Explain Spring web MVC. Design and implement simple spring web MVC framework program to display Helloworld.

Spring Web MVC Overview

Spring Web MVC is a module of the Spring framework that provides a robust and flexible framework for building web applications. It follows the Model-View-Controller (MVC) design pattern, which separates the application logic into three interconnected components:

1. **Model**: Represents the application's data and business logic.
2. **View**: Represents the presentation layer and is responsible for displaying the data.
3. **Controller**: Handles user input and interactions, updates the model, and determines the view to render.

Key Components of Spring Web MVC

1. **DispatcherServlet**: Acts as the front controller and is responsible for handling all incoming HTTP requests and dispatching them to appropriate handlers/controllers.
2. **Controller**: Processes user requests and returns a ModelAndView object containing the model and view.
3. **Model**: Contains the data that the view will display.
4. **View**: Renders the output using the data from the model.
5. **ViewResolver**: Maps logical view names to actual view implementations.

Steps to Implement a Simple Spring Web MVC Application

Let's create a simple Spring Web MVC application to display "Hello World".

1. Project Structure

```
SpringHelloWorld
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── example
│   │   │           └── SpringHelloWorld
│   │   │               ├── controller
│   │   │               │   └── HelloWorldController.java
│   │   │               └── SpringHelloWorldApplication.java
│   │   ├── resources
│   │   │   ├── application.properties
```

```
|   |   |   └── templates
|   |   |       └── hello.html
|   |   └── webapp
|   |       └── WEB-INF
|   |           └── web.xml
└── pom.xml
```

2. Maven Configuration (pom.xml)

Ensure you have the necessary dependencies for Spring Web MVC.

3. Spring Boot Application (SpringHelloWorldApplication.java)

```
This class initializes the Spring Boot application.
package com.example.SpringHelloWorld;

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

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

4. Controller (HelloWorldController.java)

This controller handles the request and returns a view.

```
package com.example.SpringHelloWorld.controller;

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

@Controller
public class HelloWorldController {
```

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

5. View (hello.html)

```
This is the Thymeleaf template that displays the message.
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello World</title>
</head>
<body>
    <h1 th:text="${message}"></h1>
</body>
</html>
```

Summary

- **Model**: The data (message attribute) in the controller.
- **View**: The hello.html Thymeleaf template.
- **Controller**: The HelloWorldController that handles the /hello request.

This simple example demonstrates how to create a Spring Web MVC application to display "Hello, World!" using Spring Boot and Thymeleaf.

Explanation of <h1 th:text="${message}"></h1>

This line is part of a Thymeleaf template. Thymeleaf is a Java template engine for processing and creating HTML, XML, JavaScript, CSS, and text. In a Spring MVC application, it is used to generate dynamic content for web pages.

Breakdown:

1. **<h1></h1>**:
    - This is an HTML element. An <h1> tag is used to define the most important heading on a web page.
2. **th:text="${message}"**:

- This is a Thymeleaf attribute. The th:text attribute is used to set the text content of the HTML element.

3. **${message}**:
    - This is a Thymeleaf expression. It refers to a variable named message that has been added to the model in the Spring controller.

Define Spring Boot. Why should we use Spring Boot Framework? List out Spring boot features.

Definition of Spring Boot

Spring Boot is an open-source framework designed to simplify the development of new Spring applications. It provides a rapid way to set up, configure, and run both simple and web-based applications. By default, Spring Boot uses sensible defaults and reduces the need for extensive configuration, thus enabling developers to get applications up and running quickly.

Why Use Spring Boot?

1. **Simplifies Development**: Spring Boot eliminates the need for extensive configuration and boilerplate code, making it easier to set up and start a Spring application.
2. **Embedded Servers**: It comes with embedded servers like Tomcat, Jetty, and Undertow, which allow you to run web applications without deploying them to an external server.
3. **Production-Ready Features**: Provides production-ready features such as metrics, health checks, and externalized configuration.
4. **Opinionated Defaults**: Offers opinionated 'starter' dependencies to simplify the build configuration.
5. **Microservices**: Ideal for building microservices architecture due to its lightweight nature and easy configuration.

Features of Spring Boot

1. **Starter Dependencies**: Simplifies Maven or Gradle configuration with a set of commonly used dependencies in a single dependency.
    - Example: spring-boot-starter-web for building web applications.
2. **Auto-Configuration**: Automatically configures your Spring application based on the dependencies you have added.
    - Example: If spring-boot-starter-web is in the classpath, Spring Boot auto-configures a web server.
3. **Embedded Servers**: Comes with embedded Tomcat, Jetty, or Undertow servers.
    - Example: You can run your application with java -jar command without deploying to an external server.
4. **Spring Boot CLI**: Command-line interface for creating, running, and testing Spring Boot applications.
    - Example: Using spring init to bootstrap a new application.

5. **Actuator**: Provides production-ready features to monitor and manage your application.
    - Example: Exposes endpoints like /actuator/health and /actuator/metrics.
6. **Spring Initializr**: A web-based tool to bootstrap a Spring Boot project.
    - Example: Available at start.spring.io.
7. **DevTools**: Enhances the development experience by providing features like automatic restarts, live reload, and configurations for improved development workflow.
    - Example: Changes to classes or resources are automatically reloaded.
8. **Externalized Configuration**: Supports flexible configuration of the application using properties or YAML files, environment variables, and command-line arguments.
    - Example: Using application.properties or application.yml for configuration.
9. **Security**: Provides built-in support for securing your application with various authentication and authorization mechanisms.
    - Example: Using spring-boot-starter-security to add security features.
10. **Logging**: Pre-configured logging framework using Logback, and support for other logging frameworks like Log4j2 and SLF4J.
    - Example: Logs are automatically written to the console and can be configured via application.properties.

Summary

Spring Boot is designed to simplify the process of building and deploying Spring applications by providing a comprehensive set of tools and features that streamline the development process. Its key features like starter dependencies, auto-configuration, embedded servers, and the Spring Boot CLI make it a powerful framework for both new and experienced developers. The production-ready features of Spring Boot, including monitoring and management tools provided by Actuator, make it an ideal choice for developing robust, maintainable, and easily deployable applications.

List out the difference between Spring Boot and Spring MVC. [3]

Aspect	Spring Boot	Spring MVC
Definition	A framework for simplifying Spring application setup and development.	A module of the Spring framework focused on building web applications using the Model-View-Controller pattern.
Setup and Configuration	Minimal setup with auto-configuration and opinionated defaults.	Requires extensive configuration and setup for dependencies, view resolvers, etc.
Starter Dependencies	Provides starter dependencies to simplify build configuration.	No concept of starter dependencies; manual dependency management.

Aspect	Spring Boot	Spring MVC
Embedded Server	Comes with embedded servers like Tomcat, Jetty, and Undertow.	Requires deployment to an external server like Tomcat or Jetty.
Application Deployment	Can run standalone applications using java -jar.	Requires WAR packaging and deployment to an external server.
Auto-Configuration	Automatically configures Spring beans based on classpath settings, other beans, and property settings.	Manual configuration of beans and components.
Spring Initializr	Provides a web-based tool (start.spring.io) for bootstrapping projects.	No dedicated tool for bootstrapping projects.
Actuator	Provides built-in endpoints for monitoring and managing applications.	No built-in support; requires additional configuration and tools.
DevTools	Offers development-time features like automatic restarts and live reload.	No built-in support for development-time tools.
Focus	Rapid application development with production-ready features.	Building web applications using MVC pattern with manual setup.
Configuration Management	Supports externalized configuration using properties and YAML files.	Configuration primarily through XML or Java-based configuration.
Security	Easy integration with Spring Security through starter dependencies.	Requires manual integration and configuration of Spring Security.
Logging	Pre-configured logging using Logback, with support for other frameworks.	Manual setup for logging frameworks like Logback or Log4j.
Community and Ecosystem	Leverages the entire Spring ecosystem with additional support for microservices.	Part of the broader Spring framework, focused on web application development.

Summary

Spring Boot is designed to simplify and speed up the development process of Spring applications with minimal configuration and built-in production-ready features. It is particularly useful for standalone applications and microservices. On the other hand, Spring MVC is a part of the Spring framework focused on building traditional web applications using the MVC pattern, requiring more manual configuration and setup.

Design and implement simple spring boot program to print “WELCOME TO SPRING BOOT”. [3]

Here is a simple Spring Boot program designed to print "WELCOME TO SPRING BOOT" when accessed through a web browser. This example includes only the essential parts needed for an exam.

Step 1: Create a Spring Boot Project

You can use Spring Initializr (start.spring.io) to generate the project structure, or create it manually. For simplicity, I'll outline the steps assuming you have the basic Spring Boot setup ready.

Step 2: Create the Main Application Class

Create a main application class annotated with @SpringBootApplication. This class will serve as the entry point for the Spring Boot application.

```
package com.example.demo;


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


@SpringBootApplication
public class DemoApplication {


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

Step 3: Create a Controller Class

Create a controller class with a method to handle HTTP GET requests and return "WELCOME TO SPRING BOOT".

```
package com.example.demo;


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


@RestController
public class WelcomeController {


    @GetMapping("/welcome")
    public String welcome() {
        return "WELCOME TO SPRING BOOT";
    }
```

```
}
```

Step 4: Application Properties

Ensure you have the necessary dependencies in your pom.xml (for Maven) or build.gradle (for Gradle). This example assumes you are using Maven.

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

Step 5: Run the Application

Run the application by executing the main method in the DemoApplication class. You can do this from your IDE or by using the command line.

Explanation:

- **Package Declaration (package com.example.demo;)**: Indicates that the class belongs to the com.example.demo package.
- **Imports (import org.springframework.boot.SpringApplication; and import org.springframework.boot.autoconfigure.SpringBootApplication;)**: These import statements bring in the necessary classes from the Spring Boot framework.
- **@SpringBootApplication Annotation**: This annotation is a convenience annotation that combines three annotations:
    - @EnableAutoConfiguration: Enables Spring Boot's auto-configuration mechanism.
    - @ComponentScan: Enables component scanning so that the web controller classes and other components you create will be automatically discovered and registered as beans in the Spring application context.
    - @Configuration: Allows to register extra beans in the context or import additional configuration classes.
- **Main Method (public static void main(String[] args))**: The entry point of the Spring Boot application.
    - SpringApplication.run(DemoApplication.class, args);: This static method launches the Spring Boot application. It sets up the default configuration, starts the Spring application context, performs class path scanning, and starts the embedded web server.

Explanation:

- **Package Declaration (package com.example.demo;)**: Indicates that the class belongs to the com.example.demo package.
- **Imports (import org.springframework.web.bind.annotation.GetMapping; and import org.springframework.web.bind.annotation.RestController;)**: These import statements bring in the necessary classes from the Spring framework for handling web requests.
- **@RestController Annotation**: This annotation is a combination of @Controller and @ResponseBody. It indicates that the class serves RESTful web services and that the methods return domain objects instead of views.
- **Method (public String welcome())**: This method handles HTTP GET requests.
    - **@GetMapping("/welcome")**: This annotation maps HTTP GET requests to the /welcome URL to the welcome method.
    - **return "WELCOME TO SPRING BOOT";**: This line returns a plain text response. When a client accesses the /welcome URL, this message is sent as the response.

Explain the following with a program: i.Spring Container ii. Dependency injection

Explanation

i. Spring Container

Spring Container is the core of the Spring Framework. It is responsible for managing the lifecycle and configuration of application objects (also known as beans). The container uses dependency injection (DI) to manage the components that make up an application. The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.

ii. Dependency Injection (DI)

Dependency Injection is a design pattern used to implement IoC (Inversion of Control), allowing the creation of dependent objects outside of a class and providing those objects to a class in different ways. This pattern helps in creating loosely coupled applications, making the code more modular, easier to test, and maintain.

Example Program

Let's create a simple Spring application demonstrating the Spring Container and Dependency Injection.

Step 1: Create a Maven Project

Ensure you have the necessary dependencies in your pom.xml:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.10</version>
    </dependency>
```

```
</dependencies>
```

Step 2: Define the Service Interface and Implementation

Service Interface: GreetingService

```
package com.example.demo;


public interface GreetingService {
    String greet();
}
```

Service Implementation: GreetingServiceImpl

```
package com.example.demo;


public class GreetingServiceImpl implements GreetingService {


    @Override
    public String greet() {
        return "Hello, Welcome to Spring!";
    }
}
```

Step 3: Define the Controller Class

Controller Class: GreetingController

```
package com.example.demo;


public class GreetingController {
    private final GreetingService greetingService;


    // Constructor-based Dependency Injection
    public GreetingController(GreetingService greetingService) {
        this.greetingService = greetingService;
    }


    public void sayGreeting() {
```

```
        System.out.println(greetingService.greet());
    }
}
```

Step 4: Create the Spring Configuration

Configuration Class: AppConfig

```
package com.example.demo;


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


@Configuration
public class AppConfig {


    @Bean
    public GreetingService greetingService() {
        return new GreetingServiceImpl();
    }


    @Bean
    public GreetingController greetingController() {
        return new GreetingController(greetingService());
    }
}
```

Step 5: Initialize the Spring Container and Use Beans

Main Class: MainApp

```
package com.example.demo;


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


public class MainApp {
```

```
    public static void main(String[] args) {
        // Initialize Spring Container
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve the GreetingController bean
        GreetingController controller =
context.getBean(GreetingController.class);

        // Use the bean
        controller.sayGreeting();
    }
}
```

Explanation of the Code

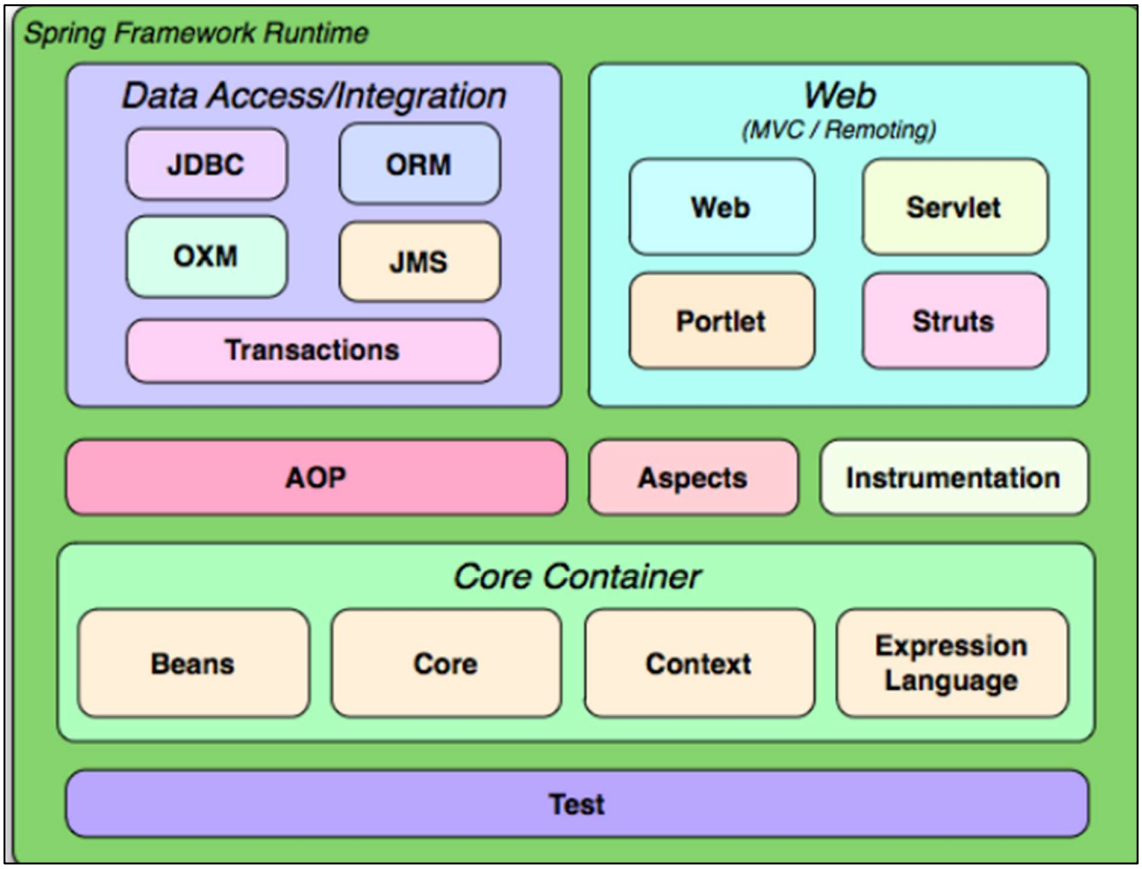
1. **Service Interface (GreetingService)**: Defines the contract for the greeting service.
2. **Service Implementation (GreetingServiceImpl)**: Implements the GreetingService interface, providing the actual greeting message.
3. **Controller Class (GreetingController)**: Depends on the GreetingService and uses it to perform its operation. The dependency is injected via the constructor.
4. **Configuration Class (AppConfig)**: Annotated with @Configuration, it defines beans using the @Bean annotation. It creates instances of GreetingServiceImpl and GreetingController.
5. **Main Class (MainApp)**:
    - Initializes the Spring Container using AnnotationConfigApplicationContext with AppConfig class.
    - Retrieves the GreetingController bean from the container.
    - Calls the sayGreeting method to print the greeting message.

Key Concepts Illustrated

- **Spring Container**: Manages the beans defined in the AppConfig class.
- **Dependency Injection**: The GreetingController class gets the GreetingService injected via its constructor.

This example demonstrates how to set up and use the Spring Container and apply Dependency Injection in a Spring application.

Explain Spring Framework Architecture in detail. [2]



The Spring Framework architecture is designed to simplify the development of enterprise applications. It provides comprehensive infrastructure support for developing Java applications. The core features of the Spring Framework can be used by any Java application, although there are extensions for building web applications on top of the Java EE platform.

Here's a detailed breakdown of the Spring Framework architecture:

Core Components of Spring Framework

1. **Core Container**
2. **Data Access/Integration**
3. **Web**
4. **AOP (Aspect Oriented Programming)**
5. **Instrumentation**
6. **Messaging**
7. **Test**

1. Core Container

The core container consists of the following modules:

- **Core Module**: Provides the fundamental parts of the framework, including the IoC (Inversion of Control) and Dependency Injection features.
- **Beans Module**: Provides BeanFactory, which is a sophisticated implementation of the factory pattern.
- **Context Module**: Builds on the solid base provided by the core and beans modules, and adds support for more enterprise-specific features such as event-propagation, declarative mechanisms to create a bean, and various ways to lookup.
- **Expression Language Module**: A powerful expression language for querying and manipulating an object graph at runtime.

2. Data Access/Integration

- **JDBC Module**: Provides a JDBC-abstraction layer that removes the need for tedious JDBC-related coding.
- **ORM Module**: Provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- **OXM Module**: Provides an abstraction layer for Object/XML mapping implementations, including JAXB, Castor, XMLBeans, JiBX, and XStream.
- **JMS Module**: Contains features for producing and consuming messages.
- **Transaction Module**: Supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (Plain Old Java Objects).

3. Web

- **Web Module**: Provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using a servlet listener.
- **Web-Servlet Module**: Contains Spring's Model-View-Controller (MVC) implementation for web applications.
- **Web-Struts Module**: Provides the support for the integration with the classic Struts 1.x web framework.
- **Web-Portlet Module**: Provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

4. AOP (Aspect Oriented Programming)

- **AOP Module**: Provides aspect-oriented programming implementation, allowing you to define method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- **Aspects Module**: Provides integration with AspectJ (a powerful and mature aspect-oriented programming extension).

5. Instrumentation

- **Instrumentation Module**: Provides support for class instrumentation and classloader implementations to be used in certain application servers.

6. Messaging

- **Messaging Module**: Provides support for STOMP over WebSocket and an abstraction for messaging protocols and integrations.

7. Test

- **Test Module**: Supports the testing of Spring components with JUnit or TestNG frameworks. It provides consistent loading of Spring ApplicationContexts and caching of those contexts.

Architecture Diagram

Below is a high-level view of the Spring Framework architecture:

IMAGE

Detailed Explanation

Core Container

- **Core Module**: This is the fundamental part of the Spring Framework. It provides the IoC (Inversion of Control) and Dependency Injection (DI) features, which are the backbone of the framework.
- **Beans Module**: This module contains the BeanFactory, which is a sophisticated implementation of the factory pattern. It separates the configuration and dependencies from the application logic.
- **Context Module**: Builds on the Core and Beans modules, adding support for internationalization (I18N), event propagation, and various ways to access resources.
- **Expression Language Module**: Offers a powerful language for querying and manipulating an object graph at runtime.

Data Access/Integration

- **JDBC Module**: Simplifies the use of JDBC by providing a layer of abstraction. It helps in reducing boilerplate code and managing resources.
- **ORM Module**: Provides integration with popular ORM frameworks like Hibernate, JPA, and iBatis, allowing for easy database interaction.
- **OXM Module**: Object/XML Mapping provides an abstraction layer to map objects to XML and vice versa using various libraries.
- **JMS Module**: Simplifies the use of the Java Messaging Service API.
- **Transaction Module**: Supports programmatic and declarative transaction management, allowing consistent transaction support across different transaction APIs.

Web

- **Web Module**: Provides the foundation for all web-based applications. It includes multipart file upload capabilities, and a way to initialize the IoC container using a servlet listener.
- **Web-Servlet Module**: This module contains Spring's Model-View-Controller (MVC) implementation, allowing the development of web applications.

- **Web-Struts Module**: Provides support for integrating with Struts, a popular web framework.
- **Web-Portlet Module**: Provides an MVC implementation to be used in a portlet environment.

AOP

- **AOP Module**: Provides support for aspect-oriented programming, allowing the definition of method interceptors and pointcuts.
- **Aspects Module**: Provides integration with AspectJ, a powerful and mature AOP framework.

Instrumentation

- **Instrumentation Module**: Provides support for class instrumentation and classloader implementations to be used in certain application servers.

Messaging

- **Messaging Module**: Provides support for messaging protocols and integrations, including STOMP over WebSocket.

Test

- **Test Module**: Supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts.

Conclusion

The Spring Framework architecture is modular and comprehensive, providing a solid foundation for building enterprise-level applications. By understanding the core components and their interactions, developers can leverage the power of Spring to create robust, maintainable, and scalable applications.

Explain some of the most used Spring Boot annotations with an example. [2]

Spring Boot simplifies the development of Java applications with a range of annotations that help to configure and manage the components and behaviors of your application. Here are some of the most commonly used Spring Boot annotations, along with examples to illustrate their use:

1. @SpringBootApplication

This is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It is typically used on the main class of a Spring Boot application.

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


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

```
}
```

2. @RestController

This annotation is used to create RESTful web services using Spring MVC. It combines @Controller and @ResponseBody.

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

@RestController
@RequestMapping("/api")
public class MyRestController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

3. @RequestMapping and @GetMapping/@PostMapping

@RequestMapping is used to map web requests to specific handler classes or methods, and @GetMapping/@PostMapping are shortcuts for @RequestMapping(method = RequestMethod.GET) and @RequestMapping(method = RequestMethod.POST) respectively.

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/greeting")
```

```
    public String getGreeting() {
        return "Hello, GET!";
    }

    @PostMapping("/greeting")
    public String postGreeting(@RequestBody String name) {
        return "Hello, " + name + "!";
    }
}
```

4. @Service

This annotation is used to mark a class as a service provider. Spring will automatically detect such classes through classpath scanning.

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

@Service
public class MyService {

    public String getServiceMessage() {
        return "Service Message";
    }
}
```

5. @Autowired

This annotation is used for automatic dependency injection. It can be used on constructors, setter methods, or fields.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

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

```
    private final MyService myService;

    @Autowired
    public MyAutowiredController(MyService myService) {
        this.myService = myService;
    }

    @GetMapping("/service")
    public String getServiceMessage() {
        return myService.getServiceMessage();
    }
}
```

6. @Repository

This annotation is used to indicate that the class provides the mechanism for storage, retrieval, search, update, and delete operation on objects.

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

@Repository
public interface MyRepository extends JpaRepository<MyEntity, Long> {
}
```

7. @Entity

This annotation is used to specify that the class is an entity and is mapped to a database table.

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class MyEntity {

    @Id
```

```
    private Long id;
    private String name;

    // getters and setters
}
```

8. @Value

This annotation is used to inject values into fields from property files or environment variables.

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    @Value("${my.property}")
    private String myProperty;

    public String getMyProperty() {
        return myProperty;
    }
}
```

9. @Configuration

This annotation indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests.

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

@Configuration
public class MyConfiguration {

    @Bean
    public MyService myService() {
        return new MyService();
```

```
    }
}
```

10. @ConditionalOnProperty

This annotation is used to conditionally enable or disable a bean based on the presence of a property.

```
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


@Configuration
public class MyConditionalConfiguration {


    @Bean
    @ConditionalOnProperty(name = "feature.enabled", havingValue = "true")
    public MyFeature myFeature() {
        return new MyFeature();
    }
}
```

Summary

These annotations are just a few examples of the many available in Spring Boot. Each serves a specific purpose to simplify and enhance the development of Java applications. Understanding and effectively using these annotations can greatly improve the productivity and maintainability of your Spring Boot applications.

Explain Bean Lifecycle in Spring Framework. [2]

The bean lifecycle in the Spring Framework consists of several phases from the time a bean is instantiated until it is destroyed. Here's a brief overview of the key stages in the lifecycle of a Spring bean:

1. Instantiation

The bean is created by the Spring container using Java reflection.

2. Populate Properties

The Spring container populates all the properties (dependencies) defined in the bean definition. This is done via dependency injection.

3. BeanNameAware and BeanFactoryAware Callbacks

If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean's ID. If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory() passing an instance of itself.

4. Pre-initialization (PostProcessBeforeInitialization)

If there are any BeanPostProcessors associated with the bean, their postProcessBeforeInitialization methods will be called.

5. Initialization

- **InitializingBean Interface**: If the bean implements the InitializingBean interface, its afterPropertiesSet method is called.
- **Custom Init Method**: If a custom init-method is specified in the bean definition, that method is called.

6. Post-initialization (PostProcessAfterInitialization)

If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization methods will be called.

7. Ready for Use

The bean is now fully initialized and ready for use by the application.

8. Destruction

- **DisposableBean Interface**: If the bean implements the DisposableBean interface, its destroy method is called.
- **Custom Destroy Method**: If a custom destroy-method is specified in the bean definition, that method is called.

Bean Lifecycle Example

```
import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;


class SimpleBean implements InitializingBean, DisposableBean {


    private String message;


    public void setMessage(String message) {
```

```
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Bean is going through afterPropertiesSet.");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("Bean will be destroyed now.");
    }

    public void customInit() {
        System.out.println("Custom init method.");
    }

    public void customDestroy() {
        System.out.println("Custom destroy method.");
    }
}

@Configuration
class SimpleConfig {

    @Bean(initMethod = "customInit", destroyMethod = "customDestroy")
    public SimpleBean simpleBean() {
        SimpleBean bean = new SimpleBean();
```

```
        bean.setMessage("Hello, Spring!");
        return bean;
    }
}


public class SimpleBeanLifecycleDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(SimpleConfig.class);
        SimpleBean simpleBean = context.getBean(SimpleBean.class);
        System.out.println(simpleBean.getMessage());
        context.close();
    }
}
```

Explanation

1. **SimpleBean Class**:
    - **Implements InitializingBean**: The afterPropertiesSet method will be called after the properties are set.
    - **Implements DisposableBean**: The destroy method will be called before the bean is destroyed.
    - **Custom Init and Destroy Methods**: These methods provide additional initialization and destruction logic.
2. **SimpleConfig Class**:
    - **Defines a Bean**: The @Bean annotation defines a simpleBean bean with custom init and destroy methods specified.
3. **SimpleBeanLifecycleDemo Class**:
    - **Main Method**: The AnnotationConfigApplicationContext is used to load the Spring context from the SimpleConfig class.
    - **Accessing the Bean**: The simpleBean bean is retrieved and its message is printed.
    - **Closing the Context**: The context is closed, triggering the destroy lifecycle methods.

Output

When you run this example, you will see the following output:

```
Bean is going through afterPropertiesSet.
```

```
Custom init method.
Hello, Spring!
Bean will be destroyed now.
Custom destroy method.
```