Theory: Beans and components

20 minutes 0 / 5 problems solved

Skip this topic Start practicing

1141 users solved this topic. Latest completion was about 9 hours ago.

§1. When do we need beans in Spring Boot

Usually, we need to create different objects in a web app to use their functionalities. Some of them need other objects to be created inside, and so on. Spring has a great option to simplify a huge and complicated chain of making objects. It can create all the necessary objects during the application startup and put them all in a container. Then each class can get any objects it needs from this container without plenty of creation and initialization constructions.

These objects are called **beans**, and a container is the **IoC container**. IoC is for Inversion of Control. When we need to create one object inside another, the IoC container defines it in a constructor or in fields. Then the container creates this object, then gives it to the host object, or **injects** it.

It helps reduce the responsibility of the host class. Using the IoC container, you don't need to create, initialize and close the injected objects inside the host one. It is the IoC that is doing it for you. That is why it is called the **inversion of control**: an object gives the control of its fields' lifecycles to a container or inverses it.

Let's explore how to tell Spring Boot what objects you want to be put in the container.

§2. @Component annotation

Imagine there is a printer with some ink supply in it. First, we will create a bean of `InkSupply`. Let's place a `@Component` annotation above a simple class representing ink supply in a printer:

```
@Component
public class InkSupply {

    private final String INK_MESSAGE = "This is ink supply";

    public InkSupply() {
    }

    public String getMessage(){

        return INK_MESSAGE;

    }

}
```

Any object of this class needs no special initialization and can be created via default constructor. When Spring Boot starts an application, it looks for **all** the `@Component` annotated classes and creates at least one object of each found class. Then it will be waiting in a container. How would we put this object to another object?

§3. Put a bean into other beans. @Autowired annotation

Now we want to place an object of `InkSupply` to a `Printer` object:

Current topic:

Beans and components ···

Topic depends on:

✕ Basic project structure ···

Table of contents:

```java
@Component
public class Printer {

    private InkSupply inkSupply;

    @Autowired
    public Printer(InkSupply inkSupply) {
        this.inkSupply = inkSupply;
    }


    public void printHello(){

        System.out.println(inkSupply.getMessage());

    }

}
```

The first thing here is `@Component` again: Spring Boot needs it to discover a class on a startup. The second thing is a new `@Autowired`annotation. We use this to tell Spring Boot that we need an `InkSupply` object from a container here.

Note that a bean created with `@Component` is a singleton. It means if you create a `CopyMachine` class and add `InkSupply` as a bean there, it would be exactly the same object, not a copied one.

That's it! We put an object to a container with a `@Component` and then take it to another object with `@Autowired` above its constructor.

§4. Put a bean into other beans without constructor

There is one more way to inject a bean. Look at the code snippet below:

```java
@Component
public class Printer {

    @Autowired
    private InkSupply inkSupply;

    public void printHello(){
        System.out.println(inkSupply.getMessage());
    }

}
```

So, if you don't want to add `InkSupply` to a constructor of `Printer`, you can just place `@Autowired` above the field instead. However, it is recommended to use constructor injection over field injection. The constructor injection makes the dependencies clearly identified, helps with thread-safety, and simplifies testing.

§5. @Bean and @Component

`@Component` is fine until we meet some problems. What if we want to create a bean of a class from some library? We cannot just add the annotation to this class because we usually cannot edit this class. Spring has an option for it: `@Bean` annotation. Look how we can create a bean of `InkSupply` without any annotation on this class:

```java
public class InkSupply {

    private final String INK_MESSAGE = "This is ink supply";

    public InkSupply() {
    }

    public String getMessage(){
        return INK_MESSAGE;

    }

}
```

Now we need to create a class which would have a method for manually creating a bean of `InkSupply` and annotate it with `@Configuration`. The method should have a `@Bean` annotation:

```java
@Configuration
public class PrinterConfiguration {

    @Bean
    public InkSupply getInkSupply(){
        InkSupply inkSupply = new InkSupply();
        //setting something in the inkSupply
        return inkSupply;
    }



}
```

Finally, we can autowire this bean as we did it before:

```java
@Component
public class Printer {

    private InkSupply inkSupply;

    @Autowired
    public Printer(InkSupply inkSupply) {
        this.inkSupply = inkSupply;
    }

}
```

Here we have created a bean of a non-modifiable class.

§6. Conclusion

So, beans are objects of any class that Spring Boot creates on a startup and provides to any other bean's constructor.

`@Component` above the class is the sign that there must be a bean of the class. If we want to create a bean outside the class manually, we should create a method returning an object of a class and annotate the method with `@Bean` and the containing class with `@Configuration`.

To show that a bean needs other beans to be created for it, we need to:

1. use `@Component` to show that it creates a host bean,
2. use `@Autowired` to show that this bean needs other beans.

Report a typo

119 users liked this theory. 15 didn't like it. **What about you?**

Start practicing

Comments (11) Hints (0) Useful links (2) Show discussion