**Spring Bean Scopes with Real-Time Examples**

In Spring, **scopes** define the lifecycle and visibility of a bean. There are **five major scopes** in Spring:

1. **Singleton** (Default)
2. **Prototype**
3. **Request** (Only for Web Applications)
4. **Session** (Only for Web Applications)
5. **Application** (Only for Web Applications)

**1. Singleton Scope (Default)**

**Definition**

* Only **one instance** of the bean is created for the entire Spring container.
* **Same instance** is shared across all requests.

**Real-Time Example: Logging Service**

A logging service should use the **same instance** throughout the application.

package com.example.service;

import lombok.extern.slf4j.Slf4j;

import org.springframework.stereotype.Service;

@Slf4j

@Service // Default scope is Singleton

public class LoggingService {

public LoggingService() {

log.info("LoggingService instance created");

}

public void logMessage(String message) {

log.info("Log: {}", message);

}

}

**Test Singleton Behavior**

@RestController

@RequestMapping("/api")

public class LogController {

@Autowired

private LoggingService loggingService1;

@Autowired

private LoggingService loggingService2;

@GetMapping("/log")

public String logTest() {

loggingService1.logMessage("Hello from loggingService1");

loggingService2.logMessage("Hello from loggingService2");

return loggingService1 == loggingService2 ? "Same instance" : "Different instances";

}

}

✅ **Output:** Same instance  
*(Only one object is created)*

**2. Prototype Scope**

**Definition**

* A **new instance** of the bean is created **every time** it is requested.

**Real-Time Example: Random Number Generator**

Each user should get a **new random number instance**.

package com.example.service;

import org.springframework.stereotype.Service;

import java.util.Random;

import org.springframework.context.annotation.Scope;

@Service

@Scope("prototype") // Creates a new instance every time

public class RandomNumberService {

private final int randomNumber;

public RandomNumberService() {

this.randomNumber = new Random().nextInt(100);

}

public int getRandomNumber() {

return randomNumber;

}

}

**Test Prototype Behavior**

@RestController

@RequestMapping("/api")

public class RandomController {

@Autowired

private RandomNumberService randomService1;

@Autowired

private RandomNumberService randomService2;

@GetMapping("/random")

public String randomTest() {

return "Random1: " + randomService1.getRandomNumber() +

", Random2: " + randomService2.getRandomNumber();

}

}

✅ **Output:**

Random1: 42, Random2: 17

*(Different instances are created)*

**3. Request Scope (Web Only)**

**Definition**

* A **new instance** of the bean is created **for each HTTP request**.

**Real-Time Example: Tracking Request ID**

Each request should have **a unique ID**.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import java.util.UUID;

@Component

@Scope("request") // Creates a new instance per request

public class RequestTracker {

private final String requestId;

public RequestTracker() {

this.requestId = UUID.randomUUID().toString();

}

public String getRequestId() {

return requestId;

}

}

**Test Request Scope**

@RestController

@RequestMapping("/api")

public class RequestController {

@Autowired

private RequestTracker requestTracker;

@GetMapping("/request")

public String requestTest() {

return "Request ID: " + requestTracker.getRequestId();

}

}

✅ **Test with Multiple Requests**

GET /api/request → Request ID: abc-123

GET /api/request → Request ID: xyz-456

*(Each request has a different ID)*

**4. Session Scope (Web Only)**

**Definition**

* A **new instance** of the bean is created **per user session**.

**Real-Time Example: Shopping Cart**

Each user should have **a separate cart**.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

import java.util.ArrayList;

import java.util.List;

@Component

@Scope("session") // Creates a new instance per session

public class ShoppingCartService {

private final List<String> items = new ArrayList<>();

public void addItem(String item) {

items.add(item);

}

public List<String> getItems() {

return items;

}}

**Test Session Scope**

@RestController

@RequestMapping("/api/cart")

public class CartController {

@Autowired

private ShoppingCartService shoppingCart;

@GetMapping("/add/{item}")

public String addItem(@PathVariable String item) {

shoppingCart.addItem(item);

return "Item added: " + item;

}

@GetMapping("/items")

public List<String> getCartItems() {

return shoppingCart.getItems();

}}

✅ **Test with Same User**

GET /api/cart/add/apple → "Item added: apple"

GET /api/cart/add/mango → "Item added: mango"

GET /api/cart/items → ["apple", "mango"]

*(Same session retains items)*

✅ **Test with Different Users**

User 1: GET /api/cart/items → ["apple", "mango"]

User 2: GET /api/cart/items → []

*(Each session has a different cart)*

**5. Application Scope (Web Only)**

**Definition**

* A **single instance** is shared **across all sessions** and **requests**.

**Real-Time Example: Site Visit Counter**

The **same counter** should be shared across users.

package com.example.service;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Component;

@Component

@Scope("application") // Shared across all users

public class VisitCounterService {

private int counter = 0;

public void increment() {

counter++;

}

public int getCount() {

return counter;

}

}

**Test Application Scope**

@RestController

@RequestMapping("/api/visit")

public class VisitController {

@Autowired

private VisitCounterService visitCounter;

@GetMapping("/increment")

public String incrementVisit() {

visitCounter.increment();

return "Total Visits: " + visitCounter.getCount();

}

}

✅ **Test Across Users**

User 1: GET /api/visit/increment → Total Visits: 1

User 2: GET /api/visit/increment → Total Visits: 2

User 3: GET /api/visit/increment → Total Visits: 3

*(Shared counter across all users)*

**Summary of Scopes**

| **Scope** | **Instance Created** | **Lifecycle** | **Real-Time Example** |
| --- | --- | --- | --- |
| **Singleton** | **One** per container | App startup to shutdown | Logging Service |
| **Prototype** | **New** per request | Created & destroyed immediately | Random Number Generator |
| **Request** | **New** per HTTP request | Destroyed after request | Request Tracker |
| **Session** | **New** per user session | Destroyed after session | Shopping Cart |
| **Application** | **One** per app | App startup to shutdown | Site Visit Counter |

**Final Thoughts**

* Use **Singleton** for shared services (e.g., Logging).
* Use **Prototype** for frequently changing data (e.g., Random Numbers).
* Use **Request** for per-request data (e.g., Request ID).
* Use **Session** for user-specific data (e.g., Shopping Cart).
* Use **Application** for shared app-wide data (e.g., Visitor Count).

### ****❓ Why Can't We Use**** @Repository ****Instead of**** @Service ****and Vice Versa?****

Technically, **Spring does not enforce restrictions** on using @Repository or @Service. However, **each annotation serves a specific purpose**, and misusing them **affects maintainability, readability, and behavior**.

## ****1️⃣**** @Repository ****vs.**** @Service ****- Purpose and Differences****

| **Annotation** | **Purpose** | **Special Features** |
| --- | --- | --- |
| **@Repository** | Used for **DAO (Data Access Object) Layer** to interact with the database | **Automatic translation of JDBC exceptions** into Spring DataAccessException |
| **@Service** | Used for **Business Logic Layer** where business rules and operations are implemented | No extra behavior, just for readability and separation of concerns |

## ****2️⃣ Why Not Use**** @Repository ****Instead of**** @Service****?****

🚫 **Problem:** @Repository is meant for **data access**, not business logic.

✅ **Example (Correct Use of @Repository and @Service)**

@Repository

public class UserRepository {

public User findById(Long id) {

// Fetch user from DB (Pretend logic)

return new User(id, "Devendra");

}

}

@Service

public class UserService {

private final UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

public String getUserGreeting(Long id) {

User user = userRepository.findById(id);

return "Hello, " + user.getName();

}

}

✅ **Correct separation**: UserRepository handles **DB operations**, UserService contains **business logic**.

🚫 **If We Use @Repository on UserService**

@Repository // ❌ Wrong annotation, because this is not a DAO class

public class UserService { ... }

🔴 **Issues:**

* Misleads developers because **it is not a DAO**.
* Exception handling behavior of @Repository (JDBC exception translation) **is unnecessary** in business logic.

## ****3️⃣ Why Not Use**** @Service ****Instead of**** @Repository****?****

🚫 **Problem:** @Service **does not provide exception translation** for DAO operations.

✅ **Example (Correct Use of @Repository)**

@Repository

public class ProductRepository {

public Product findById(Long id) {

// Code to fetch product from DB

return new Product(id, "Laptop");

}

}

🚫 **If We Use @Service on ProductRepository**

@Service // ❌ Wrong annotation, because this is a DAO

public class ProductRepository { ... }

🔴 **Issues:**

* **No automatic JDBC exception translation** (DataAccessException handling is lost).
* Code structure becomes unclear.

## ****4️⃣ Key Takeaways****

✅ @Repository → **Use for DAO layer** (Exception Translation)  
✅ @Service → **Use for Business Logic layer** (No Extra Features)  
🚫 **Do not interchange them**, as it reduces code clarity and expected behavior.

**POJO (Plain Old Java Object) in Spring**

**1. What is a POJO?**

POJO stands for **Plain Old Java Object**, which is a simple Java class that follows certain basic principles:

* It should not extend specific classes (like HttpServlet or EJB classes).
* It should not implement special interfaces (except standard Java interfaces like Serializable).
* It should have private fields with public getter and setter methods.
* It may have constructors, methods, and annotations but should remain a simple data container.

**2. Why Use POJOs in Spring?**

Spring heavily relies on POJOs because:

* They make the application more **loosely coupled** and **maintainable**.
* They allow **dependency injection (DI)** for better **testability**.
* They help in **encapsulation** of data.

**3. POJO in Different Layers of a Spring Application**

Spring uses POJOs in multiple layers of an application:

**a. Model Layer (Entity Class)**

POJOs are often used as entity classes in a Spring application, representing database tables.

public class Employee {

private int id;

private String name;

private String department;

public Employee() {}

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

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

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

}

**b. Service Layer (Business Logic)**

POJOs can be used in the service layer to encapsulate business logic.

Example:

@Service

public class EmployeeService {

public String getEmployeeDetails(Employee emp) {

return "Employee: " + emp.getName() + ", Department: " + emp.getDepartment();

}

}

**c. Controller Layer (Handling HTTP Requests)**

Spring controllers use POJOs to map JSON data from client requests.

Example:

@RestController

@RequestMapping("/employee")

public class EmployeeController {

@PostMapping("/create")

public String createEmployee(@RequestBody Employee emp) {

return "Employee " + emp.getName() + " created successfully!";

}

}

### 4. POJO vs JavaBean vs DTO

| **Feature** | **POJO (Plain Old Java Object)** | **JavaBean** | **DTO (Data Transfer Object)** |
| --- | --- | --- | --- |
| **Definition** | A simple Java object without specific restrictions. | A POJO with additional rules (serialization, no-arg constructor, getters/setters). | A specialized POJO used for transferring data between layers. |
| **Encapsulation** | Optional | Required | Required |
| **Fields** | Can be public or private | Must be private | Must be private |
| **Getters/Setters** | Optional | Mandatory | Mandatory |
| **Constructor** | Not required | Must have a no-arg constructor | Not required but recommended |
| **Implements Serializable** | Optional | Recommended (for JavaBeans standard) | Often implemented for data transfer |
| **Annotations** | Not required | Not required but used in frameworks | Often annotated (@Data, @Getter, etc.) |
| **Usage** | Any simple object in Java | Used in frameworks like JSP, JSF | Transfers data between layers (e.g., API, DB) |
| **Example Use Case** | General-purpose Java objects | Java EE, Spring Beans | REST APIs, microservices, database communication |

**5. POJO with Spring Boot**

Spring Boot automatically maps JSON requests to POJOs using Jackson.

Example of POJO with Lombok (to avoid boilerplate code):

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Product {

private int id;

private String name;

private double price;

}

* @Data: Generates getters, setters, toString(), and equals() methods.
* @NoArgsConstructor: Generates a no-argument constructor.
* @AllArgsConstructor: Generates a constructor with all fields.

**6. Conclusion**

* POJOs are simple Java objects without complex dependencies.
* Spring uses POJOs in model, service, and controller layers.
* POJOs help in making applications **modular, testable, and maintainable**.
* Spring Boot and Lombok simplify working with POJOs.

**Spring Boot 3 and Spring 6 Upgrade Notes**

### ****Upgrade Requirements:****

1. **Upgrade to Java 17 or Higher**
   * Spring Boot 3 and Spring 6 require a minimum of **Java 17**.
2. **Update Package Imports to Jakarta EE Namespace**
   * Spring Boot 3 and Spring 6 use Jakarta EE instead of Java EE.
   * **Hibernate/JPA Code:**
     + Replace: import javax.persistence.\*;
     + With: import jakarta.persistence.\*;
   * **@PostConstruct and @PreDestroy:**
     + Replace:  
       import javax.annotation.PostConstruct;  
       import javax.annotation.PreDestroy;
     + With:  
       import jakarta.annotation.PostConstruct;  
       import jakarta.annotation.PreDestroy;
   * **Hibernate Validator:**
     + Replace: import javax.validation.constraints.\*;
     + With: import jakarta.validation.constraints.\*;
     + Use **Hibernate Validator version 8.x**.
3. **Upgrade to Tomcat 10 (For Spring MVC Apps - Non Boot)**
   * Spring MVC applications require **Tomcat 10** due to Jakarta EE support.
   * Download Tomcat 10 from: [Apache Tomcat 10](https://tomcat.apache.org/download-10.cgi)

By following these updates, your code will be compatible with **Spring Boot 3** and **Spring 6**.

**How IoC (Inversion of Control) Works in the Background in Spring**

In Spring, **IoC (Inversion of Control)** is a design principle where the **control of object creation and dependency management is transferred** from the developer to the **Spring Container**. This mechanism allows for better **loose coupling** and **dependency injection**.

**1. IoC in Action - Behind the Scenes**

When you create a Spring application, the following steps happen in the background:

**Step 1: Spring Container Initialization**

* When the application starts, Spring **creates an IoC container** (like ApplicationContext).
* This container is responsible for managing **beans** (Java objects).
* Spring **scans and registers beans** defined using **XML configuration, Java-based configuration, or annotations**.

Example:

java

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Here, context is the IoC container managing beans.

**Step 2: Bean Creation & Management**

* The IoC container **automatically detects beans** (e.g., classes with @Component, @Service, etc.).
* Spring **instantiates the beans** and manages their lifecycle.
* It stores the beans in the **Bean Factory (Singleton or Prototype scope).**

Example:

java

@Component

public class EmployeeService {

public void serve() {

System.out.println("Serving Employee...");

}

}

Spring detects this class and **creates an instance automatically**.

**Step 3: Dependency Injection (DI)**

* When a class requires another bean, Spring **injects the dependency automatically**.
* The container **looks up dependencies** and **resolves them**.

Example:

java

@Service

public class CompanyService {

private final EmployeeService employeeService;

@Autowired

public CompanyService(EmployeeService employeeService) {

this.employeeService = employeeService;

}

public void process() {

employeeService.serve();

}

}

* @Autowired tells Spring to **inject the EmployeeService bean automatically**.

**Step 4: Bean Lifecycle Management**

* Spring **manages the lifecycle** of beans, including:
  + **Initialization (@PostConstruct)**
  + **Usage**
  + **Destruction (@PreDestroy)**

Example:

@Component

public class DatabaseConnection {

@PostConstruct

public void init() {

System.out.println("Initializing DB connection...");

}

@PreDestroy

public void destroy() {

System.out.println("Closing DB connection...");

}

}

Spring automatically calls init() after bean creation and destroy() before shutting down.

**2. Types of IoC Containers in Spring**

Spring provides two types of IoC containers:

1. **BeanFactory** (Lightweight, used for simple applications) BeanFactory factory = new ClassPathXmlApplicationContext("beans.xml");
2. **ApplicationContext** (More advanced, supports AOP, event propagation) ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

**3. IoC vs Traditional Object Creation**

| **Feature** | **Traditional Java** | **Spring IoC** |
| --- | --- | --- |
| Object Creation | Manually using new keyword | Automatically managed by Spring |
| Dependency Management | Developer handles dependencies | Spring injects dependencies |
| Lifecycle Control | Developer must manage manually | Spring manages lifecycle |
| Flexibility | Hardcoded dependencies | Loose coupling with DI |

Example (Without IoC):

public class Main {

public static void main(String[] args) {

EmployeeService employeeService = new EmployeeService(); // Manual object creation

CompanyService companyService = new CompanyService(employeeService);

}

}

With IoC:

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

CompanyService companyService = context.getBean(CompanyService.class);

companyService.process(); // IoC manages dependencies

**Conclusion**

* **IoC allows Spring to control object creation & dependency injection**.
* **Spring Container manages bean lifecycle & dependencies automatically**.
* **It leads to more modular, testable, and maintainable applications**

**Code to Interface in Spring - Detailed Explanation**

**1. What is "Code to Interface" in Spring?**

**Code to Interface** is a design principle where we write code using **interfaces** instead of directly referring to concrete classes. This improves **loose coupling**, making the code more flexible, maintainable, and testable.

**Spring Framework promotes this by using Dependency Injection (DI) to inject dependencies via interfaces rather than concrete implementations.**

**2. Why Use Code to Interface?**

✅ **Loose Coupling** → The implementation can be changed easily.  
✅ **Easier Unit Testing** → You can mock dependencies using interfaces.  
✅ **Better Maintainability** → Changing an implementation does not affect dependent classes.  
✅ **Follows SOLID Principles** → Especially the **Dependency Inversion Principle (DIP)**.

**3. Example of Code to Interface in Spring**

Let's implement **Code to Interface** using Spring IoC and Dependency Injection.

**Step 1: Create an Interface**

We define a PaymentService interface that provides a method for processing payments.

java

public interface PaymentService {

void processPayment(double amount);

}

**Step 2: Implement the Interface (Concrete Classes)**

We create two implementations: **CreditCardPaymentService** and **PayPalPaymentService**.

java

import org.springframework.stereotype.Service;

@Service

public class CreditCardPaymentService implements PaymentService {

@Override

public void processPayment(double amount) {

System.out.println("Processing credit card payment of $" + amount);

}

}

java

import org.springframework.stereotype.Service;

@Service

public class PayPalPaymentService implements PaymentService {

@Override

public void processPayment(double amount) {

System.out.println("Processing PayPal payment of $" + amount);

}

}

**Step 3: Inject the Dependency in a Client Class**

Instead of directly creating objects, we inject the PaymentService **interface** using @Autowired.

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

import org.springframework.stereotype.Service;

@Service

public class ShoppingCartService {

private final PaymentService paymentService;

@Autowired // Spring will automatically inject an implementation of PaymentService

public ShoppingCartService(PaymentService paymentService) {

this.paymentService = paymentService;

}

public void checkout(double amount) {

System.out.println("Checking out with amount: $" + amount);

paymentService.processPayment(amount); // Calls the method dynamically

}

}

**Step 4: Configure and Run the Application**

We create a Spring Boot application to demonstrate how IoC injects dependencies.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

@SpringBootApplication

public class PaymentApplication {

public static void main(String[] args) {

ApplicationContext context = SpringApplication.run(PaymentApplication.class, args);

ShoppingCartService shoppingCartService = context.getBean(ShoppingCartService.class);

shoppingCartService.checkout(100.0); // Will process payment dynamically

}

}

**4. How Spring Handles Dependency Injection?**

When Spring starts:

1. It **scans** the classes annotated with @Service, @Component, etc.
2. It finds that CreditCardPaymentService and PayPalPaymentService implement PaymentService.
3. It **injects one implementation automatically** into ShoppingCartService (default is the first found).
4. The checkout() method calls processPayment() **without knowing the concrete class**.

**5. What if Multiple Implementations Exist?**

If we have multiple implementations (CreditCard & PayPal), Spring doesn't know which one to inject. To solve this:

**Option 1: Use @Qualifier**

Specify the exact implementation:

@Autowired

@Qualifier("creditCardPaymentService")

private PaymentService paymentService;

**Option 2: Use @Primary**

Mark one implementation as the default:

@Service

@Primary

public class CreditCardPaymentService implements PaymentService {

// Implementation...

}

**6. Key Takeaways**

* **Code to Interface promotes loose coupling** by depending on interfaces instead of concrete classes.
* **Spring IoC container handles dependency injection**, automatically injecting the required implementation.
* **Multiple implementations can be managed** using @Qualifier or @Primary.
* **It makes unit testing easier** since we can inject mock implementations.

**7. Example Unit Test**

Since we're coding to an interface, we can easily test ShoppingCartService using a mock implementation.

java

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class ShoppingCartServiceTest {

@Test

void testCheckout() {

PaymentService mockPaymentService = mock(PaymentService.class);

ShoppingCartService shoppingCart = new ShoppingCartService(mockPaymentService);

shoppingCart.checkout(200.0);

verify(mockPaymentService).processPayment(200.0); // Verify method was called

}

}

Here, we **mock** PaymentService instead of relying on a real implementation.

**8. Summary**

| **Feature** | **Without Interface** | **With Interface (Best Practice)** |
| --- | --- | --- |
| Coupling | Tightly coupled | Loosely coupled |
| Testability | Hard to test | Easy to mock & test |
| Flexibility | Hard to switch implementations | Easy to swap implementations |
| Maintainability | High dependency | Easier to maintain |

**Coding to an interface + Spring IoC = Highly scalable & maintainable applications!** 🚀

**How the Code Works in the Background (Spring IoC & Dependency Injection)**

Your Spring application follows the **Inversion of Control (IoC) and Dependency Injection (DI)** principles. Let's go step by step to understand **how it works in the background.** 🚀

**1. Application Starts: Spring Context is Created**

**Code Execution:**

ClassPathXmlApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

**Behind the Scenes:**

1. Spring **reads the applicationContext.xml file**.
2. It **creates and manages all beans** (objects) inside the **IoC container**.
3. It **initializes** the beans based on the configuration.

**2. Bean Creation & Dependency Injection**

**Beans Defined in applicationContext.xml:**

<bean id="myFortuneService"

class="spring\_01\_demo.springdemo.HappyFortuneService">

</bean>

<bean id="myCoach" class="spring\_01\_demo.springdemo.TrackCoach">

<constructor-arg ref="myFortuneService" />

</bean>

**What Happens Internally?**

1. Spring **creates an instance** of HappyFortuneService:

HappyFortuneService myFortuneService = new HappyFortuneService();

1. Then, Spring **creates an instance** of TrackCoach, passing the myFortuneService object via constructor injection:

TrackCoach myCoach = new TrackCoach(myFortuneService);

**3. Bean Retrieval from Spring Container**

**Code Execution:**

Coach theCoach = context.getBean("myCoach", Coach.class);

**Behind the Scenes:**

* Spring **retrieves the bean** from the IoC container.
* Since TrackCoach implements Coach, Spring returns the object as type Coach.

Internally, it's equivalent to:

Coach theCoach = (Coach) springContainer.getBean("myCoach");

**4. Method Calls & Dependency Usage**

**Code Execution:**

System.out.println(theCoach.getDailyWorkout());

System.out.println(theCoach.getDailyFortune());

**Behind the Scenes:**

1. When calling theCoach.getDailyWorkout(), the method from TrackCoach is executed:

java

public String getDailyWorkout() {

return "Run a hard 5k";

}

Output:Run a hard 5k

1. When calling theCoach.getDailyFortune(), it calls getFortune() from the injected HappyFortuneService:

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

Since HappyFortuneService.getFortune() returns "Today is your lucky day!", the final output is:

vbnet

Just Do It: Today is your lucky day!

**5. Context Shutdown**

**Code Execution:**

context.close();

**Behind the Scenes:**

1. Spring **calls the destroy lifecycle methods** (if any).
2. The **Spring container is shut down**, releasing resources.

**Summary of What Happens in the Background**

| **Step** | **What Happens in Background** |
| --- | --- |
| **1. Load Context** | Spring reads applicationContext.xml, sets up the IoC container. |
| **2. Create Beans** | Spring creates HappyFortuneService and TrackCoach. |
| **3. Inject Dependencies** | TrackCoach gets HappyFortuneService injected via constructor. |
| **4. Retrieve Bean** | Spring returns TrackCoach when calling context.getBean("myCoach"). |
| **5. Execute Methods** | Calls getDailyWorkout() & getDailyFortune(), using the injected dependency. |
| **6. Close Context** | Spring releases resources and shuts down. |

**Why is This Important?**

✅ **Loose Coupling** → You can change HappyFortuneService without modifying TrackCoach.  
✅ **Easier Maintenance** → Configuration is externalized in XML (or Java-based configuration).  
✅ **Scalability** → You can replace TrackCoach with another implementation (e.g., BaseballCoach) without changing the main code.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

<bean id="myFortuneService"

class="com.luv2code.springdemo.HappyFortuneService">

</bean>

<bean id="myCoach"

class="com.luv2code.springdemo.BaseballCoach">

<constructor-arg ref="myFortuneService" />

</bean>

HappyFortuneService myFortuneService = new HappyFortuneService();

BaseballCoach myCoach = new BaseballCoach(myFortuneService);

<bean id="myFortuneService"

class="com.luv2code.springdemo.HappyFortuneService">

</bean>

<bean id="myCricketCoach"

class="com.luv2code.springdemo.CricketCoach">

<property name="fortuneService" ref="myFortuneService" />

</bean>

HappyFortuneService myFortuneService = new HappyFortuneService();

CricketCoach myCricketCoach = new CricketCoach();

myCricketCoach.setFortuneService(myFortuneService);

The **no-argument constructor** in Spring-managed beans serves several purposes:

**1. Required for JavaBean Specification**

* Spring often follows the **JavaBean convention**, which recommends a **no-arg constructor** so that objects can be instantiated easily and populated via setters or dependency injection.

**2. Allows Spring to Create Bean Instances**

* When using **Spring IoC (Inversion of Control)**, Spring **first creates an object using the no-arg constructor**, then injects dependencies (if needed).
* If a **constructor with arguments is present**, but there is **no no-arg constructor**, you **must define explicit constructor-based injection** in Spring.

**3. Default Constructor is Needed for Some Proxying Mechanisms**

* Spring uses **CGLIB proxying** for beans that are **not interfaces** (e.g., when using @Transactional).
* CGLIB requires a **no-arg constructor** to generate a subclass proxy.

**4. Helpful for Serialization & Reflection**

* Some frameworks (e.g., Hibernate, Jackson, JPA) require a no-arg constructor for creating instances via reflection or deserialization.

**Example Scenario**

**Class with No-Arg Constructor (Works Fine with Spring)**

public class TrackCoach implements Coach {

private FortuneService fortuneService;

public TrackCoach() {

// No-arg constructor (needed for Spring)

}

public TrackCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

@Override

public String getDailyWorkout() {

return "Run a hard 5k";

}

@Override

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

}

🔹 **Spring can create an instance using the no-arg constructor before injecting dependencies.**

**Class with Only Parameterized Constructor (Needs Explicit Configuration)**

java

CopyEdit

public class TrackCoach implements Coach {

private FortuneService fortuneService;

// No no-arg constructor provided

public TrackCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

@Override

public String getDailyWorkout() {

return "Run a hard 5k";

}

@Override

public String getDailyFortune() {

return "Just Do It: " + fortuneService.getFortune();

}

}

🔹 **This class has no default constructor, so Spring requires explicit constructor injection configuration in XML or Java.**

**Solution: Explicit Constructor Injection in XML**

xml

CopyEdit

<bean id="myCoach" class="com.example.TrackCoach">

<constructor-arg ref="myFortuneService" />

</bean>

**Summary**

* **Spring prefers a no-arg constructor** to create beans easily.
* If only a **parameterized constructor exists**, **Spring needs explicit configuration**.
* No-arg constructors are required for **proxying, serialization, and frameworks like Hibernate**.

### ****FAQ: Why Use CricketCoach Instead of Coach Interface?****

**Q:** Why do we use the CricketCoach class instead of the Coach interface in setter injection?

**A:** The CricketCoach class has additional methods (getTeam() and setTeam()) that are not part of the Coach interface.

* If you retrieve the bean as Coach:

Coach theCricketCoach = context.getBean("myCricketCoach", Coach.class);

You can only access getDailyWorkout() and getDailyFortune().

* If you retrieve the bean as CricketCoach:

CricketCoach theCricketCoach = context.getBean("myCricketCoach", CricketCoach.class);

You can access **all methods**, including getTeam() and setTeam().

**Conclusion:** Your choice determines method visibility—use CricketCoach for full access or Coach for interface-level access.

### ****Important Notes on Init and Destroy Methods in XML Configuration****

When defining init-method and destroy-method in Spring XML, keep these points in mind:

1. **Access Modifier:** The method can be public, protected, or private.
2. **Return Type:** Any return type is allowed, but void is recommended since the return value is not used.
3. **Method Name:** You can choose any name for the method.
4. **Arguments:** The method **must not** accept any parameters (it should be a **no-arg** method).

### ****Special Note: Destroy Lifecycle and Prototype Scope****

In Spring, the **destroy-method** does **not** work for beans with **prototype scope**.

#### ****Why?****

* For **singleton beans**, Spring **manages the lifecycle**, including calling the destroy-method when the application shuts down.
* For **prototype beans**, Spring **only manages the creation** and does **not** track the bean after it is handed over.
* Since Spring does **not** manage the full lifecycle of prototype beans, it **does not call** the destroy-method automatically.

#### ****How to Handle Cleanup for Prototype Beans?****

* Manually call the cleanup method when you are done using the prototype bean.
* Use a **custom bean processor** or manage destruction using @PreDestroy and manually invoke it in your application.

### ****Difference Between Constructor and Setter**** @Autowired ****in Spring****

Spring provides **dependency injection (DI)** using @Autowired annotation in two ways:

1. **Constructor Injection**
2. **Setter Injection**

### ****1. Constructor Injection (****@Autowired ****on Constructor)****

@Component

public class CricketCoach implements Coach {

private FortuneService fortuneService;

@Autowired

public CricketCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

}

#### ****✔ Advantages:****

* Ensures the dependency is set at the time of object creation (required dependency).
* Encourages **immutable dependencies** (good for thread safety).
* Works well for mandatory dependencies.

#### ****❌ Disadvantages:****

* If there are **too many dependencies**, it can make the constructor large and harder to manage.

### ****2. Setter Injection (****@Autowired ****on Setter Method)****

@Component

public class CricketCoach implements Coach {

private FortuneService fortuneService;

public CricketCoach() {}

@Autowired

public void setFortuneService(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

}

#### ****✔ Advantages:****

* Allows setting dependencies **after object creation** (flexible and optional dependencies).
* Useful when the dependency is **not always required**.
* Good for **dynamic injection** when dependencies may change at runtime.

#### ****❌ Disadvantages:****

* The object can be created **without dependencies**, leading to potential NullPointerException if not set later.
* Requires extra method calls, making the code slightly less efficient than constructor injection.

### ****Which One to Use?****

| **Criteria** | **Constructor @Autowired** | **Setter @Autowired** |
| --- | --- | --- |
| **Dependency Requirement** | Mandatory | Optional |
| **Immutability** | Promotes immutable dependencies | Allows changes after object creation |
| **Flexibility** | Less flexible (dependency must be provided) | More flexible (dependency can be set later) |
| **Code Complexity** | Constructor may become long | More readable for many dependencies |
| **Best Practice** | Recommended for required dependencies | Use for optional or frequently changing dependencies |

### ****Final Recommendation****

* Use **constructor injection** when the dependency is **required** and should not change.
* Use **setter injection** when the dependency is **optional** or needs to be changed dynamically.

### ****How Field Injection Works****

* Spring automatically injects the required bean into the field **without requiring explicit constructor or setter methods**.
* The field must be **declared as private**, and Spring will handle the dependency injection **via reflection**.

### ****Advantages of Field Injection****

✔ **Less Boilerplate Code** – No need for constructors or setter methods.  
✔ **Easy to Use** – Just annotate the field with @Autowired.  
✔ **Good for Simple Classes** – If dependencies are stable and do not change often, this is a quick way to inject them.

### ****Disadvantages of Field Injection****

❌ **Difficult to Unit Test** – Since the field is private, you need **reflection or Spring Context** to inject mocks during testing.  
❌ **Less Flexibility** – You cannot change the dependency dynamically as in setter injection.  
❌ **Tightly Coupled** – The class depends directly on Spring for dependency injection, reducing portability.

### ****Which One to Use?****

| **Type of Injection** | **Pros** | **Cons** |
| --- | --- | --- |
| **Constructor Injection** | Best for mandatory dependencies, promotes immutability | Requires longer constructor for multiple dependencies |
| **Setter Injection** | Good for optional dependencies, allows dynamic changes | More method calls, object can be created without dependencies |
| **Field Injection** | Simple and less code | Harder to unit test, tightly coupled with Spring |

### ****Final Recommendation****

* **Use Constructor Injection** for required dependencies.
* **Use Setter Injection** for optional dependencies.
* **Avoid Field Injection** unless in very simple cases, as it makes testing harder and increases coupling.

**@Qualifier in Spring Framework**

When multiple beans of the same type exist, Spring doesn’t know which one to inject. The @Qualifier annotation helps resolve this ambiguity by **specifying the exact bean to use**.

**1️⃣ Why Use @Qualifier?**

By default, Spring uses **type-based** dependency injection. If multiple beans of the same type exist, Spring doesn’t know which one to inject.

✅ **@Qualifier helps to select the correct bean** when there are multiple candidates.

**2️⃣ Example Without @Qualifier (Causes Error)**

@Component

public class HappyFortuneService implements FortuneService {

@Override

public String getFortune() {

return "Today is your lucky day!";

}

}

@Component

public class SadFortuneService implements FortuneService {

@Override

public String getFortune() {

return "Today is a tough day!";

}

}

@Component

public class CricketCoach implements Coach {

private FortuneService fortuneService;

@Autowired

public CricketCoach(FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

@Override

public String getDailyWorkout() {

return "Practice fast bowling for 15 minutes.";

}

@Override

public String getDailyFortune() {

return fortuneService.getFortune();

}

}

🔴 **Error: NoUniqueBeanDefinitionException**  
Spring finds two beans (HappyFortuneService and SadFortuneService) and doesn’t know which one to use.

**3️⃣ Solution Using @Qualifier**

We explicitly specify the bean name to inject.

@Component

public class CricketCoach implements Coach {

private FortuneService fortuneService;

@Autowired

public CricketCoach(@Qualifier("happyFortuneService") FortuneService fortuneService) {

this.fortuneService = fortuneService; }

@Override

public String getDailyWorkout() {

return "Practice fast bowling for 15 minutes."; }

@Override

public String getDailyFortune() {

return fortuneService.getFortune();

}}

🟢 **Now Spring injects HappyFortuneService and works fine!**

**4️⃣ @Qualifier with Field Injection**

Instead of constructor injection, you can also use it directly on a field:

@Autowired

@Qualifier("sadFortuneService")

private FortuneService fortuneService;

✅ **This injects SadFortuneService without needing a constructor or setter.**

**5️⃣ @Qualifier with Setter Injection**

@Autowired

public void setFortuneService(@Qualifier("happyFortuneService") FortuneService fortuneService) {

this.fortuneService = fortuneService;

}

✅ **This allows Spring to inject the specified dependency via a setter method.**

**6️⃣ @Qualifier with @Primary**

If you don’t want to use @Qualifier everywhere, you can use @Primary to set a **default bean**.

@Component

@Primary

public class HappyFortuneService implements FortuneService {

@Override

public String getFortune() {

return "Today is your lucky day!";

}

}

Now, Spring will **automatically** inject HappyFortuneService unless another bean is explicitly specified using @Qualifier.

**7️⃣ @Qualifier in XML Configuration**

If you are using XML-based configuration, you can specify the bean in the XML file:

<bean id="happyFortuneService" class="com.luv2code.springdemo.HappyFortuneService"/>

<bean id="myCoach" class="com.luv2code.springdemo.CricketCoach">

<constructor-arg ref="happyFortuneService"/>

</bean>

This is equivalent to using @Qualifier in annotation-based configuration.

**✨ Summary**

| **Injection Type** | **Works With @Qualifier?** | **Example** |
| --- | --- | --- |
| Constructor Injection | ✅ Yes | @Autowired @Qualifier("beanName") in constructor |
| Field Injection | ✅ Yes | @Autowired @Qualifier("beanName") on field |
| Setter Injection | ✅ Yes | @Autowired @Qualifier("beanName") in setter |
| @Primary Alternative | ✅ Yes | @Primary on a default bean |

🔹 **Use @Qualifier when you have multiple beans of the same type.**  
🔹 **Use @Primary if you want a default bean without specifying @Qualifier.**

**Injecting Properties File in Spring using Java Annotations**

In Spring, we can use @PropertySource and @Value annotations to inject properties from an external .properties file into our Spring beans.

**Step 1: Create a Properties File**

Create a file named **app.properties** inside the src/main/resources directory.

**app.properties**

properties

CopyEdit

team.name=Sunrisers Hyderabad

coach.email=thebestcoach@luv2code.com

**Step 2: Load the Properties File in Spring Configuration**

Use @PropertySource in a configuration class to load the properties file.

**Configuration Class**

java

CopyEdit

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

@Configuration

@ComponentScan("com.example") // Adjust package name as needed

@PropertySource("classpath:app.properties") // Load the properties file

public class AppConfig {

}

**Step 3: Inject Property Values into a Spring Bean**

Use @Value to inject property values into a class.

**Coach Class with Property Injection**

java

CopyEdit

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

@Component

public class CricketCoach {

@Value("${team.name}") // Inject value from properties file

private String teamName;

@Value("${coach.email}") // Inject value from properties file

private String email;

public String getTeamName() {

return teamName;

}

public String getEmail() {

return email;

}

}

**Step 4: Retrieve Bean and Print Values**

In your main application, retrieve the bean and print the injected values.

**Main Application**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class SpringApp {

public static void main(String[] args) {

// Load Spring Context

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Retrieve the bean

CricketCoach coach = context.getBean(CricketCoach.class);

// Print injected values

System.out.println("Team: " + coach.getTeamName());

System.out.println("Email: " + coach.getEmail());

// Close context

context.close();

}

}

**Expected Output**

Team: Sunrisers Hyderabad

Email: thebestcoach@luv2code.com

**Summary**

| **Annotation** | **Purpose** |
| --- | --- |
| @PropertySource("classpath:app.properties") | Loads the properties file into Spring context |
| @Value("${property.key}") | Injects the property value into a field |

**Alternative: Injecting Properties Using Environment**

Instead of @Value, you can also use Environment to access properties dynamically.

**Example Using Environment**

import org.springframework.core.env.Environment;

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

import org.springframework.stereotype.Component;

@Component

public class CricketCoach {

@Autowired

private Environment env;

public String getTeamName() {

return env.getProperty("team.name");

}

public String getEmail() {

return env.getProperty("coach.email");

}

}

**Final Recommendation**

* Use @Value for **simple** property injections.
* Use Environment for **dynamic** property access.

### ****Notes on @PostConstruct and @PreDestroy Method Signatures****

1. **Access Modifier**
   * The method can have **any access modifier** (public, protected, private).
2. **Return Type**
   * Any return type is allowed, but **void is recommended** since the return value cannot be captured.
3. **Method Name**
   * The method **can have any name**; there are no restrictions on naming.
4. **Arguments**
   * **The method must not have any parameters** (it should be a **no-arg method**).

These annotations are used for **bean lifecycle management** in Spring to **initialize and clean up resources**.

### ****Special Note about Destroy Lifecycle and Prototype Scope in Spring****

1. **Destroy Method in Spring**
   * For **singleton-scoped beans**, Spring manages the bean lifecycle, including calling the **destroy method** when the application shuts down.
   * The destroy method can be defined using:
     + @PreDestroy annotation
     + destroyMethod attribute in @Bean
     + DisposableBean interface
2. **Prototype Scope Exception**
   * **Spring does NOT call the destroy method for prototype-scoped beans.**
   * Since prototype beans are created **on demand** and **not managed after injection**, Spring does **not track their lifecycle** beyond initialization.
3. **Handling Cleanup in Prototype Beans**
   * If you need to perform cleanup for prototype beans, you must manually call the destroy method in your code, for example:

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

MyPrototypeBean bean = context.getBean(MyPrototypeBean.class);

// Manually call destroy method

bean.cleanup();

### ****Key Takeaway****

* **Singleton beans** → Spring **manages** destroy lifecycle.
* **Prototype beans** → Spring **does not** handle destruction; **manual cleanup** is required.

### ****Ways to Pass Arguments to**** getBean() ****in Spring****

Spring’s ApplicationContext.getBean() method provides multiple ways to retrieve and instantiate beans. Let’s explore each in more detail.

## ****1. Retrieve Bean by ID (String)****

This method fetches a bean from the Spring container using its **ID** (as defined in the XML configuration or annotated with @Component and @Bean). However, it returns an Object, so **explicit casting is required**.

### ****Example (XML Configuration)****

xml

<bean id="myCoach" class="com.example.TrackCoach"/>

java

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

Coach myCoach = (Coach) context.getBean("myCoach"); // Explicit cast needed

System.out.println(myCoach.getDailyWorkout());

⚠ **Downside**: Requires explicit type casting, which can lead to ClassCastException if the type does not match.

## ****2. Retrieve Bean by ID and Type****

This method eliminates the need for casting by directly specifying the type.

### ****Example****

java

Coach myCoach = context.getBean("myCoach", Coach.class);

System.out.println(myCoach.getDailyWorkout());

💡 **Best Practice**: This is the preferred method when you know both the bean ID and type.

## ****3. Retrieve Bean by Type Only****

If there is **only one** bean of a particular type, you can fetch it using just the class type.

### ****Example (Annotation-Based Config)****

java

@Configuration

@ComponentScan("com.example")

public class AppConfig {}

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Coach myCoach = context.getBean(Coach.class);

⚠ **Limitation**: If multiple beans of the same type exist, Spring will throw NoUniqueBeanDefinitionException.

✅ **Solution**: Use @Primary or @Qualifier annotations to specify which bean should be returned.

## ****4. Passing Constructor Arguments to**** getBean()

This is useful for **prototype-scoped beans**, allowing you to provide constructor arguments at runtime.

### ****Example (Prototype Bean with Constructor Arguments)****

java

@Component

@Scope("prototype")

public class CustomBean {

private String name;

private int age;

public CustomBean(String name, int age) {

this.name = name;

this.age = age;

}

public void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

CustomBean bean1 = context.getBean(CustomBean.class, "John", 25);

bean1.display(); // Output: Name: John, Age: 25

⚠ **Note**:

* This works **only for prototype-scoped beans**.
* It does not work with singleton beans since they are created **at application startup**.

## ****5. Using**** GenericApplicationContext ****for Dynamic Bean Creation****

This approach is useful for registering and retrieving beans dynamically.

### ****Example****

GenericApplicationContext context = new GenericApplicationContext();

context.registerBean(CustomBean.class, () -> new CustomBean("Dynamic User", 30));

context.refresh();

CustomBean myBean = context.getBean(CustomBean.class);

myBean.display(); // Output: Name: Dynamic User, Age: 30

💡 **Best Use Case**: This is helpful for **programmatic bean registration** when you need to create beans dynamically at runtime.

## ****6. Using Factory Methods (****@Bean****)****

When defining beans using the @Bean annotation, arguments can be passed in a factory method.

### ****Example****

@Configuration

public class AppConfig {

@Bean

public Coach myCoach() {

return new TrackCoach(new HappyFortuneService());

}

}

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Coach myCoach = context.getBean("myCoach", Coach.class);

⚠ **Limitation**: This method does not allow passing different runtime arguments dynamically.

## ****Comparison Table****

| **Method** | **Usage** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| getBean("beanId") | Retrieve bean by ID | Simple | Requires explicit casting |
| getBean("beanId", Class.class) | Retrieve bean by ID and type | No casting required | Requires knowing both ID and type |
| getBean(Class.class) | Retrieve bean by type only | Clean code, no ID needed | Fails if multiple beans exist |
| getBean(Class, Object...) | Pass constructor args at runtime | Flexible for prototype beans | Works only with prototype scope |
| GenericApplicationContext.registerBean() | Programmatic bean creation | Dynamic bean registration | More complex |
| @Bean Factory Method | Define and retrieve beans | Full control over bean creation | No runtime argument passing |

## ****Conclusion****

* **For standard bean retrieval**, use getBean("beanId", Class.class).
* **For constructor arguments**, use getBean(Class, Object...), but remember it works only for **prototype** beans.
* **For dynamic registration**, use GenericApplicationContext.registerBean().
* **For factory-based bean definitions**, use @Bean in @Configuration.

### ****Explanation of**** web.xml ****(Deployment Descriptor for Spring MVC)****

This web.xml file is a **deployment descriptor** that configures the **Spring MVC DispatcherServlet** in a Java web application. It acts as the central configuration for the **Servlet-based architecture** of Spring MVC. Let's break it down:

### ****1. Root Element:**** <web-app>

This is the root element of the web.xml file. It defines the structure of a Java web application.

xml

CopyEdit

<web-app>

### ****2. Configuring the DispatcherServlet****

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

* **<servlet>**: Defines a new servlet.
* **<servlet-name>**: Assigns a name to the servlet (dispatcher).
* **<servlet-class>**: Specifies the class handling this servlet (DispatcherServlet).

📌 **DispatcherServlet is the front controller of Spring MVC**, responsible for handling all incoming requests and delegating them to appropriate controllers.

### ****3. Providing Configuration File (****spring-mvc-demo-servlet.xml****)****

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>

</init-param>

* **<init-param>**: Used to pass initialization parameters to the servlet.
* **contextConfigLocation**: Tells Spring where to find the MVC configuration file.
* **/WEB-INF/spring-mvc-demo-servlet.xml**: This XML file contains bean definitions and other configurations (like component scanning, view resolvers, etc.).

📌 **Why is this needed?** It ensures that Spring MVC knows where to look for additional configurations related to controllers, view resolvers, and other MVC components.

### ****4. Setting Load-on-Startup Priority****

<load-on-startup>1</load-on-startup>

* Tells the servlet container to **initialize the DispatcherServlet** when the application starts.
* A **value of 1** means it should be loaded **before other servlets** (higher priority).

📌 **Why is this important?** It ensures that the DispatcherServlet is available **as soon as the application starts**, making the web application responsive from the beginning.

### ****5. Closing the Root Tag****

</servlet>

</web-app>

This marks the end of the servlet configuration and the web application descriptor.

### ****Summary****

| **Element** | **Purpose** |
| --- | --- |
| <servlet> | Defines the **DispatcherServlet** for Spring MVC. |
| <servlet-name> | Names the servlet (dispatcher). |
| <servlet-class> | Specifies Spring's **DispatcherServlet** as the controller. |
| <init-param> | Passes the path of the Spring MVC configuration file. |
| <load-on-startup> | Ensures **early loading** of the servlet for faster request handling. |

### ****How it Works?****

1. **Client makes a request** (e.g., http://localhost:8080/app/home).
2. **DispatcherServlet intercepts it**, looks for the appropriate controller.
3. **Controller processes the request**, interacts with services/models.
4. **View Resolver sends back a response** (like a JSP page or JSON output).

### ****Explanation of**** web.xml ****(Spring MVC Configuration)****

This web.xml file is a **deployment descriptor** that configures Spring MVC's **DispatcherServlet** and maps it to handle all incoming requests.

### ****1. Root Element:**** <web-app>

This defines the structure of the Java web application.

<web-app>

### ****2. Configuring DispatcherServlet****

xml

CopyEdit

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

...

</servlet>

* **Defines the DispatcherServlet**, which acts as the **Front Controller** in Spring MVC.
* **<servlet-name>dispatcher</servlet-name>** assigns a name to the servlet.
* **<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>** specifies that this servlet will handle all incoming requests.
* **The DispatcherServlet delegates requests to controllers, models, and views.**

### ****3. Mapping DispatcherServlet to Handle Requests****

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

* **Links the servlet to a URL pattern.**
* **<url-pattern>/</url-pattern>** means **all requests** will be handled by the DispatcherServlet.

📌 **What does this mean?**

* Any request (e.g., http://localhost:8080/home, http://localhost:8080/login) will be routed through DispatcherServlet, which will decide how to process them.

### ****How This Works in Spring MVC?****

1. **Client makes a request** (http://localhost:8080/home).
2. **DispatcherServlet intercepts the request** (because of / mapping).
3. **It looks for a controller** (@Controller annotated class).
4. **Controller processes the request** and returns a **view name**.
5. **View Resolver maps it to a JSP/HTML page** and sends a response.

### ****Key Takeaways****

| **Element** | **Purpose** |
| --- | --- |
| <servlet> | Defines the Spring MVC **DispatcherServlet**. |
| <servlet-class> | Specifies DispatcherServlet as the front controller. |
| <servlet-mapping> | Maps requests to DispatcherServlet. |
| <url-pattern>/</url-pattern> | Ensures **all requests** are handled by Spring MVC. |

📌 **Next Step:** You should also configure dispatcher-servlet.xml (Spring MVC config file) to define controllers, view resolvers, and component scanning.

### ****Explanation of**** spring-mvc-demo-servlet.xml ****(Spring MVC Configuration)****

This XML file is a **Spring MVC configuration file** that is loaded by the DispatcherServlet. It defines **component scanning, annotation support, and the view resolver**.

### ****1. Root Element:**** <beans>

This defines a Spring configuration file.

<beans>

### ****2. Component Scanning****

<context:component-scan base-package="com.Luv2code.springdemo" />

* This **enables component scanning** in the specified package.
* It automatically detects and registers **Spring components** like @Controller, @Service, and @Repository.

✅ **Effect:** No need to manually define beans in XML for controllers and services.

### ****3. Enable Annotation Support****

<mvc:annotation-driven />

* Enables support for:
  + @RequestMapping, @GetMapping, @PostMapping
  + @Valid for validation
  + @ResponseBody for REST APIs
  + @InitBinder for custom data binding

✅ **Effect:** Allows using **annotations instead of XML** for Spring MVC features.

### ****4. View Resolver Configuration****

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/view/" />

<property name="suffix" value=".jsp" />

</bean>

* This **configures how views (JSP pages) are resolved** in Spring MVC.
* **prefix** → /WEB-INF/view/ (JSP files are stored inside this folder).
* **suffix** → .jsp (all view names will be appended with .jsp).

✅ **Effect:** If a controller returns "home", Spring resolves it as:

/WEB-INF/view/home.jsp

### ****Key Takeaways****

| **Element** | **Purpose** |
| --- | --- |
| <context:component-scan> | Automatically detects and registers Spring components. |
| <mvc:annotation-driven> | Enables annotations for request mapping, validation, and data binding. |
| <bean class="InternalResourceViewResolver"> | Resolves view names to JSP files in /WEB-INF/view/. |

### ****How It Works in Spring MVC?****

1. **User requests a URL** (e.g., http://localhost:8080/home).
2. **DispatcherServlet processes the request**.
3. **Spring scans for a matching controller (@Controller)**.
4. **Controller returns a view name ("home")**.
5. **View Resolver maps "home" → /WEB-INF/view/home.jsp**.
6. **JSP is displayed as the response**.

✅ **Next Step:** You need controllers (@Controller) to handle requests and return view names.

### ****How to Use CSS, JavaScript, and Images in a Spring MVC Web Application?****

In a Spring MVC web application, static resources like **CSS, JavaScript, and images** are typically placed in the webapp/resources/ folder. Spring MVC serves these files using the ResourceHandlerRegistry.

## ****🔹 Step 1: Place Static Resources in the**** resources ****Folder****

Inside the src/main/webapp/ directory, create a resources folder and organize your assets:

src/main/webapp/

│── WEB-INF/

│── resources/

│ ├── css/

│ │ ├── styles.css

│ ├── js/

│ │ ├── script.js

│ ├── images/

│ ├── logo.png

## ****🔹 Step 2: Configure Spring MVC to Serve Static Resources****

Add the following configuration to your Spring MVC file (spring-mvc-demo-servlet.xml or Java Config):

### ****🔸 XML Configuration (****spring-mvc-demo-servlet.xml****)****

<mvc:resources mapping="/resources/\*\*" location="/resources/" />

### ****🔸 Java Configuration (****WebMvcConfig.java****)****

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

@EnableWebMvc

public class WebMvcConfig implements WebMvcConfigurer {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/resources/");

}

}

## ****🔹 Step 3: Link CSS, JS, and Images in JSP****

Inside your JSP files (WEB-INF/view/home.jsp), reference the static files using the <c:url> tag:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!-- Link CSS -->

<link rel="stylesheet" type="text/css" href="<c:url value='/resources/css/styles.css'/>">

<!-- Link JavaScript -->

<script src="<c:url value='/resources/js/script.js'/>"></script>

<!-- Display Image -->

<img src="<c:url value='/resources/images/logo.png'/>" alt="Logo">

## ****🔹 Step 4: Verify and Run Your Application****

1. Start your **Spring MVC** application.
2. Open the browser and check if CSS, JS, and images load correctly.
3. You can directly test by accessing:
   * **CSS:** http://localhost:8080/yourapp/resources/css/styles.css
   * **JS:** http://localhost:8080/yourapp/resources/js/script.js
   * **Image:** http://localhost:8080/yourapp/resources/images/logo.png

### ****How to Use CSS, JavaScript, and Images in a Spring Boot Web Application?****

Spring Boot serves static resources like **CSS, JavaScript, and images** from the src/main/resources/static/ directory by default.

## ****🔹 Step 1: Place Static Resources in the**** static ****Folder****

Inside your Spring Boot project, create a static folder under src/main/resources/:

src/main/resources/

│── static/

│ ├── css/

│ │ ├── styles.css

│ ├── js/

│ │ ├── script.js

│ ├── images/

│ ├── logo.png

## ****🔹 Step 2: Reference Static Files in Thymeleaf or JSP****

### ****🔸 If Using Thymeleaf (****.html ****files in**** src/main/resources/templates/****):****

<!-- Link CSS -->

<link rel="stylesheet" type="text/css" th:href="@{/css/styles.css}">

<!-- Link JavaScript -->

<script th:src="@{/js/script.js}"></script>

<!-- Display Image -->

<img th:src="@{/images/logo.png}" alt="Logo">

### ****🔸 If Using JSP (****.jsp ****files in**** src/main/webapp/WEB-INF/views/****):****

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!-- Link CSS -->

<link rel="stylesheet" type="text/css" href="${pageContext.request.contextPath}/css/styles.css">

<!-- Link JavaScript -->

<script src="${pageContext.request.contextPath}/js/script.js"></script>

<!-- Display Image -->

<img src="${pageContext.request.contextPath}/images/logo.png" alt="Logo">

## ****🔹 Step 3: Run and Test Your Application****

1. **Start your Spring Boot application.**
2. **Verify by accessing URLs in the browser:**
   * **CSS:** http://localhost:8080/css/styles.css
   * **JS:** http://localhost:8080/js/script.js
   * **Image:** http://localhost:8080/images/logo.png

## ****🔹 Step 4: Customizing Resource Locations (Optional)****

By default, Spring Boot serves static resources from /static, /public, /resources, or /META-INF/resources.

If you want to change the default location, modify application.properties:

properties

CopyEdit

spring.web.resources.static-locations=classpath:/custom-folder/

Then place your static files inside:

src/main/resources/custom-folder/