# Spring Framework - Overview

Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

Spring framework is an open source Java platform. It was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Spring is lightweight when it comes to size and transparency. The basic version of Spring framework is around 2MB.

The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications on top of the Java EE platform. Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.

## Benefits of Using the Spring Framework

# Following is the list of few of the great benefits of using Spring Framework −

# Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.

# Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.

# Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.

# Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.

# Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.

# Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

# Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.

# Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

## Dependency Injection (DI)

# The technology that Spring is most identified with is the Dependency Injection (DI) flavor of Inversion of Control. The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways. Dependency Injection is merely one concrete example of Inversion of Control.

# When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

# What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC.

# Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, we will explain this concept in a separate chapter with relevant example.

## Aspect Oriented Programming (AOP)

# One of the key components of Spring is the Aspect Oriented Programming (AOP) framework. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.

# The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect.

# The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. We will discuss more about Spring AOP concepts in a separate chapter.

# Spring Framework - Architecture

# Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

# The Spring Framework provides about 20 modules which can be used based on an application requirement.

# 

## Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

* The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
* The Bean module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
* The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
* The SpEL module provides a powerful expression language for querying and manipulating an object graph at runtime.

## Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

* The JDBC module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
* The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
* The OXM module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The Java Messaging Service JMS module contains features for producing and consuming messages.
* The Transaction module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

## Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

* The Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
* The Web-MVC module contains Spring's Model-View-Controller (MVC) implementation for web applications.
* The Web-Socket module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
* The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

## Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

* The AOP module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The Aspects module provides integration with AspectJ, which is again a powerful and mature AOP framework.
* The Instrumentation module provides class instrumentation support and class loader implementations to be used in certain application servers.
* The Messaging module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
* The Test module supports the testing of Spring components with JUnit or TestNG frameworks.

# Spring - IoC Containers

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

****

Spring provides the following two distinct types of containers.

|  |  |
| --- | --- |
| Sr.No. | Container & Description |
| 1 | [Spring BeanFactory Container](https://www.tutorialspoint.com/spring/spring_beanfactory_container.htm)  This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring. |
| 2 | [Spring ApplicationContext Container](https://www.tutorialspoint.com/spring/spring_applicationcontext_container.htm)  This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface. |

The *ApplicationContext* container includes all functionality of the *BeanFactory*container, so it is generally recommended over *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

# Spring - Bean Definition

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. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

Bean definition contains the information called configuration metadata, which is needed for the container to know the following −

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

|  |  |
| --- | --- |
| Sr.No. | Properties & Description |
| 1 | class  This attribute is mandatory and specifies the bean class to be used to create the bean. |
| 2 | name  This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). |
| 3 | scope  This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter. |
| 4 | constructor-arg  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 5 | properties  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 6 | autowiring mode  This is used to inject the dependencies and will be discussed in subsequent chapters. |
| 7 | lazy-initialization mode  A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup. |
| 8 | initialization method  A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter. |
| 9 | destruction method  A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter. |

## 

## Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. Following are the three important methods to provide configuration metadata to the Spring Container −

* XML based configuration file.
* Annotation-based configuration
* Java-based configuration

You already have seen how XML-based configuration metadata is provided to the container, but let us see another sample of XML-based configuration file with different bean definitions including lazy initialization, initialization method, and destruction method −

<?xml version = "1.0" encoding = "UTF-8"?>  
  
<beans xmlns = "http://www.springframework.org/schema/beans"  
 xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation = "http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
 <!-- A simple bean definition -->  
 <bean id = "..." class = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>  
  
 <!-- A bean definition with lazy init set on -->  
 <bean id = "..." class = "..." lazy-init = "true">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>

**lazy-init**

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.

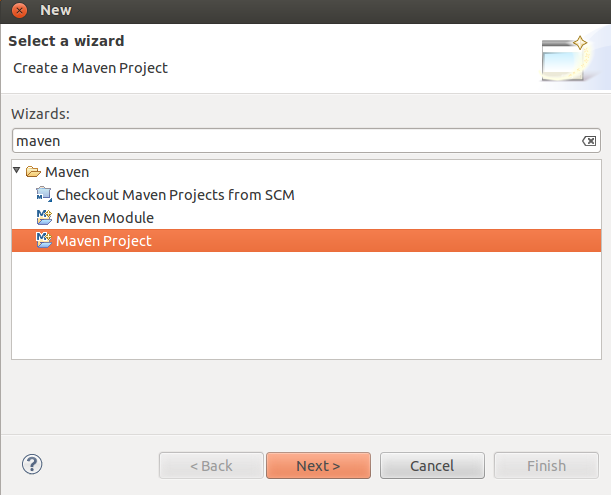
<!-- A bean definition with initialization method -->  
 <bean id = "..." class = "..." init-method = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>  
  
 <!-- A bean definition with destruction method -->  
 <bean id = "..." class = "..." destroy-method = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>  
  
 <!-- more bean definitions go here -->  
   
</beans>

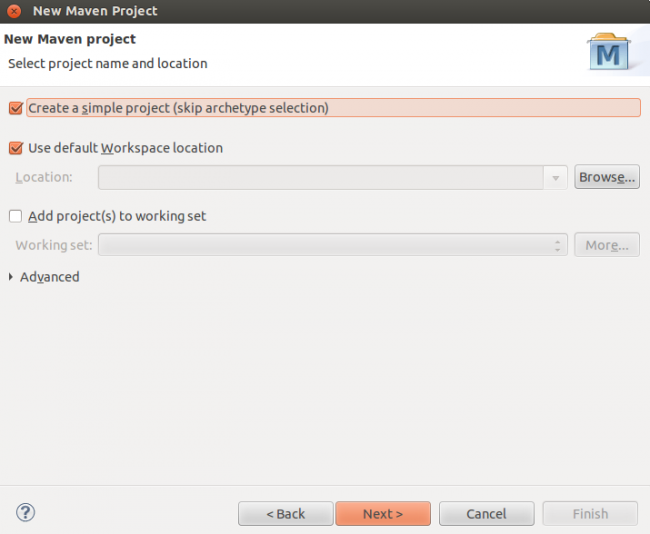
We will discuss about Annotation Based Configuration in a separate chapter. It is intentionally discussed in a separate chapter as we want you to grasp a few other important Spring concepts, before you start programming with Spring Dependency Injection with Annotations.

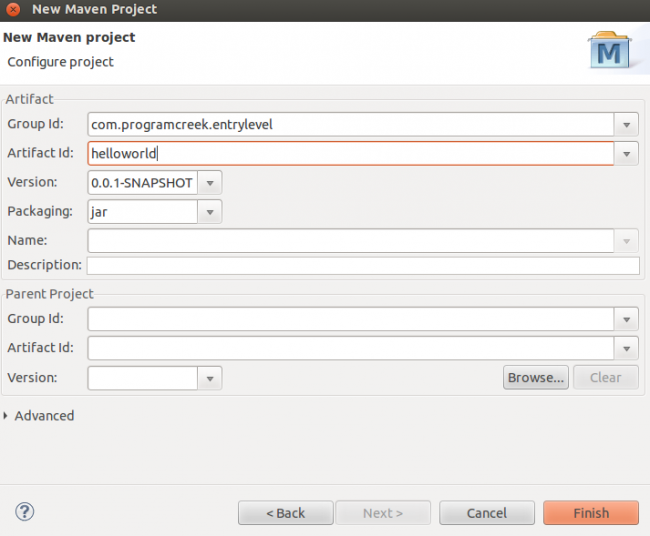
**Spring HelloWorld Example Using Eclipse and Maven**

**1. Set up a Maven Project**

Create a Maven project by using the Wizard.

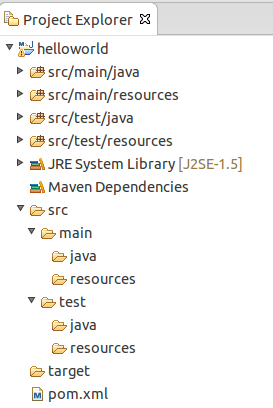






GroupId identifies the project uniquely across all projects, so we need to enforce a naming schema. ArtifactId is the name of the jar without version.

The created project under Project Explorer view likes the following:



**2. Add Spring Dependency**

Edit the pom.xml file and add dependencies of Spring packages. The pom.xml should be changed to:

**<project** xmlns="<http://maven.apache.org/POM/4.0.0>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0><http://maven.apache.org/xsd/maven-4.0.0.xsd>"**>**  
 **<modelVersion>**4.0.0**</modelVersion>**  
 **<groupId>**com.programcreek.entrylevel**</groupId>**  
 **<artifactId>**helloworld**</artifactId>**  
 **<version>**0.0.1-SNAPSHOT**</version>**  
   
 **<dependencies>**  
 **<dependency>**  
 **<groupId>**org.springframework**</groupId>**  
 **<artifactId>**spring-core**</artifactId>**  
 **<version>**${spring.version}**</version>**  
 **</dependency>**  
 **<dependency>**  
 **<groupId>**org.springframework**</groupId>**  
 **<artifactId>**spring-context**</artifactId>**  
 **<version>**${spring.version}**</version>**  
 **</dependency>**  
 **</dependencies>**  
   
 **<properties>**  
 **<spring.version>**3.2.3.RELEASE**</spring.version>**  
 **</properties>**  
**</project>**

**3. Create the Spring Bean Configuration File**

Create a bean configuation file named "applicationContext.xml" under src/main/resources directory. This xml file specifies the bean you will define in the project. Note the class is the fully qualified class name with the package.

**<beans** xmlns="<http://www.springframework.org/schema/beans>"  
 xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:p="<http://www.springframework.org/schema/p>"  
 xmlns:aop="<http://www.springframework.org/schema/aop>" xmlns:context="<http://www.springframework.org/schema/context>"  
 xmlns:jee="<http://www.springframework.org/schema/jee>" xmlns:tx="<http://www.springframework.org/schema/tx>"  
 xmlns:task="<http://www.springframework.org/schema/task>"  
 xsi:schemaLocation="<http://www.springframework.org/schema/aop><http://www.springframework.org/schema/aop/spring-aop-3.2.xsd><http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-3.2.xsd><http://www.springframework.org/schema/context><http://www.springframework.org/schema/context/spring-context-3.2.xsd><http://www.springframework.org/schema/jee><http://www.springframework.org/schema/jee/spring-jee-3.2.xsd><http://www.springframework.org/schema/tx><http://www.springframework.org/schema/tx/spring-tx-3.2.xsd><http://www.springframework.org/schema/task><http://www.springframework.org/schema/task/spring-task-3.2.xsd>"**>**  
   
 **<context:component-scan** base-package="com.programcreek.examples" **/>**   
 **<bean** id="helloWorldService"  
 class="com.programcreek.entrylevel.services.HelloWorldService"**>**  
 **<property** name="name" value="Program Creek Readers" **/>**  
 **</bean>**  
**</beans>**

**4. Create a Spring Bean**

Create HelloWorldService Bean under /src/main/java directory. The package should be the same with how you declared in the bean configuration file.

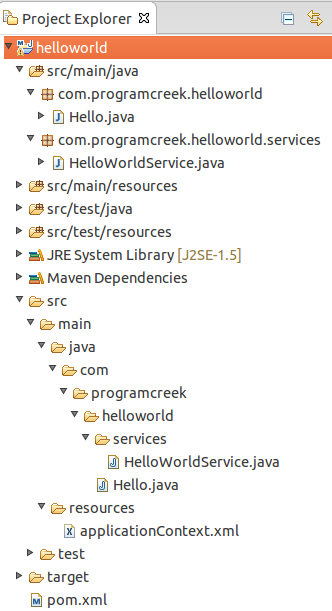
|  |
| --- |
| **package** com.programcreek.helloworld.services; **import** org.springframework.stereotype.Service;   @Service("helloWorldService") **public** **class** HelloWorldService {    **private** String name;    **public** **void** setName(String name) {  **this**.name = name;  }    **public** String sayHello() {  **return** "Hello! " + name;  } } |

5. Run the Maven Project

Create a Hello class under /src/main/java directory to test the project.

|  |
| --- |
| **package** com.programcreek.helloworld;   **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext; **import** com.programcreek.helloworld.services.HelloWorldService;    **public** **class** Hello {    @SuppressWarnings("resource")  **public** **static** **void** main(String[] args) {    *// loading the definitions from the given XML file*  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");    HelloWorldService service = (HelloWorldService) context  .getBean("helloWorldService");  String message = service.sayHello();  System.out.println(message);    *//set a new name*  service.setName("Spring");  message = service.sayHello();  System.out.println(message);  } } |

Finally, the Spring project should look like:



Right click Hello.java and run it as Java Application. The result is:

Hello! Program Creek Readers  
Hello! Spring

**6. Summary**

This is a simple example that shows how to create a Spring project by using Maven under Eclipse. The key idea of Spring framework is Inversion of Control (IoC). IoC refers to the programming style where a framework or runtime, controls the program flow. Inversion of control means we are changing the control from normal way.

**Loading the Context Definition (Instantiating a Container):**

The Application Context is Spring's advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. Additionally, it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by *org.springframework.context.ApplicationContext*interface.

The *ApplicationContext* includes all functionality of the *BeanFactory*, It is generally recommended over BeanFactory. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications.

The most commonly used ApplicationContext implementations are −

* **FileSystemXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** − This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.

Ex: ApplicationContext context = new FileSystemXmlApplicationContext("classpath:spring-app.xml");

* **WebXmlApplicationContext** − This container loads the XML file with definitions of all beans from within a web application.

We already have seen an example on ClassPathXmlApplicationContext container in *Spring Hello World Example*, and we will talk more about XmlWebApplicationContext in a separate chapter when we will discuss web-based Spring applications. So let us see one example on FileSystemXmlApplicationContext.

**Spring - Bean Scopes**

When defining a <bean> you have the option of declaring a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be prototype. Similarly, if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be singleton.

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware ApplicationContext.

|  |  |
| --- | --- |
| **Sr.No.** | **Scope & Description** |
| 1 | **Singleton**  This scopes the bean definition to a single instance per Spring IoC container (default). |
| 2 | **prototype**  This scopes a single bean definition to have any number of object instances. |
| 3 | **request**  This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| 4 | **session**  This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| 5 | **global-session**  This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

In this chapter, we will discuss about the first two scopes and the remaining three will be discussed when we discuss about web-aware Spring ApplicationContext.

**The singleton scope**

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the **scope**property to **singleton** in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

**Example**

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint*package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **HelloWorld.java** file −

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the **MainApp.java** file −

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.setMessage("I'm object A");

objA.getMessage();

HelloWorld objB = (HelloWorld) context.getBean("helloWorld");

objB.getMessage();

}

}

Following is the configuration file **Beans.xml** required for singleton scope −

<?xml version = "1.0" encoding = "UTF-8"?>

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

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

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

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

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" scope = "singleton">

</bean>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message : I'm object A

Your Message : I'm object A

**The prototype scope**

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the **scope** property to **prototype** in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with prototype scope -->

<bean id = "..." class = "..." scope = "prototype">

<!-- collaborators and configuration for this bean go here -->

</bean>

**Example**

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint*package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **HelloWorld.java** file

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the **MainApp.java** file −

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.setMessage("I'm object A");

objA.getMessage();

HelloWorld objB = (HelloWorld) context.getBean("helloWorld");

objB.getMessage();

}

}

Following is the configuration file **Beans.xml** required for prototype scope −

<?xml version = "1.0" encoding = "UTF-8"?>

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

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

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

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

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" scope = "prototype">

</bean>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message : I'm object A

Your Message : null

**Spring Beans Auto-wiring Example using XML Configuration**

Bean wiring corresponds to providing the dependencies a bean might need to complete its job. In Spring, beans can be wired together in two ways: **Manually** and **Autowiring**.

**Manual wiring**: using ref attribute in <property> or <constructor> tag

In this approach, we use the ‘ref’ attribute to refer to exact bean we want to be wired. This is the cleanest approach, and easy to understand

<!-- default example (autowire="no") -->

<bean id="driver" class="com.websystique.spring.domain.Driver">

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

</bean>

<bean id="license" class="com.websystique.spring.domain.License" >

    <property name="number" value="123456ABCD"/>

</bean>

**Autowiring** : using autowire attribute in <bean> tag

In this approach, beans can be automatically wired using Spring autowire feature. There are 4 supported options for autowiring.

* autowire="byName" : Autowiring using property name. If a bean found with same name as the property of other bean, this bean will be wired into other beans property
* autowire="byType" : If a bean found with same type as the type of property of other bean, this bean will be wired into other beans property
* autowire="constructor" : If a bean found with same type as the constructor argument of other bean, this bean will be wired into other bean constructor
* autowire="no" : No Autowiring. Same as explicitly specifying bean using ‘ref’ attribute

Let’s discuss each autowiring with an example:

#### 1. autowire=”byName” example

**Define Beans to work with**

|  |
| --- |
| package com.websystique.spring.domain;    public class Application {  private ApplicationUser applicationUser;  public ApplicationUser getApplicationUser() {  return applicationUser;  }  public void setApplicationUser(ApplicationUser applicationUser) {  this.applicationUser = applicationUser;  }  @Override  public String toString() {  return "Application [applicationUser=" + applicationUser + "]";  }  }  package com.websystique.spring.domain;  public class ApplicationUser {  private String name;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "ApplicationUser [name=" + name + "]";  }  } |
|  |

Application needs ApplicationUser dependency, and have a property named applicationUser.

**Spring Configuration XML file**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow">http://www.springframework.org/schema/beans</a>"  xmlns:xsi="<a class="vglnk" href="http://www.w3.org/2001/XMLSchema-instance" rel="nofollow">http://www.w3.org/2001/XMLSchema-instance</a>"  xmlns:context="<a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow">http://www.springframework.org/schema/context</a>"  xsi:schemaLocation="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow">[http://www.springframework.org/schema/beans</a](http://www.springframework.org/schema/beans%3c/a)>  <a class="vglnk" href="http://www.springframework.org/schema/beans/spring-beans-4.0.xsd" rel="nofollow"><http://www.springframework.org/schema/beans>  /spring-beans-4.0.xsd</a>  <a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow">[http://www.springframework.org/schema/context</a](http://www.springframework.org/schema/context%3c/a)>  <a class="vglnk" href="http://www.springframework.org/schema/context/spring-context-4.0.xsd" rel="nofollow"><http://www.springframework.org/schema/context>  /spring-context-4.0.xsd</a>">  <!-- byName example -->  <bean id="application" class="com.websystique.spring.domain.Application" autowire="byName"/>  <bean id="applicationUser" class="com.websystique.spring.domain.ApplicationUser" >  <property name="name" value="superUser"/>  </bean>  </beans> |

Notice that we have set the autowire="byName" attribute. We are nowhere referring to applicationUser bean explicitly. Since ‘application’ class have a property name ‘applicationUser’ which matches with the name of a bean defined in Spring context, that bean will be wired automatically.

**Run Application**  
Load the context and run it.

|  |
| --- |
| package com.websystique.spring;    import org.springframework.context.support.AbstractApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;    import com.websystique.spring.domain.Application;    public class AppMain {      public static void main(String args[]){          AbstractApplicationContext context = new ClassPathXmlApplicationContext("app-config.xml");            //autowire=byName          Application application = (Application)context.getBean("application");          System.out.println("Application Details : "+application);      }  } |

Following will be the output:

|  |
| --- |
| Application Details : Application [applicationUser=ApplicationUser [name=superUser]] |

#### 2. autowire=”byType” example

**Define Beans to work with**

|  |
| --- |
| package com.websystique.spring.domain;    public class Employee {        private EmployeeAddress address;        public EmployeeAddress getAddress() {          return address;      }        public void setAddress(EmployeeAddress address) {          this.address = address;      }        @Override      public String toString() {          return "Employee [address=" + address + "]";      }  } |
| package com.websystique.spring.domain;    public class EmployeeAddress {        private String Street;      private String city;        public String getStreet() {          return Street;      }        public void setStreet(String street) {          Street = street;      }        public String getCity() {          return city;      }        public void setCity(String city) {          this.city = city;      }        @Override      public String toString() {          return "EmployeeAddress [Street=" + Street + ", city=" + city + "]";      }  } |

Employee needs EmployeeAddress type dependency.

**Spring Configuration XML file**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow">http://www.springframework.org/schema/beans</a>"  xmlns:xsi="<a class="vglnk" href="http://www.w3.org/2001/XMLSchema-instance" rel="nofollow">[http://www.w3.org/2001/XMLSchema-instance</a](http://www.w3.org/2001/XMLSchema-instance%3c/a)>  " xmlns:context="<a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a>"  xsi:schemaLocation="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow"><http://www.springframework.org/schema>  /beans</a> <a class="vglnk" href="http://www.springframework.org/schema/beans/spring-beans-4.0.xsd" rel="nofollow"><http://www.springframework.org>  /schema/beans/spring-beans-4.0.xsd</a>  <a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a> <a class="vglnk" href="http://www.springframework.org/schema/context/spring-context-4.0.xsd" rel="nofollow"><http://www.springframework.org/schema>  /context/spring-context-4.0.xsd</a>">  <!-- byType example -->  <bean id="employee" class="com.websystique.spring.domain.Employee" autowire="byType"/>  <bean id="employeeAddress" class="com.websystique.spring.domain.EmployeeAddress" >  <property name="street" value="112/223,SantaVila"/>  <property name="city" value="Nebraska"/>  </bean>  </beans> |

Notice that we have set the autowire="byType" attribute. We are nowhere referring to employeeAddress bean explicitly from employee. Since ‘employee’ class have a property of type ‘EmployeeAddress’ which matches with the type of a bean ‘employeeAddress’ defined in Spring context, that bean will be wired automatically.

**Run Application**  
Load the context and run it.

|  |
| --- |
| package com.websystique.spring;    import org.springframework.context.support.AbstractApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;    import com.websystique.spring.domain.Employee;    public class AppMain {      public static void main(String args[]){          AbstractApplicationContext context = new ClassPathXmlApplicationContext("app-config.xml");            //autowire=byType          Employee employee = (Employee)context.getBean("employee");          System.out.println("Employee Details : "+employee);      }  } |

Following will be the output

|  |
| --- |
| Employee Details : Employee [address=EmployeeAddress [Street=112/223,SantaVila, city=Nebraska]] |

#### 3. autowire=”constructor” example

**Define Beans to work with**

|  |
| --- |
| package com.websystique.spring.domain;    public class Performer {        private Instrument instrument;        public Performer(Instrument instrument){          this.instrument = instrument;      }        @Override      public String toString() {          return "Performer [instrument=" + instrument + "]";      }  } |
| package com.websystique.spring.domain;    public class Instrument {        private String name;        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }        @Override      public String toString() {          return "Instrument [name=" + name + "]";      }  } |

Notice that ‘Performer’ class has a constructor which accepts a ‘Instrument’ type argument.

**Spring Configuration XML file**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow">http://www.springframework.org/schema/beans</a>"  xmlns:xsi="<a class="vglnk" href="http://www.w3.org/2001/XMLSchema-instance" rel="nofollow">[http://www.w3.org/2001/XMLSchema-instance</a](http://www.w3.org/2001/XMLSchema-instance%3c/a)>  " xmlns:context="<a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a>"  xsi:schemaLocation="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow"><http://www.springframework.org/schema>  /beans</a> <a class="vglnk" href="http://www.springframework.org/schema/beans/spring-beans-4.0.xsd" rel="nofollow"><http://www.springframework.org>  /schema/beans/spring-beans-4.0.xsd</a>  <a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a> <a class="vglnk" href="http://www.springframework.org/schema/context/spring-context-4.0.xsd" rel="nofollow"><http://www.springframework.org/schema>  /context/spring-context-4.0.xsd</a>">  <!-- constructor example -->  <bean id="performer" class="com.websystique.spring.domain.Performer" autowire="constructor"/>  <bean id="instrument" class="com.websystique.spring.domain.Instrument" >  <property name="name" value="PIANO"/>  </bean>  </beans> |

Notice that we have set the autowire="constructor" attribute. We are nowhere referring to instrument bean explicitly from performer. Since ‘performer’ class have a constructor which accepts an argument of type ‘Instrument’ which matches with the type of a bean ‘instrument’ defined in Spring context, that bean will be wired automatically.

**Run Application**  
Load the context and run it.

|  |
| --- |
| package com.websystique.spring;    import org.springframework.context.support.AbstractApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;    import com.websystique.spring.domain.Performer;    public class AppMain {      public static void main(String args[]){          AbstractApplicationContext context = new ClassPathXmlApplicationContext("app-config.xml");            //autowire=constructor          Performer performer = (Performer)context.getBean("performer");          System.out.println("Performer Details : "+performer);      }    } |

Following will be the output

|  |
| --- |
| Performer Details : Performer [instrument=Instrument [name=PIANO]] |

#### 3. autowire=”no” example

**Define Beans to work with**

|  |
| --- |
| package com.websystique.spring.domain;    public class Driver {        private License license;        public void setLicense(License license) {          this.license = license;      }        public License getLicense() {          return license;      }        @Override      public String toString() {          return "Driver [license=" + license + "]";      }  } |
| package com.websystique.spring.domain;    public class License {        private String number;        public String getNumber() {          return number;      }        public void setNumber(String number) {          this.number = number;      }        @Override      public String toString() {          return "License [number=" + number + "]";      }    } |

Notice that Driver has a dependency on License.

**Spring Configuration XML file**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow">http://www.springframework.org/schema/beans</a>"  xmlns:xsi="<a class="vglnk" href="http://www.w3.org/2001/XMLSchema-instance" rel="nofollow">[http://www.w3.org/2001/XMLSchema-instance</a](http://www.w3.org/2001/XMLSchema-instance%3c/a)>  " xmlns:context="<a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a>"  xsi:schemaLocation="<a class="vglnk" href="http://www.springframework.org/schema/beans" rel="nofollow"><http://www.springframework.org/schema/>  beans</a> <a class="vglnk" href="http://www.springframework.org/schema/beans/spring-beans-4.0.xsd" rel="nofollow"><http://www.springframework.org>  /schema/beans/spring-beans-4.0.xsd</a>  <a class="vglnk" href="http://www.springframework.org/schema/context" rel="nofollow"><http://www.springframework.org/schema/context>  </a> <a class="vglnk" href="http://www.springframework.org/schema/context/spring-context-4.0.xsd" rel="nofollow"><http://www.springframework.org/schema>  /context/spring-context-4.0.xsd</a>">  <!-- default example (autowire="no") -->  <bean id="driver" class="com.websystique.spring.domain.Driver" autowire="no">  <property name="license" ref="license"/>  </bean>  <bean id="license" class="com.websystique.spring.domain.License" >  <property name="number" value="123456ABCD"/>  </bean>  </beans> |

Notice that this time we have set autowire="no" attribute. This autowire attribute have no effect anymore on bean wiring , and can be removed altogether.There is no more auto-wiring here.

Also notice that we have set the ref atttibute to refer to a specific bean. If we don’t do that, driver’s license property will be null.

**Run Application.**  
Load the context and run it.

|  |
| --- |
| package com.websystique.spring;    import org.springframework.context.support.AbstractApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;    import com.websystique.spring.domain.Driver;    public class AppMain {      public static void main(String args[]){          AbstractApplicationContext context = new ClassPathXmlApplicationContext("app-config.xml");            //autowire=default          Driver driver = (Driver)context.getBean("driver");          System.out.println("Driver Details : "+driver);        }  } |

Following will be the output

|  |
| --- |
| Driver Details : Driver [license=License [number=123456ABCD]] |

That’s it. In the next post we will see Autowiring using Annotation based approach.

##### Limitations and Disadvantages of Autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

* Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
* Autowiring is less exact than explicit wiring. Although, as noted in the earlier table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results. The relationships between your Spring-managed objects are no longer documented explicitly.
* Wiring information may not be available to tools that may generate documentation from a Spring container.
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Map instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

**Spring - Bean Life Cycle**

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

Though, there are lists of the activities that take place behind the scene between the time of bean Instantiation and its destruction, this chapter will discuss only two important bean life cycle callback methods, which are required at the time of bean initialization and its destruction.

To define setup and teardown for a bean, we simply declare the <bean> with init method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

## Initialization callbacks

The org.springframework.beans.factory.InitializingBean interface specifies a single method −

void afterPropertiesSet() throws Exception;

Thus, you can simply implement the above interface and initialization work can be done inside afterPropertiesSet() method as follows −

public class ExampleBean implements InitializingBean {  
 public void afterPropertiesSet() {  
 // do some initialization work  
 }  
}

In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" init-method = "init"/>

Following is the class definition −

public class ExampleBean {  
 public void init() {  
 // do some initialization work  
 }  
}

## Destruction callbacks

The *org.springframework.beans.factory.DisposableBean* interface specifies a single method −

void destroy() throws Exception;

Thus, you can simply implement the above interface and finalization work can be done inside destroy() method as follows −

public class ExampleBean implements DisposableBean {  
 public void destroy() {  
 // do some destruction work  
 }  
}

In the case of XML-based configuration metadata, you can use the destroy-method attribute to specify the name of the method that has a void no-argument signature. For example −

<bean id = "exampleBean" class = "examples.ExampleBean" destroy-method = "destroy"/>

Following is the class definition −

public class ExampleBean {  
 public void destroy() {  
 // do some destruction work  
 }  
}

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

It is recommended that you do not use the InitializingBean or DisposableBean callbacks, because XML configuration gives much flexibility in terms of naming your method.

### Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint*package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of HelloWorld.java file −

package com.tutorialspoint;  
  
public class HelloWorld {  
 private String message;  
  
 public void setMessage(String message){  
 this.message = message;  
 }  
 public void getMessage(){  
 System.out.println("Your Message : " + message);  
 }  
 public void init(){  
 System.out.println("Bean is going through init.");  
 }  
 public void destroy() {  
 System.out.println("Bean will destroy now.");  
 }  
}

Following is the content of the MainApp.java file. Here you need to register a shutdown hook registerShutdownHook() method that is declared on the AbstractApplicationContext class. This will ensure a graceful shutdown and call the relevant destroy methods.

package com.tutorialspoint;  
  
import org.springframework.context.support.AbstractApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class MainApp {  
 public static void main(String[] args) {  
 AbstractApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");  
  
 HelloWorld obj = (HelloWorld) context.getBean("helloWorld");  
 obj.getMessage();  
 context.registerShutdownHook();  
 }  
}

Following is the configuration file Beans.xml required for init and destroy methods −

<?xml version = "1.0" encoding = "UTF-8"?>  
  
<beans xmlns = "http://www.springframework.org/schema/beans"  
 xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation = "http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
 <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" init-method = "init"   
 destroy-method = "destroy">  
 <property name = "message" value = "Hello World!"/>  
 </bean>  
  
</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Bean is going through init.  
Your Message : Hello World!  
Bean will destroy now.

## Default initialization and destroy methods

If you have too many beans having initialization and/or destroy methods with the same name, you don't need to declare init-method and destroy-method on each individual bean. Instead, the framework provides the flexibility to configure such situation using default-init-method and default-destroy-method attributes on the <beans> element as follows −

<beans xmlns = "http://www.springframework.org/schema/beans"  
 xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation = "http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"  
 default-init-method = "init"   
 default-destroy-method = "destroy">  
  
 <bean id = "..." class = "...">  
 <!-- collaborators and configuration for this bean go here -->  
 </bean>  
   
</beans>

### The util schema

First up is coverage of the util tags. As the name implies, the util tags deal with common, *utility* configuration issues, such as configuring collections, referencing constants, and suchlike.

To use the tags in the util schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the snippet below references the correct schema so that the tags in the util namespace are available to you.

<?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: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.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

#### <util:constant/>

Before...

<bean id="..." class="...">  
 <property name="isolation">  
 <bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"  
 class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />  
 </property>  
</bean>

The above configuration uses a Spring FactoryBean implementation, the FieldRetrievingFactoryBean, to set the value of the 'isolation' property on a bean to the value of the 'java.sql.Connection.TRANSACTION\_SERIALIZABLE' constant. This is all well and good, but it is a tad verbose and (unnecessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer's intent (*'inject this constant value'*), and it just reads better.

<bean id="..." class="...">  
 <property name="isolation">  
 <util:constant static-field="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>  
 </property>  
</bean>

##### Setting a bean property or constructor arg from a field value

[FieldRetrievingFactoryBean](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) is a FactoryBean which retrieves a static or non-static field value. It is typically used for retrieving public static finalconstants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a static field is exposed, by using the [staticField](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html#setStaticField(java.lang.String)) property:

<bean id="myField"  
 class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">  
 <property name="staticField" value="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>  
</bean>

There is also a convenience usage form where the static field is specified as the bean name:

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"  
 class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn't have to be specified for the bean reference:

<bean id="..." class="...">  
 <property name="isolation">  
 <bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"  
 class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />  
 </property>  
</bean>

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the [FieldRetrievingFactoryBean](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/FieldRetrievingFactoryBean.html) class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don't actually have to *do* anything or know anything about the Spring internals (or even about classes such as the FieldRetrievingFactoryBean). Let's look at an example to see how easy injecting an enum value is; consider this JDK 5 enum:

**package** javax.persistence;  
  
**public** enum PersistenceContextType {  
  
 TRANSACTION,  
 EXTENDED  
  
}

Now consider a setter of type PersistenceContextType:

**package** example;  
  
**public** **class** Client {  
  
 **private** PersistenceContextType persistenceContextType;  
  
 **public** **void** setPersistenceContextType(PersistenceContextType type) {  
 **this**.persistenceContextType = type;  
 }  
}

.. and the corresponding bean definition:

<bean class="example.Client">  
 <property name="persistenceContextType" value="TRANSACTION" />  
</bean>

This works for classic type-safe emulated enums (on JDK 1.4 and JDK 1.3) as well; Spring will automatically attempt to match the string property value to a constant on the enum class.

#### <util:property-path/>

Before...

*<!-- target bean to be referenced by name -->*  
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">  
 <property name="age" value="10"/>  
 <property name="spouse">  
 <bean class="org.springframework.beans.TestBean">  
 <property name="age" value="11"/>  
 </bean>  
 </property>  
</bean>  
  
*<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->*  
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

The above configuration uses a Spring FactoryBean implementation, the PropertyPathFactoryBean, to create a bean (of type int) called 'testBean.age' that has a value equal to the 'age' property of the 'testBean' bean.

After...

*<!-- target bean to be referenced by name -->*  
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">  
 <property name="age" value="10"/>  
 <property name="spouse">  
 <bean class="org.springframework.beans.TestBean">  
 <property name="age" value="11"/>  
 </bean>  
 </property>  
</bean>  
  
*<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->*  
<util:property-path id="name" path="testBean.age"/>

The value of the 'path' attribute of the <property-path/> tag follows the form 'beanName.beanProperty'.

##### Using <util:property-path/> to set a bean property or constructor-argument

PropertyPathFactoryBean is a FactoryBean that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here's an example where a path is used against another bean, by name:

// target bean to be referenced by name  
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">  
 <property name="age" value="10"/>  
 <property name="spouse">  
 <bean class="org.springframework.beans.TestBean">  
 <property name="age" value="11"/>  
 </bean>  
 </property>  
</bean>  
  
*// will result in 11, which is the value of property 'spouse.age' of bean 'person'*  
<bean id="theAge"  
 class="org.springframework.beans.factory.config.PropertyPathFactoryBean">  
 <property name="targetBeanName" value="person"/>  
 <property name="propertyPath" value="spouse.age"/>  
</bean>

In this example, a path is evaluated against an inner bean:

*<!-- will result in 12, which is the value of property 'age' of the inner bean -->*  
<bean id="theAge"  
 class="org.springframework.beans.factory.config.PropertyPathFactoryBean">  
 <property name="targetObject">  
 <bean class="org.springframework.beans.TestBean">  
 <property name="age" value="12"/>  
 </bean>  
 </property>  
 <property name="propertyPath" value="age"/>  
</bean>

There is also a shortcut form, where the bean name is the property path.

*<!-- will result in 10, which is the value of property 'age' of bean 'person' -->*  
<bean id="person.age"  
 class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

<bean id="..." class="...">  
 <property name="age">  
 <bean id="person.age"  
 class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>  
 </property>  
</bean>

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

#### <util:properties/>

Before...

*<!-- creates a java.util.Properties instance with values loaded from the supplied location -->*  
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">  
 <property name="location" value="classpath:com/foo/jdbc-production.properties"/>  
</bean>

The above configuration uses a Spring FactoryBean implementation, the PropertiesFactoryBean, to instantiate a java.util.Properties instance with values loaded from the supplied [Resource](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html) location).

After...

*<!-- creates a java.util.Properties instance with values loaded from the supplied location -->*  
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>

#### <util:list/>

Before...

*<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->*  
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">  
 <property name="sourceList">  
 <list>  
 <value>pechorin@hero.org</value>  
 <value>raskolnikov@slums.org</value>  
 <value>stavrogin@gov.org</value>  
 <value>porfiry@gov.org</value>  
 </list>  
 </property>  
</bean>

The above configuration uses a Spring FactoryBean implementation, the ListFactoryBean, to create a java.util.List instance initialized with values taken from the supplied 'sourceList'.

After...

*<!-- creates a java.util.List instance with the supplied values -->*  
<util:list id="emails">  
 <value>pechorin@hero.org</value>  
 <value>raskolnikov@slums.org</value>  
 <value>stavrogin@gov.org</value>  
 <value>porfiry@gov.org</value>  
</util:list>

You can also explicitly control the exact type of List that will be instantiated and populated via the use of the 'list-class' attribute on the <util:list/> element. For example, if we really need a java.util.LinkedList to be instantiated, we could use the following configuration:

<util:list id="emails" list-class="java.util.LinkedList">  
 <value>jackshaftoe@vagabond.org</value>  
 <value>eliza@thinkingmanscrumpet.org</value>  
 <value>vanhoek@pirate.org</value>  
 <value>d'Arcachon@nemesis.org</value>  
</util:list>

If no 'list-class' attribute is supplied, a List implementation will be chosen by the container.

#### <util:map/>

Before...

*<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->*  
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">  
 <property name="sourceMap">  
 <map>  
 <entry key="pechorin" value="pechorin@hero.org"/>  
 <entry key="raskolnikov" value="raskolnikov@slums.org"/>  
 <entry key="stavrogin" value="stavrogin@gov.org"/>  
 <entry key="porfiry" value="porfiry@gov.org"/>  
 </map>  
 </property>  
</bean>

The above configuration uses a Spring FactoryBean implementation, the MapFactoryBean, to create a java.util.Map instance initialized with key-value pairs taken from the supplied 'sourceMap'.

After...

*<!-- creates a java.util.Map instance with the supplied key-value pairs -->*  
<util:map id="emails">  
 <entry key="pechorin" value="pechorin@hero.org"/>  
 <entry key="raskolnikov" value="raskolnikov@slums.org"/>  
 <entry key="stavrogin" value="stavrogin@gov.org"/>  
 <entry key="porfiry" value="porfiry@gov.org"/>  
</util:map>

You can also explicitly control the exact type of Map that will be instantiated and populated via the use of the 'map-class' attribute on the <util:map/> element. For example, if we really need a java.util.TreeMap to be instantiated, we could use the following configuration:

<util:map id="emails" map-class="java.util.TreeMap">  
 <entry key="pechorin" value="pechorin@hero.org"/>  
 <entry key="raskolnikov" value="raskolnikov@slums.org"/>  
 <entry key="stavrogin" value="stavrogin@gov.org"/>  
 <entry key="porfiry" value="porfiry@gov.org"/>  
</util:map>

If no 'map-class' attribute is supplied, a Map implementation will be chosen by the container.

#### <util:set/>

Before...

*<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->*  
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">  
 <property name="sourceSet">  
 <set>  
 <value>pechorin@hero.org</value>  
 <value>raskolnikov@slums.org</value>  
 <value>stavrogin@gov.org</value>  
 <value>porfiry@gov.org</value>  
 </set>  
 </property>  
</bean>

The above configuration uses a Spring FactoryBean implementation, the SetFactoryBean, to create a java.util.Set instance initialized with values taken from the supplied 'sourceSet'.

After...

*<!-- creates a java.util.Set instance with the supplied values -->*  
<util:set id="emails">  
 <value>pechorin@hero.org</value>  
 <value>raskolnikov@slums.org</value>  
 <value>stavrogin@gov.org</value>  
 <value>porfiry@gov.org</value>  
</util:set>

You can also explicitly control the exact type of Set that will be instantiated and populated via the use of the 'set-class' attribute on the <util:set/> element. For example, if we really need a java.util.TreeSet to be instantiated, we could use the following configuration:

<util:set id="emails" set-class="java.util.TreeSet">  
 <value>pechorin@hero.org</value>  
 <value>raskolnikov@slums.org</value>  
 <value>stavrogin@gov.org</value>  
 <value>porfiry@gov.org</value>  
</util:set>

If no 'set-class' attribute is supplied, a Set implementation will be chosen by the container.

### The jee schema

The jee tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the jee schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jee namespace are available to you.

<?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:jee="http://www.springframework.org/schema/jee"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

#### <jee:jndi-lookup/> (simple)

Before...

<bean id="**dataSource**" class="org.springframework.jndi.JndiObjectFactoryBean">  
 <property name="jndiName" value="jdbc/MyDataSource"/>  
</bean>  
  
<bean id="userDao" class="com.foo.JdbcUserDao">  
 *<!-- Spring will do the cast automatically (as usual) -->*  
 <property name="dataSource" ref="**dataSource**"/>  
</bean>

After...

<jee:jndi-lookup id="**dataSource**" jndi-name="jdbc/MyDataSource"/>  
  
<bean id="userDao" class="com.foo.JdbcUserDao">  
 *<!-- Spring will do the cast automatically (as usual) -->*  
 <property name="dataSource" ref="**dataSource**"/>  
</bean>

#### <jee:jndi-lookup/> (with single JNDI environment setting)

Before...

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">  
 <property name="jndiName" value="jdbc/MyDataSource"/>  
 <property name="jndiEnvironment">  
 <props>  
 <prop key="foo">bar</prop>  
 </props>  
 </property>  
</bean>

After...

<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">  
 <jee:environment>foo=bar</jee:environment>  
</jee:jndi-lookup>

#### <jee:jndi-lookup/> (with multiple JNDI environment settings)

Before...

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">  
 <property name="jndiName" value="jdbc/MyDataSource"/>  
 <property name="jndiEnvironment">  
 <props>  
 <prop key="foo">bar</prop>  
 <prop key="ping">pong</prop>  
 </props>  
 </property>  
</bean>

After...

<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">  
 *<!-- newline-separated, key-value pairs for the environment (standard Properties format) -->*  
 <jee:environment>  
 foo=bar  
 ping=pong  
 </jee:environment>  
</jee:jndi-lookup>

#### <jee:jndi-lookup/> (complex)

Before...

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">  
 <property name="jndiName" value="jdbc/MyDataSource"/>  
 <property name="cache" value="true"/>  
 <property name="resourceRef" value="true"/>  
 <property name="lookupOnStartup" value="false"/>  
 <property name="expectedType" value="com.myapp.DefaultFoo"/>  
 <property name="proxyInterface" value="com.myapp.Foo"/>  
</bean>

After...

<jee:jndi-lookup id="simple"  
 jndi-name="jdbc/MyDataSource"  
 cache="true"  
 resource-ref="true"  
 lookup-on-startup="false"  
 expected-type="com.myapp.DefaultFoo"  
 proxy-interface="com.myapp.Foo"/>

#### <jee:local-slsb/> (simple)

The <jee:local-slsb/> tag configures a reference to an EJB Stateless SessionBean.

Before...

<bean id="simple"  
 class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
 <property name="jndiName" value="ejb/RentalServiceBean"/>  
 <property name="businessInterface" value="com.foo.service.RentalService"/>  
</bean>

After...

<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"  
 business-interface="com.foo.service.RentalService"/>

#### <jee:local-slsb/> (complex)

<bean id="complexLocalEjb"  
 class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
 <property name="jndiName" value="ejb/RentalServiceBean"/>  
 <property name="businessInterface" value="com.foo.service.RentalService"/>  
 <property name="cacheHome" value="true"/>  
 <property name="lookupHomeOnStartup" value="true"/>  
 <property name="resourceRef" value="true"/>  
</bean>

After...

<jee:local-slsb id="complexLocalEjb"  
 jndi-name="ejb/RentalServiceBean"  
 business-interface="com.foo.service.RentalService"  
 cache-home="true"  
 lookup-home-on-startup="true"  
 resource-ref="true">

#### <jee:remote-slsb/>

The <jee:remote-slsb/> tag configures a reference to a remote EJB Stateless SessionBean.

Before...

<bean id="complexRemoteEjb"  
 class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">  
 <property name="jndiName" value="ejb/MyRemoteBean"/>  
 <property name="businessInterface" value="com.foo.service.RentalService"/>  
 <property name="cacheHome" value="true"/>  
 <property name="lookupHomeOnStartup" value="true"/>  
 <property name="resourceRef" value="true"/>  
 <property name="homeInterface" value="com.foo.service.RentalService"/>  
 <property name="refreshHomeOnConnectFailure" value="true"/>  
</bean>

After...

<jee:remote-slsb id="complexRemoteEjb"  
 jndi-name="ejb/MyRemoteBean"  
 business-interface="com.foo.service.RentalService"  
 cache-home="true"  
 lookup-home-on-startup="true"  
 resource-ref="true"  
 home-interface="com.foo.service.RentalService"  
 refresh-home-on-connect-failure="true">

### The lang schema

The lang tags deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

These tags (and the dynamic language support) are comprehensively covered in the chapter entitled [Chapter 28, *Dynamic language support*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/dynamic-language.html). Please do consult that chapter for full details on this support and the lang tags themselves.

In the interest of completeness, to use the tags in the lang schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the lang namespace are available to you.

<?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:lang="http://www.springframework.org/schema/lang"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

### The jms schema

The jms tags deal with configuring JMS-related beans such as Spring's [MessageListenerContainers](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jms.html#jms-mdp). These tags are detailed in the section of the [JMS chapter](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jms.html) entitled [Section 23.6, “JMS Namespace Support”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jms.html#jms-namespace). Please do consult that chapter for full details on this support and the jms tags themselves.

In the interest of completeness, to use the tags in the jms schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jms namespace are available to you.

<?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:jms="http://www.springframework.org/schema/jms"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

### The tx (transaction) schema

The tx tags deal with configuring all of those beans in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled [Chapter 12, *Transaction Management*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/transaction.html).

|  |
| --- |
| [Tip] |
| You are strongly encouraged to look at the 'spring-tx.xsd' file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring's transaction configuration, and covers all of the various tags in the tx namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle. |

In the interest of completeness, to use the tags in the tx schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the tx namespace are available to you.

<?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:aop="http://www.springframework.org/schema/aop"  
 **xmlns:tx="http://www.springframework.org/schema/tx"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd**  
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">  
  
*<!-- bean definitions here -->*  
  
</beans>

|  |
| --- |
| [Note] |
| Often when using the tags in the tx namespace you will also be using the tags from the aop namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the aop schema so that the tags in the aopnamespace are available to you. |

### The aop schema

The aop tags deal with configuring all things AOP in Spring: this includes Spring's own proxy-based AOP framework and Spring's integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled [Chapter 9, *Aspect Oriented Programming with Spring*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html).

In the interest of completeness, to use the tags in the aop schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the aop namespace are available to you.

<?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:aop="http://www.springframework.org/schema/aop"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

### The context schema

The context tags deal with ApplicationContext configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as BeanfactoryPostProcessors. The following snippet references the correct schema so that the tags in the contextnamespace are available to you.

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 **xmlns:context="http://www.springframework.org/schema/context"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

|  |
| --- |
| [Note] |
| The context schema was only introduced in Spring 2.5. |

#### <property-placeholder/>

This element activates the replacement of ${...} placeholders, resolved against the specified properties file (as a [Spring resource location](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html)). This element is a convenience mechanism that sets up a [PropertyPlaceholderConfigurer](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-placeholderconfigurer) for you; if you need more control over the PropertyPlaceholderConfigurer, just define one yourself explicitly.

#### <annotation-config/>

Activates the Spring infrastructure for various annotations to be detected in bean classes: Spring's [@Required](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-required-annotation) and [@Autowired](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-annotation-config), as well as JSR 250's @PostConstruct, @PreDestroy and @Resource (if available), and JPA's @PersistenceContext and @PersistenceUnit (if available). Alternatively, you can choose to activate the individual BeanPostProcessors for those annotations explicitly.

|  |
| --- |
| [Note] |
| This element does *not* activate processing of Spring's [@Transactional](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/transaction.html#transaction-declarative-annotations) annotation. Use the [<tx:annotation-driven/>](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/transaction.html#tx-decl-explained) element for that purpose. |

#### <component-scan/>

This element is detailed in [Section 5.9, “Annotation-based container configuration”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-annotation-config).

#### <load-time-weaver/>

This element is detailed in [Section 9.8.4, “Load-time weaving with AspectJ in the Spring Framework”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-aj-ltw).

#### <spring-configured/>

This element is detailed in [Section 9.8.1, “Using AspectJ to dependency inject domain objects with Spring”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-atconfigurable).

#### <mbean-export/>

This element is detailed in [Section 24.4.3, “Configuring annotation based MBean export”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jmx.html#jmx-context-mbeanexport).

### The tool schema

The tool tags are for use when you want to add tooling-specific metadata to your custom configuration elements. This metadata can then be consumed by tools that are aware of this metadata, and the tools can then do pretty much whatever they want with it (validation, etc.).

The tool tags are not documented in this release of Spring as they are currently undergoing review. If you are a third party tool vendor and you would like to contribute to this review process, then do mail the Spring mailing list. The currently supported tool tags can be found in the file 'spring-tool.xsd' in the'src/org/springframework/beans/factory/xml' directory of the Spring source distribution.

### The jdbc schema

The jdbc tags allow you to quickly configure an embedded database or initialize an existing data source. These tags are documented in [Section 14.8, “Embedded database support”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jdbc.html#jdbc-embedded-database-support) and [Section 14.9, “Initializing a DataSource”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jdbc.html#jdbc-intializing-datasource) respectively.

To use the tags in the jdbc schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jdbc namespace are available to you.

<?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:jdbc="http://www.springframework.org/schema/jdbc"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

### The cache schema

The cache tags can be used to enable support for Spring's @CacheEvict, @CachePut and @Caching annotations. It it also supports declarative XML-based caching. See [Section 29.3.5, “Enable caching annotations”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/cache.html#cache-annotation-enable) and [Section 29.4, “Declarative XML-based caching”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/cache.html#cache-declarative-xml) for details.

To use the tags in the cache schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the cache namespace are available to you.

<?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:jdbc="http://www.springframework.org/schema/cache"**  
 xsi:schemaLocation="  
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  
**http://www.springframework.org/schema/cache http://www.springframework.org/schema/jdbc/spring-cache.xsd"**>  
  
*<!-- bean definitions here -->*  
  
</beans>

### The beans schema

Last but not least we have the tags in the beans schema. These are the same tags that have been in Spring since the very dawn of the framework. Examples of the various tags in the beans schema are not shown here because they are quite comprehensively covered in [Section 5.4.2, “Dependencies and configuration in detail”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-properties-detailed)(and indeed in that entire [chapter](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html)).

One thing that is new to the beans tags themselves in Spring 2.0 is the idea of arbitrary bean metadata. In Spring 2.0 it is now possible to add zero or more key / value pairs to <bean/> XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you are writing your own custom tags as described in the appendix entitled [Appendix F, *Extensible XML authoring*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/extensible-xml.html)).

Find below an example of the <meta/> tag in the context of a surrounding <bean/> (please note that without any logic to interpret it the metadata is effectively useless as-is).

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="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">  
  
 <bean id="foo" class="x.y.Foo">  
 **<meta key="cacheName" value="foo"/>**  
 <property name="name" value="Rick"/>  
 </bean>  
  
</beans>

In the case of the above example, you would assume that there is some logic that will consume the bean definition and set up some caching infrastructure using the supplied metadata.