Custom Bean Scope in Spring

Last Updated : 23 Jul, 2025

In Spring, Objects are managed by the Spring IOC(Inversion of Control) container, and their lifecycle is determined by the scope. Spring provides two standard scopes, which are listed below:

- **Singleton Scope:** One instance of the bean is created per Spring IOC Container. This is the default scope.
- **Prototype Scope:** A new instance of the bean is created every time it is requested.

Prerequisite: Singleton and Prototype Bean Scopes in Java Spring

Spring also offers three web-aware scopes, which are **Request Scope, Session Scope, and GlobalSession Scope**

Why Custom Scopes?

Spring scopes are powerful, but they may not always fit our application needs. For example:

- In a **multi-tenant system**, we might want a separate instance of a bean for each tenant.
- In a **thread-based application**, we might need a bean that is unique to each thread.

Custom beans allow us to define bean lifecycles that align with our application's requirements

Understanding the Scope Interface

To create a custom scope, we need to implement the **org.springframework.beans.factory.config.Scope** interface. This interface defines four methods:

- **Object get(String name, ObjectFactory<?> objectFactory):** Retrieves an object from the scope. If the object doesn't exist, it creates one using the ObjectFactory.
- **Object remove(String name):** Removes an Object from the scope.
- **void registerDestructionCallback(String name, Runnable destructionCallback):** Registers a callback to be executed when the object is destroyed or the scope ends.
- **String getConversationId():** Returns a unique identifier for the current scope (e.g., thread ID, session ID).

Step-by-Step Implementation

Let's discuss the process of creating and using a custom scope. We'll create a Thread-specific scope, where each thread gets its own instance of a bean.

Step 1. Create Custom Bean Scope Class

Implement the Scope interface to define your custom scope. Below is an example of a thread-specific scope:

```java
import org.springframework.beans.factory.ObjectFactory;
import org.springframework.beans.factory.config.Scope;

import java.util.HashMap;
import java.util.Map;

public class CustomThreadScope implements Scope {
    private final ThreadLocal<Map<String, Object>> threadScope =
ThreadLocal.withInitial(HashMap::new);

    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        Map<String, Object> scope = threadScope.get();
        return scope.computeIfAbsent(name, k -> objectFactory.getObject());
    }

    @Override
    public Object remove(String name) {
        Map<String, Object> scope = threadScope.get();
        return scope.remove(name);
    }

    @Override
    public void registerDestructionCallback(String name, Runnable callback) {
        // Implement destruction callback logic if needed
    }

    @Override
    public String getConversationId() {
        return String.valueOf(Thread.currentThread().getId());
    }
}
```

Step 2. Register the Custom Scope

In the above step, we have created a Thread bean scope class. Now we have to register this bean to the Spring application context. Then we will get the benefit of the newly created scope class in the application. Here there are two ways through which we can register custom scope i.e. programmatic registration or we can do it via using XML-based configuration. In this article, we will go with programmatic registration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.beans.factory.config.CustomScopeConfigurer;

@Configuration
public class AppConfig {

    @Bean
    public static CustomScopeConfigurer customScopeConfigurer() {
        CustomScopeConfigurer configurer = new CustomScopeConfigurer();
        configurer.addScope("thread", new CustomThreadScope());
        return configurer;
    }
}
```

Note: Here, the CustomScopeConfigurer is used to register the CustomThreadScope with the name "thread".

Step 3. Using the custom scope in Bean Definitions

Now that the custom scope is registered, we can use it in our bean definitions.

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("thread")
public class MyBean {
    private final String threadId = 
String.valueOf(Thread.currentThread().getId());

    public void printThreadId() {
        System.out.println("Bean instance created by thread: " + 
threadId);
    }
}
```

Step 4. Testing of Custom Scope

Let's test the custom scope to ensure it works as expected

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

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new 
AnnotationConfigApplicationContext(AppConfig.class);

        Runnable task = () -> {
            MyBean bean = context.getBean(MyBean.class);
            bean.printThreadId();
        };

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();
    }
}
```

Output:

```
Bean instance created by thread: 1
Bean instance created by thread: 2
```

Each thread creates its own instance of MyBean, demonstrating the custom thread scope in action.

SimpleThreadScope (Built-in-Alternative)

Spring 3.0 introduced a built-in thread scope called SimpleThreadScope. Instead of creating a custom scope we can use this out-of-the-box implementation

```
@Bean
public static CustomScopeConfigurer customScopeConfigurer() {
    CustomScopeConfigurer configurer = new CustomScopeConfigurer();
    configurer.addScope("thread", new SimpleThreadScope());
    return configurer;
}
```

Note: This eliminates the need to implement the Scope interface ourself.