Spring JavaConfig

Reference Documentation

version 1.0-m2

2007.05.08

Rod Johnson, Costin Leau

Table of Contents

Chapter 1. Introduction

As mentioned in the IoC chapter [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html], at the core of Spring IoC is the *bean* concept which defines the way in which an object is instantiated, assembled and managed by the Spring container. XML is the most popular way of describing beans configuration, though Spring itself can read from virtually any type of metadata that can be translated into Java code. Annotations [http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html], available in JDK 5+, constitute such a type as they allow source code components to provide additional metadata which can affect the runtime semantics, making them a great configuration candidate.

Chapter 2. Components

Java Configuration uses annotations to leverage Java constructs allowing beans to be created and configured by the developer without leaving the Java world. In short, the developer will instantiate and configure the beans through Java code and then instruct the container to use them. Before moving forward, please note that the Spring semantics remain the same no matter how the configuration takes place: Java or XML.

Let's look at the most important annotations on which JavaConfig relies:

2.1. `@Configuration`

The `@Configuration` annotation indicates configuration classes:

```
@Configuration
public class WebConfiguration {
   // bean definitions follow
}
```

`@Configuration` is a class (type) level annotation and indicates the defaults for the bean definitions definied by the configuration:

```
@Configuration(defaultAutowire = Autowire.BY_TYPE, defaultLazy = Lazy.FALSE)
public class DataSourceConfiguration
     extends ConfigurationSupport {
}
```

It can be considered the equivalent of <beans/> tag. It is advisable that classes with `@Configuration` annotation extend the `ConfigurationSupport` as it offers several utility methods.

2.2. `@Bean`

As the name implies, `@Bean` indicates a bean definition (the <bean/> tag). Let's start with a simple example:

```
@Bean (scope = DefaultScopes.SESSION)
public ExampleBean exampleBean() {
  return new ExampleBean();
}
```

The code above instructed the Spring container to create a bean using the method name (as bean name) and return value (for the actual bean instance). The bean has session [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-scopes-session] scope, which means the `exampleBean()` method will be called to create a new bean instance per HTTP session.

Since pure Java is used, there is no need to use:

- factory-method [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-class-static-factory-method] when dealing with static methods:

```
@Bean
public ExampleBean exampleBean() {
  return ExampleFactory.createBean();
}
```

or

- FactoryBean
[http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-extension-factorybean]/MethodInv
[http://static.springframework.org/spring/docs/2.0.x/api/index.html] for complex object creation:

```
@Bean(aliases = { "anniversaries" })
public List<Date> birthdays() {
  List<Date> dates = new ArrayList<Date>();
  Calendar calendar = Calendar.getInstance();

  calendar.set(1977, 05, 28);
  dates.add(calendar.getTime());
  dates.add(computeMotherInLawBirthday());

  return dates;
}
```

`@Bean` is a method level annotation and indicates the Java code used for creating and configuring a bean instance. The annotation supports most of the options offered by an XML bean definition such as autowiring [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-autowire], lazy-init [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-lazy-init], dependency-check [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-dependencies], depends-on [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-dependson] and scoping [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-scopes]. Also, the lifecycle [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-nature] methods and *Aware interfaces are fully supported:

```
public class AwareBean implements BeanFactoryAware {

  private BeanFactory factory;

  // BeanFactoryAware setter
  public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
   this.factory = beanFactory;
  }

  public void close(){
    // do clean-up
  }
}
```

```
@Bean(destroyMethodName = "close", lazy = Lazy.TRUE)
public AwareBean createBeanFactoryAwareBean() {
   return new AwareBean();
}
```

Besides `destroyMethodName`, `@Bean` annotation supports also `initMethodName` though its usage is discourage as one already has control over the object creation and thus can call the initializing method if needed.

2.3. `@ExternalBean`

`@ExternalBean` is a simple markup annotation used for injecting 'external' beans, defined in a parent application context. Let's look at example:

```
@Configuration
public abstract class ExternalBeanConfiguration {
  @Bean
```

```
  public TestBean james() {
    TestBean james = new TestBean();
    // inject dependency from ann()
    james.setSpouse(ann());
    return james;
  }

  // Will be taken from the parent context
  @ExternalBean
  public abstract TestBean ann();
}
```

When JavaConfig encounter `@ExternalBean`, it will override the owning method so that anytime the method is being called, the parent application context will be looked for the bean under the method name (please see the naming chapter for more details). This way, your configuration remains pure Java and refactoring friendly.

Note that `@ExternalBean` works on normal method also; the example above uses the abstract method to avoid writing dummy code that doesn't execute:

```
@Configuration
public class ExternalBeanOnNormalMethod {

  @ExternalBean
  public TestBean ann(){
      System.out.println("this code will not execute as the method " +
          "will be overriden with a bean look up at runtime");
  }
}
```

2.4. `@ScopedProxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-scopes-other-injection](please see the link for an in-depth discussion on the matter). The easiest way to create such a proxy, when using the XML configuration, is the `<aop:scoped-proxy/>` element. JavaConfig offers as alternative the `@ScopedProxy` annotation which provides the same semantic and configuration options.

The reference documentation XML scoped proxy example, looks like this under JavaConfig:

```
// a HTTP Session-scoped bean exposed as a proxy
@Bean(scope = DefaultScopes.SESSION)
@ScopedProxy
public UserPreferences userPreferences() {
   return new UserPreferences();
}

@Bean
public Service userService() {
   UserService service = new SimpleUserService();
   // a reference to the proxied 'userPreferences' bean
   service.seUserPreferences(userPreferences());
   return service;
}
```

Chapter 3. Bean Visibility

A nice JavaConfig feature is bean visibility. JavaConfig uses a method visibilty modifiers to determine if the bean resulted from that method can be accessed through by owning application context / bean factory or not.

Consider the following configuration:

```
@Configuration
public abstract class VisibilityConfiguration {

  @Bean
  public Bean publicBean() {
     Bean bean = new Bean();
     bean.setDependency(hiddenBean());
     return bean;
  }

  @Bean
  protected HiddenBean hiddenBean() {
     return new Bean("protected bean");
  }

  @Bean
  private HiddenBean secretBean() {
     Bean bean = new Bean("private bean");
     // hidden beans can access beans defined in the 'owning' context
     bean.setDependency(outsideBean());
  }

  @ExternalBean
  public abstract Bean outsideBean()
}
```

used along side the following XML configuration (for more information on mixing configuration strategies see this chapter) :

```
<beans>
 <!-- the configuration above -->
 <bean class="my.java.config.VisibilityConfiguration"/>

 <!-- Java Configuration post processor -->
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

 <bean id="mainBean" class="my.company.Bean">
    <!-- this will work -->
    <property name="dependency" ref="publicBean"/>
    <!-- this will *not* work -->
    <property name="anotherDependency" ref="hiddenBean"/>
 </bean>
</beans>
```

One JavaConfig will encounter the configuration above, it will create 3 beans : `publicBean`, `hiddenBean` and `secretBean`. All of them can see each other however, beans created in the 'owning' application context (the application context that bootstraps JavaConfig) will see only `publicBean`. Both `hiddenBean` and `secretBean` can be accessed only by beans created inside `VisibilityConfiguration`.

Any `@Bean` annotated method, which is not `public` (i.e. with `protected`, `private` and `default` visibility), will create a `'hidden'` bean.

In the example above, `mainBean` has been configured with both `publicBean` and `hiddenBean`. However, since the latter is (as the name imply) hidden, at runtime Spring will throw:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'hiddenBean' is defined
 ...
```

To provide the visibility functionality, JavaConfig takes advantage of the application context hierarchy [http://static.springframework.org/spring/docs/2.0.x/reference/beans.html] provided by the Spring container, placing all hidden beans for a particular configuration class, inside a child application context Thus, the hidden beans can access beans defined in the parent (or owning) context but not the other way around.

Chapter 4. Wire dependencies

To assemble a bean, one simply has to use the constructs provided by Java:

```
@Bean(scope = DefaultScopes.SINGLETON)
public Person rod() {
  return new Person("Rod Johnson");
}

@Bean(scope = DefaultScopes.PROTOTYPE)
public Book book() {
  Book book = new Book("Expert One-on-One J2EE Design and Development");
  book.setAuthor(rod());  // rod() method is actually a bean reference !
  return book;
}
```

In the example above, the book author is using the return value of *rod* method. However, since both *book* and *rod* methods are marked with `@Bean`, the resulting beans, managed by Spring, will respect the container semantics: *rod* bean will be a singleton while *book* bean a prototype. When creating the configuration, Spring is aware of the annotation context and will replaces the rod() method invocation with a reference to the bean named 'rod'.

The container will return a new `Book` instance (prototype) each time *book* bean is request but will return the same instance (a singleton) for *rod* bean.

The code above is equivalent to:

```
<bean id="rod" class="Person" scope="singleton">
   <constructor-arg>Rod Johnson</constructor-arg>
</bean>

<bean id="book" class="Book" scope="prototype">
   <constructor-arg>Expert One-on-One J2EE Design and Development</constructor-arg>
   <property name="author" ref="rod"/>
</bean>
```

Note that while the examples above used two common scopes types, any type of scoping can be specified:

```
@Bean (scope = "customer")
public Bag shopingBag() {
  return new Basket();
}

@Bean (scope = "shift")
public Manager shopManager() {
  ...
}
```

Chapter 5. Naming strategy

In all the examples so far, the bean resulting from the method invocation, carried the method name:

```
@Configuration
public class ColoursConfiguration {
  // create a bean with name 'blue'
  @Bean
  public Color blue() {
    ...
  }
  ...
}
```

```
// dependency lookup for the blue colour
applicationContext.getBean("blue");
```

In some cases, this naming scheme is not suitable as methods with the same name, from different classes will override each other definitions. To customize the behavior, one can implement `BeanNamingStrategy` interface to provide its own naming generation strategy.

However, before writing your own code, take a look at the options provided by the default implementation: `MethodNameStrategy`.

```
 <!-- Java Configuration post processor -->
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor">
    <property name="namingStrategy">
       <bean class="org.springframework.config.java.naming.MethodNameStrategy">
          <property name="prefix" value="CLASS"/>
       </bean>
    </property>
 </bean>
```

With this configuration, the bean creation method enclosing class will be appended to the name:

```
// dependency lookup for the blue colour using the new naming scheme
applicationContext.getBean("ColoursConfiguration.blue");
```

Chapter 6. Mixing XML and annotations

Java and XML configuration are not exclusive - both can be used inside the same Spring application. In order to retrieve a bean from an XML file, one has to use the Spring container. As mentioned, one can achieve this with @ExternalBean annotation (the recommended way). For cases where this is not suitable or desired, the underlying beanFactory used for the `@Configuration` class can be access. Out of the box, this can be achieved by extending configuration classes from `ConfigurationSupport` or by implementing the `BeanFactoryAware` interface.

Consider the following XML configuration:

```
<bean id="myBean" class="MyBean"/>
```

In order to refer to `myBean` bean when using Java, one can use the following snippets:

```
@Configuration
public class MyConfig extends ConfigurationSupport {

  @Bean
  public ExampleBean anotherBean() {
     ExampleBean bean = new ExampleBean("anotherBean");
     bean.setDep(getBean("myBean")); // use utility method to get a hold of 'myBean'
     return bean;
  }
}
```

```
@Configuration
public class MyOtherConfig implements BeanFactoryAware {
  private BeanFactory beanFactory;

  public void setBeanFactory(BeanFactory beanFactory) {
     // get access to the owning bean factory
     this.beanFactory = beanFactory;
  }

  @Bean
  public ExampleBean yetAnotherBean() {
     ExampleBean bean = new ExampleBean("yetAnotherBean");
     bean.setDep(beanFactory.getBean("myBean")); // use dependency lookup
     return bean;
  }
}
```

Again, please consider twice before using `ConfigurationSupport` and/or `BeanFactoryAware` as `@ExternalBean` offers the same capability in a refactoring friendly manner.

JavaConfig distribution contains a converted Petclinic sample that replaces some XML configuration parts, with Java and Groovy [http://groovy.codehaus.org/] - please see the samples folder for more info.

Chapter 7. Using Java Configuration

To use annotations for configuring your application, one can use:

- `AnnotationApplicationContext`

  which accepts a Ant-style pattern of class names which will scanned for annotations:

```
ApplicationContext oneConfig =
       new AnnotationApplicationContext(SimpleConfiguration.class.getName());
ApplicationContext aBunchOfConfigs =
       new AnnotationApplicationContext("**/configuration/*Configuration.class");
```

  This specialized application context will automatically read and add as beans the classpath classes matching the given pattern. The downside of this approach is that no parameterization of the configuration instances can be made.

- `Configuration` post processor

```
<beans>
 <!-- Spring configuration -->
 <bean class="org.springframework.samples.petclinic.JdbcConfiguration"/>

 <!-- Java Configuration post processor -->
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>
</beans>
```

  This second approach allows more configuration options ,as it gives control not just over the configuration processing (through `ConfigurationPostProcessor`) but also over the configuration instance itself.

  By defining the configuration as a bean, the Spring container can be used for configuring the configuration (set properties or use a certain constructor):

```
<beans>

 <!-- a possible configurable configuration -->
 <bean class="org.my.company.config.AppConfiguration">
    <property name="env" value="TESTING"/>
    <property name="monitoring" value="true"/>
    <property name="certificates" value="classpath:/META-INF/config/MyCompany.certs"/>
 </bean>

 <!-- Java Configuration post processor -->
 <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

</beans>
```

Chapter 8. Roadmap

The project is relatively young and can be considered in beta (hence, the milestone release). Future development will be focused on automatic configuration discovery and simplifications.

Feedback, bugs and suggestions are welcomed at Spring forum [http://forum.springframework.org] and Spring issue tracking [http://opensource.atlassian.com/projects/spring/].