Palash Bajpai

Spring

Interview Questions

Table of Contents

[Q1. What is difference between spring, spring boot and spring mvc? 2](#_Toc201662086)

[Q2. What are main features of Spring? 2](#_Toc201662087)

[Q3. Difference between **IoC (Inversion of Control)** and **Dependency Injection (DI)**? 3](#_Toc201662088)

[Q4. What are different types of IoC? 3](#_Toc201662089)

[Q5. How dependency Injection is implemented in Spring? 4](#_Toc201662090)

[Q6. What are roles of IoC container? 6](#_Toc201662091)

[Q7. Difference between BeanFactory and ApplicationContext 6](#_Toc201662092)

[✅ 10. What is the difference between @Autowired and @Qualifier? 8](#_Toc201662093)

## ­Q1. What is difference between spring, spring boot and spring mvc?

**Spring**

* Spring is a lightweight, modular Java framework that provides powerful features like **Dependency Injection** and **AOP** to build flexible, loosely coupled applications.
* It forms the core of the entire Spring ecosystem and supports various application types including web, batch, and enterprise apps.
* Requires **manual configuration** (XML or Java-based).

**Spring Boot**

* Spring Boot is a Spring-based framework that simplifies application development by providing **auto-configuration, starter dependencies, and embedded servers.**
* It enables rapid creation of production-ready Spring applications with minimal configuration and boilerplate code.
* Ideal for **rapid development** of micro services and web apps with minimal setup.

**Spring MVC**

* A **module of Spring** specifically for building **web applications** using the **Model-View-Controller** pattern.
* Provides **controllers, request mappings, view resolvers**, etc.
* Can be used with both traditional Spring and Spring Boot for web-layer development.

## Q2. What are main features of Spring?

* **Dependency Injection (DI):** Automatically injects object dependencies to promote loose coupling.
* **Aspect-Oriented Programming (AOP):** Separates cross-cutting concerns like logging, transactions, etc.
* **Spring MVC:** Provides a robust framework to build web applications using the MVC pattern.
* **Transaction Management:** Simplifies declarative transaction handling across different databases.
* **Integration Support:** Easily integrates with Hibernate, JPA, JMS, JDBC, etc.
* **Testability:** Supports unit testing by managing dependencies and mock creation.
* **Lightweight and Modular:** Allows you to use only the needed modules without loading the entire framework.
* **Inversion of Control (IoC):** The container manages object creation and lifecycle.
* **Event Handling:** Supports application-wide event publishing and handling.
* **Annotation Support:** Reduces boilerplate code using annotations like @Component, @Autowired, etc.

## Q3. Difference between **IoC (Inversion of Control)** and **Dependency Injection (DI)**?

Although closely related, IoC and Dependency Injection are not the same – **DI is type of IoC.**

**Inversion of Control (IoC): -** The Principle

IoC is a general **design principle** where the **control of object creation and flow** is transferred from the **application code to a container or framework**.

**Key idea:**  Instead of you calling the framework, The Framework calls your code.

**Dependency Injection (DI): -** A Technique to Achieve IoC

DI is a **specific implementation technique** of IoC where an object’s dependencies are **provided/injected by an external source**, typically the Spring container.

**Key idea:**  The container **injects dependencies** (via constructor, setter, or field), rather than the class creating them itself.

|  |  |
| --- | --- |
| **IoC** | Broad principle – "Don't call us, we'll call you" |
| **DI** | One way to implement IoC – inject what is needed |

## Q4. What are different types of IoC?

While **Dependency Injection (DI)** is the **most common type** of IoC, there are a few **other types of Inversion of Control (IoC)** patterns that also delegate control away from the application logic.

| **IoC Type** | **Description** | **Used In** |
| --- | --- | --- |
| **1. Dependency Injection (DI)** | Container injects required dependencies into a class. | ✅ Spring, Guice |
| **2. Service Locator Pattern** | Objects ask a centralized registry to give them the required service. | ❌ Legacy Spring, Java EE |
| **3. Event-based IoC** | Framework triggers handlers/listeners when specific events occur. | ✅ GUI apps, JavaScript, Spring Events |
| **4. Template Method Pattern** | A base class defines a skeleton algorithm and allows subclasses to override steps. | ✅ Spring’s JdbcTemplate, RestTemplate |
| **5. Strategy Pattern with IoC** | Algorithm behavior is injected or selected at runtime. | ✅ Used in rules engines, dynamic behavior injection |

## Q5. How dependency Injection is implemented in Spring?

How it works internally:

1. **You annotate your classes** with @Component, @Service, etc.
2. Spring scans your code (via @ComponentScan) and **creates objects** (called **beans**) for you.
3. When one bean needs another, Spring **injects the dependency** using:
   * **Constructor Injection**
   * **Setter Injection**
   * **Field Injection**
4. The Spring **IoC container manages the entire lifecycle** of these beans.

Behind the scene:

@ComponentScan

↓

ClassPath Scanner → Finds @Component classes

↓

Creates BeanDefinitions

↓

Resolves Dependencies (recursive)

↓

Instantiates Beans

↓

Injects Dependencies

↓

Post-processing (@PostConstruct)

↓

Beans stored in Singleton Container

**1. Spring Application starts**

When you run a Spring application (like new AnnotationConfigApplicationContext() or SpringApplication.run()), Spring:

* Loads the configuration (@Configuration, @ComponentScan, etc.)
* Initializes the **IoC container** (a.k.a. **ApplicationContext**)

**2. Classpath Scanning Begins**

Spring scans the packages specified in @ComponentScan (or default package) for **stereotype annotations**: @Component, @Service, @Repository, @Controller, and @Configuration.

Spring uses **reflection** and **bytecode scanning** (based on ClassPathScanningCandidateComponentProvider) to detect these.

**3. Spring Builds a Bean Definition Map**

Once it detects components, it:

* Creates a **BeanDefinition** object for each component.
* Stores them in an internal map: **Map<String, BeanDefinition> beanDefinitionMap**

Each BeanDefinition includes:

* Bean class name
* Scope (singleton/prototype)
* Bean dependencies
* Lifecycle callbacks (@PostConstruct, @PreDestroy)
* Whether it’s lazily loaded or not

This step is called Bean Definition Registration.

**4. Bean Creation(Instantiation)**

Now Spring **creates actual instances (beans)** of these classes using **reflection**, typically with:

**Object bean = clazz.getConstructor().newInstance();**

This is called **bean instantiation**.

**5. Dependency Resolution**

Before finalizing the bean, Spring checks if it has dependencies. It uses the **@Autowired**, @Inject, or constructor signatures to determine dependencies.

**6. Dependency Injection Happens**

Once all dependencies are resolved, Spring:

* Injects them into the bean
* Either by: **Calling the constructor**, **Using setters** , **Accessing private fields via reflection**

Internally, Spring uses AutowiredAnnotationBeanPostProcessor to process @Autowired.

**7. Post-processing and Initialization**

Before the bean is ready to use, Spring:

* Calls @PostConstruct methods
* Runs any BeanPostProcessor logic
* Calls afterPropertiesSet() if the bean implements InitializingBean

**8. Bean is Stored in Context**

Now the fully initialized bean is stored in a singleton map: singletonObjects.put(beanName, bean);

So future requests for the same bean don’t require re-creation.

## Q6. What are roles of IoC container?

The **IoC Container** is the **heart of the Spring Framework**. It is responsible for **managing the complete lifecycle of application objects** (called **beans**) and **wiring their dependencies**.

Roles of IoC container:

1. **Creates beans** based on configuration or annotations.
2. **Injects dependencies** automatically using constructor, setter, or field injection.
3. **Manages bean lifecycle**, including initialization and destruction.
4. **Reads configuration** from XML, annotations, or Java classes.
5. **Manages bean scopes**, like singleton, prototype, request, and session.
6. **Provides AOP support** by applying aspects like logging and transactions.
7. **Publishes and listens to events** using application event handling.
8. **Loads external resources** like properties, files, and environment variables.
9. **Applies BeanPostProcessors** for custom bean modification before/after init.
10. **Promotes loose coupling** by injecting abstractions (interfaces) instead of concrete classes.

## Q7. Difference between BeanFactory and ApplicationContext.

* **BeanFactory** is the basic IoC container in Spring that lazily loads beans and provides only core DI functionality.
* **ApplicationContext** is a **more advanced** container that **extends BeanFactory** and adds features like **AOP, internationalization, event propagation, and eager loading**.
* In practice, **ApplicationContext is preferred** in most real-world Spring applications.

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| **Type** | Basic IoC container | Advanced IoC container (extends BeanFactory) |
| **Bean Loading** | Lazy (on-demand) | Eager (at startup, by default) |
| **Annotation Support** | Limited | Full support for annotations (@Autowired) |
| **AOP Support** | Not available | Supported |
| **Event Handling** | Not supported | Supports event publication & listeners |
| **Internationalization (i18n)** | Not supported | Supported via MessageSource |
| **Environment Access** | Limited | Can access environment and profiles |
| **Use Case** | Lightweight or legacy apps | Modern, full-featured Spring apps |

## Q8. What is ComponentScan ?

**@ComponentScan** is an annotation that tells Spring **where to look for components (beans)** like @Component, @Service, @Repository, and @Controller.

It enables **automatic bean detection and registration** in the Spring container, avoiding manual bean definitions.

Spring doesn't scan packages automatically —  
You must explicitly tell it **which base packages to scan** for annotated components.

Eg:

package com.example;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan(basePackages = "com.example.service")

public class AppConfig {

}

// This tells Spring to scan com.example.service for beans

// like @Service, and register them in the container.

Eg2: Default Behaviour

package com.example;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan

public class AppConfig {

}

// If you don’t pass basePackages,

// Spring scans the package of the class where @ComponentScan is declared.

## Q8. What happens if multiple @ComponentScan annotations scan overlapping packages?

**Spring merges and avoids duplicate registration.** duplicate or overlapping packages won't cause errors.

## Q9. What is a bean?

A **bean** in Spring is just an **object** that is **managed by the Spring IoC container**.

When you annotate a class with @Component, @Service, @Repository, or define it in XML, Spring creates an instance of it and handles its **lifecycle, dependencies, and configuration**.

"**A bean is a normal Java object, but Spring takes care of creating, wiring, and managing it for you.**"

## Q10. What do you mean by Bean Lifecycle?

The **bean lifecycle** in Spring refers to the **series of steps** that a Spring-managed bean goes through — **from creation to destruction** — inside the **IoC container.**

**Bean lifecycle = From bean creation → setting values → ready to use → destroy** when the app stops.

Key Steps:

* **Bean Instantiation:** Creation of bean class instance.
* **Bean Post-processing:** Use of post-processors for customizing the beans.
* **Bean Initialization:** Use of *@PostConstruct*to set up the beans using methods.
* **Bean Usage:** Injection of beans for application-wide use
* **Bean Destruction:** Destroys the bean through methods annotated with *@PreDestroy*

## Q11. Explain bean scopes.

In Spring, **bean scope** defines **how many instances** of a bean Spring creates and **how long** those instances live within the container.

| **Scope** | **Description** |
| --- | --- |
| **singleton** (default) | One shared instance per Spring container. Created at startup. |
| **prototype** | New instance created **every time** the bean is requested. |
| **request** | One instance per **HTTP request** (Web apps only). |
| **session** | One instance per **HTTP session** (Web apps only). |
| **application** | One instance per **ServletContext** (shared across sessions). |
| **websocket** | One instance per **WebSocket session** (used in real-time apps). |

## Q12. Are beans thread safe?

**No**, Spring beans are not thread-safe by default**.**

* Default bean scope in Spring is Singleton, meaning one shared instance is used by all threads.
* If multiple threads access or modify shared fields in that bean without proper synchronization, it can lead to race conditions or inconsistent data.

Eg:

@Component

public class CounterBean {

    private int count = 0;

    public void increment() {

        count++; // ❌ Not thread-safe!

    }

    public int getCount() {

        return count;

    }

}

If two threads call increment() at the same time, the count can become inconsistent.

Ways to make beans Thread-safe

| **Strategy** | **Description** |
| --- | --- |
| Use **stateless beans** | Avoid shared mutable state in the bean |
| Use **synchronization** | Add synchronized blocks or methods |
| Use **prototype scope** | Give each request/thread a new bean |
| Use **concurrent-safe data structures** | e.g., AtomicInteger, ConcurrentHashMap |

## Q13. Can @Component, @Service, @Repository, and @Controller be used interchangeably?

Technically yes, While @Component, @Service, @Repository, and @Controller **can technically be used interchangeably** for component scanning (since they all mark a class as a Spring-managed bean), they are **not semantically or functionally identical**.

All of these get detected during **component scanning** and are registered as Spring beans.

| **Aspect** | **@Component** | **@Service** | **@Repository** | **@Controller** |
| --- | --- | --- | --- | --- |
| **Component scanning** | ✅ Yes | ✅ Yes | ✅ Yes | ✅ Yes |
| **Semantic meaning** | ❌ Generic | ✅ Service logic | ✅ DAO layer | ✅ Web layer |
| **Exception translation (Spring AOP)** | ❌ No | ❌ No | ✅ Yes (converts JPA/Hibernate exceptions) | ❌ No |
| **Web request handling** | ❌ No | ❌ No | ❌ No | ✅ Yes (via @RequestMapping, etc.) |
| **Best practice (readability)** | ❌ Confusing | ✅ Clear intent | ✅ Clear intent | ✅ Clear intent |

What can possibly go wrong:

1. **Using @Component instead of @Repository:**
   * ❌ Spring **won’t apply exception translation** (PersistenceExceptionTranslationPostProcessor won’t kick in).
   * You might see raw HibernateException instead of clean DataAccessException.
2. **Using @Component instead of @Controller:**
   * ❌ It won’t handle web requests — @RequestMapping or @GetMapping won’t work.
   * Spring doesn’t detect it as a controller in the DispatcherServlet.
3. **Using @Component instead of @Service:**
   * ✅ Technically works.
   * ❌ But semantically incorrect and confusing for others reading your code.
4. **Using @Service instead of @Controller:**
   * ❌ Web requests won’t be mapped or handled.
   * Will not participate in Spring MVC request flow.

## Q13. Can two beans have the same @Qualifier name?

**No,** spring cannot have two beans with same qualifier.

* @Qualifier("beanName") tells Spring **exactly which bean** to inject when multiple candidates of the same type exist.
* If **two or more beans have the same name**, Spring will throw a **BeanDefinitionOverrideException** at startup.

## Q13. Can you create Bean of Interface.

Technically No, Since **interfaces cannot be instantiated**, Spring can't create a bean **of an interface by itself** — you need to provide an **implementation** of that interface and then register that **implementation as a bean**.

Error will come at startup itself, only if that beam is really used somewhere

@Component  
public interface Car {  
 public void drive();  
}

If we don’t use it or do autowiring on this object, app will run easily. But if car object is used we get:

NoSuchBeanDefinitionException: No qualifying bean of type 'com.example.demo.components.Car' available: expected at least 1 bean which qualifies as autowire candidate.

**Note**: To properly see this exception, mark debug=true in application.properties

## Q13. Can we create bean for class with private constructor.

**Yes, Spring can still create a bean with a private constructor via reflection**, but **only under specific conditions**:

* The class is annotated with @Component (or similar stereotype like @Service, @Repository, @Controller)
* And Spring **uses reflection (via Constructor.newInstance() or similar low-level mechanism)** to instantiate the class
* It will **look for a no-arg constructor**, even if it's private, and **make it accessible**

This **works in Spring** because:

Spring uses ReflectionUtils to make the private constructor accessible (setAccessible(true)), and then creates the bean instance.

✅ So yes — **Spring Boot will successfully create this bean**.

## Q13. What happens to a @Scope("prototype") bean when it is autowired into another bean?

It is created **only once** — when the parent bean(singleton) is created — and the **same instance is reused** during the lifetime of the parent.

When you @Autowired a prototype-scoped bean into a singleton-scoped bean:

* Spring resolves and injects dependencies **once** at startup time.
* So even if the prototype is "supposed to be" created on every request, **this rule doesn’t apply when it's injected into a singleton** — because the singleton is created only once

## Q13. Is Singleton scope same as singleton design pattern?

**Not exactly —** they are similar in purpose but different in implementation and control.

* **Spring singleton = 1 instance per container**
* **Design pattern singleton = 1 instance per JVM (manually coded)**
* **Both limit object creation, but Spring’s version is managed and more flexible.**

| **Aspect** | **Spring singleton Scope** | **Singleton Design Pattern** |
| --- | --- | --- |
| **Who controls it?** | Spring IoC container | You (developer) write the code |
| **How many instances?** | One per Spring container | One per JVM or classloader |
| **Thread safety?** | Not guaranteed — you must manage it | Must handle thread safety manually |
| **Lazy vs Eager loading** | Eager by default (can be made lazy) | You decide when to initialize (lazy/eager) |

## Q14. What is autowiring and name the different modes of it?

**Autowiring** in Spring is the process of **automatically injecting dependencies** (i.e., required beans) into a class **without using explicit setter or constructor calls**.

Spring uses **annotations** to figure out which beans to inject into which fields, methods, or constructors.

Different modes of autowiring are:

| **Mode** | **Description** | **Usage** | **Code Example** |
| --- | --- | --- | --- |
| no *(default)* | No autowiring. Dependencies must be injected manually. | XML | **XML:**<bean id="car" class="com.example.Car" autowire="no" /> |
| byName | Matches bean **ID** with the property name and injects it. | XML | **XML:**<bean id="engine" class="com.example.Engine"/><bean id="car" class="com.example.Car" autowire="byName"/> |
| byType | Matches bean **type** with the property type and injects it. | XML | **XML:**<bean id="engine1" class="com.example.Engine"/><bean id="car" class="com.example.Car" autowire="byType"/> |
| constructor | Matches constructor parameters with bean types and injects them. | XML | **XML:**<bean id="engine" class="com.example.Engine"/><bean id="car" class="com.example.Car" autowire="constructor"/>Use Car(Engine) |
| @Autowired | Injects by **type** using annotations. | Annotation | @Autowired |
| @Qualifier | Used with @Autowired to resolve ambiguity when multiple beans of same type exist. | Annotation | @Autowired // byType  @Qualifier("engineV8") // narrow down by name  private Engine engine; |

## Q15. What are limitations of autowiring?

1. **Ambiguity with Multiple Beans of Same Type**

* If Spring finds **more than one matching bean**, it doesn’t know which one to inject.
* Leads to: NoUniqueBeanDefinitionException.
* Note: if bean of given type is not found, error we get is: NoSuchBeanDefinitionException. Also can occur if component scan don’t runs here.

@Autowired  
private Engine engine; // ❌ If two Engine beans exist, this will fail

**Fix**: Use @Qualifier or mark one bean as @Primary.

1. **Cannot autowire primitive Types or Strings easily**

Spring doesn’t know how to inject int, String, etc., unless you use @Value.

@Value("${app.version}")  
private String version;

1. **Field Injection Reduces Testability**

Using @Autowired on fields makes it harder to write unit tests with mock dependencies. Use constructor injection for improved testability.

1. **Harder to debug**

* When something fails, it’s not always clear which bean wasn’t injected or why.
* Errors like "Could not autowire. No beans of type found" can be cryptic.

## Q16. When to prefer each type of dependency Injection?

| **Injection Type** | **When to Prefer** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **Constructor Injection** | 🔹 Use when dependency is **mandatory and immutable**🔹 Best for **testability** (via constructor args) | ✅ Ensures all dependencies are provided✅ Encourages immutability | ❌ Verbose if many dependencies❌ Slightly more boilerplate |
| **Setter Injection** | 🔹 Use when dependency is **optional or replaceable later** | ✅ Flexible✅ Good for configuration or re-injection | ❌ Allows partially constructed objects |
| **Field Injection** | 🔹 Use for **quick prototyping** or small apps🔹 Not ideal for production | ✅ Clean and concise syntax | ❌ Harder to test/mock❌ Breaks encapsulation |
| **Method Injection** | 🔹 Use when the dependency is needed **only temporarily** or for a **specific logic block** | ✅ Fine-grained control✅ Good for one-off operations | ❌ Less common❌ Not very intuitive |

Q17. What is the difference between @Autowired and @Qualifier?

| **Annotation** | **Purpose** | **Works On** | **Key Point** |
| --- | --- | --- | --- |
| @Autowired | Automatically wires bean by type | Field, Constructor, Setter | Fails if multiple beans of same type exist |
| @Qualifier | Narrows down which bean to inject | Used with @Autowired | Resolves conflicts when multiple beans exist |

**@Autowired – *Injects by Type***

* Automatically injects a bean by **matching its type**.
* If there's **only one matching bean**, it works fine.
* If **multiple beans of the same type** exist, Spring throws NoUniqueBeanDefinitionException.

**@Qualifier – *Specifies Which Bean to Inject***

* Used **with @Autowired** to resolve ambiguity **when multiple beans of the same type** exist.
* Specifies the **exact bean name (ID)** to inject.

@Component  
public class Car {  
 @Autowired  
 @Qualifier("v8Engine")  
 private Engine engine; // Only injects the bean named 'v8Engine'  
}

Q18. How Java-based configurations are better than xml-configs?

| **Aspect** | **Java-based Configuration** | **XML-based Configuration** |
| --- | --- | --- |
| **Type-safety** | ✅ Compile-time type checking (errors caught early) | ❌ Runtime errors due to string-based bean IDs & classes |
| **IDE support** | ✅ Full auto-completion, refactoring, and navigation | ❌ Limited auto-completion, hard to trace in IDE |
| **Less verbose** | ✅ Cleaner, more readable code | ❌ Verbose and repetitive XML tags |
| **Reusable code** | ✅ Can use loops, conditions, logic inside config classes | ❌ XML is declarative — no real logic possible |
| **Easier testing** | ✅ Easily mock or inject dependencies for test configs | ❌ XML configs require separate parsing |
| **Annotations compatibility** | ✅ Seamlessly integrates with @Component, @Autowired, etc. | ❌ Can’t directly use annotations inside XML |
| **Flexible structure** | ✅ Supports profiles, conditional beans using @Profile, etc. | ❌ Achieving similar flexibility in XML is more complex |

## Q19. What happens when one bean is marked @Primary, but another bean is injected using @Qualifier.

**@Qualifier > @Primary**If both are present, @Qualifier wins — Spring injects the bean specified by @Qualifier, even if another bean is marked as @Primary.

**Priorities:-**

| **Case** | **Priority** |
| --- | --- |
| @Autowired without @Qualifier | Looks for one matching type. |
| @Autowired with @Qualifier | Looks for matching name/type based on Qualifier. |
| @Qualifier and @Primary | @Qualifier is given higher priority |
| Explicit property assignment (in XML or Java Config) | Highest Priority. |

## Q19. What is Spring BOM?

**BOM** stands for **Bill of Materials** — it is a **special kind of Maven POM** used to manage versions of dependencies in a centralized and consistent way.

In **Spring**, a BOM ensures that **all Spring modules** you use (like Spring Core, Spring Web, Spring Data, etc.) use **compatible versions** — so you don’t have to specify each version manually.

Add bom in pom.xml, under <dependencyManagement>

<dependencyManagement>  
<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-dependencies</artifactId>  
 <version>3.2.0</version> <!-- defines all spring lib versions -->  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
</dependencies>  
</dependencyManagement>

Now, you can add dependencies without **specifying versions.**

<dependencies>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
</dependencies>

Spring will **automatically use compatible versions** as defined in the BOM.

## Q19. What all design patterns are used in spring?

**BOM** stands for **Bill of Materials** — it is a **special kind of Maven POM** used to manage versions of dependencies

Creational Design patterns:

| **Pattern** | **One-Liner Description** |
| --- | --- |
| **Singleton** | Spring beans are singleton by default — one shared instance per container. |
| **Factory Method** | Spring uses factory methods (like @Bean or XML <bean factory-method>) to create objects. Used by BeanFactory and application context |
| **Builder** | Used internally in Spring MVC for building objects like UriComponentsBuilder. |
| **Prototype** | Defines a new bean instance every time it's requested (@Scope("prototype")). |

Structural Design patterns:

| **Pattern** | **One-Liner Description** |
| --- | --- |
| **Adapter** | Converts interfaces like JDBC to a consistent Spring interface (JdbcTemplate). |
| **Proxy** | Used in AOP (Aspect-Oriented Programming) to wrap method calls with additional logic. |
| **Decorator** | Adds additional behavior to beans using AOP or filters without altering the original class. |
| **Facade** | Provides simplified APIs (e.g., JdbcTemplate hides low-level JDBC details). |

Behavioral Design patterns:

| **Pattern** | **One-Liner Description** |
| --- | --- |
| **Template Method** | Defines the skeleton in Template classes like JdbcTemplate, letting you fill specific steps. |
| **Observer** | Implemented via Spring Events (publish-subscribe model). |
| **Strategy** | Used in components like ViewResolver or Validator to choose an algorithm at runtime. |
| **Command** | Used in Runnable beans or method references passed around in Spring. |
| **Chain of Responsibility** | Seen in Spring Security filters or HandlerInterceptors in MVC. |