Java & Spring Boot Complete Study Guide

Table of Contents

Part 1: Java Fundamentals for Spring Boot

1. Core Java Concepts

Variables and Data Types

```java
```

```java
// Primitive types
int age = 25;
double price = 99.99;
boolean isActive = true;
char grade = 'A';

// Reference types
String name = "John Doe";
List<String> items = new ArrayList<>();
```

Methods and Classes

```java
public class Student {
    private String name;
    private int age;

    // Constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter method
    public String getName() {
        return name;
    }

    // Setter method
    public void setName(String name) {
        this.name = name;
    }
}
```

Why Important for Spring Boot: Spring Boot heavily uses classes, objects, and methods. Understanding these basics helps you create entities, services, and controllers.

2. Object-Oriented Programming

Encapsulation

```java
```

```java
public class BankAccount {
    private double balance; // Private field - encapsulated

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

Inheritance

java

```java
// Parent class
public class Animal {
    protected String name;

    public void eat() {
        System.out.println(name + " is eating");
    }
}

// Child class
public class Dog extends Animal {
    public void bark() {
        System.out.println(name + " is barking");
    }
}
```

Interfaces

java

```java
public interface Repository {
    void save(Object entity);
    Object findById(Long id);
}

public class UserRepository implements Repository {
    @Override
    public void save(Object entity) {
        // Implementation
    }

    @Override
    public Object findById(Long id) {
        // Implementation
        return null;
    }
}
```

Why Important for Spring Boot: Spring Boot uses interfaces extensively (like JpaRepository), and inheritance helps organize your code structure.

3. Exception Handling

java

```java
public class UserService {
    public User findUser(Long id) {
        try {
            // Database operation
            return database.findById(id);
        } catch (DataNotFoundException e) {
            throw new UserNotFoundException("User not found with id: " + id);
        } catch (Exception e) {
            throw new ServiceException("Error occurred while finding user");
        }
    }
}

// Custom exception
public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

Why Important for Spring Boot: Exception handling is crucial for REST APIs to return proper error responses to clients.

4. Java Annotations Basics

```java
// Built-in annotations
@Override
public String toString() {
    return "Student: " + name;
}

@Deprecated
public void oldMethod() {
    // This method is deprecated
}

// Custom annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LogExecutionTime {
}
```

Why Important for Spring Boot: Spring Boot is annotation-driven. Understanding how annotations work helps you use Spring Boot effectively.

Part 2: Java Collections Framework

5. Introduction to Collections

Collections are containers that hold multiple objects. They provide a way to store, retrieve, and manipulate groups of objects.

Collection Hierarchy

```
Collection (Interface)
├── List (Interface)
│   ├── ArrayList (Class)
│   ├── LinkedList (Class)
│   └── Vector (Class)
├── Set (Interface)
│   ├── HashSet (Class)
│   ├── LinkedHashSet (Class)
│   └── TreeSet (Class)
└── Queue (Interface)
    ├── LinkedList (Class)
    └── PriorityQueue (Class)

Map (Interface) - Separate hierarchy
├── HashMap (Class)
├── LinkedHashMap (Class)
└── TreeMap (Class)
```

6. List Interface and Implementations

Lists maintain insertion order and allow duplicates.

ArrayList

java

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();

        // Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple"); // Duplicates allowed

        // Accessing elements
        String firstFruit = fruits.get(0); // "Apple"

        // Size
        int size = fruits.size(); // 3

        // Iterating
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

Implementation: Uses dynamic array internally. Good for random access. **Use Case:** When you need fast random access to elements and don't mind slower insertions/deletions in the middle.

LinkedList

```java
```

```java
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<Integer> numbers = new LinkedList<>();

        numbers.add(10);
        numbers.addFirst(5);  // Add at beginning
        numbers.addLast(20);  // Add at end

        // numbers = [5, 10, 20]

        numbers.removeFirst(); // Remove first element
        // numbers = [10, 20]
    }
}
```

Implementation: Uses doubly-linked list internally. **Use Case:** When you frequently add/remove elements at the beginning or end.

7. Set Interface and Implementations

Sets don't allow duplicate elements.

HashSet

java

```java
import java.util.HashSet;
import java.util.Set;

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> uniqueNames = new HashSet<>();

        uniqueNames.add("John");
        uniqueNames.add("Jane");
        uniqueNames.add("John"); // Duplicate - won't be added

        System.out.println(uniqueNames.size()); // 2

        // Check if element exists
        if (uniqueNames.contains("John")) {
            System.out.println("John exists in set");
        }
    }
}
```

Implementation: Uses hash table for O(1) average performance. **Use Case:** When you need unique elements and don't care about order.

TreeSet

java

```java
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> sortedNumbers = new TreeSet<>();

        sortedNumbers.add(30);
        sortedNumbers.add(10);
        sortedNumbers.add(20);

        // Elements are automatically sorted: [10, 20, 30]
        System.out.println(sortedNumbers.first()); // 10
        System.out.println(sortedNumbers.last());  // 30
    }
}
```

Implementation: Uses red-black tree (balanced binary search tree). **Use Case:** When you need unique, sorted elements.

8. Map Interface and Implementations

Maps store key-value pairs.

HashMap

java

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> ageMap = new HashMap<>();

        // Adding key-value pairs
        ageMap.put("John", 25);
        ageMap.put("Jane", 30);
        ageMap.put("Bob", 35);

        // Getting values
        Integer johnAge = ageMap.get("John"); // 25

        // Checking if key exists
        if (ageMap.containsKey("Jane")) {
            System.out.println("Jane's age: " + ageMap.get("Jane"));
        }

        // Iterating through map
        for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        // Or using forEach (Java 8+)
        ageMap.forEach((name, age) ->
            System.out.println(name + " is " + age + " years old"));
    }
}
```

Implementation: Uses hash table with separate chaining. **Use Case:** When you need fast key-value lookups and don't care about order.

TreeMap

java

```java
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Double> grades = new TreeMap<>();

        grades.put("Math", 85.5);
        grades.put("English", 92.0);
        grades.put("Science", 78.5);

        // Keys are automatically sorted alphabetically
        // Output: English, Math, Science
        grades.forEach((subject, grade) ->
            System.out.println(subject + ": " + grade));
    }
}
```

Implementation: Uses red-black tree. **Use Case:** When you need sorted key-value pairs.

9. Queue and Deque

Queue (LinkedList implementation)

java

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Add elements (enqueue)
        queue.offer("First");
        queue.offer("Second");
        queue.offer("Third");

        // Remove elements (dequeue) - FIFO
        String first = queue.poll(); // "First"
        String second = queue.poll(); // "Second"

        // Peek at next element without removing
        String next = queue.peek(); // "Third"
    }
}
```

Use Case: FIFO (First In, First Out) operations like task scheduling.

Part 3: Spring Boot Annotations

10. Understanding Annotations

Annotations are metadata that provide information about the program. In Spring Boot, they tell the framework how to handle your classes and methods.

How Annotations Work

java

```java
// This annotation tells Spring this is a REST controller
@RestController
public class UserController {

    // This annotation maps HTTP GET requests to this method
    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userService.findAll();
    }
}
```

Spring Boot scans for these annotations and automatically configures your application based on them.

11. Core Spring Annotations

@Component, @Service, @Repository

java

```java
// Generic component - Spring will manage this as a bean
@Component
public class EmailValidator {
    public boolean isValid(String email) {
        return email.contains("@");
    }
}

// Business logic layer
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        // Business logic here
        return userRepository.save(user);
    }
}

// Data access layer
@Repository
public class UserRepository {

    @Autowired
    private EntityManager entityManager;

    public User findById(Long id) {
        return entityManager.find(User.class, id);
    }
}
```

How they map to functions:

- `@Component`: Tells Spring to create and manage an instance of this class
- `@Service`: Same as @Component but indicates this handles business logic
- `@Repository`: Same as @Component but indicates this handles data access

@Autowired and @Configuration

```java
```

```java
@Configuration
public class AppConfig {

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

@Service
public class AuthService {

    // Spring automatically injects the PasswordEncoder bean
    @Autowired
    private PasswordEncoder passwordEncoder;

    public String encodePassword(String rawPassword) {
        return passwordEncoder.encode(rawPassword);
    }
}
```

How it works: Spring creates all beans and automatically injects dependencies where needed.

12. Web Layer Annotations

@RestController and @RequestMapping

```java
```

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

    @Autowired
    private UserService userService;

    // GET /api/users
    @GetMapping
    public List<User> getAllUsers() {
        return userService.findAll();
    }

    // GET /api/users/123
    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.findById(id);
    }

    // POST /api/users
    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.save(user);
    }

    // PUT /api/users/123
    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        user.setId(id);
        return userService.update(user);
    }

    // DELETE /api/users/123
    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteById(id);
    }
}
```

How they map to functions:

- `@RestController`: Combines @Controller and @ResponseBody, returns JSON by default
- `@GetMapping`: Maps HTTP GET requests to the method
- `@PostMapping`: Maps HTTP POST requests to the method
- `@PathVariable`: Extracts values from URL path

- `@RequestBody`: Converts JSON request body to Java object

Request Parameters and Validation

```java
@RestController
public class SearchController {

    // GET /search?keyword=spring&page=1&size=10
    @GetMapping("/search")
    public List<Result> search(
            @RequestParam String keyword,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size) {

        return searchService.search(keyword, page, size);
    }

    // POST with validation
    @PostMapping("/users")
    public User createUser(@Valid @RequestBody User user) {
        return userService.save(user);
    }
}

// Entity with validation
@Entity
public class User {
    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Email should be valid")
    private String email;

    @Min(value = 18, message = "Age should be at least 18")
    private int age;
}
```

13. Data Layer Annotations

JPA Annotations

```java
```

```java
@Entity
@Table(name = "users")
public class User {

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

    @Column(name = "full_name", nullable = false, length = 100)
    private String name;

    @Column(unique = true)
    private String email;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Order> orders = new ArrayList<>();

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    // Constructors, getters, setters
}
```

Repository Interface

java

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Spring automatically implements these based on method names
    List<User> findByName(String name);

    List<User> findByEmailContaining(String email);

    List<User> findByAgeGreaterThan(int age);

    // Custom query
    @Query("SELECT u FROM User u WHERE u.email = ?1")
    User findByEmail(String email);

    // Native SQL query
    @Query(value = "SELECT * FROM users WHERE name LIKE %?1%", nativeQuery = true)
    List<User> findByNameLike(String name);
}
```

How it works: Spring Data JPA automatically creates implementations for these methods based on their names.

Part 4: Building CRUD REST API

14. Setting Up Spring Boot Project

Dependencies (pom.xml)

```
xml
```

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
</dependencies>
```

Application Properties

properties

```properties
# Database configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA configuration
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# H2 Console (for testing)
spring.h2.console.enabled=true
```

15. Creating Entity Classes

java

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

import javax.persistence.*;
import javax.validation.constraints.*;
import java.time.LocalDateTime;

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

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

    @NotBlank(message = "Product name is required")
    @Size(min = 2, max = 100, message = "Product name must be between 2 and 100 characters")
    @Column(nullable = false)
    private String name;

    @Size(max = 500, message = "Description cannot exceed 500 characters")
    private String description;

    @NotNull(message = "Price is required")
    @DecimalMin(value = "0.0", inclusive = false, message = "Price must be greater than 0")
    @Column(nullable = false)
    private Double price;

    @NotNull(message = "Quantity is required")
    @Min(value = 0, message = "Quantity cannot be negative")
    @Column(nullable = false)
    private Integer quantity;

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

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    // Constructors
    public Product() {}

    public Product(String name, String description, Double price, Integer quantity) {
        this.name = name;
        this.description = description;
        this.price = price;
        this.quantity = quantity;
```

```
        this.createdAt = LocalDateTime.now();
        this.updatedAt = LocalDateTime.now();
    }

    // Getters and Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

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

    public Double getPrice() { return price; }
    public void setPrice(Double price) { this.price = price; }

    public Integer getQuantity() { return quantity; }
    public void setQuantity(Integer quantity) { this.quantity = quantity; }

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

    public LocalDateTime getUpdatedAt() { return updatedAt; }
    public void setUpdatedAt(LocalDateTime updatedAt) { this.updatedAt = updatedAt; }

    // JPA callbacks
    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
        updatedAt = LocalDateTime.now();
    }

    @PreUpdate
    protected void onUpdate() {
        updatedAt = LocalDateTime.now();
    }
}
```

16. Building REST Controllers

Repository Layer

java

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

import com.example.demo.entity.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;

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

    // Find products by name (case-insensitive)
    List<Product> findByNameContainingIgnoreCase(String name);

    // Find products by price range
    List<Product> findByPriceBetween(Double minPrice, Double maxPrice);

    // Find products in stock (quantity > 0)
    List<Product> findByQuantityGreaterThan(Integer quantity);

    // Custom query - find low stock products
    @Query("SELECT p FROM Product p WHERE p.quantity <= :threshold")
    List<Product> findLowStockProducts(@Param("threshold") Integer threshold);
}
```

Service Layer

java

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

import com.example.demo.entity.Product;
import com.example.demo.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

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

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

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

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

    // Create new product
    public Product createProduct(Product product) {
        return productRepository.save(product);
    }

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

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

        return productRepository.save(product);
    }

    // Delete product
    public void deleteProduct(Long id) {
```

```java
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not found with id: " + id));

        productRepository.delete(product);
    }

    // Search products by name
    public List<Product> searchProductsByName(String name) {
        return productRepository.findByNameContainingIgnoreCase(name);
    }

    // Get products by price range
    public List<Product> getProductsByPriceRange(Double minPrice, Double maxPrice) {
        return productRepository.findByPriceBetween(minPrice, maxPrice);
    }

    // Get products in stock
    public List<Product> getProductsInStock() {
        return productRepository.findByQuantityGreaterThan(0);
    }
}
```

17. Complete CRUD Implementation

Main Controller

java

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

import com.example.demo.entity.Product;
import com.example.demo.service.ProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "*") // Allow cross-origin requests
public class ProductController {

    @Autowired
    private ProductService productService;

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

    // GET /api/products/{id} - Get product by ID
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productService.getProductById(id);

        if (product.isPresent()) {
            return ResponseEntity.ok(product.get());
        } else {
            return ResponseEntity.notFound().build();
        }
    }

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

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

// DELETE /api/products/{id} - 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();
    }
}

// GET /api/products/search?name=productName - Search products
@GetMapping("/search")
public ResponseEntity<List<Product>> searchProducts(@RequestParam String name) {
    List<Product> products = productService.searchProductsByName(name);
    return ResponseEntity.ok(products);
}

// GET /api/products/price-range?min=10&max=100 - Get products by price range
@GetMapping("/price-range")
public ResponseEntity<List<Product>> getProductsByPriceRange(
        @RequestParam Double min,
        @RequestParam Double max) {

    List<Product> products = productService.getProductsByPriceRange(min, max);
    return ResponseEntity.ok(products);
}

// GET /api/products/in-stock - Get products in stock
@GetMapping("/in-stock")
public ResponseEntity<List<Product>> getProductsInStock() {
    List<Product> products = productService.getProductsInStock();
    return ResponseEntity.ok(products);
```

```
    }
}
```

Exception Handling

java

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

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

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    // Handle validation errors
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationExceptions(
            MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });

        return ResponseEntity.badRequest().body(errors);
    }

    // Handle runtime exceptions
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<Map<String, String>> handleRuntimeException(RuntimeException ex) {
        Map<String, String> error = new HashMap<>();
        error.put("error", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }
}
```

Main Application Class

```java
package com.example.demo;

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

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

Testing Your API

Once your application is running, you can test these endpoints:

1. **GET** `http://localhost:8080/api/products` - Get all products
2. **GET** `http://localhost:8080/api/products/1` - Get product by ID
3. **POST** `http://localhost:8080/api/products` - Create product

   ```json
   {
     "name": "Laptop",
     "description": "Gaming laptop",
     "price": 999.99,
     "quantity": 10
   }
   ```

4. **PUT** `http://localhost:8080/api/products/1` - Update product
5. **DELETE** `http://localhost:8080/api/products/1` - Delete product
6. **GET** `http://localhost:8080/api/products/search?name=laptop` - Search products

Key Concepts Summary

1. **Java Fundamentals**: Classes, objects, inheritance, and interfaces form the foundation
2. **Collections**: Lists, Sets, and Maps help manage data efficiently
3. **Annotations**: Tell Spring Boot how to handle your classes and methods
4. **Layered Architecture**: Controller → Service → Repository → Database
5. **Dependency Injection**: Spring automatically manages object creation and dependencies

Advanced Topics for Further Learning

Database Relationships

java

```java
// One-to-Many relationship example
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
    private List<Order> orders = new ArrayList<>();
}

@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private LocalDateTime orderDate;
    private Double totalAmount;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private Customer customer;
}
```

Pagination and Sorting

java

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

    @GetMapping("/paginated")
    public ResponseEntity<Page<Product>> getProductsPaginated(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "id") String sortBy,
            @RequestParam(defaultValue = "asc") String sortDirection) {

        Sort sort = sortDirection.equalsIgnoreCase("desc") ?
                Sort.by(sortBy).descending() :
                Sort.by(sortBy).ascending();

        Pageable pageable = PageRequest.of(page, size, sort);
        Page<Product> products = productService.findAll(pageable);

        return ResponseEntity.ok(products);
    }
}

// Update repository to support pagination
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByNameContainingIgnoreCase(String name, Pageable pageable);
}
```

Custom Response Objects (DTOs)

java

```java
// Data Transfer Object for API responses
public class ProductDTO {
    private Long id;
    private String name;
    private String description;
    private Double price;
    private Integer quantity;
    private String status; // Calculated field

    public ProductDTO(Product product) {
        this.id = product.getId();
        this.name = product.getName();
        this.description = product.getDescription();
        this.price = product.getPrice();
        this.quantity = product.getQuantity();
        this.status = product.getQuantity() > 0 ? "In Stock" : "Out of Stock";
    }

    // Getters and setters
}

// Modified controller method
@GetMapping
public ResponseEntity<List<ProductDTO>> getAllProducts() {
    List<Product> products = productService.getAllProducts();
    List<ProductDTO> productDTOs = products.stream()
            .map(ProductDTO::new)
            .collect(Collectors.toList());
    return ResponseEntity.ok(productDTOs);
}
```

Input Validation and Custom Validators

java

```java
// Custom validation annotation
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PriceValidator.class)
public @interface ValidPrice {
    String message() default "Price must be between 0.01 and 10000";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

// Validator implementation
public class PriceValidator implements ConstraintValidator<ValidPrice, Double> {
    @Override
    public boolean isValid(Double price, ConstraintValidatorContext context) {
        return price != null && price >= 0.01 && price <= 10000.0;
    }
}

// Usage in entity
@Entity
public class Product {
    @ValidPrice
    private Double price;
    // other fields...
}
```

Security Basics

java

```java
// Add Spring Security dependency first
// <dependency>
//     <groupId>org.springframework.boot</groupId>
//     <artifactId>spring-boot-starter-security</artifactId>
// </dependency>

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/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();
    }

    @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("admin"))
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user, admin);
    }

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

```
    }
}
```

Caching

java

```java
// Add caching dependency
// <dependency>
//     <groupId>org.springframework.boot</groupId>
//     <artifactId>spring-boot-starter-cache</artifactId>
// </dependency>

@EnableCaching
@SpringBootApplication
public class ProductApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductApiApplication.class, args);
    }
}

@Service
public class ProductService {

    @Cacheable("products")
    public Product getProductById(Long id) {
        // This method result will be cached
        return productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not found"));
    }

    @CacheEvict(value = "products", key = "#id")
    public void deleteProduct(Long id) {
        // This will remove the cached entry
        productRepository.deleteById(id);
    }

    @CachePut(value = "products", key = "#result.id")
    public Product updateProduct(Long id, Product product) {
        // This will update the cache with new value
        return productRepository.save(product);
    }
}
```

Testing Your API

java

```
// Add testing dependencies
// <dependency>
//     <groupId>org.springframework.boot</groupId>
//     <artifactId>spring-boot-starter-test</artifactId>
//     <scope>test</scope>
// </dependency>

@SpringBootTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@TestPropertySource(locations = "classpath:application-test.properties")
class ProductServiceTest {

    @Autowired
    private ProductService productService;

    @Autowired
    private ProductRepository productRepository;

    @Test
    void shouldCreateProduct() {
        // Given
        Product product = new Product("Test Product", "Description", 99.99, 10);

        // When
        Product savedProduct = productService.createProduct(product);

        // Then
        assertNotNull(savedProduct.getId());
        assertEquals("Test Product", savedProduct.getName());
        assertEquals(99.99, savedProduct.getPrice());
    }

    @Test
    void shouldThrowExceptionWhenProductNotFound() {
        // When & Then
        assertThrows(RuntimeException.class, () -> {
            productService.getProductById(999L);
        });
    }
}

@SpringBootTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class ProductControllerIntegrationTest {

    @Autowired
```

```
    private TestRestTemplate restTemplate;

    @Autowired
    private ProductRepository productRepository;

    @Test
    void shouldGetAllProducts() {
        // Given
        Product product = new Product("Test Product", "Description", 99.99, 10);
        productRepository.save(product);

        // When
        ResponseEntity<Product[]> response = restTemplate.getForEntity("/api/products", Product[].class);

        // Then
        assertEquals(HttpStatus.OK, response.getStatusCode());
        assertTrue(response.getBody().length > 0);
    }

    @Test
    void shouldCreateProduct() {
        // Given
        Product product = new Product("New Product", "Description", 149.99, 5);

        // When
        ResponseEntity<Product> response = restTemplate.postForEntity("/api/products", product, Product.class);

        // Then
        assertEquals(HttpStatus.CREATED, response.getStatusCode());
        assertNotNull(response.getBody().getId());
    }
}
```

Configuration Files

yaml

```yaml
# application.yml (alternative to .properties)
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password: ""

  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: true

  h2:
    console:
      enabled: true

  cache:
    type: simple

server:
  port: 8080
  servlet:
    context-path: /api

logging:
  level:
    com.example.demo: DEBUG
    org.springframework.web: DEBUG
```

Docker Configuration

dockerfile

```
# Dockerfile
FROM openjdk:17-jdk-slim

VOLUME /tmp

COPY target/product-api-1.0.0.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java","-jar","/app.jar"]
```

yaml

```yaml
# docker-compose.yml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    depends_on:
      - db

  db:
    image: mysql:8.0
    environment:
      MYSQL_DATABASE: productdb
      MYSQL_ROOT_PASSWORD: rootpassword
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

Common Patterns and Best Practices

1. Repository Pattern with Custom Implementations

java

```java
// Custom repository interface
public interface CustomProductRepository {
    List<Product> findProductsWithComplexQuery(String criteria);
}

// Implementation
@Repository
public class CustomProductRepositoryImpl implements CustomProductRepository {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Product> findProductsWithComplexQuery(String criteria) {
        String jpql = "SELECT p FROM Product p WHERE p.name LIKE :criteria OR p.description LIKE :criteria";
        return entityManager.createQuery(jpql, Product.class)
                .setParameter("criteria", "%" + criteria + "%")
                .getResultList();
    }
}

// Main repository extending both JpaRepository and custom interface
public interface ProductRepository extends JpaRepository<Product, Long>, CustomProductRepository {
    // Spring Data JPA methods + custom methods
}
```

2. Service Layer Pattern

java

```java
@Service
@Transactional
public class ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;

    // Constructor injection (preferred over @Autowired)
    public ProductService(ProductRepository productRepository, CategoryRepository categoryRepository) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
    }

    @Transactional(readOnly = true)
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    public Product createProductWithCategory(CreateProductRequest request) {
        Category category = categoryRepository.findById(request.getCategoryId())
                .orElseThrow(() -> new CategoryNotFoundException("Category not found"));

        Product product = new Product();
        product.setName(request.getName());
        product.setPrice(request.getPrice());
        product.setCategory(category);

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

3. Exception Hierarchy

java

```java
// Base exception
public abstract class ProductApiException extends RuntimeException {
    public ProductApiException(String message) {
        super(message);
    }

    public ProductApiException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Specific exceptions
public class ProductNotFoundException extends ProductApiException {
    public ProductNotFoundException(String message) {
        super(message);
    }
}

public class InvalidProductDataException extends ProductApiException {
    public InvalidProductDataException(String message) {
        super(message);
    }
}

// Enhanced global exception handler
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(ProductNotFoundException ex) {
        ErrorResponse error = new ErrorResponse("PRODUCT_NOT_FOUND", ex.getMessage());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(error);
    }

    @ExceptionHandler(InvalidProductDataException.class)
    public ResponseEntity<ErrorResponse> handleInvalidData(InvalidProductDataException ex) {
        ErrorResponse error = new ErrorResponse("INVALID_DATA", ex.getMessage());
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
    }
}

// Error response DTO
public class ErrorResponse {
    private String errorCode;
    private String message;
    private LocalDateTime timestamp;
```

```
    public ErrorResponse(String errorCode, String message) {
        this.errorCode = errorCode;
        this.message = message;
        this.timestamp = LocalDateTime.now();
    }

    // Getters and setters
}
```

Learning Path Recommendations

For Beginners:

1. Start with Java basics (variables, methods, classes)
2. Learn OOP concepts thoroughly
3. Practice with Collections Framework
4. Build simple Spring Boot applications
5. Understand REST API concepts
6. Practice CRUD operations

For Intermediate:

1. Learn Spring Security
2. Understand database relationships
3. Practice with different databases (MySQL, PostgreSQL)
4. Learn testing strategies
5. Understand microservices basics
6. Practice with external API integrations

For Advanced:

1. Microservices architecture
2. Message queues (RabbitMQ, Apache Kafka)
3. Caching strategies (Redis)
4. Monitoring and logging
5. Performance optimization
6. Cloud deployment (AWS, Azure, GCP)

Conclusion

This guide provides a comprehensive foundation for understanding Java and Spring Boot development. The key to mastering these technologies is consistent practice and building real projects. Start with simple CRUD applications and gradually add more complexity as you become comfortable with the fundamentals.

Remember:

- **Java fundamentals** are the foundation - invest time in understanding OOP and collections
- **Annotations** are Spring Boot's way of configuration - learn what each one does
- **Layered architecture** keeps your code organized and maintainable
- **Practice** is essential - build projects to reinforce your learning

Good luck with your Spring Boot journey!