Spring Tutorial

Open source

written by Rod Johnson

released in June 2003 by Apache 2.0

Spring is lightweight

## **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, the 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.

## **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.

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 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.

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. These objects are called Spring Beans.

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 Beans**

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.

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

<?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>

<!-- 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>

**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. |

## **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.**

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

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

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

</bean>

**Example :-**

**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>

**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

**Bean Life Cycle**

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.

To define setup and teardown for a bean, we simply declare the <bean> with **initmethod** and/or **destroy-method** parameters.

The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation.

Similarly, destroy method 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 :

**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>

**Bean Post Processors:**

The **BeanPostProcessor** interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc.

You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor interfaces and you can control the order in which these BeanPostProcessor interfaces execute by setting the **order** property provided the BeanPostProcessor implements the **Ordered**interface.

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An **ApplicationContext** automatically detects any beans that are defined with the implementation of the **BeanPostProcessor** interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation.

Example :

**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.");

}

}

This is a very basic example of implementing BeanPostProcessor, which prints a bean name before and after initialization of any bean. You can implement more complex logic before and after intializing a bean because you have access on bean object inside both the post processor methods.

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

package com.tutorialspoint;

import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.beans.BeansException;

public class InitHelloWorld implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("BeforeInitialization : " + beanName);

return bean; // you can return any other object as well

}

public Object postProcessAfterInitialization(Object bean, String beanName)

throws BeansException {

System.out.println("AfterInitialization : " + beanName);

return bean; // you can return any other object as well

}

}

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 ensures a graceful shutdown and calls 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>

<bean class = "com.tutorialspoint.InitHelloWorld" />

</beans>

Once you are done with 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 −

BeforeInitialization : helloWorld

Bean is going through init.

AfterInitialization : helloWorld

Your Message : Hello World!

Bean will destroy now.

Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

Spring Bean definition inheritance has nothing to do with Java class inheritance but the inheritance concept is same.

You can define a parent bean definition as a template and other child beans can inherit the required configuration from the parent bean.

When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent** attribute, specifying the parent bean as the value of this attribute.

Example :

Following is the configuration file **Beans.xml** where we defined "helloWorld" bean which has two properties *message1* and *message2*. Next "helloIndia" bean has been defined as a child of "helloWorld" bean by using **parent**attribute. The child bean inherits *message2* property as is, and overrides *message1* property and introduces one more property *message3*.

<?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">

<property name = "message1" value = "Hello World!"/>

<property name = "message2" value = "Hello Second World!"/>

</bean>

<bean id ="helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "helloWorld">

<property name = "message1" value = "Hello India!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

</beans>

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

package com.tutorialspoint;

public class HelloWorld {

private String message1;

private String message2;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void getMessage1(){

System.out.println("World Message1 : " + message1);

}

public void getMessage2(){

System.out.println("World Message2 : " + message2);

}

}

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

package com.tutorialspoint;

public class HelloIndia {

private String message1;

private String message2;

private String message3;

public void setMessage1(String message){

this.message1 = message;

}

public void setMessage2(String message){

this.message2 = message;

}

public void setMessage3(String message){

this.message3 = message;

}

public void getMessage1(){

System.out.println("India Message1 : " + message1);

}

public void getMessage2(){

System.out.println("India Message2 : " + message2);

}

public void getMessage3(){

System.out.println("India Message3 : " + message3);

}

}

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.getMessage1();

objA.getMessage2();

HelloIndia objB = (HelloIndia) context.getBean("helloIndia");

objB.getMessage1();

objB.getMessage2();

objB.getMessage3();

}

}

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 −

World Message1 : Hello World!

World Message2 : Hello Second World!

India Message1 : Hello India!

India Message2 : Hello Second World!

India Message3 : Namaste India!

If you observed here, we did not pass message2 while creating "helloIndia" bean, but it got passed because of Bean Definition Inheritance.

## **Bean Definition Template**

You can create a Bean definition template, which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should not specify the **class** attribute and should specify **abstract** attribute and should specify the abstract attribute with a value of **true** as shown in the following code snippet −

<?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 = "beanTeamplate" abstract = "true">

<property name = "message1" value = "Hello World!"/>

<property name = "message2" value = "Hello Second World!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

<bean id = "helloIndia" class = "com.tutorialspoint.HelloIndia" parent = "beanTeamplate">

<property name = "message1" value = "Hello India!"/>

<property name = "message3" value = "Namaste India!"/>

</bean>

</beans>

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as *abstract*. When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions.

Dependency Injection:

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. 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 (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this −

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

spellChecker = new SpellChecker();

}

}

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this −

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**.

Thus the flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

[**Constructor-based dependency injection**](https://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the constructor-based injection −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<constructor-arg ref = "spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></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 −

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **Constructor arguments resolution**

There may be an ambiguity while passing arguments to the constructor, in case there are more than one parameters. To resolve this ambiguity, 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. Consider the following class −

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {

// ...

}

}

The following configuration works fine −

<beans>

<bean id = "foo" class = "x.y.Foo">

<constructor-arg ref = "bar"/>

<constructor-arg ref = "baz"/>

</bean>

<bean id = "bar" class = "x.y.Bar"/>

<bean id = "baz" class = "x.y.Baz"/>

</beans>

Let us check one more case where we pass different types to the constructor. Consider the following class −

package x.y;

public class Foo {

public Foo(int year, String name) {

// ...

}

}

The container can also use type matching with simple types, if you explicitly specify the type of the constructor argument using the type attribute. For example −

<beans>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg type = "int" value = "2001"/>

<constructor-arg type = "java.lang.String" value = "Zara"/>

</bean>

</beans>

Finally, the best way to pass constructor arguments, use the index attribute to specify explicitly the index of constructor arguments. Here, the index is 0 based. For example −

<beans>

<bean id = "exampleBean" class = "examples.ExampleBean">

<constructor-arg index = "0" value = "2001"/>

<constructor-arg index = "1" value = "Zara"/>

</bean>

</beans>

A final note, in case you are passing a reference to an object, you need to use **ref** attribute of <constructor-arg> tag and if you are passing a value directly then you should use **value** attribute as shown above.

[**Setter-based dependency injection**](https://www.tutorialspoint.com/spring/setter_based_dependency_injection.htm)

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Here you need to check the naming convention of the setter methods. To set a variable **spellChecker** we are using **setSpellChecker()** method which is very similar to Java POJO classes. Let us create the content of another dependent class file **SpellChecker.java** −

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the setter-based injection −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

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

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

You should note the difference in Beans.xml file defined in the constructor-based injection and the setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

The second important point to note is that in case you are passing a reference to an object, you need to use **ref** attribute of <property> tag and if you are passing a **value** directly then you should use value attribute.

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

Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.

## **XML Configuration using p-namespace**

If you have many setter methods, then it is convenient to use **p-namespace**in the XML configuration file. Let us check the difference −

Let us consider the example of a standard XML configuration file with <property> tags −

<?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 = "john-classic" class = "com.example.Person">

<property name = "name" value = "John Doe"/>

<property name = "spouse" ref = "jane"/>

</bean>

<bean name = "jane" class = "com.example.Person">

<property name = "name" value = "John Doe"/>

</bean>

</beans>

The above XML configuration can be re-written in a cleaner way using p-namespace as follows −

<?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:p = "http://www.springframework.org/schema/p"

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

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

<bean id = "john-classic" class = "com.example.Person"

p:name = "John Doe"

p:spouse-ref = "jane"/>

</bean>

<bean name =" jane" class = "com.example.Person"

p:name = "John Doe"/>

</bean>

</beans>

Here, you should note the difference in specifying primitive values and object references with p-namespace. The **-ref** part indicates that this is not a straight value but rather a reference to another bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

The code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies.

The object does not look up its dependencies and does not know the location or class of the dependencies, rather everything is taken care by the Spring Framework.

Injecting Inner Beans:

As you know Java inner classes are defined within the scope of other classes, similarly, **inner beans** are beans that are defined within the scope of another bean.

<?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 = "outerBean" class = "...">

<property name = "target">

<bean id = "innerBean" class = "..."/>

</property>

</bean>

</beans>

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java** −

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the setter-based injection but using **inner beans** −

<?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">

<!-- Definition for textEditor bean using inner bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<property name = "spellChecker">

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"/>

</property>

</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 −

Inside SpellChecker constructor.

Inside setSpellChecker.

Inside checkSpelling.

Injecting Collection:

You have seen how to configure primitive data type using **value** attribute and object references using **ref** attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.

Spring offers four types of collection configuration elements which are as follows −

|  |  |
| --- | --- |
| **Sr.No** | **Element & Description** |
| 1 | **<list>**  This helps in wiring ie injecting a list of values, allowing duplicates. |
| 2 | **<set>**  This helps in wiring a set of values but without any duplicates. |
| 3 | **<map>**  This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| 4 | **<props>**  This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

You can use either <list> or <set> to wire any implementation of java.util.Collection or an **array**.

You will come across two situations

(a) Passing direct values of the collection and

(b) Passing a reference of a bean as one of the collection elements.

Example :

**JavaCollection.java** file −

package com.tutorialspoint;

import java.util.\*;

public class JavaCollection {

List addressList;

Set addressSet;

Map addressMap;

Properties addressProp;

// a setter method to set List

public void setAddressList(List addressList) {

this.addressList = addressList;

}

// prints and returns all the elements of the list.

public List getAddressList() {

System.out.println("List Elements :" + addressList);

return addressList;

}

// a setter method to set Set

public void setAddressSet(Set addressSet) {

this.addressSet = addressSet;

}

// prints and returns all the elements of the Set.

public Set getAddressSet() {

System.out.println("Set Elements :" + addressSet);

return addressSet;

}

// a setter method to set Map

public void setAddressMap(Map addressMap) {

this.addressMap = addressMap;

}

// prints and returns all the elements of the Map.

public Map getAddressMap() {

System.out.println("Map Elements :" + addressMap);

return addressMap;

}

// a setter method to set Property

public void setAddressProp(Properties addressProp) {

this.addressProp = addressProp;

}

// prints and returns all the elements of the Property.

public Properties getAddressProp() {

System.out.println("Property Elements :" + addressProp);

return addressProp;

}

}

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");

JavaCollection jc=(JavaCollection)context.getBean("javaCollection");

jc.getAddressList();

jc.getAddressSet();

jc.getAddressMap();

jc.getAddressProp();

}

}

Following is the configuration file **Beans.xml** which has configuration for all the type of collections −

<?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">

<!-- Definition for javaCollection -->

<bean id = "javaCollection" class = "com.tutorialspoint.JavaCollection">

<!-- results in a setAddressList(java.util.List) call -->

<property name = "addressList">

<list>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</list>

</property>

<!-- results in a setAddressSet(java.util.Set) call -->

<property name = "addressSet">

<set>

<value>INDIA</value>

<value>Pakistan</value>

<value>USA</value>

<value>USA</value>

</set>

</property>

<!-- results in a setAddressMap(java.util.Map) call -->

<property name = "addressMap">

<map>

<entry key = "1" value = "INDIA"/>

<entry key = "2" value = "Pakistan"/>

<entry key = "3" value = "USA"/>

<entry key = "4" value = "USA"/>

</map>

</property>

<!-- results in a setAddressProp(java.util.Properties) call -->

<property name = "addressProp">

<props>

<prop key = "one">INDIA</prop>

<prop key = "one">INDIA</prop>

<prop key = "two">Pakistan</prop>

<prop key = "three">USA</prop>

<prop key = "four">USA</prop>

</props>

</property>

</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 −

List Elements :[INDIA, Pakistan, USA, USA]

Set Elements :[INDIA, Pakistan, USA]

ap Elements :{1 = INDIA, 2 = Pakistan, 3 = USA, 4 = USA}

Property Elements :{two = Pakistan, one = INDIA, three = USA, four = USA}

## **Injecting Bean References**

The following Bean definition will help you understand how to inject bean references as one of the collection's element. Even you can mix references and values all together as shown in the following code snippet −

<?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 Definition to handle references and values -->

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

<!-- Passing bean reference for java.util.List -->

<property name = "addressList">

<list>

<ref bean = "address1"/>

<ref bean = "address2"/>

<value>Pakistan</value>

</list>

</property>

<!-- Passing bean reference for java.util.Set -->

<property name = "addressSet">

<set>

<ref bean = "address1"/>

<ref bean = "address2"/>

<value>Pakistan</value>

</set>

</property>

<!-- Passing bean reference for java.util.Map -->

<property name = "addressMap">

<map>

<entry key = "one" value = "INDIA"/>

<entry key = "two" value-ref = "address1"/>

<entry key = "three" value-ref = "address2"/>

</map>

</property>

</bean>

</beans>

To use the above bean definition, you need to define your setter methods in such a way that they should be able to handle references as well.

## **Injecting null and empty string values**

If you need to pass an empty string as a value, then you can pass it as follows −

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

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

</bean>

The preceding example is equivalent to the Java code: exampleBean.setEmail("")

If you need to pass a NULL value, then you can pass it as follows −

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

<property name = "email"><null/></property>

</bean>

The preceding example is equivalent to the Java code: exampleBean.setEmail(null)

Bean Auto-wiring :

The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

## **Autowiring Modes**

Following are the autowiring modes, which can be used to instruct the Spring container to use autowiring for dependency injection. You use the autowire attribute of the <bean/> element to specify **autowire** mode for a bean definition.

|  |  |
| --- | --- |
| **Sr.No** | **Mode & Description** |
| 1 | **no**  This is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in Dependency Injection chapter. |
| 2 | [**byName**](https://www.tutorialspoint.com/spring/spring_autowiring_byname.htm)  Autowiring by property name. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. |
| 3 | [**byType**](https://www.tutorialspoint.com/spring/spring_autowiring_bytype.htm)  Autowiring by property datatype. Spring container looks at the properties of the beans on which *autowire* attribute is set to *byType*in the XML configuration file. It then tries to match and wire a property if its **type** matches with exactly one of the beans name in configuration file. If more than one such beans exists, a fatal exception is thrown. |
| 4 | [**constructor**](https://www.tutorialspoint.com/spring/spring_autowiring_byconstructor.htm)  Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| 5 | **autodetect**  Spring first tries to wire using autowire by *constructor*, if it does not work, Spring tries to autowire by *byType*. |

You can use **byType** or **constructor** autowiring mode to wire arrays and other typed-collections.

byName :

This mode specifies autowiring by property name.

Spring container looks at the beans on which *auto-wire* attribute is set to *byName* in the XML configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file. If matches are found, it will inject those beans. Otherwise, bean(s) will not be wired.

Example-

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker() {

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** in normal condition

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

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

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

But if you are going to use autowiring 'byName', then your XML configuration file will become as follows −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor" autowire = "byName">

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></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 −

Inside SpellChecker constructor.

Inside checkSpelling.

byType :

This mode specifies autowiring by property type.

Spring container looks at the beans on which *autowire* attribute is set to *byType* in the XML configuration file. It then tries to match and wire a property if its **type** matches with exactly one of the beans name in the configuration file. If matches are found, it will inject those beans. Otherwise, bean(s) will not be wired.

Example :

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public void setSpellChecker( SpellChecker spellChecker ) {

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java** −

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** in normal condition −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

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

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

But if you are going to use autowiring 'byType', then your XML configuration file will become as follows −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor" autowire = "byType">

<property name = "name" value = "Generic Text Editor" />

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "SpellChecker" class = "com.tutorialspoint.SpellChecker"></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 −

Inside SpellChecker constructor.

Inside checkSpelling.

byConstructor :-

This mode is very similar to *byType*, but it applies to constructor arguments.

Spring container looks at the beans on which *autowire* attribute is set *constructor* in the XML configuration file. It then tries to match and wire its constructor's argument with exactly one of the beans name in the configuration file. If matches are found, it will inject those beans. Otherwise, bean(s) will not be wired.

**TextEditor.java** file −

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

private String name;

public TextEditor( SpellChecker spellChecker, String name ) {

this.spellChecker = spellChecker;

this.name = name;

}

public SpellChecker getSpellChecker() {

return spellChecker;

}

public String getName() {

return name;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java** −

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** in normal condition −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

<constructor-arg ref = "spellChecker" />

<constructor-arg value = "Generic Text Editor"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker"></bean>

</beans>

But if you are going to use autowiring 'by constructor', then your XML configuration file will become as follows −

<?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">

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor"

autowire = "constructor">

<constructor-arg value = "Generic Text Editor"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "SpellChecker" class = "com.tutorialspoint.SpellChecker"></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 −

Inside SpellChecker constructor.

Inside checkSpelling.

## **Limitations with autowiring**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing for developers to use it to wire only one or two bean definitions. Though, autowiring can significantly reduce the need to specify properties or constructor arguments but you should consider the limitations and disadvantages of autowiring before using them.

|  |  |
| --- | --- |
| **Sr.No.** | **Limitations & Description** |
| 1 | **Overriding possibility**  You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring. |
| 2 | **Primitive data types**  You cannot autowire so-called simple properties such as primitives, Strings, and Classes. |
| 3 | **Confusing nature**  Autowiring is less exact than explicit wiring, so if possible prefer using explict wiring. |

Annotation Based Configuration:

Starting from Spring 2.5 it became possible to configure the dependency injection using **annotations**.

So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

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

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider the following configuration file in case you want to use any annotation in your Spring application.

<?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-3.0.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- bean definitions go here -->

</beans>

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us look at a few important annotations to understand how they work –

|  |  |
| --- | --- |
| **Sr.No.** | **Annotation & Description** |
| 1 | [**@Required**](https://www.tutorialspoint.com/spring/spring_required_annotation.htm)  The @Required annotation applies to bean property setter methods. |
| 2 | [**@Autowired**](https://www.tutorialspoint.com/spring/spring_autowired_annotation.htm)  The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties. |
| 3 | [**@Qualifier**](https://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm)  The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired. |
| 4 | [**JSR-250 Annotations**](https://www.tutorialspoint.com/spring/spring_jsr250_annotations.htm)  Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations. |

[**@Required**](https://www.tutorialspoint.com/spring/spring_required_annotation.htm)

The **@Required** annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time.

Otherwise, the container throws a BeanInitializationException exception.

Example :

**Student.java** file −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Required;

public class Student {

private Integer age;

private String name;

@Required

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Required

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

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");

Student student = (Student) context.getBean("student");

System.out.println("Name : " + student.getName() );

System.out.println("Age : " + student.getAge() );

}

}

Following is the content of the configuration file **Beans.xml** −

<?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-3.0.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

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

<!-- try without passing age and check the result -->

<!-- property name = "age" value = "11"-->

</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 raise *BeanInitializationException* exception and print the following error along with other log messages −

Property 'age' is required for bean 'student'

Next, you can try the above example after removing the comment from 'age' property as follows −

<?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-3.0.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

</beans>

The above example will produce the following result −

Name : Zara

Age : 11

[**@Autowired**](https://www.tutorialspoint.com/spring/spring_autowired_annotation.htm)

The **@Autowired** annotation provides more fine-grained control over where and how autowiring should be accomplished.

Example :

**TextEditor.java** file −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker( ) {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** −

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

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

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

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

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

Inside SpellChecker constructor.

Inside checkSpelling.

## **@Autowired on Properties**

You can use **@Autowired** annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references. So with the usage of @Autowired on properties your **TextEditor.java** file will become as follows −

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

@Autowired

private SpellChecker spellChecker;

public TextEditor() {

System.out.println("Inside TextEditor constructor." );

}

public SpellChecker getSpellChecker( ){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the configuration file **Beans.xml** −

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

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

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

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message −

Inside TextEditor constructor.

Inside SpellChecker constructor.

Inside checkSpelling.

## **@Autowired on Constructors**

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file. Let us check the following example.

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

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class TextEditor {

private SpellChecker spellChecker;

@Autowired

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the configuration file **Beans.xml** −

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

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

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

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for textEditor bean without constructor-arg -->

<bean id = "textEditor" class = "com.tutorialspoint.TextEditor">

</bean>

<!-- Definition for spellChecker bean -->

<bean id = "spellChecker" class = "com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **@Autowired with (required = false) option**

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, however, you can turn off the default behavior by using **(required=false)** option with @Autowired.

The following example will work even if you do not pass any value for age property but still it will demand for name property. You can try this example yourself because this is similar to @Required annotation example except that only **Student.java** file has been changed.

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

public class Student {

private Integer age;

private String name;

@Autowired(required=false)

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

@Autowired

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

[**@Qualifier**](https://www.tutorialspoint.com/spring/spring_qualifier_annotation.htm)

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property.

In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

Example :

**Student.java** file −

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

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

package com.tutorialspoint;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

public class Profile {

@Autowired

@Qualifier("student1")

private Student student;

public Profile(){

System.out.println("Inside Profile constructor." );

}

public void printAge() {

System.out.println("Age : " + student.getAge() );

}

public void printName() {

System.out.println("Name : " + student.getName() );

}

}

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");

Profile profile = (Profile) context.getBean("profile");

profile.printAge();

profile.printName();

}

}

Consider the example of following configuration file **Beans.xml**

<?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-3.0.xsd

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<!-- Definition for profile bean -->

<bean id = "profile" class = "com.tutorialspoint.Profile"></bean>

<!-- Definition for student1 bean -->

<bean id = "student1" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

<!-- Definition for student2 bean -->

<bean id = "student2" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "2"/>

</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 −

Inside Profile constructor.

Age : 11

Name : Zara

**JSR-250 Annotations**

Spring also supports JSR-250 based annotations which include @PostConstruct, @PreDestroy and @Resource annotations. Though these annotations are not really required because you already have other alternates, yet let us get a brief idea about them.

## **@PostConstruct and @PreDestroy Annotations**

To define the 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, the destroy-method specifies a method that is called just before a bean is removed from the container.

You can use **@PostConstruct** annotation as an alternate of initialization callback and **@PreDestroy** annotation as an alternate of destruction callback.

**Example :-**

**HelloWorld.java** file −

package com.tutorialspoint;

import javax.annotation.\*;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public String getMessage(){

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

return message;

}

@PostConstruct

public void init(){

System.out.println("Bean is going through init.");

}

@PreDestroy

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"

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

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

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

http://www.springframework.org/schema/context

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

<context:annotation-config/>

<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.

## **@Resource Annotation**

You can use **@Resource** annotation on fields or setter methods and it works the same as in Java EE 5. The @Resource annotation takes a 'name' attribute which will be interpreted as the bean name to be injected. You can say, it follows **by-name** autowiring semantics as demonstrated in the following example −

package com.tutorialspoint;

import javax.annotation.Resource;

public class TextEditor {

private SpellChecker spellChecker;

@Resource(name = "spellChecker")

public void setSpellChecker( SpellChecker spellChecker ){

this.spellChecker = spellChecker;

}

public SpellChecker getSpellChecker(){

return spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

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.

**Java Based Configuration**

So far you have seen how we configure Spring beans using XML configuration file. If you are comfortable with XML configuration, then it is really not required to learn how to proceed with Java-based configuration as you are going to achieve the same result using either of the configurations available.

## **@Configuration & @Bean Annotations**

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. The **@Bean**annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context. The simplest possible @Configuration class would be as follows −

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

The above code will be equivalent to the following XML configuration −

<beans>

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

</beans>

Here, the method name is annotated with @Bean works as bean ID and it creates and returns the actual bean. Your configuration class can have a declaration for more than one @Bean. Once your configuration classes are defined, you can load and provide them to Spring container using *AnnotationConfigApplicationContext* as follows –

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

You can load various configuration classes as follows −

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

**Example :-**

**HelloWorldConfig.java** file

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class HelloWorldConfig {

@Bean

public HelloWorld helloWorld(){

return new HelloWorld();

}

}

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.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);

helloWorld.setMessage("Hello World!");

helloWorld.getMessage();

}

}

Once you are done creating all the source files and adding the required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, it will print the following message −

Your Message : Hello World!

## **Injecting Bean Dependencies**

When @Beans have dependencies on one another, expressing that the dependency is as simple as having one bean method calling another as follows −

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

return new Foo(bar());

}

@Bean

public Bar bar() {

return new Bar();

}

}

Here, the foo bean receives a reference to bar via the constructor injection. Now let us look at another working example.

**Example :-**

**TextEditorConfig.java** file

package com.tutorialspoint;

import org.springframework.context.annotation.\*;

@Configuration

public class TextEditorConfig {

@Bean

public TextEditor textEditor(){

return new TextEditor( spellChecker() );

}

@Bean

public SpellChecker spellChecker(){

return new SpellChecker( );

}

}

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

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker){

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck(){

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file **SpellChecker.java**

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling(){

System.out.println("Inside checkSpelling." );

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.\*;

public class MainApp {

public static void main(String[] args) {

ApplicationContext ctx =

new AnnotationConfigApplicationContext(TextEditorConfig.class);

TextEditor te = ctx.getBean(TextEditor.class);

te.spellCheck();

}

}

Once you are done creating all the source files and adding the required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, it will print the following message −

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

## **The @Import Annotation**

The **@Import** annotation allows for loading @Bean definitions from another configuration class. Consider a ConfigA class as follows −

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

You can import above Bean declaration in another Bean Declaration as follows −

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied as follows −

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

## **Lifecycle Callbacks**

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 −

public class Foo {

public void init() {

// initialization logic

}

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init", destroyMethod = "cleanup" )

public Foo foo() {

return new Foo();

}

}

## **Specifying Bean Scope**

The default scope is singleton, but you can override this with the @Scope annotation as follows −

@Configuration

public class AppConfig {

@Bean

@Scope("prototype")

public Foo foo() {

return new Foo();

}

}

**Event Handling in Spring:-**

You have seen in all the chapters that the core of Spring is the **ApplicationContext**, which manages the complete life cycle of the beans.

ApplicationContext publishes certain types of events when loading the beans.

For example, a *ContextStartedEvent* is published when the context is started and *ContextStoppedEvent* is published when the context is stopped.

Event handling in the *ApplicationContext* is provided through the *ApplicationEvent* class and *ApplicationListener* interface. Hence, if a bean implements the *ApplicationListener*, then every time an *ApplicationEvent* gets published to the ApplicationContext, that bean is notified.

Spring provides the following standard events −

|  |  |
| --- | --- |
| **Sr.No.** | **Spring Built-in Events & Description** |
| 1 | **ContextRefreshedEvent**  This event is published when the *ApplicationContext* is either initialized or refreshed. This can also be raised using the refresh() method on the *ConfigurableApplicationContext* interface. |
| 2 | **ContextStartedEvent**  This event is published when the *ApplicationContext* is started using the start() method on the *ConfigurableApplicationContext*interface. You can poll your database or you can restart any stopped application after receiving this event. |
| 3 | **ContextStoppedEvent**  This event is published when the *ApplicationContext* is stopped using the stop() method on the *ConfigurableApplicationContext*interface. You can do required housekeep work after receiving this event. |
| 4 | **ContextClosedEvent**  This event is published when the *ApplicationContext* is closed using the close() method on the *ConfigurableApplicationContext*interface. A closed context reaches its end of life; it cannot be refreshed or restarted. |
| 5 | **RequestHandledEvent**  This is a web-specific event telling all beans that an HTTP request has been serviced. |

Spring's event handling is single-threaded so if an event is published, until and unless all the receivers get the message, the processes are blocked and the flow will not continue. Hence, care should be taken when designing your application if the event handling is to be used.

## **Listening to Context Events**

To listen to a context event, a bean should implement the *ApplicationListener*interface which has just one method **onApplicationEvent()**. So let us write an example to see how the events propagates and how you can put your code to do required task based on certain events.

**Example:-**

**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 **CStartEventHandler.java** file

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

import org.springframework.context.event.ContextStartedEvent;

public class CStartEventHandler

implements ApplicationListener<ContextStartedEvent>{

public void onApplicationEvent(ContextStartedEvent event) {

System.out.println("ContextStartedEvent Received");

}

}

Following is the content of the **CStopEventHandler.java** file

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

import org.springframework.context.event.ContextStoppedEvent;

public class CStopEventHandler

implements ApplicationListener<ContextStoppedEvent>{

public void onApplicationEvent(ContextStoppedEvent event) {

System.out.println("ContextStoppedEvent Received");

}

}

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

package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

// Let us raise a start event.

context.start();

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

obj.getMessage();

// Let us raise a stop event.

context.stop();

}

}

Following is the configuration file **Beans.xml**

<?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">

<property name = "message" value = "Hello World!"/>

</bean>

<bean id = "cStartEventHandler" class = "com.tutorialspoint.CStartEventHandler"/>

<bean id = "cStopEventHandler" class = "com.tutorialspoint.CStopEventHandler"/>

</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 −

ContextStartedEvent Received

Your Message : Hello World!

ContextStoppedEvent Received

**Custom Events in Spring:**

There are number of steps to be taken to write and publish your own custom events. Follow the instructions given in this chapter to write, publish and handle Custom Spring Events.

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. All the classes will be created under this package. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create an event class, *CustomEvent* by extending **ApplicationEvent**. This class must define a default constructor which should inherit constructor from ApplicationEvent class. |
| 4 | Once your event class is defined, you can publish it from any class, let us say *EventClassPublisher* which implements *ApplicationEventPublisherAware*. You will also need to declare this class in XML configuration file as a bean so that the container can identify the bean as an event publisher because it implements the ApplicationEventPublisherAware interface. |
| 5 | A published event can be handled in a class, let us say *EventClassHandler*which implements *ApplicationListener* interface and implements *onApplicationEvent* method for the custom event. |
| 6 | Create beans configuration file *Beans.xml* under the **src** folder and a *MainApp* class which will work as Spring application. |
| 7 | 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 **CustomEvent.java** file

package com.tutorialspoint;

import org.springframework.context.ApplicationEvent;

public class CustomEvent extends ApplicationEvent{

public CustomEvent(Object source) {

super(source);

}

public String toString(){

return "My Custom Event";

}

}

Following is the content of the **CustomEventPublisher.java** file

package com.tutorialspoint;

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.context.ApplicationEventPublisherAware;

public class CustomEventPublisher implements ApplicationEventPublisherAware {

private ApplicationEventPublisher publisher;

public void setApplicationEventPublisher (ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

public void publish() {

CustomEvent ce = new CustomEvent(this);

publisher.publishEvent(ce);

}

}

Following is the content of the **CustomEventHandler.java** file

package com.tutorialspoint;

import org.springframework.context.ApplicationListener;

public class CustomEventHandler implements ApplicationListener<CustomEvent> {

public void onApplicationEvent(CustomEvent event) {

System.out.println(event.toString());

}

}

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

package com.tutorialspoint;

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ConfigurableApplicationContext context =

new ClassPathXmlApplicationContext("Beans.xml");

CustomEventPublisher cvp =

(CustomEventPublisher) context.getBean("customEventPublisher");

cvp.publish();

cvp.publish();

}

}

Following is the configuration file **Beans.xml**

<?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 = "customEventHandler" class = "com.tutorialspoint.CustomEventHandler"/>

<bean id = "customEventPublisher" class = "com.tutorialspoint.CustomEventPublisher"/>

</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 −

y Custom Event

y Custom Event

**AOP with Spring:-**

**Aspect oriented programming (AOP)**

Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns.

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 like logging, auditing, 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.

Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect. AOP is like triggers in programming languages such as Perl, .NET, Java, and others.

Spring AOP module provides interceptors to intercept an application.

For example, when a method is executed, you can add extra functionality before or after the method execution.

## **AOP Terminologies**

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

|  |  |
| --- | --- |
| **Sr.No** | **Terms & Description** |
| 1 | **Aspect**  This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. |
| 2 | **Join point**  This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. |
| 3 | **Advice**  This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework. |
| 4 | **Pointcut**  This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. |
| 5 | **Introduction**  An introduction allows you to add new methods or attributes to the existing classes. |
| 6 | **Target object**  The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object. |
| 7 | **Weaving**  Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. |

## **Types of Advice**

Spring aspects can work with five kinds of advice mentioned as follows −

|  |  |
| --- | --- |
| **Sr.No** | **Advice & Description** |
| 1 | **before**  Run advice before the a method execution. |
| 2 | **after**  Run advice after the method execution, regardless of its outcome. |
| 3 | **after-returning**  Run advice after the a method execution only if method completes successfully. |
| 4 | **after-throwing**  Run advice after the a method execution only if method exits by throwing an exception. |
| 5 | **around**  Run advice before and after the advised method is invoked. |

## **Custom Aspects Implementation**

Spring supports the **@AspectJ annotation style** approach and the **schema-based** approach to implement custom aspects. These two approaches have been explained in detail in the following sections.

|  |  |
| --- | --- |
| **Sr.No** | **Approach & Description** |
| 1 | [**XML Schema based**](https://www.tutorialspoint.com/spring/schema_based_aop_appoach.htm)  Aspects are implemented using the regular classes along with XML based configuration. |
| 2 | [**@AspectJ based**](https://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm)  @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. |

**XML Schema Based :-**

To use the AOP namespace tags described in this section, you need to import the springAOP schema as described −

<?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-3.0.xsd

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

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

<!-- bean definition & AOP specific configuration -->

</beans>

You will also need the following AspectJ libraries on the CLASSPATH of your application. These libraries are available in the 'lib' directory of an AspectJ installation, otherwise you can download them from the internet.

* aspectjrt.jar
* aspectjweaver.jar
* aspectj.jar
* aopalliance.jar

## **Declaring an aspect**

An **aspect** is declared using the **<aop:aspect>** element, and the backing bean is referenced using the **ref** attribute as follows −

<aop:config>

<aop:aspect id = "myAspect" ref = "aBean">

...

</aop:aspect>

</aop:config>

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

...

</bean>

Here "aBean" will be configured and dependency injected just like any other Spring bean as you have seen in the previous chapters.

## **Declaring a pointcut**

A **pointcut** helps in determining the join points (ie methods) of interest to be executed with different advices. While working with XML Schema-based configuration, pointcut will be defined as follows −

<aop:config>

<aop:aspect id = "myAspect" ref = "aBean">

<aop:pointcut id = "businessService"

expression = "execution(\*com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

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

...

</bean>

The following example defines a pointcut named 'businessService' that will match the execution of getName() method available in the Student class under the package com.tutorialspoint −

<aop:config>

<aop:aspect id = "myAspect" ref = "aBean">

<aop:pointcut id = "businessService"

expression = "execution(\*com.tutorialspoint.Student.getName(..))"/>

...

</aop:aspect>

</aop:config>

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

...

</bean>

## **Declaring advices**

You can declare any of the five advices inside an <aop:aspect> using the <aop:{ADVICE NAME}> element as given below −

<aop:config>

<aop:aspect id = "myAspect" ref = "aBean">

<aop:pointcut id = "businessService"

expression = "execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<!-- a before advice definition -->

<aop:before pointcut-ref = "businessService" method = "doRequiredTask"/>

<!-- an after advice definition -->

<aop:after pointcut-ref = "businessService" method = "doRequiredTask"/>

<!-- an after-returning advice definition -->

<!--The doRequiredTask method must have parameter named retVal -->

<aop:after-returning pointcut-ref = "businessService"

returning = "retVal" method = "doRequiredTask"/>

<!-- an after-throwing advice definition -->

<!--The doRequiredTask method must have parameter named ex -->

<aop:after-throwing pointcut-ref = "businessService"

throwing = "ex" method = "doRequiredTask"/>

<!-- an around advice definition -->

<aop:around pointcut-ref = "businessService" method = "doRequiredTask"/>

...

</aop:aspect>

</aop:config>

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

...

</bean>

You can use same **doRequiredTask** or different methods for different advices. These methods will be defined as a part of aspect module.

**Example :-**

Here is the content of **Logging.java** file. This is actually a sample of aspect module which defines the methods to be called at various points.

package com.tutorialspoint;

public class Logging {

/\*\*

\* This is the method which I would like to execute

\* before a selected method execution.

\*/

public void beforeAdvice(){

System.out.println("Going to setup student profile.");

}

/\*\*

\* This is the method which I would like to execute

\* after a selected method execution.

\*/

public void afterAdvice(){

System.out.println("Student profile has been setup.");

}

/\*\*

\* This is the method which I would like to execute

\* when any method returns.

\*/

public void afterReturningAdvice(Object retVal) {

System.out.println("Returning:" + retVal.toString() );

}

/\*\*

\* This is the method which I would like to execute

\* if there is an exception raised.

\*/

public void AfterThrowingAdvice(IllegalArgumentException ex){

System.out.println("There has been an exception: " + ex.toString());

}

}

Following is the content of the **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

System.out.println("Age : " + age );

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

System.out.println("Name : " + name );

return name;

}

public void printThrowException(){

System.out.println("Exception raised");

throw new IllegalArgumentException();

}

}

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");

Student student = (Student) context.getBean("student");

student.getName();

student.getAge();

student.printThrowException();

}

}

Following is the configuration file **Beans.xml**

<?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-3.0.xsd

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

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

<aop:config>

<aop:aspect id = "log" ref = "logging">

<aop:pointcut id = "selectAll"

expression = "execution(\* com.tutorialspoint.\*.\*(..))"/>

<aop:before pointcut-ref = "selectAll" method = "beforeAdvice"/>

<aop:after pointcut-ref = "selectAll" method = "afterAdvice"/>

<aop:after-returning pointcut-ref = "selectAll"

returning = "retVal" method = "afterReturningAdvice"/>

<aop:after-throwing pointcut-ref = "selectAll"

throwing = "ex" method = "AfterThrowingAdvice"/>

</aop:aspect>

</aop:config>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id = "logging" class = "com.tutorialspoint.Logging"/>

</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 −

Going to setup student profile.

Name : Zara

Student profile has been setup.

Returning:Zara

Going to setup student profile.

Age : 11

Student profile has been setup.

Returning:11

Going to setup student profile.

Exception raised

Student profile has been setup.

There has been an exception: java.lang.IllegalArgumentException

.....

other exception content

The above defined <aop:pointcut> selects all the methods defined under the package com.tutorialspoint. Let us suppose, you want to execute your advice before or after a particular method, you can define your pointcut to narrow down your execution by replacing stars (\*) in pointcut definition with the actual class and method names. Following is a modified XML configuration file to show the concept –

<?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-3.0.xsd

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

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

<aop:config>

<aop:aspect id = "log" ref = "logging">

<aop:pointcut id = "selectAll"

expression = "execution(\* com.tutorialspoint.Student.getName(..))"/>

<aop:before pointcut-ref = "selectAll" method = "beforeAdvice"/>

<aop:after pointcut-ref = "selectAll" method = "afterAdvice"/>

</aop:aspect>

</aop:config>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id = "logging" class = "com.tutorialspoint.Logging"/>

</beans>

If you execute the sample application with these configuration changes, it will print the following message −

Going to setup student profile.

Name : Zara

Student profile has been setup.

Age : 11

Exception raised

.....

other exception content

**@AspectJ Based AOP with Spring:**

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ support is enabled by including the following element inside your XML Schema-based configuration file.

<aop:aspectj-autoproxy/>

You will also need the following AspectJ libraries on the classpath of your application. These libraries are available in the 'lib' directory of an AspectJ installation, otherwise you can download them from the internet.

* aspectjrt.jar
* aspectjweaver.jar
* aspectj.jar
* aopalliance.jar

## **Declaring an aspect**

Aspects classes are like any other normal bean and may have methods and fields just like any other class, except that they will be annotated with @Aspect as follows −

package org.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class AspectModule {

}

They will be configured in XML like any other bean as follows −

<bean id = "myAspect" class = "org.xyz.AspectModule">

<!-- configure properties of aspect here as normal -->

</bean>

## **Declaring a pointcut**

A **pointcut** helps in determining the join points (ie methods) of interest to be executed with different advices. While working with @AspectJ-based configuration, pointcut declaration has two parts −

* A pointcut expression that determines exactly which method executions we are interested in.
* A pointcut signature comprising a name and any number of parameters. The actual body of the method is irrelevant and in fact should be empty.

The following example defines a pointcut named 'businessService' that will match the execution of every method available in the classes under the package com.xyz.myapp.service −

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.xyz.myapp.service.\*.\*(..))") // expression

private void businessService() {} // signature

The following example defines a pointcut named 'getname' that will match the execution of getName() method available in the Student class under the package com.tutorialspoint −

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.tutorialspoint.Student.getName(..))")

private void getname() {}

## **Declaring advices**

You can declare any of the five advices using @{ADVICE-NAME} annotations as given in the code snippet. This assumes that you already have defined a pointcut signature method businessService() −

@Before("businessService()")

public void doBeforeTask(){

...

}

@After("businessService()")

public void doAfterTask(){

...

}

@AfterReturning(pointcut = "businessService()", returning = "retVal")

public void doAfterReturnningTask(Object retVal) {

// you can intercept retVal here.

...

}

@AfterThrowing(pointcut = "businessService()", throwing = "ex")

public void doAfterThrowingTask(Exception ex) {

// you can intercept thrown exception here.

...

}

@Around("businessService()")

public void doAroundTask(){

...

}

You can define a pointcut inline for any of the advices. Following is an example to define inline pointcut for before advice −

@Before("execution(\* com.xyz.myapp.service.\*.\*(..))")

public doBeforeTask(){

...

}

**Example :-**

Here is the content of **Logging.java** file. This is actually a sample of aspect module which defines methods to be called at various points.

package com.tutorialspoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.After;

import org.aspectj.lang.annotation.AfterThrowing;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Around;

@Aspect

public class Logging {

/\*\* Following is the definition for a pointcut to select

\* all the methods available. So advice will be called

\* for all the methods.

\*/

@Pointcut("execution(\* com.tutorialspoint.\*.\*(..))")

private void selectAll(){}

/\*\*

\* This is the method which I would like to execute

\* before a selected method execution.

\*/

@Before("selectAll()")

public void beforeAdvice(){

System.out.println("Going to setup student profile.");

}

/\*\*

\* This is the method which I would like to execute

\* after a selected method execution.

\*/

@After("selectAll()")

public void afterAdvice(){

System.out.println("Student profile has been setup.");

}

/\*\*

\* This is the method which I would like to execute

\* when any method returns.

\*/

@AfterReturning(pointcut = "selectAll()", returning = "retVal")

public void afterReturningAdvice(Object retVal){

System.out.println("Returning:" + retVal.toString() );

}

/\*\*

\* This is the method which I would like to execute

\* if there is an exception raised by any method.

\*/

@AfterThrowing(pointcut = "selectAll()", throwing = "ex")

public void AfterThrowingAdvice(IllegalArgumentException ex){

System.out.println("There has been an exception: " + ex.toString());

}

}

Following is the content of the **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

System.out.println("Age : " + age );

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

System.out.println("Name : " + name );

return name;

}

public void printThrowException(){

System.out.println("Exception raised");

throw new IllegalArgumentException();

}

}

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");

Student student = (Student) context.getBean("student");

student.getName();

student.getAge();

student.printThrowException();

}

}

Following is the configuration file **Beans.xml**

<?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-3.0.xsd

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

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

<aop:aspectj-autoproxy/>

<!-- Definition for student bean -->

<bean id = "student" class = "com.tutorialspoint.Student">

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

<property name = "age" value = "11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id = "logging" class = "com.tutorialspoint.Logging"/>

</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 −

Going to setup student profile.

Name : Zara

Student profile has been setup.

Returning:Zara

Going to setup student profile.

Age : 11

Student profile has been setup.

Returning:11

Going to setup student profile.

Exception raised

Student profile has been setup.

There has been an exception: java.lang.IllegalArgumentException

.....

other exception content

**Spring JDBC :-**

While working with the database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections, etc. However, Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

So what you have to do is just define the connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. I'm going to take classic and the most popular approach which makes use of **JdbcTemplate** class of the framework. This is the central framework class that manages all the database communication and exception handling.

## **JdbcTemplate Class**

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the *JdbcTemplate* class are *threadsafe* once configured. So you can configure a single instance of a *JdbcTemplate* and then safely inject this shared reference into multiple DAOs.

A common practice when using the JDBC Template class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

## **Configuring Data Source**

Let us create a database table **Student** in our database **TEST**. We assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown in the following code snippet −

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

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

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

</bean>

## **Data Access Object (DAO)**

DAO stands for Data Access Object, which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

## **Executing SQL statements**

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

**Querying for an integer**

String SQL = "select count(\*) from Student";

int rowCount = jdbcTemplateObject.queryForInt( SQL );

**Querying for a long**

String SQL = "select count(\*) from Student";

long rowCount = jdbcTemplateObject.queryForLong( SQL );

**A simple query using a bind variable**

String SQL = "select age from Student where id = ?";

int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});

**Querying for a String**

String SQL = "select name from Student where id = ?";

String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);

**Querying and returning an object**

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(

SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

**Querying and returning multiple objects**

String SQL = "select \* from Student";

List<Student> students = jdbcTemplateObject.query(

SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

**Inserting a row into the table**

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );

**Updating a row into the table**

String SQL = "update Student set name = ? where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );

**Deleting a row from the table**

String SQL = "delete Student where id = ?";

jdbcTemplateObject.update( SQL, new Object[]{20} );

## **Executing DDL Statements**

You can use the **execute(..)** method from *jdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table −

String SQL = "CREATE TABLE Student( " +

"ID INT NOT NULL AUTO\_INCREMENT, " +

"NAME VARCHAR(20) NOT NULL, " +

"AGE INT NOT NULL, " +

"PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );

[**Spring JDBC Example**](https://www.tutorialspoint.com/spring/spring_jdbc_example.htm)

Following is the content of the Data Access Object interface file **StudentDAO.java** −

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to create

\* a record in the Student table.

\*/

public void create(String name, Integer age);

/\*\*

\* This is the method to be used to list down

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public Student getStudent(Integer id);

/\*\*

\* This is the method to be used to list down

\* all the records from the Student table.

\*/

public List<Student> listStudents();

/\*\*

\* This is the method to be used to delete

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public void delete(Integer id);

/\*\*

\* This is the method to be used to update

\* a record into the Student table.

\*/

public void update(Integer id, Integer age);

}

Following is the content of the **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of the **StudentMapper.java** file

package com.tutorialspoint;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO.

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplateObject;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplateObject = new JdbcTemplate(dataSource);

}

public void create(String name, Integer age) {

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, name, age);

System.out.println("Created Record Name = " + name + " Age = " + age);

return;

}

public Student getStudent(Integer id) {

String SQL = "select \* from Student where id = ?";

Student student = jdbcTemplateObject.queryForObject(SQL,

new Object[]{id}, new StudentMapper());

return student;

}

public List<Student> listStudents() {

String SQL = "select \* from Student";

List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());

return students;

}

public void delete(Integer id) {

String SQL = "delete from Student where id = ?";

jdbcTemplateObject.update(SQL, id);

System.out.println("Deleted Record with ID = " + id );

return;

}

public void update(Integer id, Integer age){

String SQL = "update Student set age = ? where id = ?";

jdbcTemplateObject.update(SQL, age, id);

System.out.println("Updated Record with ID = " + id );

return;

}

}

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

package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

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

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

System.out.println("------Records Creation--------" );

studentJDBCTemplate.create("Zara", 11);

studentJDBCTemplate.create("Nuha", 2);

studentJDBCTemplate.create("Ayan", 15);

System.out.println("------Listing Multiple Records--------" );

List<Student> students = studentJDBCTemplate.listStudents();

for (Student record : students) {

System.out.print("ID : " + record.getId() );

System.out.print(", Name : " + record.getName() );

System.out.println(", Age : " + record.getAge());

}

System.out.println("----Updating Record with ID = 2 -----" );

studentJDBCTemplate.update(2, 20);

System.out.println("----Listing Record with ID = 2 -----" );

Student student = studentJDBCTemplate.getStudent(2);

System.out.print("ID : " + student.getId() );

System.out.print(", Name : " + student.getName() );

System.out.println(", Age : " + student.getAge());

}

}

Following is the configuration file **Beans.xml**

<?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 ">

<!-- Initialization for data source -->

<bean id="dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

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

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

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBCTemplate"

class = "com.tutorialspoint.StudentJDBCTemplate">

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

</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 −

------Records Creation--------

Created Record Name = Zara Age = 11

Created Record Name = Nuha Age = 2

Created Record Name = Ayan Age = 15

------Listing Multiple Records--------

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

----Updating Record with ID = 2 -----

Updated Record with ID = 2

----Listing Record with ID = 2 -----

ID : 2, Name : Nuha, Age : 20

You can try and delete the operation yourself, which we have not used in the example, but now you have one working application based on Spring JDBC framework, which you can extend to add sophisticated functionality based on your project requirements. There are other approaches to access the database where you will use **NamedParameterJdbcTemplate** and **SimpleJdbcTemplate** classes, so if you are interested in learning these classes then kindly check the reference manual for Spring Framework.

[**SQL Stored Procedure in Spring**](https://www.tutorialspoint.com/spring/calling_stored_procedure.htm)

The **SimpleJdbcCall** class can be used to call a stored procedure with IN and OUT parameters. You can use this approach while working with either of the RDBMS like Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.

To understand the approach, let us take our Student table which can be created in MySQL TEST database with the following DDL −

CREATE TABLE Student(

ID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(20) NOT NULL,

AGE INT NOT NULL,

PRIMARY KEY (ID)

);

Next, consider the following MySQL stored procedure, which takes student Id and returns the corresponding student's name and age using OUT parameters. So let us create this stored procedure in your TEST database using MySQL command prompt −

DELIMITER $$

DROP PROCEDURE IF EXISTS `TEST`.`getRecord` $$

CREATE PROCEDURE `TEST`.`getRecord` (

IN in\_id INTEGER,

OUT out\_name VARCHAR(20),

OUT out\_age INTEGER)

BEGIN

SELECT name, age

INTO out\_name, out\_age

FROM Student where id = in\_id;

END $$

DELIMITER ;

Now let us write our Spring JDBC application which will implement a simple Create and Read operations on our Student table.

**Example :-**

Following is the content of the Data Access Object interface file **StudentDAO.java**

package com.tutorialspoint;

import java.util.List;

import javax.sql.DataSource;

public interface StudentDAO {

/\*\*

\* This is the method to be used to initialize

\* database resources ie. connection.

\*/

public void setDataSource(DataSource ds);

/\*\*

\* This is the method to be used to create

\* a record in the Student table.

\*/

public void create(String name, Integer age);

/\*\*

\* This is the method to be used to list down

\* a record from the Student table corresponding

\* to a passed student id.

\*/

public Student getStudent(Integer id);

/\*\*

\* This is the method to be used to list down

\* all the records from the Student table.

\*/

public List<Student> listStudents();

}

Following is the content of the **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of the **StudentMapper.java** file

package com.tutorialspoint;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student();

student.setId(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

Following is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface StudentDAO −

package com.tutorialspoint;

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;

import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import org.springframework.jdbc.core.simple.SimpleJdbcCall;

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private SimpleJdbcCall jdbcCall;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcCall = new SimpleJdbcCall(dataSource).withProcedureName("getRecord");

}

public void create(String name, Integer age) {

JdbcTemplate jdbcTemplateObject = new JdbcTemplate(dataSource);

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplateObject.update( SQL, name, age);

System.out.println("Created Record Name = " + name + " Age = " + age);

return;

}

public Student getStudent(Integer id) {

SqlParameterSource in = new MapSqlParameterSource().addValue("in\_id", id);

Map<String, Object> out = jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out\_name"));

student.setAge((Integer) out.get("out\_age"));

return student;

}

public List<Student> listStudents() {

String SQL = "select \* from Student";

List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());

return students;

}

}

Few words about the above program: The code you write for the execution of the call involves creating an *SqlParameterSource* containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The *execute* method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure. Now let us move with the main application file **MainApp.java**, which is as follows −

package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {

public static void main(String[] args) {

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

StudentJDBCTemplate studentJDBCTemplate =

(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

System.out.println("------Records Creation--------" );

studentJDBCTemplate.create("Zara", 11);

studentJDBCTemplate.create("Nuha", 2);

studentJDBCTemplate.create("Ayan", 15);

System.out.println("------Listing Multiple Records--------" );

List<Student> students = studentJDBCTemplate.listStudents();

for (Student record : students) {

System.out.print("ID : " + record.getId() );

System.out.print(", Name : " + record.getName() );

System.out.println(", Age : " + record.getAge());

}

System.out.println("----Listing Record with ID = 2 -----" );

Student student = studentJDBCTemplate.getStudent(2);

System.out.print("ID : " + student.getId() );

System.out.print(", Name : " + student.getName() );

System.out.println(", Age : " + student.getAge());

}

}

Following is the configuration file **Beans.xml**

<?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 ">

<!-- Initialization for data source -->

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

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

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

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBCTemplate"

class = "com.tutorialspoint.StudentJDBCTemplate">

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

</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 −

------Records Creation--------

Created Record Name = Zara Age = 11

Created Record Name = Nuha Age = 2

Created Record Name = Ayan Age = 15

------Listing Multiple Records--------

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

----Listing Record with ID = 2 -----

ID : 2, Name : Nuha, Age : 2

# Spring - Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as **ACID** −

* **Atomicity** − A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
* **Consistency** − This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
* **Isolation** − There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
* **Durability** − Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows −

* Begin the transaction using *begin transaction* command.
* Perform various deleted, update or insert operations using SQL queries.
* If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

## **Local vs. Global Transactions**

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case, transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

## **Programmatic vs. Declarative**

Spring supports two types of transaction management −

* [Programmatic transaction management](https://www.tutorialspoint.com/spring/programmatic_management.htm) − This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
* [Declarative transaction management](https://www.tutorialspoint.com/spring/declarative_management.htm) − This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

## **Spring Transaction Abstractions**

The key to the Spring transaction abstraction is defined by the *org.springframework.transaction.PlatformTransactionManager* interface, which is as follows −

public interface PlatformTransactionManager {

TransactionStatus getTransaction(TransactionDefinition definition);

throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

|  |  |
| --- | --- |
| **Sr.No** | **Method & Description** |
| 1 | **TransactionStatus getTransaction(TransactionDefinition definition)**  This method returns a currently active transaction or creates a new one, according to the specified propagation behavior. |
| 2 | **void commit(TransactionStatus status)**  This method commits the given transaction, with regard to its status. |
| 3 | **void rollback(TransactionStatus status)**  This method performs a rollback of the given transaction. |

The *TransactionDefinition* is the core interface of the transaction support in Spring and it is defined as follows −

public interface TransactionDefinition {

int getPropagationBehavior();

int getIsolationLevel();

String getName();

int getTimeout();

boolean isReadOnly();

}

|  |  |
| --- | --- |
| **Sr.No** | **Method & Description** |
| 1 | **int getPropagationBehavior()**  This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT. |
| 2 | **int getIsolationLevel()**  This method returns the degree to which this transaction is isolated from the work of other transactions. |
| 3 | **String getName()**  This method returns the name of this transaction. |
| 4 | **int getTimeout()**  This method returns the time in seconds in which the transaction must complete. |
| 5 | **boolean isReadOnly()**  This method returns whether the transaction is read-only. |

Following are the possible values for isolation level −

|  |  |
| --- | --- |
| **Sr.No** | **Isolation & Description** |
| 1 | **TransactionDefinition.ISOLATION\_DEFAULT**  This is the default isolation level. |
| 2 | **TransactionDefinition.ISOLATION\_READ\_COMMITTED**  Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur. |
| 3 | **TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED**  Indicates that dirty reads, non-repeatable reads, and phantom reads can occur. |
| 4 | **TransactionDefinition.ISOLATION\_REPEATABLE\_READ**  Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur. |
| 5 | **TransactionDefinition.ISOLATION\_SERIALIZABLE**  Indicates that dirty reads, non-repeatable reads, and phantom reads are prevented. |

Following are the possible values for propagation types −

|  |  |
| --- | --- |
| **Sr.No.** | **Propagation & Description** |
| 1 | **TransactionDefinition.PROPAGATION\_MANDATORY**  Supports a current transaction; throws an exception if no current transaction exists. |
| 2 | **TransactionDefinition.PROPAGATION\_NESTED**  Executes within a nested transaction if a current transaction exists. |
| 3 | **TransactionDefinition.PROPAGATION\_NEVER**  Does not support a current transaction; throws an exception if a current transaction exists. |
| 4 | **TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED**  Does not support a current transaction; rather always execute nontransactionally. |
| 5 | **TransactionDefinition.PROPAGATION\_REQUIRED**  Supports a current transaction; creates a new one if none exists. |
| 6 | **TransactionDefinition.PROPAGATION\_REQUIRES\_NEW**  Creates a new transaction, suspending the current transaction if one exists. |
| 7 | **TransactionDefinition.PROPAGATION\_SUPPORTS**  Supports a current transaction; executes non-transactionally if none exists. |
| 8 | **TransactionDefinition.TIMEOUT\_DEFAULT**  Uses the default timeout of the underlying transaction system, or none if timeouts are not supported. |

The *TransactionStatus* interface provides a simple way for transactional code to control transaction execution and query transaction status.

public interface TransactionStatus extends SavepointManager {

boolean isNewTransaction();

boolean hasSavepoint();

void setRollbackOnly();

boolean isRollbackOnly();

boolean isCompleted();

}

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean hasSavepoint()**  This method returns whether this transaction internally carries a savepoint, i.e., has been created as nested transaction based on a savepoint. |
| 2 | **boolean isCompleted()**  This method returns whether this transaction is completed, i.e., whether it has already been committed or rolled back. |
| 3 | **boolean isNewTransaction()**  This method returns true in case the present transaction is new. |
| 4 | **boolean isRollbackOnly()**  This method returns whether the transaction has been marked as rollback-only. |
| 5 | **void setRollbackOnly()**  This method sets the transaction as rollback-only. |

# Spring - MVC Framework

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

## **The DispatcherServlet**

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram −



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* −

* After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
* The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
* The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
* Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

## **Required Configuration**

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet* example −

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>\*.jsp</url-pattern>

</servlet-mapping>

</web-app>

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INFdirectory. In this case, our file will be **HelloWebservlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as *[servlet-name]-servlet.xml*and default location as *WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener *ContextLoaderListener* in your web.xml file as follows –

<web-app...>

<!-------- *DispatcherServlet* definition goes here----->

....

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory −

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

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

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

http://www.springframework.org/schema/context

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

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

</beans>

Following are the important points about **HelloWeb-servlet.xml** file −

* The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
* The *<context:component-scan...>* tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
* The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp* .

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

## **Defining a Controller**

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller**annotation indicates that a particular class serves the role of a controller. The **@RequestMapping**annotation is used to map a URL to either an entire class or a particular handler method.

@Controller

@RequestMapping("/hello")

public class HelloController {

@RequestMapping(method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation**@RequestMapping(method = RequestMethod.GET)** is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in *@RequestMapping* as follows −

@Controller

public class HelloController {

@RequestMapping(value = "/hello", method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above −

* You will define required business logic inside a service method. You can call another method inside this method as per requirement.
* Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
* A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

## **Creating JSP Views**

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp −

<html>

<head>

<title>Hello Spring MVC</title>

</head>

<body>

<h2>${message}</h2>

</body>

</html>

Here **${message}** is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

## **Spring Web MVC Framework Examples**

Based on the above concepts, let us check few important examples which will help you in building your Spring Web Applications −

|  |  |
| --- | --- |
| **Sr.No.** | **Example & Description** |
| 1 | [**Spring MVC Hello World Example**](https://www.tutorialspoint.com/spring/spring_mvc_hello_world_example.htm)  This example will explain how to write a simple Spring Web Hello World application. |
| 2 | [**Spring MVC Form Handling Example**](https://www.tutorialspoint.com/spring/spring_mvc_form_handling_example.htm)  This example will explain how to write a Spring Web application using HTML forms to submit the data to the controller and display a processed result. |
| 3 | [**Spring Page Redirection Example**](https://www.tutorialspoint.com/spring/spring_page_redirection_example.htm)  Learn how to use page redirection functionality in Spring MVC Framework. |
| 4 | [**Spring Static Pages Example**](https://www.tutorialspoint.com/spring/spring_static_pages_example.htm)  Learn how to access static pages along with dynamic pages in Spring MVC Framework. |
| 5 | [**Spring Exception Handling Example**](https://www.tutorialspoint.com/spring/spring_exception_handling_example.htm)  Learn how to handle exceptions in Spring MVC Framework. |

[**Spring MVC Hello World Example**](https://www.tutorialspoint.com/spring/spring_mvc_hello_world_example.htm)

**HelloController.java** file

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.ui.ModelMap;

@Controller

@RequestMapping("/hello")

public class HelloController {

@RequestMapping(method = RequestMethod.GET)public String printHello(ModelMap model) {

model.addAttribute("message", "Hello Spring MVC Framework!");

return "hello";

}

}

Following is the content of Spring Web configuration file **web.xml**

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Application</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

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

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

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

http://www.springframework.org/schema/context

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

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

</beans>

Following is the content of Spring view file **hello.jsp**

<%@ page contentType = "text/html; charset = UTF-8" %>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<h2>${message}</h2>

</body>

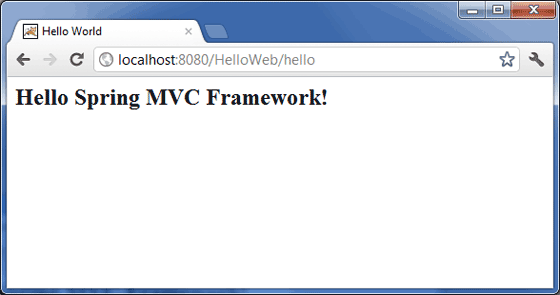
</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

* commons-logging-x.y.z.jar
* org.springframework.asm-x.y.z.jar
* org.springframework.beans-x.y.z.jar
* org.springframework.context-x.y.z.jar
* org.springframework.core-x.y.z.jar
* org.springframework.expression-x.y.z.jar
* org.springframework.web.servlet-x.y.z.jar
* org.springframework.web-x.y.z.jar
* spring-web.jar

Once you are done creating the source and configuration files, export your application. Right-click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Try to access the URL **http://localhost:8080/HelloWeb/hello** and if everything is fine with your Spring Web Application, you should see the following result −



You should note that in the given URL, **HelloWeb** is the application name and **hello** is the virtual subfolder which we have mentioned in our controller using @RequestMapping("/hello"). You can use direct root while mapping your URL using **@RequestMapping("/")**. In this case you can access the same page using short URL **http://localhost:8080/HelloWeb/** but it is advised to have different functionalities under different folders.

[**Spring MVC Form Handling Example**](https://www.tutorialspoint.com/spring/spring_mvc_form_handling_example.htm)

**Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of **StudentController.java** file

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

@RequestMapping(value = "/addStudent", method = RequestMethod.POST)

public String addStudent(@ModelAttribute("SpringWeb")Student student,

ModelMap model) {

model.addAttribute("name", student.getName());

model.addAttribute("age", student.getAge());

model.addAttribute("id", student.getId());

return "result";

}

}

Here the first service method **student()**, we have passed a blank **Student**object in the ModelAndView object with the name "command" because the Spring framework expects an object with the name "command" if you are using <form:form> tags in your JSP file. So, when **student()** method is called , it returns **student.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally a "result" view will be returned from the service method, which will result in rendering result.jsp

Following is the content of Spring Web configuration file **web.xml**

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring MVC Form Handling</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

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

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

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

http://www.springframework.org/schema/context

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

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

</beans>

Following is the content of Spring view file **student.jsp**

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Student Information</h2>

<form:form method = "POST" action = "/HelloWeb/addStudent">

<table>

<tr>

<td><form:label path = "name">Name</form:label></td>

<td><form:input path = "name" /></td>

</tr>

<tr>

<td><form:label path = "age">Age</form:label></td>

<td><form:input path = "age" /></td>

</tr>

<tr>

<td><form:label path = "id">id</form:label></td>

<td><form:input path = "id" /></td>

</tr>

<tr>

<td colspan = "2">

<input type = "submit" value = "Submit"/>

</td>

</tr>

</table>

</form:form>

</body></html>

Following is the content of Spring view file **result.jsp**

<%@page contentType = "text/html;charset = UTF-8" language = "java" %>

<%@page isELIgnored = "false" %>

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Submitted Student Information</h2>

<table>

<tr>

<td>Name</td>

<td>${name}</td>

</tr>

<tr>

<td>Age</td>

<td>${age}</td>

</tr>

<tr>

<td>ID</td>

<td>${id}</td>

</tr>

</table>

</body>

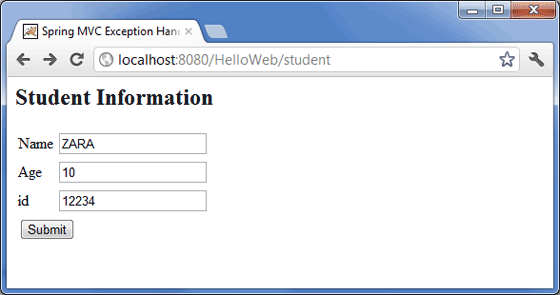
</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

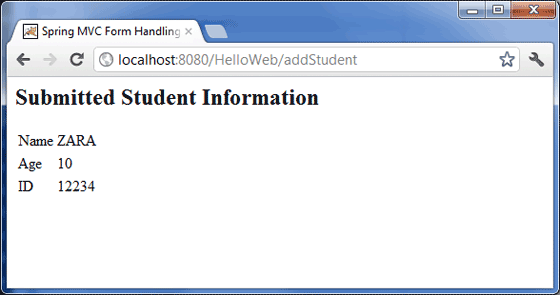
* commons-logging-x.y.z.jar
* org.springframework.asm-x.y.z.jar
* org.springframework.beans-x.y.z.jar
* org.springframework.context-x.y.z.jar
* org.springframework.core-x.y.z.jar
* org.springframework.expression-x.y.z.jar
* org.springframework.web.servlet-x.y.z.jar
* org.springframework.web-x.y.z.jar
* spring-web.jar

Once you are done with creating source and configuration files, export your application. Right click on your application and use the **Export > WAR File**option and save your **SpringWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try a URL **http://localhost:8080/SpringWeb/student** and you should see the following result if everything is fine with your Spring Web Application.



After submitting the required information, click the Submit button to submit the form. You should see the following result if everything is fine with your Spring Web Application.



[**Spring Page Redirection Example**](https://www.tutorialspoint.com/spring/spring_page_redirection_example.htm)

**WebController.java** file

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class WebController {

@RequestMapping(value = "/index", method = RequestMethod.GET)

public String index() {

return "index";

}

@RequestMapping(value = "/redirect", method = RequestMethod.GET)

public String redirect() {

return "redirect:finalPage";

}

@RequestMapping(value = "/finalPage", method = RequestMethod.GET)

public String finalPage() {

return "final";

}

}

Following is the content of Spring Web configuration file **web.xml**

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring Page Redirection</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

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

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

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

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

http://www.springframework.org/schema/context

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

<context:component-scan base-package = "com.tutorialspoint" />

<bean id = "viewResolver"

class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value=".jsp" />

</bean>

</beans>

Following is the content of Spring view file **index.jsp**. This will be a landing page, this page will send a request to access **redirect** service method which will redirect this request to another service method and finally a **final.jsp**page will be displauyed.

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring Page Redirection</title>

</head>

<body>

<h2>Spring Page Redirection</h2>

<p>Click below button to redirect the result to new page</p>

<form:form method = "GET" action = "/HelloWeb/redirect">

<table>

<tr>

<td>

<input type = "submit" value = "Redirect Page"/>

</td>

</tr>

</table>

</form:form>

</body>

</html>

Following is the content of Spring view file **final.jsp**. This is the final redirected page.

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring Page Redirection</title>

</head>

<body>

<h2>Redirected Page</h2>

</body>

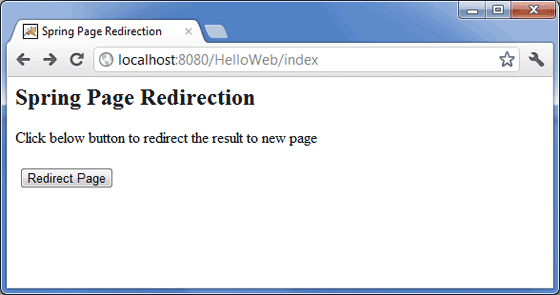
</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

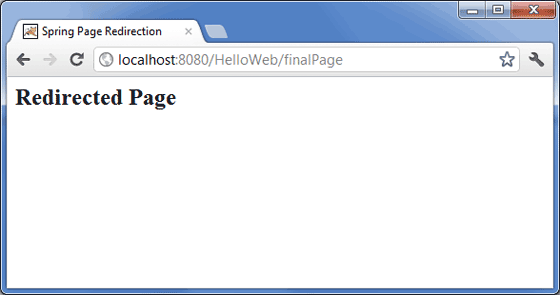
* commons-logging-x.y.z.jar
* org.springframework.asm-x.y.z.jar
* org.springframework.beans-x.y.z.jar
* org.springframework.context-x.y.z.jar
* org.springframework.core-x.y.z.jar
* org.springframework.expression-x.y.z.jar
* org.springframework.web.servlet-x.y.z.jar
* org.springframework.web-x.y.z.jar
* spring-web.jar

Once you are done with creating source and configuration files, export your application. Right click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Try a URL **http://localhost:8080/HelloWeb/index** and you should see the following result if everything is fine with your Spring Web Application.



Click the "Redirect Page" button to submit the form and to get the final redirected page. You should see the following result if everything is fine with your Spring Web Application.



[**Spring Static Pages Example**](https://www.tutorialspoint.com/spring/spring_static_pages_example.htm)

**WebController.java** file

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class WebController {

@RequestMapping(value = "/index", method = RequestMethod.GET)

public String index() {

return "index";

}

@RequestMapping(value = "/staticPage", method = RequestMethod.GET)

public String redirect() {

return "redirect:/pages/final.htm";

}

}

Following is the content of Spring Web configuration file **web.xml**

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring Page Redirection</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

<?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"

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

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

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

http://www.springframework.org/schema/mvc

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

http://www.springframework.org/schema/context

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

<context:component-scan base-package="com.tutorialspoint" />

<bean id = "viewResolver"

class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

<mvc:resources mapping = "/pages/\*\*" location = "/WEB-INF/pages/" />

<mvc:annotation-driven/>

</beans>

Here **<mvc:resources..../>** is being used to map static pages. The **mapping**attribute must be an Ant pattern that specifies the URL pattern of an http requests. The **location** attribute must specify one or more valid resource directory locations having static pages including images, stylesheets, JavaScript, and other static content. Multiple resource locations may be specified using a comma-seperated list of values.

Following is the content of Spring view file **WEB-INF/jsp/index.jsp**. This will be a landing page; this page will send a request to access **staticPage** service method, which will redirect this request to a static page available in WEB-INF/pages folder.

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring Landing Page</title>

</head>

<body>

<h2>Spring Landing Pag</h2>

<p>Click below button to get a simple HTML page</p>

<form:form method = "GET" action = "/HelloWeb/staticPage">

<table>

<tr>

<td>

<input type = "submit" value = "Get HTML Page"/>

</td>

</tr>

</table>

</form:form>

</body>

</html>

Following is the content of Spring view file **WEB-INF/pages/final.htm**.

<html>

<head>

<title>Spring Static Page</title>

</head>

<body>

<h2>A simple HTML page</h2>

</body>

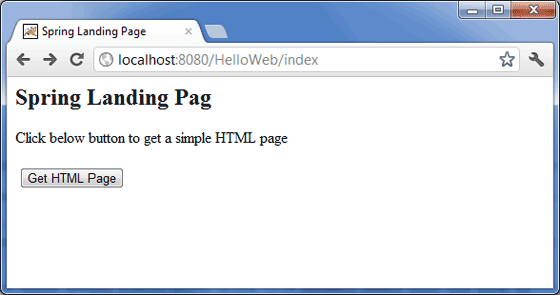
</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

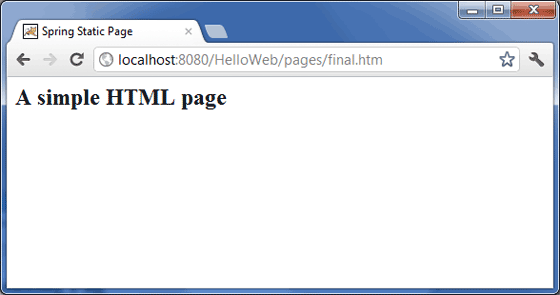
* commons-logging-x.y.z.jar
* org.springframework.asm-x.y.z.jar
* org.springframework.beans-x.y.z.jar
* org.springframework.context-x.y.z.jar
* org.springframework.core-x.y.z.jar
* org.springframework.expression-x.y.z.jar
* org.springframework.web.servlet-x.y.z.jar
* org.springframework.web-x.y.z.jar
* spring-web.jar

Once you are done creating the source and configuration files, export your application. Right-click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL **http://localhost:8080/HelloWeb/index**. If everything is fine with your Spring Web Application, you should see the following result −



Click the "Get HTML Page" button to access a static page mentioned in staticPage service method. If everything is fine with your Spring Web Application, you should see the following result.



[**Spring Exception Handling Example**](https://www.tutorialspoint.com/spring/spring_exception_handling_example.htm)

Following is the content of **Student.java** file

package com.tutorialspoint;

public class Student {

private Integer age;

private String name;

private Integer id;

public void setAge(Integer age) {

this.age = age;

}

public Integer getAge() {

return age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setId(Integer id) {

this.id = id;

}

public Integer getId() {

return id;

}

}

Following is the content of **SpringException.java** file

package com.tutorialspoint;

public class SpringException extends RuntimeException{

private String exceptionMsg;

public SpringException(String exceptionMsg) {

this.exceptionMsg = exceptionMsg;

}

public String getExceptionMsg(){

return this.exceptionMsg;

}

public void setExceptionMsg(String exceptionMsg) {

this.exceptionMsg = exceptionMsg;

}

}

Following is the content of **StudentController.java** file. Here, you need to annotate a service method using *@ExceptionHandler* where you can specify one or more exceptions to be handled. If you are specifying more than one exception then you can use comma separated values.

package com.tutorialspoint;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller

public class StudentController {

@RequestMapping(value = "/student", method = RequestMethod.GET)

public ModelAndView student() {

return new ModelAndView("student", "command", new Student());

}

@RequestMapping(value = "/addStudent", method = RequestMethod.POST)

@ExceptionHandler({SpringException.class})

public String addStudent( @ModelAttribute("HelloWeb")Student student,

ModelMap model) {

if(student.getName().length() < 5 ){

throw new SpringException("Given name is too short");

} else {

model.addAttribute("name", student.getName());

}

if( student.getAge() < 10 ){

throw new SpringException("Given age is too low");

} else {

model.addAttribute("age", student.getAge());

}

model.addAttribute("id", student.getId());

return "result";

}

}

Following is the content of Spring Web configuration file **web.xml**

<web-app id = "WebApp\_ID" version = "2.4"

xmlns = "http://java.sun.com/xml/ns/j2ee"

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

xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

<display-name>Spring Exception Handling</display-name>

<servlet>

<servlet-name>HelloWeb</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>HelloWeb</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Following is the content of another Spring Web configuration file **HelloWeb-servlet.xml**

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

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

axmlns: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

http://www.springframework.org/schema/context

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

<context:component-scan base-package = "com.tutorialspoint" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

<bean class = "org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name = "exceptionMappings">

<props>

<prop key = "com.tutorialspoint.SpringException">

ExceptionPage

</prop>

</props>

</property>

<property name = "defaultErrorView" value = "error"/>

</bean>

</beans>

Here you specified *ExceptionPage* as an exception view in case SpringException occurs, if there is any other type of exception then a generic view *error* will take place.

Following is the content of Spring view file **student.jsp**

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring MVC Exception Handling</title>

</head>

<body>

<h2>Student Information</h2>

<form:form method = "POST" action = "/HelloWeb/addStudent">

<table>

<tr>

<td><form:label path = "name">Name</form:label></td>

<td><form:input path = "name" /></td>

</tr>

<tr>

<td><form:label path = "age">Age</form:label></td>

<td><form:input path = "age" /></td>

</tr>

<tr>

<td><form:label path = "id">id</form:label></td>

<td><form:input path = "id" /></td>

</tr>

<tr>

<td colspan = "2"><input type = "submit" value = "Submit"/></td>

</tr>

</table>

</form:form>

</body>

</html>

Following is the content of Spring view file **error.jsp**

<html>

<head>

<title>Spring Error Page</title>

</head>

<body>

<p>An error occured, please contact webmaster.</p>

</body>

</html>

Following is the content of Spring view file **ExceptionPage.jsp**. Here you will access the exception instance via ${exception}.

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring MVC Exception Handling</title>

</head>

<body>

<h2>Spring MVC Exception Handling</h2>

<h3>${exception.exceptionMsg}</h3>

</body>

</html>

Following is the content of Spring view file **result.jsp**

<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>

<html>

<head>

<title>Spring MVC Form Handling</title>

</head>

<body>

<h2>Submitted Student Information</h2>

<table>

<tr>

<td>Name</td>

<td>${name}</td>

</tr>

<tr>

<td>Age</td>

<td>${age}</td>

</tr>

<tr>

<td>ID</td>

<td>${id}</td>

</tr>

</table>

</body>

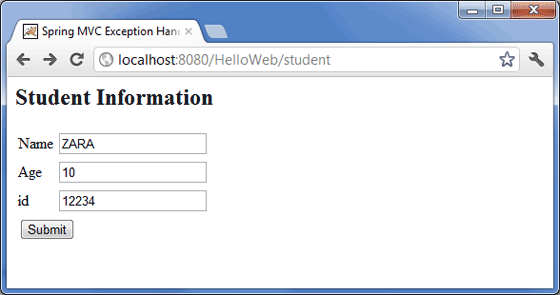
</html>

Finally, following is the list of Spring and other libraries to be included in your web application. You simply drag these files and drop them in **WebContent/WEB-INF/lib** folder.

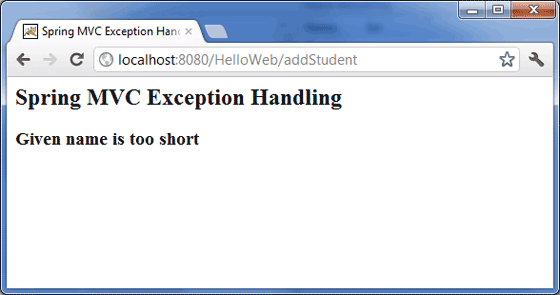
* commons-logging-x.y.z.jar
* org.springframework.asm-x.y.z.jar
* org.springframework.beans-x.y.z.jar
* org.springframework.context-x.y.z.jar
* org.springframework.core-x.y.z.jar
* org.springframework.expression-x.y.z.jar
* org.springframework.web.servlet-x.y.z.jar
* org.springframework.web-x.y.z.jar
* spring-web.jar

Once you are done creating the source and configuration files, export your application. Right-click on your application and use the **Export > WAR File**option and save your**HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other web pages from webapps folder using a standard browser. Now try to access the URL **http://localhost:8080/HelloWeb/student**. If everything is fine with your Spring Web Applicationand, you should see the following result.



Enter the values as shown above and click the Submit buttom. If everything is fine with your Spring Web Application, you should see the following result.



**Logging with Log4J**

This is a very easy-to-use Log4J functionality inside Spring applications. The following example will take you through simple steps to explain the simple integration between Log4J and Spring.

**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 second file **MainApp.java**

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.apache.log4j.Logger;

public class MainApp {

static Logger log = Logger.getLogger(MainApp.class.getName());

public static void main(String[] args) {

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

log.info("Going to create HelloWord Obj");

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

obj.getMessage();

log.info("Exiting the program");

}

}

You can generate **debug** and **error** message in a similar way as we have generated info messages. Now let us see the content of **Beans.xml** file

<?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">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

Following is the content of **log4j.properties** which defines the standard rules required for Log4J to produce log messages

# Define the root logger with appender file

log4j.rootLogger = DEBUG, FILE

# Define the file appender

log4j.appender.FILE=org.apache.log4j.FileAppender

# Set the name of the file

log4j.appender.FILE.File=C:\\log.out

# Set the immediate flush to true (default)

log4j.appender.FILE.ImmediateFlush=true

# Set the threshold to debug mode

log4j.appender.FILE.Threshold=debug

# Set the append to false, overwrite

log4j.appender.FILE.Append=false

# Define the layout for file appender

log4j.appender.FILE.layout=org.apache.log4j.PatternLayout

log4j.appender.FILE.layout.conversionPattern=%m%n

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

Your Message : Hello World!

If you check your C:\\ drive, then you should find your log file **log.out** with various log messages, like something as follows −

<!-- initialization log messages -->

Going to create HelloWord Obj

Returning cached instance of singleton bean 'helloWorld'

Exiting the program

## **Jakarta Commons Logging (JCL) API**

Alternatively you can use **Jakarta Commons Logging (JCL)** API to generate a log in your Spring application. JCL can be downloaded from the <https://jakarta.apache.org/commons/logging/>. The only file we technically need out of this package is the *commons-logging-x.y.z.jar* file, which needs to be placed in your classpath in a similar way as you had put *log4j-x.y.z.jar* in the above example.

To use the logging functionality you need a *org.apache.commons.logging.Log*object and then you can call one of the following methods as per your requirment −

* fatal(Object message)
* error(Object message)
* warn(Object message)
* info(Object message)
* debug(Object message)
* trace(Object message)

Following is the replacement of MainApp.java, which makes use of JCL API

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.apache.commons.logging. Log;

import org.apache.commons.logging. LogFactory;

public class MainApp {

static Log log = LogFactory.getLog(MainApp.class.getName());

public static void main(String[] args) {

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

log.info("Going to create HelloWord Obj");

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

obj.getMessage();

log.info("Exiting the program");

}

}

You have to make sure that you have included commons-logging-x.y.z.jar file in your project, before compiling and running the program.

Now keeping the rest of the configuration and content unchanged in the above example, if you compile and run your application, you will get a similar result as what you got using Log4J API.

**How Spring MVC works –**

It all starts with the client, which sends a request to a specific URL. When that request hit the web container e.g. Tomcat it look into web.xml and find the Servlet or Filter which is mapped to that particular URL. It the delegate that Servlet or Filter to process the request. Since Spring MVC is built on top of Servlet, this is also the initial flow of request in any Spring MVC based Java web application.  
  
Remember, Web container e.g. Tomcat is responsible for creating Servlet and Filter instances and invoking their various life-cycle methods e.g. [init()](http://javarevisited.blogspot.sg/2015/02/constructor-vs-init-method-in-servlet.html), service(), destroy(). In the case of HTTP request, HttpServlet handles that and depending upon the HTTP request method various doXXX() method is invoked by container e.g. doGet() to process GET request and doPost() to process POST request.  
  
If you remember, to enable Spring MVC, we need to declare the **DispatcherServlet** from Spring MVC jar into web.xml. This [Servlet](http://javarevisited.blogspot.sg/2017/05/difference-between-servlet-and-jsp.html)listens for a URL pattern \* as shown in below web.xml, which means all request is mapped to DispatcherServlet.  
  
Though it is not mandatory, you can have other servlet mapped to other URL if you want to, but if you are using Spring MVC to develop web application or RESTful web service, it make sense to pass through all request via DispatcherServlet.  
  
  
Here is the web.xml configuration for Spring MVC, you can see that DispatcherServlet is mapped to all request using URL pattern \*

<web-app>

<!-- The front controller of this Spring Web application, responsible

for handling all application requests -->

<servlet>

<servlet-name>Spring MVC [Dispatcher Servlet](http://javarevisited.blogspot.sg/2015/09/eclipse-javalangclassnotfoundexception.html#axzz4jWEJmi6S)</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/config/web-application-config.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

<url-pattern>\*</url-pattern>

</servlet-mapping>

</web-app>

The URL pattern is important, if the request matches the URL pattern of DispatcherServlet then it will be processed by [Spring MVC](http://javarevisited.blogspot.sg/2012/05/what-is-bean-scope-in-spring-mvc.html#axzz4jWEJmi6S) otherwise not. The DispatcherServlet the passes the request to a specific controller depending on the URL requested. How does DispatcherServlet know which request needs to be passed to which controller?  
  
  
Well, it uses the **@RequestMapping** annotation or Spring MVC configuration file to find out mapping of request URL to different controllers. It can also use specific request processing annotations e.g. @GetMapping or @PostMapping. Controller classes are also identified using @Controller and @RestController (in the case of RESTful Web Services) annotations. See [REST with Spring](http://www.baeldung.com/rest-with-spring-course?utm_source=javarevisited&utm_medium=web&utm_campaign=rws&affcode=22136_bkwjs9xa) course by Eugen to learn how to develop RESTful Web Service using Spring in depth.  
  
For example, below class is a Controller which will process any request having URI "/appointments". It also has @GetMapping, which means that method will be invoked when a GET request is received for this URL. The method annotated with @PostMapping will be invoked if the client sends a [POST request](http://www.java67.com/2014/08/difference-between-post-and-get-request.html) to the "/appointments" URI.

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

@GetMapping

public Map get() {

return appointmentBook.getAppointmentsForToday();

}

@PostMapping

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

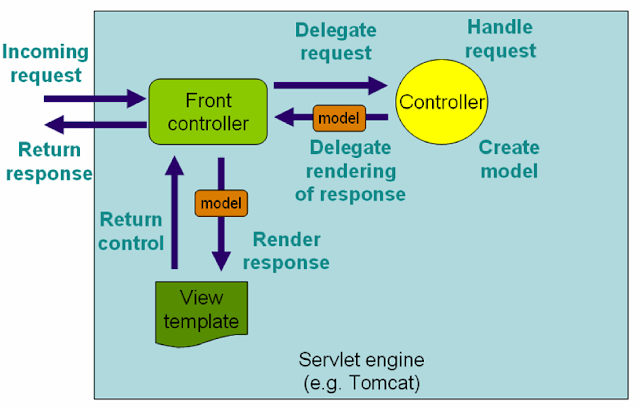
}

}

After processing the request, Controller returns a **logical view name** and model to [DispatcherServlet](http://javarevisited.blogspot.sg/2016/01/solving-javalangclassnotfoundexception-org.springframework.web.servlet.DispatcherServlet.html)and it consults view resolvers until an actual View is determined to render the output. DispatcherServlet then contacts the chosen view e.g. Freemarker or JSP with model data and it renders the output depending on the model data.  
  
This Rendered output is returned to the client as HTTP response. On it's way back it can pass to any configured Filter as well e.g. [Spring Security filter chain](http://javarevisited.blogspot.sg/2017/05/how-to-enable-spring-security-in-java-web-application.html#axzz4gg59c400) or Filters configured to convert the response to JSON or XML.  
  
The DispatcherServlet from Spring MVC framework is an implementation of Front Controller Pattern (see [Patterns of Enterprise Application Architecture](https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420?tag=javamysqlanta-20)) and it's also a Single point of entry - handle all incoming requests, but again that depends upon your URL pattern mapping and your application.  
  
It delegates requests for further processing to additional components e.g. Controllers, Views, View Resolvers, handler mappers, exception handlers etc. It can also map directly to /, but then the exception for handling static resources needs to be configured. If you look at the web.xml configuration it also pre-loaded using the [load-on-startup](http://javarevisited.blogspot.sg/2011/12/load-on-startup-servlet-webxml-example.html#axzz4jWEcwHFD) tag.

## Spring MVC work Flow

It's been often said that a picture is worth a thousand words and this is very true in the case of understanding system architecture and workflow of your application. Whatever I have said in above article, can be easily inferred by looking at following diagram which explains workflow of Spring MVC framework:

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspringmvc-intro)

The flow of [RESTful Web Service](http://javarevisited.blogspot.sg/2017/02/how-to-consume-json-from-restful-web-services-Spring-RESTTemplate-Example.html#axzz4jWEcwHFD) request is also not very different from this. It follows the same path but in the case of REST, the Controller methods are annotated with @ResponseBody which means it doesn't return a logical view name to DispatcherServlet, instead it write the output directly to HTTP response body. See [Spring REST](https://www.amazon.com/Spring-REST-Balaji-Varanasi/dp/1484208242/?tag=javamysqlanta-20) book to learn more about how to develop RESTful Web Services using Spring.  
  
In summary, here is the flow of an HTTP request in Java application created using Spring MVC framework:  
  
1) Client sends an HTTP request to a specific URL  
  
2) DispatcherServlet of Spring MVC receives the request  
  
2) It passes the request to a specific controller depending on the URL requested using @Controller and @RequestMapping annotations.  
  
3) Spring MVC Controller then returns a logical view name and model to DispatcherServlet.  
  
4) DispatcherServlet consults view resolvers until actual View is determined to render the output  
  
5) DispatcherServlet contacts the chosen view (e.g. Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data  
  
6) The rendered output is returned to the client as response  
  
That's all about **what is the flow of Spring MVC** or **how an HTTP request is processed by Spring** MVC. This is very basic but important knowledge about Spring MVC framework and every Java and Spring developer should be familiar with this. If you know how your HTTP request is processed then you can not only understand the issues better but also troubleshoot then easily and quickly.

# What is the front controller class of Spring MVC?

A front controller is defined as a controller that handles all requests for a Web Application. DispatcherServlet servlet is the front controller in Spring MVC that intercepts every request and then dispatches requests to an appropriate controller. When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring?s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.