(http://baeldung.com)

How to use the Spring FactoryBean?

Last modified: April 25, 2017

by baeldung (http://www.baeldung.com/author/baeldung/)

Spring (http://www.baeldung.com/category/spring/) +

1. Overview

There are two kinds of beans in the Spring bean container: ordinary beans and factory beans. Spring uses the former directly, whereas latter can produce objects themselves, which are managed by the framework.

And, simply put, we can build a factory bean by implementing *org.springframework.beans.factory.FactoryBean* interface.

2. The Basics of Factory Beans

2.1. Implement a *FactoryBean*

Let's look at the *FactoryBean* interface first:

```
public interface FactoryBean {
    T getObject() throws Exception;
    Class<?> getObjectType();
    boolean isSingleton();
}
```

Let's discuss the three methods:

- *getObject()* – returns an object produced by the factory, and this is the object that will be used by Spring container

- *getObjectType()* – returns the type of object that this *FactoryBean* produces
- *isSingleton()* – denotes if the object produced by this *FactoryBean* is a singleton

Now, let's implement an example *FactoryBean*. We'll implement a *ToolFactory* which produces objects of the type *Tool*:

```
public class Tool {

    private int id;

    // standard constructors, getters and setters
}
```

The *ToolFactory* itself:

```
public class ToolFactory implements FactoryBean<Tool> {

    private int factoryId;
    private int toolId;

    @Override
    public Tool getObject() throws Exception {
        return new Tool(toolId);
    }

    @Override
    public Class<?> getObjectType() {
        return Tool.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }

    // standard setters and getters
}
```

As we can see, the *ToolFactory* is a *FactoryBean*, which can produce *Tool* objects.

2.2. Use *FactoryBean* with XML-based Configuration

Let's now have a look at how to use our *ToolFactory*.

We'll start constructing a tool with XML-based configuration – *factorybean-spring-ctx.xml*:

```
<beans ...>

    <bean id="tool" class="com.baeldung.factorybean.ToolFactory">
        <property name="factoryId" value="9090"/>
        <property name="toolId" value="1"/>
    </bean>
</beans>
```

Next, we can test if the *Tool* object is injected correctly:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:factorybean-spring-ctx.xml" })
public class FactoryBeanXmlConfigTest {
    @Autowired
    private Tool tool;

    @Test
    public void testConstructWorkerByXml() {
        assertThat(tool.getId(), equalTo(1));
    }
}
```

The test result shows we manage to inject the tool object produced by the *ToolFactory* with the properties we configured in the *factorybean-spring-ctx.xml*.

The test result also shows that the Spring container uses the object produced by the *FactoryBean* instead of itself for dependency injection.

Although the Spring container uses the *FactoryBean*'s *getObject()* method's return value as the bean, you can also use the *FactoryBean* itself.

To access the *FactoryBean*, you just need to add a "&" before the bean name.

Let's try getting the factory bean and its *factoryId* property:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:factorybean-spring-ctx.xml" })
public class FactoryBeanXmlConfigTest {

    @Resource(name = "&tool")
    private ToolFactory toolFactory;

    @Test
    public void testConstructWorkerByXml() {
        assertThat(toolFactory.getFactoryId(), equalTo(9090));
    }
}
```

2.3. Use *FactoryBean* with Java-based Configuration

Use *FactoryBean* with Java-based configuration is a little different with XML-based configuration, you have to call the *FactoryBean*'s *getObject()* method explicitly.

Let's convert the example in the previous subsection into a Java-based configuration example:

```
@Configuration
public class FactoryBeanAppConfig {

    @Bean(name = "tool")
    public ToolFactory toolFactory() {
        ToolFactory factory = new ToolFactory();
        factory.setFactoryId(7070);
        factory.setToolId(2);
        return factory;
    }

    @Bean
    public Tool tool() throws Exception {
        return toolFactory().getObject();
    }
}
```

Then, we test if the *Tool* object is injected correctly:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = FactoryBeanAppConfig.class)
public class FactoryBeanJavaConfigTest {

    @Autowired
    private Tool tool;

    @Resource(name = "&tool")
    private ToolFactory toolFactory;

    @Test
    public void testConstructWorkerByJava() {
        assertThat(tool.getId(), equalTo(2));
        assertThat(toolFactory.getFactoryId(), equalTo(7070));
    }
}
```

The test result shows the similar effect as the previous XML-based configuration test.

3. Ways to Initialize

Sometimes you need to perform some operations after the *FactoryBean* has been set but before the *getObject()* method is called, like properties check.

You can achieve this by implementing the *InitializingBean* interface or using *@PostConstruct* annotation.

More details about using these two solutions have been introduced in another article: Guide To Running Logic on Startup in Spring (/running-setup-logic-on-startup-in-spring).

4. *AbstractFactoryBean*

Spring provides the *AbstractFactoryBean* as a simple template superclass for *FactoryBean* implementations. With this base class, we can now more conveniently implement a factory bean which creates a singleton or a prototype object.

Let's implement a *SingleToolFactory* and a *NonSingleToolFactory* to show how to use *AbstractFactoryBean* for both singleton and prototype type:

```java
public class SingleToolFactory extends AbstractFactoryBean<Tool> {

    private int factoryId;
    private int toolId;

    @Override
    public Class<?> getObjectType() {
        return Tool.class;
    }

    @Override
    protected Tool createInstance() throws Exception {
        return new Tool(toolId);
    }

    // standard setters and getters
}
```

And now the nonsingleton implementation:

```java
public class NonSingleToolFactory extends AbstractFactoryBean<Tool> {

    private int factoryId;
    private int toolId;

    public NonSingleToolFactory() {
        setSingleton(false);
    }

    @Override
    public Class<?> getObjectType() {
        return Tool.class;
    }

    @Override
    protected Tool createInstance() throws Exception {
        return new Tool(toolId);
    }

    // standard setters and getters
}
```

Also, the XML config for these factory beans:

```
<beans ...>

    <bean id="singleTool" class="com.baeldung.factorybean.SingleToolFactory">
        <property name="factoryId" value="3001"/>
        <property name="toolId" value="1"/>
    </bean>

    <bean id="nonSingleTool" class="com.baeldung.factorybean.NonSingleToolFactory">
        <property name="factoryId" value="3002"/>
        <property name="toolId" value="2"/>
    </bean>
</beans>
```

Now we can test if the *Worker* objects' properties are injected as we expect:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:factorybean-abstract-spring-ctx.xml" })
public class AbstractFactoryBeanTest {

    @Resource(name = "singleTool")
    private Tool tool1;

    @Resource(name = "singleTool")
    private Tool tool2;

    @Resource(name = "nonSingleTool")
    private Tool tool3;

    @Resource(name = "nonSingleTool")
    private Tool tool4;

    @Test
    public void testSingleToolFactory() {
        assertThat(tool1.getId(), equalTo(1));
        assertTrue(tool1 == tool2);
    }

    @Test
    public void testNonSingleToolFactory() {
        assertThat(tool3.getId(), equalTo(2));
        assertThat(tool4.getId(), equalTo(2));
        assertTrue(tool3 != tool4);
    }
}
```

As we can see from the tests, the *SingleToolFactory* produces singleton object, and the *NonSingleToolFactory* produces prototype object.

Note that there's no need to set singleton property in *SingleToolFactory* because, in *AbstractFactory*, singleton property's default value is *true*.

5. Conclusion

Using a *FactoryBean* can be a good practice to encapsulate complex construction logic or make configuring highly configurable objects easier in Spring.

So in this article, we introduced the basics of how to implement our *FactoryBean*, how to use it in both XML-based configuration and Java-based configuration, and some other miscellaneous aspects of *FactoryBean*, such as initialization of *FactoryBean* and *AbstractFactoryBean*.

As always, the complete source in this GitHub project (https://github.com/eugenp/tutorials/tree/master/spring-core/src/main/java/com/baeldung/factorybean).

Sort by: newest|oldest|most voted

Richard Adams

Guest

Hi, in section 2.3 you say 'you have to call the FactoryBean's getObject() method explicitly' but in your code you don't. Which is correct? This is a nice article but is a bit confusing here.

0

8 months 19 days ago

Grzegorz Piwowarek

Guest

Thanks for pointing that out. We were missing something from that config. You can check the whole class here: https://github.com/eugenp/tutorials/blob/9d7ad528b47491f680a68b917889aca1121b0c88/spring-core/src/main/java/com/baeldung/factorybean/FactoryBeanAppConfig.java (https://github.com/eugenp/tutorials/blob/9d7ad528b47491f680a68b917889aca1121b0c88/spring-core/src/main/java/com/baeldung/factorybean/FactoryBeanAppConfig.java)

We will update the article shortly.

0

8 months 19 days ago

CATEGORIES

SPRING (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

ABOUT

ABOUT BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
THE COURSES (HTTP://COURSES.BAELDUNG.COM)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
WRITE FOR BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
PRIVACY POLICY (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
TERMS OF SERVICE (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
CONTACT (HTTP://WWW.BAELDUNG.COM/CONTACT)
COMPANY INFO (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
ADVERTISE ON THE JAVA WEEKLY (HTTP://WWW.BAELDUNG.COM/JAVA-WEEKLY-SPONSORSHIP)
CONSULTING WORK (HTTP://WWW.BAELDUNG.COM/CONSULTING)