**Injecting a prototype bean into a singleton bean**

In Spring, most of the beans we work with are Singletons. If a singleton bean is wired with yet another singleton bean, there is absolutely no problem. But if it is wired with a bean which is of different scope, say prototype, how does it work? Here is the example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class RequestProcessor {        private RequestValidator validator;        public void handleRequest(String requestId){          validator.validate(requestId);          // Process the request and update      }        public RequestValidator getValidator() {          return validator;      }        public void setValidator(RequestValidator validator) {          this.validator= validator;      }    }    public class RequestValidator {        private List<String> errorMessages = new ArrayList<String>();        public RequestValidator() {          System.out.println("Validator instance created!");      }        // Validates the request and populates error messages      public void validate(String requestId){        }        public List<String> getErrorMessages() {          return errorMessages;      }    } |

And here is the spring configuration:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">      <property name="validator" ref="validator"/>  </bean>    <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/> |

With this configuration, it is expected that when ever I fetch *requestProcessor* from application context, it will be wired with a new validator as we declared the *validator*bean is of prototype scope. But this does not happen. When the application context gets initialized, it sees that *requestProcessor* is a singleton bean and initializes it to the context after wiring it with all the dependencies set. So from then onwards when we request context for *requestProcessor*, it return the same bean every time. To solve this issue, we have 2 approaches:

**1. Lookup Method injection:** For this, we have to declare the beans as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">      <lookup-method name="getValidator" bean="validator"/>  </bean>    <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/> |

The Spring Framework implements method injection by using CGLIB library to generate dynamically a subclass that overrides the method. So for the method to be overridden, we have to define that method in the class and either provide a dummy implementation for it or make it abstract. Making a method abstract implies that class also has to be made abstract which will make it difficult to unit test. So providing a dummy implementation is a better choice.

Whenever we define a bean with lookup methods, Spring creates a subclass of the bean and overrides those methods which are marked as lookup-methods. And this subclassed bean gets registered into the context. The subclass delegates all the non-lookup methods to the original class. For the lookup methods, it overrides the implementation. So in our example, when getValidator() is called, it returns a new validator instance.

We can roughly imagine our new subclass(registered in container) like this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | requestProcessor = new RequestProcessor(){      public RequestValidator getValidator(){          return context.getBean("validator");      }  };  //, in case of look up method injection, proxy is created for singleton bean |

We could have directly fetched the bean from application context in RequestProcessor itself. But this would mean that the class is directly coupled to Spring framework. To do this in a cleaner way, we can use lookup injection. This puts all the spring related stuff at one place.

**2. Scoped Proxies:** This can be implemented as:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">      <property name="validator" ref="validator"/>  </bean>    <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator">      <!-- This instructs the container to proxy the current bean-->      <aop:scoped-proxy/>  </bean> |

Remember, in case of look up method injection, proxy is created for singleton bean. But in case of scoped proxies, proxy is created for prototype bean and wired into the singleton bean during the process of registering the singleton bean in the context. The proxy thus created understands the scope and returns instances based on the requirements of the scope. So in our case, *requestProcessor* holds a reference to proxy in place of *validator*.

And in case of lookup method injection, when *requestProcessor* gets loaded into the context, *validator* will not be initialized at all. And when we call the look up method, it returns the prototype bean. But instead of calling the method, if you try to directly access the prototype bean(assuming it is accessible), it gives a Nullpointer Exception as it didn’t get initialized(We are not wiring it using *property* tag of bean)

In case of this, we can also configure how a proxy can be created. It can be done in 2 ways  
1. CGLIB library which directly subclasses the object. This is the default option of Spring. For this, we must have CGLIB library our class path.  
2. Java Dynamic Proxies. For this to be activated, we have to call:

|  |  |
| --- | --- |
| 1 | <aop:scoped-proxy proxy-target-class="false"/> |

Here in this case, we don’t need any additional libraries in our class path. But the scoped bean must implement at least one interface and it has to be referred through the same interface at all places in order to get itself wired.

**Few points to note:**  
1. Both method injection and scoped proxies work not only for prototype beans. This works more generic. Whenever a bean of different scope is injected into a singleton bean, we can use any of these techniques to ensure that we get a corresponding scope object.  
2. Note that in the proxy, the method returning the prototype bean is overridden to return a new instance for every single call.  
Suppose we want to display the error messages that we have got after validation:

|  |  |
| --- | --- |
| 1  2  3  4  5 | requestProcessor.getValidator().validate();  for(String message: requestProcessor.getValidator().getErrorMessages()){      logger.log(LogLevel.ERROR, message);  } |

This code seems to print the error messages we have got after validation process. But this will never print any error messages even if there are many validation failures. This happens because requestProcessor.getValidator() returns a new validator instance every time it is called. So for this to work, the code has to be modified as:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | RequestValidator validator = requestProcessor.getValidator();  validator.validate();  for(String message: validator.getErrorMessages()){      logger.log(LogLevel.ERROR, message);  } |

This happens only in case of prototype beans but works perfectly in case of other non-singleton scopes(request, session, global-session).