ADR-7

Layered Backend Flow with DTOs, Mappers, and Repositories

Date: 13-04-2025

Status

Accepted

Author

System Architecture Team

Context

The backend architecture for the system must support clean separation of concerns, testability, scalability, and maintainability. The team is using Spring Boot 3.x and needs a consistent convention for implementing the backend logic from API requests down to the database layer.

Decision

We adopt the following flow:

- **Frontend → Controller → Service → Repository → Entity**
- **DTOs** are used as the data exchange layer between frontend, controller, and service.
- **Mappers** are introduced for clean conversion between DTOs and Entities.
- **DAOs** are *not used*, as CrudRepository and JpaRepository in Spring Boot 3.x offer sufficient built-in data access operations.
- **Services** hold the business logic and call the repository layer to access the database.
- **Repositories** directly interface with Entity classes and handle persistence logic.

Alternative	Pros	Cons
Use DAOs explicitly	Clear separation of DB operations	Redundant in Spring Boot; adds unnecessary abstraction
Controller handles logic directly	Fewer layers to manage	Violates separation of concerns, hard to test and maintain
Skip mappers and use entities in API	Simpler setup, fewer classes	Tight coupling, risk of exposing internal data structures
Layered Flow with DTO + Mapper + Repository (Chosen)	Clean separation of concerns, secure, easily testable, aligns with Spring Boot best practices	Slightly more boilerplate due to additional DTO and mapper classes

Rationale

- Follows standard layered architecture and clean separation of concerns.
- Ensures DTOs protect internal models from exposure and allow flexibility in API evolution.
- Mappers improve maintainability and unit testing by isolating transformation logic.
- Repository interfaces in Spring Boot 3.x provide sufficient built-in CRUD functionality, making DAO layers unnecessary.
- Keeps the service layer focused on orchestrating logic and transformation.

Consequences

Positive:

- Improves maintainability, readability, and testability.
- Prevents tight coupling between persistence and external interfaces.
- Easily mockable service and repository layers for testing.

Negative:

- Slight increase in the number of classes due to DTOs and mappers.