Acceptable Use Policy

The information contained in this document is **private** and any usage of this document is governed by its acceptable use policy.

All the data collected as inputs and any data generated as outputs during this documentation is classified **private**. Such data includes both electronic and physical information.

It is deemed that you agree to all the terms associated with this document usage before using/reading any further.

Spring Framework:

IsanOpensource,lightweight, multi-tier EnterpriseApplicationframework,addressingmostoftheinfrastructuralconcernsincreatinganenterpriseapplicationusingjava.

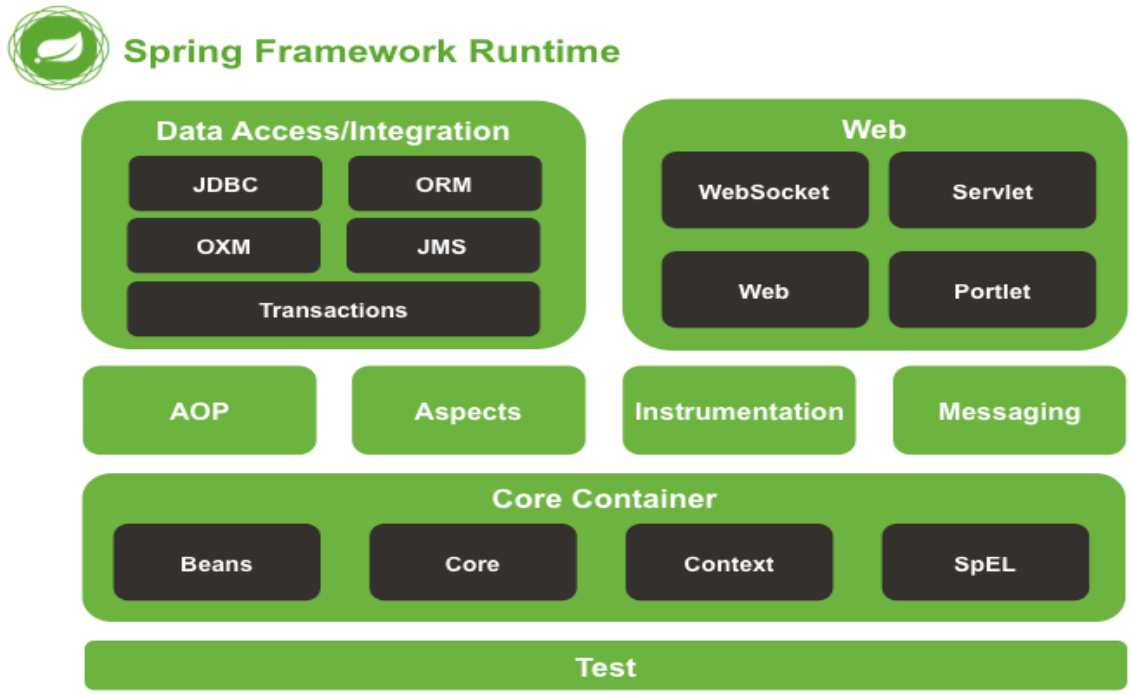
The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications. However, spring is modular, allowing you to use only those parts that you need, without having to bring in the rest

Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself.

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs.

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, and Test, as shown in the following diagram.



The IoC(Inversion of Controller) container

This module of the spring framework is an implementation of IOC Design Pattern / Principle.

The **org.springframework.beans** and **org.springframework.context** packages are the basis for Spring Framework's IoC container.

The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object.

The **ApplicationContext** is a subinterface of **BeanFactory**. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the **WebApplicationContext** for use in web applications.

The **BeanFactory** provides the configuration framework and basic functionality, and the **ApplicationContext** adds more enterprise-specific functionality.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

What is IOC

IOCisanArchitecturaldesignPrinciple,thatdescribestohaveanexternalentitytocreate and wiretheobjects. Herethe wiringis donebyusingaconceptDI(Dependency Injection (pushingapproach))

What is DI

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used. An injection is the passing of a dependency to a dependent object that would use it.

Container overview

The interface ***org.springframework.context.ApplicationContext*** represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the mentioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the ApplicationContext interface are supplied out-of-thebox with Spring. In standalone applications it is common to create an instance of ***ClassPathXmlApplicationContext*** or ***FileSystemXmlApplicationContext***. While XML has been the traditional format for defining configuration metadata, you can instruct the container to use java annotations or java config code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

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 =
        newClassPathXmlApplicationContext(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>
```

Note: All location paths are relative to the definition file doing the importing

Using the container to Access Bean

The ApplicationContext is the interface for an advanced factory capable of maintaining a registryof different beans and their dependencies. Using the method T getBean(String name, Class<T>requiredType) you can retrieve instances of your beans.

Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application.

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

DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number ofarguments, each representing a dependency. Calling a static factory method with specific arguments to construct the bean is nearly equivalent,

Constructor argument resolution

Constructor argument resolution matching occurs using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated

Setter-based dependency injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a constructor or static factory method to instantiate your bean.

Example – IoC& DI in action

```
//AccountDAOI.java
package com.pratap;
publicinterface AccountDAOI {
      publicvoid setBalance(intaccno , doublebal );
      publicdouble getBalance(intaccno);
}
```

```
//AccountDAO.java
package com.pratap;
publicclass AccountDAO implements AccountDAOI {
      publicvoid setBalance(intaccno, doublebal) {
            System.out.println("setting Balace : "+bal +
                                          " for accno :"+accno);
      }
      publicdouble getBalance(intaccno) {
            System.out.println("Getting Balance for :"+accno);
            return 1000;
      }
}
```

```
//CheckMinBal.java
package com.pratap;
publicclass CheckMinBal {
	privatedoubleminBalance;
	public CheckMinBal(doubleminBalance) {
		this.minBalance = minBalance;
	}
	publicboolean check( doublebalance ) {
		System.out.println("in chek for balance " +balance);
		if(balance>minBalance ) {
			returntrue;
		}
		returnfalse;
	}
}
```

```
//WithDrawBO.java
package com.pratap;
publicclass WithdrawBO {
	private AccountDAOI dao;
	private CheckMinBal cmb;
	public WithdrawBO( AccountDAOI dao ) {
		this.dao = dao;
	}
	publicvoid setCheckMinBal(CheckMinBal cmb ) {
		this.cmb = cmb;
	}
	publicvoid withdraw( intaccno , doubleamt ) {
		System.out.println("Hello , from withdraw.....");
		doublecurrBal = dao.getBalance(accno);
		doubleremainBal = currBal - amt; // 800

		if(cmb.check(remainBal)) {
			System.out.println("Withdrawing....");
		}else {
			System.out.println("Can't Withdraw , not satisfying
 minBalance criteria after withdraw");
		}
	}
}
```

```
//spring-bean.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.xsd">

      <beanid="accDao"class="com.pratap.AccountDAO"/>

      <beanid="cmb"class="com.pratap.CheckMinBal">
            <constructor-arg>
                  <value>500</value>
            </constructor-arg>
      </bean>

      <beanid="wbo"class="com.pratap.WithdrawBO">
            <constructor-arg>
                  <refbean="accDao"/>
            </constructor-arg>
            <propertyname="checkMinBal">
                  <refbean="cmb"/>
            </property>
      </bean>
</beans>
```

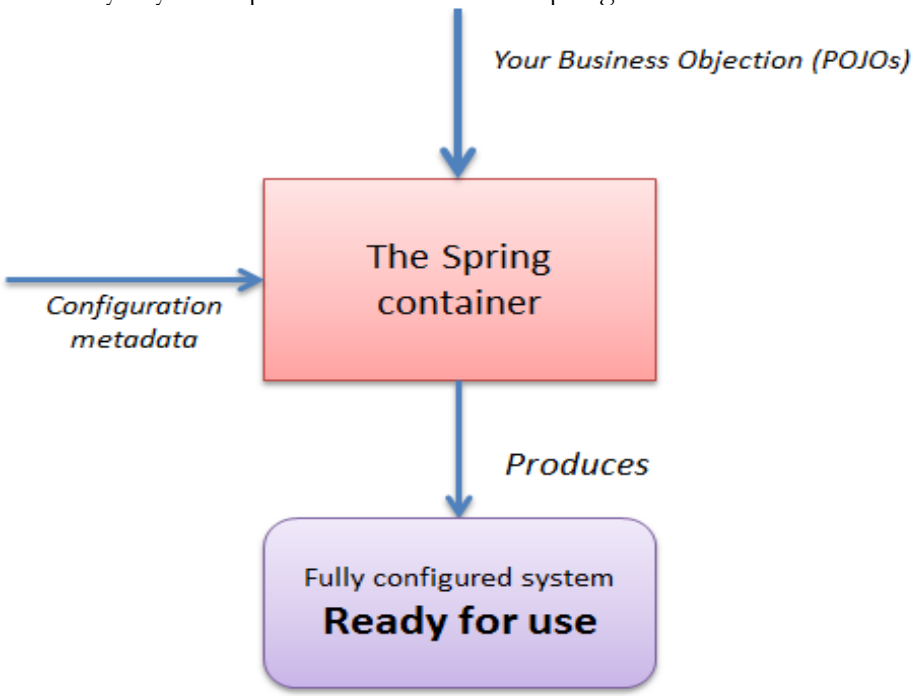
```
//App.java
package com.pratap;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class App {
      public static void main(String[] args) {
            ApplicationContext container = new
ClassPathXmlApplicationContext("com/pratap/spring-beans.xml");

            WithdrawBO wbo = (WithdrawBO) container.getBean("wbo");
            wbo.withdraw(1234, 200);
      }
}
```

Configuration metadata

As the following diagram shows, the Spring IoC container consumes a form of configuration metadata; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is used to convey key concepts and features of the Spring IoC container.



XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written.

Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.

Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus you can define beans external to yourapplication classes by using Java rather than XML files.

Configuration metadata using xml

From Spring2.0,thespringbeansxml document elementsaredefinedusingXMLSchema.XSD(XML Schema Definition)

TheBasicsetoftagthatcanbe usedtocreatethisdocument describedinspring-beans-version.xsdschema document.

WeAccessthisschemausingthefollowingnamespace.

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

Theassociated schemadocument (XSD)is
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

TheXSD
forthebeansnamespacesisavailablein***org.springframework.beans.factory.xml***packageofspring-beans-xxxjarfile.

The<beans> tag

Thisisarootelement/documentelementforthespringbeansxmldocument.

Thissupportstodefinesindividualspringbean , importotherspringbeandefinitionfile ,supportstoprovidesaliasforthespringbean.

Thefollowingchildtagsaresupported….

- <bean>*(0ormoretimes)
- <alias>*
- <import>*
- <description>
- <beans>

Inaddition,The<beans> tagsupportsfewattributesforconfiguringthedefaultvaluefortherespective parameterforallthebeandefinedinthisdocument.

The<bean>tag

Thisallowsdefiningaspringbeandefinition,this definition carriestheconcreteinformationforthe springcontainersuchthatthecontainercancreateandmanagethespringbean.

Naming beans
Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).

SpringBean

Ajavaobjectcreatedandmanagedbythespringcorecontainer.

Creatingand Managingaspringbeaninvolvesthefollowings…

- Instantiateajava objectforthisobject.
- Initializethejavaobject.
- Managethelifeoftheobject.

Springcorecontainersupports3differentstyletoInstantiateajavaobject

1. usingwithclassconstructor(onlyclassattribute)

2. usingstaticfactorymethod(class+factory-methodattribute)
3. usingnon-staticfactorymethod(factory-bean+factory-methodattribute)

Instantiation with a constructor

Thisdescribethespringcontainertoinstantiatespringbeanusingthe class constructor, Tospecifythecontainerabouttheisstyle, weneedtouse"class"attributeofthe <bean>element.

```
public class Test{
}
 Test t1 = new Test(); // core java programming
<bean id="t1" class="Test"/>          //it will use new keyword with no-arg constructor
```

Theclassattributeisafullyqualifiedclass name, whoseconstructor need to be used to createthe object.

wecandefinemultiplespringbeaninthe containercontext.wemayneedtospecifyauniqueidentityforeach springbeanforthefurtherreference.

weuseidand/ ornameattributetospecifytheidentity ofthe springbean , idattributeacceptonly onevalue ,howeverwecanspecifyaliasnameforthe springbeanusingthenameattribute.

```
<bean id="t" name="t1 , t2 ; t3 t4 " class="com.Test" />
```

Howtodefinetheconstructor?

```
class Test{
        public Test(int x){
        }
        public Test(String []s , int x){
        }
}
```

Weusethe<constructor-arg>tagtodescribeabouttheargumentofconstructor.

The<constructor-arg>tagisusedtodescribeaboutasingleconstructorargument, butnotacompleteconstructordefinition.

Tosatisfythe mostcommonrequirementofdifferentdatatypepossible,the<constructor-arg>supportsvariouschildelementssuchas…
<value> ,<ref>, <idref>, <null> , <array> , <list> , <set> , <map> , <props><bean> , <description>

<value>tag

The<value>isusedtodescribeanytypeofvalue, providedacorrespondingPropertyEditorneedsto bedefined.

We often use <value>tagforjava primitive, String… type.

However the <value>tag is intelligentto resolve various other type using the following built-in editors, we can extends <value> tag capabilities by writing our own custom property editor implementation.

Built-inProperty editors

- ✓ ByteArrayPropertyEditor
- ✓ ClassEditor
- ✓ CustomBooleanEditor
- ✓ CustomCollectionEditor
- ✓ CustomDateEditor
- ✓ CustomNumberEditor
- ✓ FileEditor
- ✓ InputStreamEditor
- ✓ LocaleEditor
- ✓ PatternEditor
- ✓ PropertiesEditor

```
class Test{
int x;
        public Test( int  x ){
        this. x = x;
}
}

<bean id=”t1” class=”Test” >
        <constructor-arg>
                <value type =”int”> 100 </value>
        </constructor-arg>
</bean>

<bean id=”t1” class=”Test” >
        <constructor-arg>
                <value type =”Address”>101 , ameerpet , AP </value>
        </constructor-arg>
</bean>

<bean … >
<constructor-arg>
<value type=”java.net.URL”>www.google.com</value>
<constructor-arg>
        </bean>
```

The<value>acceptstheStringasbodycontent,andcanconvertthemintospecifiedtypebyusingtheproperty editor.

The spring built-in editors are organized into this package **org.springframework.beans.propertyeditors**
Definingaproperty editorincludestocreateajavaclassbyimplementing **java.beans.PropertyEditor**interface.

WecanusethePropertyEditorSupportadaptorclasstominimizeourimplementation.

```
public class ExoticTypeEditor extends PropertyEditorSupport {
public void setAsText(String text) {
setValue(new ExoticType(text.toUpperCase()));
}
        }
```

Finally, we use CustomEditorConfigurer to register the new PropertyEditor with the Application Context, which will then be able to use it as needed:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
<property name="customEditors">( setter method )
<map>
<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
// key : Target Type Name
// Value : PropertyEditor Implementation class Name
</map>
</property>
        </bean>
```

```
package example;
public class ExoticType {
private String name;
public ExoticType(String name) {
this.name = name;
}
}
```

```
public class DependsOnExoticType {
private ExoticType type;
public void setType(ExoticType type) {
this.type = type;
}
        }
```

```
<bean id="sample" class="example.DependsOnExoticType">
<property name="type" value="aNameForExoticType"/>
</bean>

<bean id="sample" class="example.DependsOnExoticType">
<property name="type">
<value>aNameForExoticType</ value>
</property>
</bean>
<bean id="mappings"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="properties">
<value>
jdbc.driver.className=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mydb
</value>
</property>
</bean>

<bean id="mappings"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<property name="properties">
<props>
        <propkey=”jdbc.driver.className”>Value 1<prop>
<propkey=”jdbc.url”>Value 2<prop>
</props>
</property>
</bean>
```

<ref>tag

- describesaspringbeanreference(collaborator)
- itisanemptytag.
- supports3attributes.
    - local (removedfromspring 4.0)
    - bean
    - parent

```
classStudent{
        Admissionad;
        public Student(Admissionad){
                this.ad = ad;
        }
}
<beanid=”ad”name=”ad1”class=”Admission”/>
```

```
<beanid=”st”class=”Student”>
        <constructor-arg>
                <refbean=”ad” />
        </constructor-arg>
</bean>
```

Local attribute

Thelocalattributerefers thetargetbeanonlywithinthecurrentxmlfile.

Bean attribute

Specifying the target bean through the bean attribute of the <ref/> tag is the most general form, and allows creation of a reference to any bean in the same container or parent container context

Parent attribute

Specifying the target bean through the parent attribute creates a reference to a bean that is in a parent container of the current container. The value of the parent attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean, and the target bean must be in a parent container of the current one.

<null> tag

Describesanullvalue ,Thisisusefultorefertheconstructorargumentforwhich thevalue isnotavailable.
Isanemptytagwithoutanyattribute.

```
classTest{
        Test(intx,Strings,Objecto){
}
}
Testt=newTest(10, “hello” , null );
```

Usingspring

```
<beanid=”t”class=”Test”>
        <constructor-arg>
                <value>10</value>
        </constructor-arg>
        <constructor-arg>
                <value>Hello</value>
        </constructor-arg>
        <constructor-arg>
                <null />
        </constructor-arg>
</bean>
```

<list>tag

usedtodescribeanarrayorjava.util.Listtypeofobjectasaconstructorargument.

The<list>tagsupports0ormorechildelements.
Thefollowingchildelementsaresupported…

<value> , <ref> ,<idref> ,<null> , <list> , <set> , <map> , <props> , <array> , <bean>

EachchildtagdescribesaboutoneelementoftheList.

```
classTestBean{
        Listlist;
        TestBean(Listlist){
                this.list = list;
                s.o.p(list);
        }
}

<beanid=”tb”class=”TestBean”>
        <constructor-arg>
                <list>
                        <value>10</value>
                        <value>20</value>
                        <value>30</value>
                </list>
        </constructor-arg>
</bean>
```

<listmerge="default"value-type="">
Note :Springcreatea java.util.ArrayList by default.
The“value-type”attribute is used to specify the type of value within list collection.
<listvalue-type="java.lang.String">

Note :wecan’t specify the List Collectiontype with this <list>tag.

For more precisecontrol over the List object creation (type) , use <util:list> tag instead.

<set>tag
The<set>tagisusedtodescribejava.util.Settypeofobjectas aconstructorargument.

Thistagsupportallthechildelementssupportedby<list>tag.

```
classTest{
        Setset;
        publicTest( Setset){
                this.set = set;
        }
}

<beanid=”t”class=”Test”>
        <constructor-arg>
                <set>
```

```
                    <value>one</value>
                    <value>two</value>
                    <value>three</value>
                    <value>one<value>
                </set>
        </constructor-arg>
</bean>

<set merge="default"value-type="">

Defaultsetimplementationusedisjava.util.LinkedHashSet
```

<map>tag:

```
The<map>tagisusedtodescribejava.util.Maptypeofobject.
map:acollectionofentries,whereeachentry isa key- valuepair.
The<map>tagsupport0ormore<entry>tag.
The<entry>tagsupportstodefineakeyandavalueforthisentry.
weuse<key>tagtodefinetheentrykey.
The<key>Tagsupportthesamechildtaglisttodefinetheentrykey .
weuseanyofthe childtagsupportedby<list>tagtodescribeabouttheentryvalue.

classTest{
        Mapmap;
        publicTest(Mapmap){
                this. map = map ;
        }
}

The<entry> tagsupports the following attribute.
<entry key="" key-ref="" value=""value-ref=""value-type="">

<entrykey=”key1”value=”value1”/>
<entrykey-ref=”k1” value=”v1”/>

<beanid=”t”class=”Test”>
        <constructor-arg>
                <map>
                        <entry>
                                <key>
                                        <value>key1</value>
                                </key>
                                        <value>value1</value>
                        </entry>
                        <entry>
                                <key>
                                        <value>key2</value>
```

```
                    </key>
                        <value>value2</value>
                </entry>
            </map>
        </constructor-arg>
</bean>
```

<map merge="default" key-type="" value-type="">
Default Mapimplementation usedis java.util.LinkedHashMap

<props>tag:

The<props> tagisusedto describejava.util.Propertiestypeofobject,wherethekeyandvalueareconsideredasString.

The<props>tagsupportstohave0ormore<prop>tagaschildelement.

```
classTest{
    Propertiesps;
    Test(Propertiesps){
        this. ps = ps;
    }
}
```

The <props> supports the attribute
<props merge="default" value-type="">

The merger attribute can have the following value

- Default
- True
- false

```
<beanid=”t”class = “Test”>
    <constructor-arg>
        <props>
            <propkey=”key1”>value1</prop>
            <propkey=”key2”>value2</prop>
        </props>
    </constructor-arg>
</bean>
```

<array>tag :

Thisisusedtodescribeanjavaarrayastheconstructorargument.

```
classTest{
    intx;
    String[]items;
    publicTest(intx,String [] items){
```

```
                this.x = x;
                this.items = items;
        }
}

<beanid=”t1”class=”Test”>
        <constructor-arg>
                <valuetype=”int”>10</value>
        </constructor-arg>
        <constructor-arg>
                <array>
                        <value>one</value>
                        <value>two</value>
                </array>
        </constructor-arg>
</bean>
```

<bean>tag:(Innerbean)

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean.

An inner bean definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag. Inner beans are always anonymous and they are always created with the outer bean. It is not possible to inject inner beans into collaborating beans other than into the enclosing bean.

```
<beanid="tb6"class="com.TestBean">
        <constructor-arg>
                <array>
                        <bean class="com.Student">
                                <constructor-arg>
                                        <value>1001</value>
                                </constructor-arg>
                                <constructor-arg>
                                        <value>Dipankar</value>
                                </constructor-arg>
                        </bean>
                        <beanclass="com.Student">
                                <constructor-arg>
                                        <value>2002</value>
                                </constructor-arg>
                                <constructor-arg>
                                        <value>Santosh</value>
                                </constructor-arg>
                        </bean>
                </array>
```

</constructor-arg>
</bean>

AttributesSupportedby<constructor-arg>:

Alongwiththeabovementionedchildelements / tags,The<constructor-arg>elements supportvariousattributes,suchas…
value(short cutof<value>childtag)
ref(shortcutof<ref>childtag)
type(to define type of the value ,avoidambiguities)
name(avoidambiguities)
index(avoidambiguities)

Value: Thisattributeofthe< constructor-arg>describesthisconstructorargumentvalue.
This canbeusedasashortcutof the<value>childtag.

Ref: Thisattributeisusedasashortcutforthe<ref>childelement .

The type,name andindex attribute ofthe<constructor-arg>isusedtoavoid theambiguities in constructorresolution.

Type: The exact type of the constructor argument. Only needed to avoid ambiguities, e.g. in case of 2 single argument constructors that can both be converted from a String.

Index: Thisattributesupporttodefinetheindexpositionofthisconstructorargument.
Theindexposition startwith 0
In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the index is 0 based.

Name: Thisattributeofthe<constructor-arg>allowstousetheconstructorargumentname.
Thisattributecanbeusedwhenthesourcecodeiscompiledwithdebugflagenabled.

Javac-gSourcefile.java(-gis for enabling debugging info)

wecanevenuse@ConstructorPropertiesjdkannotationtodefineconstructorargumentname.

Instantiation with a static factory method

```
classTest{
}

Testt1=newTest();

<beanid =”t1”class=”Test”>
</bean>
```

```
classFactory {
        publicstaticTestgetTest(){
}
}

Testt2= Factory.getTest();

<beanid=”t2”class =”Factory”factory-method=”getTest”/>

classMyFactory{
        publicstaticSamplegetSample(intx){
}
}
Samples = MyFactory.getSample( 4 ) ;
<beanid=”samp”class =”MyFactory”factory-method=”getSample”>
        <constructor-arg>
                <value type=”int”>10 </value>
        </constructor-arg>
</bean>
```

Toconfigureaspringbeantobeinstantiatedusingthestaticfactorymethod , weuse“class”and“factory-method”attributeofthe<bean >tag.

ToconfiguretheDependenciesofthefactorymethod(methodparameter)weusethe<constructor-arg>childtag.

```
<beanid=”con”class=”DriverManager”factory-method=”getConnection”>
        <constructor-arg>
                <valuetype=”java.lang.String”> jdbc: oracle:thin:@localhost:1521:orcl </value >
</constructor-arg>
</bean>
```

Instantiation with instance method

```
classFactory{
        publicTestgetTest(intx){
}
}

Factoryfac=newFactory();
Testt1=fac.getTest( 4 );

<beanid=”fac”class=”Factory” >
</bean>

<beanid=”t1”factory-bean=”fac”factory-method=”getTest”>
        <constructor-arg>
```

```
            <value type=”int”> 100 < value>
        </constructor-arg>
</bean>
```

Toconfigurethenon-staticfactorymethod,weuse“factory-bean”and “factory-method” attributeofthe<bean>tag.

Weuse<constructor-arg>tospecifytheDependenciesofthenon-staticfactorymethod.

Abstract beanDefinition:

```
<beanid=”t1”class=”Test”abstract =” true” >
</bean>

<beans>
        <bean id="p1" abstract="true" class="example.ComplexObject">
                <property name="adminEmails">
                        <props>
                        <prop key="administrator">administrator@example.com</prop>
                        <prop key="support">support@example.com</prop>
                        </props>
                </property>
        </bean>
```

//spring beandefinition inheritance

```
        <bean id="child" parent="p1">
                <property name="adminEmails">
                <!-- the merge is specified on the child collection definition →
                <props merge="true">
                        <prop key="sales">sales@example.com</prop>
                        <prop key="support">support@example.co.uk</prop>
                </props>
                </property>
        </bean>
<beans>
```

ConfiguringsetterMethodinjection:

Weuse<property>tagtodescribethesettermethodinjectionforthespringbean.
Each<property>tagdescribeaboutasinglesettermethod.
The <property>tagmustappearafterthe <constructor-arg>withinthe <bean> tag.
THe<property>supportthefollowingattributes…

- name
- value
- ref

Note : The < property> supports child tag similar to <constructor-arg>

```
classAccount{
        Balancebalance;
        publicvoidsetAccountBalance(Balancebalance){
                this.balance = balance;
        }
}
```

propertyname:accountBalance
propertytype: Balance

TheconstructorbasedDependencieswillbeinjectedbeforethesetterbaseddependency.

Circular Dependency

Thesetterbaseddependencyinjectionwillbypassthecirculardependencycontext , howeveritisnotrecommendedtoadoptsuchdesign.

```
classA{
        B b;
        publicA(Bb){
                this.b = b;
        }
        //publicvoid setB(Bb){this.b = b;}
}
class B{
        Aa;
        public B(Aa){
                this.a = a;
        }
        //public void setA(Aa){this.a = a;}
}
```

```
<beanid=”bal” class=”Balance”/>
<beanid=”acc”class=”Account”>
        <propertyname =”accountBalance”ref=”bal”>

        </property>
</bean>
```

The<property>elementofthe<bean>tagalsoallowstospecifyvaluesofdifferenttypes,

weusethefollowingtagtoconfigurethesettermethodargumentdepending uponthesetter methodargumenttype.

<value> , <ref> , <idref> , <null> , <bean> , <list> , <set> , <map> , <props> , <array>

Spring2.0introducesanewapproachofconfiguringthesetter methodinjection .

Inthisapproachwehaveanopportunityto reducethenumberof<property>tag , herewe configurethesettermethodinjectionthroughadynamicattributeofthe<bean>tag.

Toworkwiththisoptionweneedtoimportthe followingnamespace.

xmlns:p="http://www.springframework.org/schema/p"

The“p”namespacecanbeusedinthe <bean>tagtodescribethesettermethod injection.

heretheattribute namewillbethepropertynameandthe valueofthisattributespecifythevalueforthepropery.

e.g

```
<beanid=”t1”class=”Test”>
        <propertyname=”count”>
                <value> 10</value>
        </property>
</bean>
```

canberewrittenusing‘p’namespace

```
<beanid=”t1”class=”Test” p:count=”10” />

<beanid=”samp” class=”Sample”/>

<beanid=”t1”class =”Test”>
        <propertyname=”sample” >
                <refbean =”samp”/>
        </property>
</bean>

<beanid=”t1”class=”Test”p:sample-ref =”samp”/>

classTest{
        Listnames;
        publicvoidsetNames( Listnames){
                this.names=names;
        }
}

<beanid=”t1”class=”Test”>
        <propertyname=”names”>
                <list>
                        <value>sachin</value>
                        <value>virat</value>
                </list>
```

```
        </property>
</bean>

<beanid=”namesList”class=”java.util.ArrayList”>
        //problemtoaddelements ,asjavacollectionframeworkdon’tsupportanysetter method.
</bean>
<beanid=”t1”class=”Test”p:names-ref =”namesList”/>
```

utilnamespace ;

Theutilnamespaceallowtodefinejavacollectionframeworkelementsasspringbean. And also helpsto add data into those collection elements.
Toworkwiththisnameweneedtoimporttheutilnamespace.

```
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/util
http://www.springframework.org/schema/util/spring-util-3.2.xsd"
```

Thisnamespacesupportsthefollowingtags…

```
        <util:list>
        <util:set>
        <util:map>
        <util:properties>
        <util:constant>
        <util:property-path>
```

```
package com;
import java.util.Arrays;
import java.util.List;
public class BeanOne {
        intcount;
        intx;
        String []contents;
        Listnames;
        publicBeanOne(intx){
                this.x = x;
        }
        publicvoidsetCount(intcount){
                this.count = count;
        }
        publicvoidsetContents(String[]contents){
                this.contents = contents;
        }
        public void setNames(List names) {
                this.names = names;
```

```
	}

	public String toString() {
		return "BeanOne [count=" + count + ", x=" + x + ", contents="
				+ Arrays.toString(contents) + ", names=" + names + "]";
	}
}

<util:listid="namelist"list-class=”java.util.ArrayList”> ( list-class )
	<value>sachin</value>
	<value>virat</value>
</util:list>
<beanid="bo"class="com.BeanOne" p:count="200" p:names-ref="namelist">
	<constructor-arg>
		<value type="int">100</value>
	</constructor-arg>
	<property name="contents">
		<array>
			<value>content1</value>
			<value>content2</value>
		</array>
	</property>
</bean>
```

cnamespace: newfromspring3.1

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 wascompiled 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"/>
```

Asperthespringcorecontainer,Thespringbeancreationiscompletedonceallitsdependenciesareinjectedsuccessfully.

Springcorecontainerprovidesapplicationextensionpointto participateinthespringinitializationphase.

Auto wiring support from spring container using XML:

- autowire (constructor , byType , byName , no)
- autowire-candidate (true / false)
- primary (true / false) adding priority to a bean for auto wiring.

- depends-on

The Spring container can autowire relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext.

- Autowiring can significantly reduce the need to specify properties or constructor arguments
- Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration.

When using XML-based configuration metadata, you specify autowire mode for a bean definition with the autowire attribute of the <bean/> element. The autowiring functionality has five modes.

no	(Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
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 propertyis 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.

- 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.

- Wiring information may not be available to tools that may generate documentation from a Spring container.

- 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.

Excluding a bean from autowiring

Depends- on :

```
<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" />
```

Note
The depends-on attribute in the bean definition can specify both an initialization time dependency and, in the case of singleton beans only, a corresponding destroy time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to thegiven bean itself being destroyed. Thus depends-on can also control shutdown order.

SpringbeanscustomInitialization / Destruction

Heretoperform/
usethisoption,weneedtoconfigurethespringbeandescribingthecontainerthatthisspringbeanisinterestedinlisteningforInitializationandDestructioncallback.

Springsupport3waysin configuringthis…
1 - ByimplementingSpringAPIinterfacesuchas

- InitializingBean interface
- DisposableBean interface

2 - Byusinginit-methodanddestroy-methodattribute ofthe <bean>tag
3-UsingJSR250Lifecycleannotation

- @PostConstruct
- @PreDestroy

https://www.youtube.com/watch?v=JfgP566BHW0

Thespringcorecontainerexecutestheinitializersinthefollowingorder….

- @PostConstruct
- InitializingBean (afterPropertiesSet())
- init-methodattributeinthe <bean>tag.

Note:Toworkwith@PostConstructand @PreDestroyweneedexplicitly enablethespringcontainersannotationprocessingbyusing<context:annotation-config/> in the spring beans config xml file

Addjsr 250annotationartifactintothepom.xml under <dependencies> tag

```
<dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>jsr250-api</artifactId>
        <version>1.0</version>
</dependency>
```

Note :Includethecontextnamespaceintothebeans.xmlfile.

Google search forJsr 250 annotationmaven dependency

Working withInitializingBean Interface :

The InitializingBeaninterfacesupports onlyonemethod.

```
publicvoidafterPropertiesSet()
```

Whileimplementingspringbean interested to listen for initialization callback byimplementing the org.springframework.beans.InitializingBeaninterface , weneedto implement the afterPropertiesSet()method of the InitializingBeaninterface.

E.g

```
public class TestBean implements InitializingBean{
        publicTestBean(){
                // TestBeanobjectinitialization
        }
        public void afterPropertiesSet(){
                //Spring bean initialization
        }
}
```

```
package com;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
public class MyBean implementsInitializingBean , DisposableBean{
```

```
	intx ;
	intcount;
	publicMyBean(intx){
		System.out.println("inconstructor");
		this.x = x;
	}
	publicvoidsetCount(intcount){
		System.out.println("in setcountmethod");
		this.count = count;

	}
	public void afterPropertiesSet() throws Exception {
		System.out.println("InafterPropertiesSet");
	}
	public void destroy() throws Exception {
		System.out.println("inDestorymethod");
	}

	publicvoidinit(){
		System.out.println(" ininitmethod ");
	}
	publicvoidfinish(){
		System.out.println("infinishmethod");
	}
	@PostConstruct
	publicvoidstart(){
		System.out.println("instart method");
	}
	@PreDestroy
	publicvoidstop(){
		System.out.println("instopmethod");
	}
}
<?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:c="http://www.springframework.org/schema/c"
	xmlns:p="http://www.springframework.org/schema/p"
	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-3.2.xsd">

	<beanid="mb"class="com.MyBean"	c:x="10"p:count="20"init-method="init"destroy-
method="finish"></bean>
```

```
        <context:annotation-config/>( checkcontext namespace )
</beans>
```

Note :we can also configure global initialization as well as global destruction callback using the default-init-method and default-destroy-methodattributes of the <beans> tag.

Oncea Springbean is created , all Dependencies are injected and also the configured Initialization methods are invoked , The spring bean is readyfor usages.

Lazy-initialized beans

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. 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.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element; for example:
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element

Bean scope :

Fromspring2.0springcontainermanagesthespringbeansineitherofthefollowingscope .

Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports seven scopes, five of which are available only if you use a web aware ApplicationContext.

- **singleton(from 1.0)**
- **prototype (from 1.0)**
- **request(from spring 2.0)**
- **session…**
- **application …**
- **global session (portlet) ….**

- websocket (new from spring 4.x)
- thread(newfrom spring 3.0)

In standaloneApplicationContext suchas :

- ClassPathXMLApplicationContext
- GenericApplicationContext
- FileSystemXMLApplicationContext
- AnnotationConfigApplicationContext.

- singleton, prototype , thread

In web enabled Application contextsuch as:

- GenericWebApplicationContext
- XMLWebApplicationContext
- StaticWebApplicationContext
- AnnotationConfigWebApplicationContext

- Request , session , application , global session , websocket

Fromspring2.0,springsupportsa“scope”attributeforthe <bean>tospecifythescopeforthespringbean.

How to configure spring scope :

till spring 1.2, weused the“singleton” attribute of the <bean> tag to configure the spring bean scope. Thisattribute takes aboolean value , where true indicates singleton . and falseindicates prototype.

In spring 2.0 , we have“scope”attribute of the <bean> tagto configure spring bean scopeas from spring 2.0 the number of spring bean scope possibilities are increased.
If the spring bean scope is not explicitly specified , it is default to “singleton”

Singletonscope:

The singleton scopeof a spring bean specifies that the spring bean should be instantiated onlyonce in this container context.

```
<beanid=”t1”class=”Test”scope=”singleton”>

<beanid=”t2”class=”Test”scope =”singleton”/>

classTest{
        intx, y;
        staticTestt;
        publicTest(){
                t=this;
                System.out.println(this);
                System.out.println(x +" : "+y);
                System.out.println("inconstructor");
                thrownewRuntimeException();
        }

        public staticvoid main(String []args){
                Testt1 =null;
                try{
```

```
                t1 = newTest();
                }catch(Exceptione){}
                System.out.println("t1 is "+t1);
                System.out.println("tis "+t);
        }
}
/*
        allocatesthememoryforthe instance member
        initialize theinstancememberswith thedefaultvalue
        encapsulatetheallocatedmemoryasobject

        invoketheconstructortoinitializethe object.
        Oncetheconstructorreturn , newkeywordreturn the object addressto the
reference variable

*/
```

Note : As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans

Prototypescope:

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made tothe container.

Theprototypebeansarecreatedbythe springcontainer , however the springcontainer don'tmanagethe lifeoftheprototypebean.

Hencethespringcontainerdoesn'tguaranteethe completelife cyclemanagementincludingthedestruction(finalization)foraprototypebean.

As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

Threadscope:

Thisisa newscopefromspring3.x
Thisscopeisnot by defaultconfiguredwith springcontainer.
Weneedtoexplicitlyconfigurethisscopewiththecontainer .

Custom Scope:

The bean scoping mechanism is extensible; You can define your own scopes, or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in singleton and prototype scopes.

Creatinga custom scope includes to write a java type subtype of org.springframework.beans.factory.config.Scope interface

public interface Scope

Strategy interface used by a ConfigurableBeanFactory, representing a target scope to hold bean instances in. This allows for extending the BeanFactory's standard scopes "singleton" and "prototype" with custom further scopes, registered for a specific key.

ApplicationContext implementations such as a WebApplicationContext may register additional standard scopes specific to their environment, e.g. "request" and "session", based on this Scope SPI.

Scope implementations are expected to be thread-safe. One Scope instance can be used with multiple bean factories at the same time, if desired (unless it explicitly wants to be aware of the containing BeanFactory), with any number of threads accessing the Scope concurrently from any number of factories.

The Scope interface has four methods to get objects from the scope, remove them from the scope, and allow them to be destroyed

Method Summary

Object	get(String name, ObjectFactory objectFactory) Return the object with the given name from the underlying scope, creating it if not found in the underlying storage mechanism.
String	getConversationId() Return the *conversation ID* for the current underlying scope, if any.
void	registerDestructionCallback(String name, Runnable callback) Register a callback to be executed on destruction of the specified object in the scope (or at destruction of the entire scope, if the scope does not destroy individual objects but rather only terminates in its entirety).
Object	remove(String name) Remove the object with the given name from the underlying scope.

Using a custom scope

After you write and test one or more custom Scope implementations, you need to make the Spring container aware of your new scope(s). The following method is the central method to register a new Scope with the Spring container:

void registerScope(String scopeName, Scope scope);

This method is declared on the ConfigurableBeanFactory interface, which is available on most of the concrete ApplicationContext implementations that ship with Spring via the BeanFactory property.

The example below uses SimpleThreadScope which is included with Spring, but not registered by default. The instructions would be the same for your own custom Scope implementations.

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

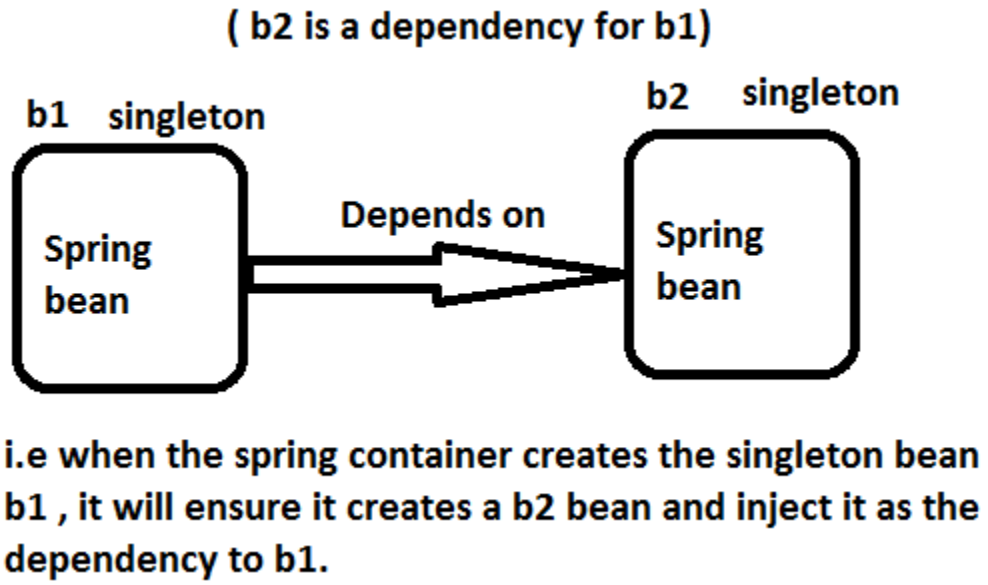
With a custom Scope implementation, you are not limited to programmatic registration of the scope. You can also do the Scope registration declaratively, using the CustomScopeConfigurer class:

```
CustomScope customScope = new CustomScope();
((ConfigurableApplicationContext) appContext).getBeanFactory().registerScope("thread", customScope);
```

Declarative style of scope configuration :

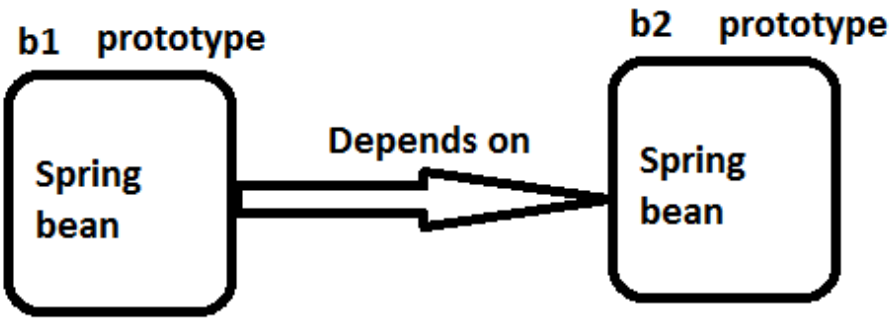
```
<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
                <map>
                        <entry key="thread">
                <bean class="org.springframework.context.support.SimpleThreadScope"/>
                        </entry>
                </map>
        </property>
</bean>
```

Scoped Bean Dependencies :



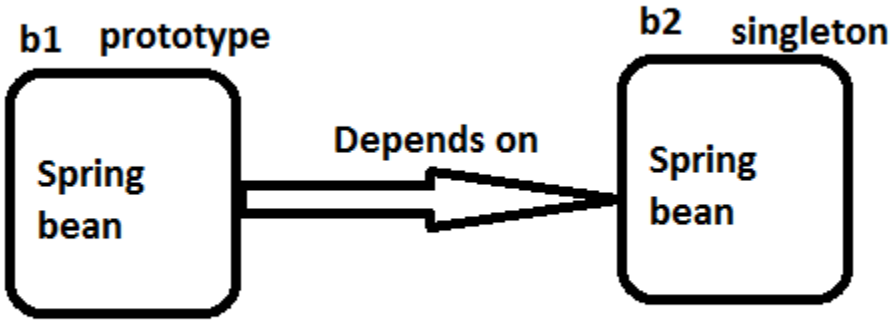
i.e when the spring container creates the singleton bean b1 , it will ensure it creates a b2 bean and inject it as the dependency to b1.

(b2 is a dependency for b1)

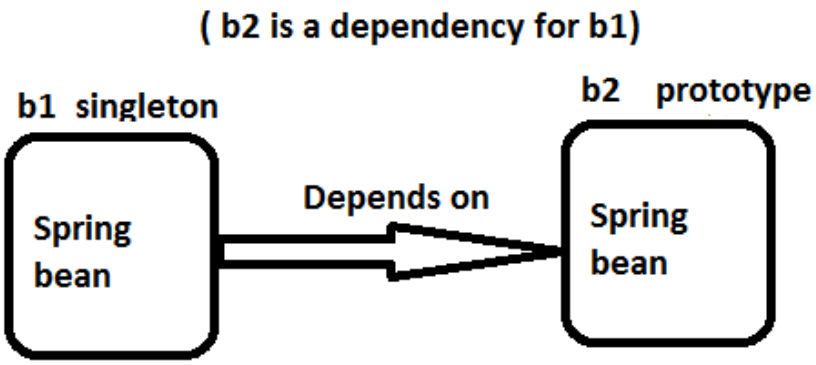


i.e on a request for the b1 (new bean) the spring container will create a b2 bean as it is the dependencies for the b1 and inject it into b1 and returns b1 bean to the client.

(b2 is a dependency for b1)



in this context , the spring container creates a single instance of b2 and inject it into all the instance of b1



in this situation , the dependency injection style doesn't work , as the container got the chance to create a single bean for b1 , but it is required to have multiple bean of type b2 within the b1 as per the business logic requreiment .

Bean Dependencies(singleton bean depends upon prototype bean)

Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

Theabove situation can be solved in 2 ways…

- **Make the bean A aware of the container by implementing ApplicationContextAware interface.**
- **Using LookupMethod injection**

Implementing ApplicationContextAware:

```
public class CommandManager implements ApplicationContextAware {
        private ApplicationContext applicationContext;

        public Object process(Map commandState) {
                // grab a new instance of the appropriate Command
                Command command = createCommand();
                // set the state on the (hopefully brand new) Command instance
                command.setState(commandState);
                return command.execute();
        }
        protected Command createCommand() {
                // notice the Spring API dependency!
                return this.applicationContext.getBean("command", Command.class);
        }


        public void setApplicationContext(
                ApplicationContext applicationContext) throws BeansException {
                this.applicationContext = applicationContext;
```

```
        }
}

classCommand{
}
```

Lookup Method Injection :

Lookup method injection is the ability of the container to override methods on container managed beans, to return the lookup result for another named bean in the container.

The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

Note

- **For this dynamic subclassing to work, the class that the Spring bean container will subclass cannot be final, and the method to be overridden cannot be final either.**
- **Unit-testing a class that has an abstract method requires you to subclass the class yourself and to supply a stub implementation of the abstract method.**
- **Concrete methods are also necessary for component scanning which requires concrete classes to pick up.**
- **A further key limitation is that lookup methods won't work with factory methods and in particular not with @Bean methods in configuration classes, since the container is not in charge of creating the instance in that case and therefore cannot create a runtime-generated subclass on the fly.**
- **Finally, objects that have been the target of method injection cannot be serialized.**

```
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();
}
```

The method to be injected requires a signature of the following form:
<public|protected> [abstract] <return-type>theMethodName(no-arguments);

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

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
```

```
        <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
        <lookup-method name="createCommand" bean="command"/>
</bean>
```

Arbitrary method replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation.

With XML-based configuration metadata, you can use the replaced-method element to replace an existing method implementation with another, for a deployed bean. Consider the following class, with a method computeValue, which we want to override:

```
public class MyValueCalculator {
        public String computeValue(String input) {
                // some real code…
        }
        // some other methods...
}
```

A class implementing the org.springframework.beans.factory.support.MethodReplacer interface provides the new method definition.

```
/**
* meant to be used to override the existing computeValue(String)
* implementation in MyValueCalculator
*/
public class ReplacementComputeValue implements MethodReplacer {
        public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
                // get the input value, work with it, and return a computed result
                String input = (String) args[0];
                …
                return ...;
        }
}

<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
        <!-- arbitrary method replacement →
        <replaced-method name="computeValue" replacer="replacementComputeValue">
                <arg-type>String</arg-type>
        </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

Customizing the nature of a bean

Spring Aware Interfaces for beans

Sometimes it is required that our beans needs to get some information about Spring container and its resources.

For example, sometime bean need to know the current Application Context using which it can perform some operations like loading specific bean from the container in a programmatic way.

So to make the beans aware about this, spring provides lot of Aware interfaces.

All we have to do is, make our bean to implement the Aware interface and implement the setter method of it.

org.springframework.beans.factory.Aware is the root marker interface.

All the Aware interfaces which we use are the sub interfaces of the Aware interface.

Some of the commonly used Aware interfaces are

1) ApplicationContextAware
Bean implementing this interface can get the current application context and this can be used to call any service from the application context

2) BeanFactoryAware
Bean implementing this interface can get the current bean factory and this can be used to call any service from the bean factory

3) BeanNameAware
Bean implementing this interface can get its name defined in the Spring container.

4) MessageSourceAware
Bean implementing this interface can get the access to message source object which is used to achieve internationalization

5) ServletContextAware
Bean implementing this interface can get the access to ServeltContext which is used to access servlet context parameters and attributes

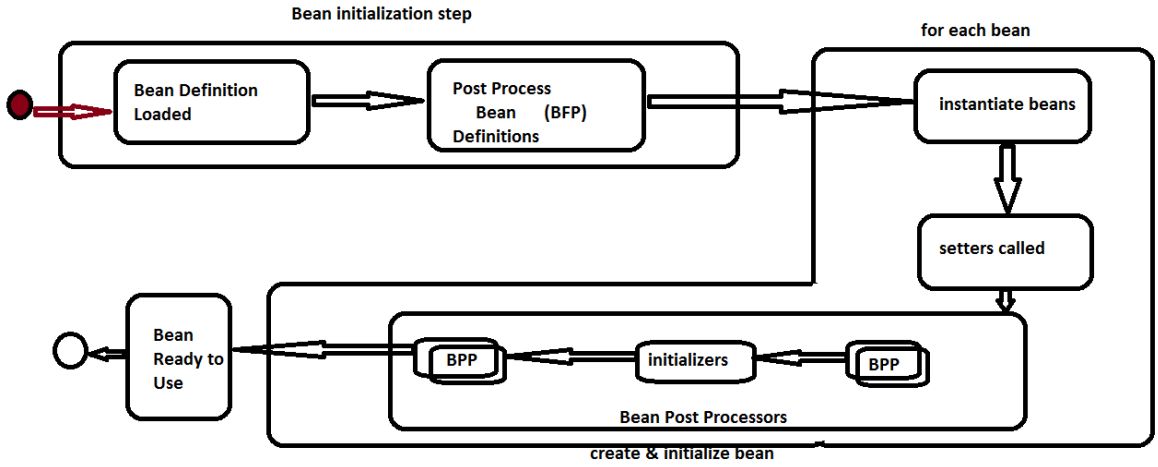
6) ServletConfigAware
Bean implementing this interface can get the access to ServletConfig object which is used to get the servlet config parameters

7) ApplicationEventPublisherAware
Bean implementing this interface can publish the application events and we need to create listener which listen this event.

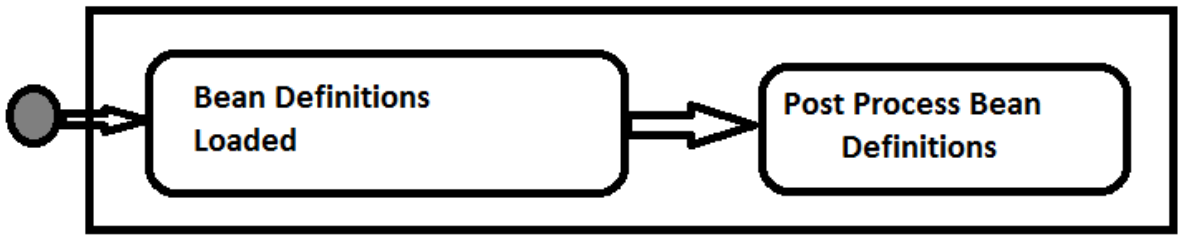
8) ResourceLoaderAware
Bean implementing this interface can load the resources from the classpath or any external file.

9) EnvironmentAware

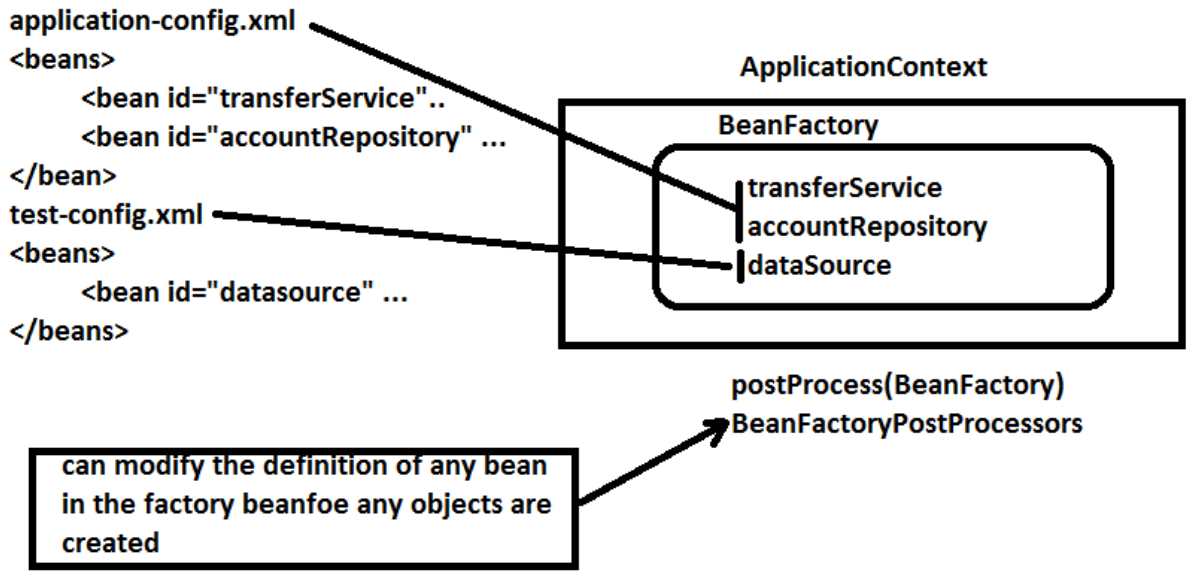
Container Extension Points



load Bean Definitions:
- The Bean definitions are read (ex , XML , Annotation or java config)
- Bean definitions are loaded into the context's BeanFactory
- each indexed under its id
- special BeanFactoryPostProcessor beans are invoked.
- These BeanFactoryPostProcessor can modify the definition of any bean



Load Bean Definitions



The BeanFactoryPostProcessor

- useful for applying transformation to groups of bean definition
  - -before any objects are actually created
- several useful implementation are provided by the framework.
  - you can also write your own
  - Implement the BeanFactoryPostProcessor

```
public interface BeanFactoryPostProcessor{
        public void postProcessBeanFactory(ConfigurableListableBeanFactory factory);
}
```

commonly used BeanFactoryPostProcessor

1. AspectJWeavingEnabler
2. ConfigurationClassPostProcessor
3. CustomAutowireConfigurer
4. CustomEditorConfigurer
5. CustomScopeConfigurer
6. DeprecatedBeanWarner
7. PlaceholderConfigurerSupport
8. PreferencesPlaceholderConfigurer
9. PropertyOverrideConfigurer
10. PropertyPlaceholderConfigurer
11. PropertyResourceConfigurer
12. PropertySourcesPlaceholderConfigurer

example:

```
<bean..>
        <context:property-placeholder location="db-config.properties"/>

        <bean id="datasource" class="com.oracle.jdbc.pool.OracleDatasource">
                <property name="URL"value="${dbURL}"/>
```

```
            <property name="user"value="${dbUSER}"/>
        </bean>
</bean>
+
db-config.properties
dbURL=jdbc:oracle...
dbUSER=system

= Results :

        <bean id="datasource" class="com.oracle.jdbc.pool.OracleDatasource">
            <property name="URL"value="jdbc:oracle..."/>
            <property name="user"value="system"/>
        </bean>
```

But where's The BeanPostProcessor
The namespace is just an elegant way to hide the corresponding bean declaration.

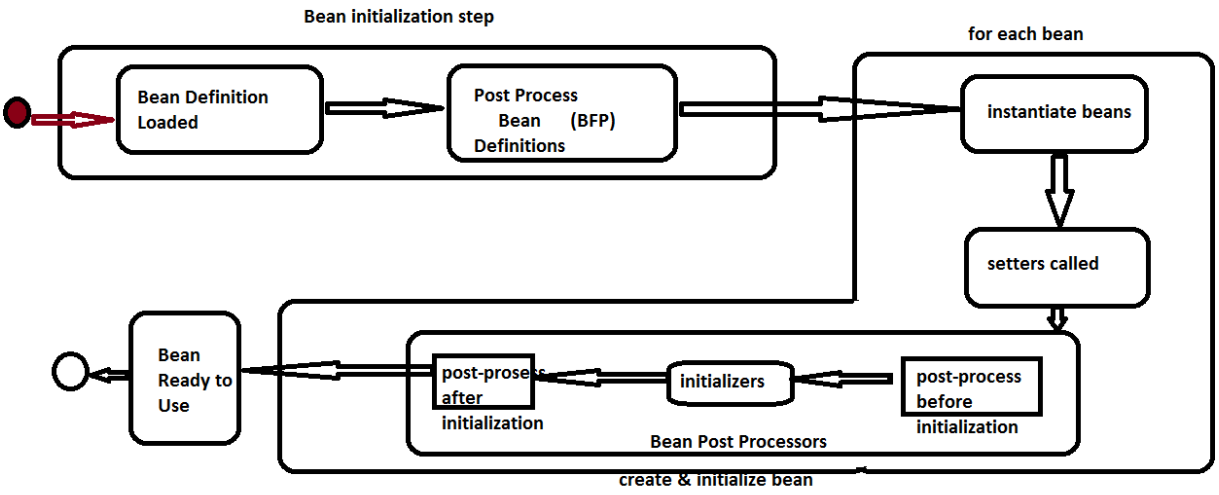
```
<context:property-placeholder location="db-config.properties"/>
```

uses

```
<bean class="org.springframework...PropertySourcePlaceHolderConfigurer">
        <property name="location" value="db-config.properties"/>
</bean>
```

PropertySourcesPlaceHolderConfigurer was introduced in spring 3.1. prior to that , PropertyPlaceHolderConfigurer was used instead.

We just finished BeanFactoryPostProcessor steps and now we have taken all of the bean definition which are potentially changed with their values in the properties file and now want to iterate over all the bean and instantiate and initialize the bean ,
then we can apply individual bean level post processing.



Bean Post Processing :

There are two types of bean post processors.

- Initializers
  - initialize the bean in instructed
  - activated by init-method or @PostConstruct

- All the rest
  - allow for additional richer configuration features.
  - may run before or after the initialize step.

BeanPostProcessor:

An important extension point in spring
- can modify bean instances in any way.
- powerful enabling feature

Spring provides many implementation
- not common to write your own (but you can)

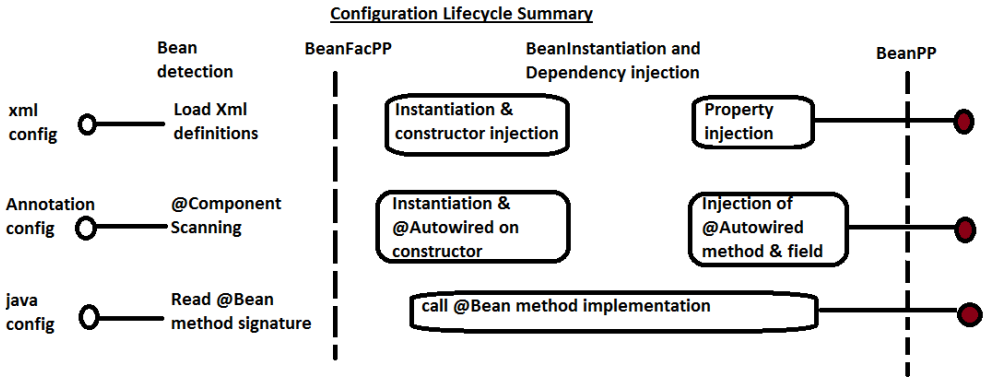
```
public interface BeanPostProcessor{
    public Object postProcessAfterInitialization(Object bean,String name)
    public Object postProcessBeforeInitialization(Object bean,String name);
}
```

After Bean Post processing it may also result a whole new different object than the supplied object (e.g like a proxy , where a proxy may insert a whole new layer of code , however the returned object must obey the same interface)

commonly used Bean Post Processors:
Activated by <context:annotation-config> (context:component-scan)

CommonAnnotationBeanPostProcessor (handles JSR 250 common annotation)
RequiredAnnotationBeanPostProcessor(enforces @Required annotation semantics)
AutowiredAnnotationBeanPostProcessor(enables recognition of @Autowired annotation
ConfigurationClassPostProcessor (enables Java configuration supports)

Others that can be configured manually :
PersistenceAnnotationBeanPostProcessor (enables use of @PersistenceContext in JPA DAO classes)
PersistenceExceptionTranslationPostProcessor (performs exception translation for classes annotated with @Repository)



Spring uses BeanPostProcessors to

Perform initialization

@PostConstruct annotation enabled by CommonAnnotationBeanPostProcessor

perform Validation

JSR 303 validation enabled by BeanValidationPostProcessor

Add behavior

@Async annotation enable by AsyncAnnotationBeanPostProcessor

Writing a custom BeanPostProcessor

Problem : spring mvc provides several ExceptionResolver strategies enabled by default, Processing order is pre-defined , How do i change the order?

solution :

Manually configure the 3 resolvers and set the order property

use a custom BeanPostProcessor to set the order.

<mvc:annoation-drivern/>

<!-- with this configuration , three ExceptionResolvers are registered:

ExceptionHandlerExceptionResolver - ExceptionResolver for @Exception annotated method.

ResponseStatusExceptionResolver - ExceptionResolver enabling @ResponseStatus mapping

DefaultHandlerExceptionResolver - ExceptionResolver for mapping exception to Http status codes.

Three ExceptionResolvers are automatically registered.

Order they are consulted by DispatcherServlet is pre-defined

what if you wanted to change the order.

Annotation-based container configuration

The way Annotations are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them.

Using Annotations configuration becomes decentralized and harder to control.

Spring 2.0 introduced the possibility of enforcing required properties with the @Required annotation.

Spring 2.5 made it possible to follow that same general approach to drive Spring's dependency injection. Essentially, the @Autowired annotation provides the same capabilities of the "autowire" attribute of the bean tag.

Spring 2.5 also added support for JSR-250 (common annotations) such as @PostConstruct, and @PreDestroy.

JSR 250 Annotation List :

Annotation name	description
Generated	Marks sources that have been generated
Resource	Declares a reference to a resource, e.g. a database
Resources	Container for multiple Resource annotations
PostConstruct	Is used on methods that need to get executed after dependency injection is done to perform any initialization.
PreDestroy	Is used on methods that are called before the instance is removed from the container
Priority	Is used to indicate in what order the classes should be used. For, e.g., the Interceptors specification defines the use of priorities on interceptors to control the order in which interceptors are called.
RunAs	Defines the role of the application during execution in a Java EE container
RolesAllowed	Specifies the security roles permitted to access method(s) in an application.
PermitAll	Specifies that all security roles are permitted to access the annotated method, or all methods in the annotated class.

DenyAll	Specifies that no security roles are allowed to invoke the specified method(s).
DeclareRoles	Used to specify the security roles by the application.
DataSourceDefinition	Is used to define a container DataSource and be registered with JNDI. The DataSource may be configured by setting the annotation elements for commonly used DataSource properties.
ManagedBean	Is used to declare a Managed Bean which are container managed objects that support a small set of basic services such as resource injection, lifecycle callbacks and interceptors.

Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named.

Inject	Identifies injectable constructors, methods, and fields.
Named	String-based qualifier.
Qualifier	Identifies qualifier annotations.
Scope	Identifies scope annotations.
Singleton	Identifies a type that the injector only instantiates once.

Note

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches

Spring uses various BeanPostProcessorto deal with these annotation , we can configure them explicitly as individual bean definition. but they can also be implicitly registered by including the following tag in an XML-based Spring configuration

<context:annotation-config/>

The above xml configuration will implicitly register the following BeanPostProcessor.

- **AutowiredAnnotationBeanPostProcessor,**
- **CommonAnnotationBeanPostProcessor,**
- **PersistenceAnnotationBeanPostProcessor,**
- **RequiredAnnotationBeanPostProcessor**

Note

<context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined.

@Required Annotation

```
@Retention(value=RUNTIME)
@Target(value=METHOD)
public @interface Required
```

The @Required annotation applies to bean property setter methods.

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 NullPointerExceptions.

@Autowired Annotation

```
@Target(value={CONSTRUCTOR,METHOD,PARAMETER,FIELD,ANNOTATION_TYPE})
@Retention(value=RUNTIME)
 @Documented
public @interface Autowired
```

```
@Autowired
public void MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
}
```

Note

As of Spring Framework 4.3, the @Autowired constructor is no longer necessary if the target bean only defines one constructor. If several constructors are available, at least one must be annotated to teach the container which one it has to use.

You can also apply the @Autowired annotation to "traditional" setter methods
You can also apply the annotation to methods with arbitrary names and/or multiple arguments:

You can apply @Autowired to fields as well and even mix it with constructors:

By default, the autowiring fails whenever zero candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies.
@Autowired(required=false)

Note

Only one annotated constructor per-class can be marked as required, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.

@Autowired's required attribute is recommended over the @Required Annotation.

@Autowiredannotation work defaultwith by-typeprinciples .hence if multiple bean of the target type found will result error.

However it can fall back to by-name principle when ambiguities arises.

@Primary Annotation

```
@Target(value={TYPE,METHOD})
@Retention(value=RUNTIME)
public @interface Primary
```

Because autowiring by-type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's @Primary annotation.

@Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.

```
@Configuration
public class MovieConfiguration {
        @Bean
        @Primary
        public MovieCatalog firstMovieCatalog() { ... }

        @Bean
        public MovieCatalog secondMovieCatalog() { ... }
        // ...
}
```

With such configuration, the following MovieRecommender will be autowired with the firstMovieCatalog

```
public class MovieRecommender {
        @Autowired
        private MovieCatalog movieCatalog;
        // ...
}
```

@Qualifier Annotation

```
@Target(value={FIELD,METHOD,PARAMETER,TYPE,ANNOTATION_TYPE})
@Retention(value=RUNTIME)
public @interface Qualifier
```

@Primary is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When more control over the selection process is required, Spring's @Qualifier annotation can be used. You can associate qualifier values with specific arguments

```
@Autowired
@Qualifier("main")
private MovieCatalog movieCatalog;

@Autowired
public void prepare(@Qualifier("main")MovieCatalog movieCatalog,
CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
}
```

The bean with qualifier value "main" is wired with the constructor argument that is qualified with the same value.We can also specify bean id as part of @Qualifier annotation

```
<bean id=”someid”class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- inject any dependencies required by this bean -->
</bean>

<bean class="example.SimpleMovieCatalog">
        <qualifier value=”other"/>
        <!-- inject any dependencies required by this bean -->
</bean>
```

JSR 250 annotations :

@Resource
@PostConstruct
@PreDestroy

@Resource Annotation

```
@Target(value={TYPE,FIELD,METHOD})
@Retention(value=RUNTIME)
public @interface Resource
```

Spring also supports injection using the JSR-250 @Resource annotation on fields or bean property setter methods.

@Resource takes a name attribute, and by default Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics,

If no name is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name

The name provided with the annotation is resolved as a bean name by the ApplicationContext of which the CommonAnnotationBeanPostProcessor is aware.

```
@Resource(name=”someMovieCatalog”)
private MovieCatalog movieCatalog;

@Resource
publicvoid setMovieCatalog(MovieCatalogmovieCatalog){

}
```

@PostConstruct and @PreDestroy

The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations. Introduced in Spring 2.5,

Classpath scanning and managed ComponentsAutoDetection

The @Component , @Repository , @Service and @Controller annotation in place and after enabling automatic component scanning, Spring will automatically import the beans into the container so you don't have to define them explicitly with XML, These annotations are called as Stereotype annotations.

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts.

The @Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick it up and pull it into the application context.

Spring can automatically detect stereotyped classes and register corresponding BeanDefinitions with the ApplicationContext, The following classes are eligible for autodetection.

```
@Component
public class Employee{}

@Component(“myEmp”)
public class Employee{}
```

The @Repository annotation is a specialization of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions (thrown from the DAO methods) eligible for translation into spring's DataAccessException.

To autodetect these classes and register the corresponding beans into the spring container we can either use <context:component-scan/> xmlelement or for java config style configuration we use the @ComponentScanannotation.

where the basePackages attribute is a common parent package for the two classes. (Alternatively, you can specify a comma/semicolon/space-separated list that includes the parent package of each class.)

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
...
}
```

The following is an alternative using XML

```
<context:component-scan base-package="org.example"/>
```

The AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both included implicitly when you use the component-scan element.

You can disable the registration of AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor by including the *annotation-config* attribute with a value of false

@Value annotation : 3.0

```
@Target(value={FIELD,METHOD,PARAMETER,ANNOTATION_TYPE})
@Retention(value=RUNTIME)
public @interface Value
```

Annotation at the field or method/constructor parameter level that indicates a default value expression for the affected argument.

Typically used for expression-driven dependency injection. Also supported for dynamic resolution of handler method parameters, e.g. in Spring MVC.

A common use case is to assign default field values using "#{systemProperties.myProp}" style expressions.

```
@Component
public class Pancard {
        @Value("#{employee.employeeName}")
        private String panHolderName;

        @Value("ABCD1234X")
        private String panNo;
...
```

```
}

@Component// Employee employee = new Employee()
public class Employee {

	@Value("123")
	private int employeeId;
	@Value("Pratap")
	private String employeeName;

	@Autowired
	private Pancard pancard;
…
}
Config.xml
<context:component-scan base-package="com.model"></context:component-scan>

ApplicationContext c = new ClassPathXmlApplicationContext("config.xml");
Employeeeemp = c.getBean("employee", Employee.class);
Pancard p = emp.getPancard();
	if( p!=null){
		System.out.println(p.getPanHolderName()+"\t"+p.getPanNo());
	}
```

Reading Data from Properties file into @value annotation

```
@Component
public class UserDetails {
	@Value("${Details.uname}")
	private String userName;

	@Value("${Details.password}")
	private String password;
	…
}
<context:component-scan base-package="com.model"></context:component-scan>
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
		<property name="locations">
			<list>
				<value>MyApp.properties</value>
			</list>
		</property>
</bean>


MyApp.properties
Details.uname=pratap
Details.password=kumar
```

```
ApplicationContextc = new ClassPathXmlApplicationContext("config.xml");
UserDetails ud = c.getBean("userDetails", UserDetails.class);
System.out.println(ud.getUserName()+"\t"+ud.getPassword());
```

Using filters to customize scanning

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as *include-filter* or *exclude-filter* sub-elements of the component-scan element. Each filter element requires the type and expression attributes. The following table describes the filtering options.

Filter Types

Filter Type	Example Expression	Description
annotation	org.example.SomeAnnotation	An annotation to be present at the type level in target components.
assignable	org.example.SomeClass	A class (or interface) that the target components are assignable to (extend/implement).
aspectj	org.example..*Service+	An AspectJ type expression to be matched by the target components.
regex	org\.example\.Default.*	A regex expression to be matched by the target components class names.
custom	org.example.MyCustomTypeFilter	A custom implementation of the org.springframework.core.type.TypeFilterinterface.

The following example shows the XML configuration ignoring all @Repository annotations and using "stub" repositories instead.

```
<beans ...>

 <context:component-scan base-package="org.example">
<context:include-filter type="regex" expression=".*stub.*Repository"/>
<context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Repository"/>
 </context:component-scan>

</beans>
```

You can also disable the default filters by providing *use-default-filters="false"* as an attribute of the <component-scan/> element. This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, or@Controller.

Different Derivatives of @Component annotation

@Component
@Controller > @Component
@RestControlelr > @Controller > @Component
@Service > @Component
@Repository> @Component
@Configuration > @Component
@SpringBootCofigruation > @Configuration > @Compoent
@SpringBootApplication > @SpringBootCofigruation > @Configuration > @Component

Component scanning using @ComponentScan :

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

```
@Configuration          //JavaConfig style
@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 {
}
```

Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation

```
@Configuration
@ComponentScan(basePackages = "org.example",
includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
excludeFilters = @Filter(Repository.class))

public class AppConfig {
...
}
```

Naming autodetected components

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGenerator strategy known to that scanner. By default, any Spring stereotype annotation (@Component, @Repository, @Service, and @Controller) that contains a name value will thereby provide that name to the corresponding bean definition.

If such an annotation contains no name value or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. if the following two components were detected, the names would be myMovieLister and movieFinderImpl:

```
@Service("myMovieLister")
public class SimpleMovieLister {
// ...}

@Repository
```

```
public class MovieFinderImpl implements MovieFinder {
// ...
}
```

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean naming strategy. First, implement the BeanNameGenerator interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner:

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
...
}
<beans>
<context:component-scan base-package="org.example"
                name-generator="org.example.MyNameGenerator" />
</beans>
```

Defining scope for the AutoDetected components :

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. However, sometimes you need a different scope which can be specified via the @Scope annotation. Simply provide the name of the scope within the annotation:

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
// ...
}
```

Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations

using Maven, the javax.inject artifact is available in the standard Maven repository.

```
<dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>
        <version>1</version>
</dependency>
```

Dependency Injection with @Inject and @Named

```
@Target(value={METHOD,CONSTRUCTOR,FIELD})
@Retention(value=RUNTIME)
@Documented
public @interface Inject{
}
```

Instead of @Autowired, @javax.inject.Inject can be used.

it is possible to use @Inject at the field level, method level and constructorargument level.

```
@Inject
public void setMovieFinder(MovieFinder movieFinder) {
this.movieFinder = movieFinder;
}
```

If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation.

```
@Inject
public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
}
```

@Named: a standard equivalent to the @Component annotation

```
@Qualifier
@Documented
@Retention(value=RUNTIME)
public @interface Named{}
```

String-based qualifier.

Example usage:

```
public class Car {
 @Inject
@Named("driver")
Seat driverSeat;

 @Inject @Named("passenger")
Seat passengerSeat;
 ...
}
```

Instead of @Component, @javax.inject.Named may be used as follows:

```
@Named("movieListener")
public class SimpleMovieLister {
        ...
}
```

Note :

When using @Named, it is possible to use component scanning in the exact same way as when using Spring annotations:

In contrast to @Component, the JSR-330 @Named annotation is not composable

Java-based container configuration

Springs new Java-configuration support are around @Configurationannotated classes and @Beanannotated methods.

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container. This plays similar roles as the <bean/> element in XML configuration.

You can use@Bean annotated methods with any Spring @Component, however, they are most often used with @Configuration beans.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions.

```
@Configuration
public class AppConfig {
        @Bean
        public MyService myService() {
                return new MyServiceImpl();
        }
}
```

The above configuration is equivalent to the following XML configuration.

```
<beans ...>
        <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Full Modevs Lite mode :

When @Bean methods are declared within classes that are not annotated with @Configuration they are referred to as being processed in a 'lite' mode. For example, bean methods declared in a @Component or even in a plain old class will be considered 'lite'.

Unlike full @Configuration, lite @Bean methods cannot easily declare inter-bean dependencies. Usually one @Bean method should not invoke another @Bean method when operating in 'lite' mode.

Only using @Bean methods within @Configuration classes is a recommended approach of ensuring that 'full' mode is always used.

Instantiating the Spring container using AnnotationConfigApplicationContext

AnnotationConfigApplicationContext, new in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.

Simple construction of AnnotationConfigApplicationContext :

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
MyService myService = ctx.getBean(MyService.class);
```

Building the container programmatically using register(Class<?>…)

An AnnotationConfigApplicationContext may be instantiated using a no-arg constructor and then configured using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext.

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class, OtherConfig.class);
ctx.register(AdditionalConfig.class);
ctx.refresh();
MyService myService = ctx.getBean(MyService.class);
```

Enabling component scanning with scan(String…)

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.scan("com.acme");
ctx.refresh();
MyService myService = ctx.getBean(MyService.class);
```

AnnotationConfigApplicationContext(java.lang.String... basePackages)

The above is similar to <context:component-scan/> xml element or @ComponentScan annotation.

Remember that @Configuration classes are meta-annotated with @Component, so they are candidates for component-scanning! In the example above, assuming that AppConfig is declared within the com.acme package (or any package underneath), it will be picked up during the call to scan(), and upon refresh() all its @Bean methods will be processed and registered as bean definitions within the container.

Using the @Bean annotation

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

You can use the @Bean annotation in a @Configuration-annotated or in a @Component-annotated class.

To declare a bean, simply annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method's return value. By default, the bean name will be the same as the method name.

```
@Configuration
public class AppConfig {
        @Bean
        public TransferService transferService() {
```

```
            return new TransferServiceImpl();
        }
}
```

The above @Bean annotated method create a spring bean named transferService available in the ApplicationContext, bound to an object instance of type TransferServiceImpl:

Configuring Bean Dependencies :

A @Bean annotated method can have an arbitrary number of parameters describing the dependencies required to build that bean.

```
@Bean
public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
}
```

Receiving lifecycle callbacks

Any classes defined with the @Bean annotation support the regular lifecycle callbacks and can use the @PostConstruct and @PreDestroy annotations from JSR-250,

The regular Spring lifecycle callbacks are fully supported as well. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods are called by the Container.

The standard set of *Aware interfaces such as BeanFactoryAware, BeanNameAware,MessageSourceAware, ApplicationContextAware, and so on are also fully supported.

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

By default, beans defined using Java config that have a public close or shutdown method are automatically enlisted with a destruction callback. If you have a public close or shutdown method and you do not wish for it to be called when the container shuts down, simply add @Bean(destroyMethod="") to your bean definition to disable the default (inferred) mode.

Specifying bean scope

Using the @Scope annotation You can specify that your beans defined with the @Bean annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section. The default scope is singleton, but you can override this with the @Scope annotation:

```
@Configuration
public class MyConfiguration {
        @Bean
        @Scope("prototype")
        public Encryptor encryptor() {
                // …
        }
```

```
}
```

Customizing bean naming

By default, configuration classes use a @Bean method's name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute.

```
@Configuration
public class AppConfig {
        @Bean(name ={ "myFoo"})
        public Foo foo() {
                return new Foo();
        }
}
```

Bean aliasing

As discussed in the section called "Naming beans", it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The name attribute of the @Bean annotation accepts a String array for this purpose.

```
@Configuration
public class AppConfig {
@Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
public DataSource dataSource() {
        // instantiate, configure and return DataSource bean…
        }
}
```

Bean description

Sometimes it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps via JMX) for monitoring purposes. To add a description to a @Bean the @Description annotation can be used:

```
@Configuration
public class AppConfig {
        @Bean
        @Description("Provides a basic example of a bean") new from spring 4.0
        public Foo foo() {
                return new Foo();
        }
}
```

Using the @Configuration annotation:

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies.

Injecting inter-bean 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();
	}
}
```

This method of declaring inter-bean dependencies only works when the @Bean method is declared within a @Configuration class. You cannot declare inter-bean dependencies using plain @Component classes.

Using the @Import annotation

Much as the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class:

```
@Configuration
public class ConfigA {
	@Bean
	public A a() {
		return new A();
	}
}
@Configuration
@Import(ConfigA.class)
public class ConfigB {
	@Bean
	public B b() {
		return new B( );
	}
}
```

Injecting dependencies on imported @Bean definitions

beans will have dependencies on one another across configuration classes.
when using @Configuration classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

```
@Configuration
public class ServiceConfig {
@Bean
public TransferService transferService(AccountRepository accountRepository) {
	return new TransferServiceImpl(accountRepository);
```

```
}
}

@Configuration
public class RepositoryConfig {
@Bean
public AccountRepository accountRepository(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
}
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {
        @Bean
        public DataSource dataSource() {
                // return new DataSource
        }
}
```

Lookup Method injection using java config :

…

How Java Based configuration works internally :

@Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

Rulesneed to be followed while Writing @Configuration classes :

There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time:

- **Configuration classes should not be final**
- **They should have a constructor with no arguments**

Note that @Configuration classes are ultimately just another bean in the container: This means that they can take advantage of @Autowired and @Value injection etc just like any other bean!

Combining Java and XML configuration

Spring's @Configuration class support does not aim to be a 100% complete replacement for Spring XML. Some facilities such as Spring XML namespaces remain an ideal way to configure the container.

In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way using, for example, ClassPathXmlApplicationContext, or in a "Java-centric" fashion using AnnotationConfigApplicationContext and the @ImportResource annotation to import XML as needed.

XML-centric use of @Configuration classes

It may be preferable to bootstrap the Spring container from XML and include @Configuration classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it will be easier to create @Configuration classes on an as-needed basis and include them from the existing XML files.

Below you'll find the options for using @Configuration classes in this kind of "XML-centric" situation.

Remember that @Configuration classes are ultimately just bean definitions in the container. In this example, we create a @Configuration class named AppConfig and include it within system-testconfig. xml as a <bean/> definition. Because <context:annotation-config/> is switched on, the container will recognize the @Configuration annotation and process the @Bean methods declared in AppConfig properly.

```
@Configuration
public class AppConfig {
@Autowired
private DataSource dataSource;
        @Bean
        public AccountRepository accountRepository() {
                return new JdbcAccountRepository(dataSource);
        }
        @Bean
        public TransferService transferService() {
                return new TransferService(accountRepository());
        }
}

<beans>
<context:annotation-config/>
<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="url" value="${jdbc.url}"/>
<property name="username" value="${jdbc.username}"/>
<property name="password" value="${jdbc.password}"/>
</bean>
</beans>

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=

ApplicationContextctx =
        new ClassPathXmlApplicationContext("classpath:/com/acme/system-testconfig.xml");
TransferService transferService = ctx.getBean(TransferService.class);
```

@Configuration class-centric use of XML with @ImportResource

In applications where @Configuration classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. In these scenarios, simply use @ImportResource and define only as much XML as is needed. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum.

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {
@Value("${jdbc.url}")
private String url;
@Value("${jdbc.username}")
private String username;
@Value("${jdbc.password}")
private String password;
@Bean
public DataSource dataSource() {
return new DriverManagerDataSource(url, username, password);
}
}
```

```
<beans>
<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
TransferService transferService = ctx.getBean(TransferService.class);
```

@PropertySource

https://www.youtube.com/watch?v=FHdPFo5KWew

Spring Environment Abstraction :

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active.
Beans may be assigned to a profile whether defined in XML or via annotations.

Bean definition profiles is a mechanism in the core container that allows for registration of different beans in different environments.

XML bean definition profiles

The XML counterpart is the profile attribute of the <beans> element.

```
package com;
public class Person {
	private final String firstName;
	public Person(String firstName) {
		this.firstName = firstName;
	}
	public String getFirstName() {
		return firstName;
	}
}
```

Note : It is also possible to avoid that split and nest <beans/> elements within the same file:
The spring-bean.xsd has been constrained to allow such elements only as the last ones in the file.

```
	<beans profile="dev">
		<bean id="employee" class="com.Person">
			<constructor-arg value="John" />
		</bean>
	</beans>

	<beans profile="prod">
		<bean id="employee" class="com.Person">
			<constructor-arg value="Bert" />
		</bean>
	</beans>
	<beans profile="default">
		<bean id="employee" class="com.Person">
			<constructor-arg value="Pratap" />
		</bean>
	</beans>
```

https://www.youtube.com/watch?v=FHdPFo5KWew

Activating Profile :

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the Environment API which is available via an ApplicationContext

```
GenericXmlApplicationContextc = new GenericXmlApplicationContext();
ConfigurableEnvironment env = c.getEnvironment();
env.setActiveProfiles("dev");
c.load("com/config.xml");
c.refresh();
Person p = c.getBean("employee", Person.class);
System.out.println(p.getFirstName());
```

Spring.profiles.active:

Default profile

The default profile represents the profile that is enabled by default.

If no profile is active,this can be seen as a way to provide a default definition for one or more beans. If any profile is enabled, the default profile will not apply.

The name of the default profile can be changed using setDefaultProfiles() on the Environment or declaratively using the spring.profiles.default property.

Additional Capabilities of the ApplicationContext

The org.springframework.context package adds the ApplicationContext interface, which extends the BeanFactory interface, in addition to extending other interfaces to provide additional functionality in a more application framework-oriented style.

To enhance BeanFactory functionality in a more framework-oriented style the context package also provides the following functionality:

- **Access to messages in i18n-style, through the MessageSource interface.**
- **Access to resources, such as URLs and files, through the ResourceLoader interface.**
- **Event publication to namely beans implementing the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.**
- **Loading of multiple (hierarchical) contexts, allowing each to be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.**

Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource, and therefore provides internationalization (i18n) functionality.

The MessageSource interface supports 3 methods.

When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name messageSource. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the ApplicationContext attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the MessageSource.

Spring provides two MessageSource implementations, ResourceBundleMessageSource and StaticMessageSource.

```
<beans>
<bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
                <list>
                        <value>format</value>
                        <value>exceptions</value>
                        <value>windows</value>
```

```
            </list>
        </property>
</bean>
</beans>
```

you have three resource bundles defined in your classpath called format, exceptions and windows. Any request to resolve a message will be handled in the JDK standard way of resolving messages through ResourceBundles

format.properties
message=Alligators rock!

exceptions.properties
argument.required=The {0} argument is required.

Remember that all ApplicationContext implementations are also MessageSource implementations and so can be cast to the MessageSource interface.

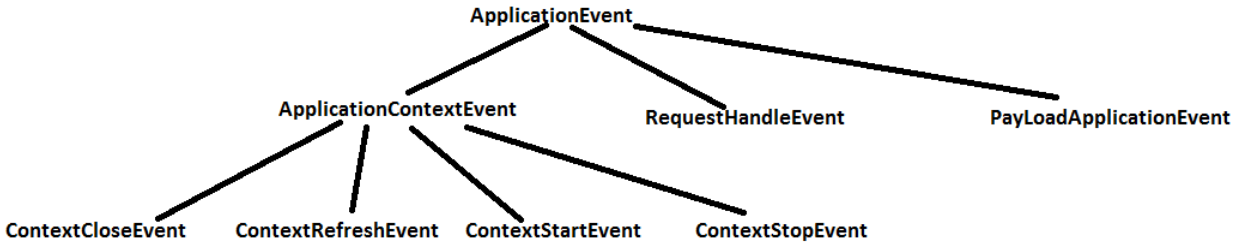
```
MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
String message = resources.getMessage("message", null, "Default", null);
System.out.println(message);
```

With regard to internationalization (i18n), Spring's various MessageSource implementations follow the same locale resolution and fallback rules as the standard JDK ResourceBundle. In short, and continuing with the example messageSource defined previously, if you want to resolve messages against the British (en-GB) locale, you would create files called format_en_GB.properties, exceptions_en_GB.properties, and windows_en_GB.properties respectively.

You can also use the MessageSourceAware interface to acquire a reference to any MessageSource that has been defined. Any bean that is defined in an ApplicationContext that implements the MessageSourceAware interface is injected with the application context's MessageSource when the bean is created and configured.

Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard Observer design pattern.



Event	Explanation
ContextRefreshedEvent	Published when the ApplicationContext is initialized or refreshed, for example, using the refresh() method on the ConfigurableApplicationContext interface. "Initialized" here means that all beans are loaded, post-processor beans are detected and activated, singletons are pre instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not.

ContextStartedEvent	Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContext interface. "Started" here means that all Lifecycle beans receive an explicit start signal. Typically this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart , for example, components that have not already started on initialization.

ContextStoppedEvent	Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContext interface. "Stopped" here means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call.

ContextClosedEvent	Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContext interface. "Closed" here means that all singleton beans are destroyed. A closed context reaches its end of life; it cannot be refreshed or restarted.

RequestHandledEvent	A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only

	applicable to web applications using Spring's DispatcherServlet.

Note : You can also create and publish your own custom events.

```
public class BlackListEvent extends ApplicationEvent {
        private final String address;
        private final String test;
        public BlackListEvent(Object source, String address, String test) {
                super(source);
                this.address = address;
                this.test = test;
        }
        // accessor and other methods...
}
```

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisher. Typically this is done by creating a class that implements ApplicationEventPublisherAware and registering it as a Spring bean.

```
public class EmailService implements ApplicationEventPublisherAware {
        private List<String> blackList;
        private ApplicationEventPublisher publisher;
        public void setBlackList(List<String> blackList) {
                this.blackList = blackList;
        }
public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
}
public void sendEmail(String address, String text) {
                if (blackList.contains(address)) {
                        BlackListEvent event = new BlackListEvent(this, address, text);
                        publisher.publishEvent(event);
                Return;
                }
                // send email…
        }
}
```

At configuration time, the Spring container will detect that EmailService implements ApplicationEventPublisherAware and will automatically call setApplicationEventPublisher()

To receive the custom ApplicationEvent, create a class that implements ApplicationListener and register it as a Spring bean.

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {
```

```
        public void onApplicationEvent(BlackListEvent event) {
                // notify appropriate parties via notificationAddress…
        }
}
<bean id="emailService" class="example.EmailService">
        <property name="blackList">
                <list>
                        <value>known.spammer@example.org</value>
                        <value>known.hacker@example.org</value>
                        <value>john.doe@example.org</value>
                </list>
        </property>
</bean>
<bean id="blackListNotifier" class="example.BlackListNotifier">
        <property name="notificationAddress" value="blacklist@example.org"/>
</bean>
```

Annotation-based Event Listeners :

As of Spring 4.2, an event listener can be registered on any public method of a managed bean via the EventListener annotation.

```
public class BlackListNotifier {
        private String notificationAddress;
        public void setNotificationAddress(String notificationAddress) {
                this.notificationAddress = notificationAddress;
        }
        @EventListener
        public void processBlackListEvent(BlackListEvent event) {
                // notify appropriate parties via notificationAddress…
        }
}
```

The method signature actually infer which even type it listens to.

If your method should listen to several events or if you want to define it with no parameter at all, the event type(s) can also be specified on the annotation itself:

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})
public void handleContextStart() {
}
```

It is also possible to add additional runtime filtering via the condition attribute of the annotation that defines a SpEL expression that should match to actually invoke the method for a particular event.

The above notifier can be rewritten to be only invoked if the test attribute of the event is equal to foo:

```
@EventListener(condition = "#event.test == 'foo'")
```

```
public void processBlackListEvent(BlackListEvent event) {
// notify appropriate parties via notificationAddress...
}
```

BeanFactory :

The BeanFactory provides the underlying basis for Spring's IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backwardcompatibility with the large number of third-party frameworks that integrate with Spring.

BeanFactory and ApplicationContext?

Because the ApplicationContext includes all functionality of the BeanFactory,

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	yes	yes
Automatic BeanPostProcessor registration	no	yes
Automatic BeanFactoryPostProcessor registration	no	yes
Convenient MessageSource access (for i18n)	no	yes
ApplicationEvent publication	no	yes