**Spring Boot**

In Spring Boot, annotations are extensively used to configure various aspects of the application. Some of the most used annotations and their meanings in Spring Boot are as follows:

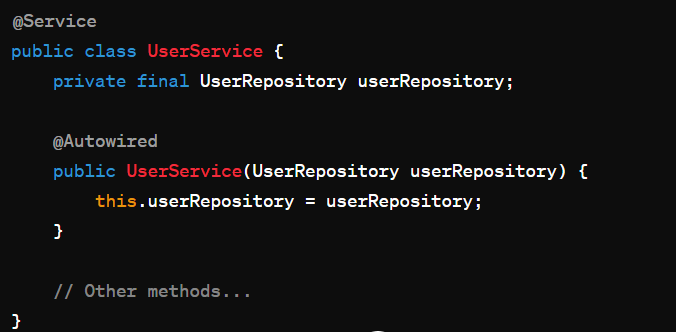
1. **@SpringBootApplication**:
   * This annotation is used to mark the main class of a Spring Boot application.
   * It combines **@Configuration**, **@EnableAutoConfiguration**, and **@ComponentScan** annotations.
   * **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.
   * **@EnableAutoConfiguration** enables Spring Boot's auto-configuration mechanism.
   * **@ComponentScan** tells Spring to scan for other components, configurations, and services in the package where the application class is located and its sub-packages.
2. **@RestController**:
   * This annotation is used to define RESTful web services.
   * It combines **@Controller** and **@ResponseBody** annotations.
   * **@Controller** indicates that the class serves the role of a controller in Spring MVC, handling HTTP requests.
   * **@ResponseBody** tells Spring MVC that the return value of the method should be serialized directly to the HTTP response body.
3. **@RequestMapping**:
   * This annotation is used to map web requests onto specific handler classes and/or handler methods.
   * It can be applied at the class level and/or method level.
   * It specifies the URL pattern that the controller or handler method will respond to.
   * It also specifies the HTTP request method (GET, POST, PUT, DELETE, etc.) that the handler method will handle.
4. **@Autowired**:
   * This annotation is used to automatically wire beans by type.
   * It injects the bean by matching the data type (class) of a bean property with the bean definition in the Spring container.
   * It can be applied to fields, constructors, or methods.
5. **@Value**:
   * This annotation is used to inject values from properties files into Spring beans.
   * It can be applied to fields in a bean class, allowing you to inject properties directly into fields.
6. **@Component**, **@Service**, **@Repository**:
   * These annotations are used to denote different types of Spring-managed components.
   * **@Component** is a generic stereotype annotation for any Spring-managed component.
   * **@Service** is used to annotate classes that perform service tasks.
   * **@Repository** is used to annotate classes that access the database or other forms of data storage.

These are just a few of the most commonly used annotations in Spring Boot. There are many more annotations available for various purposes, such as transaction management, security, caching, etc. Understanding and using these annotations effectively can greatly simplify the development of Spring Boot applications.

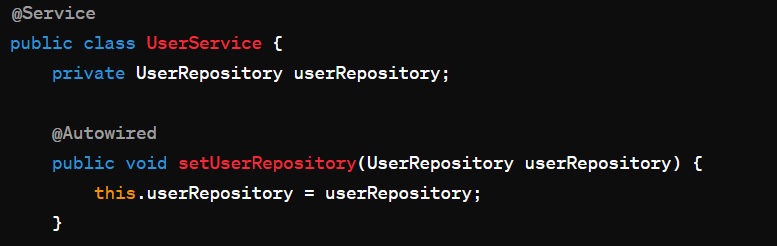
1. **Dependency Injection (DI)**:
   * **Definition**: Dependency Injection is a design pattern used to achieve Inversion of Control (IoC) in software applications. In DI, instead of creating dependencies within a class, the dependencies are provided ("injected") from outside the class.
   * **Meaning**: With DI, a class does not directly create instances of its dependencies but instead receives them from an external source. This promotes loose coupling between components, making the code more modular, testable, and maintainable.
2. **Beans**:
   * **Definition**: In the context of Spring Framework, a bean is a managed Java object created by the Spring IoC container. Beans are the core building blocks of a Spring application, representing the various components that make up the application.
   * **Meaning**: Beans are typically defined in Spring configuration files or using annotations like **@Component**, **@Service**, **@Repository**, etc. They are instantiated, configured, and managed by the Spring IoC container, allowing for easy management of dependencies and providing features like dependency injection and lifecycle management.
   * JavaBeans are standard Java classes that follow specific conventions, including having a public default (no-argument) constructor, properties accessed using getter and setter methods, and implementing the Serializable interface. JavaBeans are typically used as data containers and are not tied to any specific framework. They are designed to be easily reusable and serializable, making them suitable for various Java applications, including GUI development and data persistence.
   * Spring Beans, on the other hand, are objects managed by the Spring Framework. They are defined in Spring configuration files or through annotations like @Component, @Service, @Repository, or @Controller. Spring Beans are instantiated, configured, and managed by the Spring container, which handles their lifecycle. This includes instantiation, dependency injection, and destruction. Spring Beans can be configured to be singletons (one instance per Spring context) or prototypes (a new instance every time it is requested), among other configurations. The Spring container also manages the lifecycle of Spring Beans, including their initialization and destruction, and it is responsible for dependency injection, allowing beans to communicate with each other.
   * The lifecycle of a Spring Bean involves several stages:
     1. Instantiation: The Spring container instantiates the bean using the constructor defined in the bean class.
     2. Population of Properties: The container sets the properties of the bean using setter methods or field injection, based on the configuration provided in the Spring configuration file or annotations.
     3. Initialization: If the bean implements the InitializingBean interface or has an init-method defined in the configuration, the container calls this method to perform any initialization.
     4. Use: The bean is now ready to be used by the application, with its dependencies injected.
     5. Destruction: When the application context is closed, or when the bean is explicitly removed, the container calls the bean's destroy method if it implements the DisposableBean interface or has a destroy-method defined in the configuration, allowing the bean to release resources before being removed from the application context.
3. **Configurations**:
   * **Definition**: In Spring Framework, configuration refers to the process of specifying how the various components of a Spring application are wired together. Configuration can be done using XML configuration files, Java-based configuration classes, or annotations.
   * **Meaning**: Spring configuration defines the beans, their relationships, and other settings required by the application. It can include defining beans, specifying dependencies, enabling features like component scanning, aspect-oriented programming (AOP), and more.
4. **Interfaces**:
   * **Definition**: In Java, an interface is a reference type that defines a set of abstract methods that a class implementing the interface must provide. Interfaces can also contain constant fields.
   * **Meaning**: Interfaces provide a contract for classes to adhere to, defining a common set of behaviors without specifying their implementation details. In the context of Spring, interfaces are often used to define service contracts, allowing for loose coupling between components and facilitating dependency injection and polymorphism.

**Examples of Dependency Injection in Spring Boot:**

**Constructor Injection**:



**Setter Injection**:



**Field Injection**:

A screenshot of a computer program

Description automatically generated

**Bean Lifecycle in Spring Boot:**

The lifecycle of a bean in Spring Boot can be summarized into several stages:

1. **Instantiation**:
   * During this stage, the Spring IoC container creates an instance of the bean. This can happen through constructor injection, setter injection, or field injection.
2. **Populate Properties**:
   * If the bean has any dependencies, Spring injects them into the bean using dependency injection.
3. **Initialization**:
   * After the properties are set, the bean may need to perform some initialization tasks. This is typically done by calling initialization methods annotated with **@PostConstruct** or implementing the **InitializingBean** interface.
4. **Using the Bean**:
   * Once initialized, the bean is ready for use by other beans or components in the application.
5. **Destruction**:
   * When the application context is shut down, Spring calls the destruction methods of the beans. This can be done by annotating methods with **@PreDestroy** or implementing the **DisposableBean** interface.

@Service

public class MyService implements InitializingBean, DisposableBean {

public MyService() {

System.out.println("Constructor called");

}

@PostConstruct

public void init() {

System.out.println("Init method called");

}

@Override

public void afterPropertiesSet() throws Exception {

System.out.println("InitializingBean's afterPropertiesSet method called");

}

// Other methods...

@PreDestroy

public void destroy() {

System.out.println("Destroy method called");

}

@Override

public void destroy() throws Exception {

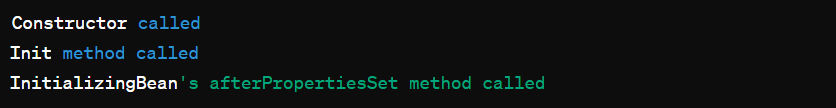
System.out.println("DisposableBean's destroy method called");

}

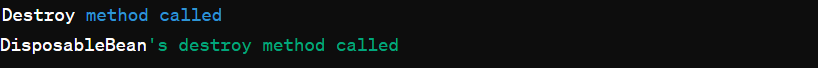
}

Here's a step-by-step guide to seeing the bean lifecycle for a service class:

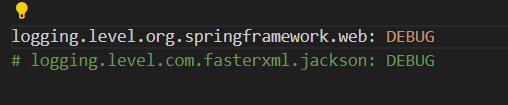
1. **Create the Service Class**: Create a service class, let's say **MyService**, with lifecycle methods annotated (**@PostConstruct**, **@PreDestroy**) or implementing lifecycle interfaces (**InitializingBean**, **DisposableBean**).
2. **Run the Spring Boot Application**: Start your Spring Boot application. You can either run it from your IDE or use Maven to build and run the application.
3. **Monitor Console Output**: As the application starts up, Spring will log the lifecycle events of the beans. Look for messages in the console that indicate the invocation of the lifecycle methods. For example:



1. **Use the Application**: After the application has started, you can use its functionality as usual. The service beans will be initialized and ready to use.
2. **Shutdown the Application**: When you stop the Spring Boot application, Spring will log the destruction of the beans. Look for messages indicating the invocation of the destruction methods. For example:



Additionally, you can enable debug logging for Spring Framework in your application's logging configuration (**application.properties** or **application.yml**). This will provide more detailed information about the bean lifecycle events, including which beans are being created and initialized. For example:



A black screen with green text

Description automatically generated

<https://meet.google.com/djq-eiay-qfn>