Palash Bajpai

Spring

Interview Questions

Table of Contents

[Q1. What is RestController and how it is different from controller? 2](#_Toc197068520)

[Q2. How to show html page via controller? 3](#_Toc197068521)

## 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.

What is bean

Scopes of bean

Roles of IoC container.

BeanFactory vs ApplicationContext

What is autowiring

**How Autowiring works internally**

**How java based configurations are better than xml based configuration**

| **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. |

### ✅ 10. **What is the difference between @Autowired and @Qualifier?**

* @Autowired – injects dependency by type.
* @Qualifier – used with @Autowired to specify which bean to inject when multiple candidates exist.