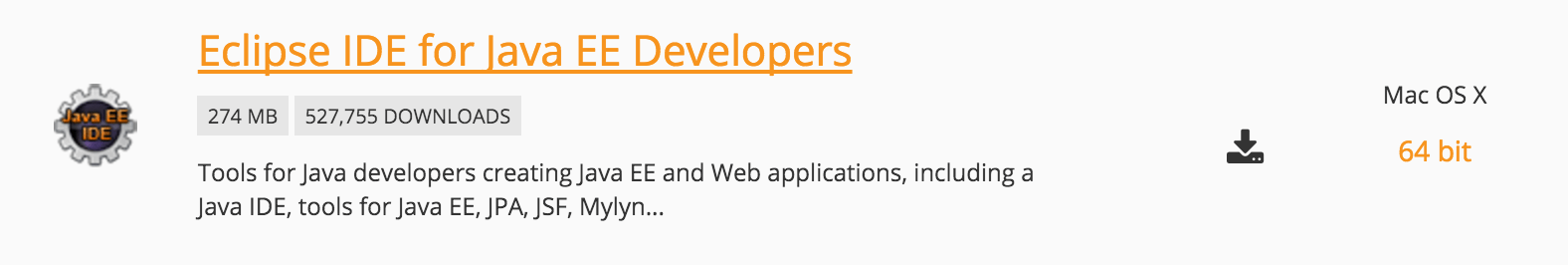
First thing first – Let’s Setup Environment

1. Tomcat 8.0.37 – Download latest Apache Tomcat from this [link](http://tomcat.apache.org/download-80.cgi).
2. Make sure you download Eclipse IDE for Java EE Developers (Oxygen v4.7.x) – Download [link](http://www.eclipse.org/downloads/eclipse-packages/). (diagram below)
3. Spring 5.0.3 (No download required) – we will use [Maven dependency](https://crunchify.com/how-to-import-all-spring-mvc-dependencies-to-your-maven-project/).
4. JDK 1.8 – Download [link](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html).

Make sure you download Java EE:



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

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

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

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

## Benefits of Using the Spring Framework

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

* Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product.
* Spring is organized in a modular fashion. Even though the number of packages and classes are substantial, you have to worry only about the ones you need and ignore the rest.
* Spring does not reinvent the wheel, instead it truly makes use of some of the existing technologies like several ORM frameworks, logging frameworks, JEE, Quartz and JDK timers, and other view technologies.
* Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.
* Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
* Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
* Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
* Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

## Dependency Injection (DI)

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

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

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

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

## Aspect Oriented Programming (AOP)

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

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

# Architecture

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

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



## Core Container

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

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

## Data Access/Integration

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

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

## Web

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

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

## Miscellaneous

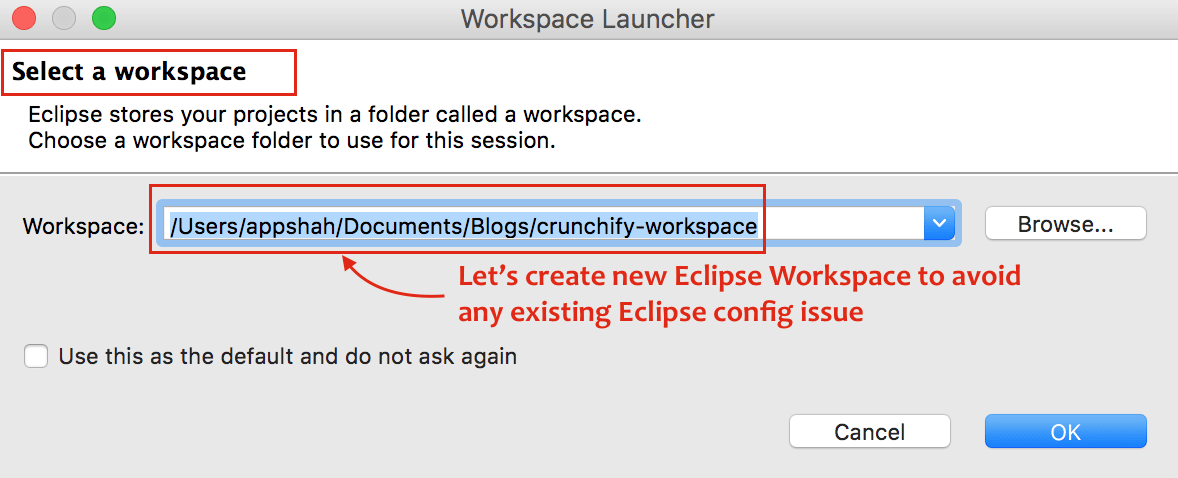
There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

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

Project -1

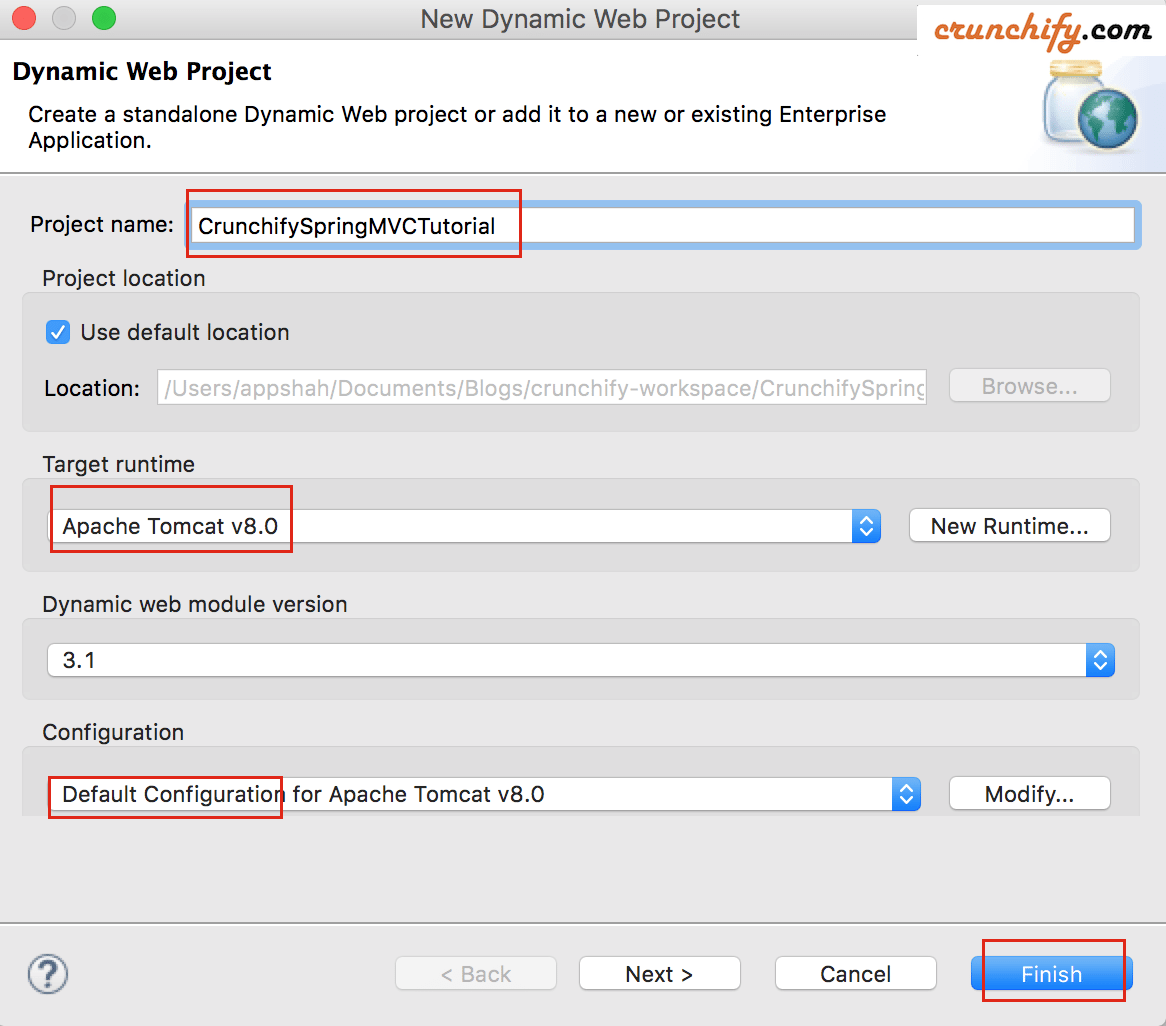
### Step-1

* Open [Eclipse](https://crunchify.com/missing-maven-settings-xml-file-for-your-eclipse-what-if-you-need-two-settings-xml-file-for-work-personal-workspace/)
* Create New Eclipse Workspace – This is must to avoid any existing workspace config issue.



### Step-2

* Click on File
* Click on New
* Choose Dynamic Web Project
* One popup window, Provide Project Name: CrunchifySpringMVCTutorial
* Make sure you use Target Runtime as Apache Tomcat 8.0. If you don’t see Target Runtime then [follow these steps](https://crunchify.com/step-by-step-guide-to-setup-and-install-apache-tomcat-server-in-eclipse-development-environment-ide/).

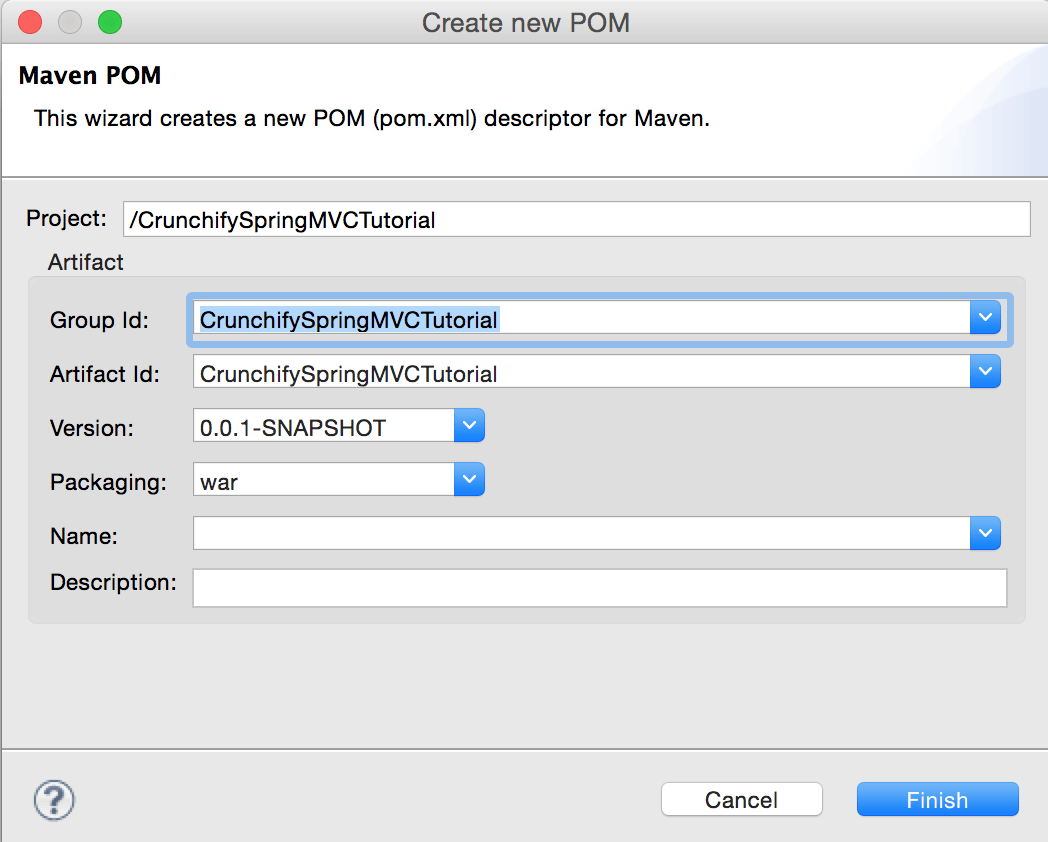


### Step-3

Convert Project to [Maven Project](https://crunchify.com/how-to-create-a-war-file-from-eclipse-using-maven-plugin-apache-maven-war-plugin-usage/) to add all required Spring MVC dependencies to project.

#### Steps:

* Right click on project
* Configure
* Convert to Maven project



### Step-4

Open pom.xml file and add below jar dependencies to project.



Here is my pom.xml file. Make sure you update Java version to 1.8 if you haven’t yet moved to JDK 1.8.

pom.xml

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>CrunchifySpringMVCTutorial</groupId>  <artifactId>CrunchifySpringMVCTutorial</artifactId>  <version>0.0.1-SNAPSHOT</version>  <build>  <sourceDirectory>src</sourceDirectory>  <plugins>  <plugin>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.1</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>    </plugins>  </build>  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>5.0.3.RELEASE</version>  </dependency>    <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  </dependencies>  </project> |

### Step-5

Create new Spring Configuration Bean file: /WebContent/WEB-INF/crunchify-servlet.xml

/WEB-INF/crunchify-servlet.xml

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:mvc="http://www.springframework.org/schema/mvc" 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.xsd          http://www.springframework.org/schema/mvc          http://www.springframework.org/schema/mvc/spring-mvc.xsd          http://www.springframework.org/schema/context          http://www.springframework.org/schema/context/spring-context.xsd">    <context:component-scan base-package="com.crunchify.controller" />    <bean id="viewResolver"  class="org.springframework.web.servlet.view.UrlBasedViewResolver">  <property name="viewClass"  value="org.springframework.web.servlet.view.JstlView" />  <property name="prefix" value="/WEB-INF/jsp/" />  <property name="suffix" value=".jsp" />  </bean>    </beans> |

In the above crunchify-servlet.xml configuration file, we have defined a tag <context:component-scan> . This will allow Spring to load all the components from package com.crunchify.controller  and all its child packages.

This will load our CrunchifyHelloWorld.class . Also we have defined a bean viewResolver. This bean will resolve the view and add prefix string /WEB-INF/jsp/  and suffix .jsp to the view in ModelAndView. Note that in our CrunchifyHelloWorld class, we have return a ModelAndView object with view name welcome. This will be resolved to path /WEB-INF/jsp/welcome.jsp .

**Step-6**

Create new file web.xml. Map Spring MVC in /WebContent/WEB-INF/web.xmlfile.

NOTE: if you don’t see web.xml file in your “dynamic web project” then follow [these steps](https://crunchify.com/eclipse-missing-web-xml-file-how-can-i-create-web-xml-in-eclipse/).

/WEB-INF/web.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" version="3.0">    <display-name>CrunchifySpringMVCTutorial</display-name>    <welcome-file-list>      <welcome-file>index.jsp</welcome-file>    </welcome-file-list>        <servlet>          <servlet-name>crunchify</servlet-name>          <servlet-class>              org.springframework.web.servlet.DispatcherServlet          </servlet-class>          <load-on-startup>1</load-on-startup>      </servlet>      <servlet-mapping>          <servlet-name>crunchify</servlet-name>          <url-pattern>/welcome.jsp</url-pattern>          <url-pattern>/index.jsp</url-pattern>          <url-pattern>/welcome.html</url-pattern>          <url-pattern>\*.html</url-pattern>      </servlet-mapping>    </web-app> |

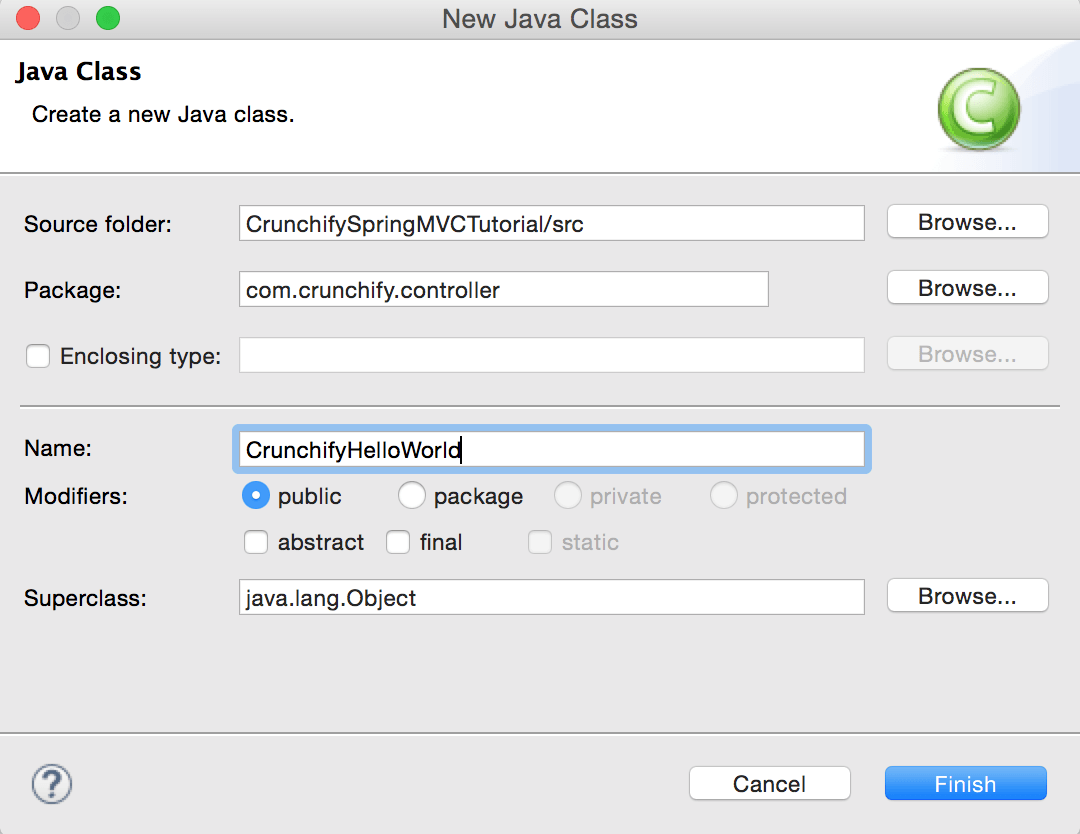
The above code in web.xml will map DispatcherServlet with url pattern /welcome.jsp. Also note that we have define index.jsp as welcome file.

One thing to note here is the name of servlet in <servlet-name> tag in web.xml. Once the DispatcherServlet is initialized, it will looks for a file name [[servlet](https://crunchify.com/how-to-do-java-servlet-session-management-using-cookies/)-name]-servlet.xml  in WEB-INF folder of web application. In this example, the framework will look for file called crunchify-servlet.xml.

**Step-7**

Create Controller Class.

* Right click on Java Resources -> src
* Click New -> Class
* Package: com.crunchify.controller
* Filename: CrunchifyHelloWorld.java



com.crunchify.controller.CrunchifyHelloWorld.java

Java

|  |
| --- |
| package com.crunchify.controller;    import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.servlet.ModelAndView;    /\*  \* author: Crunchify.com  \*  \*/    @Controller  public class CrunchifyHelloWorld {    @RequestMapping("/welcome")  public ModelAndView helloWorld() {    String message = "<br><div style='text-align:center;'>"  + "<h3>\*\*\*\*\*\*\*\*\*\* Hello World, Spring MVC Tutorial</h3>This message is coming from CrunchifyHelloWorld.java \*\*\*\*\*\*\*\*\*\*</div><br><br>";  return new ModelAndView("welcome", "message", message);  }  } |

Note that we have annotated the CrunchifyHelloWorld class with @Controller and @RequestMapping("/welcome"). When Spring scans our package, it will recognize this bean as being a [Controller bean](https://crunchify.com/working-on-spring-mvc-project-how-to-report-list-of-all-loaded-spring-beans-during-startup/) for processing requests. The @RequestMapping annotation tells [Spring](https://crunchify.com/how-to-import-all-spring-mvc-dependencies-to-your-maven-project/) that this Controller should process all requests beginning with /welcome in the URL path. That includes /welcome/\* and /welcome.html.

The helloWorld() method returns ModelAndView object. The ModelAndView object tries to resolve to a view named “welcome” and the data model is being passed back to the browser so we can access the data within the JSP. The logical view name will resolve to /WEB-INF/jsp/welcome.jsp . Logical name “welcome” which is return in ModelAndView object is mapped to path /WEB-INF/jsp/welcome.jsp.

The ModelAndView object also contains a message with key “message” and Detailed value. This is the data that we are passing to our view. Normally this will be a value object in form of java bean that will contain the data to be displayed on our view. Here we are simply passing a string.

**Step-8**

**The View –**Create new file /WebContent/index.jsp.

index.jsp

|  |
| --- |
| <html>  <head>  <title>Spring MVC Tutorial Series by Crunchify.com</title>  <style type="text/css">  body {  background-image: url('https://crunchify.com/bg.png');  }  </style>  </head>  <body>  <br>  <div style="text-align:center">  <h2>  Hey You..!! This is your 1st Spring MCV Tutorial..<br> <br>  </h2>  <h3>  <a href="welcome.html">Click here to See Welcome Message... </a>(to  check Spring MVC Controller... @RequestMapping("/welcome"))  </h3>  </div>  </body>  </html> |

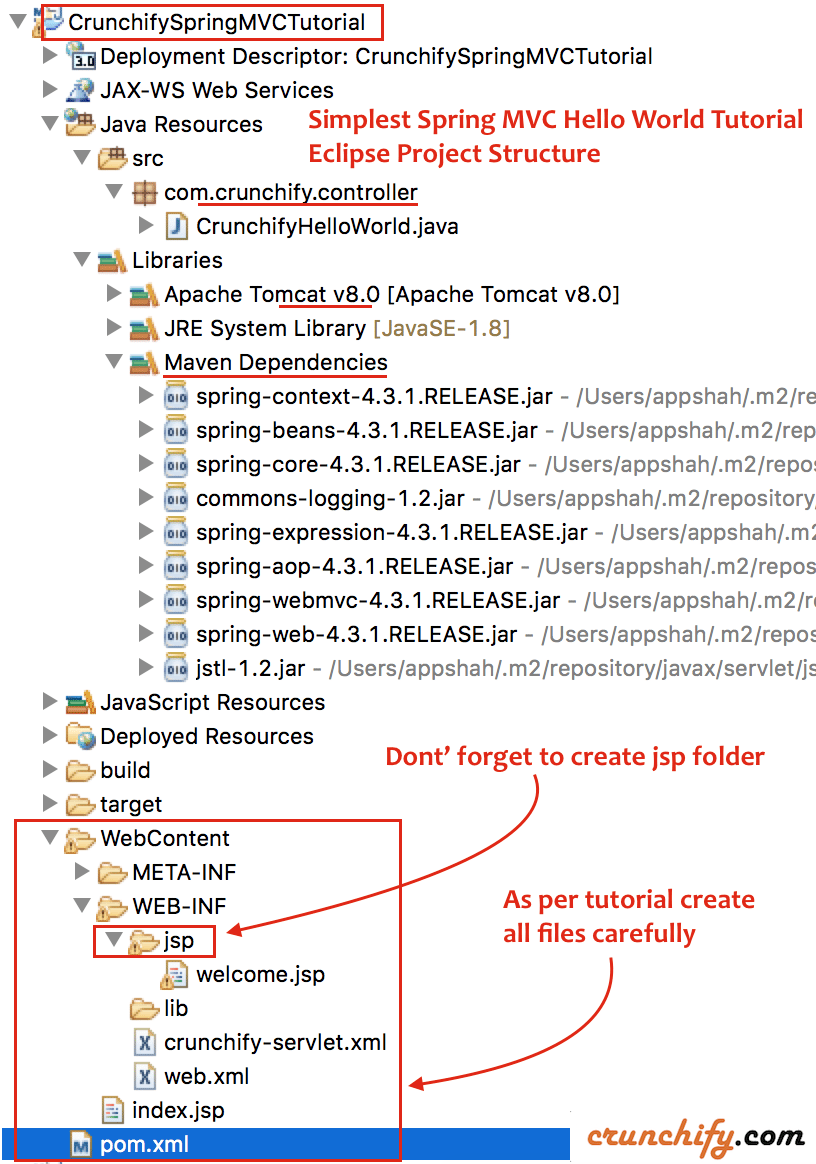
Create another file /WebContent/WEB-INF/jsp/welcome.jsp.

NOTE: Don’t forget to create jsp folder and put welcome.jsp inside that 🙂

welcome.jsp

|  |
| --- |
| <html>  <head>  <title>Spring MVC Tutorial by Crunchify - Hello World Spring MVC  Example</title>  <style type="text/css">  body {  background-image: url('https://crunchify.com/bg.png');  }  </style>  </head>  <body>${message}    <br>  <br>  <div style="font-family: verdana; padding: 10px; border-radius: 10px; font-size: 12px; text-align:center;">    Spring MCV Tutorial by <a href="https://crunchify.com">Crunchify</a>.  Click <a  href="https://crunchify.com/category/java-tutorials/"  target="\_blank">here</a> for all Java and <a  href='https://crunchify.com/category/spring-mvc/' target='\_blank'>here</a>  for all Spring MVC, Web Development examples.<br>  </div>  </body>  </html> |

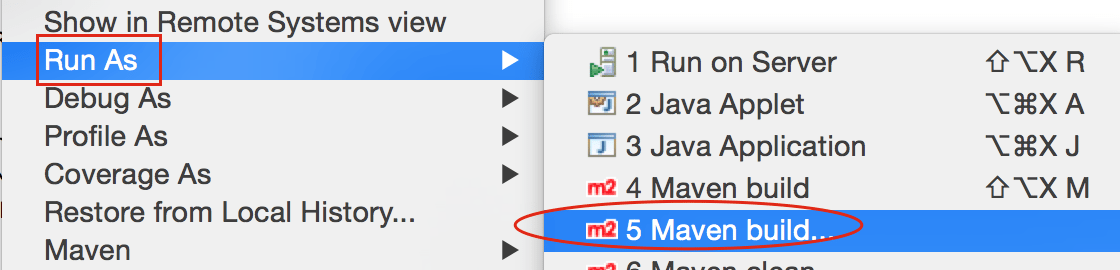
After everything this is how your workspace should look like.



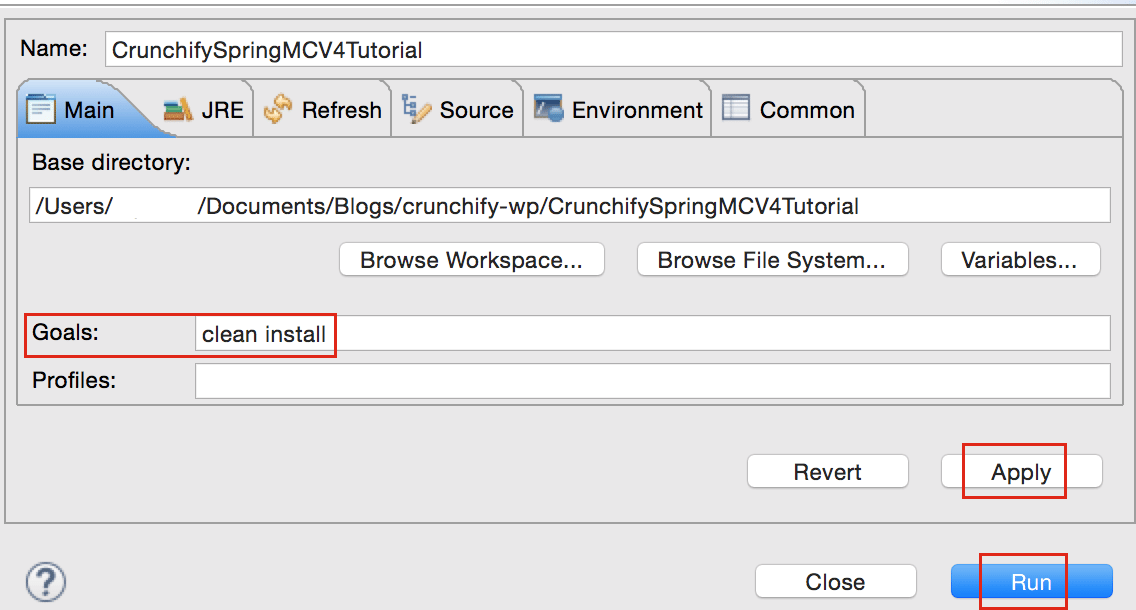
*NOTE: I’ve updated maven dependencies from 4.3.1 to 5.0.3 already. It’s good idea to keep updating your project to latest Spring MVC version.*

**Step-9**

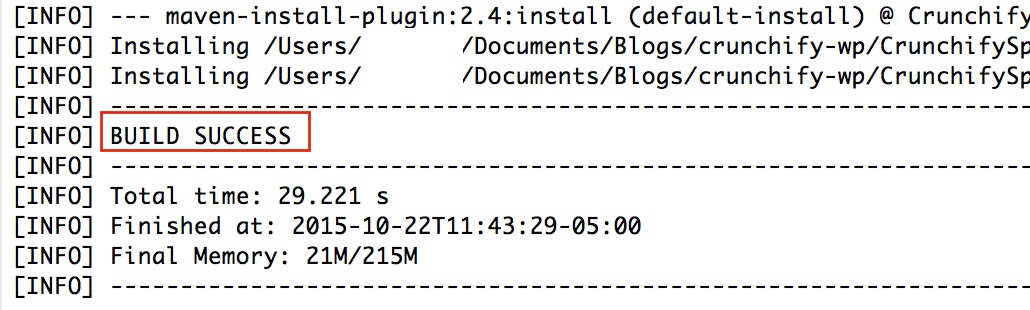
Right Click on Project -> Run As -> Maven Build...



Add Goals: clean install. Click Apply and Run.



You should see build success message:

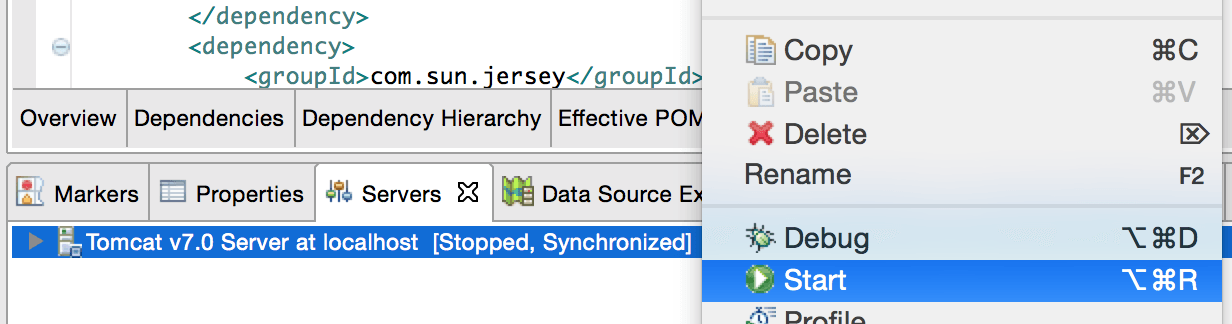


**Where are all of my .jar files?**

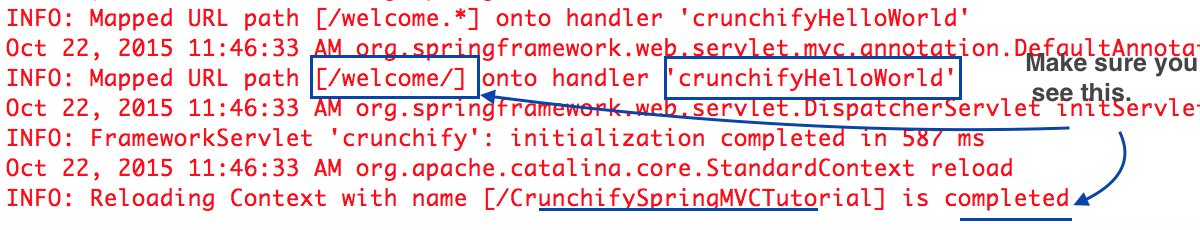
You will see all .jar files under /target folder. [Screenshot](https://cdn.crunchify.com/wp-content/uploads/2013/02/After-Maven-Clean-Install-target-folder-contains-all-.jar-files.png).

**Step-10**

* If you don't see Tomcat Server in Servers tab then follow steps to [add Apache Tomcat to Eclipse](https://crunchify.com/step-by-step-guide-to-setup-and-install-apache-tomcat-server-in-eclipse-development-environment-ide/).
* Deploy project to Apache Tomcat and start tomcat.



Make sure you see below logs. That means your application is successfully deployed on Tomcat Web Server.



**Step-11**

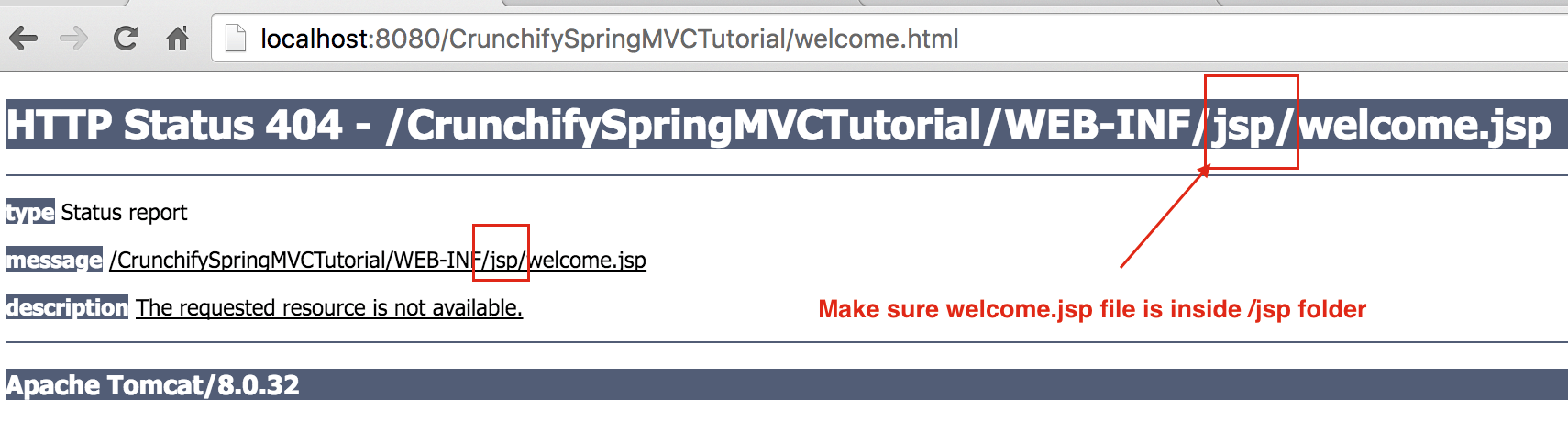
Visit: <http://localhost:8080/CrunchifySpringMVCTutorial/> and you should be all set.

Hurray.. Now you know Hello World Spring MVC 5 Example. Let me know if you encounter any exception while running this. There are lot more example you can find [here](https://crunchify.com/category/java-tutorials/).

Do you want to include JS, CSS and images into JSP file? Follow this tutorial: [Best way to Add/Integrate JS, CSS and images into JSP file using ‘mvc:resources mapping’](https://crunchify.com/spring-mvc-4-2-2-best-way-to-integrate-js-and-css-file-in-jsp-file-using-mvcresources-mapping/).

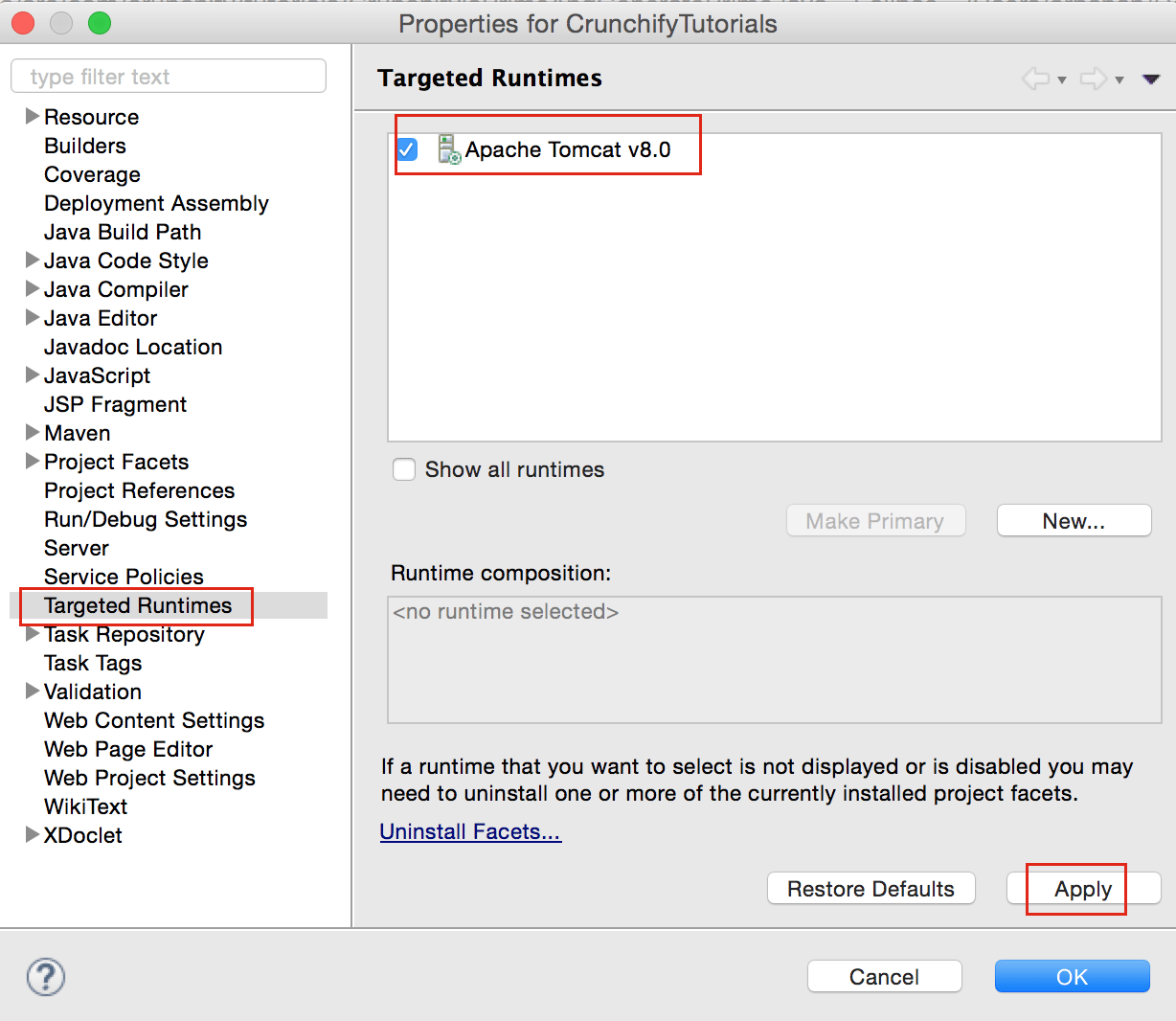
Having trouble? Any issue?

**Triaging Step-1 – Having HTTP Status 404 error?**



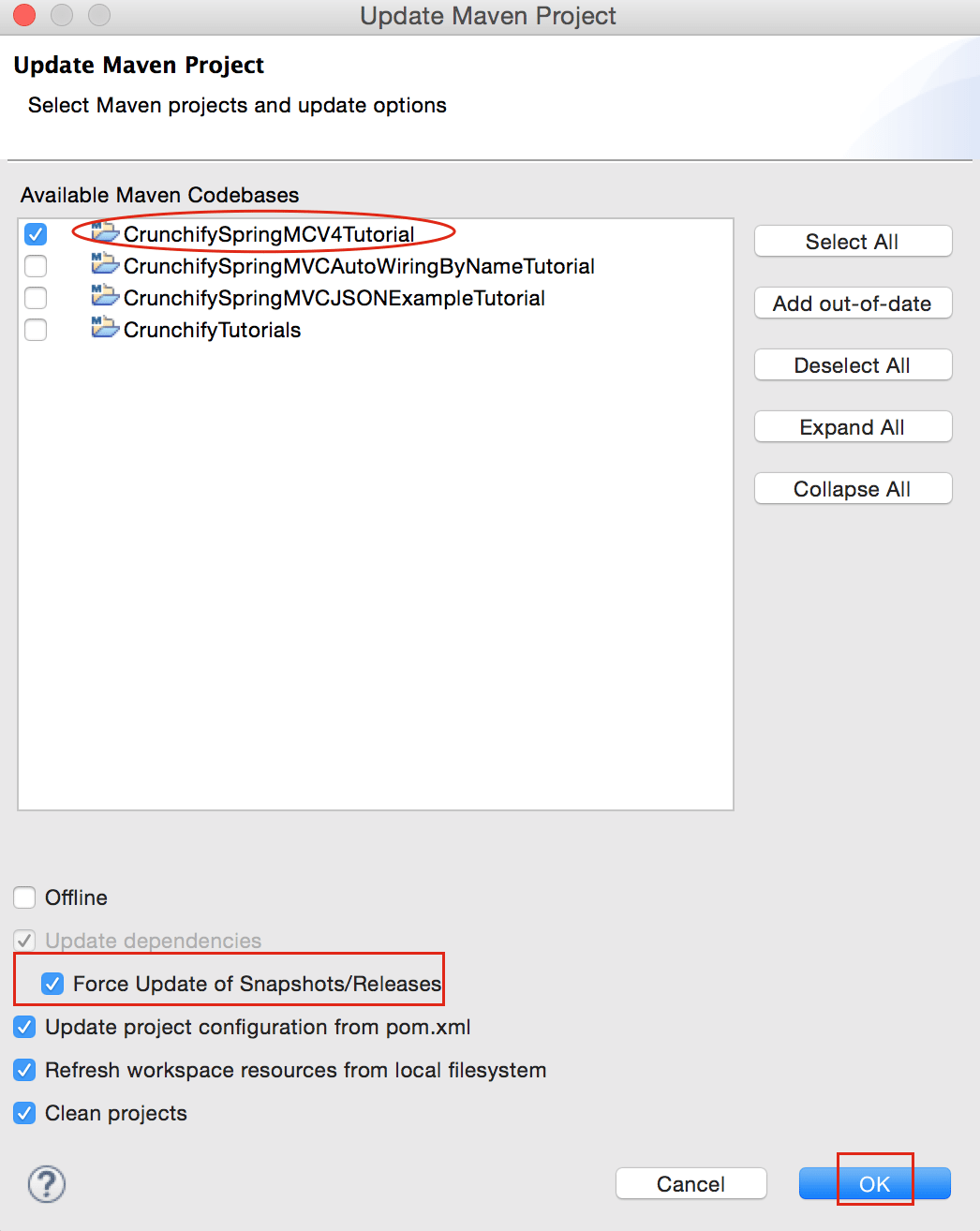
### Triaging step-2 – URL doesn’t work? Tomcat error?

Make sure you add Apache Tomcat Server to Targeted Runtime. Which you may have selected in Step-1. Tomcat 7 or 8 any – server should work.



### Triaging Step-3 – maven errors?

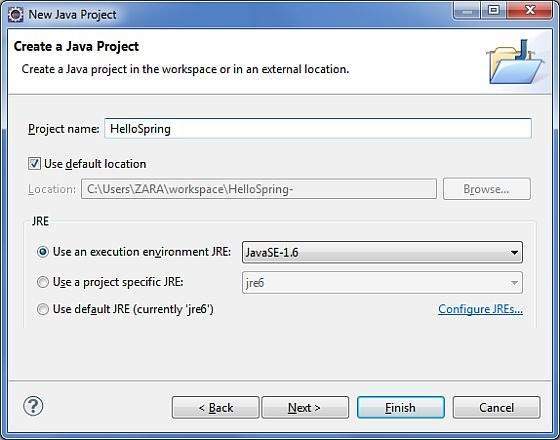
Make sure to update all maven dependencies.



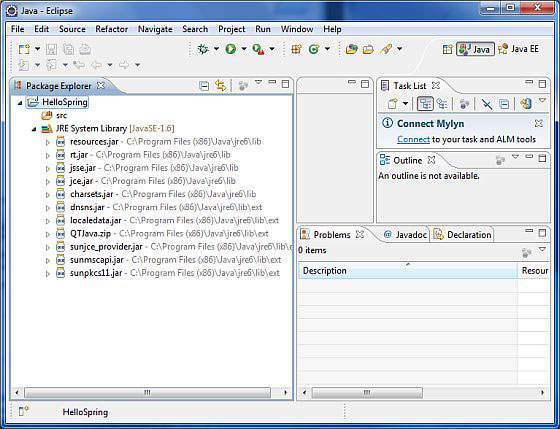
Project -2

## Step 1 - Create Java Project

The first step is to create a simple Java Project using Eclipse IDE. Follow the option **File → New → Project** and finally select **Java Project** wizard from the wizard list. Now name your project as **HelloSpring** using the wizard window as follows −

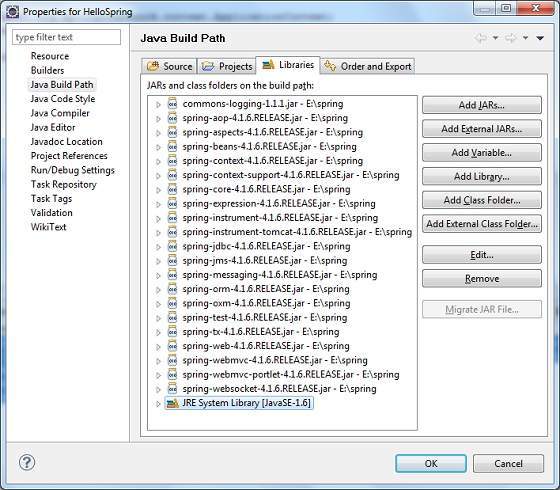


Once your project is created successfully, you will have the following content in your **Project Explorer** −



## Step 2 - Add Required Libraries

As a second step let us add Spring Framework and common logging API libraries in our project. To do this, right-click on your project name **HelloSpring** and then follow the following option available in the context menu − **Build Path → Configure Build Path** to display the Java Build Path window as follows −



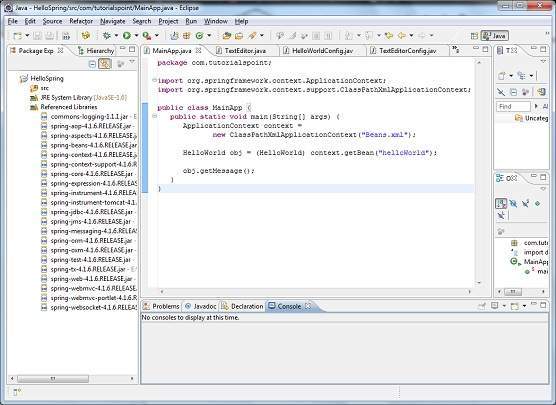
Now use **Add External JARs** button available under the **Libraries** tab to add the following core JARs from Spring Framework and Common Logging installation directories −

* commons-logging-1.1.1
* spring-aop-4.1.6.RELEASE
* spring-aspects-4.1.6.RELEASE
* spring-beans-4.1.6.RELEASE
* spring-context-4.1.6.RELEASE
* spring-context-support-4.1.6.RELEASE
* spring-core-4.1.6.RELEASE
* spring-expression-4.1.6.RELEASE
* spring-instrument-4.1.6.RELEASE
* spring-instrument-tomcat-4.1.6.RELEASE
* spring-jdbc-4.1.6.RELEASE
* spring-jms-4.1.6.RELEASE
* spring-messaging-4.1.6.RELEASE
* spring-orm-4.1.6.RELEASE
* spring-oxm-4.1.6.RELEASE
* spring-test-4.1.6.RELEASE
* spring-tx-4.1.6.RELEASE
* spring-web-4.1.6.RELEASE
* spring-webmvc-4.1.6.RELEASE
* spring-webmvc-portlet-4.1.6.RELEASE
* spring-websocket-4.1.6.RELEASE

## Step 3 - Create Source Files

Now let us create actual source files under the **HelloSpring** project. First we need to create a package called **com.tutorialspoint**. To do this, right click on **src** in package explorer section and follow the option − **New → Package**.

Next we will create **HelloWorld.java** and **MainApp.java** files under the com.tutorialspoint package.



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 second file **MainApp.java** −

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

obj.getMessage();

}

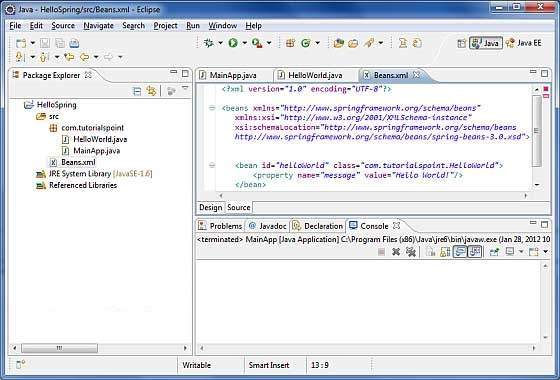
}

Following two important points are to be noted about the main program −

* The first step is to create an application context where we used framework API **ClassPathXmlApplicationContext()**. This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
* The second step is used to get the required bean using **getBean()**method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

## Step 4 - Create Bean Configuration File

You need to create a Bean Configuration file which is an XML file and acts as a cement that glues the beans, i.e. the classes together. This file needs to be created under the **src** directory as shown in the following screenshot −



Usually developers name this file as **Beans.xml**, but you are independent to choose any name you like. You have to make sure that this file is available in CLASSPATH and use the same name in the main application while creating an application context as shown in MainApp.java file.

The Beans.xml is used to assign unique IDs to different beans and to control the creation of objects with different values without impacting any of the Spring source files. For example, using the following file you can pass any value for "message" variable and you can print different values of message without impacting HelloWorld.java and MainApp.java files. Let us see how it works −

<?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>

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assigns them a unique ID as defined in **<bean>** tag. You can use **<property>** tag to pass the values of different variables used at the time of object creation.

## Step 5 - Running the Program

Once you are done with creating the source and beans configuration files, you are ready for this step, which is compiling and running your program. To do this, keep MainApp.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **MainApp**application. If everything is fine with your application, this will print the following message in Eclipse IDE's console −

Your Message : Hello World!

# IoC Containers

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.

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.

# Bean Definition

The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

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

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

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

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

## Spring Configuration Metadata

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

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

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

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

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

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

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

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

<!-- A simple bean definition -->

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

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

</bean>

<!-- A bean definition with lazy init set on -->

<bean id = "..." class = "..." lazy-init = "true">

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

</bean>

<!-- 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. |

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

## The singleton scope

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

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

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

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

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

</bean>

### Example

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

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

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

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

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

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

objA.getMessage();

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

objB.getMessage();

}

}

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

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

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

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

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

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

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

</bean>

</beans>

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

Your Message : I'm object A

Your Message : I'm object A

## The prototype scope

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

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

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

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

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

</bean>

### Example

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

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

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

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

}

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

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

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

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

objA.getMessage();

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

objB.getMessage();

}

}

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

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

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

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

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

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

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

</bean>

</beans>

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

Your Message : I'm object A

Your Message : null

# Bean Life Cycle

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

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

To define setup and teardown for a bean, we simply declare the <bean> with **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, destroymethod specifies a method that is called just before a bean is removed from the container.

## Initialization callbacks

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

void afterPropertiesSet() throws Exception;

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

public class ExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}

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

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

Following is the class definition −

public class ExampleBean {

public void init() {

// do some initialization work

}

}

## Destruction callbacks

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

void destroy() throws Exception;

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

public class ExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work

}

}

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

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

Following is the class definition −

public class ExampleBean {

public void destroy() {

// do some destruction work

}

}

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

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

### Example

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

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

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

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

public void init(){

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

}

public void destroy() {

System.out.println("Bean will destroy now.");

}

}

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

package com.tutorialspoint;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

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

obj.getMessage();

context.registerShutdownHook();

}

}

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

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

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

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

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

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

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" init-method = "init"

destroy-method = "destroy">

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

</bean>

</beans>

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

Bean is going through init.

Your Message : Hello World!

Bean will destroy now.

## Default initialization and destroy methods

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

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

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

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

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

default-init-method = "init"

default-destroy-method = "destroy">

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

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

</bean>

</beans>

# 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

The following examples show how to write, register, and use BeanPostProcessors in the context of an ApplicationContext.

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

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

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

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

public void init(){

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

}

public void destroy(){

System.out.println("Bean will destroy now.");

}

}

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

## Example

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

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

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>

# 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.

Thus, DI exists in two major variants and the following two sub-chapters will cover both of them with examples −

|  |  |
| --- | --- |
| **Sr.No.** | **Dependency Injection Type & Description** |
| 1 | [**Constructor-based dependency injection**](https://www.tutorialspoint.com/spring/constructor_based_dependency_injection.htm)  Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class. |
| 2 | [**Setter-based dependency injection**](https://www.tutorialspoint.com/spring/setter_based_dependency_injection.htm)  Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your 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. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean and it is shown below.

<?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>

## Example

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

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

Here is the content of **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

ou 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.

Now what if you want to pass plural values like Java Collection types such as List, Set, Map, and Properties. To handle the situation, 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

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

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

Here is the content of **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)

# Spring - Beans Auto-Wiring

You have learnt how to declare beans using the <bean> element and inject <bean> using <constructor-arg> and <property> elements in XML configuration file.

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.

## 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. |

# Java Based Configuration

## @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

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

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*. |
| 4 | Create Java classes *HelloWorldConfig*, *HelloWorld* and *MainApp* under the *com.tutorialspoint* package. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **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

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

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Because you are using Java-based annotations, so you also need to add *CGLIB.jar* from your Java installation directory and *ASM.jar* library which can be downloaded from *asm.ow2.org*. |
| 4 | Create Java classes *TextEditorConfig*, *TextEditor*, *SpellChecker* and *MainApp* under the *com.tutorialspoint* package. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **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 a() {

return new A();

}

}

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. The 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.

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

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

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

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

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

}

}

Following is the content of the **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

If you like, you can publish your own custom events and later you can capture the same to take any action against those custom events. If you are interested in writing your own custom events, you can check [Custom Events in Spring.](https://www.tutorialspoint.com/spring/custom_events_in_spring.htm)

# 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 Framework

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. 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. |