**Spring**

I want to develop one enterprise application with all required middle level services. What are the recommended technologies and framework for implementing all the services?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Services** | **Technology and Framework (Without Spring)** | **Introduced in Spring** | **To Integrate with**  **Spring** |
|  | Web Support | Servlet, JSP, Struts, JSF, Struts2, Flex | Spring MVC | All |
|  | Persistence Service | JDBC, Hibernate, JPA, Entity Bean (EJB) | X | All |
|  | Transaction Service | JDBC Transaction, Hibernate Transaction, JPA Transaction, EJB Container Transaction | Spring Transaction |  |
|  | Security Service | JAAS, Third Party | Spring Security | All |
|  | Remoting Service | RMI, EJB2/3, Web Service | X | All |
|  | Timer Service | JDK, EJB Container | √ | √ |
|  | Messaging Service | JMS, MDB | X | All |
|  | Resource Management | Web Container, EJB Container | Spring Container |  |
|  | Life Cycle Management | Web Container, EJB Container | Spring Container |  |
|  | Mailing Service | Java Mail API | X | All |
|  | Registry Service | RMI Registry, JNDI Registry, UDDI | X | All |
|  | Logging Service | Log4J | X |  |
|  | Unit Testing | JUnit, Mock Object, Easy Mock | √ | All |
|  | IOC | EJB3 Container | √ |  |
|  | AOP | EJB3 Container | √ |  |

**Inversion Of Control(IOC)**

IOC can be implemented in two ways.

1. Dependency Look Up
2. Dependency Injection
3. Dependency Look Up:

If you want any objects to use you can create those objects or you can look up and get it from registry. This is the normal flow of control.

1. Dependency Injection:

Dependency Injection mechanism injects the required object automatically this simplifies the programming.

Dependency Injection:

Container will instantiate the bean and give it (to the object) whenever it’s required, without waiting to be asked.

Means inject the dependency where it is required automatically by the container.

Dependency look up:

Whenever bean is required look up into the container and get the instantiated been.

Note: Now onward we’ll use the term Bean instead of instance or object or components.

The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

We can implement dependency Injection in two ways:

1. Setter Injection
2. Constructor Injection
3. Method injection (getter Injection)

Constructor injection configuration in “application.xml” file

**For primitive type and For String type**

<bean id = “bean\_name” class = “package\_name.class\_name”>

<constructor - arg>

<value> ….. </value>

</constructor – arg>

<constructor – arg value = “……” />

</bean>

**For reference of anther class**

<bean id = “hello” class = “com.app.Hello” >

<constructor-arg>

< ref local = “h”/>

</ constructor-arg>

</bean>

<bean id = “hello” class = “com.app.Hello” >

<constructor-arg ref = “h” />

</bean>

Setter injection configuration in “application.xml” file

**For Primitive and String**

<bean id = “bean\_name” class = “package\_name.class\_name”>

<property name = “property\_name”>

<value>…..<value>

</property>

<property name = “property\_name” value = “……” />

</bean>

**For Reference type**

<bean id = “hello” class = “com.app.Hello” >

<property name = “hai” ref = “h” />

</bean>

<bean id = “hello” class =“com.app.Hello” >

<property name = “hai”>

<ref local = “h” />

</property>

</bean>

<bean id = “hello” class = “com.app.Hello” p:hai-ref = “h” />

**Question:**

How to inject null value in Spring?

Answer:

In Spring, you can uses this special <null /> tag to pass a “**null**” value into constructor argument or property.

**Constructor Argument**

The wrong way to inject a null into constructor argument

<bean id="defaultMongoTypeMapper1"

**class**="org.springframework.data.mongodb.core.convert.DefaultMongoTypeMapper">

<constructor-arg name="typeKey" value="null"/>

</bean>

Correct way.

<bean id="defaultMongoTypeMapper"

**class**="org.springframework.data.mongodb.core.convert.DefaultMongoTypeMapper">

<constructor-arg name="typeKey">

<**null**/>

</constructor-arg>

</bean>

**Setter Injection**

The wrong way to inject null value into property.

<bean id="myConverter"

**class**="com.mkyong.convert.MoneyConverter">

<property name="typeMapper" value="null"/>

</bean>

Correct way.

<bean id="myConverter"

**class**="com.mkyong.convert.MoneyConverter">

<property name="typeMapper"><**null**/></property>

</bean>

Example of above two injections

1. Required jar files for this example:

commons-logging-1.1.3.jar

spring-beans-3.2.6.RELEASE.jar

spring-context-3.2.6.RELEASE.jar

spring-core-3.2.6.RELEASE.jar

spring-expression-3.2.6.RELEASE.jar

1. Xml beans definition in “applicationContext.xml”

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

</beans>

applicationContext.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<!-- <beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"> -->

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"a"*class=*"com.spring.core.A"*>

<propertyname=*"i"*value=*"100"*/>

<propertyname=*"f"*>

<value>345.56f</value>

</property>

<propertyname=*"str"*value=*"Inside A!"*/>

<propertyname=*"h"*ref=*"hello"*/>

</bean>

<beanid=*"b"*class=*"com.spring.core.B"*>

<constructor-arg>

<value>1234</value>

</constructor-arg>

<constructor-argvalue=*"345.677f"*/>

<constructor-argvalue=*"Inside B!"*/>

<constructor-argref=*"hello"*/>

</bean>

<beanid=*"hello"*class=*"com.spring.core.Hello"*/>

</beans>

**package** com.spring.core;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.AbstractApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** SpringTest1 {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

A aobj = (A) context.getBean("a");

System.*out*.println(aobj);

B bobj = (B) context.getBean("b");

System.*out*.println(bobj);

Hello h = (Hello) context.getBean("hello");

h.show();

((AbstractApplicationContext)context).registerShutdownHook();

}

}

/\*\*

\* **@author**surajkumar

\*

\* class with setter injection

\*/

**class** A{

**privateint**i;

**privatefloat**f;

**private** String str;

**private** Hello h;

**publicint** getI() {

**return**i;

}

**publicvoid** setI(**int** i) {

**this**.i = i;

}

**publicfloat** getF() {

**return**f;

}

**publicvoid** setF(**float** f) {

**this**.f = f;

}

**public** String getStr() {

**return**str;

}

**publicvoid** setStr(String str) {

**this**.str = str;

}

**public** Hello getH() {

**return**h;

}

**publicvoid** setH(Hello h) {

**this**.h = h;

}

@Override

**public** String toString() {

**return**"A [i=" + i + ", f=" + f + ", str=" + str + ", h=" + h + "]";

}

}

/\*\*

\* **@author**surajkumar

\*

\* class with constructor injection

\*/

**class** B{

**privateint**i;

**privatefloat**f;

**private** String str;

**private** Hello h;

**public** B(**int** i, **float** f, String str, Hello h) {

**super**();

**this**.i = i;

**this**.f = f;

**this**.str = str;

**this**.h = h;

}

@Override

**public** String toString() {

**return**"B [i=" + i + ", f=" + f + ", str=" + str + ", h=" + h + "]";

}

}

**class** Hello{

**publicvoid** show(){

System.*out*.println("Inside Hello !");

}

}

Output:

Apr 30, 2015 11:59:49 AM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@16fd0b7: startup date [Thu Apr 30 11:59:49 IST 2015]; root of context hierarchy

Apr 30, 2015 11:59:49 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [applicationContext.xml]

Apr 30, 2015 11:59:50 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons

INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@4839e5b5: defining beans [a,b,hello]; root of factory hierarchy

A [i=100, f=345.56, str=Inside A!, h=com.spring.core.Hello@276a38b5]

B [i=1234, f=345.677, str=Inside B!, h=com.spring.core.Hello@276a38b5]

Inside Hello !

**Question**: How should write the xml beans definition in “applicationContext.xml” file for different type of Spring application?

**Question:** Which jars are required for which type of spring application?

**Question:**How to start and stop the Spring container?

**Answer:**

**Start**

**Stop**

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. Of course, you must still configure and implement these destroy callbacks correctly.

1. To register a shutdown hook, you call the registerShutdownHook() method that is declared on the AbstractApplicationContext class:

AbstractApplicationContext ctx = **new** ClassPathXmlApplicationContext(**new** String []{"beans.xml"});

*// add a shutdown hook for the above context...*

ctx.registerShutdownHook();

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**new** String []{"beans.xml"});

((AbstractApplicationContext)ctx).registerShutdownHook();

2.

((ConfigurableApplicationContext)ctx).close()

**Question:** How many types of Spring container is available?

**Answer:**

Spring container comes into two distinct types:

1. Bean factories

(defined by the org.springframework.beans.factory.BeanFactory interface)

1. Application contexts

(defined by the org.springframework.context.ApplicationContext interface)

1. There are several implementations of BeanFactory in Spring. But the one that is most commonly used is org.springframework.beans.factory.xml.**XmlBeanFactory**

, which loads its beans based on the definitions contained in an XML file.

BeanFactory factory = new XmlBeanFactory(new FileSystemResource("c:/beans.xml"));

1. For ApplicationContext there are three that are commonly used:

* **ClassPathXmlApplicationContext** — Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
* **FileSystemXmlApplicationContext** — Loads a context definition from an XML file in the file system.
* **XmlWebApplicationContext** — Loads context definitions from an XML file contained within a web application

ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");

ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");

**Question:** If you have multiple spring configuration file in your application, How will you use all the configuration file in you application?

**Answer:**

We can use any one in following ways:

1. Using ApplicationContext subclass constructor with string array argument.

ApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[] {"Spring-Common.xml","Spring-Connection.xml","Spring-ModuleA.xml"});

1. Put all xml files under project classpath:

project-classpath**/**Spring-Common.xml

project-classpath**/**Spring-Connection.xml

project-classpath**/**Spring-ModuleA.xml

1. Best way is, to import all xml file inside another xml file, where it is required.

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"*>

<importresource=*"common/Spring-Common.xml"*/>

<importresource=*"connection/Spring-Connection.xml"*/>

<importresource=*"moduleA/Spring-ModuleA.xml"*/>

</beans>

Question: What is Method injection?

Reference:

<https://books.google.co.in/books?id=MoJyFa-6UbsC&pg=PA83&lpg=PA83&dq=getter+injection+in+spring&source=bl&ots=ApOeZJTbVy&sig=1h-hbmxInhZ7u-eXHAC-iSBUYq0&hl=en&sa=X&ved=0CFwQ6AEwCWoVChMInNa8iN-VxgIVBC2mCh3iWAA6#v=onepage&q=getter%20injection%20in%20spring&f=false>

Answer:

* Inject entire method definition into the bean.
* Inject method into java classes at runtime.
* Method injection is a type of injection where class method is replaced by an alternate implementation.
* Spring support two types of method injection

1. Method Replacement

Enable existing method (abstract or concrete) to be replaced at runtime with new implementation.

1. Getter Injection

Enable existing method (abstract or concrete) to be replaced at runtime with a new implementation that returns a specific bean from the Spring Context.

1. Method Replacement :

springContext.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"message"*class=*"com.spring.core.Message"*>

<propertyname=*"msg"*value=*"This is Static message!"*/>

<replaced-methodname=*"showMessage"*replacer=*"yahooReplacer"*/>

</bean>

<beanid=*"yahooReplacer"*class=*"com.spring.core.YahooReplacer"*/>

</beans>

Messgae.java

**package** com.spring.core;

**publicclass** Message {

**private** String msg;

**public** String getMsg() {

**return**msg;

}

**publicvoid** setMsg(String msg) {

**this**.msg = msg;

}

**public**String showMessage() {

**return** getMsg();

}

}

YahooReplcer.java

**package** com.spring.core;

**import** java.lang.reflect.Method;

**import** org.springframework.beans.factory.support.MethodReplacer;

**publicclass** YahooReplacer **implements** MethodReplacer {

@Override

**public** Object reimplement(Object arg0, Method arg1, Object[] arg2) **throws** Throwable {

**return**"This message is from Yahoo! Yahoo welcome you....";

}

}

TestMethodReplacement.java

**package** com.spring.core;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** TestMethodReplacement {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("springContext.xml");

Message mb = (Message)context.getBean("message");

System.*out*.println(mb.showMessage());

}

}

Output:

This message is from Yahoo! Yahoo welcome you....

Steps:

1. Add <**replaced-methodname**=”” **replacer** = “” /> in the bean definition whose method you want to replace.

* Provide the **name** as method name and **replacer** the new implementation which you want to replace.

<beanid=*"message"*class=*"com.spring.core.Message"*>

<replaced-methodname=*"showMessage"*replacer=*"yahooReplacer"*/>

</bean>

1. Implement the Replacer class by implementing the MethodReplacer interface (org.springframework.beans.factory.support.MethodReplacer).

* Define the bean as newly cleated replacer class.

<beanid=*"yahooReplacer"*class=*"com.spring.core.YahooReplacer"*/>

1. Create replace class by implementing **MehtodReplacer** interface and overriding **reimplement()** method

**publicclass** YahooReplacer **implements** MethodReplacer {

**public** Object reimplement(Object arg0, Method arg1, Object[] arg2) **throws** Throwable {

**return**"This message is from Yahoo! Yahoo welcome you....";

}

}

* This **reimplements()** method body will be invoked whenever we call the **showMessage()** method over **message** bean

**Getter Injection:**

* This type of injection inject a bean which is already defined in the configuration file as the return type of a method.
* **<lookup-method>** replaces a method with new implementation at runtime.

<lookup-methodname=*"getMessage"*bean=*"message"*/>

* **name** attribute is the which contains method name and **bean** contains the bean id which needs to inject.
* This inject the bean directly to the method as it return type.
* The bean should be defined in the spring configuration file.

Getter Injection.java

**package** com.spring.core;

**publicabstractclass** GetterInjection {

**public** String getMessageString() {

**return**"Hi String message from class";

}

**abstractpublic** Message getMessage();

}

springContext.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"message"*class=*"com.spring.core.Message"*>

<propertyname=*"msg"*value=*"This is Static message!"*/>

<replaced-methodname=*"showMessage"*replacer=*"yahooReplacer"*/>

</bean>

<beanid=*"yahooReplacer"*class=*"com.spring.core.YahooReplacer"*/>

<beanid=*"getterI"*class=*"com.spring.core.GetterInjection"*>

<lookup-methodname=*"getMessageString"*bean=*"msgString"*/>

<lookup-methodname=*"getMessage"*bean=*"message"*/>

</bean>

<beanid=*"msgString"*class=*"java.lang.String"*>

<constructor-arg><value>This is a getter String Message</value></constructor-arg>

</bean>

</beans>

TestMethodReplacement.java

**package** com.spring.core;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** TestMethodReplacement {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("springContext.xml");

Message mb = (Message)context.getBean("message");

//System.out.println(mb.showMessage());

System.*out*.println(mb);

GetterInjection gi = (GetterInjection)context.getBean("getterI");

System.*out*.println(gi.getMessageString());

System.*out*.println(gi.getMessage());

}

}

Output:

Jun 23, 2015 4:13:34 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@5d764be1: startup date [Tue Jun 23 16:13:34 IST 2015]; root of context hierarchy

Jun 23, 2015 4:13:34 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [springContext.xml]

Jun 23, 2015 4:13:34 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons

INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@3d3e58d4: defining beans [message,yahooReplacer,getterI,msgString]; root of factory hierarchy

com.spring.core.Message$$EnhancerByCGLIB$$cbe5cd13@77ff92f5

This is a getter String Message

com.spring.core.Message$$EnhancerByCGLIB$$cbe5cd13@77ff92f5

**Inner Beans**

You can define one bean definition in other bean. This type of configuration is called as inner beans and cannot be reused.

Following is the sample xml:

Spring Core Container has implemented based on two popular GOF design pattern.

1. Factory Pattern
2. Singleton Pattern

**Bean Scope**

Bean instance created by the Spring Container can be available in 5 Scopes.

1. Singleton
2. Prototype
3. Request
4. Session
5. Global-session
6. Singleton:

When bean scope is Singleton than only one instance will be created for that bean and the same will be used to inject in other beans and same bean instance will be return whenever you willcall “getBean()” method.

By default all beans are singleton beans.

1. Prototype:

When bean scope is prototype then a new instance will be created every time whenever you call “getBean()” method.

Following is the way to specify the prototype scope.

<bean id = “x” class = “….” scope = “prototype” />

1. Request:
2. Session:

These two are equivalent to **HttpSerevletRequest** and **HttpSession** respectively in the case of Web base application. These two scopes must be used for web application only.

1. Goblal-session:

This scope is equivalent to HttpSession in case of Portal base Web Application.

Eample2:

Class definition from Example1

applicationContext.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<!-- <beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"> -->

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"a"*class=*"com.spring.core.A"*>

<propertyname=*"i"*value=*"100"*/>

<propertyname=*"f"*>

<value>345.56f</value>

</property>

<propertyname=*"str"*value=*"Inside A!"*/>

<propertyname=*"h"*ref=*"hello"*/>

</bean>

<beanid=*"b"*class=*"com.spring.core.B"*>

<constructor-arg>

<value>1234</value>

</constructor-arg>

<constructor-argvalue=*"345.677f"*/>

<constructor-argvalue=*"Inside B!"*/>

<constructor-argref=*"hello"*/>

</bean>

<beanid=*"hello"*class=*"com.spring.core.Hello"*scope=*"prototype"*/>

</beans>

Output:

Apr 30, 2015 12:15:54 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@53ef9f1d: startup date [Thu Apr 30 12:15:54 IST 2015]; root of context hierarchy

Apr 30, 2015 12:15:54 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [applicationContext.xml]

Apr 30, 2015 12:15:54 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons

INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@5117f31e: defining beans [a,b,hello]; root of factory hierarchy

A [i=100, f=345.56, str=Inside A!, h=com.spring.core.Hello@2eb1074a]

B [i=1234, f=345.677, str=Inside B!, h=com.spring.core.Hello@6855a338]

Inside Hello !

**Bean Loading Styles**

Spring container loads the bean in two different styles.

1. Eager Loading or Aggressive Loading
2. Lazy Loading

* In case of Eager Loading, beans will be created by the Spring Container at container start up.
* In case of Lazy Loading, beans will be created by the Spring Container, when you will use it first time.
* The default loading approach is Aggressive Loading.
* To specify the Lazy Loading you can use “lazy-init” attribute as follows:

<bean id=”x” class = “…….” lazy-init = “true” />

**Injecting various type of values through DI (Dependency Injection)**

Using DI you can inject the following types:

1. Primitives
2. String
3. Wrappers
4. Collection (Map, Set, List, Properties)
5. Other Beans

**publicclass** LockerConstants {

**privatestatic** Integer *paletteSize*;

**privatestatic** String *ccPush*;

**privatestatic** String *mnicTerminationPush*;

**privatestatic** Map<Class<? **extends** CsdXml>, String>*csdXmlAbbrevMap*;

**privatestatic** List<Integer>*defaultRoleIds*;

**privatestatic** Map<String, String>*tables*;

**privatestatic** String *nicNumberRegex*;

**privatestatic** DateFormat *dateConverterDateFormat*;

**publicstatic** Integer getPaletteSize() {

**return***paletteSize*;

}

**publicstaticvoid** setPaletteSize(**final** Integer paletteSize) {

ScreenConstants.*paletteSize* = paletteSize;

}

**publicstatic** String getCcPush() {

**return***ccPush*;

}

**publicstaticvoid** setCcPush(**final** String ccPush) {

LockerConstants.*ccPush* = ccPush;

}

**publicstatic** String getMnicTerminationPush() {

**return***mnicTerminationPush*;

}

**publicstaticvoid** setMnicTerminationPush(**final** String mnicTerminationPush) {

LockerConstants.*mnicTerminationPush* = mnicTerminationPush;

}

**publicstatic** Map<Class<? **extends** CsdXml>, String> getCsdXmlAbbrevMap() {

**return***csdXmlAbbrevMap*;

}

**publicstaticvoid** setCsdXmlAbbrevMap(**final** Map<Class<? **extends** CsdXml>, String> csdXmlAbbrevMap) {

EventTypeConstants.*csdXmlAbbrevMap* = csdXmlAbbrevMap;

}

**publicstatic** List<Integer> getDefaultRoleIds() {

**return***defaultRoleIds*;

}

**publicstaticvoid** setDefaultRoleIds(**final** List<Integer> defaultRoleIds) {

ScreenConstants.*defaultRoleIds* = defaultRoleIds;

}

**publicstatic** Map<String, String> getTables() {

**return** CodeTableConstants.*tables*;

}

**publicstaticvoid** setTables(**final** Map<String, String> tables) {

CodeTableConstants.*tables* = tables;

}

**publicstatic** String getNicNumberRegex() {

**return***nicNumberRegex*;

}

**publicstaticvoid** setNicNumberRegex(**final** String nicNumberRegex) {

CsdHandlerConstants.*nicNumberRegex* = nicNumberRegex;

}

**publicstatic** DateFormat getDateConverterDateFormat() {

**return***dateConverterDateFormat*;

}

**publicstaticvoid** setDateConverterDateFormat(**final** DateFormat dateConverterDateFormat) {

ScreenConstants.*dateConverterDateFormat* = dateConverterDateFormat;

}

}

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*xmlns:jee=*"http://www.springframework.org/schema/jee"*

xmlns:util=*"http://www.springframework.org/schema/util"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd*

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"*>

<beanclass=*"com.crimsonlogic.mnis.cpdv2.constants.LockerConstants"*>

<propertyname=*"paletteSize"*value=*"5"*/>

<propertyname=*"ccPush"*value=*"CC\_PUSH"*/>

<propertyname=*"mnicTerminationPush"*value=*"MNIC\_TERMINATE\_PUSH"*/>

<propertyname=*"csdXmlAbbrevMap"*>

<util:mapkey-type=*"java.lang.Class"*>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.Births"*value=*"BTH"*/>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.BthDets"*value=*"BDT"*/>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.Deaths"*value=*"DTH"*/>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.Divorce"*value=*"DIV"*/>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.Marriages"*value=*"MAR"*/>

<entrykey=*"com.crimsonlogic.mnis.cpdv2.model.csd.ForeignReg"*value=*"FOR"*/>

</util:map>

</property>

<propertyname=*"defaultRoleIds"*>

<util:listvalue-type=*"java.lang.Integer"*>

<value>1</value>

<value>2</value>

<value>3</value>

<value>4</value>

</util:list>

</property>

<propertyname=*"tables"*>

<util:mapkey-type=*"java.lang.String"*value-type=*"java.lang.String"*>

<entrykey=*"COUNTRY"*value=*"Country"*/>

<entrykey=*"DEATH\_TYPE"*value=*"Death Type"*/>

<entrykey=*"DISTRICT"*value=*"District"*/>

<entrykey=*"MARITAL\_STATUS"*value=*"Marital Status"*/>

<entrykey=*"SEX"*value=*"Sex"*/>

</util:map>

</property>

<propertyname=*"nicNumberRegex"*value=*"^[A-Z](?:\d{12})[A-G0-9]$"*/>

<propertyname=*"dateConverterDateFormat"*>

<beanclass=*"java.text.SimpleDateFormat"*>

<constructor-argindex=*"0"*value=*"dd${date.separator}MM${date.separator}yyyy"*/>

<propertyname=*"lenient"*value=*"false"*/>

</bean>

</property>

</bean>

</beans>

**For Array Type**

in code

String[] types;

public void setTypes(String[] types) {

this.types = types;

}

class Test{

public Test(String[]values){

}

}

in spring config

<property name="types">

<list>

<value>a</value>

<value>b</value>

</list>

</property>

Or,

<property name="types" value="hugo,emil"></property>

<bean id = “test” class = “Test”>

<constructor-arg>

<list>

<value>a</value>

<value>b</value>

</list>

</constructor-arg>

</bean>

**Using p namespace:**

* With the p namespace you can reduce the tag in the spring-context.xml.
* To use the p namespace add the below namespace.

xmlns:p = “http:www.springframework.org/schema/p”

* p namespace is used only for setter injection not for constructor injection.

Example: name of the field

<bean id = “hello” class = “com.app.spring.ioc.Hello” p:a = “99” p:str = “hello guys” p:obj-ref = “ao” />

**Wiring**

* Wiring is the process of injecting the dependencies into the required bean.
* You can perform wiring in two ways:

1. Explicit Wiring
2. Implicit Wiring or Autowiring
3. Explicit Wiring:

In the case of explicit wiring you need to configure the dependencies of the bean explicitly inside “**spring-context.xml**”

|  |  |  |
| --- | --- | --- |
| class Hai {  ……  ……  } | class Hello {  Hai hai;  //setters & getters  } | class Hello {  Hai hai;  Hello(Hai hai){  this.hai = hai;  }  } |

<bean id = “h” class = “com.app.Hai” />

<bean id = “hello” class = “com.app.Hello” >

|  |
| --- |
| <property name = “hai” ref = “h” /> |

</bean>

|  |
| --- |
| <bean id = “hello” class =“com.app.Hello” >  <property name = “hai”>  <ref local = “h” />  </property>  </bean> |

|  |
| --- |
| <bean id = “hello” class = “com.app.Hello” p:hai-ref = “h” /> |

|  |
| --- |
| <bean id = “hello” class =“com.app.Hello” >  <constructor-arg>  < ref local = “h”/>  </ constructor-arg>  </bean> |

|  |
| --- |
| <bean id = “hello” class =“com.app.Hello” >  <constructor-arg ref = “h” />  </bean> |

Configuring the dependency of a bean inside another bean

- Using setter mechanism

- Using constructor

1. Implicit Wiring or Autowiring:
   * + - In case of autowiring Spring Container detects the bean dependencies and injects them automatically.
       - To provide autowiring you have to use “autowire” attribute of bean tag.
       - Following are the possible values of autowire attributes:
2. none
3. byName
4. byType
5. constructor
6. autodetect

|  |  |
| --- | --- |
| Class Hai{  -------  ------  } | Class Hello{  Hai hai;  ------  -----  } |

1. none
   * + when autowire attribute value is “none” then container won’t do the wiring automatically. This is the default value.
2. byName
   * + need to write the setter methods.
     + When autowire attribute value is “byName” then spring container will try to detect a bean id whose name is same as property name(variable name).
     + If matching bean is found then that will be injected by setter injection.
     + If matching bean is not found then property remains uninjected.
     + Bean name will be same as bean id.

|  |  |  |
| --- | --- | --- |
| class Hai{  -------  ------  } | class A{  ------  ------  } | class Hello{  Hai hai;  A aboj;  ------  -----  } |

<bean id = “hai1” class = “com.app.Hai” />

<bean id = “aboj” class = “com.app.A” />

<bean id = “hello” class = “com.app.Hello” autowire = “byName”/>

In the above example Hello bean has two dependencies:

1. Hai bean
2. A bean

But because of “byName” autowiring process only “A” bean reference will be injected. “Hai” bean reference will have null only because no matching bean found.

1. byType
   * + Need to write setter methods
     + When autowire attribute value is byType then container will try to detect the beans whose type is same as property type.

<bean id = “hello” class = “com.app.Hello” autowire = “byType” />

In above example both dependencies of Hello bean will be injected.

* + - When container finds zero matching beans for the property type then property remains uninjected.
    - When container finds exactly one matching bean then that will be injected by setter injection.
    - When container finds two or more matching beans then exception will be thrown.

**UnsatisfiedDependencyException**

1. Constructor
   * + When autowire attribute value is “constructor” then container will try to detect beans which are matching with arguments of the constructor.
     + If no matching beans found than properties remains uninjected.
     + If matching beans founds than properties will be injected through constructor injection.

If multiple matching beans found then exception will be thrown called “**UnsatisfiedDependencyException**”.

|  |  |  |
| --- | --- | --- |
| class Hai{  -------  ------  } | class A{  ------  ------  } | class Hello{  Hai hai;  A aboj;  Hello(Hai hai, A aboj){  this.hai = hai;  this.aboj = aboj;  }  ------  -----  } |

<bean id = “hai1” class = “com.app.Hai” />

<bean id = “aboj” class = “com.app.A” />

<bean id = “hello” class = “com.app.Hello” autowire = “constructor”/>

1. Autodetect
   * + When autowire attribute value is “autodetect” then container will try to resolve the autowiring process first with “constructor” autowiring then with “byType”.

**Autowiring through annotation**

* From spring 2.5 onwards following annotations have been provided to inject the bean dependencies automatically.

1. @Autowired
2. @Resource
3. @Required

To use the above annotation in your bean you need to do the following

1. Enable the context namespace
2. Write the following tag in the “spring-context.xml”.

**<context: annotation-config />**

* Injection will happen through setter injection not from the constructor injection.

1. @Autowired
   * + When a bean property is marked with @Autowired annotation then that property will be injected through “byType” autowiring process.
     + No need to write setter methods.
2. @Resource
   * + To use this annotation you must have “javaee.jar” file.
     + No need to write setter methods.
     + @Resource annotation allows you to inject the properties in two ways.
3. byName autowiring
4. byType autowiring

* byType

@Resource

Hai hai;

* byName

@Resource(name = “hai1”)

Hai hai2;

@Required

When a method is mearked with @Required annotation then that method will be called by container automatically.

Summary:

1. Using xml based explicit wiring
2. Using xml based auto wiring
3. Using annotation based autowiring (as of Spring 2.5)

**Spring Container Callbacks (Life Cycle methods)**

Spring container provides following container callbacks.

1. Initialization callbacks
2. Destruction callbacks
3. Knowing who you are
4. Knowing where you are
5. Extending Spring Container functionality
6. Initialization callbacks:
   * + When you are developing the bean you may need to initialize the bean with the required resources.
     + You can do this using wiring process or dependency injection.
     + You can also initialized the resource using various callbacks methods provided in the bean life.
     + There are three ways to initialize the Bean Resource explicitly
7. Using @PostConstruct annotation
8. Using “afterPropertiesSet()” of “InitializingBean” interface.
9. Using custom “init()” method.
10. Destruction callbacks:
    * + When Spring container shut down, it will try to destroy all the bean instance running inside the container.
      + You may need to release some resources associated with the bean at container shout down time. You can do this in three ways:
11. Using @PreDestroy annotation.
12. Using “destroy()” method of “DisposableBean” interface.
13. Using custom destroy method.

Example:

applicationContext.xml

<context:annotation-config />

<bean id = “hello” class = “com.app.Hello” init-method = “init” destroy-method = “cleanup” />

Hello.java

package com.app;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import org.springframeowrk.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializatingBean;

public class Hello implements InitializingBean,DisposableBean{

@PostConstruct

public void myInit(){

System.out.println(“myInit()”);

}

public void init(){

System.out.println(“init()”);

}

public void afterProperties(){

System.out.println(“afterProperties()”);

}

@PreDestroy

public void myDestoy(){

System.out.println(“myDestory()”);

}

public void destroy(){

System.out.println(“destroy()”);

}

public void cleanup(){

System.out.println(“cleanup()”);

}

}

**Test.java**

package com.app;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframewrok.context.support.ClassPathXmlApplicationContext;

public class Test{

public static void main(String [] args){

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(“applicationContext.xml”);

System.out.println(“Spring Container ready….”);

ctx.registerShutDownHook();

System.out.println(“spring Container down…..”);

}

}

1. **Knowing who you are**
   * + If bean wants to know its name then bean has to implement “BeanNameAware” interface and has to override the following method.

public void setBeanName(String Name)

* + - Bean name is same as the Bean ID
    - If Bean class is implementing “BeanNameAware” interface then container call “setBeanName(String name)” method automatically in Bean life.

1. **Knowing where you are**
   * + If bean wants to get the reference to the Spring Container where it is running then bean class can implement one of the following two interfaces.
2. BeanFactoryAware
3. ApplciationContextAware
   * + When Bean class is implementing “BeanFactroyAware” interface then container calls the following method automatically.

public void setBeanFactory(BeanFactory bf)

* + - When your bean class is implementing ApplicationContextAware interface then container calls the following method automatically

public void setApplicationContext(ApplicationContext ctx)

1. **Extending Spring Container Functionality**
   * + If you want to extend the container functionalities you have to develop and register “BeanPostProcessor” interface.
     + Steps to write custom BeanPostProcessor
2. Write your bean class by implementing BeanPostProcessor interface.
3. Override the following two methods available in BeanPostProcessor Interface.

* Object postProcessorAfterInitialization(Object bean, String beanName)
* Object postProcessorBeforeInitialization(Object bean, String beanName)

1. Registerthe BeanPostProcessor class with Spring Container

**Lifecycle of a Bean in the ApplicationContext Container**

When Spring Container is creating Bean instance then container will do the following tasks:

1. Container loads the Bean class into main memory.
2. Container creates the Bean instance with appropriate constructor as per the constructor injection. If no constructor injection is defined then default constructor will be called.
3. Container will try to inject the Bean dependencies using the following ways in the same order:
4. Properties marked with @Autowired and @Resource annotation will inject.
5. Properties which are matching with xml based based autowiring will inject.
6. Properties which are matching with explicit wiring will inject.
7. If bean class is implementing BeanNameAware Interface then container calls setBeanName() method.
8. If bean class is implementing BeanFactoryAware interface then container calls setBeanFactory() method.
9. If bean class is implementing ApplciationContextAware interface then container calls setApplicationContext() method.
10. If any method of the bean is marked with @PostConstruct annotation then that method will be called by the container.
11. If any BeanPostProcessor is registered with the container then container calls

postProcessorBeforeInitialization()

1. If bean class is implementing InitializingBean interface then container calls

afterPropertiesSet()

1. If any custom init() method is specified in the spring-context.xml using “init-method” attribute then that method will be called by the container.
2. If any BeanPostProcessor is registered in the container then container calls

postProcessAfterInitialization()

1. Now Bean instance is ready to use.

**When Spring Container is destroying the bean instance, container will do the following task:**

1. If any method of the bean is marked with @PreDestroy annotation then that method will be called by the container.
2. If bean class is implementing DisposableBean Interface then container calls “destroy()” method.
3. If any custom destroy method is specified using “destroy-method” attribute then that method will be called by the container.

**BeanFactoryContainer**

* **BeanFactory** is the basic spring container which is responsible to create and manage bean instances.
* You can create the BeanFactory container as follows:

1. Resource rs = newClassPathResource(“spring-context.xml”);

BeanFactory factory = new XmlBeanFactory(rs);

Hello h = (Hello) factory.getBean(“hello”);

1. Using File System resource:

Resource rs = new FileSystemResouce(“J:/myworkspace/SpringExample/src/spring-context.xml”);

**Lifecycle of a Bean in the BeanFactory container**

* When BeanFactory container creating a bean instance then container will do following tasks:

1. Container loads the Bean class into main memory
2. Container creates the Bean instance with appropriate constructor as per constructor injection.
3. Container will try to inject the Bean dependencies using following two ways:
4. Using XML based autowiring
5. Using explicit wiring
6. If Bean class is implementing BeanNameAware interface then container call setBeanName() method.
7. If Bean class is implementing BeanFactoryAware interface then container calls “setBeanFactory()” method.
8. If Bean class is implementing initializing Bean interface then container calls “afterPropertiesSet()” method.
9. If any custom init method is specified using “init-method” attribute then that will be called by the container.
10. Bean is ready to use.

**When Bean factory container is destroying the bean instance it will do the following task.**

1. If bean class is implementing DiposableBean interface then container call “destroy()” method.
2. If any custom destroy method is specified using “destroy-method” attribute then that method will be called by the container.

**Bean Definition Inheritance**

1. When you are configuring multiple beans inside the spring-context.xml you may find some common properties among the beans with the same values.
2. Configuring same properties repeatedly for multiple beans increases the length of the xml document and decreases readability.

|  |  |
| --- | --- |
| Class Hello{  int x;  int y;  String str;  } | Class Hai{  int x;  int a;  String str;  } |

Normal bean definition:

<bean id = “hello” class = “com.app.Hello”>

<property name = “x” value = “99” />

<property name = “y” value = “88” />

<property name = “str” value = “aaaa” />

</bean>

<bean id = “hai” class = “com.app.Hai”>

<property name = “x” value = “99” />

<property name = “a” value = “111” />

<property name = “str” value = “aaa” />

</bean>

**Inheritance Bean Definition:**

<bean id = “base” abstract = “true”>

<property name = “x” value = “99” />

<property name = “str” value = “aaa” />

<bean>

<bean id = “hello” class = “com.app.Hello” parent = “base”>

<property name = “x” value = “0” />

<property name = “y” value = “88” />

</bean>

<bean id = “hai” class = “com.app.Hai” parent = “base”>

<property name = “a” value = “111” />

</bean>

**Externalization Bean Properties**

1. Write the required number of properties fields and place them under src folder of the project.

For eg:

message1.properties

msg.x=99

msg.y=88

message2.properties

msg.str=aaa

1. Register the property place holder configure with spring container as follows:

<beanid=*"x"* class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<propertyname=*"locations"*>

<list>

<value>message1.properties</value>

<value>message2.properties</value>

</list>

</property>

</bean>

1. Select the bean to externalize its properties.

|  |
| --- |
| Class Hello{  int x;  int y;  String str;  } |

1. Access the required keys from the properties files as follows:

<bean id = “hello” class = “com.app.Hello”>

<property name = “x” value = “${msg.x}” />

<property name = “y” value = “${msg.y}” />

<property name = “str” value = “${msg.str}” />

</bean>

Note:

1. properties files contain name and value pair as “name=value”
2. name in all properties files should be unique.
3. If name is not unique then last appearance of the name will override the previous value. The order of the properties file will be decide as defined in the spring configuration file. (for eg. – defined in step 2)

Example 3.

spring-context.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"msg"*

class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<propertyname=*"locations"*>

<list>

<value>message1.properties</value>

<value>message2.properties</value>

</list>

</property>

</bean>

<beanid=*"ab"*class=*"com.spring.core.AB"*>

<propertyname=*"xab"*value=*"${message1.x}"*/>

<propertyname=*"yab"*value=*"${message1.y}"*/>

<propertyname=*"iab"*value=*"${message1.i}"*/>

<propertyname=*"jab"*value=*"${j1}"*/>

</bean>

<beanid=*"xy"*class=*"com.spring.core.XY"*>

<constructor-argvalue=*"${message2.x}"*/>

<constructor-argvalue=*"${message2.y}"*/>

<constructor-argvalue=*"${message2.i}"*/>

<constructor-argvalue=*"${j}"*/>

</bean>

</beans>

message1.properties

message1.x=insidemessage1x

message1.y=insidemessage1y

message1.i=545

j1=666

message2.properties

message2.x=insidemessage2x

message2.y=insidemessage2y

message2.i=54555

j=99999

SpringTest2.java

**package** com.spring.core;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.AbstractApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** SpringTest2 {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring\_context.xml");

AB ab = (AB) context.getBean("ab");

XY xy = (XY) context.getBean("xy");

System.*out*.println(ab);

System.*out*.println(xy);

((AbstractApplicationContext)context).registerShutdownHook();

}

}

**class** AB {

**private** String xab;

**private** String yab;

**privateint**iab;

**privateint**jab;

**public** String getXab() {

**return**xab;

}

**publicvoid** setXab(String xab) {

**this**.xab = xab;

}

**public** String getYab() {

**return**yab;

}

**publicvoid** setYab(String yab) {

**this**.yab = yab;

}

**publicint** getIab() {

**return**iab;

}

**publicvoid** setIab(**int** iab) {

**this**.iab = iab;

}

**publicint** getJab() {

**return**jab;

}

**publicvoid** setJab(**int** jab) {

**this**.jab = jab;

}

@Override

**public** String toString() {

**return**"AB [xab=" + xab + ", yab=" + yab + ", iab=" + iab + ", jab="

+ jab + "]";

}

}

**class** XY {

**private** String xxy;

**private** String yxy;

**privateint**ixy;

**privateint**jxy;

**public** XY(String xxy, String yxy, **int** ixy, **int** jxy) {

**super**();

**this**.xxy = xxy;

**this**.yxy = yxy;

**this**.ixy = ixy;

**this**.jxy = jxy;

}

@Override

**public** String toString() {

**return**"XY [xxy=" + xxy + ", yxy=" + yxy + ", ixy=" + ixy + ", jxy="

+ jxy + "]";

}

}

**Accessing properties from Message Bundles:**

1. When you are developing any web based application you may get a requirement to implement internationalization (i18).
2. When you are supporting multiple languages for your application you need to write multiple message bundles with some name combination.

For eg:-

messages.properties

messages\_hi.properties

messages\_en.properties

1. If you want to get the value of required key from the message bundle you need to do the following:
   1. Write the required one or more message bundles and placed them under src folder.
   2. Register the “ResourceBundleMessageSource” Interface.

<bean id = “messageSource” class = “org.springframework.context.support.ResourceBundleMessageSource”>

<property name = “baseName”>

<value>messages</value>  
 </property>

</bean>

* 1. Access the values of the required keys using the methods of MessageSource interface through application context object.

ctx.getMessage(“login.un.required”, null, null);

ctx.getMessage(“login.em.required”, null, “Email is Required.”,null);

ctx.getMessage(“app.length”, new Object[]{“username”,”5”,10}, null);

**Properties Editors:**

1. Property editors are responsible to edit the property of the bean to convert from one format to another format.

For eg:-

When you are configuring the bean properties inside spring-context.xml you are supplying the value in similar way for primitives, wrappers and String. But these values will be converted to corresponding types.

1. Spring provides multiple built in property editors, which will be called by the spring container automatically whenever it is required.
2. If required you can write your own custom property editors as follows:
   1. Write your own property editor class by extending “PropertyEditorSupport” class which is available in “java.bean” package.
   2. Override the following method.

public void setAsText(String text);

* 1. Write the required conversion logic inside “setAsText(String)” method.
  2. After developing the required number of editors. Register all the editors with the spring container.

<bean class = “org.springframework.bean.factory.config.CustomEditorConfigurer”>

<property name = ”customEditors”>

<map>

<entry key = “com.app.spring.ioc.Fee” />

<entry key = ”com.app.spring.ioc.FeeEditors” />

</map>

</property>

**Publishing Application Events:**

Steps to write application event

1. Write your custom event class by extending “ApplicationEvent” class.
2. Declare the variable to store the information related to your event.
3. Write the two augmented constructor, one with the source of an event, second with information you have to pass along with event.

**Steps to write Application Listener**

1. Write your own listener class by implementing ApplicationListener interface.
2. Override the following method:

public void onApplicationEvent() of ApplicationEvent.

1. Implement the required logic after receiving the event inside “onApplicationEven()” method.
2. Register your application listener with the spring container.

**Spring with annotation (Without XML)**

## @Bean

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports most of the attributes offered by <bean/>, such as:

[init-method](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-lifecycle-initializingbean), [destroy-method](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-lifecycle-disposablebean), [autowiring](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-autowire), [lazy-init](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-lazy-init), [dependency-check](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-dependencies), [depends-on](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-dependson) and [scope](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-scopes).

**Declaring a bean**

To declare a bean, simply annotate a method with the @Bean annotation.

When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory. By default, the bean name will be the same as the method name (see [bean naming](http://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch02s02.html) for details on how to customize this behavior).

The following is a simple example of a @Bean method declaration:

@Configuration

**public** **class** AppConfig {

@Bean

**public** customerService CustomerService() {

**return** **new** CustomerServiceImpl();

}

}

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

<beans>

<bean name=*"customerService"* class=*"com.test.CustomerServiceImpl"*/>

</beans>

Both will result in a bean named **customerService** being available in the BeanFactory / ApplicationContext, bound to an object instance of type CustomerServiceImpl:

*customerService -> com.test.CustomerServiceImpl*

### **Injecting dependencies**

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

@Configuration

**public** **class** AppConfig {

@Bean

**public** Foo foo() {

**return** **new** Foo(bar());

}

@Bean

**public** Bar bar() {

**return** **new** Bar();

}

}

In the example above, the foo bean receives a reference to bar via constructor injection.

### **Receiving lifecycle callbacks**

JavaConfig, like the core Spring Framework, supports use of JSR-250 "Common Annotations". For example:

**public** **class** FooService {

@PostConstruct

**public** **void** init() {

// custom initialization logic

}

}

@Configuration

@AnnotationDrivenConfig

**public** **class** ApplicationConfig {

@Bean

**public** FooService fooService() {

**return** **new** FooService();

}

}

In the above example, FooService declares @PostConstruct .

By declaring JavaConfig's @AnnotationDrivenConfig on The @Configuration class, this annotation will be respected by the container and called immediately after construction.

#### Using Spring interfaces

Spring's [lifecycle](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-nature)callbacks are fully supported. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods will be called by the container in accordance with their Javadoc.

#### 2.2.3.3.  Using @Bean initMethodName / destroyMethodName attributes

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-methodattributes to the bean element:

**public** **class** Foo {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** Bar {

**public** **void** cleanup() {

*// destruction logic*

}

}

@Configuration

**public** **class** AppConfig {

@Bean(initMethodName="init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethodName="cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}

Of course, in the case of Foo above, it would be equally as valid to call the init() method directly during construction:

@Configuration

**public** **class** AppConfig {

@Bean

**public** Foo foo() {

Foo foo = **new** Foo();

foo.init();

**return** foo;

}

*// ...*

}

|  |  |
| --- | --- |
| [Tip] | **Tip** |
| Remember that because you are working directly in Java, you can do anything you like with your objects, and do not always need to rely on the container! |

### 2.2.4. Using \*Aware interfaces

The standard set of \*Aware interfaces such as [BeanFactoryAware](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-aware-beanfactoryaware), [BeanNameAware](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-aware-beannameaware), [MessageSourceAware](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#context-functionality-messagesource), [ApplicationContextAware](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#context-functionality-events), etc. are fully supported. Consider an example class that implements BeanFactoryAware:

**public** **class** AwareBean **implements** BeanFactoryAware {

**private** BeanFactory factory;

*// BeanFactoryAware setter (called by Spring during bean instantiation)*

**public** **void** setBeanFactory(BeanFactory beanFactory) **throws** BeansException {

**this**.factory = beanFactory;

}

**public** **void** close(){

*// do clean-up*

}

}

If the class above were declared as a bean as follows:

@Configuration

**public** **class** AppConfig {

@Bean

**public** AwareBean awareBean() {

**return** **new** AwareBean();

}

}

its setBeanFactory method will be called during initialization, providing the bean with access to its enclosing BeanFactory.

### 2.2.5. Specifying bean scope

#### 2.2.5.1. Using @Bean's scope attribute

JavaConfig makes available each of the four standard scopes specified in [Section 3.4, "Bean Scopes"](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-scopes) of the Spring reference documentation.

The DefaultScopes class provides string constants for each of these four scopes. SINGLETON is the default, and can be overridden by supplying thescope attribute to @Bean annotation:

@Configuration

**public** **class** MyConfiguration {

@Bean(scope=DefaultScopes.PROTOTYPE)

**public** Encryptor encryptor() {

*// ...*

}

}

#### 2.2.5.2. @ScopedProxy

Spring offers a convenient way of working with scoped dependencies through [scoped proxies](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-scopes-other-injection). The easiest way to create such a proxy when using the XML configuration is the <aop:scoped-proxy/> element. JavaConfig offers equivalent support with the @ScopedProxy annotation, which provides the same semantics and configuration options.

If we were to port the the XML reference documentation scoped proxy example (see link above) to JavaConfig, it would look like the following:

*// 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;

}

#### 2.2.5.3. Lookup method injection

As noted in the core documentation, [lookup method injection](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-method-injection) is an advanced feature that should be comparatively rarely used. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. JavaConfig provides a natural means for implementing this pattern. Note that the example below is adapted from the example classes and configuration in the core documentation linked above.

**package** fiona.apple;

**public** **abstract** **class** CommandManager {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

JavaConfig can easily create a subclass of CommandManager where the abstract createCommand() is overridden in such a way that it 'looks up' a brand new (prototype) command object:

@Bean(scope=DefaultScopes.PROTOTYPE)

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with command() overridden*

*// to return a new prototype Command object*

**return** **new** CommandManager() {

**protected** Command command() {

**return** asyncCommand();

}

}

}

### 2.2.6. Customizing bean naming

By default, JavaConfig uses a @Bean method's name as the name of the resulting bean. This functionality can be overridden, however, using theBeanNamingStrategy extension point.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

JavaConfigApplicationContext ctx = **new** JavaConfigApplicationContext();

ctx.setBeanNamingStrategy(**new** CustomBeanNamingStrategy());

ctx.addConfigClass(MyConfig.**class**);

ctx.refresh();

ctx.getBean("customBeanName");

}

}

|  |  |
| --- | --- |
| [Note] | **Note** |
| JavaConfigApplicationContext will be covered in detail in [Chapter 3, *Using @Configuration classes*](http://docs.spring.io/spring-javaconfig/docs/1.0.0.M4/reference/html/ch03.html)  For more details, see the API documentation for BeanNamingStrategy. |

### 2.2.7. Working with Spring FactoryBean implementations

Spring provides many implementations of the FactoryBean interface. Usually these classes are used to support integrations with other frameworks. Take for example org.springframework.orm.hibernate3.LocalSessionFactoryBean. This class is used to create a Hibernate SessionFactory and requires as dependencies the location of Hibernate mapping files and a DataSource. Here's how it is commonly used in XML:

<beans>

<bean id="sessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

<property name="mappingResources">

<list>

<value>com/acme/Bank.hbm.xml</value>

<value>com/acme/Account.hbm.xml</value>

<value>com/acme/Customer.hbm.xml</value>

</list>

</property>

</bean>

<bean id="dataSource" class="...">

<*!-- ... --*>

</bean>

</beans>

The Spring container recognizes that LocalSessionFactoryBean implements the FactoryBean interface, and thus treats this bean specially: An instance ofLocalSessionFactoryBean is instantiated, but instead of being directly returned, instead the getObject() method is invoked. It is the object returned from this call getObject() that is ultimately registered as the sessionFactory bean.

How then would we use LocalSessionFactoryBean in JavaConfig? The best approach is to extend the ConfigurationSupport base class and use thegetObject() method:

@Configuration

**public** **class** DataAccessConfig **extends** ConfigurationSupport {

@Bean

**public** SessionFactory sessionFactory() {

LocalSessionFactoryBean factoryBean = **new** LocalSessionFactoryBean();

factoryBean.setDataSource(dataSource());

ArrayList<String> mappingFiles = **new** ArrayList<String>();

mappingFiles.add("com/acme/Bank.hbm.xml");

mappingFiles.add("com/acme/Account.hbm.xml");

mappingFiles.add("com/acme/Customer.hbm.xml");

factoryBean.setMappingResources(mappingFiles);

**return** **this**.getObject(SessionFactory.**class**, factoryBean);

}

*// ... other beans, including dataSource() ...*

}

Notice the call to this.getObject(Class, FactoryBean)? This call ensures that any container callbacks are invoked on the FactoryBean object, and then returns the value from the FactoryBean's getObject() in a type-safe fashion.

## 3. Using @Configuration classes

## 3.1.  Bootstrapping applications with JavaConfigApplicationContext

JavaConfigApplicationContext provides direct access to the beans defined by @Configuration-annotated classes. For more information on theApplicationContext API in general, please refer to the [Core Spring documentation](http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-introduction).

### 3.1.1. Construction Options

Instantiating the JavaConfigApplicationContext can be done by supplying @Configuration class literals to the constructor, and/or strings representing packages to scan for @Configuration classes.

#### 3.1.1.1. Construction by class literal

Each of the class literals supplied to the constructor will be processed, and for each @Bean method encountered, JavaConfig will create a bean definition and ultimately instantiate and initialize the bean.

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(AppConfig.**class**);

Service service = context.getBean(Service.**class**);

Passing multiple @Configuration classes:

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext(AppConfig.**class**, DataConfig.**class**);

Service service = context.getBean(Service.**class**);

#### 3.1.1.2. Construction by base package

Base packages will be scanned for the existence of any @Configuration classes. Any candidate classes will then be processed much as if they had been supplied directly as class literals to the constructor.

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext("com.acme.app.configuration");

Service service = context.getBean(Service.**class**);

Passing multiple base packages:

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext("com.acme.configuration", "com.acme.other");

Service service = context.getBean(Service.**class**);

Matching packages and classes by wildcard:

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext("\*\*/configuration/\*\*/\*.class", "\*\*/other/\*Config.class");

Service service = context.getBean(Service.**class**);

|  |  |
| --- | --- |
| [Note] | **Note** |
| The wildcard syntax for matching packages and classes above is based on [Ant Patterns](http://ant.apache.org/manual/dirtasks.html#patterns) |

#### 3.1.1.3. Post-construction configuration

When one or more classes/packages are supplied as constructor arguments, a JavaConfigApplicationContext instance cannot be further configured. If post-construction configuration is preferred or required, use either the no-arg constructor, configure by calling setters, then manually refresh the context. After the call to refresh(), the context will be 'closed for configuration'.

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext();

context.setParent(otherConfig);

context.setConfigClasses(AppConfig.**class**, DataConfig.**class**);

context.setBasePackages("com.acme.configuration");

context.refresh();

Service service = (Service) context.getBean("serviceA");

|  |  |
| --- | --- |
| [Note] | **Note** |
| Whenever multiple packages and/or classes are used to instantiate a JavaConfigApplicationContext, order matters. This is important when considering what happens if two configuration classes define a bean with the same name. The last-specified class wins. |

### 3.1.2. Accessing beans with getBean()

JavaConfigApplicationContext provides several variants of the getBean() method for accessing beans.

#### 3.1.2.1. Type-safe access

The preferred method for accessing beans is with the type-safe getBean() method.

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(...);

Service service = context.getBean(Service.**class**);

##### Disambuguation options

If more than one bean of type Service had been defined in the example above, the call to getBean() would have thrown an exception indicating an ambiguity that the container could not resolve. In these cases, the user has a number of options for disambiguation:

###### Indicating a @Bean as primary

Like Spring's XML configuration, JavaConfig allows for specifying a given @Bean as primary:

@Configuration

**public** **class** MyConfig {

@Bean(primary=Primary.TRUE)

**public** Service myService() {

**return** **new** Service();

}

@Bean

**public** Service backupService() {

**return** **new** Service();

}

}

After this modification, all calls to getBean(Service.class) will return the primary bean

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(...);

Service service = context.getBean(Service.**class**); *// returns the myService() primary bean*

###### Disambiguation by bean name

JavaConfig provides a getBean() variant that accepts both a class and a bean name for cases just such as these.

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(...);

Service service = context.getBean(Service.**class**, "myService");

Because bean ids must be unique, this call guarantees that the ambiguity cannot occur.

###### Retrieve all beans of a given type

It is also reasonable to call the getBeansOfType() method in order to return all beans that implement a given interface:

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(...);

Map matchingBeans = context.getBeansOfType(Service.**class**);

Note that this latter approach is actually a feature of the Core Spring Framework's AbstractApplicationContext (which JavaConfigApplicationContextextends) and is not type-safe, in that the returned Map is not parameterized.

#### 3.1.2.2. String-based access

Beans may be accessed via the traditional string-based getBean() API as well. Of course this is not type-safe and requires casting, but avoids any potential ambiguity entirely:

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(...);

Service service = (Service) context.getBean("myService");

**Modularizing configurations**

While the simplest configuration may be expressed as a single class that exposes several beans, it is often desirable to *modularize* configurations for reuse and clarity.

**4.1. Partitioning bean definitions into multiple @Configuration classes**

The simplest technique for modularizing configurations is to split up a single @Configuration class into multiple smaller classes:

*// monolithic configuration*

@Configuration

**public** **class** AppConfig {

@Bean

**public** ServiceA serviceA() {

*// ...*

}

@Bean

**public** ServiceB serviceB() {

*// ...*

}

*// assume many bean definitions follow*

}

The above configuration class might be supplied as a parameter to JavaConfigApplicationContext:

JavaConfigApplicationContext context = **new** JavaConfigApplicationContext(AppConfig.**class**);

ServiceA serviceA = context.getBean(ServiceA.**class**);

ServiceB serviceB = context.getBean(ServiceB.**class**);

We can easily partition this configuration such that bean definitions are spread across two classes, instead of one:

*// partitioned configuration*

@Configuration

**public** **class** AppConfigA {

@Bean

**public** ServiceA serviceA() {

*// ...*

}

}

@Configuration

**public** **class** AppConfigB {

@Bean

**public** ServiceB serviceB() {

*// ...*

}

}

Now simply supply both configuration classes to the constructor of JavaConfigApplicationContext:

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext(AppConfigA.**class**, AppConfigB.**class**);

*// both beans are still available in the resulting application context*

ServiceA serviceA = context.getBean(ServiceA.**class**);

ServiceB serviceB = context.getBean(ServiceB.**class**);

## Referencing beans across @Configuration classes

One configuration class may need to reference a bean defined in another configuration class (or in XML, for that matter). The preferred mechanism for doing this is using Spring's @Autowired annotation:

### 4.2.1. Direct bean references with @Autowired

One @Configuration class may directly reference bean instances registered from another using Spring's @Autowired annotation.

@Configuration

**public** **class** ConfigOne {

@Bean

**public** AccountRepository accountRepository() {

*// create and return an AccountRepository object*

}

}

@Configuration

@AnnotationDrivenConfig

**public** **class** ConfigTwo {

@Autowired AccountRepository accountRepository;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

Given that both these configuration classes are supplied to the application context at runtime, the AccountRepository bean declared in ConfigOne will be autowired (injected) into the AccountRepository field in ConfigTwo.

JavaConfigApplicationContext context =

**new** JavaConfigApplicationContext(ConfigOne.**class**, ConfigTwo.**class**);

### 4.2.2. Fully-qualified bean references with @Autowired

In addition to being able to reference any particular bean definition as seen above, one @Configuration class may reference the instance of any other@Configuration class using @Autowired. This works because the @Configuration classes themselves are instantiated and managed as individual Spring beans.

@Configuration

**public** **class** ConfigOne {

@Bean

**public** AccountRepository accountRepository() {

*// create and return an AccountRepository object*

}

}

@Configuration

@AnnotationDrivenConfig

**public** **class** ConfigTwo {

@Autowired ConfigOne configOne;

@Bean

**public** TransferService transferService() {

*// transferService references accountRepository in a 'fully-qualified' fashion:*

**return** **new** TransferServiceImpl(configOne.accountRepository());

}

}

|  |  |
| --- | --- |
| [Tip] | **Tip** |
| The 'fully-qualified' approach is generally preferred as it provides a the significant advantage of being able to easily navigate within an IDE to the source of the referenced bean |

**Aggregating @Configuration classes with @Import**

Thus far, we've seen how to break up bean definitions into multiple @Configuration classes and how to reference those beans across @Configurationboundaries. These scenarios have required providing all @Configuration classes to the constructor of a JavaConfigApplicationContext, and this is not always ideal. Often it is preferable to use an *aggregation* approach, where one @Configuration class logically *imports* the bean definitions defined by another.

The @Import annotation provides just this kind of support, and it is the direct equivalent of the <import/> element found in Spring beans XML files.

@Configuration

**public** **class** DataSourceConfig {

@Bean

**public** DataSource dataSource() {

**return** **new** DriverManagerDataSource(...);

}

}

@Configuration

@AnnotationDrivenConfig

@Import(DataSourceConfig.**class**) *// <-- AppConfig imports DataSourceConfig*

**public** **class** AppConfig **extends** ConfigurationSupport {

@Autowired DataSourceConfig dataSourceConfig;

@Bean

**public** **void** TransferService transferService() {

**return** **new** TransferServiceImpl(dataSourceConfig.dataSource());

}

}

The bootstrapping of this application is simplified, as it only needs to supply AppConfig when instantiating a JavaConfigApplicationContext.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

JavaConfigApplicationContext ctx =

**new** JavaConfigApplicationContext(AppConfig.**class**); *// specifies single class*

*// ...*

}

}

Multiple configurations may be imported by supplying an array of classes to the @Import annotation

@Configuration

@Import({ DataSourceConfig.**class**, TransactionConfig.**class** })

**public** **class** AppConfig **extends** ConfigurationSupport {

*// @Bean methods here can reference @Bean methods in DataSourceConfig or TransactionConfig*

}

**ConfigurationSupport**

As a convenience, @Configuration classses may extend ConfigurationSupport, primarily in order to facilitate easy lookup of beans from the enclosingApplicationContext.

@Configuration

**public** **class** AppConfig **extends** ConfigurationSupport {

@Bean

**public** Service serviceA() {

*// referencing a bean of type DataSource declared elsewhere*

DataSource dataSource = **this**.getBean(DataSource.**class**);

**return** **new** ServiceImpl(dataSource);

}

}

## Working with externalized values

## 5.1.  @ExternalValue

**What about PropertyOverrideConfigurer?**

Those familiar with XML configuration will notice that there is not a direct equivalent for the popular PropertyOverrideConfigurer or the more recent<context:property-placeholder/>. However, the combined use of JavaConfig's @ExternalValue and various ValueSource annotations provide similar functionality.

Externally defined values such as usernames, passwords, file paths, and the like may be accessed using @ExternalValue and one or more of JavaConfig's ValueSource annotations.

### 5.1.1. @ExternalValue fields

@Configuration

@PropertiesValueSource("classpath:com/acme/db.properties")

**public** **class** AppConfig {

@ExternalValue("datasource.username") String username;

@Bean

**public** TestBean testBean() {

**return** **new** TestBean(username);

}

}

com/acme/db.properties will be read from the classpath and the value associated with key datasource.username will be injected into the username field. The contents of db.properties might be as follows:

datasource.username=scott

datasource.password=tiger

...

|  |  |
| --- | --- |
| [Note] | **Note** |
| An array of properties file locations may be supplied to @PropertiesValueSource, and along with classpath:, all of the standard Spring resource-loading prefixes are supported, such as file: and http:.  @Configuration  @PropertiesValueSource({"classpath:com/acme/a.properties", "file:/opt/acme/b.properties"})  **public** **class** AppConfig {  *// ...*  } |

### 5.1.2. @ExternalValue methods

@ExternalValue may also be used as a method-level annotation

@Configuration

@PropertiesValueSource("classpath:com/acme/db.properties")

**public** **abstract** **class** AppConfig {

@ExternalValue("datasource.username")

**abstract** String username();

@Bean

**public** TestBean testBean() {

**return** **new** TestBean(username());

}

}

The primary advantage to using @ExternalValue methods is that rather than injecting the external value just once (as is done in the case of @ExternalValuefields), @ExternalValue methods are evaluated every time they're referenced. As this is not usually required, @ExternalValue fields are the preferred method. A downside of @ExternalValue methods is that they should be abstract, requiring you to declare the entire @Configuration class abstract, and this is not in alignment with the semantics users typically associate with using the abstract keyword.

**5.2. Available ValueSource annotations**

* @PropertiesValueSource
* @EnvironmentValueSource
* @SystemPropertiesValueSource

ValueSource annotations may be used in conjunction:

@Configuration

@EnvironmentValueSource

@SystemPropertiesValueSource

@PropertiesValueSource("classpath:com/acme/db.properties")

**public** **class** AppConfig {

@ExternalValue("datasource.username") String username;

@Bean

**public** TestBean testBean() {

**return** **new** TestBean(username);

}

}

In this example, datasource.username will be looked for in db.properties, in the set of environment variables present at runtime and in the system properties.

## Combining configuration styles

JavaConfig can be used in conjunction with any or all of Spring's other container configuration approaches.

## 6.1. JavaConfig and XML

### 6.1.1.  Bootstrapping JavaConfig from XML with ConfigurationPostProcessor

You may desire or be required to use XML as the primary mechanism for configuring the container, but wish to selectively use @Configuration classes to define certain beans. For such cases, JavaConfig provides ConfigurationPostProcessor, a Spring BeanPostProcessor capable of processing @Configurationclasses.

<beans>

<*!-- first, define your individual @Configuration classes as beans --*>

<bean class="com.myapp.config.AppConfig"/>

<bean class="com.myapp.config.DataConfig"/>

<*!-- be sure to include the JavaConfig bean post-processor --*>

<bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

</beans>

Then, bootstrap an XML ApplicationContext:

ApplicationContext context = **new** ClassPathXmlApplicationContext("application-config.xml");

The beans defined in AppConfig and DataConfig will be available via context.

#### 6.1.1.1. Configuring configurations

An added benefit that comes along with bootstrapping JavaConfig from XML is that the configuration bean instances are eligible, just as any other bean, for dependency injection:

<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>

<*!-- JavaConfig post-processor --*>

<bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>

</beans>

### 6.1.2.  Bootstrapping XML from JavaConfig with @ImportXml

The @ImportXml annotation is provided to support importing beans defined in XML into @Configuration classes.

datasource-config.xml:

<beans>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>

<property name="driverClassName" value="org.hsqldb.jdbcDriver"/>

<property name="username" value="sa"/>

<property name="password" value=""/>

</bean>

</beans>

@Configuration

@AnnotationDrivenConfig *// enable the @Autowired annotation*

@ImportXml("classpath:com/company/app/datasource-config.xml")

**public** **class** Config {

*// autowire the DataSource bean declared in datasource-config.xml*

@Autowired DataSource dataSource;

@Bean

**public** FooRepository fooRepository() {

*// inject the autowired-from-XML dataSource*

**return** **new** JdbcFooRepository(dataSource);

}

@Bean

**public** FooService fooService() {

**return** **new** FooServiceImpl(fooRepository());

}

}

|  |  |
| --- | --- |
| [Tip] | **Tip** |
| Regardless of the bootstrapping mechanism used - ConfigurationPostProcessor or @ImportXml - bean references may always be bi-directional. XML-defined beans may reference @Configuration-defined beans and vice-versa. |

## JavaConfig and Annotation-Driven Configuration

### 6.2.1. @AnnotationDrivenConfig

Spring 2.5 introduced a new style of dependency injection with Annotation-Driven Injection. In Spring XML, Annotation-Driven Injection is enabled in the container by declaring

<context:annotation-config/>

In JavaConfig, this same functionality is enabled with the @AnnotationDrivenConfig annotation

@Configuration

@AnnotationDrivenConfig

**public** **class** Config {

*// may now use @Autowired to reference beans from other @Configuration classes, XML, etc*

}

### 6.2.2. @ComponentScan

An equivalent for Spring XML's <context:component-scan/> is provided with the @ComponentScan annotation.

**package** com.company.foo;

@Service

**public** **class** FooServiceImpl **implements** FooService {

**private** **final** FooRepository fooRepository;

@Autowired

**public** FooService(FooRepository fooRepository) {

**this**.fooRepository = fooRepository;

}

*// ...*

}

**package** com.company.foo;

@Repository

**public** **class** JdbcFooRepository **implements** FooRepository {

**private** **final** DataSource dataSource;

@Autowired

**public** FooRepository(DataSource dataSource) {

**this**.dataSource = dataSource;

}

*// ...*

}

@Configuration

@ComponentScan("com.company") *// search the com.company package for @Component classes*

@ImportXml("classpath:com/company/data-access-config.xml") *// XML with DataSource bean*

**public** **class** Config {

}

Because Spring's @Service and @Repository stereotype annotations are each meta-annotated with @Component, they are candidates for component scanning. Because FooServiceImpl and JdbcFooRepository both reside underneath the com.company package, they will be discovered during component scanning and will be autowired together. @ImportXml pulls in the DataSource bean, ensuring it will be available for autowiring into JdbcFooRepository.

With the above very minimal configuration in the Config class, we can bootstrap and use the application as follows:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

JavaConfigApplicationContext ctx = **new** JavaConfigApplicationContext(Config.**class**);

FooService fooService = ctx.getBean(FooService.**class**);

fooService.doStuff();

}

}

## Transaction-management support

## 7.1. @AnnotationDrivenTx

JavaConfig provides full support for the annotation-driven declarative transaction management features provided by the core Spring Framework with the@AnnotationDrivenTx annotation:

**public** **class** FooServiceImpl **implements** FooService {

@Transactional

**public** **void** doStuff() {

*// invoke multiple calls to data access layer*

}

}

@Configuration

@AnnotationDrivenTx

**public** **class** Config {

@Bean

**public** FooService fooService() {

**return** **new** FooServiceImpl(fooRepository());

}

@Bean

**public** FooRepository fooRepository() {

**return** **new** JdbcFooRepository(dataSource());

}

@Bean

**public** PlatformTransactionManager transactionManager() {

**return** **new** DataSourceTransactionManeger(dataSource());

}

@Bean

**public** DataSource dataSource() {

*// create and return a new JDBC DataSource ...*

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

JavaConfigApplicationContext ctx = **new** JavaConfigApplicationContext(Config.**class**);

*// The FooService retrieved from the container will be proxied for tx management*

FooService fooService = ctx.getBean(FooService.**class**);

*// call the @Transactional method on the proxy - transactional behavior is guaranteed*

fooService.doStuff();

}

}

Like Spring XML's <tx:annotation-driven> element, @AnnotationDrivenTx expects the presence of a bean named transactionManager of typePlatformTransactionManager as in the example above. Should you wish to forego this convention and name a transaction manager bean another name, you may do as follows:

@Configuration

@AnnotationDrivenTx(transactionManager="txManager") *// specify explicitly the bean to use*

**public** **class** Config {

@Bean

**public** PlatformTransactionManager txManager() {

**return** **new** DataSourceTransactionManeger(dataSource());

}

*// other beans...*

}

The other attributes available to the @AnnotationDrivenTx are similar to the attributes to the <tx:annotation-driven> element. See the [related documentation](http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html#transaction-declarative-annotations)and the JavaDoc for @AnnotationDrivenTx for details.

**AOP (Aspect Oriented Programming)**

* AOP is a new kind of programming language whose main goal is to provide clean separation between cross cutting concerns and core business logic.

**AOP Terminology:**

1. Aspect
2. Advice
3. Join Point
4. Point Cut
5. Advicer
6. Introduction
7. Target
8. Proxy
9. Weaving
10. **Aspect: (crosscutting concern commonly used in the application)**

Aspect is a crosscutting concern which is commonly required for various requirements of a module and various modules of your enterprise application.

For eg:

Logging, Security and transaction or the aspects which are commonly required.

1. **Advice:**

Advice is a java class which has actual implementation of an aspect.

For eg:

Advice is a java class where logging related codes.

1. **Join Point:**

Join Point is a point in a program execution where exactly you want to apply advices.

Spring AOP supports various joint points related to method.

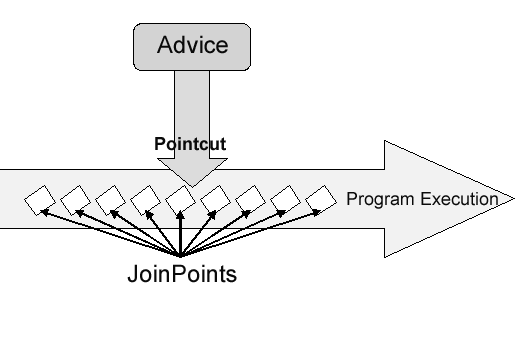
For eg:

1. Method before
2. Method returning successfully
3. Method throwing exception

Aspect J framework provides field level advices also.

1. **Point Cut:**

Point Cut is collection of joint points.



1. **Advicer**

Advicer is a combination of point cut and advice i.e. advices know where to apply (advice) and what to apply (advice).

1. **Introduction**

Introduction is a process of adding new variables and methods to the class dynamically at runtime.

1. **Target**

Target is an object on which you want to apply the advice.

1. **Proxy**

Proxy is an object which you will get after applying advice to the target object.

1. **Weaving:**

Weaving is the process of applying advicers to the target object at joins point to get the proxy objects.

* Weaving can be done in three ways:

1. Compile time weaving (Source Code level)
2. Load Time Weaving (Byte Code Level)
3. Run Time Weaving (Native code level)

* Following are various AOP providers available in market:

1. Spring AOP
2. AspectJ AOP
3. JBoss AOP

* Spring offers three different ways to implement AOP

1. Classical model (Using AOP API)
2. Annotation Based Model
3. Schema Based Model

b and c are available in Spring 2 onwards.

1. Implementing AOP using AOP API (Classical modal)

* Spring AOP API provides the following four types of advices.

1. Method Before Advice
2. After Returning Advice
3. Throws Advice
4. Method Interceptor

Types of advice:

1. *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
2. *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.
3. *After throwing advice*: Advice to be executed if a method exits by throwing an exception.
4. *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).
5. *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

|  |  |
| --- | --- |
|  |  |
| Before Advice | * Service which need to execute before business logic. * During compilation time the services will not be applied to our logic, services will apply only at run time. * To create a Before Advice, you should implement MethodBeforeAdviceinterface. * **org.springframework.aop.MethodBeforeAdvice;** * You need to override “before()” method. * The services which are implemented in before() method will be executed before business logic.   public class  beforeAdvice implements MethodBeforeAdvice  {  @Override  **publicvoid** before(Method arg0, Object[] arg1, Object arg2) **throws** Throwable {      }}  Advice invoked before a method is invoked. Such advices cannot prevent the method call proceeding, unless they throw a Throwable. |
| After Advice | * Service which need to execute after business logic. * During compilation time the services will not be applied to our logic, services will apply only at run time. * To create an After Advice, you should implement AfterReturningAdvice interface. * AfterReturningAdvice interface is given in org.sp-fw.aop.\* package. * You need to override “afterReturning()” method. * The services which are implemented in afterReturning() method will be executed before business logic.  |  |  | | --- | --- | |  | public class  AfterAdvice implements AfterReturningAdvice  {      public void afterReturning(Object returnValue,Object args[], Object target)throws Exception      {                  //My After Logic...      }  } |   After returning advice is invoked only on normal method return, not if an exception is thrown. Such advice can see the return value, but cannot change it. |
| Throws Advice | * Run after the method throws an exception * **implements** ThrowsAdvice   **publicvoid** afterThrowing(IllegalArgumentException e)**throws**Throwable{ System.out.println("XXXXXXXXX");  }  Tag interface for throws advice.  There are not any methods on this interface, as methods are invoked by reflection. Implementing classes must implement methods of the form:  void afterThrowing([Method, args, target], ThrowableSubclass);  Some examples of valid methods would be:  public void afterThrowing(Exception ex)  public void afterThrowing(RemoteException)  public void afterThrowing(Method method, Object[] args, Object target, Exception ex)  public void afterThrowing(Method method, Object[] args, Object target, ServletException ex)  The first three arguments are optional, and only useful if we want further information about the joinpoint, as in AspectJ **after-throwing** advice.  **Note:** If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a RuntimeException; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. **Do not throw an undeclared checked exception that is incompatible with the target method's signature!** |
| Around Advice  Or  MethodInterceptor | * Run around the method execution, combine all three advices above. * **implements** MethodInterceptor * **public**Object invoke(MethodInvocation methodInvocation)**throws**Throwable * methodInvocation.proceed(); |
|  |  |

|  |  |
| --- | --- |
| **Packages** |  |
| org.springframework.aop | MethodBeforAdvice  AfterReturningAdvice  ThrowsAdvice |
| org.aopalliance.intercept | MethodInterceptor |
| org.springframework.aop.aspectj.autoproxy | AspectJAwareAdvisorAutoProxyCreator |
| org.springframework.aop.support | JdkRegexpMethodPointcut  DefaultPointcutAdvisor |
| org.springframework.aop.framework | ProxyFactoryBean |
| org.springframework.aop.aspectj | AspectJExpressionPointcut |

**AspectJ annotation support**

@Aspect

@Pointcut

@Before

@AfterReturning

@AfterThrowing

@After

@Arround

**Schema based support**

<aop:config>

<aop:aspect>

<aop:pointcut>

<aop:before>

<aop:after-returning>

<aop:after-throwing>

<aop:after>

<aop:around>

Example:

1. Advice
2. Target Class
3. aop-config.xml

additional jar files require for this example are:

spring-aop-3.2.6.RELEASE.jar

aopalliance-1.0.jar

1. Advice

MethodAfterFundTransfer.java

MethodBeforeFundTransfer.java

ThrowErrorFundTrnasfer.java

**MethodBeforeFundTransfer.java**

**package** com.test.spring.aop;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.MethodBeforeAdvice;

**publicclass** MethodBeforeFundTransfer **implements** MethodBeforeAdvice {

@Override

**publicvoid** before(Method arg0, Object[] arg1, Object arg2) **throws** Throwable {

System.*out*.println("Welcome to Visit Rich & Poor Bank!");

}

}

**MethodAfterFundTransfer.java**

**package** com.test.spring.aop;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.AfterReturningAdvice;

**publicclass** MethodAfterFundTransfer **implements**AfterReturningAdvice {

@Override

**publicvoid**afterReturning(Object arg0, Method arg1, Object[] arg2, Object arg3) **throws** Throwable {

System.*out*.println("Thank you for Visiting Rich & Poor Bank! ");

}

}

**ThrowErrorFundTrnasfer.java**

**package** com.test.spring.aop;

**import** java.lang.reflect.Method;

**import** org.springframework.aop.ThrowsAdvice;

**publicclass** ThrowErrorFundTrnasfer **implements** ThrowsAdvice {

**publicvoid** afterThrowing(Method method, Object[] args, Object target, Exception ex){

System.*out*.println("Sorry! Please deposite some amount to reach the minimum balance 5000.");

}

}

1. Target Class

FundTransferEligiblity.java

**package** com.test.spring.aop;

**publicclass** FundTransferEligiblity {

**publicboolean** fundTransferEligible(**int** balance) **throws** InSufficientFund {

**if** (balance > 5000) {

System.*out*.println("Sufficient Fund for transfer!");

**returntrue**;

} **else** {

**thrownew** InSufficientFund("you have not sufficient fund to transfer!");

}

}

}

**class** InSufficientFund **extends** Exception {

/\*\*

\*

\*/

**privatestaticfinallong***serialVersionUID* = 1L;

InSufficientFund() {

**super**();

}

InSufficientFund(String msg) {

**super**(msg);

}

}

1. **Client Class**

**package** com.test.spring.aop;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** TestClientWithAop {

**publicstaticvoid** main(String[] args) {

**int** balance = 9000;

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-aop-ft.xml");

FundTransferEligiblity fte = (FundTransferEligiblity) context.getBean("fdTarget");

FundTransferEligiblity fte1 = (FundTransferEligiblity) context.getBean("fdProxy");

**try** {

//fte.fundTransferEligible(balance);

// output: you have not sufficient fund to transfer!

fte1.fundTransferEligible(balance);

} **catch** (InSufficientFund e) {

System.*out*.println(e.getMessage());

}

}

}

/\*Welcome to Visit Rich & Poor Bank!

Sorry! Please deposite some amount to reach the minimum balance 5000.

you have not sufficient fund to transfer!\*/

/\*Welcome to Visit Rich & Poor Bank!

Sufficient Fund for transfer!

Thank you for Visiting Rich & Poor Bank!\*/

1. aop-config.xml

**spring-aop-ft.xml**

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"*>

<beanid=*"fdTarget"*class=*"com.test.spring.aop.FundTransferEligiblity"*/>

<beanid=*"fdMethodBefore"*class=*"com.test.spring.aop.MethodBeforeFundTransfer"*/>

<beanid=*"fdMethodAfter"*class=*"com.test.spring.aop.MethodAfterFundTransfer"*/>

<beanid=*"fdThrowError"*class=*"com.test.spring.aop.ThrowErrorFundTrnasfer"*/>

<beanid=*"fdProxy"*class=*"org.springframework.aop.framework.ProxyFactoryBean"*>

<propertyname=*"targetClass"*value=*"com.test.spring.aop.FundTransferEligiblity"*/>

<propertyname=*"target"*ref=*"fdTarget"*/>

<propertyname=*"interceptorNames"*>

<list>

<value>fdMethodBefore</value>

<value>fdMethodAfter</value>

<value>fdThrowError</value>

</list>

</property>

</bean>

</beans>

org.springframework.aop.framework.ProxyFactoryBean

target

Set the given object as target. Will create a SingletonTargetSource for the object.

targetClass

Set a target class to be proxied, indicating that the proxy should be castable to the given class.

Internally, an [org.springframework.aop.target.EmptyTargetSource](eclipse-javadoc:%E2%98%82=TestSpringAOP/lib%5C/spring-aop-3.2.6.RELEASE.jar%3Corg.springframework.aop.framework(AdvisedSupport.class%E2%98%83AdvisedSupport~setTargetClass~Ljava.lang.Class%5C%3C*%3E;%E2%98%82org.springframework.aop.target.EmptyTargetSource) for the given target class will be used. The kind of proxy needed will be determined on actual creation of the proxy.

This is a replacement for setting a "targetSource" or "target", for the case where we want a proxy based on a target class (which can be an interface or a concrete class) without having a fully capable TargetSource available

interceptorNames

Set the list of Advice/Advisor bean names. This must always be set to use this factory bean in a bean factory.

The referenced beans should be of type Interceptor, Advisor or Advice The last entry in the list can be the name of any bean in the factory. If it's neither an Advice nor an Advisor, a new SingletonTargetSource is added to wrap it. Such a target bean cannot be used if the "target" or "targetSource" or "targetName" property is set, in which case the "interceptorNames" array must contain only Advice/Advisor bean names.

**NOTE: Specifying a target bean as final name in the "interceptorNames" list is deprecated and will be removed in a future Spring version.** Use the ["targetName"](eclipse-javadoc:%E2%98%82=TestSpringAOP/lib%5C/spring-aop-3.2.6.RELEASE.jar%3Corg.springframework.aop.framework(ProxyFactoryBean.class%E2%98%83ProxyFactoryBean~setInterceptorNames~%5C%E2%98%83Ljava.lang.String;%E2%98%82%E2%98%82setTargetName) property instead.

**Point Cuts and Advisors**

Spring provides two ways to define the point cuts:

1. jdk regular expression based point cut
2. Aspect expression based point cuts
3. jdk regular expression based point cut (reg exp):

<bean id = “appPointCut” class = “org.springframework.aop.support.JdkRegexpMethodPointCut”>

<property name = “pattern” value =”.add\*()” />

</bean>

1. AspectJ expression based point cuts:

<bean id = “org.springframework.aop.aspectj.AspectJExpressionPointCut”>

<property name = “expression” value = “execution(public \*Hello.m\*(..))”/>

</bean>

AspectJ expression syntax

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) **throws**-pattern?)

eg:-

execution( public \* Hello.m\*(..))

execution( int \*.add\*(..))

execution( \* \* service.\*(..))

execution( \* com.app.p1.Hello\*.\*(..))

execution( \* com.app.p1.Hello\*.\*(..) throws HelloException)

execution( \*service.\*(..)) //-> invalid expression

The execution of any public method:

* execution(**public** \* \*(..))

The execution of any method with a name beginning with "set":

* execution(\* set\*(..))

The execution of any method defined by the AccountService interface:

* execution(\* com.xyz.service.AccountService.\*(..))

The execution of any method defined in the service package:

* execution(\* com.xyz.service.\*.\*(..))

The execution of any method defined in the service package or a sub-package:

* execution(\* com.xyz.service..\*.\*(..))

**Advisor**

* Advicer should know what to (advice) apply and where to apply(point cut)
* Configure the following to form the advicer

<bean id = “mba” class = “com.app.MBAdvice” />

<bean id = “mbAdvicer” class = “org.springframework.aop.support.DefaultPointcutAdvisor” />

<property name = “advice” ref = “mba” />

<property name = “pointcut” ref = “appPointCut”/>

</bean>

**Autoproxying**

1. If you want to create proxy objects for any given target object you need to configure your bean in the “spring-context.xml” with ProxyFactoryBean class.
2. If you have more business services to apply the advicers than you need to configure more beans with ProxyFactoryBean to create the proxy objects. This increases the complexity for the developer and also increases the size of your “spring-context.xml”.
3. To avoid this autoproxy concept is provided in spring2.0.
4. Autoproxying is responsible to detect the beans automatically to create the proxy objects without configuring ProxyFactoryBean.
5. You can implement autoproxy in two ways:
6. Annotation based autoproxying
7. Xml based autoproxying
8. To implement any of the above stated autoproxy way you need to do the following steps commonly.
9. Enable the AOP namespace.
10. Write the following tag to enable the autoproxy.

<aop : aspectj-autoproxy />

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.0.xsd "*>

*<aop:aspectj-autoproxy />*

</beans>

1. Schema based support (Xml based autoproxying)
   * + To implement xml schema based autoproxying we need to configure all the pointcut and target into the configuration file. (spring-context.xml)
     + We need to write the class with the implementation of cross cutting concerns.
     + Methods can be of any name but its parameters should be according to the type of advice:

* For Before:

**publicvoid** logBefore(JoinPoint joinPoint) {…}

* For After:

**publicvoid** logAfter(JoinPoint joinPoint) {…}

* For After Returning:

**publicvoid** logAfterReturning(JoinPoint joinPoint, Object result) {…}

* For Throwing:

**publicvoid** logAfterThrowing(JoinPoint joinPoint, Throwable error) {…}

* For Around:

**publicvoid** logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {…}

* + - These method’s name will be configured in the configuration file as the “method” attribute of the corresponding advice tag (for example: <aop:before>).
    - Configure this class in the configuration file.

<beanid=*"advice"*class=*"com.test.spring.aop.log.LogAdvice"*/>

* + - No need to use “<aop:aspectj-autoproxy />”, in the configuration file.
    - Use <aop:config></aop:config> to configure point cut and to apply the aspects on the target object.

<aop:config>

<aop:aspectid=*"loggingAspect"*ref=*"advice"*>

<aop:pointcutid=*"beforePointCut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doing\*(..))"*/>

<aop:beforemethod=*"logBefore"*pointcut-ref=*"beforePointCut"*/>

</aop:aspect>

</aop:config>

* <aop:aspect > take the aspect class (Advice), which contains the implementation of the cross cutting concern.
* <aop:pointcut > define the point cut, expression attribute is mandatory. The value of the expression attribute will be the method of the target class where we need to apply the cross cutting concern.
* <aop:before> needs to apply cross cutting concern before method.
* Two attributesare required:
  + - * method
      * pointcut-ref
    - method: the method name where cross cutting concern is defined in the aspect class. method name should be just method name without “()”.
    - pointcut-ref: the id of the pointcut where we want to call the method.
* <aop:after>
  + - * method
      * pointcut-ref
* <aop:after-returning>
  + - * method
      * returning : parameter name in aspect method
      * pointcut-ref
* <aop:after-throwing>
  + - * method
      * throwing : parameter name in aspect method
      * pointcut-ref
* <aop:around>
  + - * method
      * pointcut-ref

<aop:config>

<aop:aspectid=*"loggingAspect"*ref=*"advice"*>

<aop:pointcutid=*"beforePointCut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doing\*(..))"*/>

<aop:beforemethod=*"logBefore"*pointcut-ref=*"beforePointCut"*/>

<aop:pointcutid=*"afterPointCut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doing\*(..))"*/>

<aop:aftermethod=*"logAfter"*pointcut-ref=*"afterPointCut"*/>

<aop:pointcutid=*"afterReturning"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisWithReturn(..))"*/>

<aop:after-returningmethod=*"logAfterReturning"*returning=*"result"*pointcut-ref=*"afterReturning"*/>

<aop:pointcutid=*"afterThrowingError"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisError(..))"*/>

<aop:after-throwingmethod=*"logAfterThrowing"*throwing=*"error"*pointcut-ref=*"afterThrowingError"*/>

<aop:pointcutid=*"aroundPointcut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisError(..))"*/>

<aop:aroundmethod=*"logAround"*pointcut-ref=*"aroundPointcut"*/>

</aop:aspect>

</aop:config>

Example:

1. LogTarget.java
2. LogAdvice.java
3. spring-context.xml
4. TestAop.java
5. LogTarget.java

**package** com.test.spring.aop.log;

**publicclass** LogTarget {

**publicvoid** doingThis() {

System.*out*.println("inside doingThis()....");

}

**public** Integer doingThisWithReturn() {

System.*out*.println("doingThisWithReturn()..........");

**return** 5000;

}

**publicint**doingThisError(**int** i) **throws** Exception {

**if** (i < 5000) {

**thrownew** Exception("there is some problem....");

}

**return** i;

}

}

1. LogAdvice.java

**package** com.test.spring.aop.log;

**import** java.util.Arrays;

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.ProceedingJoinPoint;

**publicclass** LogAdvice {

**publicvoid** logBefore(JoinPoint joinPoint) {

System.*out*.println("logBefore() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("\*\*\*\*\*\*");

}

**publicvoid** logAfter(JoinPoint joinPoint) {

System.*out*.println("logAfter() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("\*\*\*\*\*\*");

}

**publicvoid** logAfterReturning(JoinPoint joinPoint, Object result) {

System.*out*.println("logAfterReturning() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("Method returned value is : " + result);

System.*out*.println("\*\*\*\*\*\*");

}

**publicvoid** logAfterThrowing(JoinPoint joinPoint, Throwable error) {

System.*out*.println("logAfterThrowing() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("Exception Message : " + error.getMessage());

System.*out*.println("\*\*\*\*\*\*");

}

**publicvoid** logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {

System.*out*.println("logAround() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("advice Method arguments : " + Arrays.*toString*(joinPoint.getArgs()));

System.*out*.println("Around before is running!");

joinPoint.proceed();

System.*out*.println("Around after is running!");

System.*out*.println("\*\*\*\*\*\*");

}

}

1. spring-context.xml

<?xmlversion=*"1.0"*encoding=*"UTF-8"*?>

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.0.xsd "*>

<beanid=*"advice"* class=*"com.test.spring.aop.log.LogAdvice"*/>

<beanid=*"target"*class=*"com.test.spring.aop.log.LogTarget"*/>

<aop:config>

<aop:aspectid=*"loggingAspect"*ref=*"advice"*>

<aop:pointcutid=*"beforePointCut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doing\*(..))"*/>

<aop:beforemethod=*"logBefore"*pointcut-ref=*"beforePointCut"*/>

<aop:pointcutid=*"afterPointCut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doing\*(..))"*/>

<aop:aftermethod=*"logAfter"*pointcut-ref=*"afterPointCut"*/>

<aop:pointcutid=*"afterReturning"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisWithReturn(..))"*/>

<aop:after-returningmethod=*"logAfterReturning"*returning=*"result"*pointcut-ref=*"afterReturning"*/>

<aop:pointcutid=*"afterThrowingError"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisError(..))"*/>

<aop:after-throwingmethod=*"logAfterThrowing"*throwing=*"error"*pointcut-ref=*"afterThrowingError"*/>

<aop:pointcutid=*"aroundPointcut"*expression=*"execution(\* com.test.spring.aop.log.LogTarget.doingThisError(..))"*/>

<aop:aroundmethod=*"logAround"*pointcut-ref=*"aroundPointcut"*/>

</aop:aspect>

</aop:config>

</beans>

1. TestAop.java

**package** com.test.spring.aop.log;

**import** org.springframework.context.ApplicationContext;

**import**org.springframework.context.support.AbstractApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** TestAop {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-context.xml");

LogTarget lt = (LogTarget)context.getBean("target");

//lt.doingThis();

//lt.doingThisWithReturn();

**try** {

lt.doingThisError(300);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

System.*out*.println(e.getMessage());

}

//((AbstractApplicationContext)context).registerShutdownHook();

}

}

1. Annotation based autoproxying
   * + To implement annotation based autoproxying we need one Advice class.
     + This advice class will contain the implementation of all the aspects (cross-cutting concerns).
     + We’ll have to use “@Aspect” annotation for this class.

org.aspectj.lang.annotation.Aspect

* + - We can use below four kind of annotation:

@Before

* org.aspectj.lang.annotation.Before
* This annotation supports, method before cross cutting concern implementation.
* **publicvoid** logBefore(JoinPoint joinPoint)

@After

* org.aspectj.lang.annotation.After
* This annotation supports, method after cross cutting concern implementation.
* **publicvoid** logAfter(JoinPoint joinPoint)

@AfterReturning

* org.aspectj.lang.annotation.AfterReturning
* This annotation supports, method after cross cutting concern and support the modification of the return value implementation.
* **returning = “result Parameter”** attribute is required.
* **publicvoid** logAfterReturning(JoinPoint joinPoint, Object result)

@AfterThrowing

* org.aspectj.lang.annotation.AfterThrowing
* This annotation supports, the cross cutting concern implementation after throwing error from the method.
* **throwing = “error Parameter”** attribute is required.
* **publicvoid** logAfterThrowing(JoinPoint joinPoint, Throwable error)

@Around

* org.aspectj.lang.annotation.Around
* This annotation supports, the cross cutting concern implementation before and after method call.
* **publicvoid** logAround(ProceedingJoinPoint joinPoint) **throws** Throwable

1. LoggingAdvice.java
2. LogTarget.java
3. spring-aop.xml
4. TestAnnotation.java
5. LoggingAdvice.java

**package** com.test.spring.aop.annotation;

**import** java.util.Arrays;

**import** org.aspectj.lang.JoinPoint;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.aspectj.lang.annotation.After;

**import** org.aspectj.lang.annotation.AfterReturning;

**import** org.aspectj.lang.annotation.AfterThrowing;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

@Aspect

**publicclass** LoggingAdvice {

@Before("execution(\* com.test.spring.aop.annotation.LogTarget.doingThis(..))")

**publicvoid** logBefore(JoinPoint joinPoint) {

System.*out*.println("logBefore() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("\*\*\*\*\*\*");

}

@After("execution(\* com.test.spring.aop.annotation.LogTarget.doingThis(..))")

**publicvoid** logAfter(JoinPoint joinPoint) {

System.*out*.println("logAfter() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("\*\*\*\*\*\*");

}

@AfterReturning(pointcut = "execution(\* com.test.spring.aop.annotation.LogTarget.doingThisWithReturn(..))", returning = "result")

**publicvoid** logAfterReturning(JoinPoint joinPoint, Object result) {

System.*out*.println("logAfterReturning() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("Method returned value is : " + result);

System.*out*.println("\*\*\*\*\*\*");

}

@AfterThrowing(pointcut = "execution(\* com.test.spring.aop.annotation.LogTarget.doingThisError(int))", throwing = "error" )

**publicvoid** logAfterThrowing(JoinPoint joinPoint, Throwable error) {

System.*out*.println("logAfterThrowing() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("Exception Message : " + error.getMessage());

System.*out*.println("\*\*\*\*\*\*");

}

@Around("execution(\* com.test.spring.aop.annotation.LogTarget.doingThisWithReturn(..))")

**publicvoid** logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {

System.*out*.println("logAround() is running!");

System.*out*.println("advice Method name : " + joinPoint.getSignature().getName());

System.*out*.println("advice Method arguments : " + Arrays.*toString*(joinPoint.getArgs()));

System.*out*.println("Around before is running!");

joinPoint.proceed();

System.*out*.println("Around after is running!");

System.*out*.println("\*\*\*\*\*\*");

}

}

1. **LogTarget.java**

**package** com.test.spring.aop.annotation;

**publicclass** LogTarget {

**publicvoid** doingThis() {

System.*out*.println("inside doingThis()....");

}

**public**Integer doingThisWithReturn() {

System.*out*.println("doingThisWithReturn()..........");

**return** 5000;

}

**publicint** doingThisError(**int** i) **throws** Exception {

**if** (i < 5000) {

**thrownew** Exception("there is some problem....");

}

**return** i;

}

}

1. **spring-aop.xml**

<beansxmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.0.xsd "*>

<aop:aspectj-autoproxy/>

<beanid=*"target"*class=*"com.test.spring.aop.annotation.LogTarget"*/>

<beanid=*"advice"*class=*"com.test.spring.aop.annotation.LoggingAdvice"*/>

</beans>

**4. TestAnnotation.java**

**package** com.test.spring.aop.annotation;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**publicclass** TestAnnotation {

**publicstaticvoid** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-aop.xml");

LogTarget lt = (LogTarget) context.getBean("target");

System.*out*.println("start Execution.....");

//lt.doingThis();

lt.doingThisWithReturn();

/\*try {

lt.doingThisError(4000);

} catch (Exception e) {

System.out.println(e.getMessage());

}\*/

}

}

Cross-cutting concern can be

Logging

Validation

Caching

Security

Transaction

Monitoring

Error Handling etc.

## Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based**approach to implement custom aspects.

1. XML Schema base Implementation(Schema-based AOP support)

* If you prefer an XML-based format, then Spring also offers support for defining aspects using the new "aop" namespace tags.
* Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config>element in an application context configuration). An <aop:config> element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

Declaring an aspect

* An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute:

<aop:config>

<aop:aspectid="myAspect"ref="aBean">

...

</aop:aspect>

</aop:config>

<beanid="aBean"class="...">

...

</bean>

1. AspectJ Implementation

Enabling @AspectJ Support

* configuring Spring AOP based on @AspectJ aspects, and autoproxying beans
* By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.
* AspectJ’s aspectjweaver.jar library is on the classpath of your application (version 1.6.8 or later).
* The @AspectJ support can be enabled with XML or Java style configuration

Enabling @AspectJ Support with Java configuration

* To enable @AspectJ support with Java @Configuration add the @EnableAspectJAutoProxy annotation:

@Configuration

@EnableAspectJAutoProxy

**publicclass** AppConfig {

}

Enabling @AspectJ Support with XML configuration

* To enable @AspectJ support with XML based configuration use the aop:aspectj-autoproxy element:
* <aop:aspectj-autoproxy/>

Declaring an aspect

* A regular bean definition in the application context, pointing to a bean class that has the @Aspect annotation:

<beanid="myAspect"class="org.xyz.NotVeryUsefulAspect">

*<!-- configure properties of aspect here as normal -->*

</bean>

* the NotVeryUsefulAspect class definition, annotated with org.aspectj.lang.annotation.Aspect annotation;

**package** org.xyz;

**import** org.aspectj.lang.annotation.Aspect;

*@Aspect*

**publicclass** NotVeryUsefulAspect {

}

* Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

Declaring a pointcut

**Spring Data Access**

**Question:**What is the importance of DAO design pattern?

**Question:**How many ways are there to establish the connection with database server?

**Question:** What is a transaction and what is the importance of managing transaction?

**Question:** What are ACID properties?

**Question:**What are transaction concurrency problem?

**Question:** What are transaction isolation levels?

**Question:**Flat transaction vs Nested Transaction?

**Question:**Locale transaction vs distributed transaction?

**Question:**How to manage the transaction in JDBC, Hibernate, JPA and EJB?

**Question:**How to specify the required isolation levels in JDBC, Hibernate, JPA and EJB?

**Spring DAO Support:**

1. DAO design patterns main goal is to decouple persistence layer components to business layer components.
2. Spring supports DAOs in two ways:
3. Using Built in DAO classes
4. Using DataAccessExceptionAccessHieararchy
5. Using Built in DAO classes:

* DaoSupprots
* JdbcDaoSupports
* HibernateDaoSupports
* JpaDaoSupports
* IbatisDaoSupports

1. Using DataAccessExceptionAccessHieararchy
2. Spring provides specialized exception hierarchy which is to solve the problem associated with persistence provider implementation like Jdbc, Hibernate, JPA.
3. In Jdbc and Hibernate we will get always SQLException and HibernateException for any kind of underlying problem i.e. on categorization of exceptions.
4. The exception thrown by Jdbc API and HIberntate API are compile time exception so you need to report about these exceptions every time either by writing try catch repeatedly or by propagating.

* These problems are eliminated in spring with clear categorization of exceptions.
* For all exception DataAccessException is the root exception and is RuntimeException.

**SpringDataSourceSupprots**

You are established the connection with the database or data source in two ways.

1. Driver Manager Data Source:
2. JNDI Data Source:
3. Driver Manager Data Source:

To use Driver Manager Connection you need to configure the following bean in the spring-context.xml.

<bean id = “ds” class = “org.springframework.jdbc.datasourc.DriverManagerDataSource”>

<property name = “driverClassName” value = “…….”/>

<property name = “url” value = “…….”/>

<property name = “username” value = “…….”/>

<property name = “password” value = “…….”/>

</bean>

1. JNDI Data Source:

* Make sure that some application server is installed and running.
* Configure the data source in application server (assume that AppDataSourceJNDI is the name specified for the JNDI name of the Data Source).

<bean id = “ds” class = “org.springframework.jndi.JndiTemplat” />

<property name = “environment” />

<props>

<prop key = “Context.INITIAL\_CONTEXT\_FACTORY”>weblogic.jndi.WLInitialContextFactory</prop>

<prop key = “Context.PROVIDER\_URL”>t3://localhost:7001</prop>

</props>

</property>

</bean>

<bean id = “ds” class = “org.springframework.jndi.JndiObjectFactoryBean”>

<property name = “jndiName” value = “AppDataSourceJNDI”>

<property name = “jndiTemplat” ref = “jndiTemp” />

</bean>

**Spring Transaction Support**

Spring provide a uniform way to perform the transaction with various persistence technologies and frameworks.

Spring provides special transaction manager called PlateformTransactionManager (which is root for all the transaction in Spring).

* Following are the concrete implementation of PlateformTransactionManagerInterface for various persistence providers.

1. DataSourceTransactionManager
2. HibernateTransactionManager
3. JPATransactionManager

You can implement the transaction in spring in the following ways:

1. Programmatic transaction
2. Declarative transaction
3. Using TransactionProxyFactory bean
4. Using Namespace support
5. Using Annotation support

**Spring with JDBC**

**Spring with Hibernate**

**Spring with JPA**

**Spring Transaction Management**

1. When you are managing the transaction in spring you need to indentifying required transaction attributes and required isolation levels.
2. There are six transactional attributes which are provided as constants in TransactionDefinition interface follows:

PROPAGATION\_REQUIRED

PROPAGATION\_REQUIRED\_NEW

PROPAGATION\_SUPPORTS

PROPAGATION\_NOT\_SUPPORTED

PROPAGATION\_MANDATORY

PROPAGATION\_NEVER

|  |  |  |
| --- | --- | --- |
|  | New transaction state | Parent transaction state |
| REQUIRED | New transaction will be started. Your method will run in new transaction. | Your method will run in parent transaction. |
| REQUIRED\_NEW | SAME | Parent transaction will be suspended, new transaction will be started. Your method will run on new transaction. Parent transaction will be resumed. |
| SUPPORTS | Your method will run in new transaction. | Your method will run in parent transaction. |
| NOT\_SUPPORTED | Your method will run in new transaction. | Your method will run in parent transaction. |
| MANDATORY | Exception will be thrown | Your method will run in parent transaction. |
| NEVER | Your method will run in new transaction. | Exception will be thrown. |

1. To avoid the transaction concurrency problems you need to select required transaction isolation levels.
2. In Spring all the isolation levels are defined in “TransactionDefinition” interface as follows:
3. ISOLATION\_READ\_UNCOMMITED
4. ISOLATION\_READ\_COMMITED
5. ISOLATION\_REPEATABLE\_READ
6. ISOLATION\_SERIALIZABLE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Dirty Read** | **Repeatable Read** | **Phantom Read** | **Locking** |
| **ISOLATION\_READ\_UNCOMMITED** | √ | √ | √ |  |
| **ISOLATION\_READ\_COMMITED** | X | √ | √ | Column Lock |
| **ISOLATION\_REPEATABLE\_READ** | X | X | √ | Row Lock |
| **ISOLATION\_SERIALIZABLE** | X | X | X | Table Lock |

* Spring offers four different ways to implement the transaction:

1. Programmatic transaction
2. Declarative transaction using TransactionProxyBean
3. Declarative transaction using Annotation
4. Declarative transaction using namespace based
5. Programmatic transaction with Spring:
6. **With JDBC**
7. Configure data source (either DriveManager or DataSource or JNDIDataSource).
8. Configure DataSorceTransactionManager, org.springframework.jdbc.datasource and inject data source.
9. Configure JdbcTemplat or SimpleJdbcTemplat and inject data source.
10. Configure the required DAO and inject both DataSourceTransactionManager and JdbcTemplat
11. Write the DAO implementation as follows:

public class JdbcHelloDao{

@Autowired

JdbcTemplat jtemp;

@Autowired

DataSourceTransactionManager txManager;

void add(){

DefaultTransactionDefinition txdef = new DefaultTransactionDefinition();

txdef.setIslationLevel(TransactionDefinition.ISOLATION\_READ\_COMMITED);

txdef.setPropagationBehaviour(TransactionDefinition.PROPAGATION\_REQUIRES\_NEW);

TransactionState ts = null;

try{

ts = txManager.getTransaction(txdef);

……..

txManager.commint(ts);

} catch(Exception e){

txManager.rollback(ts);

}

}

1. **With Hibernate**
2. **With JPA**

**Spring Dependency Injection and Inversion of Control**

IoC is also known as *dependency injection* (DI). It is a process whereby objects define theirdependencies, that is, the other objects they work with, only through constructor arguments, arguments

to a factory method, or properties that are set on the object instance after it is constructed or returned

from a factory method. The container then *injects* those dependencies when it creates the bean. This

process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself

controlling the instantiation or location of its dependencies by using direct construction of classes, or a

mechanism such as the *Service Locator* pattern.

**BeanFactory Vs ApplicationContext**

The org.springframework.beans and org.springframework.context packages are thebasis for Spring Framework’s IoC container. The BeanFactory interface provides an advancedconfiguration mechanism capable of managing any type of object. ApplicationContext is a subinterfaceof BeanFactory. It adds easier integration with Spring’s AOP features; message resourcehandling (for use in internationalization), event publication; and application-layer specific contexts suchas the WebApplicationContext for use in web applications.

the BeanFactory provides the configuration framework and basic functionality, and theApplicationContext adds more enterprise-specific functionality. The ApplicationContext isa complete superset of the BeanFactory

**ApplicationContext**

The interface org.springframework.context.ApplicationContext represents the Spring IoCcontainer and is responsible for instantiating, configuring, and assembling the aforementioned beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by readingconfiguration metadata. The configuration metadata is represented in XML, Java annotations, or Javacode.

In standalone applications it is common to create an instance of

ClassPathXmlApplicationContext or FileSystemXmlApplicationContext.

**Instantiating a container**

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an

ApplicationContext constructor are actually resource strings that allow the container to load

configuration metadata from a variety of external resources such as the local file system, from the Java

CLASSPATH, and so on.

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

**Composing XML-based configuration metadata**

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration

file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments.

This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively,

use one or more occurrences of the <import/>element to load bean definitions from another file or

files. For example:

**<beans>**

**<import resource**=**"services.xml"/>**

**<import resource**=**"resources/messageSource.xml"/>**

**<import resource**=**"/resources/themeSource.xml"/>**

**<bean id**=**"bean1" class**=**"..."/>**

**<bean id**=**"bean2" class**=**"..."/>**

**</beans>**

external bean definitions are loaded from three files, services.xml,

messageSource.xml, and themeSource.xml. All location paths are relative to the definition file

doing the importing, so services.xml must be in the same directory or classpath location as the file

doing the importing, while messageSource.xml and themeSource.xml must be in a resources

location below the location of the importing file. As you can see, a leading slash is ignored, but given

that these paths are relative, it is better form not to use the slash at all. The contents of the files being

imported, including the top level <beans/>element, must be valid XML bean definitions according to

the Spring Schema.

**Dependency injection**

*Dependency injection* (DI) is a process whereby objects define their dependencies, that is, the other

objects they work with, only through constructor arguments, arguments to a factory method, or properties

that are set on the object instance after it is constructed or returned from a factory method. The container

then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse,

hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of

its dependencies on its own by using direct construction of classes, or the *Service Locator* pattern.

**Constructor-based dependency injection**

**package** x.y;

**public class** Foo {

**public** Foo(Bar bar, Baz baz) {

*// ...*

}

}

**<beans>**

**<bean id**=**"foo" class**=**"x.y.Foo">**

**<constructor-arg ref**=**"bar"/>**

**<constructor-arg ref**=**"baz"/>**

**</bean>**

**<bean id**=**"bar" class**=**"x.y.Bar"/>**

**<bean id**=**"baz" class**=**"x.y.Baz"/>**

**</beans>**

**package** examples;

**public class** ExampleBean {

*// Number of years to calculate the Ultimate Answer*

**private int** years;

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

**public** ExampleBean(**int** years, String ultimateAnswer) {

**this**.years = years;

**this**.ultimateAnswer = ultimateAnswer;

}

}

Explicitly specify the type of the constructor argument using the type attribute. For example:

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

**<constructor-arg type**=**"int" value**=**"7500000"/>**

**<constructor-arg type**=**"java.lang.String" value**=**"42"/>**

**</bean>**

Use the index attribute to specify explicitly the index of constructor arguments. For example:

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

**<constructor-arg index**=**"0" value**=**"7500000"/>**

**<constructor-arg index**=**"1" value**=**"42"/>**

**</bean>**

use the constructor parameter name for value disambiguation:

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

**<constructor-arg name**=**"years" value**=**"7500000"/>**

**<constructor-arg name**=**"ultimateanswer" value**=**"42"/>**

**</bean>**

**Setter-based dependency injection**

**public class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private int** i;

**public void** setBeanOne(AnotherBean beanOne) {

**this**.beanOne = beanOne;

}

**public void** setBeanTwo(YetAnotherBean beanTwo) {

**this**.beanTwo = beanTwo;

}

**public void** setIntegerProperty(**int** i) {

**this**.i = i;

}

}

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

*<!-- setter injection using the nested <ref/> element -->*

**<property name**=**"beanOne">**

**<ref bean**=**"anotherExampleBean"/>**

**</property>**

*<!-- setter injection using the neater ref* attribute -->

**<property name**=**"beanTwo" ref**=**"yetAnotherBean"/>**

**<property name**=**"integerProperty" value**=**"1"/>**

**</bean>**

**<bean id**=**"anotherExampleBean" class**=**"examples.AnotherBean"/>**

**<bean id**=**"yetAnotherBean" class**=**"examples.YetAnotherBean"/>**

**public class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private int** i;

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

**this**.beanOne = anotherBean;

**this**.beanTwo = yetAnotherBean;

**this**.i = i;

}

}

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

*<!-- constructor injection using the nested <ref/> element -->*

**<constructor-arg>**

**<ref bean**=**"anotherExampleBean"/>**

**</constructor-arg>**

*<!-- constructor injection using the neater ref* attribute -->

**<constructor-arg ref**=**"yetAnotherBean"/>**

**<constructor-arg type**=**"int" value**=**"1"/>**

**</bean>**

**<bean id**=**"anotherExampleBean" class**=**"examples.AnotherBean"/>**

**<bean id**=**"yetAnotherBean" class**=**"examples.YetAnotherBean"/>**

**Static factory method**

**public class** ExampleBean {

*// a private constructor*

**private** ExampleBean(...) {

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// some other operations...*

**return** eb;

}

}

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean" factory-method**=**"createInstance">**

**<constructor-arg ref**=**"anotherExampleBean"/>**

**<constructor-arg ref**=**"yetAnotherBean"/>**

**<constructor-arg value**=**"1"/>**

**</bean>**

**<bean id**=**"anotherExampleBean" class**=**"examples.AnotherBean"/>**

**<bean id**=**"yetAnotherBean" class**=**"examples.YetAnotherBean"/>**

The following example uses the p-namespace for even more succinct XML configuration.

**<beans xmlns**=**"http://www.springframework.org/schema/beans"**

**xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"**

**xmlns:p**=**"http://www.springframework.org/schema/p"**

**xsi:schemaLocation**=**"http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/spring-beans.xsd">**

**<bean id**=**"myDataSource" class**=**"org.apache.commons.dbcp.BasicDataSource"**

**destroy-method**=**"close"**

**p:driverClassName**=**"com.mysql.jdbc.Driver"**

**p:url**=**"jdbc:mysql://localhost:3306/mydb"**

**p:username**=**"root"**

**p:password**=**"masterkaoli"/>**

**</beans>**

**Collections**

In the <list/>, <set/>, <map/>, and <props/>elements, you set the properties and arguments of

the Java Collection types List, Set, Map, and Properties, respectively.

**<bean id**=**"moreComplexObject" class**=**"example.ComplexObject">**

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

**<property name**=**"adminEmails">**

**<props>**

**<prop key**=**"administrator">**administrator@example.org**</prop>**

**<prop key**=**"support">**support@example.org**</prop>**

**<prop key**=**"development">**development@example.org**</prop>**

**</props>**

**</property>**

*<!-- results in a setSomeList(java.util.List) call -->*

**<property name**=**"someList">**

**<list>**

**<value>**a list element followed by a reference**</value>**

**<ref bean**=**"myDataSource" />**

**</list>**

**</property>**

*<!-- results in a setSomeMap(java.util.Map) call -->*

**<property name**=**"someMap">**

**<map>**

**<entry key**=**"an entry" value**=**"just some string"/>**

**<entry key** =**"a ref" value-ref**=**"myDataSource"/>**

**</map>**

**</property>**

*<!-- results in a setSomeSet(java.util.Set) call -->*

**<property name**=**"someSet">**

**<set>**

**<value>**just some string**</value>**

**<ref bean**=**"myDataSource" />**

**</set>**

**</property>**

**</bean>**

*The value of a map key or value, or a set value, can also again be any of the following elements:*

bean | ref | idref | list | set | map | props | value | null

**public class** Foo {

**private** Map<String, Float> accounts;

**public void** setAccounts(Map<String, Float> accounts) {

**this**.accounts = accounts;

}

}

**<beans>**

**<bean id**=**"foo" class**=**"x.y.Foo">**

**<property name**=**"accounts">**

**<map>**

**<entry key**=**"one" value**=**"9.99"/>**

**<entry key**=**"two" value**=**"2.75"/>**

**<entry key**=**"six" value**=**"3.99"/>**

**</map>**

**</property>**

**</bean>**

**</beans>**

**Null and empty string values**

Spring treats empty arguments for properties and the like as empty Strings. The following XML-based

configuration metadata snippet sets the email property to the empty String value ("").

**<bean class**=**"ExampleBean">**

**<property name**=**"email" value**=**""/>**

**</bean>**

The preceding example is equivalent to the following Java code: exampleBean.setEmail(""). The

<null/>element handles null values. For example:

**<bean class**=**"ExampleBean">**

**<property name**=**"email">**

**<null/>**

**</property>**

**</bean>**

**XML shortcut with the p-namespace**

**<beans xmlns**=**"http://www.springframework.org/schema/beans"**

**xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"**

**xmlns:p**=**"http://www.springframework.org/schema/p"**

**xsi:schemaLocation**=**"http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/spring-beans.xsd">**

**<bean name**=**"classic" class**=**"com.example.ExampleBean">**

**<property name**=**"email" value**=**"foo@bar.com"/>**

**</bean>**

**<bean name**=**"p-namespace" class**=**"com.example.ExampleBean"**

**p:email**=**"foo@bar.com"/>**

**</beans>**

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring

to include a property declaration. As previously mentioned, the p-namespace does not have a schema

definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

**<beans xmlns**=**"http://www.springframework.org/schema/beans"**

**xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"**

**xmlns:p**=**"http://www.springframework.org/schema/p"**

**xsi:schemaLocation**=**"http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/spring-beans.xsd">**

**<bean name**=**"john-classic" class**=**"com.example.Person">**

**<property name**=**"name" value**=**"John Doe"/>**

**<property name**=**"spouse" ref**=**"jane"/>**

**</bean>**

**<bean name**=**"john-modern"**

**class**=**"com.example.Person"**

**p:name**=**"John Doe"**

**p:spouse-ref**=**"jane"/>**

**<bean name**=**"jane" class**=**"com.example.Person">**

**<property name**=**"name" value**=**"Jane Doe"/>**

**</bean>**

**</beans>**

**XML shortcut with the c-namespace**

Similar to the the section called “XML shortcut with the p-namespace”, the *c-namespace*, newly

introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments

rather then nested constructor-arg elements.

Let’s review the examples from the section called “Constructor-based dependency injection” with the

c: namespace:

<beans xmlns=***"http://www.springframework.org/schema/beans"***

xmlns:xsi=***"http://www.w3.org/2001/XMLSchema-instance"***

xmlns:c=***"http://www.springframework.org/schema/c"***

xsi:schemaLocation=***"http://www.springframework.org/schema/beans***

http:*//www.springframework.org/schema/beans/spring-beans.xsd">*

<bean id=***"bar"* class**=***"x.y.Bar"***/>

<bean id=***"baz"* class**=***"x.y.Baz"***/>

<!-- traditional declaration -->

<bean id=***"foo"* class**=***"x.y.Foo"***>

<constructor-arg ref=***"bar"***/>

<constructor-arg ref=***"baz"***/>

<constructor-arg value=***"foo@bar.com"***/>

</bean>

<!-- c-namespace declaration -->

<bean id=***"foo"* class**=***"x.y.Foo"*** c:bar-ref=***"bar"*** c:baz-ref=***"baz"*** c:email=***"foo@bar.com"***/>

</beans>

The c: namespace uses the same conventions as the p: one (trailing -ref for bean references) for

setting the constructor arguments by their names. And just as well, it needs to be declared even though

it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was

compiled without debugging information), one can use fallback to the argument indexes:

<!-- c-namespace index declaration -->

<bean id=***"foo"* class**=***"x.y.Foo"*** c:\_0-ref=***"bar"*** c:\_1-ref=***"baz"***/>

**Compound property names**

**<bean id**=**"foo" class**=**"foo.Bar">**

**<property name**=**"fred.bob.sammy" value**=**"123" />**

**</bean>**

The foo bean has a fred property, which has a bob property, which has a sammy property, and that final

sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the

bob property of fred must not be null after the bean is constructed, or a NullPointerException

is thrown.

**Using depends-on**

If a bean is a dependency of another that usually means that one bean is set as a property of another.

Typically you accomplish this with the <ref/>element in XML-based configuration metadata. However,

sometimes dependencies between beans are less direct; for example, a static initializer in a class needs

to be triggered, such as database driver registration. **The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.** The following example uses the depends-on attribute to express a dependency on a single bean:

**<bean id**=**"beanOne" class**=**"ExampleBean" depends-on**=**"manager"/>**

**<bean id**=**"manager" class**=**"ManagerBean" />**

To express a dependency on multiple beans, supply a list of bean names as the value of the dependson

attribute, with commas, whitespace and semicolons, used as valid delimiters:

**<bean id**=**"beanOne" class**=**"ExampleBean" depends-on**=**"manager,accountDao">**

**<property name**=**"manager" ref**=**"manager" />**

**</bean>**

**<bean id**=**"manager" class**=**"ManagerBean" />**

**<bean id**=**"accountDao" class**=**"x.y.jdbc.JdbcAccountDao" />**

**Lazy-initialized beans**

By default, ApplicationContext implementations eagerly create and configure all singleton beans

as part of the initialization process.

When this behavior is *not* desirable, you can prevent pre-instantiation of a singleton bean by

marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a

bean instance when it is first requested, rather than at startup.

**<bean id**=**"lazy" class**=**"com.foo.ExpensiveToCreateBean" lazy-init**=**"true"/>**

**<bean name**=**"not.lazy" class**=**"com.foo.AnotherBean"/>**

You can also control lazy-initialization at the container level by using the default-lazy-init attribute

on the <beans/>element; for example:

**<beans default-lazy-init**=**"true">**

*<!-- no beans will be pre-instantiated... -->*

**</beans>**

**Autowiring collaborators**

When using XML-based configuration metadata 10, you specify autowire mode for a bean definition

with the autowire attribute of the <bean/>element. The autowiring functionality has five modes.

|  |  |
| --- | --- |
| **Mode** | **Explanation** |
| No | (Default) No autowiring. |
| byname | Autowiring by property name. Spring looks for  a bean with the same name as the property  that needs to be autowired. For example, if a  bean definition is set to autowire by name, and  it contains a *master* property (that is, it has a  *setMaster(..)* method), Spring looks for a bean  definition named master, and uses it to set the  property. |
| byType | Allows a property to be autowired if exactly one  bean of the property type exists in the container.  If more than one exists, a fatal exception is  thrown, which indicates that you may not use  *byType* autowiring for that bean. If there are no  matching beans, nothing happens; the property  is not set |
| constructor | Analogous to *byType*, but applies to constructor  arguments. If there is not exactly one bean of  the constructor argument type in the container, a fatal error is raised. |
|  |  |

**Limitations and disadvantages of autowiring**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general,

it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

• Explicit dependencies in property and constructor-arg settings always override autowiring.

You cannot autowire so-called *simple* properties such as primitives, Strings, and Classes (and

arrays of such simple properties). This limitation is by-design.

• Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful

to avoid guessing in case of ambiguity that might have unexpected results, the relationships between

your Spring-managed objects are no longer documented explicitly.

• Wiring information may not be available to tools that may generate documentation from a Spring

container.

• Multiple bean definitions within the container may match the type specified by the setter method

or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily

a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily

resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

• Abandon autowiring in favor of explicit wiring.

• Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false as

described in the next section.

• Designate a single bean definition as the *primary* candidate by setting the primary attribute of its

<bean/>element to true.

As of Spring 3.2 it is no longer necessary to add CGLIB to your

classpath, because CGLIB classes are repackaged under org.springframework and distributed

within the spring-core JAR. This is done both for convenience as well as to avoid potential

conflicts with other projects that use differing versions of CGLIB.

**Bean scopes**

Spring Framework supports five scopes, three of which are available only if you use a webaware

ApplicationContext.

|  |  |
| --- | --- |
| **Scope** | **Description** |
| Singleton | (Default) Scopes a single bean definition to a  single object instance per Spring IoC container. |
| prototype | Scopes a single bean definition to any number of object instances. |
| Request | Scopes a single bean definition to the lifecycle  of a single HTTP request; that is, each HTTP  request has its own instance of a bean created  off the back of a single bean definition. Only  valid in the context of a web-aware Spring  ApplicationContext. |
| Session | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global session | Scopes a single bean definition to the lifecycle  of a global HTTP Session. Typically only  valid when used in a portlet context. Only  valid in the context of a web-aware Spring  ApplicationContext. |





**the Class name substitution PropertyPlaceholderConfigurer**

the PropertyPlaceholderConfigurer to externalize property values from a bean

definition in a separate file using the standard Java Properties format. Doing so enables the person

deploying an application to customize environment-specific properties such as database URLs and

passwords, without the complexity or risk of modifying the main XML definition file or files for the

container.

**<bean class**=**"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">**

**<property name**=**"locations" value**=**"classpath:com/foo/jdbc.properties"/>**

**</bean>**

**<bean id**=**"dataSource" destroy-method**=**"close"**

**class**=**"org.apache.commons.dbcp.BasicDataSource">**

**<property name**=**"driverClassName" value**=**"${jdbc.driverClassName}"/>**

**<property name**=**"url" value**=**"${jdbc.url}"/>**

**<property name**=**"username" value**=**"${jdbc.username}"/>**

**<property name**=**"password" value**=**"${jdbc.password}"/>**

**</bean>**

The actual values come from another file in the standard Java Properties format:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

You can use the PropertyPlaceholderConfigurer to substitute class names, which is

sometimes useful when you have to pick a particular implementation class at runtime. For

example:

**<bean class**=**"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">**

**<property name**=**"locations">**

**<value>**classpath:com/foo/strategy.properties**</value>**

**</property>**

**<property name**=**"properties">**

**<value>**custom.strategy.class=com.foo.DefaultStrategy**</value>**

**</property>**

**</bean>**

**<bean id**=**"serviceStrategy" class**=**"${custom.strategy.class}"/>**

<?xml version="1.0" encoding="UTF-8"?>

**<beans xmlns**=**"http://www.springframework.org/schema/beans"**

**xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"**

**xmlns:context**=**"http://www.springframework.org/schema/context"**

**xsi:schemaLocation**=**"http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/spring-beans.xsd**

**http://www.springframework.org/schema/context**

**http://www.springframework.org/schema/context/spring-context.xsd">**

**<context:annotation-config/>**

**</beans>**

<context:annotation-config/>only looks for annotations on beans in the same

application context in which it is defined. This means that, if you put <context:annotationconfig/>

in a WebApplicationContext for a DispatcherServlet, it only checks for

@Autowired beans in your controllers, and not your services. See Section 16.2, “The

DispatcherServlet” for more information.

**@Required**

The @Required annotation applies to bean property setter methods, as in the following example:

**public class** SimpleMovieLister {

**private** MovieFinder movieFinder;

@Required

**public void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

This annotation simply indicates that the affected bean property must be populated at configuration time,

through an explicit property value in a bean definition or through autowiring. The container throws an

exception if the affected bean property has not been populated; this allows for eager and explicit failure,

avoiding NullPointerException s or the like later on. It is still recommended that you put assertions

into the bean class itself, for example, into an init method. Doing so enforces those required references

and values even when you use the class outside of a container.

===========================

Spring is an open source framework, created by Rod Johnson and described in his

book *Expert One-on-One: J2EE Design and Development.*

*Lightweight*—Spring is lightweight in terms of both size and overhead. The

bulk of the Spring Framework can be distributed in a single JAR file that

weighs in at just over 2.5 MB.

*Dependency Injection*—Spring promotes loose coupling through a technique

known as dependency injection (DI). When DI is applied, objects are passively

given their dependencies instead of creating or looking for dependent

objects for themselves. You can think of DI as JNDI in reverse—instead

of an object looking up dependencies from a container, the container gives

the dependencies to the object at instantiation without waiting to be asked.

*Aspect-oriented*—Spring comes with rich support for aspect-oriented programming

(AOP) that enables cohesive development by separating application

business logic from system services (such as auditing and

transaction management). Application objects do what they’re supposed

to do—perform business logic—and nothing more. They are not responsible

for (or even aware of) other system concerns, such as logging or transactional

support.

*Container*—Spring is a container in the sense that it contains and manages

the lifecycle and configuration of application objects. In Spring, you can

declare how each of your application objects should be created, how they

should be configured, and how they should be associated with each other.

■*Framework*—Spring makes it possible to configure and compose complex

applications from simpler components. In Spring, application objects are

composed declaratively, typically in an XML file. Spring also provides much

infrastructure functionality (transaction management, persistence framework

integration, etc.), leaving the development of application logic to you.



The core module’s

BeanFactory makes Spring a container, but the context module is what makes it a

framework. This module extends the concept of BeanFactory, adding support for

internationalization (I18N) messages, application lifecycle events, and validation.

When applying DI, objects are given their

dependencies at creation time by some external

entity that coordinates each object in the system.

The key benefit of DI is loose coupling. If an object only knows about its

dependencies by their interface (not their implementation or how they were

instantiated) then the dependency can be swapped out with a different implementation

without the depending object knowing the difference.

For example, if the Foo class in figure 1.2 only knows about its Bar dependency

through an interface then the actual implementation of Bar is of no importance

to Foo. Bar could be a local POJO, a remote web service, an EJB, or a mock implementation

for a unit test—Foo doesn’t need to know or care.

Spring container comes into two distinct types:

1. Bean factories

(defined by the org.springframework.beans.factory.BeanFactory interface)

1. Application contexts

(defined by the org.springframework.context.ApplicationContext interface)

In a Spring application, objects are

created, wired together, and live within the

Spring container.

There are several implementations of BeanFactory in Spring. But the one that

is most commonly used is org.springframework.beans.factory.xml.XmlBean-

Factory, which loads its beans based on the definitions contained in an XML file.

BeanFactory factory =

new XmlBeanFactory(new FileSystemResource("c:/beans.xml"));

Resource implementation Purpose

org.springframework.core.io.ByteArray

Resource

Defines a resource whose content is given by an array of bytes

org.springframework.core.io.ClassPath

Resource

Defines a resource that is to beretrieved from the classpath

org.springframework.core.io.Descriptive

Resource

Defines a resource that holds a resource description but no actual readable resource

org.springframework.core.io.FileSystem

Resource

Defines a resource that is to be retrieved from the file system

org.springframework.core.io.InputStream

Resource

Defines a resource that is to be retrieved from an input stream

org.springframework.web.portlet.context.

PortletContextResource

Defines a resource that is available in a portlet context

org.springframework.web.context.support.

ServletContextResource

Defines a resource that is available in a servlet context

org.springframework.core.io.UrlResource Defines a resource that is to be retrieved from a given URL

For ApplicationContext there are three that are

commonly used:

■ClassPathXmlApplicationContext—Loads a context definition from an

XML file located in the classpath, treating context definition files as classpath

resources.

■FileSystemXmlApplicationContext—Loads a context definition from an

XML file in the file system.

■XmlWebApplicationContext—Loads context definitions from an XML file

contained within a web application

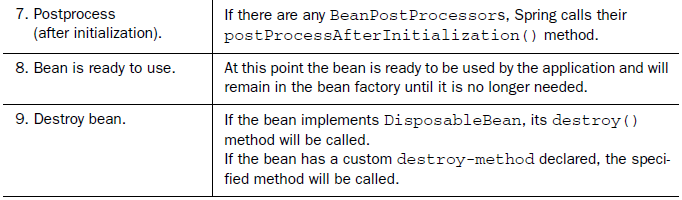
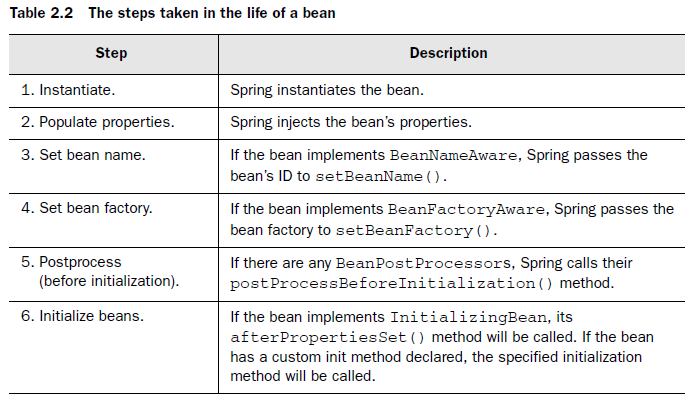
ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");

ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");

FileSystemXmlApplicationContextwill look for foo.xml in a specific location within the file system, whereas Class-PathXmlApplicationContext will look for foo.xml anywhere in the classpath(including JAR files).

the ApplicationContext interface extends the BeanFactory interface.

you can retrieve a bean by using the getBean() method.



The only difference between BeanFactory and ApplicationContext life cycle is that if the bean implements the ApplicationContextAware interface, the setApplicationContext() method is called.

<bean id="duke" class="com.springinaction.springidol.Juggler" />

The id attribute gives the bean a name by which it will be referred to in the Spring container. This bean will be known as duke. Meanwhile, the class attribute tells Spring what type the bean will be.

## Types of Advice

Spring aspects can work with five kinds of advice mentioned below:

|  |  |
| --- | --- |
| **Advice** | **Description** |
| before | Run advice before the a method execution. |
| after | Run advice after the a method execution regardless of its outcome. |
| after-returning | Run advice after the a method execution only if method completes successfully. |
| after-throwing | Run advice after the a method execution only if method exits by throwing an exception. |
| around | Run advice before and after the advised method is invoked. |

## Custom Aspects Implementation

Spring supports the **@AspectJ annotation style** approach and the **schema-based**approach to implement custom aspects. These two approaches have been explained in detail in the following two sub chapters

|  |  |
| --- | --- |
| **Approach** | **Description** |
| [**XML Schema based**](http://www.tutorialspoint.com/spring/schema_based_aop_appoach.htm) | Aspects are implemented using regular classes along with XML based configuration. |
| [**@AspectJ based**](http://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm) | @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. |

https://www.logicbig.com/tutorials/spring-framework/spring-core/injecting-collections.html