Persistence Layer

December 21, 2024

1 Introduction

The **persistence** layer acts as an intermediary between the application's business logic and the data storage systems. Its main roles include:

- **Storing data** : It saves data generated by the application so that it can be retrieved later, even after the application is closed.
- **Retrieving data** : It fetches stored data when needed, allowing the application to display or manipulate this information.

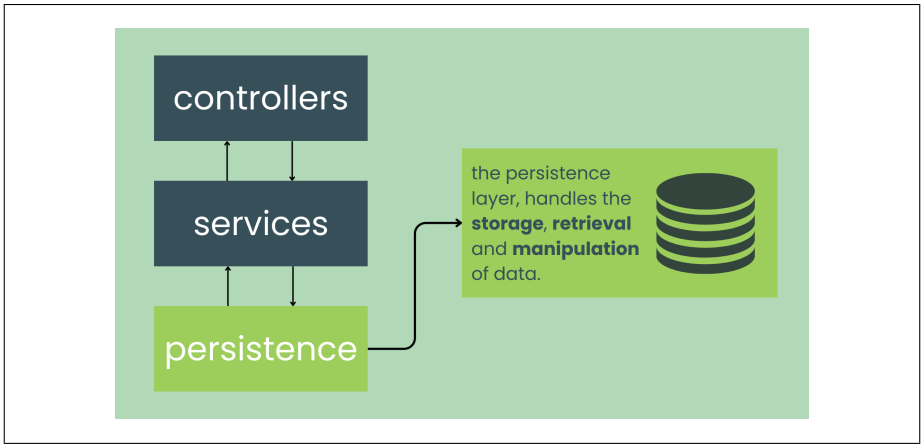

Figure 1: Application layers

In the next sections, we are going to walk trough the different approaches to set the persistence layer up for a multi-layered application.

2 JDBC classic approach

2.0.1 JDBC API

- **JDBC** (Java Database Connectivity) is an API (Application Programming Interface) that enables Java applications to interact with **relational databases**. It provides a standard set of interfaces and classes to connect to databases, execute SQL queries, and manage the results.
- **Low level API** : JDBC is a low-level API, meaning developers manually manage database connections, handle SQL exceptions, and close resources.

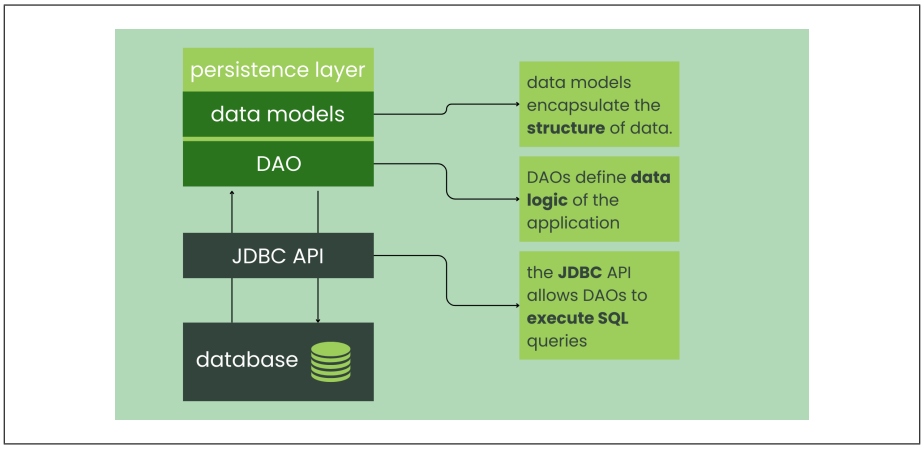

Figure 2: JDBC API

2.1 Setup for the connection

The first step in setting up the persistence layer is to include the JDBC dependency in the application classpath, either manually or by adding the appropriate Maven dependency tag.

The architecture outlined below represents the foundation for establishing a connection in the application.

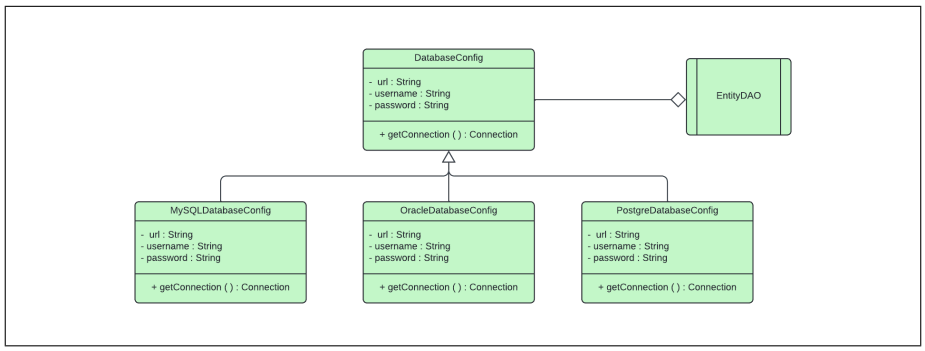

Figure 3: Database Connection Setup

- **Database Configuration**: This is an abstract class that defines the common behavior for all database configuration classes. The `getConnection()` method in this class returns the connection to the specified RDBMS.
- **Child Classes**: Each child class is responsible for establishing a connection to a specific RDBMS, inheriting from the base `DatabaseConfiguration` class.
- **DAO Class**: The DAO (Data Access Object) class handles the implementation of data logic, executing SQL queries, and using the connection provided by the database configuration class to interact with the database.

2.2 Business Data Logic

The business data logic is built upon two core components.

The architecture shown below outlines the general structure of the persistence layer in a **CRUD** application.

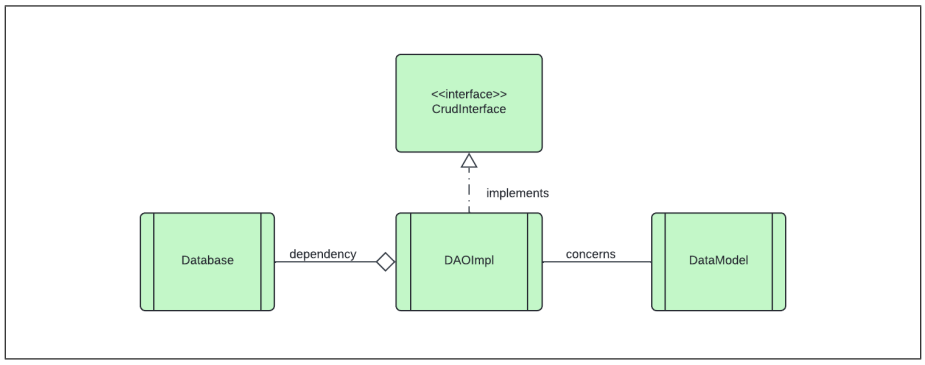

Figure 4: Persistence Layer Architecture in a CRUD Application

Key Components:

- **Data Model**: A class that defines the common behavior for all concrete data model classes, representing the structure of the data that will be managed by the application.
- **CRUD Interface**: An interface that defines the method signatures for the core `CRUD` operations: Create, Read, Update, and Delete.
- **DAO Implementation**: Concrete classes that implement the CRUD methods for a specific data model, handling the logic for interacting with the database.

2.3 Coding example

Below is a simple coding example illustrating how to implement the persistence layer for a `User` entity using JDBC. This includes database configuration, a data model, a CRUD interface, and a DAO implementation. The full code is available on the project's GitHub repository:
`https://github.com/D0esN0tM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jdbc_classic_approach`.

2.3.1 Database Configuration

The **DatabaseConfiguration** class is an abstract utility class that centralizes the database connection logic. This ensures that the connection details (like URL, username, and password) are managed in one place, promoting reusability and maintainability.

Explanation:

- `DriverManager.getConnection`: Establishes a connection to the database using the specified URL, username, and password.
- Constants like `URL`, `USER`, and `PASSWORD` store the configuration details securely and make them easily updatable.
- `Connection`: Represents an active database connection, which is necessary for executing SQL queries.

2.3.2 User Model

The **User** class serves as a data model to represent the structure of a record in the database. Each instance corresponds to a row in the `users` table.

Explanation:

- Fields like `id`, `name`, and `email` map directly to the columns of the database table.
- Getter and setter methods allow encapsulation, enabling controlled access to these fields.

2.3.3 CRUD Interface

The **UserDAO** interface defines the contract for performing **CRUD** operations on the `User` data.

Explanation:

- `createUser(User user)`: Adds a new user record to the database.
- `getUserById(int id)`: Fetches a user record based on its ID.
- `getAllUsers()`: Retrieves all user records from the database.
- `updateUser(User user)`: Updates an existing user record with new details.
- `deleteUser(int id)`: Deletes a user record based on its ID.

2.3.4 DAO Implementation

The **UserDAOImpl** class implements the `UserDAO` interface. It provides the concrete logic for interacting with the database.

Explanation:

- `PreparedStatement`: Used to safely execute parameterized queries, preventing SQL injection attacks.
- `ResultSet`: Stores the result of `SELECT` queries, allowing iteration through the returned rows.
- Each method utilizes a `try-with-resources` block to ensure that connections, statements, and other resources are closed automatically.

3 JDBC template approach

3.1 JDBC template

The Spring **JdbcTemplate** is a high-level API in the Spring Framework that simplifies database interactions by handling common tasks such as connection management, SQL statement execution, and exception handling, making data access code more concise and reducing boilerplate associated with the standard **JDBC API**.

NOTE : In order to use JDBC template, it is necessary to declare it as a dependency in the `pom.xml` file

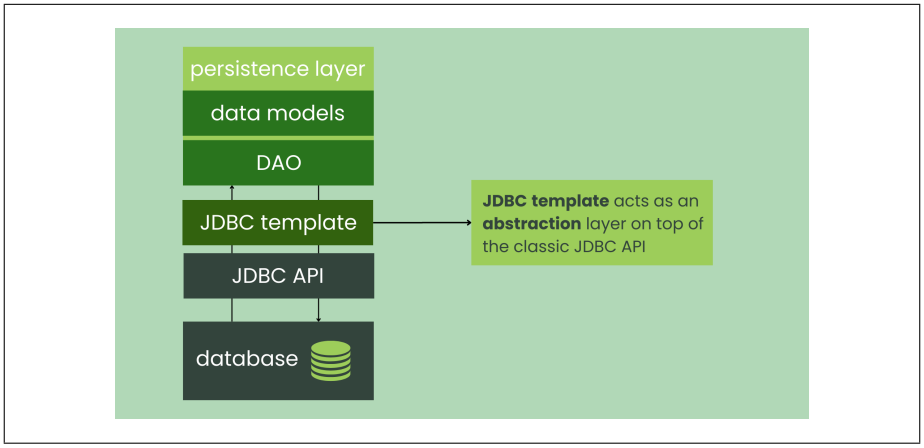

Figure 5: JDBC template

3.2 Setup for the Connection

In a Spring Boot application, connection details are defined in the `application.properties` file. Based on this configuration, the Spring IoC (Inversion of Control) container automatically generates a `DataSource` object, which is then injected into the `JdbcTemplate` object to enable database operations.

```
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/
    mydatabase
spring.datasource.username=myuser
spring.datasource.password=mypassword
spring.datasource.driver-class-name=com.mysql.cj.
    jdbc.Driver
```

Listing 1: MySQL database configuration

3.3 Data Business Logic

In the context of using the JDBC template, the persistence layer follows an architecture similar to the previous approach. Each data model has a corresponding DAO class responsible for managing its database operations.
However, instead of using `Connection` objects to interact with the database, DAOs now use the `JDBC template`.

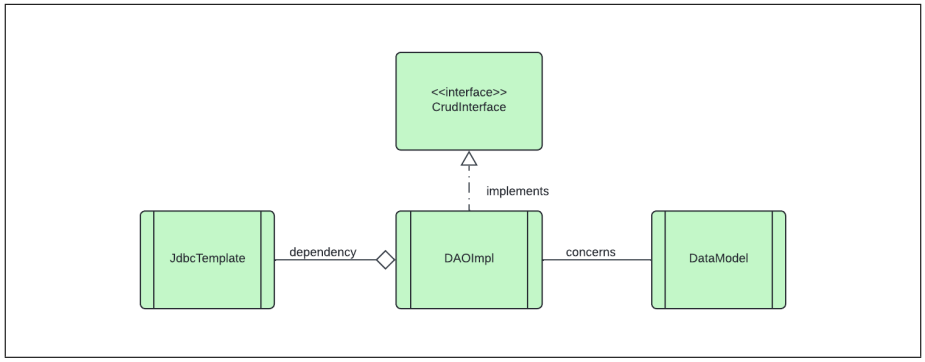

Figure 6: JDBC template

Setting up the persistence layer in this context involves several key steps:

1. **Configuring JdbcTemplate as a Bean**: In a `@Configuration` class, define `JdbcTemplate` as a Spring bean. This allows the `JdbcTemplate` instance to be injected as a dependency into the DAO classes, enabling them to perform database operations.

2. **Creating Data Models**: Define data model classes that represent the entities in the database. These classes will be used to map database records to Java objects.

3. **Defining the CRUD Interface**: Create an interface that specifies the CRUD (Create, Read, Update, Delete) operations. This interface will be implemented by the DAOs to handle data access in a consistent way.

4. **Implementing DAOs**: Develop DAO classes that provide concrete implementations of the CRUD operations defined in the interface. These classes will use `JdbcTemplate` to interact with the database.

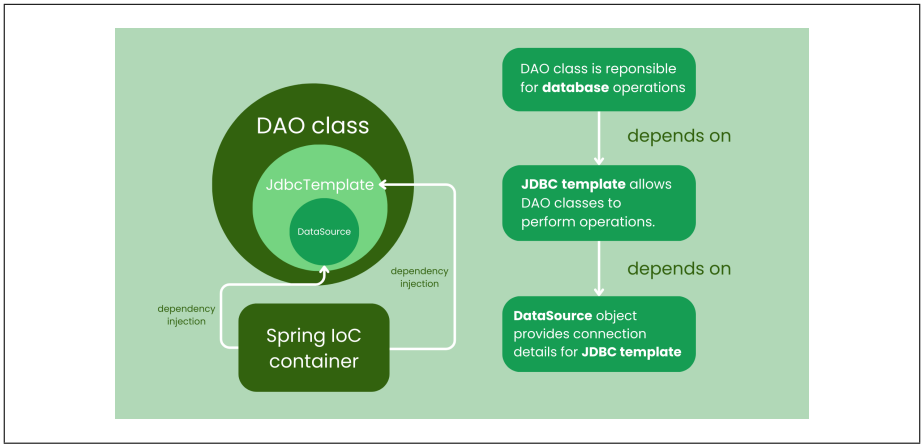

Figure 7: Dependency Injection in the Persistence Layer

3.4 Coding example

The following example demonstrates the implementation of a persistence layer for managing a `Product` entity using Spring JDBC. It includes database configuration, a data model, a DAO interface, and a DAO implementation. The full code is available on the project's GitHub repository: `https://github.com/D0esN0tM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jdbctemplate_example`.

3.4.1 Database Configuration

The **DatabaseConfig** class defines the `JdbcTemplate` bean, which is essential for executing SQL operations in Spring applications.

Explanation:

- `JdbcTemplate`: A Spring-provided utility for interacting with the database in a straightforward and efficient way.
- `DataSource`: Injected into the `JdbcTemplate`, it provides the connection to the database.

3.4.2 Product Data Model

The **Product** class serves as a data model for the `Product` entity, representing a database record.

Explanation:

- The `@Data`, `@AllArgsConstructor`, and `@NoArgsConstructor` annotations from Lombok automatically generate boilerplate code like getters, setters, and constructors.

- Fields like `id`, `label`, and `price` map directly to the database table's columns.

3.4.3 Product DAO Interface

The **ProductDao** interface defines the CRUD operations for the `Product` entity.

Explanation:

- Methods like `save`, `update`, and `deleteById` handle the persistence logic for the `Product` entity.
- `findById` and `findAll` allow retrieval of one or multiple products, respectively.

3.4.4 DAO Implementation

The **ProductDaoImpl** class implements the `ProductDao` interface, providing concrete methods for interacting with the database.

Explanation:

- `RowMapper`: Maps each row of the result set to a `Product` object.
- `KeyHolder`: Captures the auto-generated ID when a new product is inserted into the database.
- Each method employs `JdbcTemplate` to execute the SQL queries efficiently.

4 Java Persistence API Approach

This section outlines the usage of the Java Persistence API (JPA) and Hibernate for implementing a persistence layer in Java applications. It covers an introduction to JPA, the setup process for establishing a database connection, and the integration of business logic.

4.1 JPA and Hibernate

The Java Persistence API (JPA) is a standard specification for object-relational mapping (ORM) in Java. It provides a platform-independent approach to interact with relational databases by mapping Java objects to database tables. Hibernate is a popular JPA implementation that offers advanced ORM features and seamless integration with JPA.

Key Features:

- Eliminates boilerplate code for managing database operations.
- Provides annotations for mapping Java classes to database tables.

- Offers a query language (JPQL) for interacting with the database in an object-oriented manner.
- Manages database connections and transactions automatically.

Benefits of JPA and Hibernate:

- Simplifies database operations through entity management.
- Reduces the likelihood of SQL injection by using parameterized queries.
- Improves portability across different databases.

4.2 Setup for the Connection

To use JPA and Hibernate in a project, certain configurations are required to establish a connection to the database and manage entities. In a Spring Boot application, the connection and JPA settings can be configured in the `application.properties` file.

Steps for Setup:

1. Add dependencies for JPA and Hibernate in the `pom.xml` file (Maven) or `build.gradle` file (Gradle).
2. Create a configuration file (`application.properties`) to define database properties such as URL, username, password, and Hibernate-specific settings.
3. Annotate Java classes with JPA annotations (`@Entity`, `@Table`, etc.) to define them as entities mapped to database tables.
4. Use the `EntityManagerFactory` to create an `EntityManager`, which provides an interface for interacting with the persistence context.

Application Properties Configuration: To configure the MySQL database connection in a Spring Boot application, the following properties should be added to the `application.properties` file:

Example of `application.properties` Configuration:

```
# MySQL Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/
    testdb?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.
    jdbc.Driver
spring.jpa.database-platform=org.hibernate.dialect.
    MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

This configuration ensures that Spring Boot uses Hibernate as the JPA provider and establishes the connection to the MySQL database with the necessary credentials and settings.

4.3 Data Business Logic

The business logic layer interacts with the persistence layer to perform CRUD operations and apply business rules.

Responsibilities of the Business Logic Layer:

- Validating and transforming data before persistence.
- Coordinating between the persistence layer and other layers of the application (e.g., controllers).
- Managing database transactions, ensuring data consistency.
- Abstracting the persistence logic from the presentation layer.

Using Repositories: Repositories in JPA are interfaces that allow you to define and manage database interactions with minimal boilerplate code. Hibernate can extend these repositories to provide custom queries and advanced features.

Entity Lifecycle Management: JPA provides different states for entities such as *transient*, *persistent*, and *detached*, allowing developers to manage the entity's lifecycle effectively.

4.4 Coding Example

In this section, we will walk through a simple example demonstrating how to set up and use JPA with Hibernate in a Spring Boot application. The full code is available on the project's GitHub repository:
`https://github.com/DOesNOtM1tter/spring_boot_project-/tree/master/supporting_project/persistence_examples/jpa_examples`.

Step 1: Entity Class

To define a JPA entity, we need to annotate the class with `@Entity` and map it to a table in the database using `@Table`. Additionally, we use `@Id` to mark the primary key field and `@GeneratedValue` for automatic generation of the ID.

Explanation:

- `@Entity`: Marks the class as a JPA entity.
- `@Id`: Specifies the primary key of the entity.

- `@GeneratedValue`: Configures the strategy for generating the primary key (e.g., `IDENTITY` for auto-incremented IDs).
- `@Column`: Maps the fields to the database columns, with options like `nullable` to define column constraints.

4.4.1 Product Repository

The repository interface extends `JpaRepository`, providing built-in CRUD operations.

Explanation:

- `JpaRepository`: Provides methods like `save()`, `findById()`, `findAll()`, and `deleteById()` for database operations without needing custom implementations.
- Custom query methods can be added if needed.

4.4.2 Service Layer

The service layer interacts with the repository to handle business logic.

Explanation:

- The service layer is used to abstract the business logic, making the controller layer cleaner.
- The `@Autowired` annotation injects the repository, allowing easy dependency injection.
- Methods like `saveProduct()` and `getProductById()` provide access to the repository's methods for creating, retrieving, and deleting products.

4.4.3 Controller Layer

The controller exposes the service methods via RESTful endpoints for front-end interaction.

Explanation:

- `@RestController`: Marks the class as a REST controller that handles HTTP requests.
- `@RequestMapping`: Specifies the base URL for the controller's endpoints.
- `@PostMapping`, `@GetMapping`, and `@DeleteMapping`: Handle HTTP POST, GET, and DELETE requests respectively.
- `@RequestBody` and `@PathVariable`: Handle request payloads and dynamic URL parameters.

Step 5: Application Configuration

Ensure that the application is properly configured to run a Spring Boot JPA application. The configuration details can be set in the `application.properties` file, as described in the previous section.

This setup creates a complete flow for managing entities in a Spring Boot application with JPA and Hibernate. With minimal configuration, Spring Boot handles most of the boilerplate code, allowing developers to focus on business logic.

5 Conclusion

In this chapter, we explored different approaches for managing the persistence layer in a Spring Boot application. We covered the implementation of persistence with JDBC, detailing the database configuration, data models, and CRUD interfaces, as well as using `JdbcTemplate` to interact with the database. In parallel, we also discussed setting up persistence with JPA and Hibernate, configuring the database connection, and defining entities using JPA annotations. These two approaches offer different advantages depending on the project's needs: JDBC for more manual and controlled operations, and JPA/Hibernate for automated management and simplified object-relational mapping. By combining these techniques, we can build a robust, flexible, and maintainable persistence layer.