**How to Inject Prototype Scoped Bean into a Singleton Bean in Spring**

In this post we'll see different ways of injecting prototype scoped bean into singleton scoped bean in Spring so that new instance of prototype scoped bean is created every time.

**Problem with injecting prototype scoped bean into singleton scoped bean**

If we go by the definition of the singleton and prototype beans, it says -

* **Singleton scope** - Only one shared instance of a **singleton bean** is managed by the container, and all requests for beans with an id matching that bean definition result in that one specific bean instance being returned by the Spring container.
* **Prototype Scope** - Prototype scope for a bean results in the creation of a new bean instance every time a request for that specific bean is made.

Now we are confronted with a situation when we want to inject a **prototype scoped bean** into a **singleton scoped bean** in Spring. Since dependencies are resolved at instantiation time which means if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean.

The prototype instance is the sole instance that is ever supplied to the singleton scoped bean. You cannot dependency-inject a prototype-scoped bean (new instance of bean every time) into your singleton bean, because that [dependency injection](http://netjs.blogspot.com/2015/09/what-is-dependency-injection-in-spring.html) occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies.

But that's not what you want, you have given a bean prototype scope with an intention that new instance of it should be created every time.

So let's see the **problem** first with some code. Let's say you have two classes **RequestManager** and**RequestHandler**. RequestManager is configured as a singleton bean where as RequestHandler is defined with a prototype scope.

<bean id="requestManager" class="org.netjs.prog.RequestManager">

<property name="requestHandler" ref="requestHandler" ></property>

</bean>

<bean id="requestHandler" class="org.netjs.prog.RequestHandler" scope="prototype">

</bean>

**RequestManager Class**

public class RequestManager {

private RequestHandler requestHandler;

public void handleRequest(){

requestHandler.handleRequest();

}

public RequestHandler getRequestHandler() {

return requestHandler;

}

public void setRequestHandler(RequestHandler requestHandler) {

this.requestHandler = requestHandler;

}

}

**RequestHandler Class**

public class RequestHandler {

RequestHandler(){

System.out.println("In Request Handler Constructor");

}

public void handleRequest(){

System.out.println("Handling request");

}

}

Now if you run this code, using this class -

public class App {

public static void main( String[] args ){

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext

("appcontext.xml");

RequestManager bean = (RequestManager) context.getBean("requestManager");

// calling method three times

bean.handleRequest();

bean.handleRequest();

bean.handleRequest();

context.close();

}

}

**Output**

In Request Handler Constructor

Handling request

Handling request

Handling request

Here, though method is called thrice, [constructor](http://netjs.blogspot.com/2015/04/constructor-in-java.html) is called only once which means RequestHandler instance is created only once.

**Solutions for injecting prototype scoped bean into singleton scoped bean**

There are 3 solutions to ensure that a new instance is created every time when prototype scoped bean is injected into a singleton scoped bean.

1. Implementing the ApplicationContextAware interface.
2. Lookup method injection
3. aop:scoped-proxy

**Implementing the ApplicationContextAware interface**

One **solution** is to implement the [ApplicationContextAware interface](https://netjs.blogspot.com/2018/03/applicationcontextaware-beannameaware-interface-spring.html) and use the application context reference to get the bean with in the code. In that case RequestManager class will look like this. *Note that according to the Spring docs this is not a good solution because the business code is aware of and coupled to the Spring Framework (look at the imports in the code). So if you want you can safely skip to the "Lookup method injection" solution*.

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class RequestManager implements ApplicationContextAware{

private RequestHandler requestHandler;

private ApplicationContext applicationContext;

public void handleRequest(){

requestHandler = getRequestHandler();

requestHandler.handleRequest();

}

// method to return new instance

public RequestHandler getRequestHandler() {

return applicationContext.getBean("requestHandler", RequestHandler.class);

//return requestHandler;

}

/\*public void setRequestHandler(RequestHandler requestHandler) {

this.requestHandler = requestHandler;

}\*/

@Override

public void setApplicationContext(ApplicationContext applicationContext)

throws BeansException {

this.applicationContext = applicationContext;

}

}

And in XML configuration reference for RequestHandler will be removed from the RequestManager configuration.

<bean id="requestManager" class="org.netjs.prog.RequestManager">

</bean>

<bean id="requestHandler" class="org.netjs.prog.RequestHandler" scope="prototype">

**Lookup method injection in Spring**

Lookup method injection is another way to inject prototype scoped bean into a singleton bean in Spring. You can define a look up method in your bean definition using the <lookup-method> element.

Lookup method injection is the ability of the container to [override methods](http://netjs.blogspot.com/2015/04/method-overriding-in-java.html) on container managed beans, to return the lookup result for another named bean in the container. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

In this case the **XML Configuration** will look like this

<bean id="requestManager" class="org.netjs.prog.RequestManager">

<lookup-method name="getRequestHandler" bean="requestHandler"/>

</bean>

<bean id="requestHandler" class="org.netjs.prog.RequestHandler" scope="prototype" />

Note that the bean which is defined with the look up method will be **dynamically subclassed** by the Spring framework (**using CGLIB library**) and this subclass will override and provide implementation for the methods which are configured as **look-up method**.

The dynamically generated proxy will *delegate all the non-lookup methods to the original class*. For the lookup methods it will use the implementation it has provided.

Since look-up method has to be implemented so it has to be either defined as abstract method or you can provide some dummy implementation.

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class.

If you are providing a **dummy implementation** then your RequestManager class will look like this -

public class RequestManager{

private RequestHandler requestHandler;

public void handleRequest(){

requestHandler = getRequestHandler();

requestHandler.handleRequest();

}

// dummy implmentation, configured as look-up method

public RequestHandler getRequestHandler() {

return null;

}

}

In case you are defining the **method as abstract** then you will have to mark the class also as [abstract class](http://netjs.blogspot.com/2015/04/abstract-class-in-java.html). It may create problem with in your whole implementation and also make the unit-testing difficult. Anyway in case you want it to be an abstract method then the RequestManager class will look like this -

public abstract class RequestManager{

private RequestHandler requestHandler;

public void handleRequest(){

requestHandler = getRequestHandler();

requestHandler.handleRequest();

}

public abstract RequestHandler getRequestHandler();

}

Now if you run it using this test class -

public class App {

public static void main( String[] args ){

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext

("appcontext.xml");

RequestManager bean = (RequestManager) context.getBean("requestManager");

bean.handleRequest();

bean.handleRequest();

bean.handleRequest();

context.close();

}

}

**Output**

In Request Handler Constructor

Handling request

In Request Handler Constructor

Handling request

In Request Handler Constructor

Handling request

Now you can see that three instances of RequestHandler are created for three separate calls.

Some of the points to remember when using look-up method -

* For this dynamic subclassing to work, the class that the Spring bean container will subclass cannot be final, and the method to be overridden cannot be final either.
* Unit-testing a class that has an abstract method requires you to subclass the class yourself and to supply a stub implementation of the abstract method.
* Concrete methods are also necessary for component scanning which requires concrete classes to pick up.

**Using aop:scoped-proxy in Spring**

Third way to inject prototype scoped bean in a singleton bean is using aop scoped proxy. Though Spring docs say "*You do not need to use the <aop:scoped-proxy/> in conjunction with beans that are scoped as singletons or prototypes.*" As it is more suitable to be used in the scenario when you are working with request, session and application scope and want to resolve the problem of how long do you want your bean to live.

As Spring docs say "you don't" not "you shouldn't" so we can anyway use it with singleton and prototype too.

When using aop scoped proxy the XML configuration will look like this -

<bean id="requestManager" class="org.netjs.prog.RequestManager">

<property name="requestHandler" ref="requestHandler"/>

</bean>

<bean id="requestHandler" class="org.netjs.prog.RequestHandler" scope="prototype">

<aop:scoped-proxy/>

</bean>

Note that with look up method solution it was the singleton bean which was getting proxied but with aop scoped proxy it is the prototype bean which will be proxied.

So if we take our classes as example, the container will create a proxy object of the **RequestHandler**which can fetch the real **RequestHandler** class object from the defined scoping mechanism (prototype, request, session etc.)

The container injects this proxy object into the requestManager bean, which is unaware that this requestHandler reference is a proxy.

When a RequestManager instance invokes a method on the dependency-injected RequestHandler object, it actually is invoking a method on the proxy. The proxy then fetches the real RequestHandler object and delegates the method invocation onto the retrieved real RequestHandler object.

There are 2 ways to create proxy class using aop scoped proxy.

1. Using CGLIB library, this is the default option.
2. Using JDK interface-based proxies for such scoped beans, by specifying false for the value of the proxy-target-class attribute of the <aop:scoped-proxy/> <aop:scoped-proxy proxy-target-class="false" />

**RequestManager Class when using aop scoped proxy**

public class RequestManager{

private RequestHandler requestHandler;

public void handleRequest(){

requestHandler.handleRequest();

}

public RequestHandler getRequestHandler() {

return requestHandler;

}

public void setRequestHandler(RequestHandler requestHandler) {

this.requestHandler = requestHandler;

}

}

**RequestHandler Class**

public class RequestHandler {

RequestHandler(){

System.out.println("In Request Handler Constructor");

}

public void handleRequest(){

System.out.println("Handling request");

}

}

**Output**

In Request Handler Constructor

Handling request

In Request Handler Constructor

Handling request

In Request Handler Constructor

Handling request

That's all for this topic **How to Inject Prototype Scoped Bean into a Singleton Bean in Spring**. If you have any doubt or any suggestions to make please drop a comment. Thanks!