Scopes of beans

Theory

13 minutes reading

Unskip this topic Start practicing

Spring provides several annotations to declare container-managed objects named **beans**. They are created once and injected automatically into all target places of your application. So far you have relied only on default container settings to work with the beans but, as a matter of fact, there is much more room for their customization. One of the things you can customize is the **scope** of beans that defines the whole lifecycle of a bean: when it is created, injected, and deleted. In this topic, you will learn about various scopes, which will allow you to work with beans in a more flexible way.

2 required topics
@Bean vs @Component
ApplicationContext
1 dependent topic
Bean lifecycle

§1. Setting up the scope of a bean

As you already know, Spring beans are **singletons** by default. It means that they are created once and then reused wherever possible. As a result, you will have multiple references to the same bean. Here, a singleton is just one of many possible bean scopes.

In total, Spring provides six scopes: singleton, prototype, request, session, application, and websocket. The first two scopes can be used in any Spring applications, including console-based, while the four others are only available in web applications and rely on HTTP-based concepts such as HTTP requests, sessions, and so on.

In this topic, we'll discuss the *singleton* and *prototype* scopes in detail and then provide some basic information about the four other scopes to keep all the related concepts together.

To set up a scope, you should use the `@Scope` annotation that accepts the name of the scope as the value and allows you to define the scope of a bean when it is declared. This annotation can be applied to both `@Bean`-annotated method and `@Component`-annotated classes.

We'll see how it works in a moment.

§2. A template for future examples

To explain the idea of scopes, we will use a simple bean for counting integer numbers declared in the `AppConfig` class.

▼ Java

```java
@Configuration
public class AppConfig {

    @Bean
    public AtomicInteger createCounter() {
        return new AtomicInteger();
    }
}
```

▼ Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun createCounter(): AtomicInteger {
        return AtomicInteger()
    }
}
```

Another component of the app will be the `AppRunner` class, which is used to start the app as a standard console application.

▼ Java

```
@Component
public class AppRunner implements CommandLineRunner {
    private final AtomicInteger counter1;
    private final AtomicInteger counter2;

    public AppRunner(AtomicInteger counter1, AtomicInteger counter2)
{
        this.counter1 = counter1;
        this.counter2 = counter2;
    }



    @Override

    public void run(String... args) {

        counter1.addAndGet(2);

        counter2.addAndGet(3);

        counter1.addAndGet(5);

        System.out.println(counter1.get());

        System.out.println(counter2.get());

    }

}
```

▼ Kotlin

```
@Component
class AppRunner(
    private val counter1: AtomicInteger,
    private val counter2: AtomicInteger
) : CommandLineRunner {

    override fun run(vararg args: String) {
        counter1.addAndGet(2)
        counter2.addAndGet(3)

        counter1.addAndGet(5)

        println(counter1.get())

        println(counter2.get())

    }

}
```

Note that the class has two fields that are both counters. The `run` method just increments the counters and then prints the results. Depending on the configured scope of the `counter` bean, we will see different results of this code in the next sections.

§3. Singleton scope

As we have already mentioned, by default, any bean in Spring has the singleton scope and this scope is exceptionally useful in typical applications. It means that the container creates only one instance of it for the whole `ApplicationContext` and injects it in all beans when expected.

This scope can be also specified explicitly by putting the `@Scope` annotation with the string `"singleton"` as the value.

▼ Java

```java
@Bean
@Scope("singleton")
public AtomicInteger createCounter() { /* ... */ }
```

▼ Kotlin

```kotlin
@Bean
@Scope("singleton")
fun createCounter(): AtomicInteger { /* ... */ }
```

As an alternative to the string, there is a predefined constant in the `ConfigurableBeanFactory` class.

```kotlin
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
```

> Since the `@Scope` annotation is optional for singleton beans, you should decide whether to write it according to the conventions used in your project and team.

If we start the `AppRunner` class with the bean configured in such a way, we will see the following output:

```
10
10
```

The reason is that both `counter1` and `counter2` refer to the same bean, which is actually a singleton.

§4. Prototype scope

In some cases, we don't want to have the same bean used in different parts of an application. To turn a bean into a non-singleton, there is the prototype scope. When we use it, the container returns a new bean every time it should be injected into the target place. To define a bean with this scope, we must use the `@Scope` annotation with the string `"prototype"` as the value.

▼ Java

```java
@Bean
@Scope("prototype")
public AtomicInteger createCounter() { /* ... */ }
```

▼ Kotlin

```
@Bean
@Scope("prototype")
fun createCounter(): AtomicInteger { /* ... */ }
```

There is also a convenient predefined constant in the `ConfigurableBeanFactory` class.

```
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

To demonstrate the idea of this scope in practice, let's look at our example again. If we change the scope of the `counter` bean to prototype, we will get another result:

```
7
3
```

It means that the `counter1` and `counter2` fields refer to different beans that have independent states.

> When a prototype-scoped bean has a dependency on a singleton, a new prototype-scoped bean is created only once when the singleton's constructor is initialized.

Injecting a prototype-scoped bean into another prototype-scoped bean, or injecting a singleton into a prototype-scoped bean is allowed as well.

§5. Other bean scopes

In addition to the previous two scopes, there are four other scopes that are only available in web applications:

- **Request** is a scope that allows a bean to be created and available for the whole lifecycle of an HTTP request. It means that if a request is processed by several Spring components, the request-scoped bean will be available in all of these components.
- **Session** is a scope that allows a bean to be created and available for the whole HTTP session that may include a sequence of HTTP requests connected by cookies/session ID into a single session.
- **Application** is a scope that allows a bean to be created and available for several applications (`ApplicationContext`) running in the same `ServletContext` . This scope is broader than the singleton scope which is scoped to a single application context only.
- **Websocket** is a scope that allows a bean to be created and available during the complete lifecycle of a WebSocket session.

To use any of these scopes, you need to pass its name to the `@Scope` annotation, e.g. `@Scope("request")` . There are also several additional annotations like `@RequestScope` , `@SessionScope` , and `@ApplicationScope` , which are the shortcuts for `@Scope` with a corresponding value. You can use these annotations only if your application has a dependency on the **spring-web** module.

§6. Conclusion

In this topic, you've explored the idea of bean scopes that allow configuring when beans are created and injected. We demonstrated how to use the `@Scope` annotation and the difference between singleton and prototype-scoped beans, which can be used in any Spring application. We also mentioned the four other scopes (request, session, application, and websocket) that were specifically developed for web applications. You don't have to remember all of them right now. It is enough to get the main idea and continue studying them in web-related topics.

Table of contents:

Report a typo

78 users liked this piece of theory. 2 didn't like it. **What about you?**

Start practicing

Unskip this topic

Comments (9) Useful links (0) Show discussion