GUIDE TO CLEAR

SPRING-BOOT MICROSERVICE INTERVIEW

AJAY RATHOD

Your Comprehensive Guide to Nailing Spring-Boot and Microservice Interviews

Sample copy

Contents

Introduction

Welcome to "**Guide to Clear Spring Boot Microservice Interviews**," a comprehensive guide designed to help Java developers excel in their interviews focused on Spring Boot microservices.

"As a Java Developer, I've experienced the real challenges of preparing for Spring Boot Microservice interviews—it's a vast topic that demands comprehensive readiness.

Throughout my preparation and job-hunting journey, I've discovered that, alongside Spring Boot and Microservices, interview questions often revolve around the following related topics:

- Spring Cloud
- Spring Boot with AWS
- Knowledge of AWS cloud
- Continuous Integration and Continuous Deployment (CICD)
- Jenkins
- Kubernetes, Docker, Message broker
- Middleware
- Reactive Spring

Recognizing the significance of these areas, I've included them in the book to ensure comprehensive preparation for interviews."

The Spring Boot framework has gained immense popularity for building robust and scalable microservices. Its simplicity, convention-over-configuration approach, and powerful features have made it a go-to choose for developing modern applications. However, cracking interviews that assess your proficiency in Spring Boot microservices requires a solid understanding of not only the framework itself but also related concepts, design patterns, RESTful APIs, databases, and more.

This book is structured to cover a wide range of topics, starting with the fundamentals of the Spring Framework. We'll dive into dependency injection,

bean lifecycle, scopes, and handling exceptions in Spring. With this foundation in place, we'll then explore the Spring Boot framework in detail, understanding its annotations, profiles, component scanning, and auto-configuration capabilities.

As we progress, we'll delve into microservice architecture and its advantages over monolithic applications. We'll discuss the design principles of microservices, along with various patterns and strategies for implementing them effectively. Topics like distributed tracing, communication between microservices, handling security, and fault tolerance mechanisms will be thoroughly explored.

To help you grasp the practical aspects, we'll also cover the development of RESTful APIs, focusing on HTTP methods, request/response handling, security, and best practices. In addition, we'll explore Spring Cloud, Spring Boot with AWS Scenario based interview question on spring boot microservice and commonly encountered in microservice interviews.

Each chapter is designed to provide a comprehensive understanding of the topic at hand, and to reinforce your knowledge, we've included a wide range of interview questions throughout the book. These questions cover both theoretical concepts and practical scenarios, allowing you to test your understanding and prepare for the challenging questions you may encounter in real-world interviews.

Whether you are aiming for a junior or senior-level position, "Guide to Clear Spring Boot Microservice Interviews" will serve as your go-to resource, providing you with the insights, knowledge, and confidence needed to excel in your interviews and land your dream job.

Best of luck on your interview preparation, and let's get started!

Best Regards,

Ajay Rathod

Why You Should Prepare for Spring-Boot & Microservice Interview

As a backend developer, I can offer several reasons why preparing for a Spring Boot microservices interview is crucial.

Foremost, relying solely on core Java skills won't suffice in your career or daily job. Why? Because Java technology invariably serves as the backbone in most backend server applications, employing frameworks like Spring, Spring Boot, Hibernate, Struts, among others.

During my early career days, I had to delve into frameworks like Spring Boot and Hibernate for my current job tasks and interviews. It's safe to say, Java interviews almost always include questions on Spring Boot and microservices.

Learning and preparing these frameworks yield numerous advantages. They foster technical career growth and ease transitions between jobs. I often receive queries from experienced developers aiming to switch jobs but struggle due to their lack of familiarity with Spring Boot microservice frameworks. They seek guidance on where to commence learning and project building.

It's invaluable to gain hands-on experience with Spring or Spring Boot projects, as practical work surpasses theoretical learning. Hence, I encourage my readers to both learn and work on projects within these technologies.

Regardless of your experience level, this book provides insights into interview expectations, serving as a knowledge repository. If you opt for a learning approach, perusing the chapters sequentially will offer a comprehensive understanding of crucial topics.

For those pursuing Java developer roles where Spring Boot microservices are a desired skill, this book serves as a valuable resource, aiding in quicker preparation.

Along with Spring-Boot, Microservice is one more topic which is inseparable from it, mostly in Microservice environment spring boot is used as implementation and having good knowledge on this microservice is crucial.

How to Prepare for Spring-Boot & Microservice Interview

In this chapter, I'll outline a strategy to excel in Spring Boot and Microservice interviews—a strategy that has proven effective and can significantly benefit you as well.

Firstly, establishing a solid foundation in the Spring framework, Restful Web Services, Spring MVC, Spring Data JPA, and Spring Cloud is essential. These form the basics necessary to crack the Spring Boot Microservice Interview.

If you're already familiar with the Spring framework, understanding concepts like the Spring IOC container, dependency injection, Autoconfiguration, Actuators, starter dependency, and others is advantageous. If not, start learning and practice by writing programs related to these concepts.

For newcomers, I recommend exploring YouTube tutorials and Udemy courses. The Spring Boot microservices community offers ample support online, providing numerous free resources to kickstart your learning journey.

Here's how I began:

Prerequisites:

Solid understanding of Core Java and Restful Web Services.

Understand Core Concepts: Ensure a firm grasp of Spring Boot fundamentals, covering aspects like dependency injection, annotations, auto-configuration, and Spring Boot starters.

RESTful APIs: Comprehend the design, implementation, and consumption of RESTful APIs using Spring Boot, emphasizing resource mapping, HTTP methods, and data serialization.

Database Integration: Possess knowledge about integrating databases with Spring Boot, encompassing ORM tools like Hibernate/JPA, database transactions, and data manipulation.

In Spring Boot, these topics carry significance:

Spring Boot Data JPA

Spring Boot Security

Spring Cloud

Spring Boot Testing

Spring Boot Caching

Spring Boot with AWS

Additionally, common questions often revolve around Global Exceptional handler, Actuators, Starter dependency, configuration management, and connecting databases using Spring-Boot.

Microservices Architecture: Acquaint yourself with microservices architecture principles, such as service discovery, API gateways, fault tolerance, and distributed data management, as these frequently appear in interviews alongside questions about Spring, Spring Boot, and microservices.

Testing and Documentation: Familiarize yourself with testing microservices using tools like JUnit, Mockito, and Swagger for comprehensive API documentation ensuring functional and API endpoint coverage.

Security and Authorization: Understand security mechanisms such as OAuth, JWT, and Spring Security to secure microservices and implement authorization.

Deployment and Monitoring: Learn about containerization using Docker, orchestration with Kubernetes, and monitoring tools like Spring Boot Actuator and Prometheus.

Review Common Interview Questions: Get acquainted with common Spring Boot and microservices interview questions, encompassing design patterns, best practices, and troubleshooting scenarios—topics covered in this book.

Participate in as many interviews as possible to grasp the latest trends and note down topics for focused preparation. This strategy worked wonders for me.

This book documents real interview questions to better equip you.

Lastly, concentrate on the topics you're most confident in. Your responses will shape the interview's outcome. Best of luck!

Chapter 1: Spring-Framework

What is Spring Framework?

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.

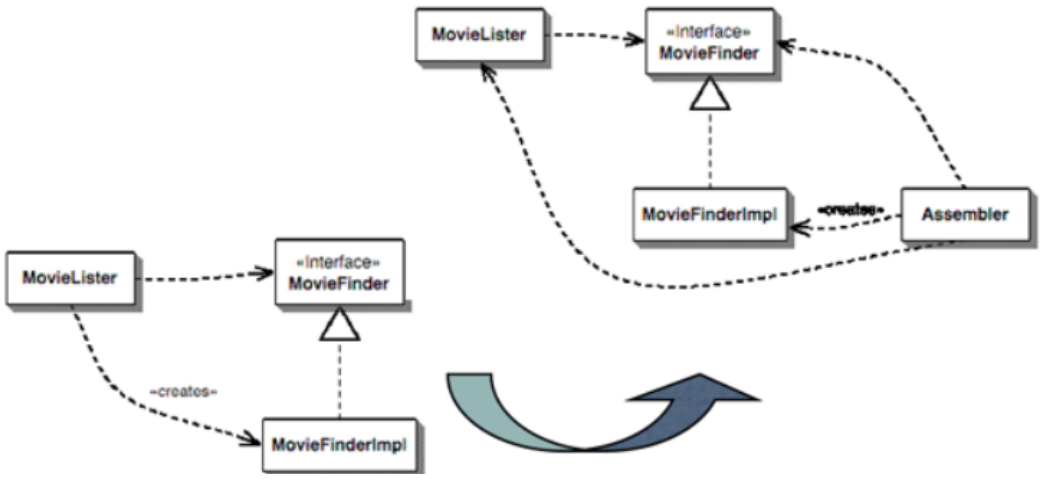
A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

What is Inversion of control?

In traditional programming, you would have to manually create and manage all the objects that your application needs. This can be a complex and error-prone process.

This diagram shows how IOC looks like,

Don't let the object create itself the instances of the object that it references. This job is delegated to the container (assembler in the picture).



The Spring Framework's IoC container simplifies this process by taking over the responsibility of creating and managing objects. You simply tell Spring what objects you need, and it will create them for you and provide them to your code. This is called dependency injection.

Dependency injection makes your code more modular and easier to maintain. It also reduces the risk of errors, because you no longer have to worry about creating objects correctly.

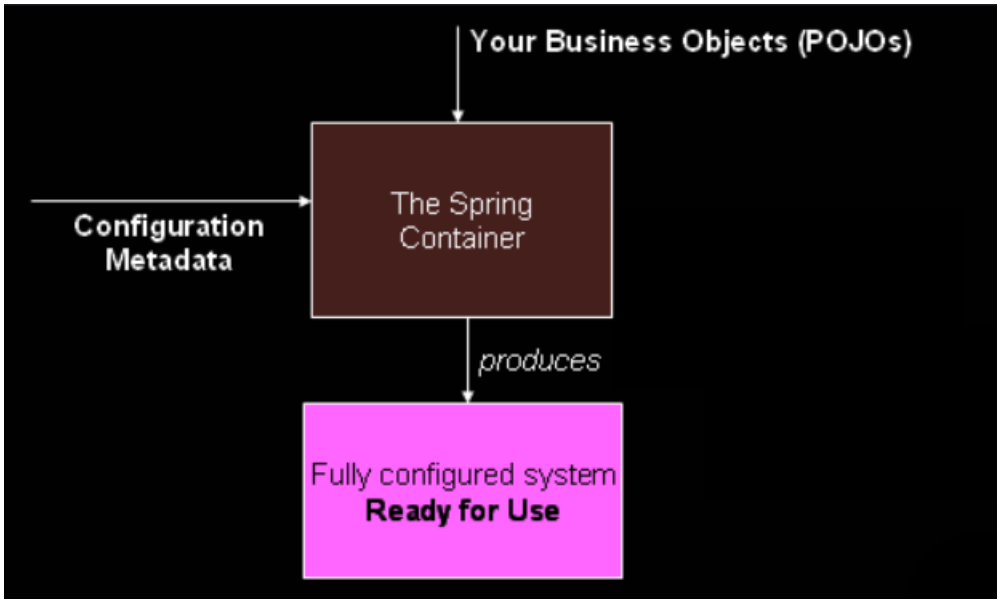
IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a

factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes

What is Spring IOC Container?

The org.springframework.context.ApplicationContext interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.



In How many ways we can define the configuration in spring?

We can do that in two ways

1. XML based config : It can be useful to have bean definitions span multiple XML files. Often, each individual XML configuration file represents a logical layer or module in your architecture.

2. Java-based configuration: define beans external to your application classes by using Java rather than XML files. To use these features, see the @Configuration, @Bean, @Import, and @DependsOn annotations.

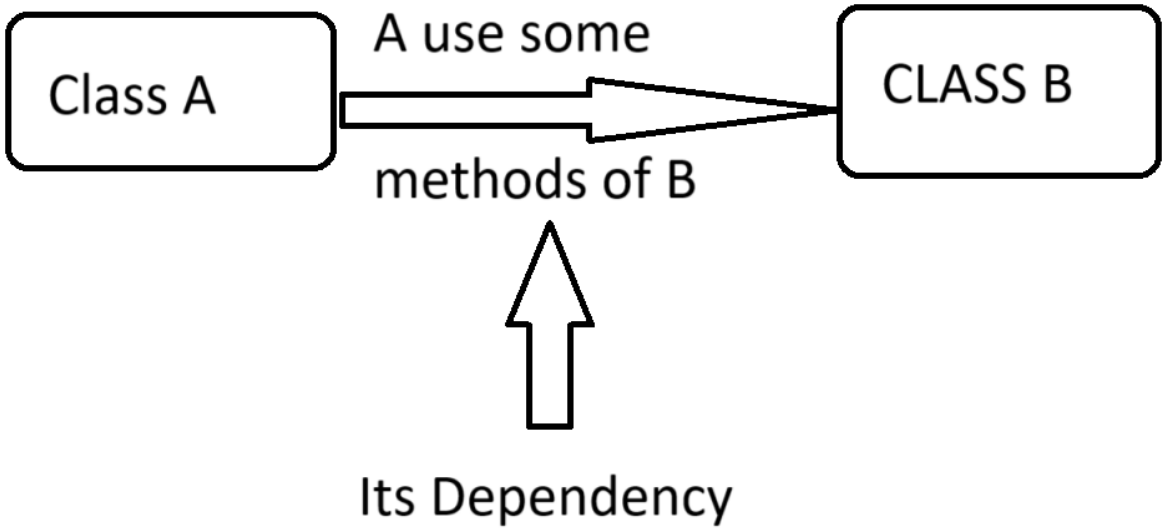
What is Dependency injection?

Dependency injection (DI) is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are

set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.

Code is cleaner with the DI principle, and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies. As a result, your classes become easier to test, particularly when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants: Constructor-based dependency injection and Setter-based dependency injection. Typically, Diagram Looks like this,



Difference between Inversion of Control and Dependency Injection?

Feature	Inversion of Control (IoC)	Dependency Injection (DI)
Concept	Design principle	Implementation pattern
Focus	Overall control flow of the application	Providing dependencies to objects
Goal	Decouple objects and increase flexibility	Improve maintainability and testability
Mechanism	External entity (e.g., framework) controls object creation and dependencies	Inject dependencies into objects through constructors, setters, or method arguments

Relationship	IoC is the broader concept; DI is a specific way to implement IoC	DI is a tool used to achieve IoC principles
Benefits	Increased flexibility, testability, and maintainability	Improved code organization, reduced coupling, and easier testing

What are the types of dependency injection and what benefit we are getting using that?

Dependency injection (DI) is a design pattern that allows objects to be supplied with their dependencies, rather than having to create them themselves. There are several types of dependency injection, each with its own benefits:

Constructor injection: In this type of injection, the dependencies are passed to the constructor of the class when it is instantiated. This ensures that the class always has the required dependencies and can be useful for enforcing class invariants.
The following example shows a class that can only be dependency-injected with constructor injection:

```
public class SimpleMovieLister {

	// the SimpleMovieLister has a dependency on a MovieFinder
	private final MovieFinder movieFinder;

	// a constructor so that the Spring container can inject a MovieFinder
	public SimpleMovieLister(MovieFinder movieFinder) {
		this.movieFinder = movieFinder;
	}

	// business logic that actually uses the injected MovieFinder is omitted...
}
```

Setter injection: In this type of injection, the dependencies are passed to setter methods of the class after it has been instantiated. This allows the class to be reused in different contexts, as the dependencies can be changed at runtime.
e.g.

```
public class SimpleMovieLister {

	// the SimpleMovieLister has a dependency on the MovieFinder
	private MovieFinder movieFinder;

	// a setter method so that the Spring container can inject a MovieFinder
	public void setMovieFinder(MovieFinder movieFinder) {
		this.movieFinder = movieFinder;
	}
```

```
    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Which one is better Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the @Autowired annotation on a setter method can be used to make the property be a required dependency; however, constructor injection with programmatic validation of arguments is preferable.

The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not null. Furthermore, constructor-injected components are always returned to the client (calling) code in a fully initialized state.

What is Method Injection?

Method Injection is a design pattern commonly used in Spring and other frameworks to inject dependencies into objects. Unlike field injection or constructor injection, which inject dependencies directly into fields or constructors, method injection injects dependencies into specific method arguments.

How inversion of control works inside the container?

Inversion of Control (IoC) is a design pattern that allows control to be transferred from the application code to an external container. In the context of a Java application, this container is often referred to as an IoC container or a dependency injection (DI) container.

IoC containers are responsible for creating and managing objects, and they do this by relying on a set of configuration rules that define how objects are created and wired together.

Here's how IoC works inside an IoC container:

Configuration: In order to use an IoC container, you need to configure it with a set of rules that define how objects should be created and wired together. This configuration is typically done using XML or Java annotations.

Object creation: When your application requests an object from the container, the container uses the configuration rules to create a new instance of the requested object.

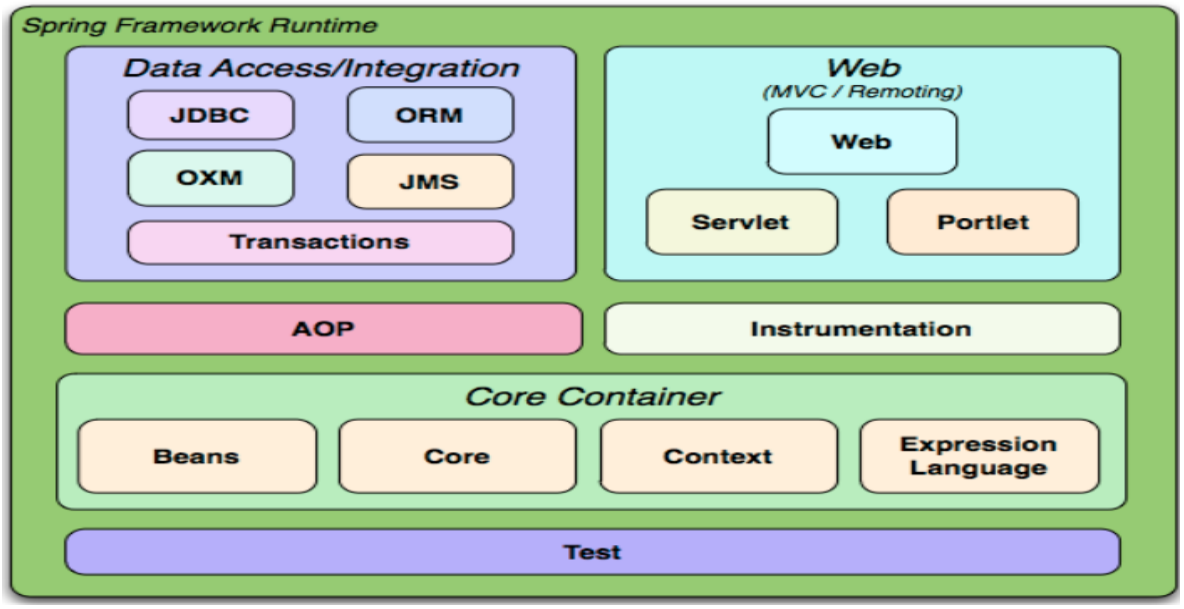
Dependency injection: The container injects any required dependencies into the newly created object. These dependencies are typically defined in the configuration rules.

Object lifecycle management: The container manages the lifecycle of the objects it creates. This means that it's responsible for creating, initializing, and destroying objects as required by the application.

Inversion of control: By relying on the container to create and manage objects, the application code no longer has direct control over the object creation process. Instead, the container takes on this responsibility, and the application code simply requests the objects it needs from the container.

What are different spring modules?

The Spring Framework is a comprehensive Java framework that consists of several modules, each providing a specific set of features and functionalities. These modules are organized into categories based on their primary purpose. Here's an overview of the major Spring modules and their respective roles:



Core Container: The Core Container module forms the foundation of the Spring Framework, providing essential services like dependency injection, bean lifecycle management, and resource management. It's responsible for creating, configuring, and managing objects throughout the application.

Data Access/Integration: The Data Access/Integration module focuses on simplifying data access and integration with various data sources, including relational databases, object-relational mapping (ORM) frameworks, and messaging systems. It provides features like JDBC abstraction, ORM integration, and message-oriented middleware (MOM) support.

Web (MVC/Remoting): The Web module caters to building web applications using the Model-View-Controller (MVC) pattern and provides remoting capabilities for distributed applications. It includes support for various web technologies like Servlet, Portlet, and Struts.

AOP (Aspect-Oriented Programming): The AOP module enables aspect-oriented programming, a technique for modularizing cross-cutting concerns like

logging, security, and transaction management. It provides features like aspect declaration, aspect weaving, and aspect execution.

Instrumentation: The Instrumentation module facilitates monitoring and performance management of Spring applications. It provides features like bean lifecycle tracing, memory profiling, and performance metrics collection.

Test: The Test module offers tools and frameworks for testing Spring applications, including support for dependency injection in unit tests, integration tests, and web application tests.

Here is the diagram to visualize the spring framework runtime as per module

What are Spring MVC, Spring AOP and Spring Core modules?

Reason to ask this question is, these three modules are very important while developing the spring-based application.

Spring MVC, Spring AOP, and Spring Core are three essential modules of the Spring Framework that play crucial roles in building robust and scalable Java applications.

Spring MVC (Model-View-Controller)

Spring MVC is a web framework that implements the Model-View-Controller (MVC) architecture, a popular pattern for separating application logic, user interface presentation, and data management. It provides a layered approach to handling web requests, making it easier to develop maintainable and testable web applications.

Key features of Spring MVC include:

Dispatcher Servlet: Centralizes request handling and dispatching to appropriate controllers.

Controller classes: Handle user requests by processing data, interacting with the model, and selecting appropriate views.

View technologies: Supports various templating engines like JSP, FreeMarker, Thymeleaf, and Velocity to render dynamic content.

Spring AOP (Aspect-Oriented Programming)

Spring AOP provides an implementation of aspect-oriented programming (AOP), a technique for modularizing cross-cutting concerns like logging, security, and transaction management. It allows developers to encapsulate these concerns as aspects and apply them to specific points within the application's execution flow.

Key features of Spring AOP include:

Aspect declaration: Defines aspects, specifying the cross-cutting concern and the pointcuts where it should be applied.

Aspect weaving: Integrates aspects into the application's execution flow, applying the aspect's behaviour at specific join points.

Aspect execution: Handles the invocation of aspect advice, which defines the actions to be taken at the join points.

Spring Core

Spring Core is the foundation of the Spring Framework, providing essential services like dependency injection, bean lifecycle management, and resource management. It's responsible for creating, configuring, and managing objects throughout the application.

Key features of Spring Core include:

Dependency injection: Automatically supplies objects with their dependencies, reducing code complexity and improving modularity.

Bean lifecycle management: Handles the creation, initialization, destruction, and scope of objects within the Spring application context.

Resource management: Manages resources like database connections, files, and messaging queues, simplifying resource acquisition and release.

How Component Scanning(@ComponentScan) Works?

Annotation Discovery: The Spring Framework uses annotation-based configuration to identify Spring beans. It scans the specified packages and subpackages for classes annotated with @Component, @Service, @Repository, @Controller, or other stereotype annotations that indicate a bean's role in the application.

Bean Creation and Registration: Upon discovering annotated classes, the Spring Framework creates instances of those classes and registers them as Spring beans in the application context. The application context maintains a registry of all managed beans, making them accessible for dependency injection.

Bean Configuration: The Spring Framework applies default configuration rules to the registered beans. These rules include dependency injection, bean lifecycle management, and resource management. Developers can further customize bean configuration using annotations, XML configuration files, or programmatic configuration.

e.g.

Configuration Class:

```
@Configuration

@ComponentScan("com.example. springapp")

public class AppConfig {
```

}

This configuration class defines two annotations:

- @Configuration: Marks this class as a source of bean definitions for Spring.
- @ComponentScan: Specifies the base package for component scanning. Spring will scan this package and its sub-packages for classes annotated with @Component, @Service, @Repository, or @Controller, and register them as Spring beans.

What is ApplicationContext and how to use inside spring?

ApplicationContext is the core container for managing beans and providing services to Spring applications. It simplifies bean configuration, dependency injection, and resource management.

This how we should use.

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

MyService myService = context.getBean("myService", MyService.class);

myService.doSomething();

context.close();
```

This code shows how to create an ApplicationContext from an XML configuration file, retrieve a bean by name, invoke a method on the bean, and close the ApplicationContext.

What is BeanFactory and how to use inside spring?

BeanFactory is an interface that represents a basic container for managing beans in a Spring application. It provides methods for creating, retrieving, and configuring beans. While ApplicationContext is a more advanced and commonly used container, BeanFactory offers a simpler interface for basic bean management.

Here is example,

```
BeanFactory factory = new XmlBeanFactory("applicationContext.xml");

MyService myService = factory.getBean("myService", MyService.class);

myService.doSomething();
```

What is Bean?

Bean:

In Spring, a bean is a managed object within the Spring IoC container. It is an instance of a class that is created, managed, and wired together by the Spring

framework. Beans are the fundamental building blocks of Spring applications and are typically configured and defined using annotations or XML configuration.

What are Bean scopes?

In Spring Framework, a bean scope defines the lifecycle and the visibility of a bean within the Spring IoC container. Spring Framework provides several built-in bean scopes, each with a specific purpose and behaviour.
The following are the most commonly used bean scopes in Spring Framework:
Singleton:

(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.

Prototype:

Scopes a single bean definition to any number of object instances.

Request:

Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.

Session:

Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

Application:

Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

Websocket:

Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

What is the Spring bean lifecycle?

In Spring Framework, a bean is an object that is managed by the Spring IoC container. The lifecycle of a bean is the set of events that occur from its creation until its destruction.

- The Spring bean lifecycle can be divided into three phases: instantiation, configuration, and destruction.
- Instantiation: In this phase, Spring IoC container creates the instance of the bean. Spring Framework supports several ways of instantiating a bean, such as through a constructor, a static factory method, or an instance factory method.

- Configuration: In this phase, Spring IoC container configures the newly created bean. This includes performing dependency injection, applying any bean post-processors, and registering any initialization and destruction call-backs.
- Destruction: In this phase, Spring IoC container destroys the bean instance. It is the last phase of the Spring bean lifecycle.

In addition to these three phases, Spring Framework also provides several callbacks that allow developers to specify custom initialization and destruction logic for a bean. These callbacks include:

- @PostConstruct: Invoked after the bean has been constructed and all dependencies have been injected
- init-method: Specifies a method to be called after the bean has been constructed and all dependencies have been injected
- destroy-method: Specifies a method to be called just before the bean is destroyed.
- @PreDestroy: Invoked before the bean is destroyed.

The Spring bean lifecycle is controlled by the Spring IoC container, which creates, configures, and manages the lifecycle of the beans. Developers can take advantage of the bean lifecycle callbacks to add custom initialization and destruction logic to their beans, making it easier to manage the lifecycle of their objects and ensuring that resources are properly.

What is the bean lifecycle in terms of application context?

The bean lifecycle within an application context refers to the various stages a bean goes through from its creation to its destruction. The application context is responsible for managing this lifecycle, ensuring that beans are properly initialized, used, and destroyed at the appropriate times.

Stages of the Bean Lifecycle:

- Bean Creation: The bean is instantiated, either through constructor injection, setter-based injection, or other mechanisms.
- Bean Configuration: The bean's properties are set and any necessary initialization callbacks are invoked.
- Bean Usage: The bean is used by the application, providing its designated functionality.
- Bean Destruction: The bean is destroyed when it is no longer needed, releasing resources and performing any necessary cleanup tasks.

Application Context's Role in Bean Lifecycle:

- The application context plays a central role in managing the bean lifecycle, providing various mechanisms to control the creation, configuration, usage, and destruction of beans.
- Bean Configuration Metadata: The application context stores configuration metadata for beans, defining how they should be created, configured, and managed.

- Bean Lifecycle Hooks: The application context provides hooks to define custom behavior at various stages of the bean lifecycle, such as initialization and destruction callbacks.
- Bean Scope Management: The application context manages the scope of beans, determining their visibility and lifetime within the application.
- Bean Lifecycle Listeners: The application context supports bean lifecycle listeners, allowing components to be notified of changes in the lifecycle of other beans.

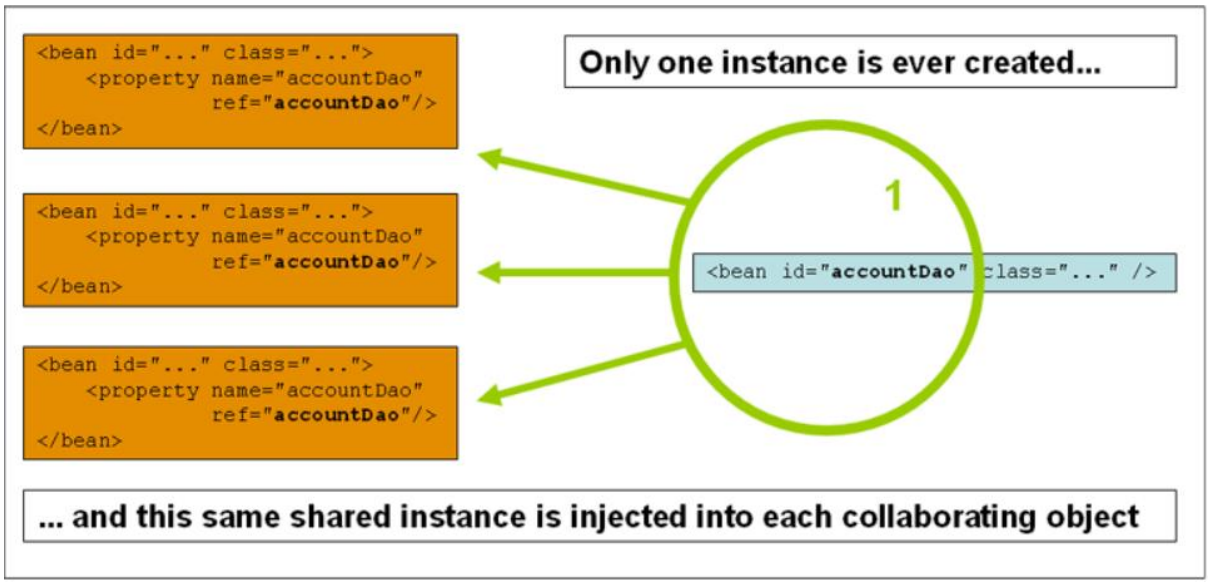
Difference Between BeanFactory and ApplicationContext?

Feature	BeanFactory	ApplicationContext
Functionality	Basic bean management	Extended bean management with additional services
Bean loading	On-demand	All at startup (by default)
Supported scopes	Singleton, Prototype	Singleton, Prototype, Request, Session, etc.
Services	Limited	Message resources, events, etc.
Use cases	Simple applications	Complex applications with advanced needs

What is default bean scope in spring?

The default bean scope in Spring is singleton. This means that only one instance of a bean will be created and managed by the Spring container, regardless of how many times it is requested. This scope is useful for beans that are stateless, meaning that they do not maintain any internal state and can be safely shared across different parts of the application.

Below diagram how one instance is created,



How bean is loaded inside spring, can you tell the difference between Lazy loading and Eager loading?

Bean Loading in Spring:

The Spring Framework uses a process called bean instantiation to create beans and make them available to the application. This process involves the following steps:

Scanning for beans: The Spring Framework scans the application context for classes that are annotated with Spring annotations such as @Component, @Service, @Repository, and @Controller.

Creating bean instances: The Spring Framework creates an instance of each bean class that it finds.

Configuring beans: The Spring Framework configures each bean by setting its properties and injecting its dependencies.

Initializing beans: The Spring Framework initializes each bean by calling its initialization callbacks.

Lazy Loading vs. Eager Loading (Important interview Question)

In Spring, beans can be either lazy loaded or eager loaded. Lazy loading means that a bean is not created until it is first requested by the application. Eager loading means that a bean is created when the application starts up.

The default behavior in Spring is to lazy load beans. This can improve the performance of the application because it means that only the beans that are actually needed are created. However, lazy loading can also make the application more difficult to debug because it can be hard to track down which beans have been created and when.

Eager loading can be useful for beans that are needed by the application as soon as it starts up. For example, an application might need to connect to a database as soon as it starts up. In this case, it would make sense to eagerly load the bean that is responsible for connecting to the database.

How to Specify Lazy Loading and Eager Loading

You can specify whether a bean should be lazy loaded or eagerly loaded using the @Lazy annotation. For example, the following code will create a bean that is lazy loaded:

```
@Lazy

@Service

public class MyService {

}
```

The following code will create a bean that is eagerly loaded:

```
@Service
public class MyEagerService {
}
```

when to use lazy loading and eager loading:

Use lazy loading for beans that are not needed by the application as soon as it starts up.

Use eager loading for beans that are needed by the application as soon as it starts up.

Use eager loading for beans that have a high startup cost.

Use lazy loading for beans that are not frequently used.

How @Autowire annotation works?

The @Autowired annotation works by using reflection to find the appropriate dependency to inject. It will first look for a bean with the same type as the dependency. If it cannot find a bean of the same type, it will look for a bean with a qualifier that matches the name of the dependency. If it still cannot find a bean to inject, it will throw an exception.

What are the Types of Autowiring?

There are four types of autowiring that the @Autowired annotation supports:

- byType: This is the default type of autowiring. The Spring Framework will look for a bean with the same type as the dependency.
- byName: The Spring Framework will look for a bean with the same name as the dependency.
- byConstructor: The Spring Framework will look for a constructor that takes a single argument of the type of the dependency.
- byQualifier: The Spring Framework will look for a bean with a qualifier that matches the specified value.

How to exclude a Bean from Autowiring?

In Spring's XML format, set the autowire-candidate attribute of the <bean/> element to false. The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as @Autowired).

Difference between @Autowire and @Inject in spring?

The @Autowired and @Inject annotations are both used for dependency injection in Spring applications. However, there are some key differences between the two annotations.

@Autowired(Use when developing a Spring application)

The @Autowired annotation is a Spring-specific annotation. It is used to inject dependencies into beans that are managed by the Spring container. The @Autowired annotation can be used with all four types of autowiring: byType, byName, byConstructor, and byQualifier.

@Inject(Use when developing a non-Spring application)

The @Inject annotation is a standard JSR-330 annotation. It is used to inject dependencies into beans that are managed by any dependency injection container that supports JSR-330. The @Inject annotation can only be used with byType and byName autowiring.

Is Singleton bean thread safe?

No, singleton beans are not thread safe by default. This means that if two threads try to access a singleton bean at the same time, they may get conflicting results. This is because singleton beans are shared across all threads, and any changes made to a singleton bean by one thread will be visible to all other threads.

Difference between Singleton and prototype bean?

Feature	Singleton	Prototype
Scope	Application-wide	Per-request
Instances	One	Multiple
Use cases	Stateless beans	Stateful beans

What is @Bean annotation in spring?

The @Bean annotation is a Spring Framework annotation that marks a method as a bean definition. Bean definitions tell the Spring container how to create and manage beans. When a method is annotated with @Bean, the Spring container will create an instance of the bean class returned by the method and manage its lifecycle.

```
@Configuration
public class MyConfiguration {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

In this example, the myService() method is annotated with @Bean. This tells the Spring container to create an instance of the MyServiceImpl class and manage its lifecycle. The bean can then be injected into other beans using the @Autowired annotation.

What is @Configuration annotation?

The @Configuration annotation is a Spring Framework annotation that marks a class as a configuration class. Configuration classes are used to define beans in a Spring application. When a class is annotated with @Configuration, the Spring container will scan the class for methods that are annotated with @Bean and use those methods to define beans.

```
@Configuration
public class MyConfiguration {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

How to configure Spring profiles?

Spring profiles provide a way to segregate parts of your application configuration and make them only available in certain environments

There are two ways to configure Spring profiles:

1. Using the Spring Boot Properties File

The Spring Boot properties file, typically named application.properties or application.yml, is a convenient way to configure Spring profiles. To configure a profile using the properties file, simply add the following property:

e.g.

```
spring.profiles.active=profile1,profile2
```

What is @component and @profile and @value annotation?

Feature	@Component	@Profile	@Value
Purpose	Indicates that a class is a component	Indicates that a bean is only available in certain environments	Injects property values into beans
Scope	Bean scope	Profile scope	Property scope
Usage	Any class that is a bean	Any bean that is annotated with @Component, @Configuration, or @Bean	Any field or method parameter

What is $ and # do inside @value annotation?

The $ symbol is used to inject property values from various sources, such as properties files, environment variables, and system properties. For example, the following code injects the value of the app.name property from the application.properties file:

e.g.

@Value("${app.name}")

private String appName;

The # symbol is used to access SpEL expressions. SpEL (Spring Expression Language) is a powerful expression language that can be used to perform complex operations on property values. For example, the following code injects the value of the app.name property and converts it to uppercase:

e.g.

@Value("#{'${app.name}'.toUpperCase()}")

private String appNameUppercase;

What is the stateless bean in spring? name it and explain it.

A stateless bean in Spring Framework is a bean that does not maintain any state between method invocations. This means that the bean does not store any information about the previous invocations, and each method call is handled independently.

Stateless beans are typically used for services that perform actions or calculations, but do not maintain any state between invocations. This can include services that perform mathematical calculations, access external resources, or perform other tasks that do not require the bean to maintain state.

Stateless beans can be implemented as singleton beans, and multiple clients can share the same instance of the bean. Since stateless beans do not maintain any state, they can be easily scaled horizontally by adding more instances of the bean to handle the increased load.

Stateless beans also have the advantage of being simpler and easier to reason about, since they do not have to worry about maintaining state between invocations. Additionally, since stateless beans do not maintain any state, they can be easily serialized and replicated for high availability and scalability.

How is the bean injected in spring?

In Spring, a bean is injected (or wired) into another bean using the Dependency Injection (DI) pattern. DI is a design pattern that allows a class to have its dependencies provided to it, rather than creating them itself.

Spring provides several ways to inject beans into other beans, including:
Constructor injection: A bean can be injected into another bean by passing it as a constructor argument. Spring will automatically create an instance of the dependent bean and pass it to the constructor.

```
public class BeanA {
    private final BeanB beanB;
    public BeanA(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Setter injection: A bean can be injected into another bean by passing it as a setter method argument. Spring will automatically call the setter method and pass the dependent bean.

```
public class BeanA {
    private BeanB beanB;
    @Autowired
    public void setBeanB(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Field injection: A bean can be injected into another bean by annotating a field with the @Autowired annotation. Spring will automatically set the field with the dependent bean.

```
public class BeanA {
    @Autowired
    private BeanB beanB;
}
```

Interface injection: A bean can be injected into another bean by implementing an interface. Spring will automatically set the field with the dependent bean.

```
public class BeanA implements BeanBUser {
    @Autowired
    private BeanB beanB;
}
```

It's important to note that, you can use any combination of the above methods, but you should choose the appropriate one depending on your use case.

Also, Spring uses a technique called Autowiring to automatically wire beans together, Autowiring can be done by type, by name, or by constructor.

By default, Spring will try to autowire beans by type, but if there are multiple beans of the same type, it will try to autowire by name using the bean's name

defined in the configuration file.

How to handle cyclic dependency between beans?

Let's say for example: Bean A is dependent on Bean B and Bean B is dependent on Bean A. How does the spring container handle eager & lazy loading?

A cyclic dependency between beans occurs when two or more beans have a mutual dependency on each other, which can cause issues with the creation and initialization of these beans.
There are several ways to handle cyclic dependencies between beans in Spring:
Lazy Initialization: By using the @Lazy annotation on one of the beans involved in the cycle, it can be initialized only when it is actually needed.

```
@Lazy
@Autowired
private BeanA beanA;
```

Constructor injection: Instead of using setter or field injection, you can use constructor injection, which will make sure that the dependencies are provided before the bean is fully initialized.

```
public class BeanA {
    private final BeanB beanB;

    public BeanA(BeanB beanB) {
        this.beanB = beanB;
    }
}
```

Use a proxy: A proxy can be used to break the cycle by delaying the initialization of one of the beans until it is actually needed. Spring AOP can be used to create a proxy for one of the beans involved in the cycle.

Use BeanFactory: Instead of injecting the bean directly, you can use BeanFactory to retrieve the bean when it's actually needed.

```
public class BeanA {
    private BeanB beanB;

    @Autowired
    public BeanA(BeanFactory beanFactory) {
        this.beanB = beanFactory.getBean(BeanB.class);
    }
}
```

What would you call a method before starting/loading a Spring boot application?

In Spring Boot, there are several methods that can be called before starting or loading a Spring Boot application. Some of the most commonly used methods are:

main() method: The main() method is typically the entry point of a Spring Boot application. It is used to start the Spring Boot application by calling the SpringApplication.run() method.

@PostConstruct method: The @PostConstruct annotation can be used to mark a method that should be called after the bean has been constructed and all dependencies have been injected. This can be used to perform any necessary initialization before the application starts.

CommandLineRunner interface: The CommandLineRunner interface can be implemented by a bean to run specific code after the Spring Application context has been loaded.

ApplicationRunner interface: The ApplicationRunner interface can be implemented by a bean to run specific code after the Spring Application context has been loaded and the Application arguments have been processed.

@EventListener : The @EventListener annotation can be used to register a method to listen to specific Application events like ApplicationStartingEvent, ApplicationReadyEvent and so on.

How to handle exceptions in the spring framework?

There are several ways to handle exceptions in the Spring Framework:

try-catch block: You can use a try-catch block to catch and handle exceptions in the method where they occur. This approach is useful for handling specific exceptions that are likely to occur within a particular method.

@ExceptionHandler annotation: You can use the @ExceptionHandler annotation on a method in a @Controller class to handle exceptions that are thrown by other methods in the same class. This approach is useful for handling specific exceptions in a centralized way across multiple methods in a controller.

@ControllerAdvice annotation: You can use the @ControllerAdvice annotation on a class to define a global exception handler for multiple controllers in your application. This approach is useful for handling specific exceptions in a centralized way across multiple controllers.

HandlerExceptionResolver interface: You can implement the HandlerExceptionResolver interface to create a global exception handler for your entire application. This approach is useful for handling specific exceptions in a centralized way across the entire application.

ErrorPage: You can define an ErrorPage in your application to redirect to a specific page when a certain exception occurs. This approach is useful for displaying a user-friendly error page when an exception occurs.

@ResponseStatus annotation: You can use the @ResponseStatus annotation on an exception class to define the HTTP status code that should be returned when the exception is thrown.

How filter work in spring?

In Spring, filters act as interceptors for requests and responses, processing them before and after they reach the actual application logic. They're like "gatekeepers" that can modify, deny, or allow requests based on predefined rules.

Here's how they work in a nutshell:

1. Configuration:

- You define filters as beans in your Spring configuration.
- Specify the order in which filters should be executed.

2. Request Flow:

- When a client sends a request, it goes through each filter in the specified order.
- Each filter can:
- Inspect the request headers, body, and other attributes.
- Modify the request content or headers.
- Decide to continue processing the request or terminate it.

3. Response Flow:

- Once the request reaches the application logic and receives a response, the response flows back through the filters in reverse order.
- Filters can again:
- Inspect the response headers and body.
- Modify the response content or headers.

4. Common Use Cases:

- Security filters: Validate user authentication, authorize access, and prevent security vulnerabilities.
- Logging filters: Log information about requests and responses for debugging and analysis.
- Compression filters: Compress responses to reduce bandwidth usage.
- Caching filters: Cache frequently accessed resources to improve performance.

Benefits:

- Intercept and modify requests and responses: Provide more control over application behavior.
- Centralize common tasks: Avoid duplicating code for security, logging, etc.
- Chain multiple filters: Achieve complex processing logic by combining multiple filters.

What is DispatcherServlet?

DispatcherServlet acts as the central "front controller" for Spring MVC applications. It is a Servlet that receives all incoming HTTP requests and delegates them to appropriate controller classes. The DispatcherServlet is responsible for identifying the appropriate handler method for each request and invoking it, ensuring that the request is processed correctly.

The following example of the Java configuration registers and initializes the DispatcherServlet, which is auto-detected by the Servlet container.

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

	@Override

	public void onStartup(ServletContext servletContext) {

		// Load Spring web application configuration

		AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();

		context.register(AppConfig.class);

		// Create and register the DispatcherServlet

		DispatcherServlet servlet = new DispatcherServlet(context);

		ServletRegistration.Dynamic registration =
servletContext.addServlet("app", servlet);

		registration.setLoadOnStartup(1);

		registration.addMapping("/app/*");

	}

}
```

What is @Controller annotation in spring?

The @Controller annotation is a Spring stereotype annotation that indicates that a class serves as a web controller. It is primarily used in Spring MVC applications to mark classes as handlers for HTTP requests. When a class is annotated with @Controller, it can be scanned by the Spring container to identify its methods as potential handlers for specific HTTP requests.

Spring MVC provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces. The following example shows a controller defined by annotations:

e.g.

```
@Controller
public class HelloController {
	@GetMapping("/hello")
	public String handle(Model model) {
		model.addAttribute("message", "Hello World!");
		return "index";
	}
}
```

How Controller maps appropriate methods to incoming request?

You can use the @RequestMapping annotation to map requests to controller methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.Request Mapping Process:

There are also HTTP method specific shortcut variants of @RequestMapping:

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

Request Reception: The DispatcherServlet receives an incoming HTTP request containing the request URI, HTTP method (GET, POST, PUT, DELETE, etc.), and request parameters.

Mapping Lookup: The DispatcherServlet utilizes a HandlerMapping component to lookup the appropriate handler method for the received request. The HandlerMapping maintains a registry of mappings between request patterns and handler methods.

Pattern Matching: The HandlerMapping compares the request URI and HTTP method against the registered request patterns. It uses pattern matching rules to identify the most specific matching pattern.

Handler Method Identification: Once the matching pattern is identified, the HandlerMapping retrieves the corresponding handler method from its registry. This handler method is the one responsible for handling the incoming request.

Method Invocation: The DispatcherServlet invokes the identified handler method, passing the request object as an argument. The handler method processes the request's logic and generates an appropriate response.

Response Handling: After the handler method completes its execution, the DispatcherServlet receives the generated response object. It prepares the response by setting appropriate headers and content, and sends the response back to the client.

This is the sample program,

```
@RestController
@RequestMapping("/persons")
class PersonController {
        @GetMapping("/{id}")
        public Person getPerson(@PathVariable Long id) {
                // ...
        }
        @PostMapping
        @ResponseStatus(HttpStatus.CREATED)
        public void add(@RequestBody Person person) {
                // ...
        }
}
```

Difference between @Requestparam and @Pathparam annotation?

@Requestparam:

You can use the @RequestParam annotation to bind Servlet request parameters (that is, query parameters or form data) to a method argument in a controller.

Here code example,

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {
	@GetMapping
	public String setupForm(@RequestParam("petId") int petId, Model model)
{
		Pet pet = this.clinic.loadPet(petId);
		model.addAttribute("pet", pet);
		return "petForm";
	}
}
```

Using @RequestParam we are binding petId

By default, method parameters that use this annotation are required, but you can specify that a method parameter is optional by setting the @RequestParam annotation's required flag to false or by declaring the argument with an java.util.Optional wrapper.

@Pathparam

Function:

It allows you to map variables from the request URI path to method parameters in your Spring controller.

This gives you a cleaner and more flexible way to handle dynamic data in your API.

Usage:

You place the @Pathparam annotation on a method parameter.

Inside the annotation, you specify the name of the variable in the URI path that should be bound to the parameter.

Feature	@RequestParam	@PathParam
Data Source	Query Parameters	Path Variables
Location	URL after ?	Embedded in URL Path

Required Parameters	Optional (default)	Required
Encoding	Decoded	Not Encoded
Usage	Additional Information, Filtering	Essential Resource Identifiers

What is session scope used for?

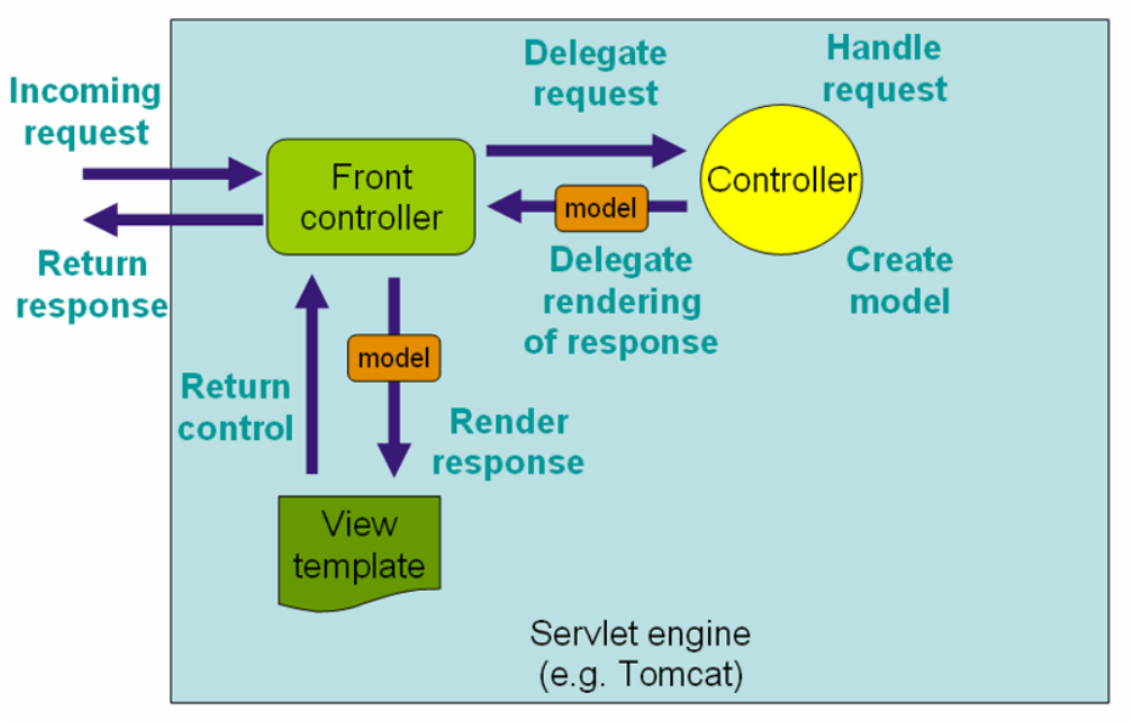
session scope is a way of managing the lifecycle of objects that are bound to a specific HTTP session. When an object is created in session scope, it is stored in the session and is accessible to all requests that belong to the same session. This can be useful for storing user-specific information or maintaining state across multiple requests.

Difference between @component @Service @Controller @Repository annotation?

Annotation	Role	Usage
@Component	General-purpose component	Any Spring-managed bean
@Service	Service layer component	Business logic components
@Controller	Presentation layer component	Web controller classes
@Repository	Data access layer component	DAO classes

Spring-MVC flow in detail?

Spring MVC is a popular web framework for building Java web applications. It provides a Model-View-Controller architecture that separates the application logic into three components: the model, the view, and the controller. The Spring MVC flow involves the following steps:



The requesting processing workflow in Spring Web MVC (high level)

Client sends a request: The user sends a request to the Spring MVC application through a browser or any other client application.

DispatcherServlet receives the request: The DispatcherServlet is a central controller in the Spring MVC architecture. It receives the request from the client and decides which controller should handle the request.

HandlerMapping selects the appropriate controller: The HandlerMapping component maps the request URL to the appropriate controller based on the URL pattern configured in the Spring configuration file.

Controller processes the request: The controller handles the request and performs the necessary processing logic. It may interact with the model component to retrieve data or update the data.

Model updates the data: The model component manages the data and provides an interface for the controller to retrieve or update the data.

ViewResolver selects the appropriate view: The ViewResolver component maps the logical view name returned by the controller to the actual view template.

View renders the response: The view template is rendered to generate the response. It may include data from the model component.

DispatcherServlet sends the response: The DispatcherServlet sends the response back to the client through the appropriate view technology, such as JSP, HTML, or JSON.

The Spring MVC flow is a cyclical process, as the client may send additional requests to the application, and the cycle repeats.

Can singleton bean scope handle multiple parallel requests?

A singleton bean in Spring has a single instance that is shared across all requests, regardless of the number of parallel requests. This means that if two requests are processed simultaneously, they will share the same bean instance and access to the bean's state will be shared among the requests.

However, it's important to note that if the singleton bean is stateful, and the state is shared among requests, this could lead to race conditions and other concurrency issues. For example, if two requests are trying to modify the same piece of data at the same time, it could lead to data inconsistencies.

To avoid these issues, it's important to make sure that any stateful singleton beans are designed to be thread-safe. One way to do this is to use synchronization or other concurrency control mechanisms such as the synchronized keyword, Lock or ReentrantLock classes, or the @Transactional annotation if the bean is performing database operations.

On the other hand, if the singleton bean is stateless, it can handle multiple parallel requests without any issues. It can be used to provide shared functionality that doesn't depend on the state of the bean.
In conclusion, a singleton bean can handle multiple parallel requests, but it's important to be aware of the state of the bean and to ensure that it's designed to be thread-safe if it has shared state.

Tell me the Design pattern used inside the spring framework.

The Spring Framework makes use of several design patterns to provide its functionality. Some of the key design patterns used in Spring are:

- Inversion of Control (IoC): This pattern is used throughout the Spring Framework to decouple the application code from the framework and its components. The IoC container is responsible for managing the lifecycle of beans and injecting dependencies between them.

- Singleton: A singleton pattern is used to ensure that there is only one instance of a bean created in the Spring IoC container. The singleton pattern is used to create a single instance of a class, which is shared across the entire application.

- Factory: The factory pattern is used in Spring to create objects of different classes based on the configuration. Spring provides a factory pattern to create beans, which is based on the factory method design pattern.

- Template Method: The template method pattern is used in Spring to provide a common structure for different types of operations. Spring provides several template classes such as JdbcTemplate, Hibernate Template, etc. that provide a common structure for performing database operations.

- Decorator: The decorator pattern is used in Spring to add additional functionality to existing beans. The Spring AOP (Aspect-Oriented Programming) module uses the decorator pattern to add additional functionality to existing beans through the use of proxies.

- Observer: The observer pattern is used in Spring to notify other beans of changes to the state of a bean. Spring provides the ApplicationEvent and ApplicationListener interfaces, which can be used to implement the observer pattern.

- Command: The command pattern is used in Spring to encapsulate the execution of a particular piece of code in a command object. This pattern is used in Spring to create reusable and testable code.

- Façade: The façade pattern is used in Spring to simplify the interface of a complex system. The Spring Framework uses the façade pattern to provide a simplified interface for interacting with its components.

These are just a few examples of the design patterns used in Spring, there are many more. Spring framework makes use of these patterns to provide a consistent and simple way to build applications, making it easier to manage complex systems.

How do factory design patterns work in terms of the spring framework?

In Spring, the factory design pattern is used to create objects of different classes based on the configuration. The Spring IoC container uses the factory pattern to create beans, which is based on the factory method design pattern.

The factory method is a design pattern that provides a way to create objects of different classes based on a factory interface. In Spring, the IoC container acts as the factory, and the factory interface is represented by the BeanFactory or ApplicationContext interfaces.

The IoC container is responsible for creating and managing the lifecycle of beans. When you define a bean in the configuration, the IoC container will use

the factory pattern to create an instance of the bean. The IoC container will then manage the lifecycle of the bean, including injecting dependencies, initializing the bean, and destroying the bean when it is no longer needed.

Here's an example of how you can define a bean in Spring using the factory design pattern:

```
@Configuration
public class MyConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

In this example, the myService() method is annotated with @Bean. This tells Spring to create an instance of the MyService class when the IoC container is created. The IoC container will use the factory pattern to create the instance and manage its lifecycle.

Another way to use factory pattern in spring is to use FactoryBean interface, which allows you to create beans that are created by a factory method, it's a factory of bean. The FactoryBean interface defines a single method, getObject(), which returns the object that should be exposed as the bean in the Spring application context.

How the proxy design pattern is used in spring?

The proxy design pattern is used in Spring to add additional functionality to existing objects. The Spring Framework uses the proxy pattern to provide AOP (Aspect-Oriented Programming) functionality, which allows you to add cross-cutting concerns, such as logging, security, and transaction management, to your application in a modular and reusable way.

In Spring, AOP proxies are created by the IoC container, and they are used to intercept method calls made to the target bean. This allows you to add additional behaviour, such as logging or security checks, before or after the method call is made to the target bean.

AOP proxies are created using one of three proxy types: JDK dynamic proxies, CGLIB proxies, or AspectJ proxies.

JDK dynamic proxies: This is the default proxy type in Spring, and it is used to proxy interfaces.

CGLIB proxies: This proxy type is used to proxy classes, and it works by creating a subclass of the target bean.
AspectJ proxies: This proxy type uses the AspectJ library to create proxies, and it allows you to use AspectJ pointcuts and advice in your application.
Spring uses the proxy pattern to provide AOP functionality by generating a proxy object that wraps the target bean. The proxy object will intercept method calls made to the target bean, and it will invoke additional behavior, such as logging or security checks, before or after the method call is made to the target bean.

Here's an example of how you can use Spring AOP to add logging to a bean:

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.example.service.*.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("Started method: " + joinPoint.getSignature().getName());
    }
}
```

In this example, the LoggingAspect class is annotated with @Aspect and @Component to make it a Spring bean. The @Before annotation is used to specify that the logBefore() method should be executed before the method call is made to the target bean. The logBefore() method uses the JoinPoint argument to log the name of the method that is being called.

What if we call singleton bean from prototype or prototype bean from singleton How many objects returned?

When a singleton bean is called from a prototype bean or vice versa, the behaviour depends on how the dependency is injected.

If a singleton bean is injected into a prototype bean, then each time the prototype bean is created, it will receive the same instance of the singleton bean. This is because the singleton bean is only created once during the startup of the application context, and that same instance is then injected into the prototype bean each time it is created.

On the other hand, if a prototype bean is injected into a singleton bean, then each time the singleton bean is called, a new instance of the prototype bean will be created. This is because prototype beans are not managed by the container, and a new instance is created each time a dependency is injected.

Here's an example to illustrate this:

```
@Component
@Scope("singleton")
public class SingletonBean {
  // code for singleton bean
}
@Component
@Scope("prototype")
public class PrototypeBean {
  @Autowired
  private SingletonBean singletonBean;
  // code for prototype bean
}
```

In this example, when a prototype bean is created and injected with the singleton bean, it will receive the same instance of the singleton bean each time it is created. However, if the singleton bean is created and injected with the prototype bean, it will receive a new instance of the prototype bean each time it is called.

It's important to note that mixing singleton and prototype scopes in a single application context can lead to unexpected behaviour and should be avoided unless necessary. It's best to use one scope consistently throughout the application context.

Spring boot vs spring why choose one over the other?

Here are some reasons to choose Spring Framework:

- You need a comprehensive set of features and capabilities for your application.
- You want to build a modular application where you can pick and choose only the components that you need.
- You need a high degree of flexibility and customization in your application.

Here are some reasons to choose Spring Boot:

- You want to quickly set up a stand-alone Spring application without needing to do a lot of configurations.
- You want to take advantage of pre-configured dependencies and sensible defaults.

- You want to easily deploy your application as a self-contained executable JAR file.

Overall, both Spring and Spring Boot are powerful frameworks that can be used to build enterprise-level applications. The choice between them depends on the specific needs of your application and the level of flexibility and customization that you require.