5. **Unzip and Open:** Unzip the downloaded file and open the project in your favorite IDE (IntelliJ IDEA, VS Code, Eclipse, etc.).

Step 2: Create the Product Entity

The Product entity will represent a row in our database table.

1. **Create Package:** Inside src/main/java/com/example/springdata, create a new package named entity.
2. **Create Product.java:** Inside the entity package, create a new Java class named Product.java and add the following code:

```
// src/main/java/com/example/springdata/entity/Product.java
package com.example.springdata.entity;

import jakarta.persistence.Entity; // JPA annotation to mark this class as an entity
import jakarta.persistence.GeneratedValue; // For automatic ID generation
import jakarta.persistence.GenerationType; // Strategy for ID generation
import jakarta.persistence.Id; // Marks the primary key field
import lombok.AllArgsConstructor; // Lombok: Generates a constructor with all fields
import lombok.Data; // Lombok: Generates getters, setters, toString, equals, hashCode
import lombok.NoArgsConstructor; // Lombok: Generates a no-argument constructor

@Entity // Marks this class as a JPA entity, mapping it to a database table
@Data // Lombok annotation to auto-generate getters, setters, toString(), equals(), and hashCode()
@NoArgsConstructor // Lombok annotation to auto-generate a no-argument constructor
@AllArgsConstructor // Lombok annotation to auto-generate a constructor with all fields
public class Product {
    @Id // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Configures ID
generation strategy (auto-increment)
    private Long id;
    private String name;
    private String description;
    private double price;
    private int quantity;
}
```

- **Explanation:**
    - @Entity: Tells JPA that this class is a database entity.
    - @Id: Marks id as the primary key.
    - @GeneratedValue(strategy = GenerationType.IDENTITY): Configures the database to automatically generate the id for new products.
    - @Data, @NoArgsConstructor, @AllArgsConstructor: Lombok annotations that drastically reduce boilerplate code by generating constructors, getters, setters, toString(), equals(), and hashCode() methods automatically.

Step 3: Create the ProductRepository Interface

This interface will extend Spring Data JPA's JpaRepository to provide automatic CRUD operations and derived query methods.

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

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

import com.example.springdata.entity.Product; // Import the Product entity
import org.springframework.data.jpa.repository.JpaRepository; // Spring Data JPA's core repository interface
import org.springframework.data.jpa.repository.Query; // Import for @Query annotation
import org.springframework.data.repository.query.Param; // Import for @Param annotation
import org.springframework.stereotype.Repository; // Optional: for clarity, marks this as a repository component

import java.util.List;
import java.util.Optional;
```

```
@Repository // Indicates that this interface is a "repository" component
// JpaRepository<Entity, ID_Type> provides methods for CRUD operations and
more
public interface ProductRepository extends JpaRepository<Product, Long> {

    // --- Derived Query Methods ---

    // Find products by name (Spring Data generates the query automatically)
    Optional<Product> findByName(String name);

    // Find products by price less than a given value
    List<Product> findByPriceLessThan(double price);

    // Find products by quantity greater than a given value
    List<Product> findByQuantityGreaterThan(int quantity);

    // Find products by name containing a specific string (case-insensitive)
    List<Product> findByNameContainingIgnoreCase(String keyword);

    // --- Custom Query with @Query (JPQL) ---

    // Find products with price between a min and max value
    // Using named parameters (:minPrice, :maxPrice) for better readability
    @Query("SELECT p FROM Product p WHERE p.price BETWEEN :minPrice
AND :maxPrice")
    List<Product> findProductsByPriceRange(@Param("minPrice") double minPrice,
@Param("maxPrice") double maxPrice);

    // Find products by description containing a keyword, ordered by name
    @Query("SELECT p FROM Product p WHERE p.description LIKE %:keyword%
ORDER BY p.name ASC")
    List<Product>
findProductsByDescriptionContainingOrderByPriceAsc(@Param("keyword") String
keyword);

    // Find products with quantity less than a value, and price greater than a value
    @Query("SELECT p FROM Product p WHERE p.quantity < ?1 AND p.price >
?2")
    List<Product> findProductsLowStockHighPrice(int maxQuantity, double
minPrice);
}
```

- **Explanation:**
  - @Repository: A stereotype annotation that makes this interface a Spring component.
  - JpaRepository<Product, Long>: By extending this, ProductRepository automatically inherits methods like save(), findById(), findAll(), deleteById(), etc., for the Product entity with Long as its ID type.
  - findByName(), findByPriceLessThan(), etc.: These are "derived query methods." Spring Data automatically generates the SQL queries based on the method names and the entity's properties. You don't write any SQL!
  - @Query: This new annotation allows you to write custom JPQL (Java Persistence Query Language) or native SQL queries directly.
  - SELECT p FROM Product p WHERE p.price BETWEEN :minPrice AND :maxPrice: This is a JPQL query. p refers to the Product entity. :minPrice and :maxPrice are named parameters, which are mapped to method arguments using @Param.
  - ?1, ?2: These are indexed parameters, referring to the first and second method arguments respectively.

Step 4: Create the ProductService

The service layer will encapsulate business logic and interact with the ProductRepository.

1. **Create Package:** Inside src/main/java/com.example/springdata, create a new package named service.
2. **Create ProductService.java:** Inside the service package, create a new Java class named ProductService.java and add the following code:

   ```
   // src/main/java/com/example/springdata/service/ProductService.java
   package com.example.springdata.service;

   import com.example.springdata.entity.Product; // Import Product entity
   import com.example.springdata.repository.ProductRepository; // Import ProductRepository
   import org.springframework.stereotype.Service; // Marks this class as a service component
   import org.springframework.transaction.annotation.Transactional; // For transaction management

   import java.util.List;
   import java.util.Optional;

   @Service // Indicates that this class is a "service" component
   public class ProductService {

       private final ProductRepository productRepository; // Inject the repository

       // Constructor-based dependency injection (recommended)
       public ProductService(ProductRepository productRepository) {
           this.productRepository = productRepository;
       }
   ```

```
// --- CRUD Operations ---

@Transactional // Ensures this method runs within a database transaction
public Product createProduct(Product product) {
    // Business logic can be added here before saving
    return productRepository.save(product); // Save the product
}

public Optional<Product> getProductById(Long id) {
    return productRepository.findById(id); // Find product by ID
}

public List<Product> getAllProducts() {
    return productRepository.findAll(); // Get all products
}

@Transactional
public Product updateProduct(Long id, Product updatedProduct) {
    // Find the existing product
    Optional<Product> existingProductOptional = productRepository.findById(id);

    if (existingProductOptional.isPresent()) {
        Product existingProduct = existingProductOptional.get();
        // Update fields
        existingProduct.setName(updatedProduct.getName());
        existingProduct.setDescription(updatedProduct.getDescription());
        existingProduct.setPrice(updatedProduct.getPrice());
        existingProduct.setQuantity(updatedProduct.getQuantity());
        // Save the updated product (Spring Data will update if ID exists)
        return productRepository.save(existingProduct);
    } else {
            throw new RuntimeException("Product not found with ID: " + id);
        }
    }

    @Transactional
    public void deleteProduct(Long id) {
        productRepository.deleteById(id); // Delete product by ID
    }

    // --- Using Derived Query Methods from Repository ---

    public Optional<Product> getProductByName(String name) {
        return productRepository.findByName(name);
    }

    public List<Product> getProductsCheaperThan(double price) {
        return productRepository.findByPriceLessThan(price);
    }

    public List<Product> getProductsInStockGreaterThan(int quantity) {
        return productRepository.findByQuantityGreaterThan(quantity);
    }

    public List<Product> searchProductsByName(String keyword) {
        return productRepository.findByNameContainingIgnoreCase(keyword);
    }

    // --- Using Custom Query Methods with @Query (JPQL) ---

    public List<Product> getProductsByPriceRange(double minPrice, double
maxPrice) {
        return productRepository.findProductsByPriceRange(minPrice, maxPrice);
    }

    public List<Product> getProductsByDescriptionContaining(String keyword) {
        return
productRepository.findProductsByDescriptionContainingOrderByPriceAsc(keyword)
;
    }

    public List<Product> getProductsLowStockHighPrice(int maxQuantity, double
minPrice) {
        return productRepository.findProductsLowStockHighPrice(maxQuantity,
minPrice);
    }
}
```

- **Explanation:**
    - @Service: Marks this class as a Spring service, making it eligible for component scanning and dependency injection.
    - @Autowired (implicitly by constructor injection): Spring automatically injects an instance of ProductRepository into ProductService.
    - @Transactional: Ensures that the methods are executed within a database transaction. If an unchecked exception occurs, the transaction will be rolled back.
    - New methods getProductsByPriceRange, getProductsByDescriptionContaining, and getProductsLowStockHighPrice are added to expose the custom JPQL queries defined in the repository.

Step 5: Configure H2 Database

We need to tell Spring Boot to use the H2 in-memory database and enable its console for easy viewing.

1. **Open application.properties:** Navigate to src/main/resources/application.properties.
2. **Add Configuration:** Add the following lines to the file:

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

# H2 Database Configuration
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

# JPA Properties
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update # Automatically create/update schema based
on entities
spring.jpa.show-sql=true # Show SQL queries in console
spring.jpa.properties.hibernate.format_sql=true # Format SQL for readability
```

   - **Explanation:**
     - spring.h2.console.enabled=true: Enables the H2 web console.
     - spring.h2.console.path=/h2-console: Sets the URL path for the console.
     - spring.datasource.url=jdbc:h2:mem:testdb: Configures H2 as an in-memory database named testdb. The data will be lost when the application stops.

    - spring.jpa.hibernate.ddl-auto=update: Hibernate (the JPA implementation) will automatically create or update the database schema based on your Product entity. For production, validate or none is often preferred.
    - spring.jpa.show-sql=true: Logs the SQL statements executed by JPA.

Step 6: Create a Simple REST Controller (Optional, but useful for testing)

This controller will expose REST endpoints to interact with our ProductService.

1. **Create Package:** Inside src/main/java/com.example/springdata, create a new package named controller.
2. **Create ProductController.java:** Inside the controller package, create a new Java class named ProductController.java and add the following code:

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

import com.example.springdata.entity.Product; // Import Product entity
import com.example.springdata.service.ProductService; // Import ProductService
import org.springframework.http.HttpStatus; // HTTP status codes
import org.springframework.http.ResponseEntity; // For building HTTP responses
import org.springframework.web.bind.annotation.*; // REST annotations

import java.util.List;
import java.util.Optional;

@RestController // Marks this class as a REST controller
@RequestMapping("/api/products") // Base URL path for all endpoints in this
controller
public class ProductController {

    private final ProductService productService; // Inject the service

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    // POST /api/products - Create a new product
    @PostMapping
    public ResponseEntity<Product> createProduct(@RequestBody Product
product) {
        Product createdProduct = productService.createProduct(product);
        return new ResponseEntity<>(createdProduct, HttpStatus.CREATED); //
Return 201 Created
    }

    // GET /api/products/{id} - Get a product by ID
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productService.getProductById(id);
        return product.map(p -> new ResponseEntity<>(p, HttpStatus.OK)) // Return
200 OK if found
                    .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND)); // Return
404 Not Found if not
    }

    // GET /api/products - Get all products
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productService.getAllProducts();
        return new ResponseEntity<>(products, HttpStatus.OK); // Return 200 OK
    }

    // PUT /api/products/{id} - Update an existing product
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product product) {
        try {
            Product updated = productService.updateProduct(id, product);
            return new ResponseEntity<>(updated, HttpStatus.OK); // Return 200 OK
        } catch (RuntimeException e) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Return 404 if
product not found
        }
    }

    // DELETE /api/products/{id} - Delete a product
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT); // Return 204 No
Content
    }

    // --- Endpoints using Derived Query Methods ---

    // GET /api/products/search/name?name={productName}
    @GetMapping("/search/name")
    public ResponseEntity<Product> getProductByName(@RequestParam String
name) {
        Optional<Product> product = productService.getProductByName(name);
        return product.map(p -> new ResponseEntity<>(p, HttpStatus.OK))
                     .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
    }

    // GET /api/products/search/cheaper-than?price={maxPrice}
    @GetMapping("/search/cheaper-than")
    public ResponseEntity<List<Product>>
getProductsCheaperThan(@RequestParam double price) {
        List<Product> products = productService.getProductsCheaperThan(price);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // GET /api/products/search/in-stock-greater-than?quantity={minQuantity}
    @GetMapping("/search/in-stock-greater-than")
    public ResponseEntity<List<Product>>
getProductsInStockGreaterThan(@RequestParam int quantity) {
        List<Product> products =
productService.getProductsInStockGreaterThan(quantity);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // GET /api/products/search/keyword?keyword={searchKeyword}
    @GetMapping("/search/keyword")
    public ResponseEntity<List<Product>>
searchProductsByName(@RequestParam String keyword) {
        List<Product> products = productService.searchProductsByName(keyword);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // --- New Endpoints using Custom @Query (JPQL) Methods ---

    // GET /api/products/custom/price-range?minPrice={min}&maxPrice={max}
    @GetMapping("/custom/price-range")
    public ResponseEntity<List<Product>> getProductsByPriceRange(
            @RequestParam double minPrice, @RequestParam double maxPrice) {
        List<Product> products = productService.getProductsByPriceRange(minPrice,
maxPrice);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // GET /api/products/custom/description-keyword?keyword={keyword}
    @GetMapping("/custom/description-keyword")
    public ResponseEntity<List<Product>> getProductsByDescriptionContaining(
            @RequestParam String keyword) {
        List<Product> products =
productService.getProductsByDescriptionContaining(keyword);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }

    // GET /api/products/custom/low-stock-high-
price?maxQuantity={maxQ}&minPrice={minP}
    @GetMapping("/custom/low-stock-high-price")
    public ResponseEntity<List<Product>> getProductsLowStockHighPrice(
            @RequestParam int maxQuantity, @RequestParam double minPrice) {
        List<Product> products =
productService.getProductsLowStockHighPrice(maxQuantity, minPrice);
        return new ResponseEntity<>(products, HttpStatus.OK);
    }
}
```

- **Explanation:**
    - @RestController: Combines @Controller and @ResponseBody, meaning methods return data directly (e.g., JSON) rather than view names.
    - @RequestMapping("/api/products"): Sets the base path for all endpoints in this controller.
    - @PostMapping, @GetMapping, @PutMapping, @DeleteMapping: Map HTTP methods to specific controller methods.
    - @RequestBody: Maps the HTTP request body (e.g., JSON) to a Java object (Product).
    - @PathVariable: Extracts a variable from the URL path (e.g., {id}).
    - @RequestParam: Extracts a parameter from the query string (e.g., ?name=value).
    - ResponseEntity: Provides full control over the HTTP response (status code, headers, body).
    - New endpoints under /api/products/custom/ are added to test the JPQL queries.

Step 7: Run the Application and Test

1. **Run Spring Boot Application:**
   - Locate the main application class: src/main/java/com/example/springdata/SpringDataDemoApplication.java.
   - Right-click on it and select "Run 'SpringDataDemoApplication.main()'" (or use your IDE's run button).
   - Look for "Started SpringDataDemoApplication" in the console output.
2. **Access H2 Console:**
   - Once the application is running, open your web browser and go to: http://localhost:8080/h2-console
   - **JDBC URL:** Ensure it's jdbc:h2:mem:testdb (matching your application.properties).
   - **User Name:** sa
   - **Password:** password
   - Click "Connect". You should see the H2 console, and if you execute SELECT * FROM PRODUCT;, you'll see an an empty table (or data if you've already added some).
3. **Test with a Tool (e.g., Postman, Insomnia, or curl):**
   - **Create Product (POST):**
     - **URL:** http://localhost:8080/api/products
     - **Method:** POST
     - **Headers:** Content-Type: application/json
     - **Body (Raw JSON):**

       ```
       {
         "name": "Laptop",
         "description": "High-performance laptop",
         "price": 1200.00,
         "quantity": 50
       }
       ```

    - Send the request. You should get a 201 Created response with the created product details (including the generated ID).
- **Create Another Product (POST):**

```
{
   "name": "Mouse",
   "description": "Wireless ergonomic mouse",
   "price": 25.50,
   "quantity": 200
}
```

- **Create a Third Product (POST):**

```
{
   "name": "Keyboard",
   "description": "Mechanical gaming keyboard",
   "price": 150.00,
   "quantity": 75
}
```

- **Get All Products (GET):**
    - **URL:** http://localhost:8080/api/products
    - **Method:** GET
    - Send the request. You should get a 200 OK response with a list of all products.
- **Get Product by ID (GET):** (Replace 1 with the actual ID from your POST response)
    - **URL:** http://localhost:8080/api/products/1
    - **Method:** GET
    - Send the request. You should get a 200 OK response with the specific product.

- **Update Product (PUT):** (Replace 1 with the actual ID)
  - **URL:** http://localhost:8080/api/products/1
  - **Method:** PUT
  - **Headers:** Content-Type: application/json
  - **Body (Raw JSON):**

```
{
   "id": 1,
   "name": "Gaming Laptop",
   "description": "Ultra-performance gaming laptop with RTX 4080",
   "price": 1800.00,
   "quantity": 45
}
```

  - Send the request. You should get a 200 OK response with the updated product.
- **Search Products by Name (GET - Derived Query):**
  - **URL:** http://localhost:8080/api/products/search/name?name=Mouse
  - **Method:** GET
  - Send the request. You should get the product named "Mouse".
- **Search Products Cheaper Than (GET - Derived Query):**
  - **URL:** http://localhost:8080/api/products/search/cheaper-than?price=100.00
  - **Method:** GET
  - Send the request. You should get products with a price less than $100.00.
- **Search Products In Stock Greater Than (GET - Derived Query):**
  - **URL:** http://localhost:8080/api/products/search/in-stock-greater-than?quantity=60
  - **Method:** GET
  - Send the request. You should get products with quantity greater than 60.
- **Search Products by Keyword (GET - Derived Query):**
  - **URL:** http://localhost:8080/api/products/search/keyword?keyword=gaming

        - **Method:** GET
        - Send the request. You should get products whose name or description contains "gaming" (case-insensitive).
    - **Delete Product (DELETE):** (Replace 1 with the actual ID)
        - **URL:** http://localhost:8080/api/products/1
        - **Method:** DELETE
        - Send the request. You should get a 204 No Content response.
        - Verify by trying to GET the product by ID again, which should now return 404 Not Found.

Step 8: Test Custom Queries with @Query (JPQL)

Now, let's test the new endpoints that use our custom JPQL queries. Make sure you have some products created (you can re-run the POST requests from Step 7).

- **Test Products by Price Range (GET - Custom JPQL Query):**
    - **URL:** http://localhost:8080/api/products/custom/price-range?minPrice=100.00&maxPrice=1000.00
    - **Method:** GET
    - Send the request. You should get products whose price is between $100 and $1000 (e.g., "Keyboard").
- **Test Products by Description Keyword (GET - Custom JPQL Query):**
    - **URL:** http://localhost:8080/api/products/custom/description-keyword?keyword=wireless
    - **Method:** GET
    - Send the request. You should get products whose description contains "wireless" (e.g., "Mouse").
- **Test Products Low Stock High Price (GET - Custom JPQL Query):**
    - **URL:** http://localhost:8080/api/products/custom/low-stock-high-price?maxQuantity=100&minPrice=50.00
    - **Method:** GET
    - Send the request. This should return products with quantity less than 100 AND price greater than $50 (e.g., "Keyboard" if its quantity is below 100)

Activity 4.1: Spring AOP “Logging with Aspects”

This activity will guide you through understanding and implementing Aspect-Oriented Programming (AOP) in a Spring Boot application. We will use AOP to add logging functionality to a UserService without modifying its core business logic.

STEP 1: Project Setup (Spring Boot)

We'll use Spring Initializr to set up a new Spring Boot project.

1. **Go to Spring Initializr:** Open your web browser and navigate to https://start.spring.io/.
2. **Configure Your Project:**
   - **Project:** Maven Project
   - **Language:** Java
   - **Spring Boot:** Choose the latest stable version (e.g., 3.x.x).
   - **Group:** com.example.aop.app
   - **Artifact:** aop-demo
   - **Name:** aop-demo
   - **Description:** Spring AOP Logging Demo
   - **Package Name:** com.example.aop
   - **Packaging:** Jar
   - **Java:** Choose Java 17 or higher.
3. **Add Dependencies:** In the "Dependencies" section, search for and add the following:
   - **Spring Web:** For a basic REST controller to trigger our service.
   - **Spring AOP:** Essential for AOP functionality.
     Note: If you cannot find "Spring AOP" directly in the search, you can generate the project without it and add it manually in the next step.
   - **Lombok:** (Optional but recommended) Reduces boilerplate code.
4. **Generate and Download:** Click the "Generate" button. Download the .zip file.
5. **Import into IDE:** Unzip the downloaded file and import the project into your IDE (IntelliJ IDEA, Eclipse, VS Code).