Spring Start Here

Chapter-5: The Spring context: Bean scopes and life cycle

Upcode Software
Engineer Team

-2024-

CONTENT

THE SPRING CONTEXT: BEAN SCOPES AND LIFE CYCLE

Thus far we have discussed several essential things about object instances managed by Spring (beans).

1) We talked about **creating beans**
2) We discussed **establishing relationships among beans**
3) We discussed the **need to use abstractions**

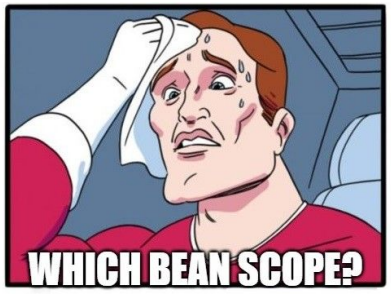
Spring has multiple different approaches for **creating beans** and **managing** their life cycle, and in the Spring world we name these approaches **scopes**. In this chapter, we discuss two scopes you'll often find in Spring apps: **singleton** and **prototype**.

THE SPRING CONTEXT: BEAN SCOPES AND LIFE CYCLE

We discuss two more bean scopes

1) We discuss the singleton bean scope
2) We continue by discussing the prototype bean scope

Our focus will be on how the **prototype scope** is different from **singleton** and real-world situations in which you'd need to apply one or another.



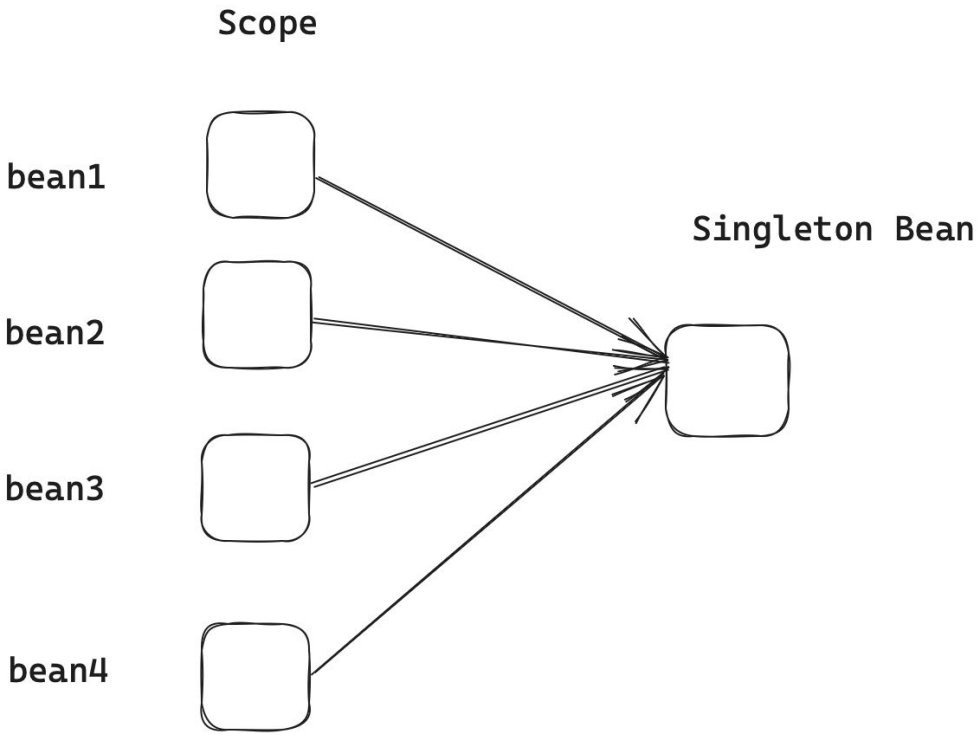
1. USING THE SINGLETON BEAN SCOPE (1/1)

The singleton bean scope defines Spring's default approach for managing the beans in its context.

- Spring creates a singleton bean when it loads the context and assigns the bean a name (sometimes also referred to as bean ID). We name this scope singleton because you always get the same instance when you refer to a specific bean.

- You can have more instances of the same type in the Spring context if they have different names

1. USING THE SINGLETON BEAN SCOPE (1/2)



HOW SINGLETON BEANS WORK(1/1)

For Spring, the singleton concept allows multiple instances of the same type, and singleton means unique per name but not unique per app (figure 5.1).

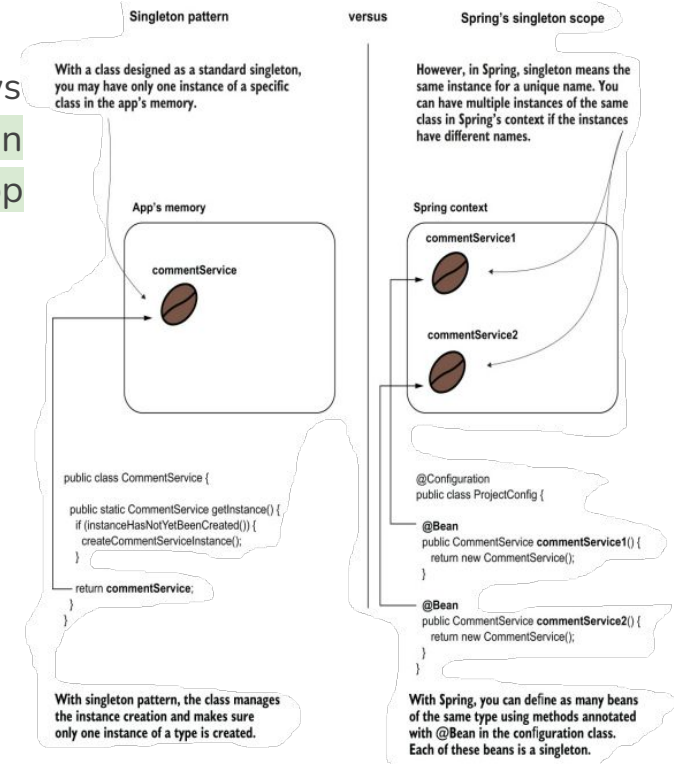
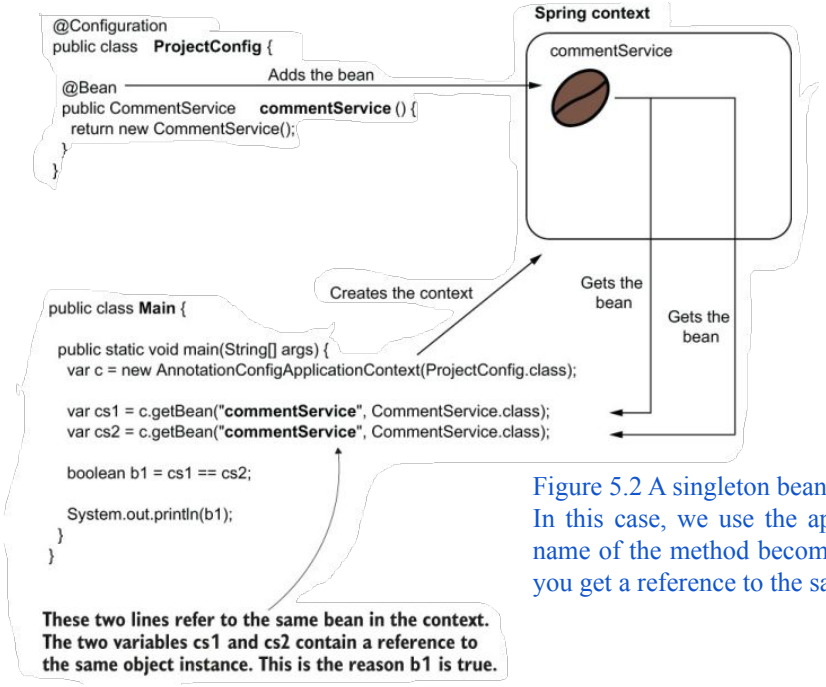


Figure 5.1 When one refers to a singleton class in an app, they mean a class that offers only one instance to the app and manages the creation of that instance. In Spring, however, singleton doesn't mean the context has only one instance of that type. It just means that a name is assigned to the instance, and the same instance will always be referred through that name.

HOW SINGLETON BEANS WORK(1/2)

DECLARING SINGLETON-SCOPED BEAN WITH @BEAN



We do this to prove we get the same instance every time we refer to the bean

Figure 5.2 A singleton bean. The app initializes the context when starting and adds a bean. In this case, we use the approach with the @Bean annotation to declare the bean. The name of the method becomes the identifier of the bean. Wherever you use that identifier, you get a reference to the same instance

1. USING THE SINGLETON BEAN SCOPE

Let’s write the code and run it to conclude this example. (using @Bean annotation)

CommentService.java

```java
package uz.temur.chapter5.sq_ch5_ex1;

public class CommentService {
}
```

ProjectConfig.java

```java
@Configuration
public class ProjectConfig {

    Adds the CommentService bean to the Spring context
    @Bean //@
    public CommentService commentService() {
        return new CommentService();
    }
}
```

Main.java

```java
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    Running the app will print "true" in the console because, being a singleton bean, Spring returns the same reference every time
    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);
        var cs1 = c.getBean( name: "commentService", CommentService.class);
        var cs2 = c.getBean( name: "commentService", CommentService.class);
        boolean b1 = cs1 == cs2; //@
        System.out.println(b1);
    }                    RESULT
}

/** ❶ Because the two variables hold the same reference, the result of this operation is true. */
```

Run — Main

```
C:\Users\admin\.jdks\openjdk-22.0.1\bin\java.exe ...
true

Process finished with exit code 0
```

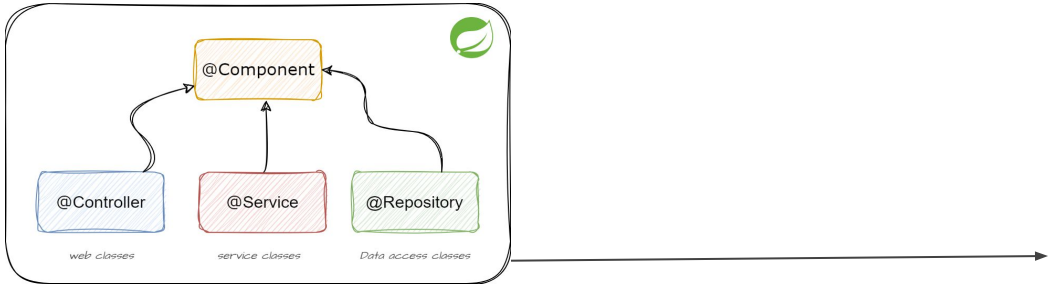
How singleton beans work(1/1)

DECLARING SINGLETON BEANS USING STEREOTYPE ANNOTATIONS

As mentioned earlier

- Spring's behavior for singleton beans isn't any different when using stereotype annotations than when you declared them with the @Bean annotation. But in this section, I'd like to enforce this statement with an example.

Spring stereotype annotations



How singleton beans work(1/3)

Two service classes depend on a repository

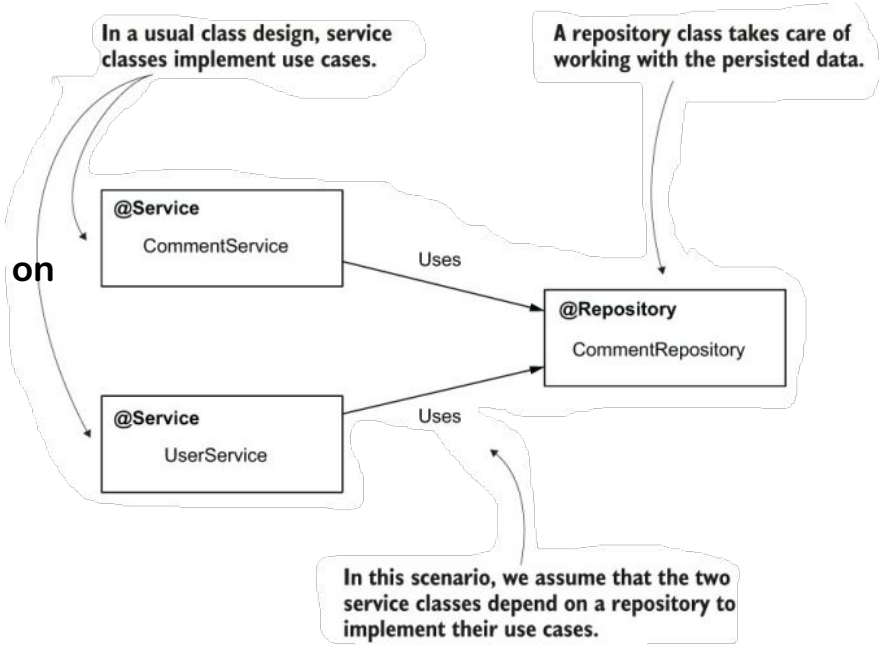


Figure 5.3 A scenario class design. Two service classes depend on a repository to implement their use cases. When designed as singleton beans, Spring's context will have one instance of each of these classes.

How singleton beans work(1/2)

We focus on the relationship between beans and how Spring establishes the links in its context

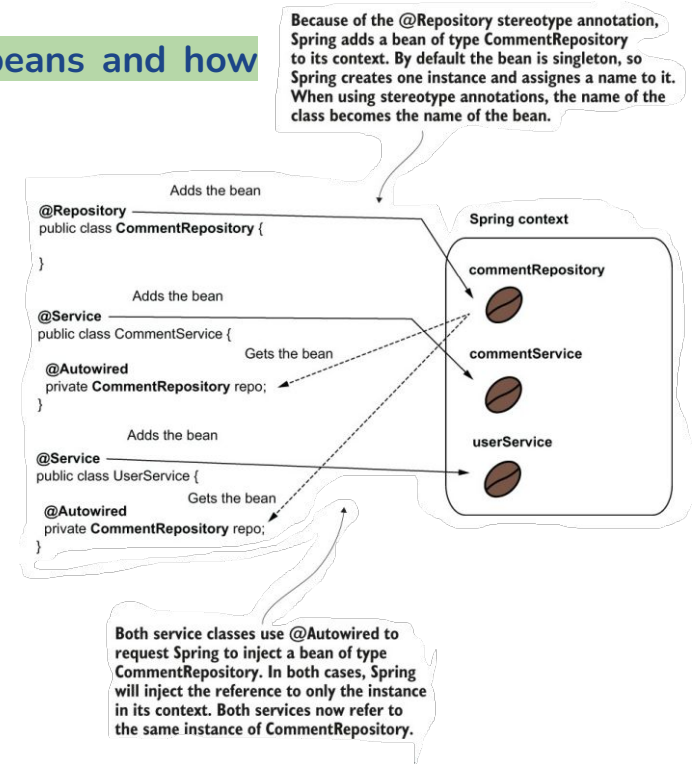


Figure 5.4 The beans are also singleton-scoped when using stereotype annotations to create them. When using @Autowired to request Spring to inject a bean reference, the framework injects the reference to the singleton bean in all the requested places.

Let's demonstrate this behavior with an example. (using Stereotype annotations)

Main.java

```java
import uz.temur.chapter5.sq_ch5_ex2.services.UserService;

public class Main {
    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(
                ProjectConfig.class); //● Creates the Spring context based on the configuration class
        var s1 = c.getBean(CommentService.class); //●
        var s2 = c.getBean(UserService.class); //●
        //● Gets the references of the two service beans in the Spring context
        boolean b = //● Compares the references for the repository dependency injected by Spring
        s1.getCommentRepository() == s2.getCommentRepository();
        System.out.println(b); //●
        //Because the dependency (CommentRepository) is singleton, both services contain the same reference,
        //so this line always prints "true."
    }
}
```

CommentRepository.java

```java
@Repository
public class CommentRepository {
}
```

CommentService.java

```java
@Service
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
```

ProjectConfig.java

```java
package uz.temur.chapter5.sq_ch5_ex2;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = {
        "uz.temur.chapter5.sq_ch5_ex2.services",
        "uz.temur.chapter5.sq_ch5_ex2.repositories"})
public class ProjectConfig {
}
```

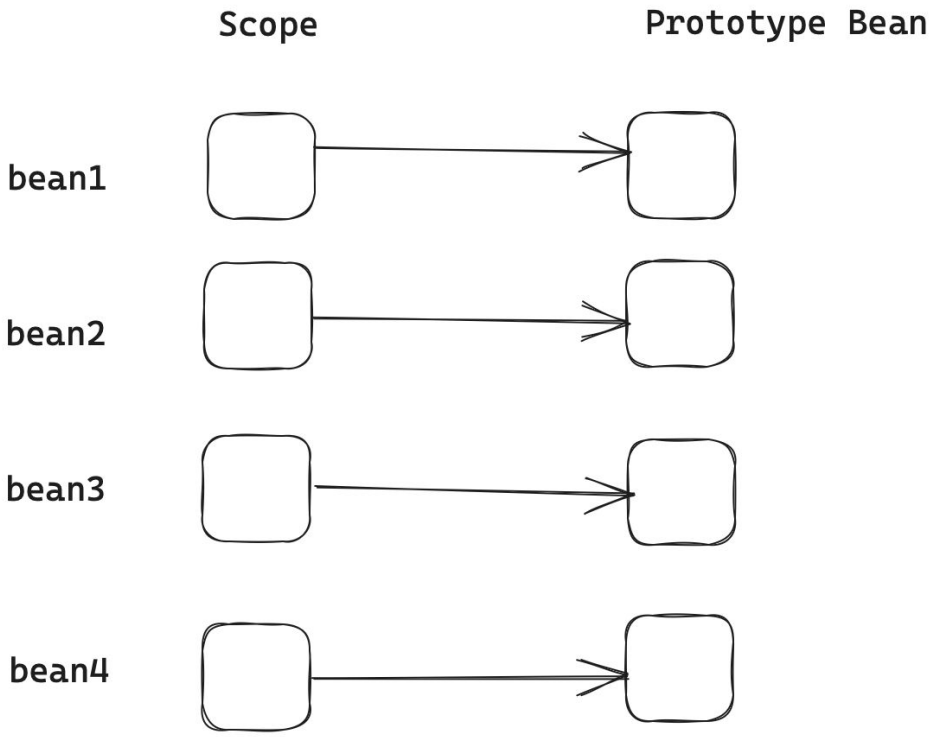
UserService.java

```java
@Service
public class UserService {

    @Autowired
    private CommentRepository commentRepository;

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
```

Run uz.temur.chapter5.sq_ch5_ex2.Main

```
C:\Users\admin\.jdks\openjdk-22.0.1\bin\java.exe ...
true
```

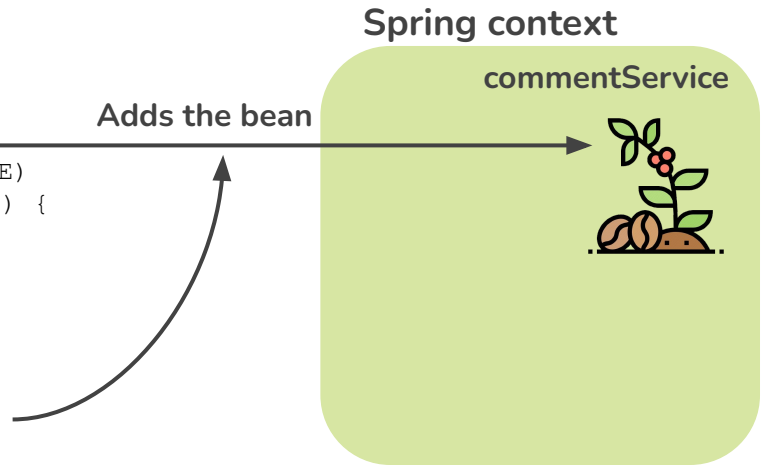
2. USING THE PROTOTYPE BEAN SCOPE



2.1. HOW PROTOTYPE BEANS WORK

The idea is straightforward. Every time you request a reference to a prototype-scoped bean, Spring creates a new object instance. For prototype beans, Spring doesn't create and manage an object instance directly. The framework manages the object's type and creates a new instance every time someone requests a reference to the bean.

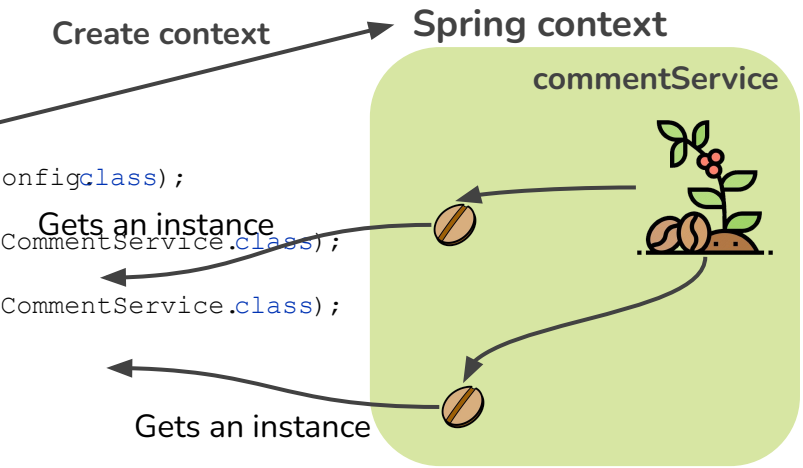
```
@Configuration
public class ProjectConfig {

    @Bean
    @Scope(BeanDefinition.SCOPE_PROTOTYPE)
    public CommentService commentService() {
        return new CommentService();
    }
}
```



Spring creates a bean and adds it to its context. Spring uses the type of the bean to create new instances each time they are requested.

2.1. HOW PROTOTYPE BEANS WORK

```
public class Main {
   public static void main(String[] args) {
       var c = new

AnnotationConfigApplicationContext(ProjectConfig.class);

       var cs1 = c.getBean("commentService",CommentService.class);

       var cs2 = c.getBean("commentService",CommentService.class);

       boolean b1 = cs1 == cs2;
       System.out.println(b1);
   }
}
```



Spring creates a new instance every time the getBean() method is called. The variables cs1 and cs2 always contain references to two different instances.

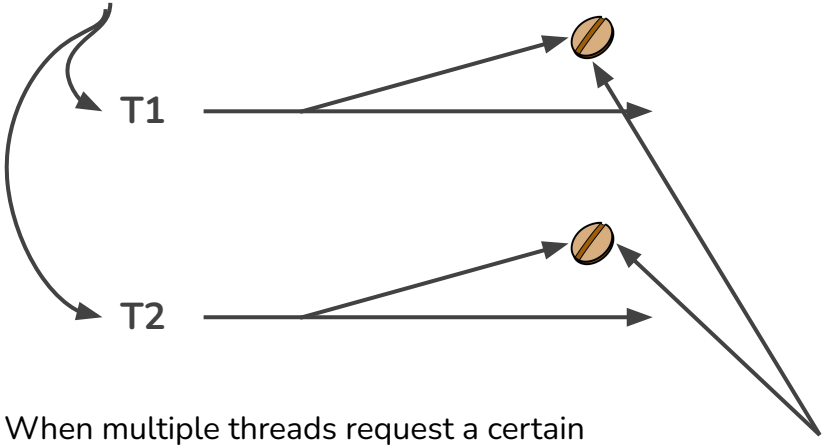
This line always prints “false.”

2.1. HOW PROTOTYPE BEANS WORK

Analyse:
We use the @Scope annotation to change the bean scope in prototype. The bean is now represented as a coffee plant because you get a new object instance each time you refer to it. For this reason, variables cs1 and cs2 will always contain different references, so the output of the code is always “false.”

2.1. HOW PROTOTYPE BEANS WORK

These arrows represent the execution timelines of two different threads named T1 and T2.



Spring context

When multiple threads request a certain prototype bean, each thread gets a different instance. This way, the threads cannot run into a race condition

If the two threads get this bean, each will get and work with different object instances. Each thread has its own instance.

2.1. HOW PROTOTYPE BEANS WORK

Declaring prototype-scoped beans with @bean(1/2)

```
@Configuration
public class ProjectConfig {

    @Bean
    @Scope(BeanDefinition.SCOPE_PROTOTYPE)
    public CommentService commentService() {
        return new CommentService();
    }
}
```

Makes this bean prototype-scoped

```
public class CommentService {
}
```

2.1. HOW PROTOTYPE BEANS WORK

Declaring prototype-scoped beans with @bean(2/2)

Testing Spring's behavior for the prototype bean in the Main class

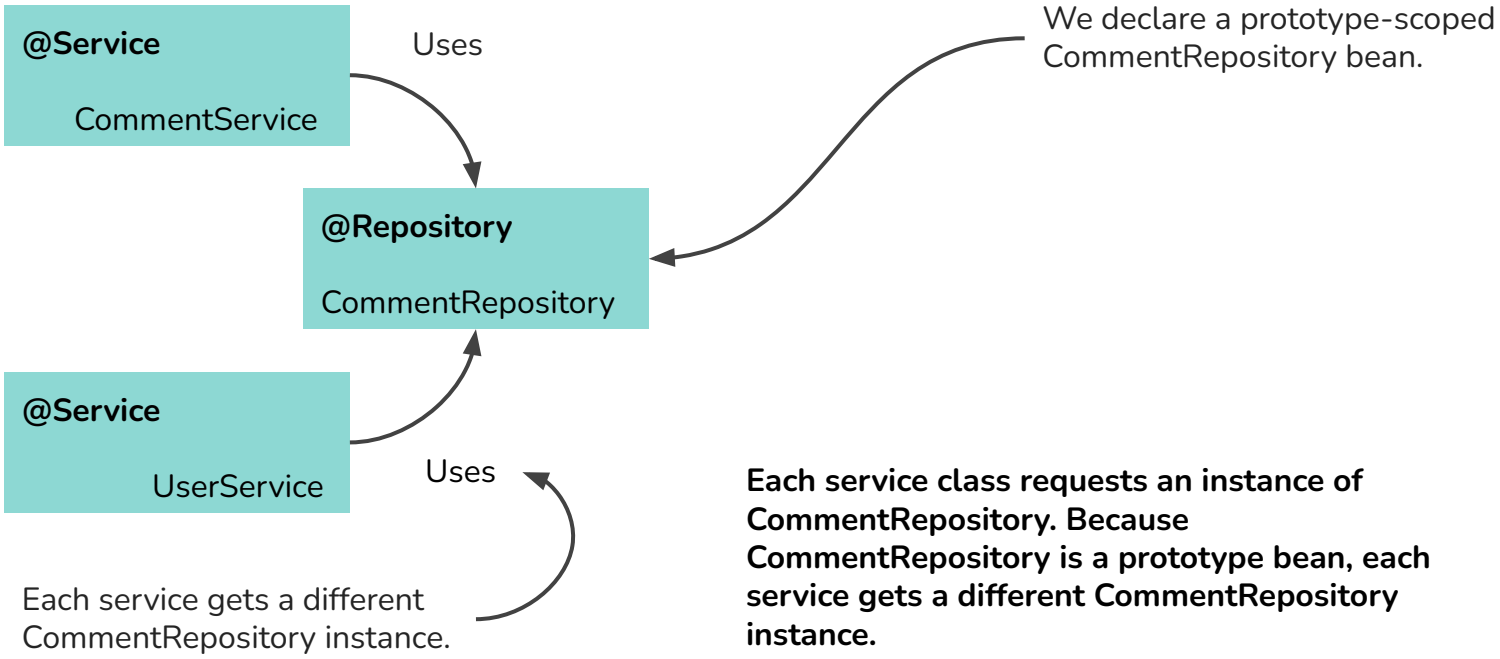
```
public class Main {
   public static void main(String[] args) {
       var c = new AnnotationConfigApplicationContext(ProjectConfig. class);
       var cs1 = c.getBean("commentService", CommentService.class);
       var cs2 = c.getBean("commentService", CommentService.class);
       boolean b1 = cs1 == cs2;

       System.out.println(b1);
   }
}
```

The two variables cs1 and cs2 contain references to different instances.

This line always prints "false" in the console.

2.1. HOW PROTOTYPE BEANS WORK

Declaring prototype-scoped beans using stereotype annotations



Each service class requests an instance of CommentRepository. Because CommentRepository is a prototype bean, each service gets a different CommentRepository instance.

2.1. HOW PROTOTYPE BEANS WORK

Declaring prototype-scoped beans using stereotype annotations

```
@Repository
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class CommentRepository {
}
```

```
@Service
public class CommentService {
    @Autowired
    private CommentRepository commentRepository;
    public CommentRepository getCommentRepository()
{
        return commentRepository;
    }
}
```

```
@Configuration
@ComponentScan(basePackages = {"services",
"repositories"})
public class ProjectConfig {
}
```

2.1. HOW PROTOTYPE BEANS WORK

Declaring prototype-scoped beans using stereotype annotations

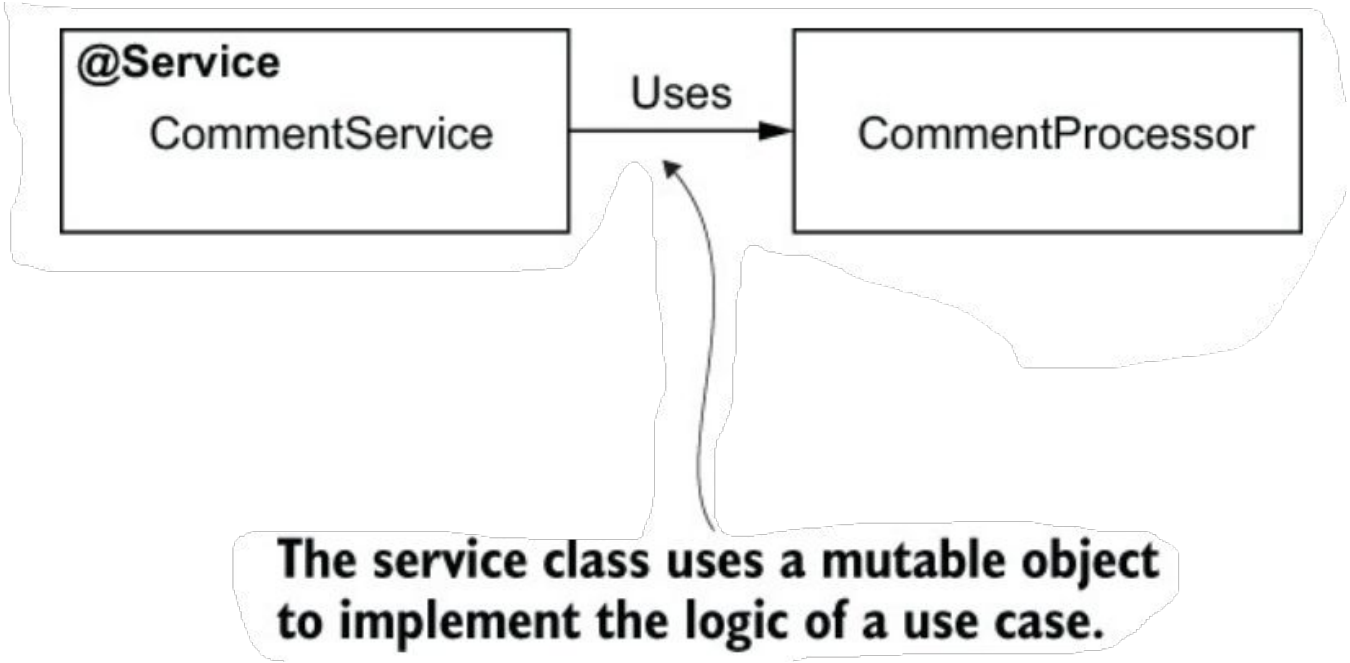
Testing Spring's behavior for injecting the prototype bean in the Main class

```
public class Main {
   public static void main(String[] args) {
       var c = new AnnotationConfigApplicationContext(ProjectConfig. class);

       var s1 = c.getBean(CommentService. class);
       var s2 = c.getBean(UserService. class);

       boolean b =
               s1.getCommentRepository() == s2.getCommentRepository();

       System.out.println(b);
   }
}
```

Gets references from the context for the service beans

Compares the references for the injected CommentRepository instances. Because CommentRepository is a prototype bean, the result of the comparison is always **false**.

2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS



2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

A mutable object: a potential candidate to the prototype scope

```
public class CommentProcessor {
   private Comment comment;
   public void setComment(Comment comment) {
       this.comment = comment;
   }
   public void getComment() {
       return this.comment;
   }

   public void processComment() {
       // changing the comment attribute
   }

   public void validateComment() {
       // validating and changing the comment attribute
   }
}
```

These two methods alter the value of the Comment attribute.

2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

A service using a mutable object to implement a use case

```
@Service
public class CommentService {
   public void sendComment(Comment c) {
       CommentProcessor p = new CommentProcessor();

       p.setComment(c);
       p.processComment(c);
       p.validateComment(c);

       c = p.getComment();
      // do something further
   }
}
```

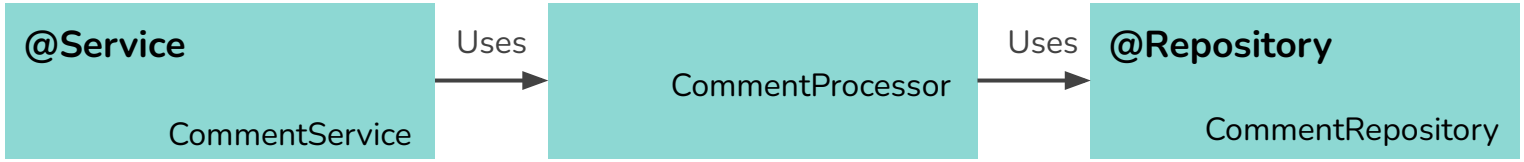
Creates a CommentProcessor instance

Uses the CommentProcessor instance to alter the Comment instance

Gets the modified Comment instance and uses it further

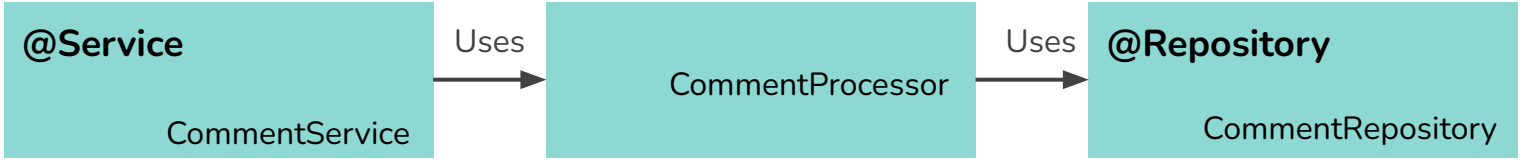
2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

The CommentProcessor object is not even a bean in the Spring context. Does it need to be a bean?



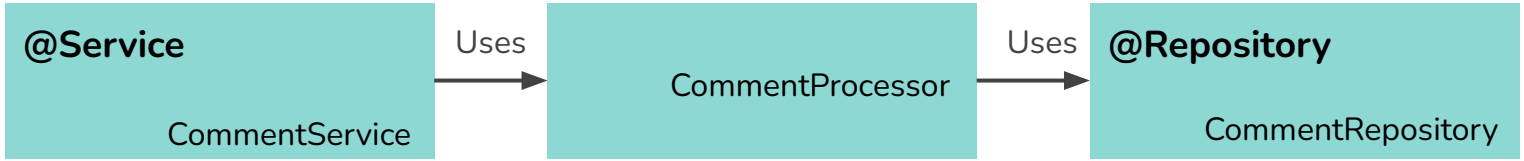
2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

The CommentProcessor object needs a bean from the Spring context. The easiest way to get an instance of CommentRepository is to request a DI. But to do this, Spring needs to know about CommentProcessor, so the CommentProcessor object needs to be a bean in the context.



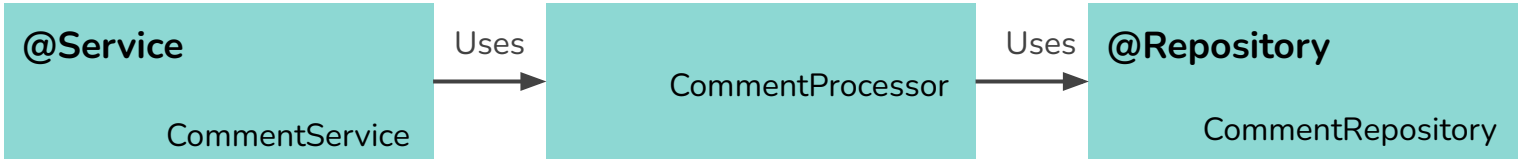
2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

We make CommentProcessor a bean in the Spring context. But can it be singleton scoped?



2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

No. If we define this bean as singleton and multiple threads use it concurrently, we get into a race condition.We would not be sure which comment provided by which thread is processed and if the comment was processed correctly.



2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

We can change the CommentProcessor class to be a prototype bean, as presented in the next code snippet:

```
@Component
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class CommentProcessor {
   @Autowired
   private CommentRepository
commentRepository;
   // Omitted code
}
```

2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

What will happen if we inject to field of CommentSerivice?

```
@Service
public class CommentService {
    @Autowired
    private CommentProcessor p;

    public void sendComment(Comment c) {

        p.setComment(c);
        p.processComment(c);
        p.validateComment(c);

        c = p.getComment();
        // do something further

    }
}
```

Spring injects this bean when creating the CommentService bean. But because CommentService is singleton, Spring will also create and inject the CommentProcessor just once.

2.2. PROTOTYPE BEANS IN REAL-WORLD SCENARIOS

Using CommentProcessor as prototype bean

```
@Service
public class CommentService {
   @Autowired
   private ApplicationContext context;
   public void sendComment(Comment c) {

       CommentProcessor p =
context.getBean(CommentProcessor.class);

       p.setComment(c);
       p.processComment(c);
       p.validateComment(c);
       c = p.getComment();
       // do something further
   }
}
```

A new CommentProcessor instance is always provided here.

2.3. COMPARISON BETWEEN SINGLETON AND PROTOTYPE BEAN SCOPES

Singleton	Prototype
The framework associates a name with an actual object instance.	A name is associated with a type.
Every time you refer to a bean name you'll get the same object instance.	Every time you refer to a bean name, you get a new instance.
You can configure Spring to create the instances when the context is loaded or when first referred.	The framework always creates the object instances for the prototype scope when you refer to the bean.
Singleton is the default bean scope in Spring.	You need to explicitly mark a bean as a prototype.
It's not recommended that a singleton bean to have mutable attributes.	A prototype bean can have mutable attributes.

CONCLUSION(1/2)

- In Spring, the scope of beans defines how the framework manages the object instances.
- Spring offers two bean scopes: singleton and prototype.
  - With singleton, Spring manages the object instances directly in its context. Each instance has a unique name, and using that name you always refer to that specific instance. Singleton is Spring's default.
  - With prototype, Spring considers only the object type. Each type has a unique name associated with it. Spring creates a new instance of that type every time you refer to the bean name.
- You can configure Spring to create a singleton bean either when the context is initialized (eager) or when the bean is referred for the first time (lazy). By default, a bean is eagerly instantiated.

CONCLUSION(2/2)

- In apps, we most often use singleton beans. Because anyone referring to the same name gets the same object instance, multiple different threads could access and use this instance. For this reason, it's advisable to have the instance immutable.
- If, however, you prefer to have mutating operations on the bean's attribute, it's your responsibility to take care of the thread synchronization. If you need to have a mutable object like a bean, using the prototype scope could be a good option.
- Be careful with injecting a prototype-scoped bean into a singleton-scoped bean. When you do something like this, you need to be aware that the singleton instance always uses the same prototype instance, which Spring injects when it creates the singleton instance. This is usually a vicious design because the point of making a bean prototype-scoped is to get a different instance for every use.

Reference

1. Spring Start Here
2. Spring Framework Series - Bean

Resources

MANNING

Thank you!

Presented by

Temurmalik Nomozov

(temirmaliknomozov@gmail.com)