Akademia Górniczo-Hutnicza
Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki
Kierunek Informatyka i Systemy Inteligentne

Design Patterns

ORM in Java (CoreORM)

Documentation

Mateusz Kotarba
Konrad Małek
Barłomiej Mazgaj

Kraków, 11 listopada 2025

1 Logical architecutre

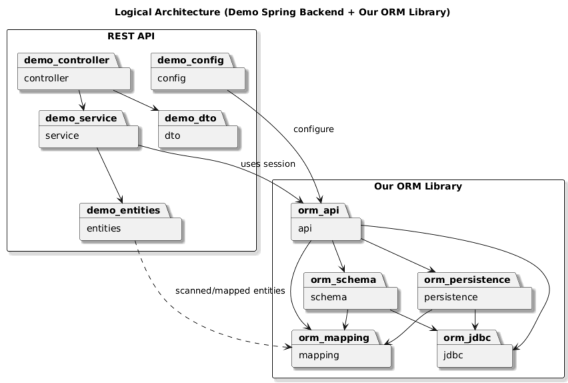


Figure presents the logical decomposition of the solution into two main layers: the **Demo Spring Backend (REST API)** and the **Our ORM Library**. The diagram focuses on package-level responsibilities and the primary dependency flow between them.

Layers

1. **Application Layer (Demo Spring Backend / REST API)**
   - **entities**: domain model classes defined by the application (e.g., `Employee`, `Account`). These classes are the source of mapping metadata used by the ORM.
   - **controller**: REST endpoints responsible for request handling, validation at the API boundary, and orchestration of use cases via the service layer.
   - **service**: business logic and use-case implementation. Services coordinate repositories and enforce application rules.
   - **dto**: request/response objects used for API communication, decoupling external contracts from internal entity representations.
   - **config**: application configuration responsible for bootstrapping the ORM (e.g., creating and wiring `Configuration` and data source settings).
2. **ORM Library Layer**
   - **api**: public entry points exposed to the application (e.g., configuration, session). This package defines the main integration surface for consumers.
   - **mapping**: discovery and metadata construction for user-defined entities (e.g., scanning annotated classes and building an internal mapping model used by the ORM runtime).
   - **persistence**: core persistence engine (e.g., change tracking, coordination, entity persisting/loading strategies, inheritance strategies) built on top of mapping metadata.
   - **jdbc**: low-level database access layer responsible for executing SQL commands, managing JDBC connections, and handling dialect-specific concerns where applicable.
   - **schema**: schema-related utilities such as generating or validating database schema based on the mapping metadata (optional at runtime depending on configuration).

Dependency Flow

- Within the application, the dependency direction follows a typical Spring layering: `controller` → `service` →.
- The `service` package depends on the ORM **api** package to open sessions and perform persistence operations.
- The application's `entities` are processed by the ORM **mapping** package to build metadata that drives persistence and schema generation.
- Inside the ORM library, **api** orchestrates **mapping**, **persistence**, **schema**, and **jdbc**; the **persistence** and **schema** packages use **jdbc** to interact with the database.

2 Physical architecutre

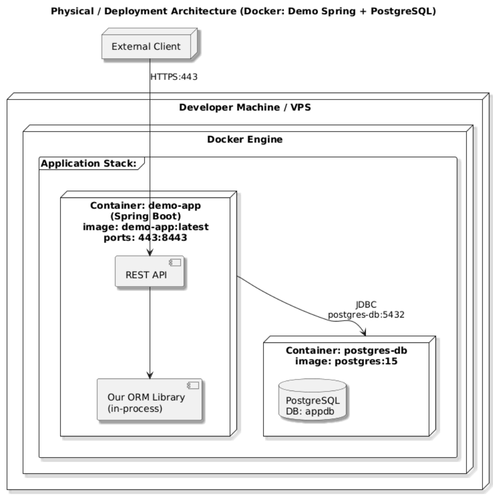


Figure presents the runtime deployment of the system on a **Developer Machine or VPS** using **Docker**. The solution is deployed as a small application stack consisting of two containers connected through Docker's internal networking.

Components

- **External Client** — any consumer of the backend (e.g., browser frontend, mobile app, Postman, or another service). The client communicates with the backend over HTTPS.
- **Container: demo-app (Spring Boot)** — the main backend application. It exposes a **REST API** and executes all business logic. The application uses **Our ORM Library** as an *in-process* dependency (a Java library loaded into the same JVM). Therefore, the ORM is not a separate service and does not require its own container.
- **Container: postgres-db (PostgreSQL)** — the relational database container responsible for persistent storage of application data.

Communication

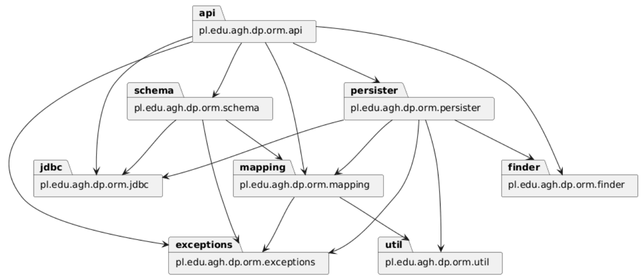
- The **External Client** connects to the backend via **HTTPS on port 443**. Inside the container, the application listens on **8443**, and Docker port mapping forwards `443` on the host to `8443` in the container.
- The **demo-app** container connects to **postgres-db** using **JDBC** over the Docker internal network, targeting `postgres-db:543` This keeps database traffic internal to the container network and avoids exposing the database publicly unless explicitly required.

Key design points

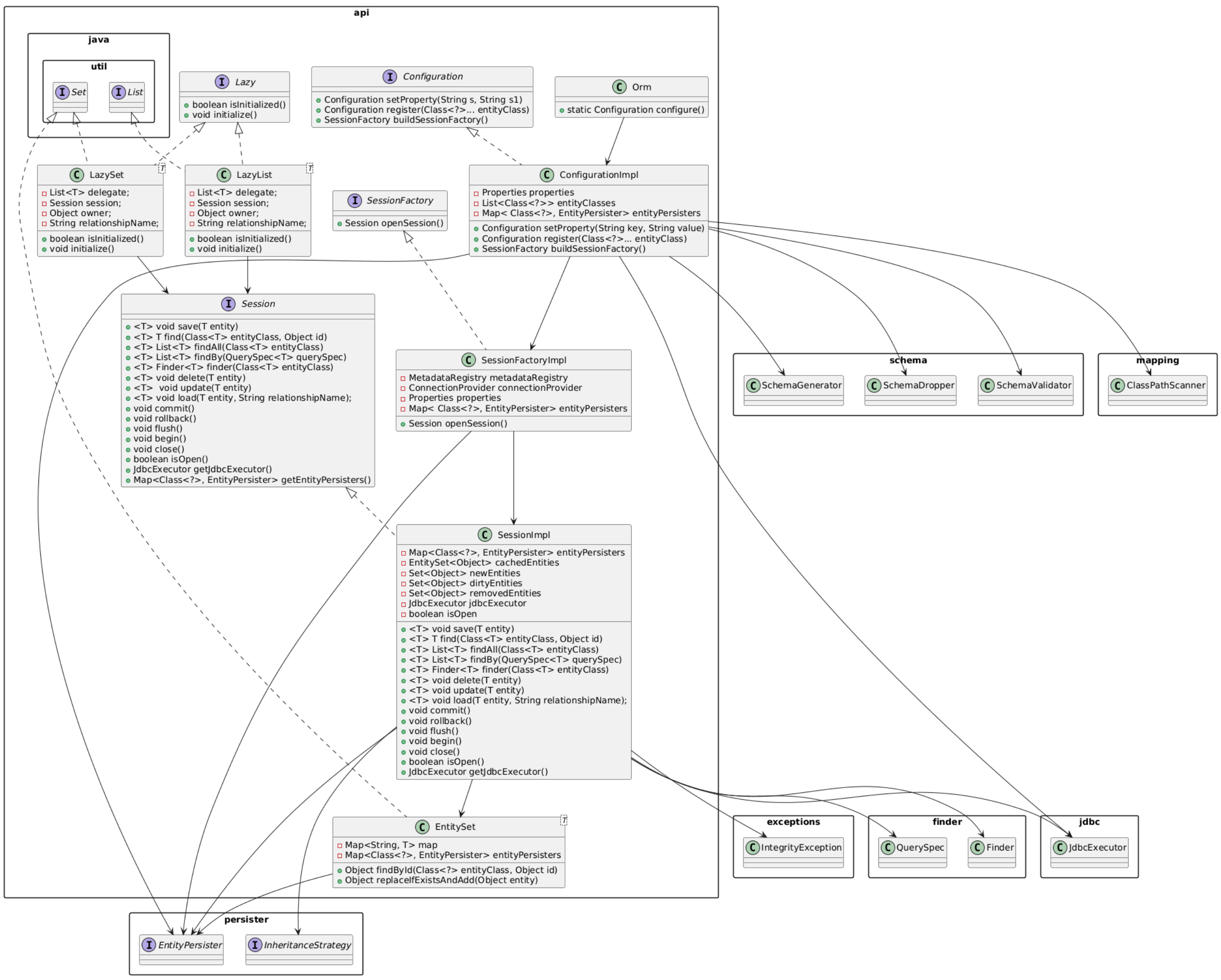
- The backend and database are separated into dedicated containers, which improves reproducibility and simplifies local/VPS deployment.
- The ORM operates *inside* the backend container (library-level integration), so the only external runtime dependency of the backend is the database connection.
- The database container is treated as an internal infrastructure component; clients never communicate with it directly.

3 Class diagram - ORM

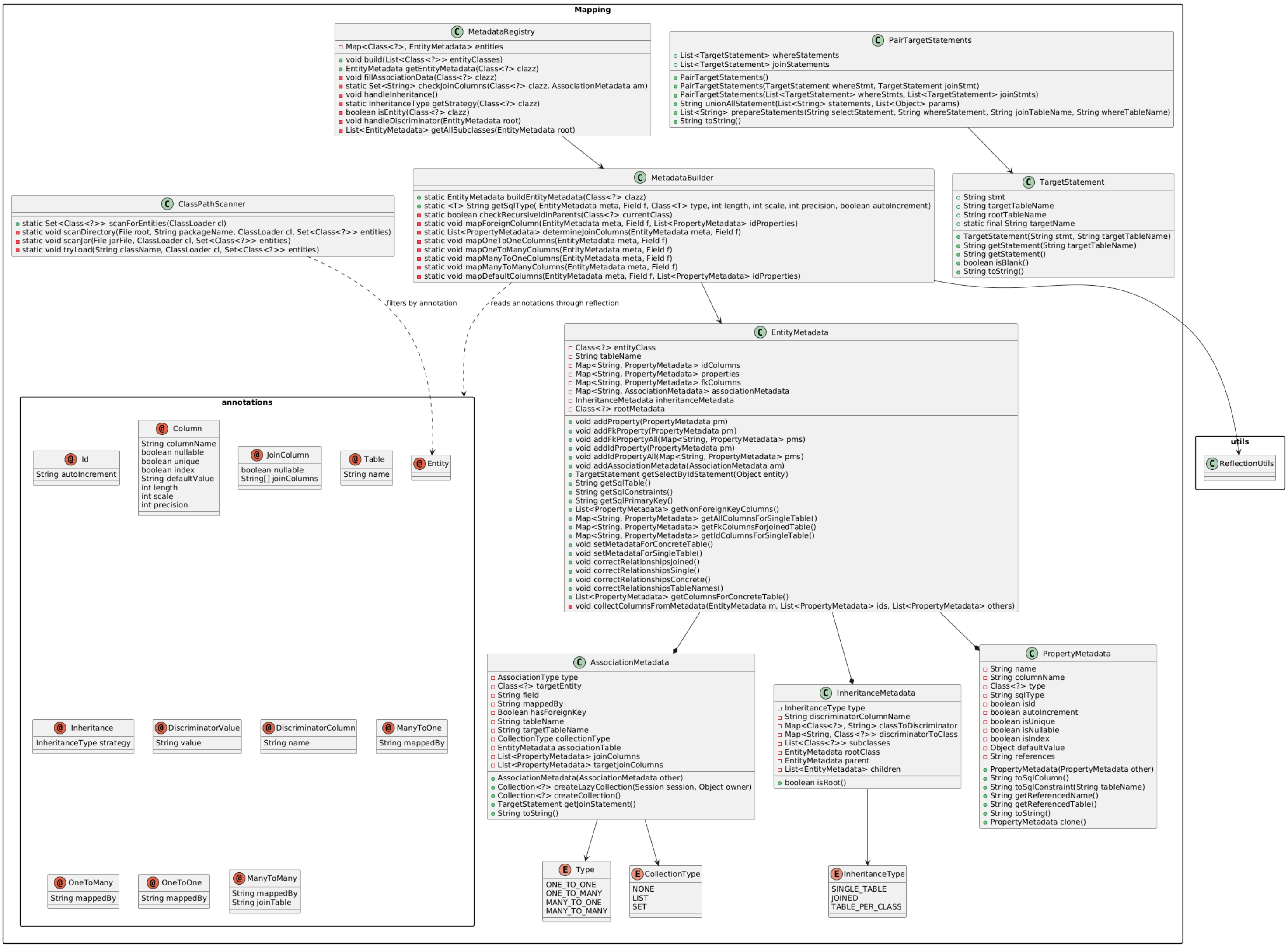
3.1 Package diagram



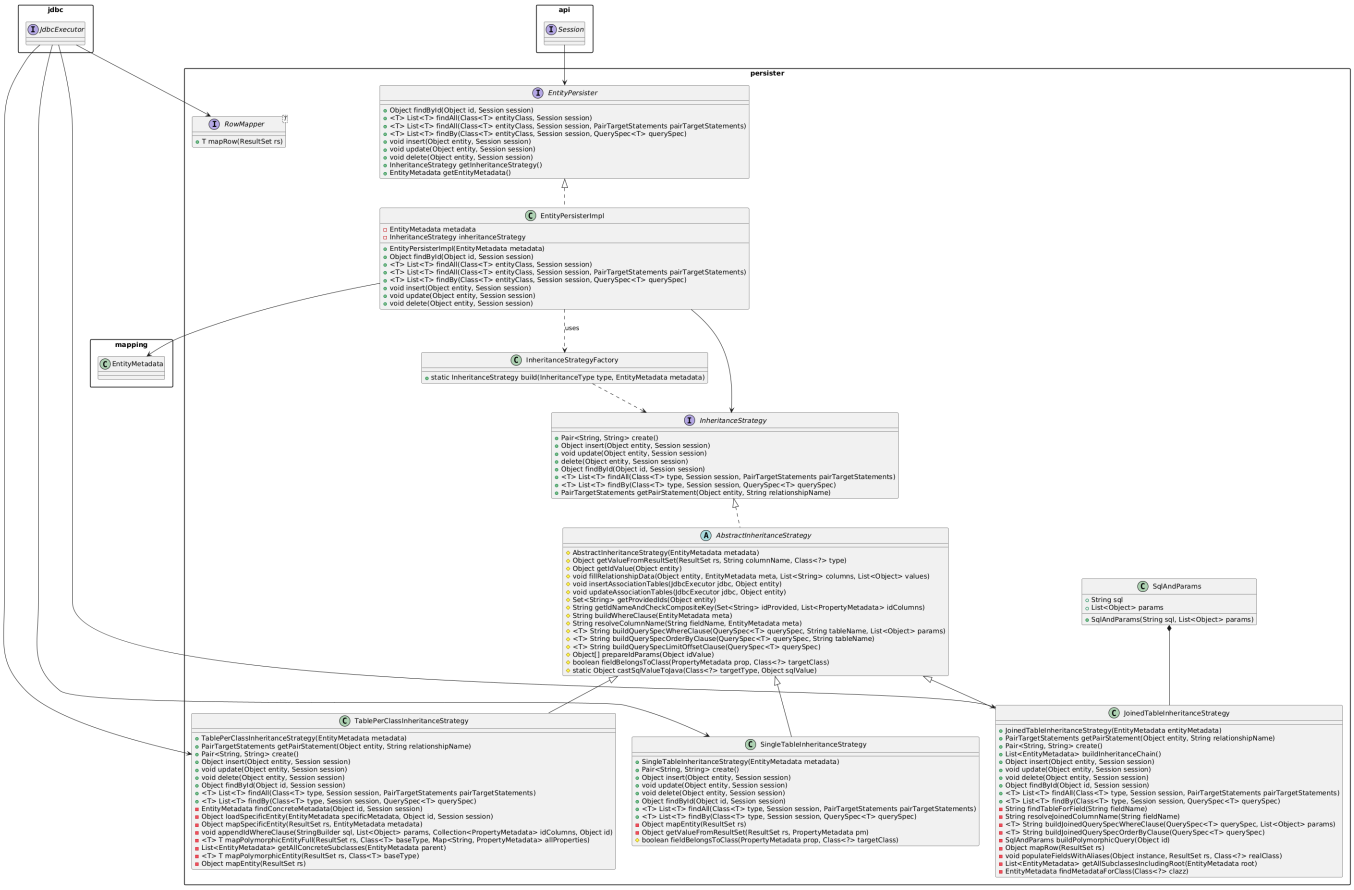
3.2 Package diagram - api



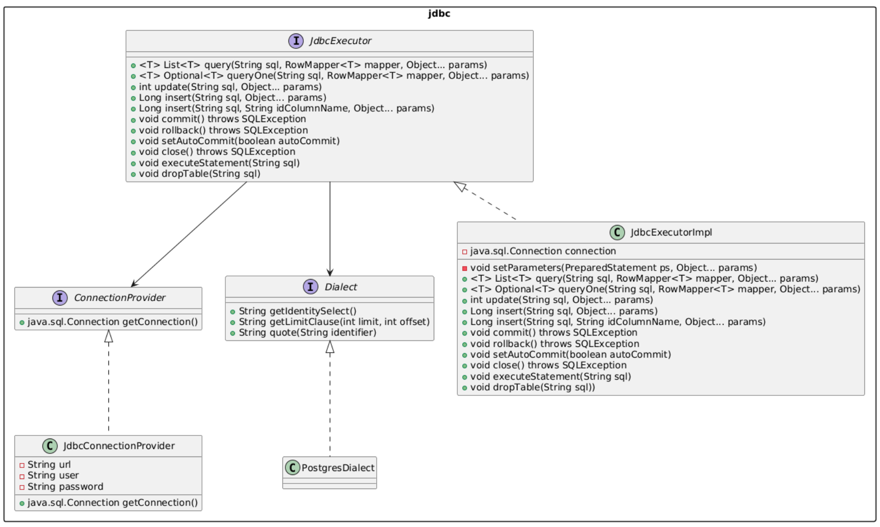
3.3 Package diagram - mapping



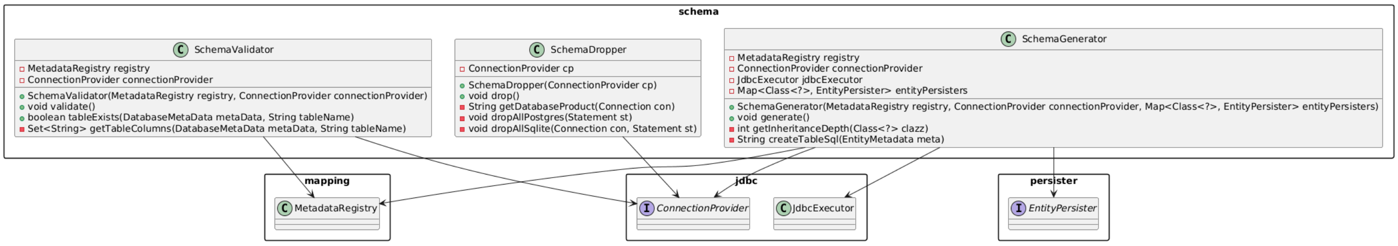
3.4 Package diagram - persister



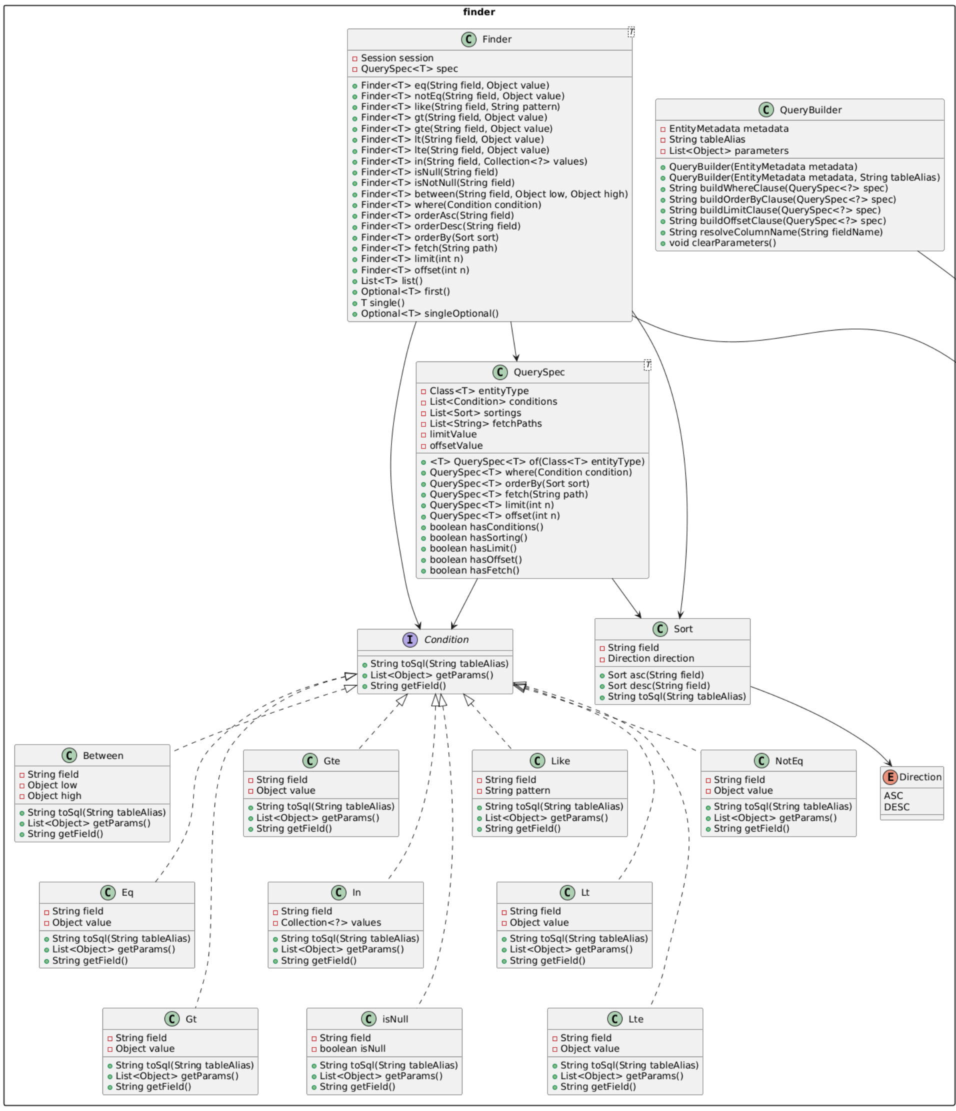
3.5 Package diagram- jdbc



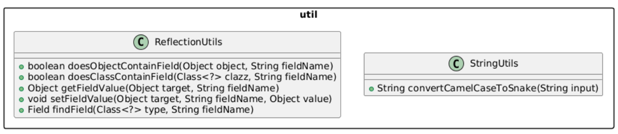
3.6 Package diagram - schema



3.7 Package diagram - finder

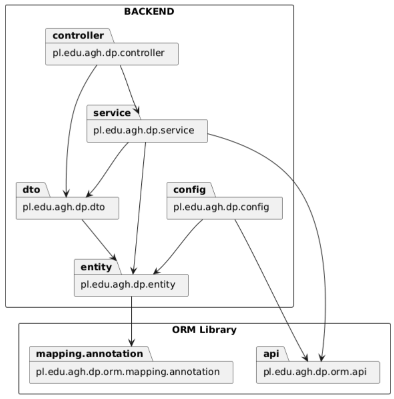


3.8 Package diagram - util

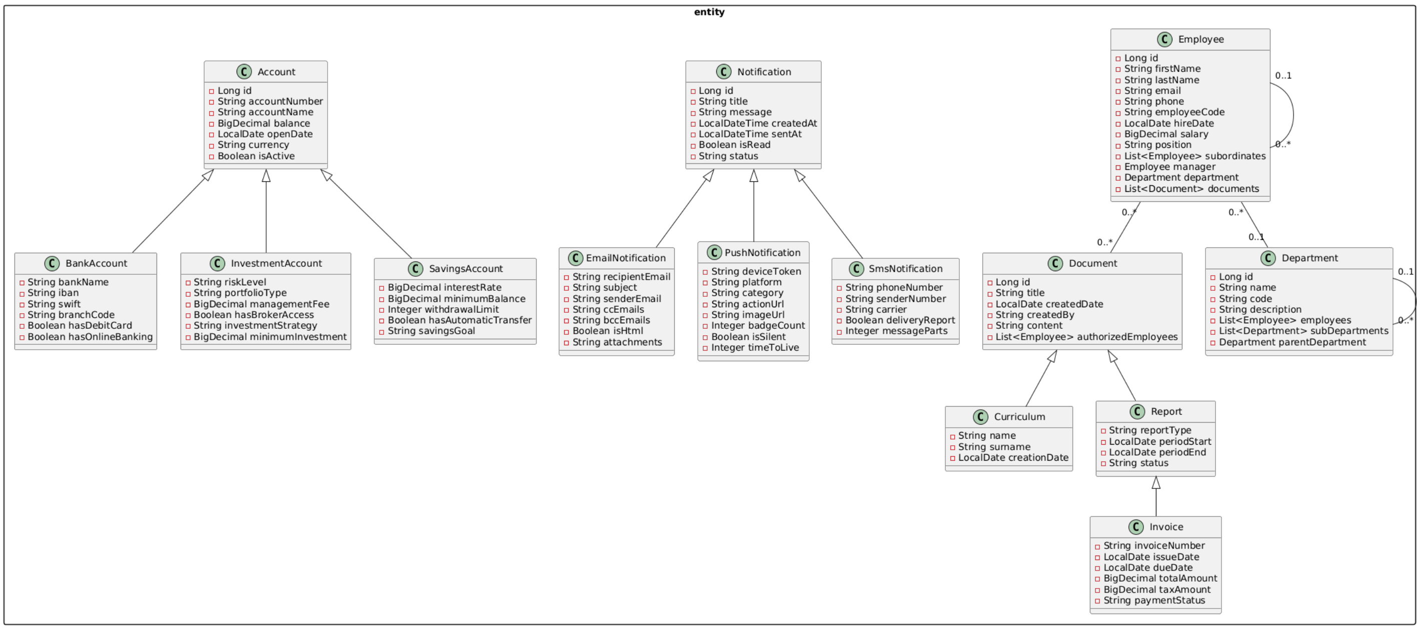


4 Class diagram - Demo Application

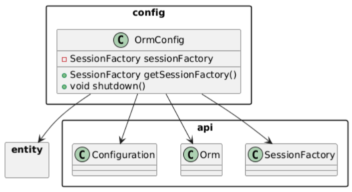
4.1 Package diagram



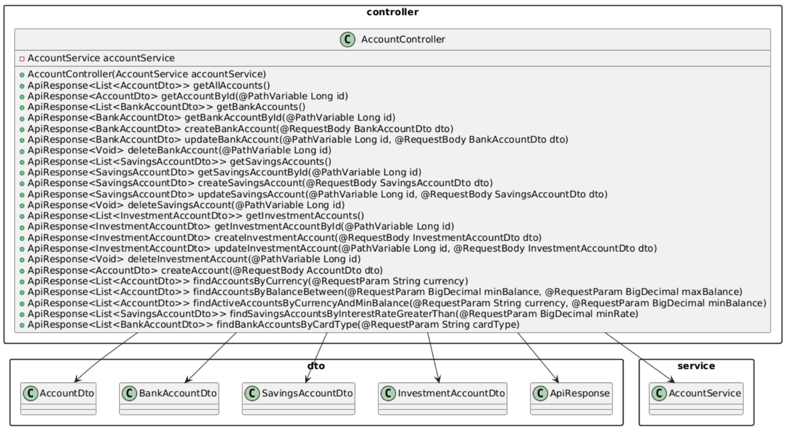
4.2 Package diagram - entity



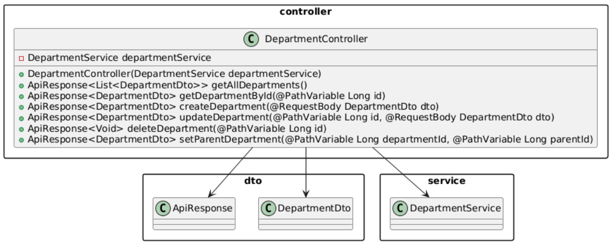
4.3 Package diagram - config



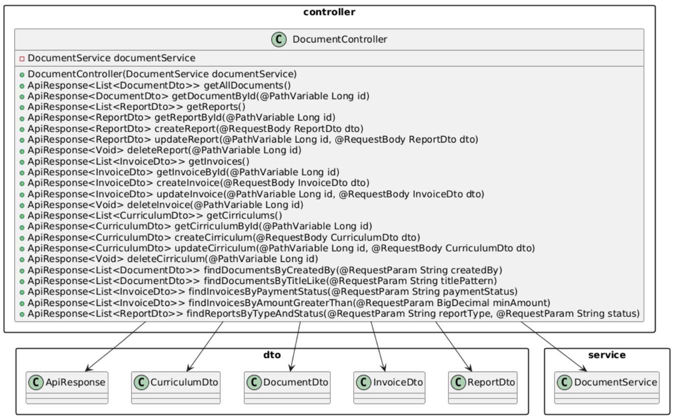
4.4 Class diagram - AccountController (package controller)



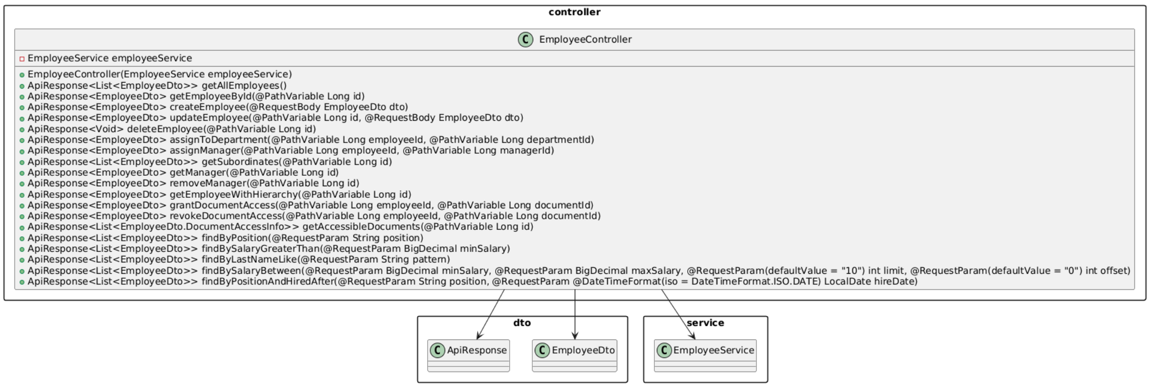
4.5 Class diagram - DepartmentController (package controller)



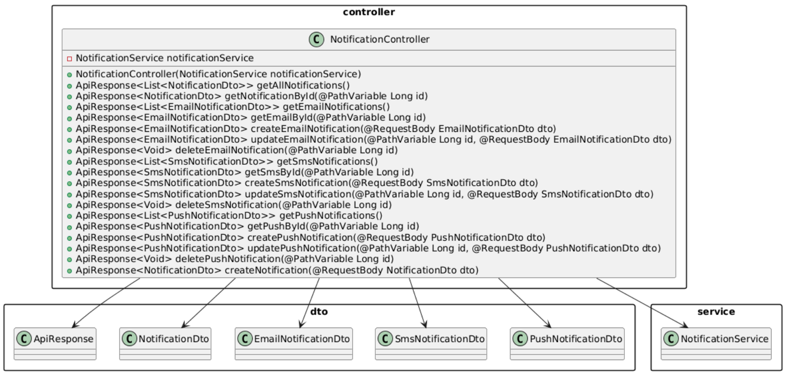
4.6 Class diagram - DocumentController (package controller)



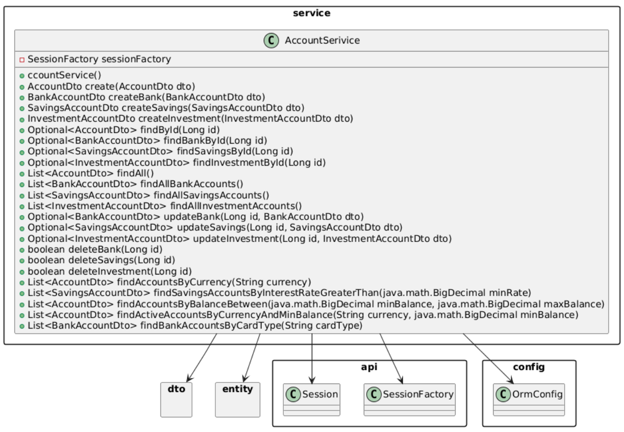
4.7 Class diagram - EmployeeController (package controller)



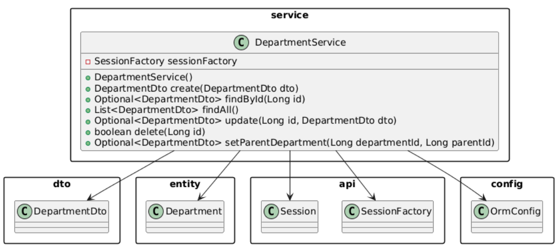
4.8 Class diagram - NotificationController (package controller)



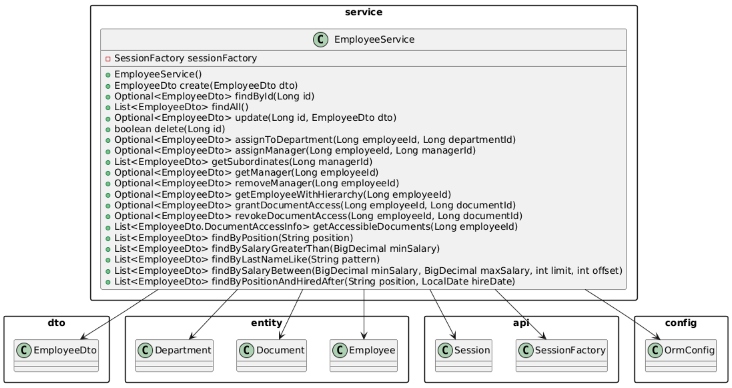
4.9 Class diagram - AccountService (package service)



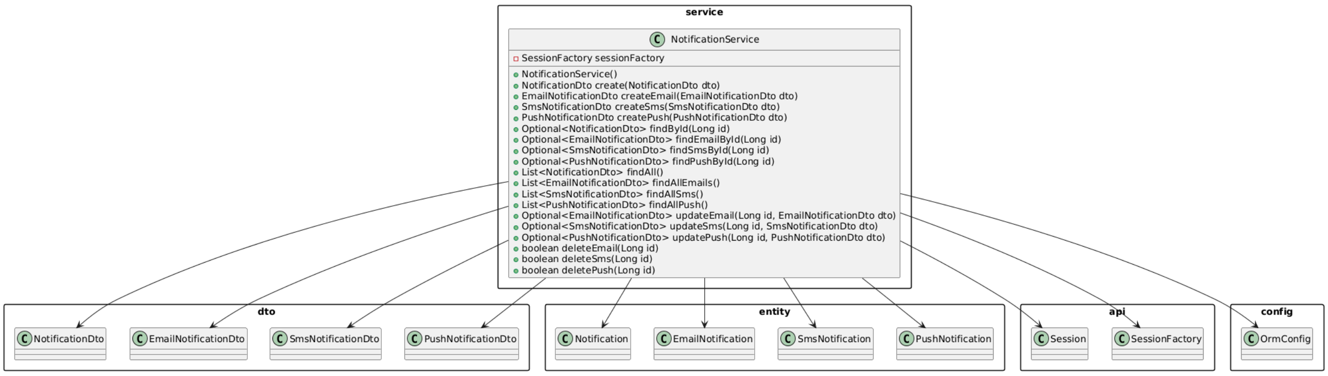
4.10 Class diagram - DepartmentService (package service)



4.11 Class diagram - EmployeeService (package service)



4.12 Class diagram - NotificationService (package service)



5 Used Design Patterns

5.1 Data Mapper

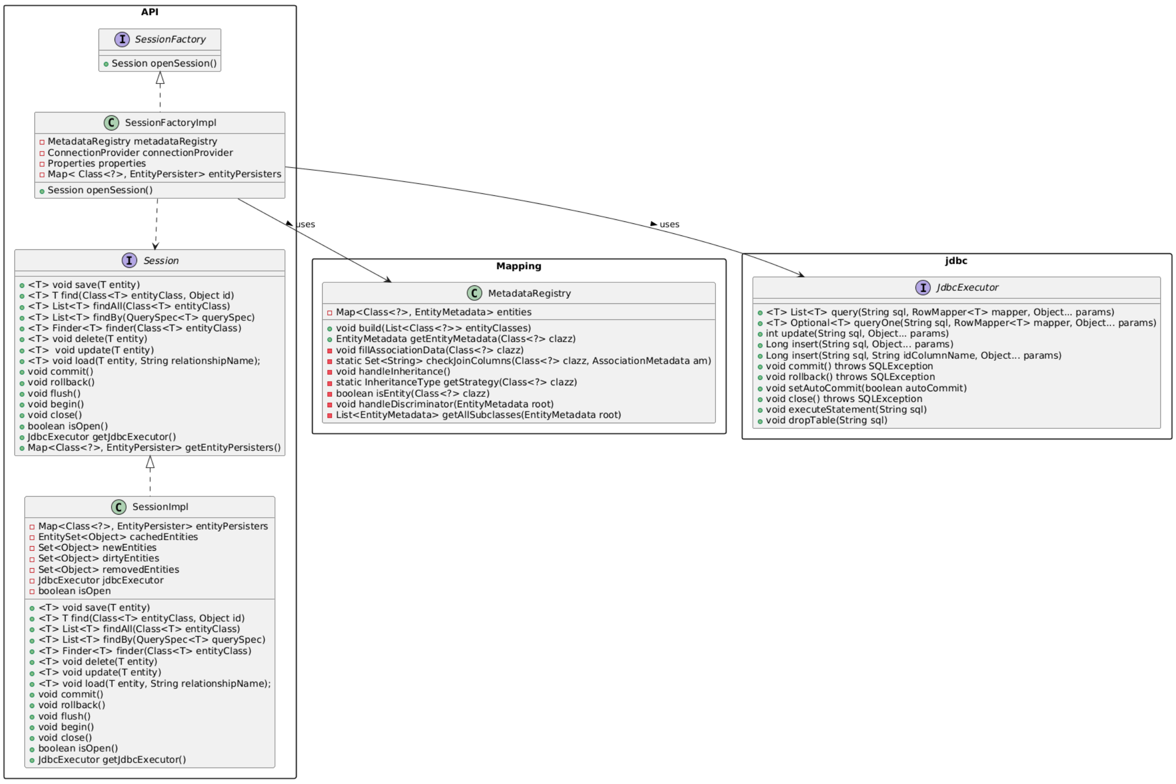
Reason for use: The Data Mapper pattern aims to create an abstraction layer between the database and business logic, enabling their independent development. It maps data from database objects to in-memory data structures and vice versa, minimizing direct dependencies between the core application logic and the underlying database structure. This separation is essential for maintaining ease and flexibility in Java programming.

Negative consequences / trade-offs

- Larger number of classes: entity + metadata (EntityMetaData, InheritenceMetaData, AssociationMetaData) + mapper (EntityPersister)
- Requirement to maintain consistency between metadata and the class (e.g. scanning whether the class has changed)
- More intermediate mechanisms (metadata, reflection, dynamically generated SQL) compared to Active Record

5.2 Factory Method

Reason for use: This creational design pattern lets a class defer instantiation to subclasses, enhancing code flexibility and maintenance.

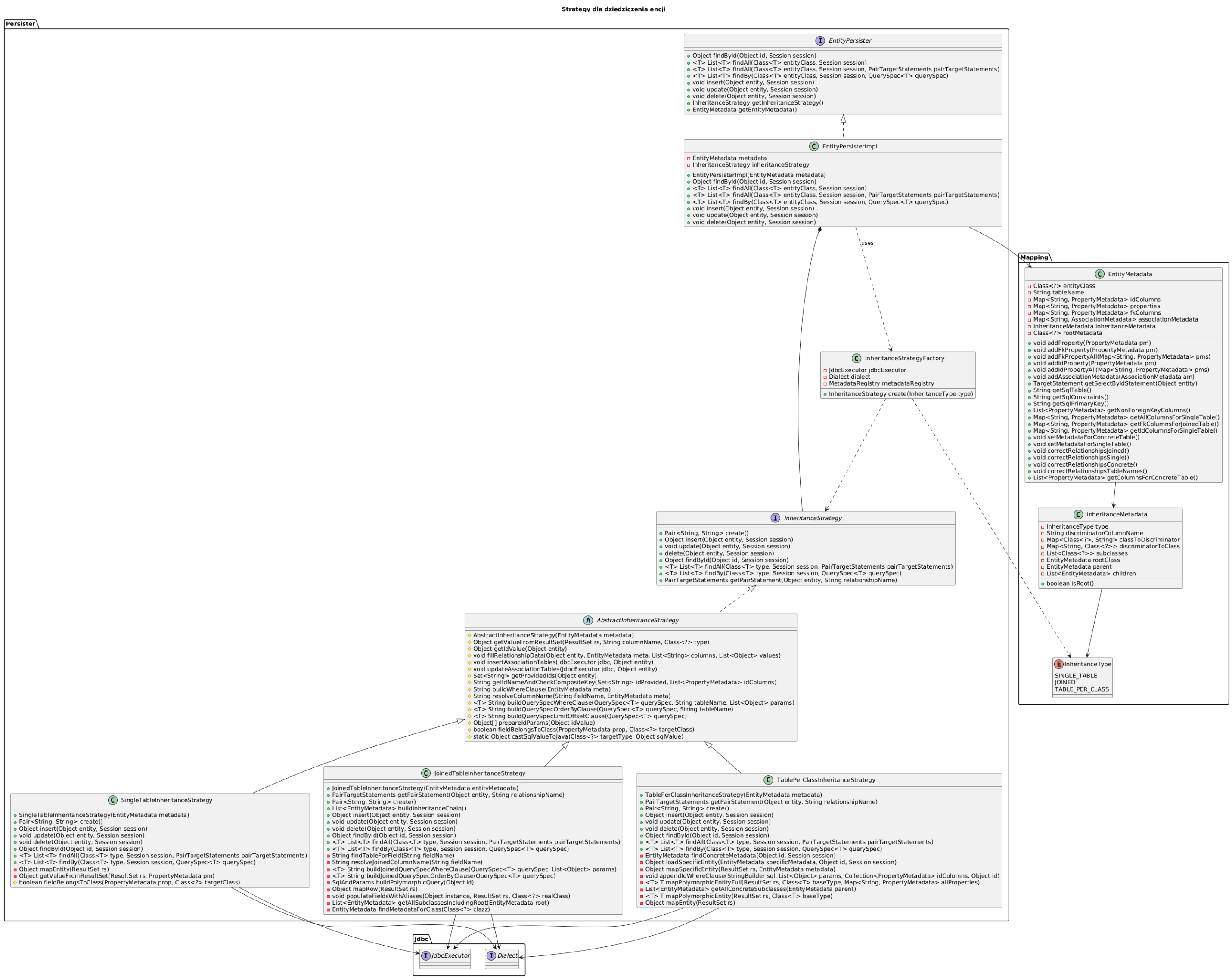


Negative consequences / trade-offs

- Can complicate the code by requiring the addition of new subclasses to implement the extended factory methods.
- Indirection (Code Readability Overhead) - The code becomes harder to follow for new developers due to the added layer of abstraction.

5.3 Strategy

Reason for use: The Strategy pattern is intended to enable the selection of a method for converting class inheritance into table creation and relationships in the database. The client can choose one of the available strategies belonging to the inheritance strategy family. This makes it possible to apply flexible and interchangeable strategies within the same system.



Negative consequences / trade-offs

- **Increased number of classes and wiring.** Introducing Strategy typically adds an interface, multiple implementations, and a selection mechanism (e.g., configuration or factory), which increases the overall amount of code and configuration to maintain.
- **More indirection, harder debugging and tracing.** The runtime behavior depends on which strategy is selected, so it can be less straightforward to follow the execution path and understand why a particular decision was made.
- **Potential performance overhead.** Although usually small, additional abstraction layers may introduce minor runtime overhead (extra allocations, dynamic dispatch). In an ORM context, this must be controlled to avoid overhead in hot paths (e.g., SQL generation/execution loops).

5.4 Inheritance Strategies

	Single Table Inheritance (STI)	**Joined Inheritance**	**Table Per Class Inheritance**
Database structure	One table for the entire hierarchy.	A table for the base class + a table for each subclass (with JOINs).	A separate table for each concrete class (with duplicated inherited fields).
Advantages	1. Simple and fast queries (no JOINs). 2. Easy to implement. 3. Good performance for small hierarchies.	1. No NULLs in tables (better normalization). 2. Flexible for large hierarchies. 3. Easy to add subclasses.	1. Fast queries (no JOINs). 2. No NULLs. 3. Good read performance for specific types.
Disadvantages	1. May result in many NULLs in subclass-specific columns. 2. Poor normalization (everything in one table). 3. Difficult to manage for large hierarchies with many unique fields.	1. Slower queries due to JOINs (especially for deep hierarchies). 2. Increased SQL query complexity. 3. Higher resource usage caused by JOINs.	1. Column duplication (inherited fields are repeated across tables). 2. Difficult changes to the base class (require modifications to multiple tables). 3. Does not support polymorphism for base-class queries without UNIONs.
Performance	High for read/write operations (single table).	Medium – JOINs slow things down, but good for normalized data.	High for reading specific types, but UNIONs for polymorphic queries may reduce performance.