Computer science → Backend → Spring Boot → Core container

Spring components

Theory Practice 100% completed, 1 problem solved

Theory

14 minutes reading

Unskip this topic Start practicing

As you already know, Spring IoC provides a great way to manage beans that are declared with the help of the `@Bean` annotation. They can be initialized at startup and automatically wired to other beans when it's needed. However, this is not the only way to declare container-managed objects in a Spring application, and you will often use a different approach based on the **component** concept.

In this topic, you will learn what a component is, when to use it and how components differ from beans that we discussed previously.

§1. Components

In Spring, a **component** is a special kind of class that can be autodetected by Spring IoC and used for dependency injection. Components are mostly used to:

- ensure a high level of decoupling between different parts of an application;
- assign responsibilities to classes in a more efficient way.

To define a component, there is a special class-level annotation `@Component` from the `org.springframework.stereotype` package. Spring IoC automatically identifies all classes annotated with it and creates corresponding managed beans. By default, there is only one bean for every component.

Usually, a component has one or more non-static methods that can be invoked from outside the component. However, in some situations, there are components without `public` methods.

Imagine there is a component called `PasswordGenerator` that can produce random passwords of a specified length.

▼ Java

```java
import org.springframework.stereotype.Component;
import java.util.Random;

@Component
public class PasswordGenerator {
    private static final String CHARACTERS =
"abcdefghijklmnopqrstuvwxyz";
    private static final Random random = new Random();

    public String generate(int length) {
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < length; i++) {
            int index = random.nextInt(CHARACTERS.length());
            result.append(CHARACTERS.charAt(index));
        }
        return result.toString();
```

3 required topics

✓ Final variables

✓ Interfaces

✓ Spring beans

3 dependent topics

✓ @Bean vs @Component

Scheduling

Spring Security Crypto

```
    }
}
```

▼ Kotlin

```
import org.springframework.stereotype.Component
import java.util.Random

@Component
class PasswordGenerator {
    companion object {
        private const val CHARACTERS = "abcdefghijklmnopqrstuvwxyz"
        private val random = Random()
    }

    fun generate(length: Int): String {
        val result = StringBuilder()
        for (i in 0 until length) {
            val index: Int = random.nextInt(CHARACTERS.length)
            result.append(CHARACTERS[index])
        }
        return result.toString()
    }
}
```

Any object of this class needs no special initialization and can be created via the default constructor. When Spring Boot starts an application, it looks for all the `@Component`-annotated classes and creates objects of these classes, which will then be waiting in the container.

> There's no need to worry if you haven't got a chance to use the `Random` or `StringBuilder` classes yet. All you need to know is that the first class can help us get random numbers and the second one creates a string by appending new elements.

§2. Interacting with command line

Before moving on, let's look at one facility of the Spring Framework and learn a bit more about components. To simplify the following explanation, we are going to declare a special component that will allow us to interact with the standard I/O.

To achieve it, the component should implement the `CommandLineRunner` interface and override the `run` method. It is just an equivalent of the `main` method in console applications. You can write any piece of code there and it will be executed once the Spring application starts.

▼ Java

```
@Component
public class Runner implements CommandLineRunner {

    @Override
    public void run(String... args) {
        System.out.println("Hello, Spring!");
    }
}
```

▼ Kotlin

```kotlin
@Component
class Runner : CommandLineRunner {
    override fun run(vararg args: String) {
        println("Hello, Spring!")
    }
}
```

The `run` method will be automatically invoked by the Spring framework. If you start an application that contains this component, you will see the result in the log:

```
Hello, Spring!
```

> You don't need to use `CommandLineRunner` in every Spring application but this component can be used as a temporary solution when debugging or studying new features of the framework.

§3. Autowiring components

All beans created automatically for components can be injected into each other using the `@Autowired` annotation. The dependency injection mechanism works exactly the same way as you've seen with `@Bean`-annotated methods.

Since they are both Spring components, our previously declared classes `PasswordGenerator` and `Runner` can use the dependency injection mechanism. As an example, we will provide a modified version of the `Runner` component that contains the autowired bean of the `PasswordGenerator` component.

▼ Java

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    @Autowired
    public Runner(PasswordGenerator generator) {
        this.generator = generator;
    }

    @Override
    public void run(String... args) {
        System.out.println("A short password: " +
generator.generate(5));
        System.out.println("A long password: " +
generator.generate(10));
    }
}
```

▼ Kotlin

```kotlin
@Component
class Runner @Autowired constructor(private val generator:
PasswordGenerator) : CommandLineRunner {
    override fun run(vararg args: String) {
        println("A short password: " + generator.generate(5))
        println("A long password: " + generator.generate(10))
    }
}
```

Here we use the `@Autowired` annotation to tell Spring Boot that we need a `PasswordGenerator` object from the container. If you start this application, you will see the output:

```
A short password: bqtik
A long password: tjgdpswzbd
```

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

> It is important to know that a bean created with `@Component` is a singleton by default. It means that if you declare another component and inject `PasswordGenerator` there, it will be exactly the same object, not a copied one. This default behavior can be modified and you will learn how to do it in the next lessons.

> There is an important restriction in Spring: you cannot declare circular dependencies between any beans (including components). If you do it, your application will not start and you will get an error: **The dependencies of some of the beans in the application context form a cycle**.

§4. Where to put the @Autowired annotation

There are several possible places where you can put the `@Autowired` annotation that are worth knowing.

1. As you've seen before, it is possible to put it on top of a constructor:

▼ Java

```
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    @Autowired
    public Runner(PasswordGenerator generator) {
        this.generator = generator;
    }


    // run

}
```

▼ Kotlin

```
@Component
class Runner @Autowired constructor(private val generator: PasswordGenerator) : CommandLineRunner {
    // run
}
```

2. Place the `@Autowired` annotation before the constructor argument:

▼ Java

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    public Runner(@Autowired PasswordGenerator generator) {
        this.generator = generator;
    }

    // run
}
```

▼ Kotlin

```kotlin
@Component
class Runner(@Autowired private val generator: PasswordGenerator) : CommandLineRunner {
    // run
}
```

3. Place the `@Autowired` annotation directly on the field to be injected:

▼ Java

```java
@Component
public class Runner implements CommandLineRunner {

    @Autowired
    private PasswordGenerator generator;

    // run
}
```

▼ Kotlin

```kotlin
@Component
class Runner : CommandLineRunner {
    @Autowired
    private lateinit var generator: PasswordGenerator

    // run
}
```

4. As another alternative, you can even omit the annotation and use a constructor. It is possible because Spring IoC knows all the components and can inject them by the type when it is needed:

▼ Java

```java
@Component
public class Runner implements CommandLineRunner {
    private final PasswordGenerator generator;

    public Runner(PasswordGenerator generator) {
        this.generator = generator;
    }

    // run
}
```

▼ Kotlin

```kotlin
@Component
class Runner(private val generator: PasswordGenerator) : CommandLineRunner {
```

Table of contents:

```
    // run
}
```

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

> When you use constructor injection, the `@Autowired` annotation can be omitted, but it is required when you use field injection, otherwise your fields will be `null`. We're going to continue using the annotation explicitly to make the learning process a bit easier.

§5. Conclusion

Components are a special kind of classes that Spring IoC creates during startup and provides to any other bean constructor. The `@Component` annotation placed above a class means that there must be a bean of that class. Like other beans, a component can be injected using the `@Autowired` annotation. Moreover, this annotation can be used together with a constructor or without it.

As a bonus, we developed a small but useful Spring Boot application that generates random passwords using only two components. One of those components implemented the `CommandLineRunner` interface to be able to interact with the standard I/O. We hope this component-based approach will encourage you to build flexible and well-decomposed applications in the future!

Report a typo

137 users liked this piece of theory. 9 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing   Unskip this topic

Comments (3)   Useful links (1)   Show discussion

JetBrains Academy

Tracks
Pricing
For organizations

About
Contribute
Careers

😎 Become beta tester

Be the first to see what's new

Terms Support

Made with ♥ by Hyperskill and JetBrains