Data JPA. We'll keep the Oracle JDBC driver.

1. **Open pom.xml:**
   - Locate your project's pom.xml file in the root directory.
2. **Replace spring-boot-starter-jdbc with spring-boot-starter-data-jpa:**
   - **Find and remove** the following dependency:

     ```
     <!-- REMOVE THIS -->
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-jdbc</artifactId>
     </dependency>
     ```

   - **Add** the spring-boot-starter-data-jpa dependency:

     ```
     <!-- Add this dependency for Spring Data JPA (includes Hibernate) -->
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-data-jpa</artifactId>
     </dependency>

     <!-- Keep your Oracle JDBC Driver dependency -->
     <dependency>
         <groupId>com.oracle.database.jdbc</groupId>
         <artifactId>ojdbc11</artifactId>
         <version>23.4.0.24.05</version> <!-- Ensure this matches your downloaded version -->
         <scope>runtime</scope>
     </dependency>
     ```

3. **Reload Maven Project:**
   - Your IDE will prompt you to reload the project. Confirm this to download the new libraries.

Step 2: Configure application.properties for JPA/Hibernate

Now, we'll adjust your application.properties to enable Hibernate's DDL (Data Definition Language) capabilities and show SQL queries, replacing the spring.sql.init.mode for schema management.

1. **Open src/main/resources/application.properties:**
2. **Modify Database Configuration:**
   - **Keep** your existing Oracle connection properties (spring.datasource.url, username, password, driver-class-name).
   - **Remove** spring.sql.init.mode=always and spring.sql.init.continue-on-error=true as Hibernate will now manage the schema based on your entities.
   - **Add JPA/Hibernate specific properties:**

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

# --- Existing Oracle Database Configuration (keep these) ---
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

# --- Existing HikariCP Connection Pool Configuration (keep these) ---
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

# --- NEW: JPA/Hibernate Properties ---
# ddl-auto: Controls Hibernate's schema generation
# 'update': Updates the schema based on entities (good for dev)
# 'create-drop': Creates schema on startup, drops on shutdown (good for tests)
# 'none': No DDL operations by Hibernate (RECOMMENDED for production
```

```
with Flyway/Liquibase)
spring.jpa.hibernate.ddl-auto=update

# Show SQL queries generated by Hibernate in the console
spring.jpa.show-sql=true
# Format the SQL queries for better readability
spring.jpa.properties.hibernate.format_sql=true
# Specify the database dialect for Hibernate to optimize SQL generation for
Oracle
spring.jpa.database-platform=org.hibernate.dialect.OracleDialect
```

- **Explanation:**
  - spring.jpa.hibernate.ddl-auto=update: This tells Hibernate to compare your entity classes with the database schema and update the schema if necessary. For development, this is convenient. For production, you'd typically set this to none and use dedicated migration tools like Flyway or Liquibase.
  - spring.jpa.show-sql=true and spring.jpa.properties.hibernate.format_sql=true: These are extremely useful for debugging, allowing you to see the SQL queries Hibernate generates.
  - spring.jpa.database-platform=org.hibernate.dialect.OracleDialect: Ensures Hibernate generates SQL optimized for Oracle.

Step 3: Modify Product.java to be a JPA Entity

Your Product class needs to be annotated with JPA annotations to tell Hibernate how to map it to a database table.

1. **Open src/main/java/com/example/demo/model/Product.java:**
2. **Add JPA Annotations:**
   - Use jakarta.persistence.* imports for Spring Boot 3+ (which uses Jakarta EE 9+).
   - Annotate the class with @Entity and @Table.

- Annotate the id field with @Id and @GeneratedValue.
- Annotate other fields with @Column.

```
// src/main/java/com/example/demo/model/Product.java
package com.example.demo.model;

import jakarta.persistence.Column; // For column mapping
import jakarta.persistence.Entity; // To mark as a JPA entity
import jakarta.persistence.GeneratedValue; // For ID generation strategy
import jakarta.persistence.GenerationType; // For ID generation strategy types
import jakarta.persistence.Id; // To mark primary key
import jakarta.persistence.Table; // To specify table name

@Entity // Marks this class as a JPA entity
@Table(name = "PRODUCTS") // Maps this entity to the "PRODUCTS" table in the database
public class Product {
    @Id // Designates 'id' as the primary key
    // For Oracle 12c+ with GENERATED BY DEFAULT ON NULL AS IDENTITY, GenerationType.IDENTITY works.
    // For older Oracle or sequence-based IDs, you would use GenerationType.SEQUENCE
    // with @SequenceGenerator.
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "NAME", nullable = false, length = 255) // Maps to 'NAME' column, not null, max length 255
    private String name;

    @Column(name = "PRICE", nullable = false) // Maps to 'PRICE' column, not null
```

```
    private double price;

    // Constructors (keep existing ones)
    public Product() {}

    public Product(String name, double price) { // Constructor without ID for new products
        this.name = name;
        this.price = price;
    }

    // Getters and Setters (keep existing ones)
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    @Override
    public String toString() {
        return "Product{id=" + id + ", name='" + name + "', price=" + price + '}';
    }
}
```

- **Explanation:**
  - @Entity: Tells Hibernate that this class is a persistent entity.
  - @Table(name = "PRODUCTS"): Specifies the database table name. If omitted, Hibernate defaults to the class name (Product).
  - @Id: Marks the id field as the primary key.
  - @GeneratedValue(strategy = GenerationType.IDENTITY): Configures the

primary key generation strategy. For Oracle 12c+ (including 23ai) using GENERATED BY DEFAULT ON NULL AS IDENTITY, IDENTITY works correctly.

- @Column: Maps object fields to database columns. nullable=false enforces a NOT NULL constraint, and length sets the column size for strings.

Step 4: Create ProductRepository using JpaRepository

Now, we'll create a new ProductRepository interface that extends JpaRepository. This will automatically provide all standard CRUD operations and allow for custom query methods.

1. **Open src/main/java/com/example/demo/repository/ProductRepository.java:**
   - **Delete the existing ProductRepository.java content** (the JdbcTemplate based one).
2. **Add the new JpaRepository-based interface:**

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

import com.example.demo.model.Product; // Import our Product entity
import org.springframework.data.jpa.repository.JpaRepository; // Import JpaRepository
import org.springframework.data.jpa.repository.Modifying; // For DML queries
import org.springframework.data.jpa.repository.Query; // For custom queries
import org.springframework.data.repository.query.Param; // For named parameters in queries
import org.springframework.stereotype.Repository; // Optional, but good practice
import org.springframework.transaction.annotation.Transactional; // For transactional DML queries

import java.util.List;

@Repository // Marks this interface as a Spring Data JPA repository
```

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // JpaRepository provides out-of-the-box implementations for:
    // save(), findById(), findAll(), deleteById(), existsById(), count(), etc.
    // It also includes methods for pagination and sorting.

    // --- Custom Query Methods (Derived from Method Names) ---
    // Spring Data JPA automatically generates SQL based on the method name.
    List<Product> findByNameContainingIgnoreCase(String name); // SELECT * FROM PRODUCTS WHERE LOWER(NAME) LIKE LOWER('%name%')
    List<Product> findByPriceGreaterThan(double price); // SELECT * FROM PRODUCTS WHERE PRICE > price

    // --- Custom JPQL Query ---
    // @Query allows you to write custom queries using JPQL (Java Persistence Query Language)
    // JPQL operates on entities (Product p) not directly on tables/columns.
    @Query("SELECT p FROM Product p WHERE p.name LIKE %:name% AND p.price > :minPrice")
    List<Product> findProductsByNameAndMinPrice(@Param("name") String name, @Param("minPrice") double minPrice);

    // --- Custom Native SQL Query ---
    // @Query with nativeQuery = true allows you to write raw SQL.
    // This is less portable but useful for database-specific features.
    @Query(value = "SELECT * FROM PRODUCTS WHERE PRICE < ?1", nativeQuery = true)
    List<Product> findProductsByPriceLessThanNative(double maxPrice);

    // --- Custom DML Query (UPDATE) ---
    // @Modifying is required for DML operations (UPDATE, DELETE) with @Query.
    // @Transactional is crucial to ensure the operation is committed.
```

```
    @Modifying
    @Transactional
    @Query("UPDATE Product p SET p.name = :newName WHERE p.id = :id")
    int updateProductName(@Param("id") Long id, @Param("newName") String
newName);
}
```

- **Explanation:**
  - JpaRepository<Product, Long>: Inherits all CRUD, pagination, and sorting methods for the Product entity with Long as its primary key type.
  - findByNameContainingIgnoreCase, findByPriceGreaterThan: Examples of **query derivation**. Spring Data JPA parses these method names and generates the appropriate SQL queries automatically.
  - @Query("SELECT p FROM Product p WHERE ..."): Demonstrates a **JPQL query**. p refers to the Product entity.
  - @Query(value = "SELECT * FROM PRODUCTS WHERE ...", nativeQuery = true): Demonstrates a **native SQL query**. PRODUCTS refers to the actual table name.
  - @Modifying and @Transactional: Essential for custom DML operations (UPDATE, DELETE) defined with @Query.

Step 5: Modify ProductController to use JpaRepository

Finally, we'll update your ProductController to use the JpaRepository methods. This will simplify your controller code significantly.

1. **Open src/main/java/com/example/demo/controller/ProductController.java:**
2. **Make the following changes:**
   - **Replace** the ProductRepository injection from the JDBC version with the new JpaRepository-based one.
   - **Update all CRUD methods** to use the simpler JpaRepository methods.
   - **Add new endpoints** to demonstrate the custom query methods from JpaRepository.

```
// 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
JpaRepository
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.ArrayList; // Used for converting Iterable to List
import java.util.List;
import java.util.Optional;

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

    private final ProductRepository productRepository; // Inject the JpaRepository

    @Autowired
    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); // save()
handles both create and update
        return new ResponseEntity<>(savedProduct, HttpStatus.CREATED);
    }

    // READ All Products (GET /api/products)
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = new ArrayList<>();
        productRepository.findAll().forEach(products::add); // findAll() returns
Iterable, convert to List
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // READ Product by ID (GET /api/products/{id})
    @GetMapping("/{id}")
```

```
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productRepository.findById(id); // findById()
returns Optional
        return product.map(value -> new ResponseEntity<>(value, HttpStatus.OK))
                .orElseGet(() -> new
ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    // UPDATE Product (PUT /api/products/{id})
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        Optional<Product> productOptional = productRepository.findById(id); //
First, find the existing product
        if (productOptional.isPresent()) {
            Product existingProduct = productOptional.get();
            // Update the fields from the request body
            existingProduct.setName(productDetails.getName());
            existingProduct.setPrice(productDetails.getPrice());
            Product updatedProduct = productRepository.save(existingProduct); //
save() updates if ID exists
            return new ResponseEntity<>(updatedProduct, 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) {
        if (productRepository.existsById(id)) { // Check if product exists before
deleting
            productRepository.deleteById(id);
            return new ResponseEntity<>(HttpStatus.NO_CONTENT); // Successful
deletion
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Product not
found to delete
        }
    }

    // --- New Endpoints to Demonstrate Custom JPA Queries ---

    // Custom Query Endpoint: Derived method
(findByNameContainingIgnoreCase)
```

```
    // GET /api/products/search/name?name=laptop
    @GetMapping("/search/name")
    public ResponseEntity<List<Product>>
searchProductsByName(@RequestParam String name) {
        List<Product> products =
productRepository.findByNameContainingIgnoreCase(name);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // Custom Query Endpoint: JPQL Query (findProductsByNameAndMinPrice)
    // GET /api/products/search/jpql?name=smart&minPrice=500
    @GetMapping("/search/jpql")
    public ResponseEntity<List<Product>> searchProductsByJpql(
        @RequestParam String name, @RequestParam double minPrice) {
        List<Product> products =
productRepository.findProductsByNameAndMinPrice(name, minPrice);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // Custom Query Endpoint: Native SQL Query
(findProductsByPriceLessThanNative)
    // GET /api/products/search/native?maxPrice=400
    @GetMapping("/search/native")
    public ResponseEntity<List<Product>>
searchProductsByNative(@RequestParam double maxPrice) {
        List<Product> products =
productRepository.findProductsByPriceLessThanNative(maxPrice);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // Custom DML Query Endpoint: Update Product Name
(updateProductName)
    // PUT /api/products/update-name/{id}?newName=NewNameValue
    @PutMapping("/update-name/{id}")
    public ResponseEntity<String> updateProductName(@PathVariable Long id,
@RequestParam String newName) {
        int updatedRows = productRepository.updateProductName(id, newName);
        if (updatedRows > 0) {
            return new ResponseEntity<>("Product name updated successfully for
ID: " + id, HttpStatus.OK);
        } else {
            return new ResponseEntity<>("Product not found for ID: " + id,
HttpStatus.NOT_FOUND);
        }
    }
}
```

- **Explanation:** The controller now interacts solely with the JpaRepository, making the code cleaner and more object-oriented. Spring Data JPA handles all the underlying SQL generation and object mapping.

Step 6: Run and Test the Application

Now, run your Spring Boot application. Hibernate will automatically create/update the PRODUCTS table based on your Product entity, and you can test the new JPA-based CRUD and custom query operations.

1. **Ensure src/main/resources/schema.sql and data.sql are removed or renamed.**
   - With spring.jpa.hibernate.ddl-auto=update, Hibernate will manage the schema. If schema.sql is present and spring.sql.init.mode=always is still active, you might get conflicts. It's best to remove or rename them for this activity.
2. **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 Hibernate logs, including the SQL queries it generates for table creation/update (if ddl-auto=update and the table doesn't exist or needs changes).
3. **Verify Database Schema (Optional, using SQL Developer/SQLcl in VM):**
   - Connect to your Oracle DB 23ai instance.
   - Run DESCRIBE PRODUCTS; to confirm the table structure matches your Product entity.
   - Run SELECT * FROM PRODUCTS; initially it might be empty if ddl-auto created a fresh table.
4. **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": "Wireless Earbuds", "price": 129.99}
        - **Expected Response:** 201 Created with the new product's details (ID will be auto-generated by Oracle and returned by JPA).
    - **READ All Products (GET):**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products
        - **Expected Response:** 200 OK with a list including the product(s) 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/1)
        - **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": "Premium Wireless Earbuds", "price": 149.99}
        - **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/1)
        - **Expected Response:** 204 No Content.

    Test Custom Query Endpoints:

    - **Search by Name (Derived Query):**
        - **Method:** GET

        - **URL:** http://localhost:8080/api/products/search/name?name=earbuds
        - **Expected Response:** 200 OK with products containing "earbuds" in their name (case-insensitive).
    - **Search by Name and Min Price (JPQL Query):**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/search/jpql?name=wireless&minPrice=100
        - **Expected Response:** 200 OK with products matching criteria.
    - **Search by Max Price (Native SQL Query):**
        - **Method:** GET
        - **URL:** http://localhost:8080/api/products/search/native?maxPrice=150
        - **Expected Response:** 200 OK with products cheaper than 150.
    - **Update Product Name (Custom DML Query):**
        - **Method:** PUT
        - **URL:** http://localhost:8080/api/products/update-name/{ID_TO_UPDATE}?newName=UpdatedProductName (e.g., http://localhost:8080/api/products/2?newName=NewMonitorName)
        - **Expected Response:** 200 OK with success message. Verify with GET /api/products/2.

You have successfully migrated your Spring Boot application from JDBC to JPA, leveraging Hibernate and Spring Data JPA. You've learned how to:

- Configure JPA in application.properties.
- Annotate your Product class to become a JPA entity.
- Define a JpaRepository interface for automatic CRUD operations.
- Implement custom queries using method name derivation, JPQL, and native SQL.

This transition significantly reduces boilerplate code and allows you to work with database entities in a more object-oriented manner, improving productivity and maintainability.

Activity 10.1: Spring Boot Caching

This guide will walk you through implementing caching in your existing Spring Boot application, which currently uses Spring Data JPA with Oracle Database. You will add caching capabilities to your service layer to reduce database load and improve response times for frequently accessed data.

Objective

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

- Utilizes Spring's caching abstraction.
- Implements @Cacheable, @CachePut, and @CacheEvict annotations.
- Reduces database calls for cached data.
- Demonstrates the benefits of caching through API invocations.

Prerequisites

- **Completed the "Spring Boot JPA (Hibernate, Annotations, CrudRepository)" activity.** You should have a working Spring Boot project with Product entity, JpaRepository-based ProductRepository, and ProductController configured to interact with your Oracle Database 23ai.
- **Oracle Database 23ai running in your VM.** Ensure it's accessible and contains the PRODUCTS table (Hibernate's ddl-auto=update should handle this on startup if the table doesn't exist).
- **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: Add Spring Boot Caching Dependency

First, we need to add the spring-boot-starter-cache dependency to your project's pom.xml. This starter brings in Spring's caching abstraction and auto-configures a