that our logic handles unexpected scenarios gracefully.
- **Error Handling:** Verifying that appropriate exceptions are thrown and handled correctly when errors occur.

This rigorous unit testing approach ensures that our core business logic remains robust, facilitating quick identification and resolution of issues early in the development cycle.

5.2 Integration Testing

Complementing our unit tests, our integration tests validate the interactions between different layers of the application. We simulate real-world conditions by using an in-memory **H2** database to test our JPA repositories and REST endpoints. These tests verify that our application components—such as the service layer, REST resources, and persistence layer—work together seamlessly.

5.3 Continuous Integration

Our continuous integration process is integrated with Maven, which automatically runs both unit and integration tests on each code change. This automation ensures that system stability is maintained throughout the development lifecycle, and any issues are caught before deployment.

6 Learning Outcomes

This project provided us with deep insights into designing distributed systems using Hexagonal Architecture. Through the development of **milu-backend**, we learned:

- **Decoupling Business Logic from Technical Infrastructure:** The project reinforced the importance of isolating core business rules from external dependencies such as databases and frameworks. By leveraging interfaces and concrete implementations, we created a system that is both flexible and easy to maintain.

- **Clear API Contracts with Ports and Adapters:** Defining explicit contracts between our domain layer and external systems improved both internal communication and external integration. This clarity has simplified maintenance and facilitated future scalability.
- **Hypermedia-Driven REST APIs:** Implementing hypermedia (HATEOAS) principles allowed us to enrich API responses with contextual links, enabling clients to discover available actions dynamically and interact with the API in a self-descriptive manner.

Overall, the project has significantly enriched our technical and collaborative skills, laying a strong foundation for building resilient and scalable distributed systems.