Computer science → Backend → Spring Boot → Core container

ApplicationContext

Theory Practice 100% completed, 6 problems solved

Theory

18 minutes reading

Start practicing

1 required topic

✓ @Bean vs @Component

1 dependent topic

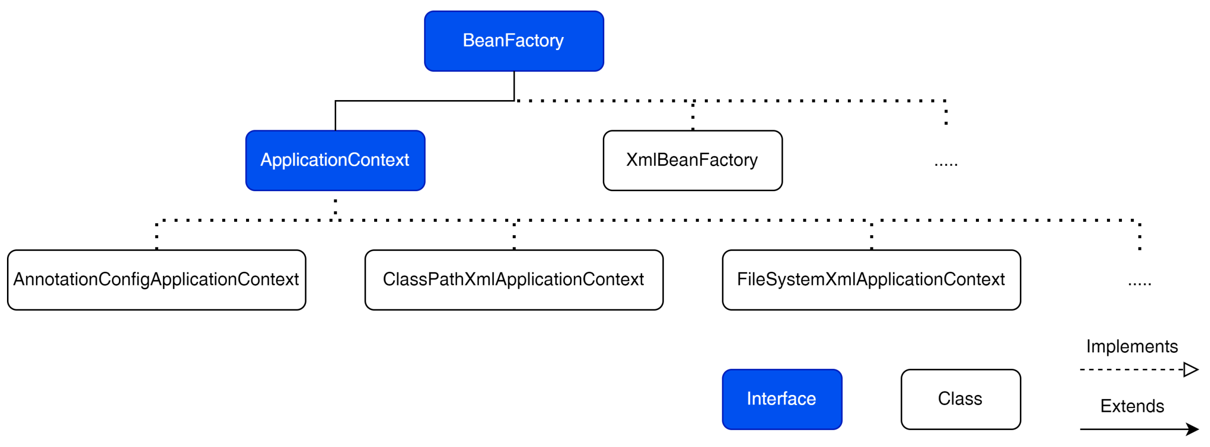
✓ Scopes of beans

The core concept of Spring is the **IoC (Inversion of Control) container**, which is responsible for managing beans. The IoC container is an object that is engaged in creating other objects (beans) and injecting dependencies in them. There are two primary interfaces that represent the IoC container: `BeanFactory` and `ApplicationContext`. These interfaces have many implementations for different purposes. In this topic, we will take a closer look at `ApplicationContext`, and consider some examples.

§1. BeanFactory & ApplicationContext

`BeanFactory` is a **root** interface for accessing the Spring IoC container. As for `ApplicationContext`, it extends it. That is, `ApplicationContext` gets its basic functionality from `BeanFactory` and extends it with additional features. For example, `ApplicationContext` supports AOP integration, event publication, all types of bean scopes, and more. Since `ApplicationContext` provides more opportunities, it is advisable to use it instead of `BeanFactory`.

In this diagram, you can see some of the implementations of `BeanFactory` and `ApplicationContext`:


BeanFactory
ApplicationContext
XmlBeanFactory
.....
AnnotationConfigApplicationContext
ClassPathXmlApplicationContext
FileSystemXmlApplicationContext
.....
Interface
Class
Implements
Extends

As you know, there are two ways to create metadata (bean definitions): by using an XML configuration file or annotations. `ApplicationContext` and `BeanFactory` are created based on metadata. That's why their implementations contain such words as "Xml" or "Annotation", indicating the type of metadata.

The main difference between these interfaces is that `BeanFactory` doesn't support annotation-based configuration, while `ApplicationContext` does. This fact gives a significant advantage to `ApplicationContext` over `BeanFactory`, because it's recommended to use annotation-based configuration for all new Spring applications.

§2. Creating an application context

Now let's create our own `ApplicationContext` based on annotation configuration.

Let's say we want to store objects of the `Person` type in our container. So, first of all, let's create a `Person` class:

▼ Java

```java
public class Person {

   private String name;

   public Person(String name) {
       this.name = name;
   }
}
```

▼ Kotlin

```kotlin
class Person(private val name: String)
```

An application context is created based on a configuration class, which means we need a configuration class that describes what objects (beans) will be created inside the IoC container:

▼ Java

```java
@Configuration
public class Config {

    @Bean
    public Person personMary() {
        return new Person("Mary");
    }
}
```

▼ Kotlin

```kotlin
@Configuration
open class Config {

    @Bean
    open fun personMary() = Person("Mary")
}
```

We filled this configuration class with one bean definition (metadata), based on which a future bean will be created inside the container.

Here is what we "say" to our `ApplicationContext`:

1. "Create a `Person` object with the property `name` = `Mary` ";
2. "Call the created bean `personMary` ";
3. "Place the bean into the IoC container".

`@Configuration` contains the `@Component` annotation inside, which also tells `ApplicationContext` to create a bean based on the `Config` class. So, before creating the `personMary` bean, `ApplicationContext` also needs to create a `config` bean and place it in the IoC container.

Now it's time to create an application context based on that `Config` class and get all the bean names (just String names, not objects) from it.

▼ Java

```java
public class Application {

    public static void main(String[] args) {
        var context = new
AnnotationConfigApplicationContext(Config.class);

System.out.println(Arrays.toString(context.getBeanDefinitionNames()));
```

```
    }
}
```

▼ Kotlin

```
fun main(args: Array<String>) {
    val context =
AnnotationConfigApplicationContext(Config::class.java)
    println(context.beanDefinitionNames.contentToString())
}
```

If we were using XML configuration, we would create, for example, `ClassPathXmlApplicationContext`.

In the output, among a number of internal beans (necessary for the work of a Spring application), you can see the names of our created beans: `config` and `personMary`.

```
[..., ..., config, personMary]
```

Also, `ApplicationContext` has overloaded the `getBean()` methods inherited from the `BeanFactory`:

▼ Java

- `T getBean(Class<T> requiredType)`
- `Object getBean(String name)`
- `T getBean(String name, Class<T> requiredType)`

▼ Kotlin

- `fun getBean(requiredType: Class<T>): T`
- `fun getBean(name: String): Any`
- `fun getBean(name: String, requiredType: Class<T>): T`

Let's get our bean (whole object) from the container by the `Person` class:

▼ Java

```
context.getBean(Person.class); // returns a Person object
```

▼ Kotlin

```
context.getBean(Person::class.java) // returns a Person object
```

If there are several beans of the same class in the container, you need to specify a unique bean name in the `getBean()` method. Otherwise, an exception will be thrown.

▼ Java

```
context.getBean("personMary"); // returns an Object object
context.getBean("personMary", Person.class) // returns a Person
object
```

▼ Kotlin

```
context.getBean("personMary") // returns an Any object
context.getBean("personMary", Person::class.java) // returns a
Person object
```

§3. @ComponentScan

Another way to create a bean is by using the `@Component` annotation placed above a class.

Let's create two `@Component` classes: `Book` and `Movie`:

▶ Java

▶ Kotlin

We aim to put these components into the same application context that is based on the `Config` class. In order for the configuration class to know about the existence of the `@Component` classes, the `@ComponentScan` annotation is used. We put this annotation above the configuration class name:

▼ Java

```java
@ComponentScan
@Configuration
public class Config {
    // bean definitions
}
```

▼ Kotlin

```kotlin
@ComponentScan
@Configuration
open class Config {
    // bean definitions
}
```

By default, `@ComponentScan` scans all the classes in the current package and all of its subpackages, looking for `@Component` classes (and all the other annotations containing `@Component` : `@Service` , `@Configuration` , and so on).

Now our context knows about the new beans: `book` and `movie` .

▼ Java

```java
public class Application {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(Config.class);

        // [..., ..., book, config, movie, personMary, ..., ...]
        System.out.println(Arrays.toString(context.getBeanDefinitionNames()));
    }
}
```

▼ Kotlin

```kotlin
fun main(args: Array<String>) {
    val context = AnnotationConfigApplicationContext(Config::class.java)

    // [..., ..., book, config, movie, personMary, ..., ...]
    println(context.beanDefinitionNames.contentToString())
}
```

You can change the default behavior of `@ComponentScan` and explicitly specify one or more base packages for scanning:

▼ Java

```java
@ComponentScan(basePackages = "packageName")
```

▼ Kotlin

```kotlin
@ComponentScan(basePackages = ["packageName"])
```

Since in `@ComponentScan` the `value` attribute is defined as an alias of `basePackages`, and vice versa, you can use them interchangeably. All these variants work in the same way:

▼ Java

```java
@ComponentScan(basePackages = "packageName")

@ComponentScan(value = "packageName")

@ComponentScan("packageName")
```

▼ Kotlin

```kotlin
@ComponentScan(basePackages = ["packageName"])

@ComponentScan(value = ["packageName"])

@ComponentScan("packageName")
```

§4. ApplicationContext in Spring Boot

Usually, in a Spring Boot application, our class is the entry point of the app, so we annotate it with the `@SpringBootApplication` annotation, and in the `main` method, we run our application.

▼ Java

```java
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
     SpringApplication.run(Application.class, args);
  }
}
```

▼ Kotlin

```kotlin
@SpringBootApplication
open class Application

fun main(args: Array<String>) {
    runApplication<Application>(*args)
}
```

During the execution of the `run` method, an application context is created. We can get the context and see what bean definitions it contains:

▼ Java

```java
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    ConfigurableApplicationContext context =
SpringApplication.run(Application.class, args);

System.out.println(Arrays.toString(context.getBeanDefinitionNames()));
  }
}
```

▼ Kotlin

```
@SpringBootApplication
open class Application

fun main(args: Array<String>) {
    val context = runApplication<Application>(*args)
    println(context.beanDefinitionNames.contentToString())
}
```

In the output, you can see an array of the bean names used in the application. Even if we didn't create our own beans, we will see a lot of internal beans created by Spring Boot to keep the application running.

`@SpringBootApplication` contains the `@ComponentScan` annotation, so our Spring Boot context will know about our custom `@Component` (`@Configuration` , `@Service` , `@Repository` , and so on) classes.

Another way to access `ApplicationContext` is to use the `@Autowired` annotation. Just inject the context created in the `SpringApplication.run(...)` method into another component. This allows us to get the context anywhere in the application.

Let's get all bean names from `ApplicationContext` inside the `Runner` class:

▼ Java

```
@Component
public class Runner implements CommandLineRunner {

    @Autowired
    ApplicationContext applicationContext;

    @Override
    public void run(String... args) throws Exception {

Arrays.toString(applicationContext.getBeanDefinitionNames());
    }
}
```

▼ Kotlin

```
@Component
open class Runner(var applicationContext: ApplicationContext) :
CommandLineRunner {

    override fun run(vararg args: String) {
        applicationContext.beanDefinitionNames.contentToString()
    }
}
```

§5. Conclusion

In this topic, we learned about the `ApplicationContext` interface representing the core Spring concept: the IoC container. It inherits `BeanFactory` and has methods for managing beans in the container. There are many implementations of `ApplicationContext` , for example, `AnnotationConfigApplicationContext` for annotation-based configuration, and `ClassPathXmlApplicationContext` for XML configuration. In Spring Boot, the `SpringApplication.run()` method takes over creating a context based on found beans, which definitely makes our life easier!

Report a typo

53 users liked this piece of theory. **5** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

This content was created 7 months ago and updated 2 days ago. Share your feedback below in comments to help us improve it!

Comments (5) Useful links (0) Show discussion