


[Computer science](#) → [Backend](#) → [Spring Boot](#) → [Core container](#)

Spring beans

Theory

Practice

 100% completed, 1 problem solved ▾

Theory

🕒 21 minutes reading

Unskip this topic

Start practicing

We often need to create different objects in an application to use their functionalities. Some of them need other objects as their dependencies, which in turn require other objects and so on. Spring offers a great way to simplify this huge and complicated chain of creating objects. It can create all the necessary objects during the application startup and put them all in a container. Then each class can retrieve any objects it needs from this container, no more creation and initialization constructions are needed!

These container-managed objects are known as **beans** and they organize the backbone of your application. They look exactly like standard Java or Kotlin objects but can be created during the application startup, registered, and then injected into different parts of an application by the container.

In this topic, we will look at how the **Spring IoC container** helps us initialize and use beans. We are going to start with the simplest possible example to grasp the basic idea and then gradually make it more complicated. Understanding how **DI (Dependency Injection)** works is crucial for all further topics in our materials. If you are already familiar with Spring or Spring Boot, the information in this topic may seem quite basic, but we hope you'll learn something new anyway!

§1. Initial preparations

Before you start using beans, make sure that you have the basic Spring Boot application.

▼ Java

```
1  @SpringBootApplication
2  public class DemoSpringApplication {
3
4      public static void main(String[] args) {
5          SpringApplication.run(DemoSpringApplication.class, args);
6      }
7
8  }
```


▼ Kotlin

```
1  @SpringBootApplication
2  class DemoSpringApplication
3
4  fun main(args: Array<String>) {
5      runApplication<DemoSpringApplication>(*args)
6  }
```


If you don't have it, you can generate it in your IDE or using [this website](#) and then follow the explanation here.

So, to start using beans, we are going to modify this piece of code and the **dependency injection** mechanism that is the core part of the entire Spring Boot framework.

1 required topic

✓  [IoC Container](#) ▾

1 dependent topic

✓  [Spring components](#) ▾

§2. Declaring beans

Beans are usually declared in the classes that have the `@Configuration` annotation, but it is also possible to declare them in the class containing the `@SpringBootApplication` annotation. In this lesson, we'll learn how to do it in both of these ways.

To declare a bean, you need to have a method containing the `@Bean` annotation. The result of executing this method will be a bean that is managed by the IoC container.

Here is an example of a simple bean declared in a configuration class:

▼ Java

```
1  @Configuration
2  public class Addresses {
3
4      @Bean
5      public String address() {
6          return "Green Street, 102";
7      }
8
9  }
```

▼ Kotlin

```
1  @Configuration
2  class Addresses {
3      @Bean
4      fun address(): String {
5          return "Green Street, 102"
6      }
7  }
```

This means that when you start the application, there will be a manageable string bean named `address` that contains the value `"Green Street, 102"`. Spring automatically invokes the method with the `@Bean` annotation during the startup in order to initialize all the declared beans.

By default, beans are **singletons**. It means that there is only one object for the whole application. But this default behavior can be changed. You will learn more about it in the following topics.

By default, the name of a bean is the same as the name of the method that produces it. However, the `@Bean` annotation allows you to rename it not to depend on the name of the method. All you need to do is specify the new name in the annotation, for example `@Bean("greenStreet")`. In this case, the name of the bean is `greenStreet`.

§3. Autowiring beans

Now that you have declared a bean, you can use it to create other beans that depend on it.

The Spring IoC container provides the **dependency injection (DI)** mechanism that allows us to do that. A bean that has a suitable type can be automatically injected into a method annotated with `@Bean`. There is also the `@Autowired` annotation that marks a constructor, a field, or a method as to be injected by Spring's DI.

In this next example, let's introduce an additional class that represents customers.

▼ Java

```
1  class Customer {
2      private final String name;
3      private final String address;
4
5      Customer(String name, String address) {
6          this.name = name;
7          this.address = address;
8      }
9
10     // getters
11
12     @Override
13     public String toString() {
14
15         return "Customer{" +
16
17             "name='" + name + '\'' +
18
19             ", address='" + address + '\'' +
20
21             '}';
22     }
23 }
```

▼ Kotlin

```
1  data class Customer(val name: String, val address: String)
```

This class includes the `address` field which we are going to get from our previous bean to create a new `Customer` object.

Considering that this is only the first topic about beans, for simplicity, all the following methods will be declared in the class annotated with `@SpringBootApplication`. However, in real-world applications, you will often see bean-annotated methods declared in `@Configuration` classes.

Here is a method that returns an object of this class as a bean. The `@Autowired` annotation marks the method parameter to be automatically injected.

▼ Java

```
1  @SpringBootApplication
2  public class DemoSpringApplication {
3
4      public static void main(String[] args) {
5          SpringApplication.run(DemoSpringApplication.class, args);
6      }
7
8      @Bean
9      public Customer customer(@Autowired String address) {
10
11          return new Customer("Clara Foster", address);
12      }
13 }
```

▼ Kotlin

```

1  @SpringBootApplication
2  class DemoSpringApplication {
3
4      @Bean
5      fun customer(@Autowired address: String): Customer {
6          return Customer("Clara Foster", address)
7      }
8  }
9
10 fun main(args: Array<String>) {
11     runApplication<DemoSpringApplication>(*args)
12 }

```

Spring DI injects the `address` bean into this method and this bean can be used to construct a new object of the `Customer` class. The injection works because the type of the bean we need is the same as the type of the bean produced earlier, and Spring Container can inject that bean. Even if the argument had another name (e.g. `addr`), this code would work as expected.

The `@Autowired` annotation is not always required to inject a bean. Spring will inject an appropriate bean anyway if a method has the `@Bean` annotation (or some others). We are always going to explicitly use this annotation in our examples to make the explanation easier for beginners. When you get more experience with Spring, you can decide whether to avoid this annotation.

You may wonder how we can be sure that both of the methods are invoked and the beans are successfully created. There is no need to introduce any new concepts — we can just create the third temporary bean depending on `Customer` and print the autowired bean. Add it in the same class where you've put the previous bean.

▼ Java

```

1  @Bean
2  public Customer temporary(@Autowired Customer customer) {
3      System.out.println(customer);
4      return customer;
5  }

```

▼ Kotlin

```

1  @Bean
2  fun temporary(@Autowired customer: Customer): Customer {
3      println(customer)
4      return customer
5  }

```

Now if you run the application, you will see the information about the customer in the log.

```

1  Customer{name='Clara Foster', address='Green Street, 102'}

```

Keep in mind that Spring prints a lot of log messages, and this info will be among them because beans are initialized during the application startup.

It means that Spring IoC correctly created all our beans and the beans are injected successfully.

There are no restrictions on the number of bean injection points in the code.

§4. Distinguishing beans of the same type

As we mentioned before, the location of an injection point is determined by the type of bean. But what if we have several beans of the same type and want to use a particular one? Fortunately, there is the `@Qualifier` annotation that allows us to specify the name of the bean we need to use.

▼ Java

```
1  @Bean
2  public String address1() {
3      return "Green Street, 102";
4  }
5
6  @Bean
7  public String address2() {
8      return "Apple Street, 15";
9  }
10
11
12 @Bean
13 public Customer customer(@Qualifier("address2") String address) {
14     return new Customer("Clara Foster", address);
15 }
16
17
18 @Bean
19 public Customer temporary(@Autowired Customer customer) {
20     // Customer{name='Clara Foster', address='Apple Street, 15'}
21     System.out.println(customer);
22     return customer;
23 }
```

▼ Kotlin

```
1  @Bean
2  fun address1(): String {
3      return "Green Street, 102"
4  }
5
6  @Bean
7  fun address2(): String {
8      return "Apple Street, 15"
9  }
10
11
12 @Bean
13 fun customer(@Qualifier("address2") address: String): Customer {
14     return Customer("Clara Foster", address)
15 }
```

```
1
6  @Bean
1
7  fun temporary(@Autowired customer: Customer): Customer {
1
8      // Customer{name='Clara Foster', address='Apple Street, 15'}
1
9      println(customer)
2
0      return customer
2
1  }
```

In this example, we specify the name of the bean we need to use to build the customer. The last bean named `temporary` is created only to print the information during the startup of the application.

If we deleted the `@Qualifier` from the customer method, the application wouldn't start and we'd get an error that there are several beans that can be injected:

```
Parameter 0 of method customer in
org.hyperskill.beans.DemoSpringApplication
required a single bean, but 2 were found:
- address1: defined by method 'address1' in
org.hyperskill.beans.DemoSpringApplication
- address2: defined by method 'address2' in
org.hyperskill.beans.DemoSpringApplication
```

So, if this error occurs, you just need to determine which bean you want to use and apply the `@Qualifier` annotation.

§5. Beans vs standard objects

Now you have a general idea of what Spring beans are and how to use them. But should you always use only beans in Spring and forget about standard objects? The answer is no. You can still use standard objects by creating them manually following the object-oriented programming approach:

▼ Java

```
1  String address = "Green Street, 102";
2  Customer customer = new Customer("Clara Foster", address);
```

▼ Kotlin

```
1  val address = "Green Street, 102"
2  val customer = Customer("Clara Foster", address)
```

In real applications, beans are usually used to form a backbone of your app and separate it by layers and configuration files, but most domain objects (like students, accounts, courses, etc.) are standard objects. In this topic, we deliberately used rather synthetic examples to show the logic behind beans without additional complexity. In the following topics, you will encounter a lot of examples where beans are much more convenient than standard objects.

§6. Conclusion

In this topic, you've learned about the capability of the Spring IoC container to create and inject beans on startup. Usually, beans are declared in classes annotated with the `@Configuration` annotation, but they can also be declared in the class annotated with the `@SpringBootApplication`. To declare a bean, you should create a method that has the `@Bean` annotation, and the result of this method will be a managed bean. The `@Autowired` annotation is used to mark that

Table of contents:

- [1 Spring beans](#)
- [§1. Initial preparations](#)
- [§2. Declaring beans](#)
- [§3. Autowiring beans](#)
- [§4. Distinguishing beans of the same type](#)
- [§5. Beans vs standard objects](#)
- [§6. Conclusion](#)
- [Discussion](#)

there is a bean injection expected and the `@Qualifier` annotation can help us specify the name of the bean to be injected in case of ambiguous situations.

Report a typo

😞 Thanks for your feedback!

Write here how we could improve this theory

Start practicing

Unskip this topic

This content was created over 1 year ago and updated 12 days ago. [Share your feedback below in comments to help us improve it!](#)

Comments (35)

Useful links (7)

Show discussion

JetBrains Academy

Tracks

Pricing

For organizations

About

Contribute

Careers

Become beta tester

Be the first to see what's new