SOFTWARE ACADEMY

Spring & Spring Boot

Building Professional REST APIs with Spring Boot, JPA, and MySQL

Module 1	Module 2	Module 3	Module 4	Module 5	Module 6
FUNDAMENTALS	SYSTEMS & TOOLS	JAVA PROGRAMMING	WEB DEVELOPMENT	JAVA FRAMEWORKS	FINAL PROJECT
Software Development Fundamentals	Version Control System	Object-Oriented Analysis & Design	HTML 5 CSS 3 JavaScript	Spring	Mentoring
Relational Databases	Issue tracking and project management tools	Programming Concept & Java Core Api	Java EE Core API	JPA / Hibernate	Implementation
Database Design & Normalization	Development platform	Libraries, build tools and dependency management	Servlets	JAX-RS / Jersey	Practical approach
UML, MySQL	GitHub, JIRA	IntelliJ, Maven	Tomcat	Spring Boot	SDLC

Module

5

SPRING & SPRING BOOT

Building REST APIs with Spring Boot

Agenda

Part 1: Understanding REST APIs

Before we dive into Spring Boot, we need to understand what we're building

What is an API?

API = Application Programming Interface

An API is a way for different software applications to communicate with each other.

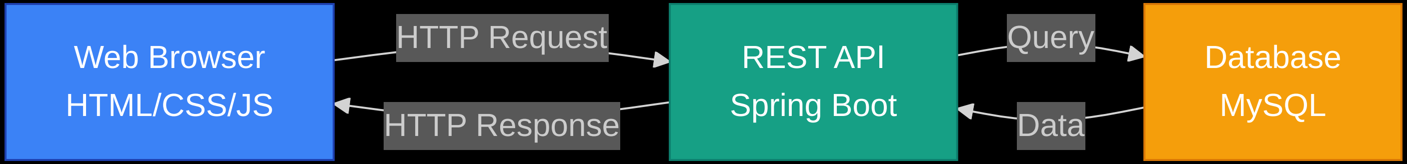
Real-World Analogy

Think of a **restaurant**:

- **You (Frontend/Client)**: Want to order food
- **Waiter (API)**: Takes your order to the kitchen and brings back your food
- **Kitchen (Backend/Server)**: Prepares the food

The waiter (API) is the interface between you and the kitchen. You don't need to know how the kitchen works, you just need to know how to communicate with the waiter!

How Web Applications Work



Three Main Components:

1. **Frontend (Client):** What users see and interact with (HTML, CSS, JavaScript)
2. **Backend (Server):** Business logic and data processing (Spring Boot) ← **Today's Focus**
3. **Database:** Stores all the data (MySQL)

What is REST?

REST = **RE**presentational **S**tate **T**ransfer

REST is an architectural style for designing networked applications. It uses HTTP protocol.

Key Principles of REST:

1. **Client-Server Architecture:** Separation of concerns
2. **Stateless:** Each request contains all information needed
3. **Resource-Based:** Everything is a resource (Users, Products, Orders)
4. **Standard HTTP Methods:** GET, POST, PUT, DELETE

REST API = Web Service that follows REST principles
We'll build REST APIs that can be consumed by any frontend (web, mobile, desktop)

HTTP Methods (CRUD Operations)

HTTP Methods define what action you want to perform:

HTTP Method	CRUD Operation	Purpose	Example
GET	**Read**	Retrieve data	Get list of products
POST	**Create**	Create new resource	Add new product
PUT	**Update**	Update existing resource	Update product details
DELETE	**Delete**	Remove resource	Delete a product

CRUD = Create, Read, Update, Delete (Basic operations for any application)

REST API Example - Product Management

Let’s say we’re building an online shop. Here are typical API endpoints:

```
Base URL: http://localhost:8080/api

GET    /products              → Get all products
GET    /products/1            → Get product with ID 1
POST   /products              → Create new product
PUT    /products/1            → Update product with ID 1
DELETE /products/1            → Delete product with ID 1
```

URL Structure:

- `/api` : API prefix
- `/products` : Resource name (plural)
- `/1` : Resource ID (specific item)

HTTP Request & Response

HTTP Request Structure

```
POST /api/products HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Laptop",
  "price": 999.99,
  "category": "Electronics"
}
```

Request Components:

- **Method:** POST
- **URL:** /api/products
- **Headers:** Content-Type, Authorization, etc.
- **Body:** Data being sent (JSON format)

HTTP Response

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 1,
  "name": "Laptop",
  "price": 999.99,
  "category": "Electronics",
  "createdAt": "2024-01-15T10:30:00"
}
```

Response Components:

- **Status Code:** 201 Created
- **Headers:** Content-Type, etc.
- **Body:** Data being returned (JSON format)

HTTP Status Codes

Status codes tell you what happened with your request:

Code Range	Category	Common Codes
2xx	✅ Success	200 OK, 201 Created, 204 No Content
4xx	× Client Error	400 Bad Request, 401 Unauthorized, 404 Not Found
5xx	💥 Server Error	500 Internal Server Error, 503 Service Unavailable

Most Important Ones:

- **200 OK:** Request successful
- **201 Created:** Resource created successfully
- **400 Bad Request:** Invalid data sent
- **404 Not Found:** Resource doesn’t exist

JSON Format

JSON = JavaScript Object Notation

JSON is the standard format for exchanging data in REST APIs.

```json
{
  "id": 1,
  "name": "Laptop",
  "price": 999.99,
  "inStock": true,
  "tags": ["electronics", "computers"],
  "specifications": {
    "brand": "Dell",
    "ram": "16GB",
    "storage": "512GB SSD"
  }
}
```

JSON Data Types: String, Number, Boolean, Array, Object, null

Part 2: Spring Framework

Understanding the foundation before Spring Boot

What is the Spring Framework?

Spring Framework is a comprehensive framework for building enterprise Java applications.

Created: 2002 by Rod Johnson

Purpose: Simplify Java enterprise development

Core Concept: Inversion of Control (IoC)

Instead of you creating and managing objects, Spring does it for you!

Think of Spring as a smart container that creates, configures, and manages all your application objects (called "beans")

Traditional Java vs Spring

Without Spring (Traditional Way)

```java
public class OrderService {
    private EmailService emailService;
    private PaymentService paymentService;

    public OrderService() {
        // You manually create dependencies
        this.emailService = new EmailService();
        this.paymentService = new PaymentService();
    }

    public void createOrder(Order order) {
        paymentService.processPayment(order);
        emailService.sendConfirmation(order);
    }
}
```

Problems: Tight coupling, hard to test, hard to change implementations

With Spring Framework

```java
@Service
public class OrderService {
    private final EmailService emailService;
    private final PaymentService paymentService;

    // Spring automatically injects dependencies
    @Autowired
    public OrderService(EmailService emailService,
                        PaymentService paymentService) {
        this.emailService = emailService;
        this.paymentService = paymentService;
    }

    public void createOrder(Order order) {
        paymentService.processPayment(order);
        emailService.sendConfirmation(order);
    }
}
```

Spring creates and injects the dependencies automatically! This is Dependency Injection (DI)

Dependency Injection (DI)

Dependency Injection = Providing objects that a class needs from outside

Three Types of DI in Spring:

1. **Constructor Injection** (Recommended ✅)

```java
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

Dependency Injection Types (Continued)

2. **Setter Injection**

```java
@Service
public class UserService {
    private UserRepository userRepository;

    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

3. **Field Injection (Not Recommended ⚠️)**

```java
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
}
```

Always use Constructor Injection - it's the best practice!

Spring Annotations

Annotations tell Spring how to manage your classes:

Annotation	Purpose	Example
`@Component`	Generic Spring-managed component	Utility classes
`@Service`	Business logic layer	OrderService, UserService
`@Repository`	Data access layer	UserRepository
`@Controller`	Web layer (traditional MVC)	Web pages
`@RestController`	REST API endpoints	REST APIs ← **We'll use this!**
`@Autowired`	Inject dependencies	Constructor injection

These annotations help Spring identify and manage your components automatically!

The Problem with Spring Framework

While Spring Framework is powerful, it requires a lot of configuration:

Traditional Spring Challenges:

1. **XML Configuration:** Hundreds of lines of XML
2. **Manual Dependency Management:** Add each library manually
3. **Server Configuration:** Deploy to external Tomcat
4. **Boilerplate Code:** Lots of repetitive setup

```xml
<!-- Traditional Spring Configuration (XML) -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="password"/>
</bean>
<!-- ... hundreds more lines ... -->
```

This is where Spring Boot comes to the rescue! 🚀

Part 3: Spring Boot Fundamentals

Spring Boot = Spring Framework + Convention over Configuration

What is Spring Boot?

Spring Boot makes it easy to create production-ready Spring applications.

Released: 2014 (built on top of Spring Framework)

Motto: "Just Run!" ⚡

Key Features:

1. **Auto-Configuration:** Automatically configures your application based on dependencies
2. **Starter Dependencies:** Pre-packaged dependency sets
3. **Embedded Server:** No need for external Tomcat - server is built-in!
4. **Production-Ready:** Built-in monitoring, health checks
5. **No XML Configuration:** Everything in Java or properties files

Spring Boot = Spring Framework - Complexity + Productivity

Spring vs Spring Boot

Aspect	Spring Framework	Spring Boot
Configuration	Extensive XML/Java config	Auto-configured
Dependency Management	Manual	Starter POMs
Server	External (Tomcat, etc.)	Embedded
Setup Time	Hours/Days	Minutes
Production Ready	Need additional setup	Built-in features

Analogy:

- **Spring Framework:** Building a car from individual parts
- **Spring Boot:** Buying a ready-to-drive car with all features

Spring Boot Starters

Starters are pre-configured dependency packages for common tasks.

Common Starters:

```xml
<!-- Web applications with REST APIs -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- Database access with JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- MySQL Driver -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Spring Boot Starters (Continued)

```xml
<!-- Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<!-- Validation -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Each starter brings in all necessary dependencies automatically! No need to manage versions.

Creating a Spring Boot Project

Option 1: Spring Initializr (Web Interface)

1. Go to https://start.spring.io/
2. Choose:
   - **Project**: Maven
   - **Language**: Java
   - **Spring Boot**: 3.4.1 (latest stable)
   - **Java**: 21
3. Add Dependencies:
   - Spring Web
   - Spring Data JPA
   - MySQL Driver
4. Click "Generate" → Download ZIP → Extract → Open in IntelliJ

Creating Spring Boot Project (Continued)

Option 2: IntelliJ IDEA (Built-in)

1. **File** → **New** → **Project**
2. Select **Spring Initializr**
3. Configure settings (same as web interface)
4. Add dependencies
5. Click **Create**

Both methods generate the exact same project structure!

Spring Boot Project Structure

```
my-spring-boot-app/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/example/demo/
│   │   │       ├── DemoApplication.java        ← Main class
│   │   │       ├── controller/                 ← REST Controllers
│   │   │       ├── service/                    ← Business Logic
│   │   │       ├── repository/                 ← Database Access
│   │   │       └── model/                      ← Entity Classes
│   │   └── resources/
│   │       ├── application.properties          ← Configuration
│   │       └── static/                         ← Static files
│   └── test/                                   ← Unit tests
├── pom.xml                                     ← Maven dependencies
└── README.md
```

Main Application Class

Every Spring Boot app starts from a main class:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

@SpringBootApplication combines three annotations:

- @Configuration - Java-based configuration
- @EnableAutoConfiguration - Auto-configure based on dependencies
- @ComponentScan - Scan for components in package and sub-packages

Configuration Files

application.properties

```properties
# Server Configuration
server.port=8080

# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA Configuration
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

# Application Name
spring.application.name=demo
```

Configuration Files (YAML Alternative)

application.yml (Alternative format)

```yaml
server:
  port: 8080

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: password
    driver-class-name: com.mysql.cj.jdbc.Driver

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQLDialect

  application:
    name: demo
```

Part 4: Building REST APIs

Creating REST Controllers in Spring Boot

Your First REST Controller

```java
package com.example.demo.controller;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, Spring Boot!";
    }

    @GetMapping("/hello/{name}")
    public String helloName(@PathVariable String name) {
        return "Hello, " + name + "!";
    }
}
```

Test it: Run application → Open browser → http://localhost:8080/api/hello

Controller Annotations Explained

Annotation	Purpose
`@RestController`	Marks class as REST API controller
`@RequestMapping`	Base URL for all endpoints in controller
`@GetMapping`	Handle GET requests
`@PostMapping`	Handle POST requests
`@PutMapping`	Handle PUT requests
`@DeleteMapping`	Handle DELETE requests
`@PathVariable`	Extract value from URL path

Path Variables vs Request Parameters

Path Variable (Part of URL)

```java
@GetMapping("/users/{id}")
public User getUser(@PathVariable Long id) {
    return userService.findById(id);
}

// URL: http://localhost:8080/api/users/5
```

Request Parameter (Query String)

```java
@GetMapping("/users/search")
public List<User> searchUsers(@RequestParam String name) {
    return userService.findByName(name);
}

// URL: http://localhost:8080/api/users/search?name=John
```

Request Body Example

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.save(user);
    }
}
```

Request:

```
POST /api/users
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com"
}
```

Spring automatically converts JSON to Java object (@RequestBody) and Java object to JSON (return value)!

ResponseEntity for Better Control

```java
@PostMapping
public ResponseEntity<User> createUser(@RequestBody User user) {
    User savedUser = userService.save(user);
    return ResponseEntity
            .status(HttpStatus.CREATED)  // 201 Created
            .body(savedUser);
}

@GetMapping("/{id}")
public ResponseEntity<User> getUser(@PathVariable Long id) {
    return userService.findById(id)
            .map(ResponseEntity::ok)      // 200 OK
            .orElse(ResponseEntity
                    .notFound()           // 404 Not Found
                    .build());
}
```

ResponseEntity gives you control over status codes, headers, and response body

Layered Architecture Pattern

Why Layered Architecture?

Benefits:

1. **Separation of Concerns:** Each layer has a specific responsibility
2. **Maintainability:** Easy to find and fix bugs
3. **Testability:** Test each layer independently
4. **Reusability:** Service logic can be used by multiple controllers
5. **Scalability:** Easy to add new features

Anti-Pattern: Don't put database logic in controllers!

Bad: Controller → Database directly

Good: Controller → Service → Repository → Database

Part 5: Database Integration

Spring Data JPA & MySQL

What is JPA?

JPA = Java Persistence API

JPA is a specification for Object-Relational Mapping (ORM) in Java.

ORM = Object-Relational Mapping

Java Object (Entity)

```java
public class User {
    private Long id;
    private String name;
    private String email;
}
```

Database Table

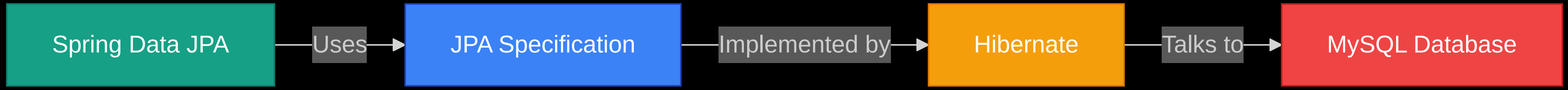
```sql
CREATE TABLE users (
    id BIGINT PRIMARY KEY,
    name VARCHAR(255),
    email VARCHAR(255)
);
```

ORM automatically maps Java objects to database tables. No need to write SQL!

JPA vs Hibernate vs Spring Data JPA

Technology	What is it?
JPA	Specification (interface) - defines standards
Hibernate	Implementation of JPA - most popular
Spring Data JPA	Spring’s abstraction over JPA - makes it even easier

Relationship:



When you use Spring Data JPA, you get Hibernate automatically as the JPA provider!

Creating an Entity

```java
package com.example.demo.model;

import jakarta.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Double price;

    private String description;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

```

Entity Annotations Explained

Annotation	Purpose
`@Entity`	Marks class as JPA entity (database table)
`@Table`	Specifies table name (optional, defaults to class name)
`@Id`	Marks primary key field
`@GeneratedValue`	Auto-generate ID values
`@Column`	Customize column properties
`@Transient`	Field not persisted to database

Common GenerationType Strategies:

- **IDENTITY**: Database auto-increment (MySQL, PostgreSQL)
- **SEQUENCE**: Database sequence (Oracle, PostgreSQL)
- **AUTO**: Let JPA choose (default)

Complete Entity with Getters/Setters

```java
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Double price;

    private String description;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    // Default constructor (required by JPA)
    public Product() {
        this.createdAt = LocalDateTime.now();
    }

```

Entity Getters and Setters

```java
    // Getters
    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Double getPrice() {
        return price;
    }

    public String getDescription() {
        return description;
    }

    public LocalDateTime getCreatedAt() {
        return createdAt;
    }

    // Setters
    public void setId(Long id) {
        this.id = id;
```

Entity Setters (Continued)

```java
    public void setPrice(Double price) {
        this.price = price;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void setCreatedAt(LocalDateTime createdAt) {
        this.createdAt = createdAt;
    }

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

Spring Data JPA Repository

```java
package com.example.demo.repository;

import com.example.demo.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.List;

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

    // Spring Data JPA automatically implements these!
    // No need to write any SQL or implementation code!

    // Custom query methods (Spring generates SQL automatically)
    List<Product> findByName(String name);
    List<Product> findByPriceLessThan(Double price);
    List<Product> findByNameContaining(String keyword);
}
```

Just by extending JpaRepository, you get all CRUD methods for FREE!

Built-in Repository Methods

By extending `JpaRepository<Product, Long>`, you automatically get:

```java
// Create
Product save(Product product);

// Read
Optional<Product> findById(Long id);
List<Product> findAll();
boolean existsById(Long id);
long count();

// Update (same as save)
Product save(Product product);

// Delete
void deleteById(Long id);
void delete(Product product);
void deleteAll();
```

You don't write ANY implementation code - Spring Data JPA generates everything!

Query Method Naming Convention

Spring Data JPA generates SQL based on method names:

Method Name	Generated SQL
`findByName(String name)`	`WHERE name = ?`
`findByPriceLessThan(Double price)`	`WHERE price < ?`
`findByPriceGreaterThan(Double price)`	`WHERE price > ?`
`findByNameAndPrice(String name, Double price)`	`WHERE name = ? AND price = ?`
`findByNameOrPrice(String name, Double price)`	`WHERE name = ? OR price = ?`
`findByNameContaining(String keyword)`	`WHERE name LIKE %?%`
`findByOrderByPriceAsc()`	`ORDER BY price ASC`

Service Layer

```java
package com.example.demo.service;

import com.example.demo.model.Product;
import com.example.demo.repository.ProductRepository;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class ProductService {

    private final ProductRepository productRepository;

    // Constructor injection
    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    public Optional<Product> getProductById(Long id) {
        return productRepository.findById(id);}
```

Service Layer (Continued)

```java
    public Product createProduct(Product product) {
        return productRepository.save(product);
    }

    public Product updateProduct(Long id, Product productDetails) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not found"));

        product.setName(productDetails.getName());
        product.setPrice(productDetails.getPrice());
        product.setDescription(productDetails.getDescription());

        return productRepository.save(product);
    }

    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }

    public List<Product> searchProducts(String keyword) {
        return productRepository.findByNameContaining(keyword);
    }
}
```

Part 6: Complete CRUD REST API

Putting it all together: Controller + Service + Repository

Complete REST Controller - Part 1

```java
package com.example.demo.controller;

import com.example.demo.model.Product;
import com.example.demo.service.ProductService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "*") // Allow frontend to call this API
public class ProductController {

    private final ProductService productService;

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

Complete REST Controller - Part 2

```java
    // GET /api/products - Get all products
    @GetMapping
    public ResponseEntity<List<Product>> getAllProducts() {
        List<Product> products = productService.getAllProducts();
        return ResponseEntity.ok(products);
    }

    // GET /api/products/1 - Get product by ID
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        return productService.getProductById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    // GET /api/products/search?keyword=laptop
    @GetMapping("/search")
    public ResponseEntity<List<Product>> searchProducts(
            @RequestParam String keyword) {
        List<Product> products = productService.searchProducts(keyword);
        return ResponseEntity.ok(products);
    }
```

Complete REST Controller - Part 3

```
    // POST /api/products - Create new product
    @PostMapping
    public ResponseEntity<Product> createProduct(
            @RequestBody Product product) {
        Product savedProduct = productService.createProduct(product);
        return ResponseEntity
                .status(HttpStatus.CREATED)
                .body(savedProduct);
    }

    // PUT /api/products/1 - Update product
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(
            @PathVariable Long id,
            @RequestBody Product product) {
        try {
            Product updatedProduct = productService.updateProduct(id, product);
            return ResponseEntity.ok(updatedProduct);
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }
```

Complete REST Controller - Part 4

```java
        // DELETE /api/products/1 - Delete product
        @DeleteMapping("/{id}")
        public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
            try {
                productService.deleteProduct(id);
                return ResponseEntity.noContent().build();
            } catch (RuntimeException e) {
                return ResponseEntity.notFound().build();
            }
        }
    }
```

This controller provides a complete REST API with all CRUD operations!

MySQL Database Setup

Step 1: Install MySQL

Download and install MySQL from: https://dev.mysql.com/downloads/

Step 2: Create Database

```sql
CREATE DATABASE productdb;
```

Step 3: Configure Spring Boot

```properties
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/productdb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Understanding ddl-auto Options

```
spring.jpa.hibernate.ddl-auto=update
```

Value	What it does
`none`	No action - database must exist
`validate`	Validate schema, make no changes
`update`	Update schema if needed (safe for development) ✅
`create`	Drop and recreate schema each time ⚠️
`create-drop`	Create on start, drop on shutdown 💥

Development: Use `update`
Production: Use `validate` or `none` + migration tools (Flyway/Liquibase)

Running Your Application

Option 1: From IntelliJ IDEA

1. Open main class (`DemoApplication.java`)
2. Click green ▶️ play button
3. Or right-click → Run

Option 2: From Maven

```
mvn spring-boot:run
```

Option 3: Build JAR and Run

```
mvn clean package
java -jar target/demo-0.0.1-SNAPSHOT.jar
```

Application starts on: **http://localhost:8080**

Testing Your API with Postman

Install Postman

Download from: https://www.postman.com/downloads/

Test Endpoints:

1. **GET all products**

```
GET http://localhost:8080/api/products
```

2. **GET product by ID**

```
GET http://localhost:8080/api/products/1
```

3. CREATE new product

```
POST http://localhost:8080/api/products
Content-Type: application/json

{
  "name": "Laptop",
  "price": 999.99,
  "description": "High-performance laptop"
}
```

Testing API - More Examples

4. UPDATE product

```http
PUT http://localhost:8080/api/products/1
Content-Type: application/json

{
  "name": "Gaming Laptop",
  "price": 1299.99,
  "description": "Updated description"
}
```

5. DELETE product

```http
DELETE http://localhost:8080/api/products/1
```

6. SEARCH products

```http
GET http://localhost:8080/api/products/search?keyword=laptop
```

Exception Handling

Create a custom exception:

```java
package com.example.demo.exception;

public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}
```

Update service to throw exception:

```java
public Product updateProduct(Long id, Product productDetails) {
    Product product = productRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException(
                    "Product not found with id: " + id));

    product.setName(productDetails.getName());
    product.setPrice(productDetails.getPrice());
    product.setDescription(productDetails.getDescription());

    return productRepository.save(product);
}
```

Global Exception Handler

```java
package com.example.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Map<String, Object>> handleResourceNotFound(
            ResourceNotFoundException ex) {

        Map<String, Object> error = new HashMap<>();
        error.put("timestamp", LocalDateTime.now());
        error.put("message", ex.getMessage());
        error.put("status", HttpStatus.NOT_FOUND.value());

```

Request Validation

Add validation dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Add validation annotations to entity:

```java
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    @Size(min = 3, max = 100, message = "Name must be between 3-100 characters")
    private String name;

    @NotNull(message = "Price is required")
    @DecimalMin(value = "0.01", message = "Price must be greater than 0")
    private Double price;}
```

Validation in Controller

```java
@PostMapping
public ResponseEntity<Product> createProduct(
        @Valid @RequestBody Product product) {
    Product savedProduct = productService.createProduct(product);
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(savedProduct);
}
```

Handle validation errors:

```java
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<Map<String, String>> handleValidationErrors(
        MethodArgumentNotValidException ex) {

    Map<String, String> errors = new HashMap<>();
    ex.getBindingResult().getFieldErrors().forEach(error ->
        errors.put(error.getField(), error.getDefaultMessage())
    );

    return ResponseEntity
            .badRequest()
```

Entity Relationships - One-to-Many

Example: Category has many Products

```java
@Entity
@Table(name = "categories")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL)
    private List<Product> products = new ArrayList<>();

    // Getters, setters, constructors...
}
```

Entity Relationships - Many-to-One

```java
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Name is required")
    private String name;

    @NotNull(message = "Price is required")
    private Double price;

    private String description;

    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

```

CORS Configuration

CORS = Cross-Origin Resource Sharing

Allows frontend (running on different port) to call your API.

Option 1: Per Controller

```java
@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "http://localhost:3000") // Your frontend URL
public class ProductController {
    // ...
}
```

Option 2: Global Configuration

```java
@Configuration
public class CorsConfig {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**")
                        .allowedOrigins("http://localhost:3000")
                        .allowedMethods("GET", "POST", "PUT", "DELETE");
            }
        };
    }
}
```

Connecting Frontend to Backend

Example: Vanilla JavaScript calling your API

```javascript
// Get all products
fetch('http://localhost:8080/api/products')
  .then(response => response.json())
  .then(products => {
    console.log(products);
    // Display products in HTML
  })
  .catch(error => console.error('Error:', error));
// Create new product
fetch('http://localhost:8080/api/products', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'New Product',
    price: 99.99,
    description: 'Product description'
  })
})
  .then(response => response.json()).then(product => console.log('Created:', product))
  .catch(error => console.error('Error:', error));
```

Part 7: Basic Security

Introduction to Spring Security

What is Spring Security?

Spring Security is a framework that handles authentication and authorization.

Key Concepts:

1. **Authentication:** Who are you? (Login with username/password)
2. **Authorization:** What can you do? (User roles and permissions)

Common Use Cases:

- User login/logout
- Password encryption
- JWT tokens for REST APIs
- Role-based access control (Admin, User, etc.)

Security is a complex topic - we'll cover just the basics today!

Adding Spring Security

Add dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

What happens automatically:

1. All endpoints are now protected
2. Default login page at `/login`
3. Default username: `user`
4. Password: Generated in console logs

```
Using generated security password: a1b2c3d4-e5f6-g7h8-i9j0-k1l2m3n4o5p6
```

This is just basic security - not suitable for production!

Custom Security Configuration

```java
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

Security Configuration (Continued)

```java
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
            throws Exception {
        http
            .csrf().disable() // Disable CSRF for REST APIs
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/products/**").permitAll() // Public
                .anyRequest().authenticated() // Everything else needs login
            )
            .httpBasic(); // Use HTTP Basic Authentication

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
```

Creating Users in Memory

```java
    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.builder()
                .username("user")
                .password(passwordEncoder().encode("password"))
                .roles("USER")
                .build();

        UserDetails admin = User.builder()
                .username("admin")
                .password(passwordEncoder().encode("admin123"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
}
```

Test: Username: `admin`, Password: `admin123`

Role-Based Access Control

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
        throws Exception {
    http
        .csrf().disable()
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/products/search").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/products/**").permitAll()
            .requestMatchers(HttpMethod.POST, "/api/products/**")
                .hasRole("ADMIN")
            .requestMatchers(HttpMethod.PUT, "/api/products/**")
                .hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE, "/api/products/**")
                .hasRole("ADMIN")
            .anyRequest().authenticated()
        )
        .httpBasic();

    return http.build();
}
```

Now only ADMIN can create, update, or delete products. Everyone can view.

Testing Secured Endpoints

With Postman:

1. Select request type (GET, POST, etc.)
2. Go to **Authorization** tab
3. Select **Type:** Basic Auth
4. Enter **Username:** admin
5. Enter **Password:** admin123
6. Send request

With cURL:

```
# Public endpoint (no auth needed)
curl http://localhost:8080/api/products

# Protected endpoint (needs auth)
curl -u admin:admin123 -X POST \
  http://localhost:8080/api/products \
  -H "Content-Type: application/json" \
  -d '{"name":"Laptop","price":999.99}'
```

Spring Profiles

Profiles allow different configurations for different environments:

```properties
# application-dev.properties (Development)
spring.datasource.url=jdbc:mysql://localhost:3306/devdb
spring.jpa.show-sql=true

# application-prod.properties (Production)
spring.datasource.url=jdbc:mysql://production-server:3306/proddb
spring.jpa.show-sql=false
```

Activate profile:

```properties
# application.properties
spring.profiles.active=dev
```

Or via command line:

```
java -jar myapp.jar --spring.profiles.active=prod
```

Best Practices Summary

1. **Layered Architecture:** Controller → Service → Repository
2. **Constructor Injection:** Always use constructor injection for dependencies
3. **DTOs:** Use Data Transfer Objects for API requests/responses (separate from entities)
4. **Exception Handling:** Use @RestControllerAdvice for global exception handling
5. **Validation:** Always validate input with @Valid
6. **Security:** Never expose admin endpoints without authentication
7. **Configuration:** Use profiles for different environments
8. **Logging:** Use proper logging instead of System.out.println
9. **Testing:** Write unit tests for services and integration tests for APIs
10. **Documentation:** Comment your code and use tools like Swagger/OpenAPI

Complete Project Structure

```
src/main/java/com/example/demo/
├── DemoApplication.java
├── config/
│   ├── CorsConfig.java
│   └── SecurityConfig.java
├── controller/
│   └── ProductController.java
├── service/
│   └── ProductService.java
├── repository/
│   └── ProductRepository.java
├── model/
│   ├── Product.java
│   └── Category.java
├── exception/
│   ├── ResourceNotFoundException.java
│   └── GlobalExceptionHandler.java
└── dto/
    ├── ProductRequest.java
    └── ProductResponse.java

src/main/resources/
├── application.properties
```

Common Annotations Cheat Sheet

Annotation	Purpose
`@SpringBootApplication`	Main application class
`@RestController`	REST API controller
`@Service`	Service layer
`@Repository`	Data access layer
`@Entity`	JPA entity (database table)
`@GetMapping`	Handle GET requests
`@PostMapping`	Handle POST requests

HTTP Status Code Reference

Code	Name	When to Use
200	OK	Successful GET/PUT request
201	Created	Successful POST (resource created)
204	No Content	Successful DELETE (no content returned)
400	Bad Request	Invalid input data
401	Unauthorized	Authentication required
403	Forbidden	Authenticated but not authorized
404	Not Found	Resource doesn’t exist

Next Steps

What to Learn Next:

1. **DTO Pattern:** Separate API models from database entities
2. **MapStruct:** Object mapping library
3. **Swagger/OpenAPI:** API documentation
4. **JWT Authentication:** Token-based auth for REST APIs
5. **Testing:** JUnit, Mockito, MockMvc
6. **Database Migrations:** Flyway or Liquibase
7. **Caching:** Spring Cache with Redis
8. **Async Processing:** @Async and message queues
9. **Microservices:** Spring Cloud
10. **Deployment:** Docker, Kubernetes, Cloud platforms

Useful Resources

Official Documentation:

- Spring Boot: https://spring.io/projects/spring-boot
- Spring Data JPA: https://spring.io/projects/spring-data-jpa
- Spring Security: https://spring.io/projects/spring-security

Learning Resources:

- Spring Guides: https://spring.io/guides
- Baeldung: https://www.baeldung.com/spring-boot
- Spring Boot Reference: https://docs.spring.io/spring-boot/docs/current/reference/html/

Tools:

- Spring Initializr: https://start.spring.io/
- Postman: https://www.postman.com/
- MySQL Workbench: https://www.mysql.com/products/workbench/

Quick Reference: Maven Commands

```
# Clean and build
mvn clean install

# Run application
mvn spring-boot:run

# Run tests
mvn test

# Package to JAR
mvn package

# Skip tests during build
mvn clean install -DskipTests

# Run with specific profile
mvn spring-boot:run -Dspring-boot.run.profiles=dev

# Clean Maven cache
mvn dependency:purge-local-repository
```

Troubleshooting Common Issues

Port Already in Use

```
# Find process using port 8080
lsof -i :8080  # Mac/Linux
netstat -ano | findstr :8080  # Windows

# Kill process
kill -9 <PID>  # Mac/Linux
taskkill /PID <PID> /F  # Windows
```

Database Connection Error

1. Check MySQL is running
2. Verify database exists
3. Check username/password
4. Check URL format: `jdbc:mysql://localhost:3306/dbname`

Entity Not Found

1. Check `@Entity` annotation
2. Verify entity is in correct package

Thank You! 🎉

Questions?

Ready to build amazing REST APIs with Spring Boot!

Next Session: Building the Frontend with HTML, CSS, and JavaScript