@PostConstruct

https://www.baeldung.com/spring-postconstruct-predestroy#:~:text=Spring%20calls%20the%20methods%20annotated,it%20can't%20be%20static

The `@PostConstruct` annotation is a callback annotation in Java that allows you to specify a method to be invoked immediately after a bean has been constructed and before any other initialization methods are called. In the context of Spring Framework, `@PostConstruct` is commonly used to perform initialization tasks on a bean after its dependencies have been injected and before it's put into service.

Here are some common use cases for `@PostConstruct`:

1. **Initialization of resources**: You can use `@PostConstruct` to initialize resources such as database connections, file handles, or network connections that your bean depends on.

2. **Setup tasks**: Perform setup tasks or configuration after the bean has been constructed. This could include initializing properties, configuring third-party libraries, or performing other initialization logic.

3. **Validation**: Validate the state of the bean after it has been constructed and before it's used. This can help ensure that the bean is in a valid state before it starts processing requests.

4. **Logging**: Perform logging or auditing tasks to track the initialization process of the bean.

Here's an example of how to use `@PostConstruct` in a Spring bean:

```java
import javax.annotation.PostConstruct;

public class MyBean {

    private String message;

    @PostConstruct
    public void init() {
        // Initialization logic goes here
        message = "Hello, world!";
        System.out.println("Bean initialized: " + message);
    }
```

```
    // Other methods and properties
}
```

In this example, the `init()` method annotated with `@PostConstruct` will be automatically invoked by the Spring container after the bean has been instantiated, but before any other initialization methods or lifecycle callbacks are called.

———————

In Spring Framework, if you prefer not to use the JSR-250 annotations like `@PostConstruct` and `@PreDestroy` but still want to achieve initialization and destruction of beans without coupling them to Spring-specific interfaces or annotations, you can use the `init-method` and `destroy-method` attributes in your bean definitions.

Here's how you can configure initialization and destruction methods using XML bean definitions:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd">

    <!-- Define bean MyBean with custom init and destroy methods -->
    <bean id="myBean" class="com.example.MyBean"
          init-method="customInit" destroy-method="customDestroy">
    </bean>

</beans>
```

In this configuration:

- The `init-method` attribute specifies the name of the method to be invoked for bean initialization. This method should be public and take no arguments.
- The `destroy-method` attribute specifies the name of the method to be

invoked for bean destruction. This method should also be public and take no arguments.

Here's an example of a bean class `MyBean` with custom initialization and destruction methods:

```java
public class MyBean {

    public void customInit() {
        // Custom initialization logic
        System.out.println("Bean initialization logic executed.");
    }

    public void customDestroy() {
        // Custom destruction logic
        System.out.println("Bean destruction logic executed.");
    }

    // Other methods and properties
}
```

With this configuration, when the Spring container initializes the `myBean` bean, it will automatically call the `customInit()` method after instantiation, and when the bean is destroyed (e.g., when the application context is shut down), it will call the `customDestroy()` method.

This approach allows you to specify custom initialization and destruction methods for your beans without introducing coupling to Spring-specific annotations or interfaces, providing a more flexible and decoupled way to manage bean lifecycle.