**Group Information:**

| Mohammed Abdul Bari Waseem- 500225922  Syed Abdul Qadeer – 500228186  Mohammed Aafaq Hussain -500228114  Anas Baseer Mohammed – 500229851  Balakrishna Prateek Gopal Nandiwada - 500220142 |  |
| --- | --- |
|  |  |
| **Week-5**  **Assignment 1: Initial Setup**  **Assignment 2: Implementing Service Layer**  **Assignment 3: Integrating DAO Layer with Service** |  |

**Group Assignment-3**

**1. Introduction**

In this assignment, we integrated the **DAO layer** with the **service layer** to handle CRUD operations for the "Product" entity in a Spring Boot application. This assignment builds on the Week 2 implementation, where we established the service layer for business logic. In Week 3, we integrated the DAO layer to manage database interactions and completed the implementation of CRUD functionalities.

**2. DAO Layer Integration**

**2.1. Purpose of the DAO Layer**

The **DAO (Data Access Object)** layer serves as an abstraction between the service layer and the database. It provides methods to interact with the database, allowing us to perform CRUD operations on the "Product" entity without directly handling SQL queries in the service layer.

**2.2. DAO Interface: ProductDao.java**

The DAO interface defines the method signatures for performing CRUD operations on the "Product" entity. These methods are:

* saveProduct(Product product)
* getAllProducts()
* getProductById(Long id)
* updateProduct(Product product)
* deleteProduct(Long id)

These methods ensure interaction with the database is separated from business logic in the service layer.

**CODE SCREENSHOT:**

A screenshot of a computer

Description automatically generated

**2.3. DAO Implementation: ProductDaoImpl.java**

In the DAO implementation class (ProductDaoImpl.java), we use Spring's EntityManager to implement the CRUD methods defined in the ProductDao interface. The EntityManager is responsible for executing the necessary queries on the database.

A screenshot of a computer

Description automatically generated

**3. Service Layer Integration**

**3.1. Purpose of the Service Layer**

The **service layer** is responsible for implementing the business logic and handling CRUD operations through the DAO layer. It acts as an intermediary between the controller (which handles HTTP requests) and the DAO layer (which interacts with the database). The service layer also manages exception handling and transactions.

**3.2. Service Class: ProductService.java**

The service class, ProductService, implements business logic for the "Product" entity. It contains methods such as:

* createProduct(Product product)
* getAllProducts()
* getProductById(Long id)
* updateProduct(Product product)
* deleteProduct(Long id)

These methods delegate the actual CRUD operations to the DAO layer while handling any necessary validation or exception management.

A screenshot of a computer

Description automatically generated

**4. Controller Layer Integration**

**4.1. Purpose of the Controller Layer**

The **controller layer** is responsible for handling HTTP requests from the client (such as Postman or a web interface) and mapping those requests to corresponding service methods. The controller layer acts as the entry point to the backend, making it accessible via REST API endpoints.

**4.2. Controller Class: ProductController.java**

The ProductController.java class contains REST API endpoints that allow clients to perform CRUD operations on products. The methods are mapped to HTTP verbs as follows:

* POST /products — to create a new product
* GET /products — to retrieve all products
* GET /products/{id} — to retrieve a product by its ID
* PUT /products/{id} — to update a product
* DELETE /products/{id} — to delete a product

The controller layer calls the service layer methods to perform the necessary CRUD operations.

A screenshot of a computer

Description automatically generated

**5. Data Source Configuration**

The **application.properties** file is configured to connect to an H2 in-memory database. This allows for testing and development without the need for a physical database

A screenshot of a computer

Description automatically generated

**6. Testing with Postman**

We used **Postman** to test the REST API endpoints, ensuring that all CRUD operations function as expected. Below is the summary of the Postman tests conducted:

**6.1. Creating a Product (POST Request)**

* **Endpoint:** POST /products
* **Request Body:** A JSON object representing the product to be created.
* **Response:** A 201 Created status code with the details of the newly created product.

A screenshot of a computer

Description automatically generated

**6.2. Retrieving All Products (GET Request)**

* **Endpoint:** GET /products
* **Response:** A 200 OK status code with a JSON array of all products in the database.

A screenshot of a computer

Description automatically generated

**6.3. Retrieving a Product by ID (GET Request)**

* **Endpoint:** GET /products/{id}
* **Response:** A 200 OK status code with the details of the requested product, if found.

A screenshot of a computer

Description automatically generated

**6.4. Updating a Product (PUT Request)**

* **Endpoint:** PUT /products/{id}
* **Request Body:** A JSON object with updated product details.
* **Response:** A 200 OK status code with the updated product details.

A screenshot of a computer

Description automatically generated

**6.5. Deleting a Product (DELETE Request)**

* **Endpoint:** DELETE /products/{id}
* **Response:** A 200 OK status code confirming the product was deleted.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**7. Challenges Faced**

**7.1. Challenge 2: Exception Handling**

Handling exceptions gracefully across layers was important. In case of any errors in the DAO layer (e.g., database issues), we caught them in the service layer and returned appropriate error messages to the client.

**A screenshot of a computer

Description automatically generated**

**8. Optimizations**

**8.1. Optimizing Database Interactions**

In the DAO layer, we ensured that database interactions were optimized by using batch operations where possible and reducing unnecessary database queries.

**8.2. Exception Handling Improvements**

We refined exception handling to provide more meaningful error messages and reduce the likelihood of uncaught exceptions breaking the application.

A screenshot of a computer

Description automatically generated

--------------------------------------------------------------------------------------------------------------**EXPLANATION**

**1. DAO Layer Integration**

The DAO (Data Access Object) layer is responsible for abstracting the interaction between the application and the underlying database. In this assignment, we integrated the DAO layer into our Spring Boot application to handle CRUD operations for the Product entity.

* **Role in the Application**: The DAO layer provides methods to interact with the database, encapsulating all SQL queries and operations (like saving, retrieving, updating, and deleting products). It ensures that the service layer does not directly interact with the database, maintaining separation of concerns and promoting modularity.
* **Interaction with Other Layers**:
  + **Service Layer**: The DAO layer is injected into the service layer. The service layer calls the DAO methods to perform business logic. This maintains a clean separation of concerns, where the service layer handles the business rules, while the DAO layer manages database interactions.
  + **Controller Layer**: The controller layer handles HTTP requests, delegating business logic to the service layer. In our case, the service layer, in turn, delegates data operations to the DAO layer.

**2. Role of the DAO Layer in the Application Architecture**

In a typical Spring Boot application architecture, the DAO layer functions as the bridge between the persistence (database) and the application’s business logic.

* **Data Persistence Management**: The DAO layer directly interacts with the database, abstracting the database-specific implementation details from the rest of the application. This promotes a clean separation of concerns.
* **CRUD Operations**: It manages CRUD (Create, Read, Update, Delete) operations through defined methods. In our project, these methods are used by the service layer to perform various operations on the Product entity.
* **Database Abstraction**: The DAO layer provides an abstraction over the database, allowing for easier changes in the database implementation without affecting the business logic or controller layers. For instance, if we decided to switch from H2 to MySQL or MongoDB, only the DAO layer would need to be updated.
* **Interaction with Other Layers**: The DAO layer works in conjunction with the service layer to ensure that the business logic and database operations are kept separate. The service layer calls the DAO methods to perform database operations, and the controller layer calls the service methods to handle HTTP requests.

**3. Challenges Faced During Integration and Solutions**

* **Challenge 1**: Ensuring proper transaction management.
  + **Solution**: We resolved this challenge by adding the @Transactional annotation at the service layer, ensuring that all database operations within a method are part of a single transaction. This prevents partial updates and maintains the integrity of the database.
* **Challenge 2**: Ensuring separation of concerns between service and DAO layers.
  + **Solution**: We ensured that the service layer only handles business logic, while the DAO layer is responsible for database interactions. By using dependency injection (@Autowired) in the service layer to access DAO methods, we adhered to the principle of single responsibility.
* **Challenge 3**: Handling exceptions properly between layers.
  + **Solution**: In the service layer, we implemented exception handling to capture any errors thrown by the DAO layer and return meaningful error messages to the client. This ensures that the application handles failures gracefully and provides useful feedback to the user.
* **Challenge 4**: Testing DAO and service layer methods independently.
  + **Solution**: We used unit testing frameworks like JUnit to test the DAO layer methods with mock data and the service layer methods by simulating database interactions. This ensures that both layers function correctly in isolation before integration.

**4. Optimizations and Improvements Made During DAO Integration**

* **Optimization 1**: Improved performance with batch processing.
  + During the integration of the DAO layer, we optimized the save and update operations by implementing batch processing, which reduces the number of database calls for large sets of data.
* **Optimization 2**: Utilization of JpaRepository methods.
  + We leveraged Spring Data JPA’s JpaRepository for the Product entity to reduce the amount of custom code needed for CRUD operations. This significantly improved the readability and maintainability of the codebase.
* **Optimization 3**: Enhanced error handling.
  + We improved the error handling mechanisms across the DAO and service layers by introducing custom exception classes. This allowed us to provide more specific error messages and ensured that exceptions were handled appropriately.
* **Improvement 1**: Code refactoring for better readability.
  + We refactored the code by modularizing the methods and adding clear comments to the codebase, improving maintainability and readability.
* **Improvement 2**: Added meaningful logging.
  + We integrated logging at key points in the DAO and service layers to track the application’s behavior, which aids in debugging and monitoring.