**1. Setup Environment**

**# Software Downloads:**

IntelliJ [https://www.jetbrains.com/idea/download/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbXpTNUxYYTE3cmd3cENVcXZWR3VabHp6MXRRQXxBQ3Jtc0ttY2RWT1R3N0c3RFdXQW91cTVMckJXUkZmZU1hdGg3Q19VX2tlLXFnVW41RE1fTnMzVXVzUzJ2SmI4VmZ6czI2dDFEUWZ4WGFpR09Wem5oYkV1eDlDaUpfa05vOWJlMUdOMTJCM2dYam1nck5uV2lHMA&q=https%3A%2F%2Fwww.jetbrains.com%2Fidea%2Fdownload%2F&v=z8AWVe5se7M)

MySQL workbench: [https://dev.mysql.com/downloads/workb...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa3lvYVI2RTlxX2E1T3VNY1JzUmROTjRrcGdIZ3xBQ3Jtc0trX2s1YWN6U3czeGZEZnI3a08zOEphQzJvR3BIVkN4Z3VtaVBoSFMyaHFmdXRkbTlqUm5saUtyVXh0aHNLUlBULUxWLVhiV2E1T2RULWlBN0NiRjlfRWdRZVFPUGVuRU03R05XTEVCMnNXRDFVc3k0aw&q=https%3A%2F%2Fdev.mysql.com%2Fdownloads%2Fworkbench%2F&v=z8AWVe5se7M)

MySQL server: [https://dev.mysql.com/downloads/mysql/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqblZMelo1TG4zSzlSS3ZOZGJqYnFjMDBLWEozZ3xBQ3Jtc0tuSDF0NzVvT0xxX1RZN1ltWFh4RVVzbnhuY3hmZ2xtMC1CRk53c1VMVUxEWFhPVE1HVkFzSmk5aV9uWEJPSl9mb2Z0cm9oeEFJUFQyTElYNUxqT3lZNk1KbnpKenJVMk9KMF8xZE9kelFTczc2YWpFcw&q=https%3A%2F%2Fdev.mysql.com%2Fdownloads%2Fmysql%2F&v=z8AWVe5se7M)

Postman: [https://www.postman.com/downloads/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbVp2ZXN0dHFoU29BNFMtUnpJTk00ZWlUZ2V0QXxBQ3Jtc0trUGdVM0ZLUlc1cnRPZHZ4V2g2S3A1V01JbzM1X2tnYmxyaFE3MXlZMEJYR0I5VnhDQ1RwejhnWHFRdXVVakk5N1VidzFSNGFVVFlrVFBvcDFnTHU2cU8zT21DXzZxZUR6WnhuSVUtc3VWRDRILVpJcw&q=https%3A%2F%2Fwww.postman.com%2Fdownloads%2F&v=z8AWVe5se7M)

Spring Initializr: [https://start.spring.io/](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa3A4WDkzckNqTHM2eU8zX01FbHQ3SUpXQ3lqUXxBQ3Jtc0ttcFJhZ2F2NTJTT1JkSm8tY3NpSV9kbnFYdFc0Sk1zX2ZuVi1hV0JZRDRkMkp3TWlxRHlRRFRmSklMZS1rMGlnSFpWRlpXV0JmUGV4NE55NFJJWmNWU2cxRGJMcU9VOFgxemdPVy1UOUlXRUtZTTFsYw&q=https%3A%2F%2Fstart.spring.io%2F&v=z8AWVe5se7M)

Dependencies: Lombok, Spring Web, Spring Data JPA, Spring Data JDBC

1. **Lombok**: Lombok is a library that helps reduce boilerplate code in Java classes. It provides annotations to automatically generate getters, setters, constructors, and other repetitive code, making development faster and cleaner.
2. **Spring Web**: Spring Web provides essential features for building web applications in Spring Boot. It includes components for handling HTTP requests, routing, and serving web content. It also supports RESTful web services development.
3. **Spring Data JPA**: Spring Data JPA simplifies data access in Spring applications using the Java Persistence API (JPA). It provides abstraction over JPA implementations and offers features like repository support, query creation, and pagination, making database interaction more efficient and easier to manage.
4. **Spring Data JDBC**: Spring Data JDBC is an alternative to ORM frameworks like Hibernate for database access. It provides a simpler approach to working with relational databases by mapping Java objects directly to database tables without the need for entity classes or complex mappings. This results in better performance and more control over SQL queries.

**# Imp Files**

**1. Project/pom.xml:** Contains all the things included when we did initializer setup

**2. src/main/resources/application.properties:** contains configuration settings for the application. These settings can include properties related to database connection, server port, logging configuration, security settings, and various other application-specific parameters.

**3. src/main/java/com.example.demo/NoBsSpringbootApplication:** Add print statement to verify System.out.println("Hello World");

MySQL connector dependency [https://mvnrepository.com/artifact/co...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa192bWRtWmk1Vnk4Qkc0aHpidUlTYW13SU1xUXxBQ3Jtc0tsajdEUUJEZ3BrUlVSQmxYSDhvNFFyWUFRYXFSUmZlZ1Y0a1dXYU5pcHZIUWRBQ3ZlZzliR1FOcTNSVHJ2U0RTc1k2QjJGckNzd0xqQzdGQndHVU5NbXFfVlB0bVN5TWVDZXJzTTVSNkYxZkhnanZZZw&q=https%3A%2F%2Fmvnrepository.com%2Fartifact%2Fcom.mysql%2Fmysql-connector-j&v=z8AWVe5se7M)

**# Application.properties:**

spring.datasource.url=jdbc:mysql://localhost:3306/nobs

spring.datasource.username=root

spring.datasource.password=12345

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

**# MySQL Workbench:**

Setup new connection > name: nobs >password: 12345

Rootpassword=12345

**# SQL query to create your database:**

CREATE database nobs;

**2. The Controller (1/2), @Get Endpoint & Postman**

**# Annotations**

Spring Boot utilizes annotations extensively to simplify the configuration and development of Spring applications. Here's a brief overview of some commonly used annotations in Spring Boot:

1. @SpringBootApplication: Marks the main class of the application. It combines three annotations: @Configuration, @EnableAutoConfiguration, and @ComponentScan.
2. @Controller: Indicates that a class serves as a controller in Spring MVC, handling HTTP requests.
3. @RestController: A specialized version of @Controller that's used for RESTful web services. It automatically serializes return objects into JSON or XML.
4. @RequestMapping: Maps HTTP requests to handler methods in controllers. It's used to specify the URL mapping for a particular controller or controller method.
5. @Autowired: Marks a constructor, field, or setter method to automatically inject beans by type.
6. @Service: Indicates that a class is a service component in Spring. It's used to mark service classes.
7. @Repository: Marks a class as a data access component in Spring. It's typically used to indicate a DAO (Data Access Object) class.
8. @Component: Marks a class as a Spring component. Spring will automatically detect and register these components during component scanning.
9. @Configuration: Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.
10. @EnableAutoConfiguration: Enables Spring Boot's automatic configuration mechanism, which automatically configures the Spring application based on the dependencies and settings present in the classpath.

**# Creating REST Controller (Location of an endpoint)**

@GetMapping  
public String getProduct(){  
 System.*out*.println("get Products method");  
 return "get Products endpoint";  
}

@GetMapping  
public ResponseEntity getProduct(){  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(Collections.*emptyList*());  
}

**ResponseEntity:** Allows us to control how we respond

**3. Create mySQL Table, @Entity, @Repository**

**#1. Create Java Object**

Create Model folder in Product folder where the model class is going to live.

**Product.java**

@Entity // Marks this class as a JPA entity, representing a table in the database.  
@Data // Generates boilerplate code for getters, setters, toString, equals, and hashCode methods.  
@Table(name="product") // Specifies the name of the table in the database corresponding to this entity.  
public class Product {  
 @Id // Marks the primary key field of the entity.  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*) // Specifies the generation strategy for the primary key.  
 @Column(name ="id") // Specifies the column name in the database table for this field.  
 private Integer id;  
  
 @Column(name = "name")  
 private String name;  
  
 @Column(name = "description")  
 private String description;  
  
 @Column(name = "price")  
 private Double price;  
  
 @Column(name = "quantity")  
 private Integer quantity;  
}

**#2. Create Table in MySql**

use nobs;

CREATE TABLE product (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR (255) ,

description VARCHAR (255) ,

price DOUBLE,

quantity INT

);

**#3. Create Repository Interface: Connects MySql to Java code.**

This interface extends JpaRepository, which provides methods for basic CRUD operations on the Product entity. JpaRepository takes two parameters: the entity type (Product) and the type of its primary key (Integer). The @Repository annotation indicates that this interface should be detected by Spring component scanning and registered as a bean in the Spring application context.

@Repository  
public interface ProductRepository extends JpaRepository<Product, Integer> {  
}

**4. The Controller (2/2) @Post @Put @Delete**

**#1. Post**

@PostMapping  
public ResponseEntity createProduct(@RequestBody Product product){  
 productRepository.save(product);  
 return ResponseEntity.*ok*().build();  
 // .build() is used to construct and finalize the ResponseEntity with an HTTP status code indicating success (200 OK) without providing a body.  
}  
// The @RequestBody annotation binds the HTTP request body to the parameter product in the method createProduct().

**#2. Put**

@PutMapping("/{id}")  
public ResponseEntity updateProduct(@PathVariable Integer id, @RequestBody Product product){  
 product.setId(id);  
 productRepository.save(product);  
 return ResponseEntity.*ok*().build();  
}

**#3. Delete**

@DeleteMapping("/{id}")  
public ResponseEntity deleteProduct(@PathVariable Integer id){  
 Product product = productRepository.findById(id).get();  
 productRepository.delete(product);  
 return ResponseEntity.*ok*().build();  
}

**5. Command Query Responsibility Segregation (CQRS): Separates Commands & Queries**

It is a design pattern used in software engineering to segregate the operations that read data (queries) from the operations that modify data (commands) into separate responsibilities.

**A black and white sign

Description automatically generated**

**6. Query Handler**

Create a new interface query & new folder queryhandlers (where all query handlers for products will live) for this.

A screenshot of a computer

Description automatically generated

**# Query Interface**  
This code defines a generic interface Query<I, O> in a Spring Boot application. It specifies a method execute(I input) that takes an input of type I and returns a ResponseEntity containing a result of type O. This interface is meant to represent a query operation in a Command Query Responsibility Segregation (CQRS) architecture, where I represents the input parameters for the query and O represents the output/result of the query.

package com.example.demo;  
import org.springframework.http.ResponseEntity;  
public interface Query <I, O>{  
 ResponseEntity<O> execute(I input);  
}

**#1 Get All Products Query Handler**

@GetMapping  
public ResponseEntity<List<Product>> getProducts(){  
 return getAllProductsQueryHandler.execute(null);  
}

@Service // Tells Spring boot that this is your business logic  
public class GetAllProductsQueryHandler implements Query<Void, List<Product>> {  
  
 @Autowired // Injects an instance of ProductRepository into this class.  
 private ProductRepository productRepository;  
 @Override  
 // Executes the query to retrieve all products from the database.  
 public ResponseEntity<List<Product>> execute(Void input) {  
 // Returns a ResponseEntity with HTTP status OK and the list of products retrieved from the repository.// Returns a ResponseEntity with HTTP status OK and the list of products retrieved from the repository.  
 return ResponseEntity.*ok*(productRepository.findAll());  
 }  
}

**#2 Get Product Query Handler**

@Service // Tells Spring boot that this is your business logic  
public class GetProductQueryHandler implements Query<Integer, Product> {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 @Override  
 public ResponseEntity<Product> execute(Integer id) {  
 // Optionals: Lets go to repository lets try to find by id if found return the product else throw a null pointer exception  
 Optional<Product> product = productRepository.findById(id);  
 if(product.isEmpty()){  
 // throw an exception  
 throw new RuntimeException("Product not found");  
 }  
 return ResponseEntity.*ok*(product.get());  
 }  
}

**7. Command Handler and Custom Validation**

**#Create Command.java interface**

This Java code defines an interface called Command in a package named com.example.demo. The interface has a method execute that takes an object of type E (representing an entity) and returns a ResponseEntity containing an object of type T.

In simpler terms, this interface is like a blueprint for a command. It declares a method called execute that can be used to perform some action on an object (entity). The method returns a response, which is wrapped inside a ResponseEntity. The types E and T are placeholders for different types of objects that the method can handle.

package com.example.demo;  
  
import org.springframework.http.ResponseEntity;  
  
// E for entity and T is generic in JAVA  
public interface Command<E, T> {  
 ResponseEntity<T> execute(E entity);  
}

**#1. Create Product Command Handler**

@Service  
@Override  
public ResponseEntity execute(Product product) {  
 validateProduct(product);  
 productRepository.save(product);  
 return ResponseEntity.*ok*().build();  
 // .build() is used to construct and finalize the ResponseEntity with an HTTP status code indicating success (200 OK) without providing a body.  
}  
  
private void validateProduct(Product product){  
 // Logic needs to be added to validate the fields  
 // Name - Non-Null, No whitespace, Not Empty  
 if(StringUtils.*isBlank*(product.getName())){  
 // Throw Exception  
 throw new RuntimeException("Product name cannot be empty");  
 }  
 // Description  
 if(StringUtils.*isBlank*(product.getDescription())){  
 // Throw Exception  
 throw new RuntimeException("Description name cannot be empty");  
 }  
 // Price  
 if(product.getPrice() <= 0.0){  
 // Throw Exception  
 throw new RuntimeException("Price Cannot be negative");  
 }  
 // Quantity  
 if(product.getQuantity() <= 0){  
 // Throw Exception  
 throw new RuntimeException("Quantity Cannot be negative");  
 }  
}

**#2. Update Product Command Handler**

We cannot pass two parameters for update command handler, therefore we have to create a new model for id & product.

@Data  
public class UpdateProductCommand {  
 private int id;  
 private Product product;  
  
 public UpdateProductCommand(int id, Product product) {  
 this.id = id;  
 this.product = product;  
 }  
}

@Service  
public class UpdateProductCommandHandler implements Command<UpdateProductCommand, ResponseEntity> {  
 @Autowired  
 private ProductRepository productRepository;  
 @Override  
 public ResponseEntity<ResponseEntity> execute(UpdateProductCommand command) {  
 Product product = command.getProduct();  
 product.setId(command.getId());  
 productRepository.save(product);  
 return ResponseEntity.*ok*().build();  
 }  
}

**#3. Delete Product Command Handler**

@Service  
public class DeleteProductCommandHandler implements Command<Integer, ResponseEntity> {  
 @Autowired  
 private ProductRepository productRepository;  
 @Override  
 public ResponseEntity<ResponseEntity> execute(Integer id) {  
 Product product = productRepository.findById(id).get();  
 productRepository.delete(product);  
 return ResponseEntity.*ok*().build();  
 }  
}

**7. Dependency Injection and @Autowired**

1. When we annotate this (@Autowired private ProductRepository productRepository;) with @Autowired spring boot goes and finds ‘ProductRepository’ brings it over to the class and injects it so that we can use it in the class, which is called as **dependency injection** and here we are using **Field Injection**.

2. **Field injection** in dependency injection involves injecting dependencies **directly into the fields of a class**, typically through reflection, rather than using constructor or setter injection.

3. In order for this to work ProductRepository or any other components need to be annotated with **@Repository, @Service or @Component.**

4. When there is an annotation java spring boot will come into the annotated class, it will recognize it as a **Spring Bean** and will bring it to the main class that we intend to use the bean in.

5. Another way to do it is Constructor Injection (like in angular typescript).

private final ProductRepository productRepository;  
private final GetAllProductsQueryHandler getAllProductsQueryHandler;  
private final GetProductQueryHandler getProductQueryHandLer;  
private final CreateProductCommandHandler createProductCommandHandler;  
private final UpdateProductCommandHandler updateProductCommandHandler;  
private final DeleteProductCommandHandler deleteProductCommandHandler;  
  
@Autowired  
public ProductController(ProductRepository productRepository, GetAllProductsQueryHandler getAllProductsQueryHandler, GetProductQueryHandler getProductQueryHandLer, CreateProductCommandHandler createProductCommandHandler, UpdateProductCommandHandler updateProductCommandHandler, DeleteProductCommandHandler deleteProductCommandHandler) {  
 this.productRepository = productRepository;  
 this.getAllProductsQueryHandler = getAllProductsQueryHandler;  
 this.getProductQueryHandLer = getProductQueryHandLer;  
 this.createProductCommandHandler = createProductCommandHandler;  
 this.updateProductCommandHandler = updateProductCommandHandler;  
 this.deleteProductCommandHandler = deleteProductCommandHandler;  
}

**8. DTO’s (Data Transfer Objects)**

# Data Transfer Objects (DTOs) are objects used to encapsulate and transport data between different layers of an application, typically between the **presentation and business layers**, providing a clean separation and facilitating efficient communication.

**# Advantages:** Sending Less Data Over Network, Hide Sensitive Data, Hide Internal Database Implementation, Roles Based Access Control For Same Database Data.

# Ex. For Product Class (id, name, description, price, quantity) we can have a Product DTO Class with just name, description and price.

@Data  
public class ProductDTO {  
 private String name;  
 private String description;  
 private Double price;  
  
 public ProductDTO(Product product) {  
 this.name = product.getName();  
 this.description = product.getDescription();  
 this.price = product.getPrice();  
 }  
}

# In ProductController we are going to update Product to ProductDTO for get mappings.

@GetMapping  
public ResponseEntity<List<ProductDTO>> getProducts(){  
 return getAllProductsQueryHandler.execute(null);  
}

public class GetAllProductsQueryHandler implements Query<Void, List<ProductDTO>> {  
 @Autowired // Injects an instance of ProductRepository into this class.  
 private ProductRepository productRepository;  
 @Override  
 // Executes the query to retrieve all products from the database.  
 public ResponseEntity<List<ProductDTO>> execute(Void input) {  
 /\*  
 In this Java code snippet, a list of ProductDTO objects is being created from the entities retrieved by the productRepository. It first retrieves all products from the repository, then converts each product entity into a ProductDTO object using the map() function with ProductDTO::new as the mapping function, which likely invokes a constructor of ProductDTO. Finally, the result is collected into a list using the  
toList() terminal operation.  
 \*/  
 List<ProductDTO> productDTOs = productRepository  
 .findAll()  
 .stream()  
 .map(ProductDTO::new)  
 .toList();  
 // Returns a ResponseEntity with HTTP status OK and the list of products retrieved from the repository.// Returns a ResponseEntity with HTTP status OK and the list of products retrieved from the repository.  
 return ResponseEntity.*ok*(productDTOs);  
 }  
}

**9. Exception Handling**

When an Exception is thrown, we need to tell UI what happened in an elegant way, and not a generic response like this, as status should be 404 (Product not found).

{

    "timestamp": "2024-03-28T17:46:57.686+00:00",

    "status": 500,

    "error": "Internal Server Error",

    "path": "/products/3"

}

**#1. Built in Spring Exception Handling (Properties File)**

1. Go to application.properties file and add these lines.

server.error.include-message=ALWAYS  
server.error.include-stacktrace=ALWAYS

2. In QueryHandler

public class GetProductQueryHandler implements Query<Integer, ProductDTO> {  
  
 @Autowired  
 private ProductRepository productRepository;  
  
 @Override  
 public ResponseEntity<ProductDTO> execute(Integer id) {  
 // Optionals: Lets go to repository lets try to find by id if found return the product else throw a null pointer exception  
 Optional<Product> product = productRepository.findById(id);  
 if(product.isEmpty()){  
 // throw an exception  
 throw new RuntimeException("Product not found");  
 }  
 return ResponseEntity.*ok*(new ProductDTO(product.get()));  
 }  
}

**#2. Custom Exception Handling at Controller Level (@ExceptionHandler)**

1. Create a new folder at demo level called Exceptions to contain all the custom exceptions which extends RuntimeException.

2. Create a ExceptionHandler in ProductController.

@ExceptionHandler(ProductNotFoundException.class)  
public ResponseEntity<String> handleProductNotFoundException(){  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body("Product Not Found");  
}

3. In Command Handler do it like this

Optional<Product> optionalProduct = productRepository.findById(id);  
if(optionalProduct.isEmpty()){  
 throw new ProductNotFoundException();  
}

**#3. Global Exception Handling (@ControllerAdvice)**

1. Create GlobalExceptionHandler in Exceptions folder using @ControllerAdvice which catches all the exceptions thrown by all the controllers across the system.

@ControllerAdvice  
public class GlobalExceptionHandler {  
 @ExceptionHandler(ProductNotFoundException.class)  
 public ResponseEntity<SimpleResponse> handleProductNotFoundException(){  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(new SimpleResponse("Product Not Found"));  
 }  
}

SimpleResponse.java

@Data  
@AllArgsConstructor  
public class SimpleResponse {  
 private String message;  
}

2. Bundling and grouping exceptions, create CustomBaseException.java

@Data  
public class CustomBaseException extends RuntimeException {  
 private HttpStatus status;  
 private SimpleResponse simpleResponse;  
  
 public CustomBaseException(HttpStatus status, SimpleResponse simpleResponse) {  
 this.status = status;  
 this.simpleResponse = simpleResponse;  
 }  
}

public class ProductNotFoundException extends CustomBaseException{  
  
 public ProductNotFoundException() {  
 super(HttpStatus.*BAD\_REQUEST*, new SimpleResponse("Product Not Found"));  
 }  
}

3. Change CreateProductCommandHandler to include more info with ProductNotValidException

public class ProductNotValidException extends CustomBaseException{  
 public ProductNotValidException(String message) {  
 super(HttpStatus.*BAD\_REQUEST*, new SimpleResponse(message));  
 }  
}

private void validateProduct(Product product){  
 if(StringUtils.*isBlank*(product.getName())){  
 throw new ProductNotValidException("Product name cannot be empty");  
 }  
 if(StringUtils.*isBlank*(product.getDescription())){  
 throw new ProductNotValidException("Description name cannot be empty");  
 }  
 if(product.getPrice() <= 0.0){  
 throw new ProductNotValidException("Price Cannot be negative");  
 }  
}

**9. Unit Testing**

**#1 Writing Unit Tests**

**#2 Running Unit Tests**

**#3 Naming**

**#4 Annotations @SPRINGBOOTTEST, @INJECTMOCKS, @MOCK, @TEST**

**#5 Testing Theory (Given/When/Then OR Arrange/Act/Assert)**

// This tells it to use entire project to be able to run  
@SpringBootTest(classes = NoBsSpringbootApplication.class)  
public class CreateProductCommandHandlerTest {  
 // Annotation for command handler we actually want to test  
 @InjectMocks  
 private CreateProductCommandHandler createProductCommandHandler;  
  
 // Using @Mock so that we don't use actual product repository  
 @Mock  
 private ProductRepository productRepository;  
  
 // To name a test we will follow this format  
 // MethodName\_StateUnderTest\_ExpectedBehavior  
 // This will annotate a method that represents a unit test  
 @Test  
 public void createProductCommandHandler\_validProduct\_returnsSuccess(){  
 // Given  
 Product product = new Product();  
 product.setId(1);  
 product.setPrice(9.99);  
 product.setName("Best Chocolate");  
 product.setDescription("Silky Dark");  
 product.setQuantity(10);  
  
 // When  
 ResponseEntity response = createProductCommandHandler.execute(product);  
  
 // Then  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 }  
  
 @Test  
 public void createProductCommandHandler\_invalidPrice\_throwsInvalidPriceException(){  
 // Given  
 Product product = new Product();  
 product.setId(1);  
 product.setPrice(-9.99);  
 product.setName("Best Chocolate");  
 product.setDescription("Silky Dark");  
 product.setQuantity(10);  
  
 // When / Then 1  
 ProductNotValidException exception = *assertThrows*(ProductNotValidException.class, () -> createProductCommandHandler.execute(product));  
  
 // Then 2  
 *assertEquals*("Product price cannot be negative", exception.getSimpleResponse().getMessage());  
 }  
}

**# For queryHandler**

@SpringBootTest(classes = NoBsSpringbootApplication.class)  
public class GetProductQueryHandlerTest {  
 @InjectMocks  
 private GetProductQueryHandler getProductQueryHandler;  
  
 @Mock  
 private ProductRepository productRepository;  
  
 @Test  
 public void getProductQueryHandler\_validId\_returnsProductDTO(){  
 // Given  
 Product product = new Product();  
 product.setId(1);  
 product.setPrice(-9.99);  
 product.setName("Best Chocolate");  
 product.setDescription("Silky Dark");  
 product.setQuantity(10);  
  
 ProductDTO expectedDTO = new ProductDTO(product);  
  
 *when*(productRepository.findById(product.getId())).thenReturn(Optional.*of*(product));  
  
 // When  
 ResponseEntity<ProductDTO> actualResponse =getProductQueryHandler.execute(product.getId());  
  
 *assertEquals*(expectedDTO, actualResponse.getBody());  
 *assertEquals*(HttpStatus.*OK*, actualResponse.getStatusCode());  
  
 }  
  
 @Test  
 public void getProductQueryHandler\_inValidId\_throwsProductNotFoundException(){  
 *when*(productRepository.findById(1)).thenReturn(Optional.*empty*());  
  
 // When / Then 1  
 ProductNotFoundException exception = *assertThrows*(ProductNotFoundException.class, () -> getProductQueryHandler.execute(1));  
  
 //Then  
 *assertEquals*("Product Not Found", exception.getSimpleResponse().getMessage());  
 }  
  
}

**10. JDBC, JPA, Hibernate & Spring Data JPA**

Layers of abstraction:

A diagram of a program

Description automatically generated with medium confidence

1. JDBC (Java Database Connectivity): Technical Connection that links JAVA to MySql

spring.datasource.url=jdbc:mysql://localhost:3306/nobs

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

2. JPA (Java Persistence API): Defines how a JAVA object is mapped to a relational database. This is how we convert database data into a Java object & vice versa. JPA is abstract.

@Repository  
public interface ProductRepository extends JpaRepository<Product, Integer> {}

3. Hibernate: Its is an implementation of JPA, also known as ORN (Object Relational Mapper). Converts POJOS (Plain Old Java Objects) to database queries.

4. Spring Data JPA: It simplifies the implementation of JPA-based data access layers. It provides repositories as a way to interact with databases using the Java Persistence API (JPA) in a more streamlined and concise manner, reducing boilerplate code.

A screen shot of a computer

Description automatically generated

**11. Custom Queries @Query**

These methods are created In Product Repository

1. Get all products under a specific price

@Repository  
public interface ProductRepository extends JpaRepository<Product, Integer> {  
 // : Tells spring data JPA that maxPrice is a variable name, also we can't use \* so using p.  
 @Query("SELECT p FROM Product p where p.price < :maxPrice")  
 List<Product> findProductsWithPriceLessThan(@Param("maxPrice") double maxPrice);  
}

@GetMapping("/search/{maxPrice}")  
public ResponseEntity<List<Product>> findProductByPrice(@PathVariable Double maxPrice){  
 return ResponseEntity.*ok*(productRepository.findProductsWithPriceLessThan(maxPrice));  
}

2. Get limited data field for DTO

Create new constructor in DTO to handle just fields instead of whole Product

public ProductDTO(String name, String description, Double price){  
 this.name = name;  
 this.description = description;  
 this.price = price;  
}

@Query("SELECT new com.example.demo.Product.Model.ProductDTO(p.name, p.description, p.price) FROM Product p")  
List<ProductDTO> getAllProductDTOs();

In GetAllProductsQueryHandler

List<ProductDTO> productDTOs = productRepository.getAllProductDTOs();  
return ResponseEntity.*ok*(productDTOs);

3. Native Query: It is saying that this query is native to MySql and it will only work there and not in PostGres.

@Query(value = "SELECT p FROM Product p where p.price < :maxPrice", nativeQuery = true)

**12. Spring Data JPA**

If we write our method at repository level in a structured way, we don’t need to write SQL query. Spring Data will interpret method and generate those queries for us.

<https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html>

Ex.

A screen shot of a computer program

Description automatically generated

**13. Query String Parameters**

Not unique to Java Spring Boot. HTTP requests -> can pass data in the URL in the form

of a key:value pair.

Ex. /products/search?description=gaming, /products/search?description=gaming&name=Xbox

In Product Repository

// Using Spring Data JPA here  
List<Product> findByDescriptionContaining(String keyword);  
  
// Same using Query  
@Query("SELECT p FROM Product p WHERE p.description LIKE %:description%")  
List<Product> customQueryMethod(@Param(value="description") String description);

In Product Controller

@GetMapping("/search")  
public ResponseEntity<List<Product>> searchProducts(@RequestParam(value="description") String description){  
 return ResponseEntity.*ok*(productRepository.findByDescriptionContaining(description));  
}

**13. Relational Mappings**

**#1 @OneToOne (One Customer maps to One Address)**

1. Create table and entries

CREATE TABLE address (

id BIGINT PRIMARY KEY AUTO\_INCREMENT,

street VARCHAR(255),

city VARCHAR(255),

state VARCHAR(255)

);

CREATE TABLE customer (

id BIGINT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(255),

last\_name VARCHAR(255),

address\_id BIGINT,

FOREIGN KEY (address\_id) REFERENCES address(id)

);

Customer

@Entity  
@Table(name="customer")  
@Data  
public class Customer {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "id")  
 private Integer id;  
  
 @Column(name = "first\_name")  
 private String firstName;  
  
 @Column(name = "last\_Name")  
 private String lastName;  
  
 @OneToOne(cascade = CascadeType.*ALL*)  
 @JoinColumn(name = "address\_id")  
 private Address address;  
}

Repository

@Repository  
public interface CustomerRepository extends JpaRepository<Customer, Integer> {  
}

Address

@Entity  
@Table(name = "address")  
@Data  
public class Address {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "id")  
 private Integer id;  
  
 @Column(name = "street")  
 private String street;  
  
 @Column(name = "city")  
 private String city;  
  
 @Column(name = "state")  
 private String state;  
}

Customer Controller

@RestController  
@RequestMapping("/customer")  
public class CustomerController {  
 @Autowired private CustomerRepository customerRepository;  
  
 @GetMapping("/{id}")  
 public ResponseEntity<Customer> getCustomerById(@PathVariable Integer id){  
 return ResponseEntity.*ok*(customerRepository.findById(id).get());  
 }  
}

**#2 @OneToMany (One Customer maps to Many Address)**

public class Customer {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "id")  
 private Integer id;  
  
 @Column(name = "first\_name")  
 private String firstName;  
  
 @Column(name = "last\_Name")  
 private String lastName;  
  
 // (cascade = CascadeType.ALL) If we update a customer who also has an address attached it will also update the address  
 @OneToMany(cascade = CascadeType.*ALL*)  
 @JoinColumn(name = "customer\_id")  
 private List<Address> address;  
}