Step 1: Update pom.xml with JDBC and Oracle Driver Dependencies

First, we need to add the necessary dependencies for Spring JDBC and the Oracle Database driver.

1. **Open pom.xml:**
   - Locate your project's pom.xml file in the root directory.
2. **Add Dependencies:**
   - Add spring-boot-starter-jdbc (which includes HikariCP by default).
   - Add the Oracle JDBC driver (ojdbc11).

```
<!-- pom.xml -->
<dependencies>
    <!-- Existing dependencies (web, aop, etc.) -->

    <!-- Spring Boot JDBC Starter (includes HikariCP) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>

    <!-- Oracle JDBC Driver for Java 11+ / Java 21 -->
    <!-- IMPORTANT: This driver is often NOT in public Maven repos. -->
    <!-- You might need to download it from Oracle and install it manually to your local Maven repo. -->
    <!-- For example: mvn install:install-file -Dfile=/path/to/ojdbc11.jar -DgroupId=com.oracle.database.jdbc -DartifactId=ojdbc11 -Dversion=23.4.0.24.05 -Dpackaging=jar -->
    <dependency>
        <groupId>com.oracle.database.jdbc</groupId>
        <artifactId>ojdbc11</artifactId>
        <version>23.4.0.24.05</version> <!-- Use the version you downloaded -->
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

3. **Reload Maven Project:**
   - Your IDE will prompt you to reload the project to download the new libraries. Confirm this. Or right click on pom.xml file -> maven -> sync project.

Step 2: Configure application.properties for Oracle DB

Now, we'll configure the connection details for your Oracle Database 23ai instance in application.properties. This is where you specify the database URL, credentials, and connection pool settings.

1. **Open src/main/resources/application.properties:**
2. **Add Oracle Database Configuration:**
    - Replace <VM_IP_ADDRESS> with the actual IP address of your Oracle Developer VM.
    - Replace XEPDB1 with your Oracle 23ai PDB's service name (this is the most common setup for modern Oracle VMs). If you are using an older Oracle XE with a SID, use jdbc:oracle:thin:@<VM_IP_ADDRESS>:1521:XE.
    - Replace your_oracle_user and your_oracle_password with your Oracle database credentials.

```
# src/main/resources/application.properties

# --- Oracle Database 23ai Configuration ---
# Connects to Oracle using the Thin Client and Service Name (common for PDBs)
spring.datasource.url=jdbc:oracle:thin:@<VM_IP_ADDRESS>:1521/XEPDB1
spring.datasource.username=your_oracle_user
spring.datasource.password=your_oracle_password
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

# --- HikariCP Connection Pool Configuration (Optional, but Recommended) ---
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=2
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.idle-timeout=600000
spring.datasource.hikari.max-lifetime=1800000
spring.datasource.hikari.pool-name=SpringBootOraclePool

# --- Database Initialization Settings ---
# Set to 'always' to run schema.sql and data.sql on startup for Oracle
spring.sql.init.mode=always
# Optional: Continue on error during script execution (useful for development)
spring.sql.init.continue-on-error=true
```

- **Explanation:**
  - spring.datasource.url: The connection string for your Oracle database.
  - spring.datasource.username, spring.datasource.password: Your database login credentials.
  - spring.datasource.driver-class-name: Explicitly defines the Oracle JDBC driver.
  - spring.datasource.hikari.*: These properties configure HikariCP, Spring Boot's default connection pool, for optimal performance.
  - spring.sql.init.mode=always: This property tells Spring Boot to always execute schema.sql and data.sql files found in src/main/resources on application startup, even for non-embedded databases like Oracle. This is crucial for automatic table creation and initial data loading.

Step 3: Prepare Database Initialization Scripts (schema.sql, data.sql)

Spring Boot can automatically run SQL scripts on startup to create your database schema and populate it with initial data.

1. **Create src/main/resources/schema.sql:**
   - Create a new file named schema.sql directly inside your src/main/resources folder.
   - Add the SQL to create your PRODUCTS table. We'll use GENERATED BY DEFAULT ON NULL AS IDENTITY for auto-incrementing IDs, which is the modern Oracle 12c+ way.

```
-- src/main/resources/schema.sql

-- Drop table if it exists to allow re-running for development
-- This is useful during development to ensure a clean state on each restart
DROP TABLE PRODUCTS CASCADE CONSTRAINTS;

-- Create the PRODUCTS table with an auto-incrementing ID
CREATE TABLE PRODUCTS (
    ID NUMBER(10) GENERATED BY DEFAULT ON NULL AS IDENTITY
```

```
PRIMARY KEY,
    NAME VARCHAR2(255) NOT NULL,
    PRICE NUMBER(10, 2) NOT NULL
);
```

2. **Create src/main/resources/data.sql:**
    - Create a new file named data.sql directly inside your src/main/resources folder.
    - Add some initial data to your PRODUCTS table.

```
-- src/main/resources/data.sql

INSERT INTO PRODUCTS (NAME, PRICE) VALUES ('Laptop', 1200.00);
INSERT INTO PRODUCTS (NAME, PRICE) VALUES ('Smartphone', 750.00);
INSERT INTO PRODUCTS (NAME, PRICE) VALUES ('Monitor', 300.00);
```

    - **Explanation:** When your Spring Boot application starts, it will first connect to the Oracle database, then execute schema.sql to create the PRODUCTS table, and finally execute data.sql to insert these initial products.

Step 4: Create a ProductRepository using JdbcTemplate

Now, we'll create a dedicated repository layer that interacts with the database using Spring's JdbcTemplate. This replaces the in-memory list from your previous activities.

1. **Create a new package:** Inside src/main/java/com/example/demo, create a new package named repository.
2. **Create ProductRepository.java:** Inside the repository package, create a new Java class named ProductRepository.java and add the following code:

```
// src/main/java/com/example/demo/repository/ProductRepository.java
package com.example.demo.repository;


import com.example.demo.model.Product; // Import our Product model
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate; // Spring's simplified JDBC client
import org.springframework.jdbc.core.RowMapper; // To map ResultSet rows to Product objects
```

```
import org.springframework.stereotype.Repository; // Marks this as a repository component

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.Optional;

@Repository // Indicates that this class is a data repository
public class ProductRepository {

    private final JdbcTemplate jdbcTemplate; // Injected by Spring

    @Autowired // Spring automatically injects the auto-configured JdbcTemplate
    public ProductRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // RowMapper: A helper to convert each row of a ResultSet into a Product object
    private RowMapper<Product> productRowMapper = new RowMapper<Product>() {
        @Override
        public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
            Product product = new Product();
            product.setId(rs.getLong("ID"));
            product.setName(rs.getString("NAME"));
            product.setPrice(rs.getDouble("PRICE"));
            return product;
        }
    };
```

```
    // --- CRUD Operations using JdbcTemplate ---

    // CREATE: Inserts a new product into the database
    public Product save(Product product) {
        // For Oracle's GENERATED BY DEFAULT ON NULL AS IDENTITY, omit ID
in INSERT.
        String sql = "INSERT INTO PRODUCTS (NAME, PRICE) VALUES (?, ?)";
        jdbcTemplate.update(sql, product.getName(), product.getPrice());

        // Note: For simplicity, we are not retrieving the auto-generated ID here.
        // In a real application, you might use SimpleJdbcInsert or query for the last ID
        // if you need the generated ID immediately after insertion.
        return product;
    }

    // READ All: Retrieves all products from the database
    public List<Product> findAll() {
        String sql = "SELECT ID, NAME, PRICE FROM PRODUCTS";
        return jdbcTemplate.query(sql, productRowMapper);
    }

    // READ By ID: Retrieves a single product by its ID
    public Optional<Product> findById(Long id) {
        String sql = "SELECT ID, NAME, PRICE FROM PRODUCTS WHERE ID = ?";
        try {
            // queryForObject expects exactly one result. If none, it throws
EmptyResultDataAccessException.
            Product product = jdbcTemplate.queryForObject(sql, productRowMapper,
id);
            return Optional.ofNullable(product); // Wrap in Optional
        } catch (org.springframework.dao.EmptyResultDataAccessException e) {
```

```
            return Optional.empty(); // Return empty Optional if product not found
        }
    }

    // UPDATE: Updates an existing product in the database
    public int update(Product product) {
        String sql = "UPDATE PRODUCTS SET NAME = ?, PRICE = ? WHERE ID =
?";
        return jdbcTemplate.update(sql, product.getName(), product.getPrice(),
product.getId());
    }

    // DELETE: Deletes a product by its ID
    public int deleteById(Long id) {
        String sql = "DELETE FROM PRODUCTS WHERE ID = ?";
        return jdbcTemplate.update(sql, id);
    }
}
```

- **Explanation:**
  - @Repository: A stereotype annotation that marks this class as a Spring repository, enabling component scanning and exception translation.
  - JdbcTemplate: Spring's core class for simplifying JDBC operations. Spring Boot auto-configures it, so we just Autowired it.
  - RowMapper: An interface used to map each row of a ResultSet to a Java object (Product in this case). This avoids manual rs.getString() and rs.getLong() calls for each column.
  - jdbcTemplate.update(): Used for DML operations (INSERT, UPDATE, DELETE).
  - jdbcTemplate.query(): Used for SELECT queries that return multiple rows.
  - jdbcTemplate.queryForObject(): Used for SELECT queries that are

expected to return a single object.

Step 5: Modify ProductController to use ProductRepository

Now, we'll update your ProductController to use the new ProductRepository for data access, completely removing the in-memory list.

1. **Open src/main/java/com/example/demo/controller/ProductController.java:**
2. **Make the following changes:**
   - **Remove** the in-memory List<Product> products and AtomicLong counter.
   - **Remove** the ProductService injection and the AOP demo endpoints (as they are not relevant to database CRUD).
   - **Inject ProductRepository** instead.
   - **Update all CRUD methods** (createProduct, getAllProducts, getProductById, updateProduct, deleteProduct) to call the corresponding methods on productRepository.

```
// src/main/java/com/example/demo/controller/ProductController.java
package com.example.demo.controller;

import com.example.demo.model.Product;
import com.example.demo.repository.ProductRepository; // Import the new repository
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductRepository productRepository; // Inject the repository

    @Autowired // Spring injects the ProductRepository bean
    public ProductController(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    // CREATE Product (POST /api/products)
```

```
    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product
product) {
        Product savedProduct = productRepository.save(product); // Delegate to
repository
        return new ResponseEntity<>(savedProduct, HttpStatus.CREATED);
    }

    // READ All Products (GET /api/products)
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productRepository.findAll(); // Delegate to
repository
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // READ Product by ID (GET /api/products/{id})
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        return productRepository.findById(id) // Delegate to repository
                .map(product -> new ResponseEntity<>(product, HttpStatus.OK))
                .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    // UPDATE Product (PUT /api/products/{id})
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        // Ensure the ID in the path matches the ID in the body for update
        productDetails.setId(id);
        int updatedRows = productRepository.update(productDetails); // Delegate
to repository
        if (updatedRows > 0) {
            return new ResponseEntity<>(productDetails, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Product not
found to update
        }
    }

    // DELETE Product (DELETE /api/products/{id})
    @DeleteMapping("/{id}")
    public ResponseEntity<HttpStatus> deleteProduct(@PathVariable Long id) {
        int deletedRows = productRepository.deleteById(id); // Delegate to
repository
        if (deletedRows > 0) {
```

```
            return new ResponseEntity<>(HttpStatus.NO_CONTENT); // Successful
deletion
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Product not
found to delete
        }
    }

    // Removed AOP demo endpoints and ProductService injection as they are
not
    // directly related to database CRUD operations in this context.
}
```

- **Explanation:** The controller now acts as a thin layer, receiving HTTP requests and delegating the actual data persistence logic to the ProductRepository. This adheres to a layered architecture, improving maintainability.

Step 6: Run and Test the Application

Now, run your Spring Boot application and test the CRUD operations. This time, the data will be persisted in your Oracle Database 23ai!

1. **Run the application:**
   - **From IDE:** Right-click on your main application class (e.g., DemoApplication.java) and select "Run 'DemoApplication.main()'".
   - **From Command Line (Maven):** Open your terminal, navigate to the root directory of your project, and run:
     mvn spring-boot:run
   - **Observe Console Output:** You should see logs indicating HikariCP initializing the connection pool and Spring Boot executing schema.sql and data.sql. Look for messages like "Executing SQL script from class path resource..."
2. **Verify Database Initialization (Optional, using SQL Developer/SQLcl in VM):**
   - Connect to your Oracle DB 23ai instance using SQL Developer or SQLcl.
   - Run SELECT * FROM PRODUCTS;
   - You should see the three products ("Laptop", "Smartphone", "Monitor") inserted by data.sql.

3. **Test CRUD Operations via API (using Postman or curl):**
    - **CREATE a New Product (POST):**
        - **Method:** POST
        - **URL:** http://localhost:8080/api/products
        - **Headers:** Content-Type: application/json
        - **Body (raw, JSON):** {"name": "Smart Speaker", "price": 99.99}
        - **Expected Response:** 201 Created with the new product's details (ID will be auto-generated by Oracle).
    - **READ All Products (GET):**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products
        - **Expected Response:** 200 OK with a list including the initial products and the one you just created.
    - **READ a Specific Product by ID (GET):**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/{ID_OF_CREATED_PRODUCT} (e.g., http://localhost:8080/api/products/4 if 4 was the generated ID)
        - **Expected Response:** 200 OK with the specific product's details.
    - **UPDATE a Product (PUT):**
        - **Method:** PUT
        - **URL:** http://localhost:8080/api/products/{ID_TO_UPDATE} (e.g., http://localhost:8080/api/products/1)
        - **Headers:** Content-Type: application/json
        - **Body (raw, JSON):** {"name": "Updated Laptop Pro", "price": 1600.00}
        - **Expected Response:** 200 OK with the updated product details.
    - **DELETE a Product (DELETE):**
        - **Method:** DELETE
        - **URL:** http://localhost:8080/api/products/{ID_TO_DELETE} (e.g., http://localhost:8080/api/products/2)
        - **Expected Response:** 204 No Content.
    - **Verify Persistence:** Restart your Spring Boot application. Then, perform a GET

/api/products request. You should still see all the products you created, updated, and deleted (except the deleted ones) because they are now stored in the Oracle database!

You have successfully connected your Spring Boot application to an Oracle Database 23ai instance, configured JDBC, utilized connection pooling, initialized your database schema and data, and performed full CRUD operations using JdbcTemplate. This activity demonstrates how Spring Boot simplifies robust database integration, enabling your applications to manage persistent data effectively.

Activity 9.1: Spring Boot JPA (Hibernate, Annotations, CrudRepository)

This guide will walk you through migrating your existing Spring Boot application's data access layer from JDBC (JdbcTemplate) to JPA using Hibernate and Spring Data JPA. You will learn to define entities with annotations, use JpaRepository for effortless CRUD operations, and implement custom queries.

Objective

By the end of this activity, you will have a Spring Boot application that:

- Uses Spring Data JPA with Hibernate as its ORM solution.
- Maps Java Product objects to a database table using JPA annotations.
- Performs all CRUD operations and custom queries via a JpaRepository interface.
- Demonstrates the power of Object-Relational Mapping compared to raw JDBC.

Prerequisites

- **Completed the "Spring Boot Database & JDBC with Oracle" activity.** You should have a working Spring Boot project (e.g., demo or product-api) with Product model, ProductRepository (JDBC-based), ProductController, and application.properties configured for Oracle DB.
- **Oracle Database 23ai running in your VM.** Ensure it's accessible from your Spring Boot application.
- **Java Development Kit (JDK) 21.**
- **Apache Maven 3.6+** (or Gradle).
- An Integrated Development Environment (IDE) like **IntelliJ IDEA**, **Eclipse (with STS)**, or **VS Code (with Spring Boot extensions)**.
- An API testing tool like **Postman** or **curl**.

Step 1: Update pom.xml for Spring Data JPA

First, we need to replace the spring-boot-starter-jdbc dependency with spring-boot-starter-data-jpa. This starter includes Hibernate (the default JPA provider) and Spring