Spring With Maven

Introduction to Spring Framework

The **Spring Framework** is a powerful, open-source framework for building enterprise-level Java applications. It provides comprehensive infrastructure support for developing Java applications by addressing common concerns like dependency management, transaction handling, and integration with various technologies.

Spring is widely used because it simplifies Java application development by providing a modular and extensible architecture. It promotes best practices such as **loose coupling, testability, and separation of concerns**.

Overview of the Spring Framework

Spring is designed to provide a **lightweight and modular** approach to Java application development. It enables developers to build applications that are easy to maintain, scalable, and testable.

Spring applications are based on **Plain Old Java Objects (POJOs)**, meaning they do not require extensive configuration or adherence to specific patterns.

Some of the key features of Spring include:

- **Lightweight and modular:** You can use specific parts of Spring as needed.
- **Inversion of Control (IoC):** Manages object creation and dependencies.
- **Aspect-Oriented Programming (AOP):** Separates cross-cutting concerns (e.g., logging, security).
- **Transaction Management:** Handles database transactions efficiently.
- **Integration with other frameworks:** Works well with Hibernate, JPA, JMS, and others.
- **Spring Boot:** An extension of Spring that simplifies application development with minimal configuration.

Inversion of Control (IoC) and Dependency Injection (DI)

Inversion of Control (IoC)

Inversion of Control (IoC) is a design principle that shifts the responsibility of object creation and dependency management from the programmer to the Spring container. Instead of manually creating objects and managing dependencies, Spring does it automatically.

This makes the application more flexible, scalable, and easier to test. IoC is implemented through **Dependency Injection (DI)**.

Dependency Injection (DI)

Dependency Injection (DI) is a technique used to achieve IoC by injecting dependencies into an object rather than allowing the object to create its own dependencies.

Types of Dependency Injection in Spring

1. **Constructor Injection –** Dependencies are provided through a class constructor.
2. **Setter Injection –** Dependencies are injected via setter methods.
3. **Field Injection –** Dependencies are injected directly into fields using @Autowired.

Example of Dependency Injection:

Using **Constructor Injection** in Spring:

```
@Component
class Database {
  public void connect() {
    System.out.println("Connected to Database");
  }
}


@Component
class Application {
  private Database database;

  @Autowired  // Injecting the dependency
  public Application(Database database) {
    this.database = database;
  }
```

```
  public void start() {
    database.connect();
    System.out.println("Application Started");
  }
}

@Configuration
@ComponentScan("com.example")
class AppConfig {}

public class Main {
  public static void main(String[] args) {
    ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
    Application app = context.getBean(Application.class);
    app.start();
  }
}
```

Here, Application does not create its own Database instance. Instead, **Spring injects the dependency**, following IoC principles.

Spring Modules

Spring is a **modular** framework, meaning it consists of several modules that can be used independently or together, depending on the application needs.

1. Spring Core Module

The **Core module** provides the foundation of the framework, including **IoC and DI**. It contains the **BeanFactory** which manages Spring beans and their dependencies.

2. Spring AOP (Aspect-Oriented Programming)

Spring's **AOP module** allows developers to separate **cross-cutting concerns** like logging, transaction management, security, and performance monitoring from business logic.

Example: Using AOP to log method execution time:

```
@Aspect
@Component
class LoggingAspect {
  @Before("execution(* com.example.service.*.*(..))")
  public void logBeforeMethod() {
    System.out.println("Method execution started...");
  }
}
```

This ensures that logging is handled separately from business logic.

3. Spring Data Access Module

Spring provides an abstraction layer for **JDBC (Java Database Connectivity)**, making it easier to interact with databases. It eliminates the need for boilerplate JDBC code and enhances transaction management.

Example of Spring JDBC Template:

```
@Autowired
private JdbcTemplate jdbcTemplate;

public void saveUser(String name, String email) {
  String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
  jdbcTemplate.update(sql, name, email);
}
```

4. Spring ORM (Object-Relational Mapping)

Spring integrates with **Hibernate, JPA, and MyBatis** for ORM support. It simplifies database interaction by managing sessions, transactions, and entity mappings.

Example of Hibernate Integration:

```
@Entity
@Table(name = "users")
class User {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private String email;
}
```

5. Spring MVC (Model-View-Controller)

The **Spring MVC module** is used for building web applications. It follows the MVC design pattern, separating the application logic into **Model (data), View (UI), and Controller (business logic).**

Example of a Spring MVC Controller:

```
@Controller
@RequestMapping("/user")
public class UserController {

  @GetMapping("/hello")
  public String sayHello(Model model) {
    model.addAttribute("message", "Hello, Spring MVC!");
    return "hello";
  }
}
```

Other Important Modules

- **Spring Security:** Provides authentication and authorization.
- **Spring Cloud:** Helps build microservices.
- **Spring Boot:** Simplifies Spring development by providing pre-configured settings.

Benefits of Using Spring in Java Applications

1. **Loose Coupling:** IoC and DI remove dependencies between objects, making the code more maintainable.
2. **Simplified Development:** Spring Boot and its modules reduce boilerplate code.
3. **Testability:** Spring applications are easier to test because of DI.
4. **Integration Support:** Works seamlessly with Hibernate, JPA, JMS, REST, etc.
5. **Aspect-Oriented Programming (AOP):** Helps separate concerns like logging, security, and transactions.
6. **Scalability:** Ideal for building scalable, enterprise-grade applications.
7. **Security:** Built-in security features for authentication and authorization.
8. **Transaction Management:** Reduces the need for manual transaction handling in databases.

Conclusion

Spring Framework is a powerful and flexible framework for Java development. It follows best practices like **Inversion of Control (IoC), Dependency Injection (DI), and Aspect-Oriented Programming (AOP)** to simplify enterprise application development. By using Spring modules like **Core, AOP, ORM, MVC, and Security**, developers can build **robust, scalable, and maintainable applications**.

Would you like me to explain any topic in more detail or provide a hands-on example? 😊

Setting up a Spring Project with Maven

Spring applications often use **Maven** as a build and dependency management tool. Maven simplifies the project setup, manages dependencies, compiles code, runs tests, and packages the application for deployment.

---

1. Introduction to Maven Build Tool

Maven is a powerful build automation and project management tool for Java applications. **I**t follows a **convention-over-configuration** approach, reducing the need for extensive configurations.

Key Features of Maven:

- **Dependency Management:** Automatically downloads required JAR files from repositories.
- **Build Automation:** Compiles, tests, and packages Java applications.
- **Standardized Directory Structure:** Provides a consistent project structure.
- **Plugins Support:** Allows additional functionality like code analysis, reporting, and deployment.
- **Lifecycle Management:** Handles build phases like compile, test, package, and install.

Maven Project Structure:

A typical Maven project follows this structure:

```
my-spring-app/
├-- src/main/java       # Java source code
├-- src/main/resources  # Configuration files (e.g., application.properties)
├-- src/test/java       # Test cases
├-- pom.xml             # Maven configuration file
```

---

2. Creating a New Maven Project

Step 1: Install Maven (If Not Installed)

Before creating a project, ensure that Maven is installed. Run the following command to check the version:

mvn -v

If not installed, download and install Maven from Apache Maven.

Step 2: Create a Maven Project Using Command Line

Run the following command to create a new Spring project:

mvn archetype:generate -DgroupId=com.example -DartifactId=my-spring-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

- groupId: Identifies the project uniquely (e.g., com.example).
- artifactId: The project name (e.g., my-spring-app).
- archetypeArtifactId: Specifies the project type (here, a basic Java application).
- DinteractiveMode=false: Disables interactive mode for automatic setup.

Step 3: Navigate to the Project Directory

cd my-spring-app

---

3. Adding Spring Dependencies in pom.xml

The pom.xml file is the core configuration file in a Maven project. It manages dependencies, plugins, and build settings.

To set up a **Spring project**, add the following dependencies inside <dependencies> in pom.xml:

```
<dependencies>
  <!-- Spring Core -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.20</version>
  </dependency>
```

```
<!-- Spring MVC (For Web Applications) -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.20</version>
</dependency>

<!-- Spring Boot Starter (For Spring Boot Projects) -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <version>2.6.4</version>
</dependency>

<!-- Spring Boot Test (For Unit Testing) -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <version>2.6.4</version>
  <scope>test</scope>
</dependency>

<!-- Spring JDBC (For Database Connectivity) -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.3.20</version>
</dependency>
```

```
    <!-- H2 Database (For Testing Purposes) -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Step 4: Update Dependencies

Run the following command to download and update dependencies:

```
mvn clean install
```

This will download all the required JAR files and store them in the local Maven repository.

---

4. Configuring Maven for Building and Managing Dependencies

Maven Lifecycle Phases

Maven follows a **build lifecycle**, which includes the following key phases:

- **clean –** Removes previous build files.
- **compile –** Compiles Java source files.
- **test –** Runs unit tests.
- **package –** Packages the compiled code into a JAR or WAR file.
- **install** – Installs the package to the local repository.
- **deploy –** Deploys the package to a remote repository.

Building the Spring Project

To **compile** the project, run:

```
mvn compile
```

To **package** the project into a JAR file:

mvn package

To **install** the JAR in the local Maven repository:

mvn install

To **run tests**:

mvn test

---

5. Running a Simple Spring Application

Step 1: Create a Spring Configuration Class

Create a new class AppConfig.java to define Spring beans.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.example")
public class AppConfig {
  @Bean
  public HelloWorld helloWorld() {
    return new HelloWorld();
  }
}
```

Step 2: Create a Simple Spring Bean

Create HelloWorld.java with a simple method:

```
import org.springframework.stereotype.Component;

@Component
public class HelloWorld {
   public void sayHello() {
      System.out.println("Hello, Spring with Maven!");
   }
}
```

Step 3: Create the Main Class

Create Main.java to load the Spring context and run the application.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
      HelloWorld helloWorld = context.getBean(HelloWorld.class);
      helloWorld.sayHello();
   }
}
```

Step 4: Run the Application

Compile and run the application using Maven:

```
mvn compile
mvn exec:java -Dexec.mainClass="com.example.Main"
```

or run the Main class in your IDE.

Expected Output:

Hello, Spring with Maven!

Conclusion

Setting up a **Spring project with Maven** involves:

1. **Installing Maven** and creating a **Maven project**.
2. **Adding Spring dependencies** in pom.xml.
3. **Configuring Maven** for building and managing dependencies.
4. **Creating and running a simple Spring application**.

This approach makes **dependency management, build automation, and project organization** easy, helping developers focus on writing business logic rather than handling configurations manually.

Would you like a more advanced Spring Boot setup or a web application example? 😊

Spring IoC Container

1. Understanding the IoC Container

The **Inversion of Control (IoC) Container** is the core of the **Spring Framework**, responsible for managing the lifecycle and dependencies of objects (**beans**) in a Spring application. It automatically **creates, configures, and manages** beans, ensuring loose coupling and making the application more maintainable.

Key Responsibilities of the IoC Container:

- **Instantiating beans** (objects managed by Spring).
- **Managing dependencies** using **Dependency Injection (DI)**.
- **Configuring beans** through XML, annotations, or Java configuration.
- **Handling the lifecycle** of beans.

Types of IoC Containers in Spring

Spring provides two main types of IoC containers:

1. **BeanFactory** (Lightweight and basic container)
2. **ApplicationContext** (Enhanced container with advanced features)

---

2. Configuring the Spring IoC Container Using XML

Before Spring Boot, XML-based configuration was a common way to define beans. The configuration file, usually named beans.xml, is placed in the src/main/resources folder.

Step 1: Create an XML Configuration File (beans.xml)

Define beans inside <beans> in an XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="<http://www.springframework.org/schema/beans>"
    xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
    xsi:schemaLocation="<http://www.springframework.org/schema/beans>
    <http://www.springframework.org/schema/beans/spring-beans.xsd>">

  <!-- Defining a simple bean -->
  <bean id="helloWorld" class="com.example.HelloWorld">
    <property name="message" value="Hello, Spring IoC!"/>
  </bean>

</beans>
```

---

3. Defining Beans and Their Dependencies

A **bean** is an object managed by the Spring IoC container. Beans can have dependencies, which Spring resolves automatically.

Step 2: Create the Bean Class (HelloWorld.java)

```
package com.example;

public class HelloWorld {
```

```
    private String message;

    // Setter method for dependency injection
    public void setMessage(String message) {
        this.message = message;
    }

    public void printMessage() {
        System.out.println(message);
    }
}
```

Step 3: Load the IoC Container and Access Beans

We use the ClassPathXmlApplicationContext to load the XML configuration.

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        // Load the IoC container from XML configuration
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

        // Retrieve the bean from the container
        HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");

        // Call the method
```

```
        helloWorld.printMessage();
    }
}
```

Expected Output:

Hello, Spring IoC!

---

4. ApplicationContext vs. BeanFactory

Both ApplicationContext and BeanFactory are IoC containers in Spring, but ApplicationContext is more powerful.

Feature	BeanFactory	ApplicationContext
Bean Instantiation	Lazy-loaded	Eagerly loaded
Supports AOP	No	Yes
Event Propagation	No	Yes
Internationalization	No	Yes
Recommended for	Lightweight apps	Enterprise apps

Example Using BeanFactory

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class BeanFactoryExample {
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));
        HelloWorld hello = (HelloWorld) factory.getBean("helloWorld");
        hello.printMessage();
```

```
  }
}
```

However, **BeanFactory is now deprecated**, and ApplicationContext is preferred.

Conclusion

- The **Spring IoC Container** is responsible for managing objects (**beans**) in a Spring application.
- **XML-based configuration** allows defining beans and their dependencies in an external file.
- **ApplicationContext** is preferred over **BeanFactory** because of its additional features.

Would you like an example using **Java-based configuration** instead of XML? 😊

Spring Bean Configuration

Spring provides multiple ways to configure beans in an application:

1. **XML-based Configuration** (Older approach, not recommended for new projects).
2. **Annotation-based Configuration** (Preferred for modern applications).
3. **Java-based Configuration** using @Configuration (Recommended for flexibility and maintainability).
4. **Mixing XML and Java-based Configurations** (For projects migrating from XML to Java).

---

1. Using Annotations for Bean Configuration

Spring allows defining beans directly in Java classes using annotations like @Component, @Service, @Repository, and @Controller. This eliminates the need for XML configuration.

Example: Using @Component to Define a Bean

```
import org.springframework.stereotype.Component;
```

```
@Component // Marks this class as a Spring-managed bean
public class HelloWorld {
  public void printMessage() {
    System.out.println("Hello, Spring with Annotations!");
  }
}
```

Step 2: Enable Component Scanning in AppConfig.java

To automatically detect and register beans, enable **component scanning** in a configuration class.

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.example") // Scans this package for @Component-annotated classes
public class AppConfig {
}
```

Step 3: Load the IoC Container in Main.java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
  public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
    HelloWorld hello = context.getBean(HelloWorld.class);
    hello.printMessage();
```

```
    }
}
```

Output:

Hello, Spring with Annotations!

---

2. Component Scanning and Stereotype Annotations

Spring provides **stereotype annotations** to classify different types of components:

Annotation	Description
@Component	Generic component (used for any Spring-managed bean).
@Service	Used for **service-layer** components.
@Repository	Used for **data-access-layer** (DAO) components.
@Controller	Used for **Spring MVC controllers** (web applications).

Example: Using Different Stereotype Annotations

```
import org.springframework.stereotype.Service;

@Service  // Marks this as a service component
public class UserService {
    public void printUser() {
        System.out.println("User service is working!");
    }
}
```

If @ComponentScan("com.example") is enabled, UserService will be automatically registered in the Spring container.

---

3. Java-Based Configuration with @Configuration

Instead of XML, we can define beans in Java classes using @Bean inside a @Configuration class.

Example: Java-Based Configuration Without @ComponentScan

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration  // Marks this class as a Spring configuration class
public class AppConfig {

   @Bean  // Defines a bean manually
   public HelloWorld helloWorld() {
      return new HelloWorld();
   }
}
```

HelloWorld Class (Without Annotations)

```
public class HelloWorld {
   public void printMessage() {
      System.out.println("Hello, Spring with Java-based Configuration!");
   }
}
```

Main Class (Using Java Configuration)

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
   public static void main(String[] args) {
```

```
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        HelloWorld hello = context.getBean(HelloWorld.class);
        hello.printMessage();
    }
}
```

Output:

Hello, Spring with Java-based Configuration!

---

4. Mixing XML and Java-Based Configurations

For applications migrating from XML to Java configuration, both can be used together.

Step 1: XML Configuration (beans.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="<http://www.springframework.org/schema/beans>"
    xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
    xsi:schemaLocation="<http://www.springframework.org/schema/beans>
    <http://www.springframework.org/schema/beans/spring-beans.xsd>">

    <bean id="helloWorld" class="com.example.HelloWorld"/>
</beans>
```

Step 2: Java-Based Configuration (AppConfig.java)

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
```

```
@ImportResource("classpath:beans.xml")  // Imports XML configuration into Java
public class AppConfig {
}
```

Step 3: Load IoC Container in Main.java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
      HelloWorld hello = (HelloWorld) context.getBean("helloWorld");
      hello.printMessage();
   }
}
```

Output:

```
Hello, Spring with XML!
```

Summary

Configuration Type	Pros	Cons
XML Configuration	Easy to separate configuration from code	Verbose and harder to maintain
Annotation-Based (@Component)	Reduces boilerplate code	Harder to configure without modifying code

Configuration Type	Pros	Cons
Java-Based (@Configuration)	Flexible and maintainable	Requires Java knowledge
Mixed XML & Java	Useful for migration	Can be complex

For new applications, **Java-based configuration** (@Configuration + @Bean) or **annotation-based configuration** (@ComponentScan) is recommended.

Would you like an example of a **Spring Boot-based setup** for a modern Spring application? 🚀

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that allows the Spring IoC container to inject dependencies into a bean rather than the bean creating its own dependencies. This promotes **loose coupling**, better testability, and maintainability.

Spring supports multiple ways to achieve dependency injection:

1. **Constructor Injection**
2. **Setter Injection**
3. **Autowiring Dependencies (@Autowired)**
4. **Using @Qualifier for Resolving Conflicts**
5. **Using @Resource and @Inject Annotations**

---

1. Constructor Injection

With **constructor-based DI**, dependencies are injected via a constructor. This ensures immutability, making it useful when a class requires mandatory dependencies.

Example: Constructor Injection

```
import org.springframework.stereotype.Component;


@Component
public class Engine {
  public void start() {
    System.out.println("Engine started...");
```

```
    }
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Car {
    private Engine engine;

    // Constructor injection
    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving...");
    }
}

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
```

```
@ComponentScan("com.example")
public class AppConfig {
}

public class Main {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
      Car car = context.getBean(Car.class);
      car.drive();
   }
}
```

Output:

Engine started...

Car is moving...

Pros:

✅ Ensures immutability (dependencies are final).

✅ Ideal for **mandatory** dependencies.

✅ Recommended for **dependency injection in immutable objects**.

Cons:

❌ Becomes complex if there are too many dependencies.

---

2. Setter Injection

Setter-based injection allows dependencies to be injected after object creation, making it useful for **optional** dependencies.

Example: Setter Injection

```
import org.springframework.stereotype.Component;

@Component
public class Engine {
    public void start() {
        System.out.println("Engine started...");
    }
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Car {
    private Engine engine;

    // Setter method for dependency injection
    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving...");
    }
}
```

Pros:

✅ Suitable for **optional** dependencies.

✅ More readable when there are **multiple dependencies**.

Cons:

❌ The object can be **partially constructed** before dependencies are injected, which may lead to NullPointerException.

---

3. Autowiring Dependencies (@Autowired)

Spring allows automatic dependency resolution using **@Autowired**.

Autowiring Strategies:

- **By Type:** Matches a bean by its type (@Autowired private Engine engine;)
- **By Constructor:** Uses constructor-based injection (@Autowired Car(Engine engine))
- **By Setter:** Uses setter-based injection (@Autowired public void setEngine(Engine engine))

---

4. Using @Qualifier for Resolving Conflicts

If multiple beans of the same type exist, Spring does not know which one to inject. We can resolve this using **@Qualifier**.

Example: Using @Qualifier to Specify a Bean

```
import org.springframework.stereotype.Component;


@Component("petrolEngine")
public class PetrolEngine implements Engine {
  public void start() {
    System.out.println("Petrol engine started...");
  }
}
```

```
@Component("dieselEngine")
public class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started...");
    }
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
public class Car {
    private Engine engine;

    @Autowired
    public Car(@Qualifier("petrolEngine") Engine engine) { // Specify which bean to inject
        this.engine = engine;
    }

    public void drive() {
        engine.start();
        System.out.println("Car is moving...");
    }
}
```

Output:

Petrol engine started...

Car is moving...

Why use @Qualifier?

✅ Useful when multiple beans of the same type exist.

✅ Helps avoid **NoUniqueBeanDefinitionException**.

---

5. Using @Resource and @Inject Annotations

Spring also supports @Resource and @Inject, which come from **Java's JSR-250 and JSR-330 specifications**, respectively.

@Resource (JSR-250)

@Resource is similar to @Autowired but allows injection by **name instead of type**.

```
import javax.annotation.Resource;

import org.springframework.stereotype.Component;


@Component

public class Car {

  @Resource(name = "petrolEngine")  // Injecting by bean name

  private Engine engine;


  public void drive() {

    engine.start();

    System.out.println("Car is moving...");

  }

}
```

@Inject (JSR-330)

@Inject is similar to @Autowired, but it does not support required=false.

```
import javax.inject.Inject;
```

```
import org.springframework.stereotype.Component;

@Component
public class Car {
   @Inject  // Works like @Autowired
   private Engine engine;

   public void drive() {
      engine.start();
      System.out.println("Car is moving...");
   }
}
```

Difference Between @Autowired, @Inject, and @Resource

Annotation	Provided By	Injection Type	Can Be Used on Constructor?
@Autowired	Spring	By Type	✅ Yes
@Inject	JSR-330 (Jakarta EE)	By Type	✅ Yes
@Resource	JSR-250 (Jakarta EE)	By Name	❌ No

Summary

Injection Type	Best Use Case
Constructor Injection	When dependencies are mandatory and should not be changed.
Setter Injection	When dependencies are optional or changeable.
@Autowired Annotation	For automatic dependency injection (by type).
@Qualifier Annotation	When multiple beans of the same type exist.

Injection Type	Best Use Case
@Resource Annotation	For dependency injection by name.
@Inject Annotation	Java EE alternative to @Autowired.

For **modern Spring applications**, **constructor injection** with @Autowired is preferred for **mandatory dependencies**, while **setter injection** is useful for **optional dependencies**.

Would you like me to demonstrate a **real-world example** combining these approaches? 🚀

Spring AOP (Aspect-Oriented Programming)

1. Introduction to AOP Concepts

Aspect-Oriented Programming (AOP) is a programming paradigm that enables separation of **cross-cutting concerns** (such as logging, security, transactions) from business logic.

Key Concepts in AOP:

Concept	Description
Aspect	A module that contains logic for a **cross-cutting concern** (e.g., logging, security).
Advice	The action taken by an **Aspect** (e.g., before, after method execution).
Joinpoint	A **point** in the execution flow where an **Aspect** can be applied (e.g., method execution).
Pointcut	A **predicate** that selects Joinpoints (e.g., all methods in a package).
Weaving	The process of linking **aspects** with **business logic**.

---

2. Creating Aspects and Advice

Spring AOP allows defining **aspects** using @Aspect and **advice** methods (@Before, @After, etc.).

Example: Logging Aspect (Advice Before Method Execution)

Step 1: Define the Business Service

import org.springframework.stereotype.Service;

```
@Service
public class PaymentService {
   public void processPayment() {
      System.out.println("Processing Payment...");
   }
}
```

Step 2: Define the Aspect (Logging Logic)

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

   @Before("execution(* com.example.PaymentService.processPayment(..))")
   public void logBeforeMethod() {
      System.out.println("Logging: Payment method is about to be executed.");
   }
}
```

Step 3: Configure Spring for AOP Support

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
```

```
@Configuration
@EnableAspectJAutoProxy  // Enables AOP proxying
@ComponentScan("com.example")
public class AppConfig {
}
```

Step 4: Run the Application

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
   public static void main(String[] args) {
      ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
      PaymentService paymentService = context.getBean(PaymentService.class);
      paymentService.processPayment();
   }
}
```

Output:

```
Logging: Payment method is about to be executed.
Processing Payment...
```

---

3. Pointcuts and Joinpoints

Pointcuts: Defining Where Aspects Should Apply

Spring AOP provides different **pointcut expressions**:

Expression	Description
execution(* package.Class.method(..))	Matches method execution
within(com.example.*)	Matches all methods inside a package
@annotation(org.springframework.transaction.annotation.Transactional)	Matches methods with a specific annotation
bean(paymentService)	Matches beans with a specific name

Example: Pointcut Matching All Methods in PaymentService

```
@Aspect
@Component
public class LoggingAspect {

  @Before("execution(* com.example.PaymentService.*(..))")
  public void logBeforeMethods() {
    System.out.println("Logging: A method in PaymentService is being executed.");
  }
}
```

---

4. AOP Proxying Mechanisms

Spring uses **proxy objects** to implement AOP in two ways:

1. **JDK Dynamic Proxy** (Interface-based)
2. **CGLIB Proxy** (Class-based, when no interface exists)

Spring automatically selects the appropriate proxy mechanism.

- If a class implements an **interface**, Spring uses **JDK dynamic proxy**.
- If no interface is present, Spring uses **CGLIB (a subclass-based proxy mechanism)**.

To **force CGLIB proxying**, set:

@EnableAspectJAutoProxy(proxyTargetClass = true)

---

5. Integrating AOP with Spring Applications

AOP is commonly used for:

✅ **Logging**

✅ **Security**

✅ **Transaction Management**

✅ **Performance Monitoring**

✅ **Caching**

Example: Transaction Management Using AOP

```
import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;


@Aspect

@Component

public class TransactionAspect {


  @Before("execution(* com.example.BankService.transferMoney(..))")

  public void startTransaction() {
```

```
        System.out.println("Transaction Started...");
    }
}
```

Output:

Transaction Started...

Executing transferMoney() method...

Summary

Concept	Description
Aspect	Defines cross-cutting concerns (e.g., LoggingAspect).
Advice	Action executed before/after a method (@Before, @After).
Joinpoint	A point in the application (e.g., method execution).
Pointcut	Specifies which joinpoints an aspect applies to.
Proxying	Uses JDK Dynamic Proxy or CGLIB.

When to Use Spring AOP?

- When **cross-cutting concerns** need to be **modularized**.
- When you need to add **logging, security, or transactions** dynamically.

Would you like a **Spring Boot example** with AOP and logging? 🚀

Spring MVC and ORM: A Comprehensive Guide

Spring MVC (Model-View-Controller) is a framework used for building **web applications** in Java, while **ORM (Object-Relational Mapping)** allows interaction with databases using objects instead of SQL queries.

---

1. Overview of MVC and ORM

Spring MVC (Model-View-Controller)

Spring MVC is based on the **MVC pattern**, which separates concerns in a web application:

- **Model** → Manages data and business logic (e.g., User, Order).
- **View** → Handles UI presentation (e.g., JSP, Thymeleaf).
- **Controller** → Handles HTTP requests and delegates tasks to services.

Spring ORM (Object-Relational Mapping)

Spring integrates ORM frameworks like:

- **Hibernate** → Most widely used ORM framework in Java.
- **JPA (Java Persistence API)** → Standardized ORM for Java applications.

Spring provides Spring Data JPA to simplify database interactions.

---

2. Spring MVC Configuration

To set up a **Spring MVC project**, you need:

1. A **Spring Boot starter project** with dependencies.
2. A **Spring configuration class** (@Configuration).
3. A **Controller, Model, and View**.

Dependencies (Spring Boot + Spring MVC + Hibernate)

Add the following dependencies in pom.xml:

```
<dependencies>
  <!-- Spring Boot Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot JPA (for ORM) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- H2 Database (For Testing) -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <!-- Thymeleaf (For View Layer)  -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
</dependencies>
```

---

3. Spring MVC Components (Controller, Model, View)

3.1 Controller Layer (Handles Requests)

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

  @GetMapping("/")
```

```
    public String home(Model model) {
        model.addAttribute("message", "Welcome to Spring MVC!");
        return "home";  // Returns "home.html" from templates
    }
}
```

3.2 Model Layer (Entity + Repository + Service)

Defining an Entity

```
import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Getters and Setters
}
```

Repository (DAO Layer)

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
```

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

Service Layer (Business Logic)

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {

  @Autowired
  private UserRepository userRepository;

  public List<User> getAllUsers() {
    return userRepository.findAll();
  }
}
```

3.3 View Layer (Thymeleaf Template)

home.html (Inside src/main/resources/templates/)

```
<!DOCTYPE html>
<html xmlns:th="<http://www.thymeleaf.org>">
<head>
  <title>Home</title>
</head>
```

```
<body>
  <h1 th:text="${message}"></h1>
</body>
</html>
```

---

4. Form Handling in Spring MVC

Spring MVC provides easy **form handling** with model binding.

Controller for Handling Form Submission

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/users")
public class UserController {

  @GetMapping("/add")
  public String showForm(Model model) {
    model.addAttribute("user", new User());
    return "user-form";
  }

  @PostMapping("/save")
  public String saveUser(@ModelAttribute User user) {
    userService.save(user);
    return "redirect:/users";
  }
```

```
}
```

Thymeleaf Form (user-form.html)

```
<form th:action="@{/users/save}" method="post" th:object="${user}">
  <input type="text" th:field="*{name}" placeholder="Enter Name" required/>
  <input type="email" th:field="*{email}" placeholder="Enter Email" required/>
  <button type="submit">Save User</button>
</form>
```

---

5. Querying with Spring Data JPA

Spring provides a simple way to query databases.

Find Users by Name

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
  List<User> findByName(String name);
}
```

```
public List<User> searchUsers(String name) {
  return userRepository.findByName(name);
}
```

---

6. Validation in Spring MVC

Spring provides **Java Bean Validation (JSR-303)** for input validation.

Add Validation Annotations to the Model

```
import jakarta.validation.constraints.*;
```

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

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

  @Email(message = "Invalid email format")
  private String email;

  // Getters and Setters
}
```

Enable Validation in the Controller

```
import jakarta.validation.Valid;
import org.springframework.validation.BindingResult;

@PostMapping("/save")
public String saveUser(@Valid @ModelAttribute User user, BindingResult result) {
  if (result.hasErrors()) {
    return "user-form";  // Return to form if errors exist
  }
  userService.save(user);
  return "redirect:/users";
}
```

---

7. Exception Handling in Spring MVC

Spring provides **Global Exception Handling** using @ControllerAdvice.

Create a Global Exception Handler

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


@ControllerAdvice
public class GlobalExceptionHandler {


   @ExceptionHandler(Exception.class)
   public String handleException(Exception ex, Model model) {
      model.addAttribute("error", ex.getMessage());
      return "error";
   }
}
```

error.html (Inside templates/)

```
<h2>Error Occurred: <span th:text="${error}"></span></h2>
```

---

8. Summary of Key Concepts

Feature	**Spring MVC**	**Spring ORM**
Model	Represents business objects	Entities mapped to DB tables
View	HTML/JSP/Thymeleaf	Not applicable
Controller	Handles HTTP requests	Not applicable
Repository (DAO)	Not required	Uses JpaRepository
Form Handling	@ModelAttribute, @PostMapping	Not applicable

Feature	Spring MVC	Spring ORM
Querying	Not applicable	Uses JPA Methods (findByName())
Validation	@Valid, @NotBlank	Bean Validation (JSR-303)
Exception Handling	@ControllerAdvice, @ExceptionHandler	Not applicable

Would you like a **working GitHub project** with this setup? 🚀

Introduction to Spring Boot

1. Overview of Spring Boot

Spring Boot is a **framework** built on top of **Spring** that simplifies application development by providing:

✅ **Auto-configuration** (No need for complex XML setup)

✅ **Embedded servers** (Tomcat, Jetty, etc., included by default)

✅ **Standalone applications** (Run with java -jar)

✅ **Production-ready features** (Monitoring, logging, etc.)

Why Use Spring Boot?

- **Eliminates boilerplate code** for Spring applications.
- **Reduces manual configuration** using sensible defaults.
- **Quick application startup** with embedded web servers.
- **Easy integration** with databases, security, and cloud platforms.

---

2. Simplifying Spring Configuration with Boot

Before Spring Boot, configuring a Spring project required:

- **XML configuration** for bean definitions.
- **Manual dependency management** in pom.xml.
- **Setting up a web server separately** (Tomcat, Jetty).

With **Spring Boot**, all of this is simplified using **Auto-configuration** and **Starter dependencies**.

Example: Traditional Spring vs. Spring Boot

Without Spring Boot (Traditional Spring)

```
<!-- Traditional Spring requires a lot of manual dependencies -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
</dependency>
```

With Spring Boot (Simplified)

```
<!-- Spring Boot Starter Web automatically includes necessary dependencies -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot **auto-configures** everything, reducing manual setup.

---

3. Creating a Spring Boot Application

Step 1: Setup a Spring Boot Project

Use **Spring Initializr** (**https://start.spring.io/**) or create a Maven project manually.

Step 2: Add Dependencies (pom.xml)

```
<dependencies>
  <!-- Spring Boot Web Starter (For REST API & MVC) -->
  <dependency>
```

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

  <!-- Spring Boot Starter for JPA (Database Support) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- H2 Database (In-memory DB for testing) -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

Step 3: Create the Main Application Class

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

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

@SpringBootApplication enables Auto-Configuration, Component Scanning, and Configuration.

Step 4: Create a REST Controller

```
import org.springframework.web.bind.annotation.GetMapping;

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

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


@RestController

@RequestMapping("/api")

public class HelloController {


  @GetMapping("/hello")

  public String sayHello() {

    return "Hello, Spring Boot!";

  }

}
```

Step 5: Run the Application

Run with:

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

or

```
java -jar target/my-spring-boot-app.jar
```

Step 6: Test the API

Open your browser or Postman and visit:

<http://localhost:8080/api/hello>

You should see:

Hello, Spring Boot!

---

4. Auto-Configuration & Convention over Configuration

Auto-Configuration

Spring Boot **automatically configures** required components based on dependencies.

For example:

- If spring-boot-starter-web is added → Embedded **Tomcat** is auto-configured.
- If spring-boot-starter-data-jpa is added → **Hibernate & Datasource** are auto-configured.

Example: Configuring a Database in application.properties

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

No need to write Hibernate or JDBC configurations manually! 🎉

---

5. Summary

Feature	**Traditional Spring**	**Spring Boot**
Configuration	Manual XML/Java config	Auto-configured
Dependencies	Manually managed	Uses **Starter Dependencies**
Server Setup	External Tomcat	Embedded Tomcat
Application Startup	Complex setup	java -jar

Why Spring Boot?

✅ **Rapid Development**

✅ **Less Configuration**

✅ **Microservices Ready**

✅ **Production-Ready Features**

Would you like a **Spring Boot + Database CRUD Example**? 🚀

Reactive Programming with Spring WebFlux

1. Overview of Reactive Web Framework and Non-blocking I/O

What is Reactive Programming?

Reactive Programming is an **asynchronous, event-driven** programming model that handles data streams efficiently. It is designed to process large volumes of data with minimal resource consumption.

Why Use Reactive Programming?

✅ **Handles concurrent requests efficiently**

✅ **Uses fewer threads compared to traditional blocking models**

✅ **Improves performance in high-load applications**

✅ **Best suited for microservices, streaming data, and NoSQL databases**

Spring WebFlux vs Spring MVC

Feature	**Spring MVC (Blocking)**	**Spring WebFlux (Non-blocking)**
Programming Model	Synchronous (Thread per request)	Asynchronous (Event-driven)
Threading Model	Uses Servlet API (Blocking)	Uses Reactive Streams (Non-blocking)
Performance	Higher resource consumption	Efficient with low memory footprint
Best Used For	Traditional web applications	Real-time, streaming, or high-throughput applications

---

2. Creating and Configuring Reactive Controllers

Adding Dependencies in pom.xml

To enable Spring WebFlux, add:

```
<dependencies>
  <!-- Spring Boot WebFlux Starter -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
  </dependency>

  <!-- Spring Boot Starter for Reactive MongoDB -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
  </dependency>
</dependencies>
```

Creating a Reactive Controller

Spring WebFlux provides @RestController with Flux and Mono for reactive request handling.

```
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
@RequestMapping("/api/reactive")
public class ReactiveController {
```

```
    @GetMapping("/hello")
    public Mono<String> sayHello() {
        return Mono.just("Hello, Reactive World!");
    }

    @GetMapping("/numbers")
    public Flux<Integer> getNumbers() {
        return Flux.range(1, 10);  // Streams numbers 1 to 10
    }
}
```

Understanding Mono and Flux

Return Type	Description	Example
Mono	Emits **one** or **zero** item	Mono.just("Hello")
Flux	Emits **multiple** items	Flux.range(1, 10)

---

3. Handling HTTP Requests and Responses with Mono and Flux

Service Layer with Reactive Streams

```
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class ReactiveService {

    public Mono<String> getGreeting() {
        return Mono.just("Welcome to Reactive Spring!");
```

```
    }

    public Flux<String> getNames() {
        return Flux.just("Alice", "Bob", "Charlie");
    }
}
```

Controller Using Service

```
@RestController
@RequestMapping("/api/service")
public class ReactiveServiceController {

    private final ReactiveService reactiveService;

    public ReactiveServiceController(ReactiveService  reactiveService) {
        this.reactiveService = reactiveService;
    }

    @GetMapping("/greeting")
    public Mono<String> greeting() {
        return reactiveService.getGreeting();
    }

    @GetMapping("/names")
    public Flux<String> names() {
        return reactiveService.getNames();
    }
}
```

✅ **Mono and Flux make APIs non-blocking** by handling data asynchronously.

---

4. Integrating with NoSQL Databases Using Reactive Repositories

Spring Data provides **reactive repositories** for databases like **MongoDB, Cassandra, and Redis**.

1. Configure MongoDB in application.properties

```
spring.data.mongodb.uri=mongodb://localhost:27017/reactive_db
```

2. Create a Reactive MongoDB Entity

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "products")
public class Product {
  @Id
  private String id;
  private String name;
  private double price;

  // Constructors, Getters, Setters
}
```

3. Create a Reactive Repository

```
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Flux;
```

```
public interface ProductRepository extends ReactiveMongoRepository<Product, String>
{
   Flux<Product> findByPriceBetween(double min, double max);
}
```

4. Create a Reactive Service for MongoDB

```
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class ProductService {

   private final ProductRepository productRepository;

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

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

   public Mono<Product> getProductById(String id) {
      return productRepository.findById(id);
   }

   public Mono<Product> saveProduct(Product product) {
```

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

5. Create a Reactive REST API Controller

```
import org.springframework.web.bind.annotation.*;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

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

    private final ProductService productService;

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

    @GetMapping
    public Flux<Product> getAllProducts() {
        return productService.getAllProducts();
    }

    @GetMapping("/{id}")
    public Mono<Product> getProduct(@PathVariable String id) {
        return productService.getProductById(id);
    }
```

```
    @PostMapping
    public Mono<Product> createProduct(@RequestBody Product product) {
        return productService.saveProduct(product);
    }
}
```

✅ **Now, MongoDB handles data reactively without blocking threads.**

---

5. Writing Unit and Integration Tests for Reactive Services

Unit Test for Service Layer

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import reactor.test.StepVerifier;

public class ProductServiceTest {

    private final ProductRepository productRepository =
Mockito.mock(ProductRepository.class);
    private final ProductService productService = new
ProductService(productRepository);

    @Test
    void testGetAllProducts() {
        Mockito.when(productRepository.findAll())
                .thenReturn(Flux.just(new Product("1", "Laptop", 800.0)));
```

```
    StepVerifier.create(productService.getAllProducts())
        .expectNextMatches(product -> product.getName().equals("Laptop"))
        .verifyComplete();
  }

  @Test
  void testGetProductById() {
    Mockito.when(productRepository.findById("1"))
        .thenReturn(Mono.just(new Product("1", "Phone", 500.0)));

    StepVerifier.create(productService.getProductById("1"))
        .expectNextMatches(product -> product.getName().equals("Phone"))
        .verifyComplete();
  }
}
```

✅ **StepVerifier is used to test reactive streams.**

---

6. Summary of Key Concepts