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());  
 }  
}