**What is the Spring Container?**

The Spring container is responsible for instantiating, configuring, and assembling the Spring beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich inter-dependencies between those objects.

**The responsibilities of IOC container are:**

* Instantiating the bean
* Wiring the beans together
* Configuring the beans
* Managing the bean’s entire life-cycle

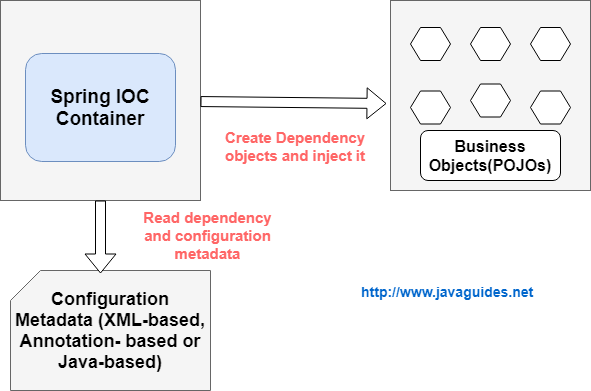
The *org.springframework.beans* and *org.springframework.context* packages are the basis for Spring Framework’s IoC container. Spring framework provides two distinct types of containers.

1. *BeanFactory* container
2. *ApplicationContext* container

*BeanFactory* is the root interface of Spring IOC container. *ApplicationContext* is the child interface of *BeanFactory* interface that provides Spring AOP features, i18n etc.

One main difference between *BeanFactory* and *ApplicationContext* is that *BeanFactory* only instantiates bean when we call *getBean()* method while *ApplicationContext* instantiates singleton bean when the container is started, It doesn't wait for *getBean()* method to be called.  
  
Check out a difference between BeanFactory vs ApplicationContext in-detail at [**BeanFactory vs ApplicationContext in Spring?**](https://www.javaguides.net/2019/01/beanfactory-vs-applicationcontext-in-spring.html)

The following diagram shows a high-level view of how Spring works. Your application classes are combined with configuration metadata so that, after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.

**[](https://3.bp.blogspot.com/-CsI_TxPJk_Y/W7cJZnb8z0I/AAAAAAAAEGw/oxiVbiAVSj8hrk06ynWoLqfcSolTTf3GgCLcBGAs/s1600/spring-ioc-container.png)**

**What is Configuration Metadata?**

From the above diagram, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

Three ways we can supply Configuration Metadata to Spring IoC container

1. [**XML-based configuration**](https://www.javaguides.net/2018/10/spring-ioc-container-xml-config-example.html)
2. [**Annotation-based configuration**](https://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html)
3. [**Java-based configuration**](https://www.javaguides.net/2018/06/spring-java-based-configuration-basics.html)

**How to Create a Spring Container?**

Spring provides many ApplicationContext interface implementations that we use are;

1. *AnnotationConfigApplicationContext*: If we are using Spring in standalone Java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
2. *ClassPathXmlApplicationContext*: If we have spring bean configuration XML file in a standalone application, then we can use this class to load the file and get the container object.
3. *FileSystemXmlApplicationContext*: This is similar to *ClassPathXmlApplicationContext* except that the XML configuration file can be loaded from anywhere in the file system.

*AnnotationConfigWebApplicationContext* and *XmlWebApplicationContext* for web applications.

Let's write a code to create Spring container:

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

Note that we are supplying configuration metadata via applicationContext.xml file(XML-based configuration).

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

Note that we are supplying configuration metadata via AppConfig.class file.

The most used API that implements the BeanFactory is the XmlBeanFactory.

XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("applicationContext.xml"));

Next, how to Retrieve bean from spring container?

**How to Retrieve Bean from Spring Container?**

Both *BeanFactory* and *ApplicationContext* interface provides *getBean()* method to retrieve bean from spring container.

**ApplicationContext getBean() Example:**

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

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

**BeanFactory getBean() Example:**

XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("beans.xml"));

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

### Spring IOC Container XML Config Example

In a previous article, we have discussed  [**What is Spring IOC Container and how it work**](http://www.javaguides.net/2018/10/spring-ioc-container-overview.html)**s**, Now in this article, we will discuss a simple example to demonstrate Spring IOC Container with XML-based configuration metadata?

**The Spring IOC container is responsible for instantiating, configuring, and assembling the Spring beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.**

Three ways we can supply Configuration Metadata to Spring IoC container

1. [**XML-based configuration**](http://www.javaguides.net/2018/10/spring-ioc-container-xml-config-example.html)
2. [**Annotation-based configuration**](http://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html)
3. [**Java-based configuration**](http://www.javaguides.net/2018/06/spring-java-based-configuration-basics.html)

In this example, we will supply XML-based configuration metadata to Spring IoC container.

## Spring IOC Container XML Config Example

### Spring Application Development Steps

Follow these three steps to develop a spring application:

1. Create a simple Maven Project
2. Add Maven Dependencies
3. Configure *HellowWorld* Spring Beans
4. Create a Spring Container
5. Retrieve Beans from Spring Container

### Tools and technologies used

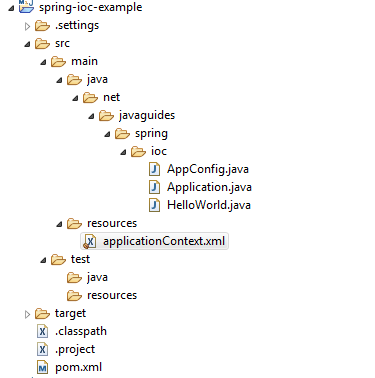
* Spring Framework - 5.1.0.RELEASE
* JDK - 8 or later
* Maven - 3.2+
* IDE - Eclipse Mars/STS

## 1. Create a simple Maven Project

Create a simple maven project using your favorite IDE and refer below diagram for packaging structure. If you are new to maven then read this article [How to Create a Simple Maven Project](http://www.javaguides.net/2018/06/Guide-to-Create-a-Simple-Maven-Project.html).

### Project Structure

Below diagram shows a project structure for your reference -

[](https://1.bp.blogspot.com/-6lZKm08bwAQ/W7cLFY8iRXI/AAAAAAAAEG8/ACfZD5l-c7cR5zVmYYTAyByW4CSi4jFJgCLcBGAs/s1600/ioc-xml-project-structure.PNG)

## 2. Add Maven Dependencies

<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>net.javaguides.spring</groupId>

<artifactId>spring-ioc-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.0.RELEASE</version>

</dependency>

</dependencies>

<build>

<sourceDirectory>src/main/java</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

## 3. Configure HelloWorld Spring Beans

### What Is a Spring Bean?

This is a very simple question that is often overcomplicated. Usually, Spring beans are Java objects that are managed by the Spring container.

Here is a HelloWorld Spring bean:

package net.javaguides.spring.ioc;

public class HelloWorld {

private String message;

public void setMessage(String message) {

this.message = message;

}

public void getMessage() {

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

}

}

### Configuration Metadata - Configure HelloWorld Spring 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">

<bean id="helloWorld" class="net.javaguides.spring.ioc.HelloWorld">

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

</bean>

</beans>

## 4. Create a Spring Container

If we have spring bean configuration XML file in a standalone application, then we can use ClassPathXmlApplicationContext class to load the file and get the container object.

package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

public static void main(String[] args) {

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

}

}

## 5. Retrieve Beans from Spring Container

*ApplicationContext* interface provides *getBean()* method to retrieve bean from spring container.

package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

public static void main(String[] args) {

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

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

obj.getMessage();

}

}

### Output

My Message : Hello World!

### Spring IOC Container Java Config Example

In a previous article, we have discussed  [**What is Spring IOC Container and how it works**](https://www.javaguides.net/2018/10/spring-ioc-container-overview.html), Now in this article, we will discuss a simple example to demonstrate Spring IOC Container with Java-based configuration metadata?

We will use the latest Spring release - Spring 5.1.0.RELEASE

**The Spring IOC container is responsible for instantiating, configuring, and assembling the Spring beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.**

Three ways we can supply Configuration Metadata to Spring IoC container

1. XML-based configuration
2. [**Annotation-based configuration**](https://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html)
3. [**Java-based configuration**](https://www.javaguides.net/2018/06/spring-java-based-configuration-basics.html)

In this example, we will supply Java-based configuration metadata to Spring IoC container.

## Spring IOC Container Java Config Example

### Spring Application Development Steps

Follow these five steps to develop a spring application:

1. Create a simple Maven Project
2. Add Maven Dependencies
3. Configure *HellowWorld* Spring Beans
4. Create a Spring Container
5. Retrieve Beans from Spring Container

### Tools and technologies used

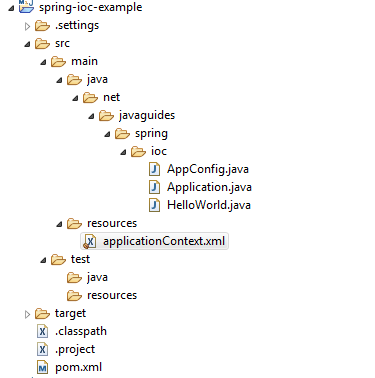
* Spring Framework - 5.1.0.RELEASE
* JDK - 8 or later
* Maven - 3.2+
* IDE - Eclipse Mars/STS

## 1. Create a simple Maven Project

Create a simple maven project using your favorite IDE and refer below diagram for packaging structure. If you are new to maven then read this article [**How to Create a Simple Maven Project**](https://www.javaguides.net/2018/06/Guide-to-Create-a-Simple-Maven-Project.html).

### Project Structure

Below diagram shows a project structure for your reference -

**[](https://1.bp.blogspot.com/-6lZKm08bwAQ/W7cLFY8iRXI/AAAAAAAAEG8/ACfZD5l-c7cR5zVmYYTAyByW4CSi4jFJgCLcBGAs/s1600/ioc-xml-project-structure.PNG)**

## 2. Add Maven Dependencies

<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>net.javaguides.spring</groupId>

<artifactId>spring-ioc-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.0.RELEASE</version>

</dependency>

</dependencies>

<build>

<sourceDirectory>src/main/java</sourceDirectory>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

## 3. Configure HelloWorld Spring Beans

### What is a Spring Bean?

This is a very simple question that is often overcomplicated. Usually, Spring beans are Java objects that are managed by the Spring container.

Here is a HelloWorld Spring bean:

package net.javaguides.spring.ioc;

public class HelloWorld {

private String message;

public void setMessage(String message) {

this.message = message;

}

public void getMessage() {

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

}

}

### Configuration Metadata - Configure HelloWorld Spring Beans

package net.javaguides.spring.ioc;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

@Bean

public HelloWorld helloWorld() {

HelloWorld helloWorld = new HelloWorld();

helloWorld.setMessage("Hello World!");

return helloWorld;

}

}

Spring [**@Configuration**](https://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) annotation is part of the spring core framework. Spring Configuration annotation indicates that the class has [**@Bean**](https://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html) definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.

## 4. Create a Spring Container

If we have spring bean configuration XML file in a standalone application, then we can use *ClassPathXmlApplicationContext* class to load the file and get the container object.

package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

context.close();

}

}

## 5. Retrieve Beans from Spring Container

*ApplicationContext* interface provides *getBean()* method to retrieve bean from spring container.

package net.javaguides.spring.ioc;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Application {

public static void main(String[] args) {

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

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

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

obj.getMessage();

context.close();

}

}

### Output

My Message : Hello World!

### Guide to Dependency Injection in Spring

**Dependency injection (DI)**is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using a direct construction of classes, or the [Service Locator pattern](http://ramesh-java-design-patterns.blogspot.com/2018/02/service-locator-pattern.html).

The Spring Framework [Inversion of Control (IoC)](http://www.javaguides.net/2018/10/spring-ioc-container-overview.html) component is the nucleus of the framework. It uses dependency injection to assemble Spring-provided (also called infrastructure components) and development-provided components in order to rapidly wrap up an application.

|  |
| --- |
| [https://1.bp.blogspot.com/-aMDbfdbSeCA/W0HXn13XpxI/AAAAAAAACvU/53qt__R7uSM-QLWbVmaP96Pa6m1r8HbWwCEwYBhgL/s400/spring-ioc.png](https://1.bp.blogspot.com/-aMDbfdbSeCA/W0HXn13XpxI/AAAAAAAACvU/53qt__R7uSM-QLWbVmaP96Pa6m1r8HbWwCEwYBhgL/s1600/spring-ioc.png) |
| *Figure: Spring IoC purpose* |

### Advantages of Dependency Injection

* **Decoupling:**Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies.
* **Easier to test:** As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

**DI exists in two major variants :**

* Constructor-based dependency injection
* Setter-based dependency injection.

## Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.  
  
In the below diagram, the highlighted part shows the Constructor-based dependency injection.

[](https://1.bp.blogspot.com/-DJN2-6F43DI/W7TReM6oMWI/AAAAAAAAEEw/-SXUnp6TMy4F7AWnCW2saHIzsKRGctQ2ACLcBGAs/s1600/constructor-based-injection.PNG)

### Constructor-based dependency injection example

Let's see the complete example to demonstrate Constructor-based dependency injection.  
As we know, spring java and annotation based configurations are quite easy to use in spring based applications so I prefer to use a mix of java and annotation based spring configurations.

Let's create spring configuration file using java class *AppConfiguration* which is annotated with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) annotation. This is equivalent to spring XML configuration file without beans definition.

package com.javadevsguide.springframework.di.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.javadevsguide.springframework.di")

public class AppConfiguration {

}

Now create a *MessageService* interface and provide more than two implementations for it.

public interface MessageService {

public void sendMsg(String message);

}

Let's implement the *MessageService* interface. There are many ways to send a message like through email, SMS, twitter etc.

@Service("EmailService")

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("SMSService")

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("TwitterService")

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

Note that there are multiple implementations for *MessageService* interface so to avoid ambiguity, let's use [@Qualifier](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation.  
It's time to demonstrate the usage of Constructor-based dependency injection. To avoid decoupling always use interfaces or abstract base classes as an instance variable and constructor arguments. In this example, we have used *MessageService* interface.

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

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

import org.springframework.stereotype.Component;

import com.javadevsguide.springframework.di.service.MessageService;

@Component

public class ConstructorBasedInjection {

private MessageService messageService;

@Autowired

public ConstructorBasedInjection(@Qualifier("TwitterService")

MessageService messageService) {

super();

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

It's time to test the usage of Constructor-based dependency injection. Let's create IOC container object that is an *ApplicationContext* object and get the beans from it.

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.javadevsguide.springframework.di.config.AppConfiguration;

import com.javadevsguide.springframework.di.field.FieldBasedInjection;

public class TestApplication {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

FieldBasedInjection fieldBasedInjection = applicationContext.getBean(FieldBasedInjection.class);

fieldBasedInjection.processMsg("twitter message sending ");

}

}

Note that we have used *AppConfiguration* class annotated with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) for configurations.

## Setter-based dependency injection

**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.

The following example shows a class that can only be dependency-injected using pure **setter injection**. This class is conventional Java.  
In the below diagram, the highlighted part shows the **setter-based dependency injection.**

[](https://1.bp.blogspot.com/-PcTS3vVRJeY/W7TRuGF6kvI/AAAAAAAAEE8/u01k0wXMMqA24w32pJzldRJCmp5keZapgCLcBGAs/s1600/setter-based-injection.PNG)

### Setter-based dependency injection example

Let's see the complete example to demonstrate Setter-based dependency injection.  
As we know, spring java and annotation based configurations are quite easy to use in spring based applications so I prefer to use a mix of java and annotation based spring configurations.

In this example, we have used spring Java-based container configuration.

Let's create spring configuration file using java class AppConfiguration and annotated with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) annotation. This is equivalent to spring XML configuration file without beans definition.

package com.javadevsguide.springframework.di.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.javadevsguide.springframework.di")

public class AppConfiguration {

}

Now create *MessageService* interface and provide more than two implementations for it.

public interface MessageService {

public void sendMsg(String message);

}

Let's implement the *MessageService* interface. There are many ways to send a message like through email, SMS, twitter etc.

@Service("EmailService")

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("SMSService")

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("TwitterService")

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

Note that there are multiple implementations for *MessageService* interface so to avoid ambiguity, let's use [@Qualifier](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation.  
It's time to demonstrate the usage of Setter-based dependency injection. To avoid decoupling always use interfaces or abstract base classes as an instance variable and setter method arguments. In this example, we have used the MessageService interface.

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

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

import org.springframework.stereotype.Component;

import com.javadevsguide.springframework.di.service.MessageService;

@Component

public class SetterBasedInjection {

private MessageService messageService;

@Autowired

@Qualifier("TwitterService")

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

It's time to test the usage of Constructor-based dependency injection. Let's create IOC container object that is an *ApplicationContext* object and get the beans from it.

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.javadevsguide.springframework.di.config.AppConfiguration;

import com.javadevsguide.springframework.di.field.FieldBasedInjection;

public class TestApplication {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

SetterBasedInjection setterBasedInjection = applicationContext.getBean(SetterBasedInjection.class);

setterBasedInjection.processMsg("twitter message sending ");

}

}

### Spring Dependency Injection via Setter Example

**posted by**[**Ramesh Fadatare**](https://www.blogger.com/profile/14691512106162803120)**on**[**June 09, 2018**](https://www.javaguides.net/2018/06/spring-dependency-injection-via-setter.html)

In this article, we will learn how to use setter-based dependency injection in Spring applications.

In **setter dependency injection**, the [Spring IOC container](https://www.javaguides.net/2018/10/spring-ioc-container-overview.html) injects a component’s dependencies via JavaBean-style setter methods. A component’s setters expose the dependencies the IoC container can manage.

In the below diagram, the highlighted part of the code shows a setter-based dependency injection example.

[](https://2.bp.blogspot.com/-tSfODdT3xCw/W7TWb8GnrUI/AAAAAAAAEFM/gBGEatM0LhsVGeX6R9hMIadbCq0GTdw-gCLcBGAs/s1600/setter-based-injection.PNG)

# Spring Dependency Injection via Setter Example

Let's see the complete example to demonstrate the Setter-based dependency injection. In this example, we have used spring Java-based container configuration.

**You can download the source code of this article from my GitHub repository (link given at end of this article).**

## Maven POM Dependencies

Let's add a spring-context dependency to maven project like:

<project

xmlns="http://maven.apache.org/POM/4.0.0"

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

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javadevsguide.springframework</groupId>

<artifactId>spring-dependency-injection</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.8.1</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

## Spring Components

Let's create a few Spring beans or components as follows.

## MessageService

Let's create the *MessageService* interface and provide more than two implementations for it:

public interface MessageService {

public void sendMsg(String message);

}

## EmailService, SMSService, and TwitterService

Let's implement the *MessageService* interface. There are many ways to send a message like email, SMS, Twitter, etc.

@Service("EmailService")

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("SMSService")

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("TwitterService")

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

Note that there are multiple implementations for the *MessageService* interface so to avoid ambiguity, let's use [@Qualifier](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation.

## Setter-based dependency injection

It's time to demonstrate the usage of Setter-based dependency injection. To avoid decoupling always use interfaces or abstract base classes as an instance variable and setter method arguments.

In this example, we have used the *MessageService* interface.

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

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

import org.springframework.stereotype.Component;

import com.javadevsguide.springframework.di.service.MessageService;

@Component

public class SetterBasedInjection {

private MessageService messageService;

@Autowired

@Qualifier("TwitterService")

public void setMessageService(MessageService messageService) {

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

## Spring Configuration

Let's create a spring configuration file using java class AppConfiguration and annotated with [@Configuration](http://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html) annotation. This is equivalent to the spring XML configuration file without beans definition.

package com.javadevsguide.springframework.di.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.javadevsguide.springframework.di")

public class AppConfiguration {

}

## Testing

Let's create an IOC container object that is an ApplicationContext object and get the beans from it.

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.javadevsguide.springframework.di.config.AppConfiguration;

import com.javadevsguide.springframework.di.field.FieldBasedInjection;

public class TestApplication {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

FieldBasedInjection fieldBasedInjection = applicationContext.getBean(FieldBasedInjection.class);

fieldBasedInjection.processMsg("twitter message sending ");

}

}

Once you run above class will run spring application as stand-alone and print below output to console.

Output:

twitter message sending

### Spring Dependency Injection via Constructor Example

**posted by**[**Ramesh Fadatare**](https://www.blogger.com/profile/14691512106162803120)**on**[**June 09, 2018**](https://www.javaguides.net/2018/06/spring-dependency-injection-via.html)

In this article, we will learn how to use constructor-based dependency injection in Spring Applications.  
  
We can use either [Spring Annotation Based Container Configuration](http://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html) or [spring Java-based container configuration](http://www.javaguides.net/2018/06/spring-java-based-configuration-basics.html) or a mix of both to demonstrate this example

**Constructor-based DI** is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

In the below diagram, the highlighted code shows a Constructor-based dependency injection example.

[](https://1.bp.blogspot.com/-DJN2-6F43DI/W7TReM6oMWI/AAAAAAAAEEw/-SXUnp6TMy4F7AWnCW2saHIzsKRGctQ2ACLcBGAs/s1600/constructor-based-injection.PNG)

Let's create a spring configuration file using java class *AppConfiguration* which is annotated with [@Configuration annotation](https://www.javaguides.net/2018/09/spring-configuration-annotation-with-example.html). This is equivalent to spring XML configuration file without beans definition.

### AppConfiguration.java

package com.javadevsguide.springframework.di.config;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

@Configuration

@ComponentScan("com.javadevsguide.springframework.di")

public class AppConfiguration {

}

Now create a *MessageService* interface and provide more than two implementations for it.

### MessageSerivce.java

public interface MessageService {

public void sendMsg(String message);

}

Let's implement the *MessageService* interface. There are many ways to send a message like through email, SMS, twitter etc.

### Create EmailService, SMSService, and TwitterService Classes

@Service("EmailService")

public class EmailService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("SMSService")

public class SMSService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

@Service("TwitterService")

public class TwitterService implements MessageService{

public void sendMsg(String message) {

System.out.println(message);

}

}

Note that there are multiple implementations for *MessageService* interface so to avoid ambiguity, let's use [@Qualifier](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation.

It's time to demonstrate the usage of Constructor-based dependency injection. To avoid decoupling always use interfaces or abstract base classes as an instance variable and constructor arguments. In this example, we have used the MessageService interface.

### ConstructorBasedInjection.java

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

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

import org.springframework.stereotype.Component;

import com.javadevsguide.springframework.di.service.MessageService;

@Component

public class ConstructorBasedInjection {

private MessageService messageService;

@Autowired

public ConstructorBasedInjection(@Qualifier("TwitterService")

MessageService messageService) {

super();

this.messageService = messageService;

}

public void processMsg(String message) {

messageService.sendMsg(message);

}

}

It's time to test the usage of Constructor-based dependency injection. Let's create an IOC container object that is an *ApplicationContext* object and get the beans from it.

### Application.java

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.javadevsguide.springframework.di.config.AppConfiguration;

import com.javadevsguide.springframework.di.constructor.ConstructorBasedInjection;

public class Application {

public static void main(String[] args) {

ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfiguration.class);

ConstructorBasedInjection constructorBasedInjection = applicationContext.getBean(ConstructorBasedInjection.class);

constructorBasedInjection.processMsg("twitter message sending ");

}

}

### Guide to Spring Bean Scopes

In this guide, we will learn the different types of bean scopes in the Spring framework.

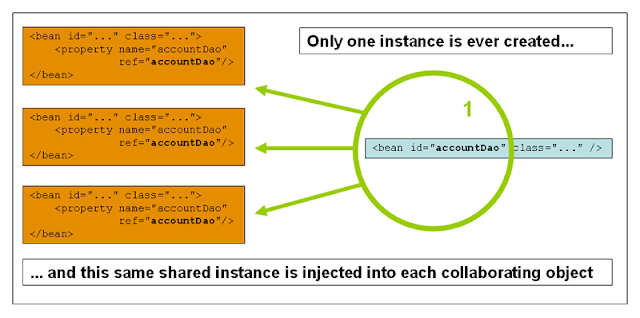
Spring Framework supports following bean scopes :

* **singleton:** (Default) Scopes a single bean definition to a single object instance per Spring IoC container.
* **prototype:** Scopes a single bean definition to any number of object instances.
* **request:** Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
* **session:** Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
* **application:** Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
* **WebSocket:** Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

## The singleton scope

Only one shared an instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a **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.

[](https://1.bp.blogspot.com/-bybbpmVy2XA/Wxv_Qi-2LaI/AAAAAAAACcg/Gd3RaXN9jDQqJS2MuMl33jKdlfx8VpqHACLcBGAs/s1600/spring-singleton-scope.png)

**Key points**

1. Spring’s concept of a singleton bean differs from the [Singleton Pattern](https://ramesh-java-design-patterns.blogspot.in/2017/12/singleton-design-pattern.html) as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader.
2. Spring Singleton beans are not threaded safe.
3. The singleton scope is the default scope in Spring.

To define a bean as a singleton in XML, we would write, for example:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

To define a bean as a **singleton** in **java based bean configuration**, we would write, for example:

@Configuration

public class AppConfiguration {

@Bean

@Scope("singleton") // default scope

public UserService userService(){

return new UserService();

}

}

## The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a **getBean()** method call on the container.

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

The following diagram illustrates the Spring prototype scope. A data access object (DAO) is not typically configured as a prototype, because a typical **DAO**does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram. The following example defines a bean as a prototype in XML:

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

```xml

To define a bean as a prototype in java based bean configuration, you would write, for example:

```java

@Configuration

public class AppConfiguration {

@Bean

@Scope("prototype")

public UserService userService(){

return new UserService();

}

}

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance.

### Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Method injection

### Request, session, application, and WebSocket scopes

The **request, session, application, and websocket**scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, an IllegalStateException will be thrown complaining about an unknown bean scope.

### Initial web configuration

To support the scoping of beans at the request, session, application, and websocket levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes, singleton and prototype.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet, then no special setup is necessary: DispatcherServlet already exposes all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s DispatcherServlet (for example, when using JSF or Struts), you need to register the org.springframework.web.context.request.RequestContextListener ServletRequestListener. For Servlet 3.0+, this can be done programmatically via the WebApplicationInitializer interface. Alternatively, or for older containers, add the following declaration to your web application’s web.xml file:

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

Alternatively, if there are issues with your listener setup, consider using Spring’s RequestContextFilter. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate.

<web-app>

...

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>requestContextFilter</filter-name>

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

</filter-mapping>

...

</web-app>

DispatcherServlet, RequestContextListener, and RequestContextFilter all do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

## Request scope

Consider the following XML configuration for a bean definition:

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the **loginAction** bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java Config, the @RequestScope annotation can be used to assign a component to the request scope.

@RequestScope

@Component

public class LoginAction {

// ...

}

## Session scope

Consider the following XML configuration for a bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

When using annotation-driven components or Java Config, the **@SessionScope** annotation can be used to assign a component to the session scope.

@SessionScope

@Component

public class UserPreferences {

// ...

}

## Application scope

Consider the following XML configuration for a bean definition:

<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>

The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application. That is, the appPreferences bean is scoped at the ServletContext level, stored as a regular ServletContext attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

When using annotation-driven components or Java Config, the **@ApplicationScope** annotation can be used to assign a component to the application scope.

@ApplicationScope

@Component

public class AppPreferences {

// ...

}

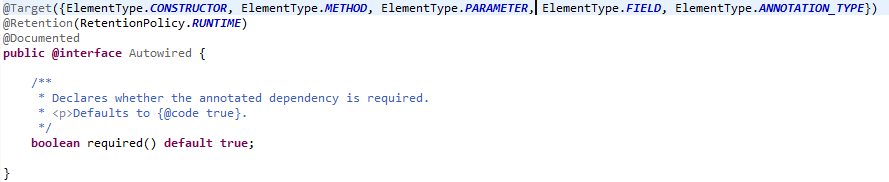
### Spring @Autowired Annotation with Example

**posted by**[**Ramesh Fadatare**](https://www.blogger.com/profile/14691512106162803120)**on**[**September 13, 2018**](https://www.javaguides.net/2018/09/spring-autowired-annotation-with-example.html)

In this article, we will discuss a very important Spring dependency injection annotation that is a *@Autowired* annotation.

We can use the *@Autowired* to mark a dependency which Spring is going to resolve and inject. We can use this annotation with a **constructor**, **setter**, or **field**injection.

The below diagram shows an internal implementation of *@Autowired*annotation:

[](https://4.bp.blogspot.com/-kgVR_wZXEpE/W5pBL_hqz-I/AAAAAAAADto/ykqZJNchoj4EWd7mG5eHRG4cXrdQ-vigQCLcBGAs/s1600/Autowired.PNG)

## Constructor Injection

@RestController

public class CustomerController {

private CustomerService customerService;

@Autowired

public CustomerController(CustomerService customerService) {

this.customerService = customerService;

}

}

## Setter Injection

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

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

@RestController

public class CustomerController {

private CustomerService customerService;

@Autowired

public void setCustomerService(CustomerService customerService) {

this.customerService = customerService;

}

}

## Field Injection

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

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

@RestController

public class CustomerController {

@Autowired

private CustomerService customerService;

}

## Apply @Autowired Annotation on Different level

*@Autowired* annotation can be applied on a different level or scenarios. Below are the few examples for the same.

**Example:** Apply the annotation to methods with arbitrary names and/or multiple arguments:

public class MovieRecommender {

private MovieCatalog movieCatalog;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired

public void prepare(MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

**Example:**Apply *@Autowired* to fields as well and even mix it with constructors:

public class MovieRecommender {

private final CustomerPreferenceDao customerPreferenceDao;

@Autowired

private MovieCatalog movieCatalog;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

**Example:**Apply *@Autowired* annotation to a field or method that expects an array of that type:

public class MovieRecommender {

@Autowired

private MovieCatalog[] movieCatalogs;

// ...

}

**Example:** Apply *@Autowired* annotation to the same applies for typed collections:

public class MovieRecommender {

private Set<MovieCatalog> movieCatalogs;

@Autowired

public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

// ...

}

You may express the non-required nature of a particular dependency through Java 8’s java.util.Optional:

public class SimpleMovieLister {

@Autowired

public void setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

We can also use *@Autowired* for interfaces that are well-known resolvable dependencies: *BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource*.

public class MovieRecommender {

@Autowired

private ApplicationContext context;

public MovieRecommender() {

}

// ...

}

## @Autowired Annotation Optional Elements

* boolean required - Declares whether the annotated dependency is required. Example:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired(required = false)

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

### Spring Annotation Based Container Configuration

**posted by**[**Ramesh Fadatare**](https://www.blogger.com/profile/14691512106162803120)**on**[**July 07, 2018**](https://www.javaguides.net/2018/07/spring-annotation-based-container-configuration.html)

## Overview

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.

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

## Why Spring Configuration with Annotations?

### XML configuration can be verbose

### Easy to configure your Spring beans with Annotations

### Annotations minimize the XML configuration

## Spring Annotation Based Configuration Basics

### How to enable annotation-based wiring?

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>

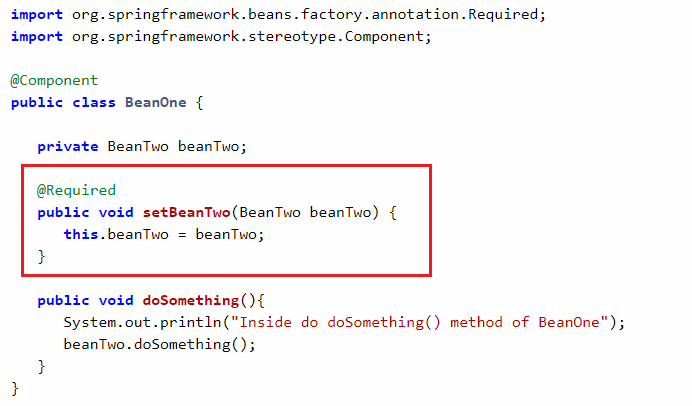
Once *<context:annotation-config />* is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

Let's discuss different annotations that Spring provides to configure annotation-based metadata configuration.

* **@Required**
* **@Autowired**
* **@Qualifier**
* **@Primary**
* **@Resource**
* **@PostConstruct and @PreDestroy (These are JSR-250 annotations)**

### @Required

The @Required annotation is method-level annotation and applied to the setter method of a bean.  
This annotation simply indicates that the setter method must be configured to be dependency-injected with a value at configuration time.

**[](https://2.bp.blogspot.com/-FZaiGMKbx6c/W7dA1ETUxbI/AAAAAAAAEHU/_z7ckknH30wHNut9_eEpB5GBr2YRxZ9SgCLcBGAs/s1600/required-annotation.PNG)**

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

### @Autowired

We can use the @Autowired to mark a dependency which Spring is going to resolve and inject. We can use this annotation with a constructor, setter, or field injection.

#### Constructor injection

class Car {

private Engine engine;

@Autowired

Car(Engine engine) {

this.engine = engine;

}

}

### Setter injection

class Car {

private Engine engine;

@Autowired

void setEngine(Engine engine) {

this.engine = engine;

}

}

### Field injection

class Car {

@Autowired

private Engine engine;

}

Read more about the importance of @Autowired annotation on [**Spring @Autowired Annotation with Example**](http://www.javaguides.net/2018/09/spring-autowired-annotation-with-example.html)

### @Primary Annotation

Use *@Primary* to give higher preference to a bean when there are multiple beans of the same type.

Let’s describe the problem in detail.

**Why is @Primary Needed?**

In some cases, we need to register more than one bean of the same type.

In this example we have *mySQLConnection()* and *oracleConnection()* beans of the *Connection* type:

@Configuration

public class Config {

@Bean

public Connection mySQLConnection() {

return new MySQLConnection();

}

@Bean

public Connection oracleConnection() {

return new OracleConnection();

}

}

Spring throws *NoUniqueBeanDefinitionException* if we try to run the application.

To access beans with the same type we usually use [**@Qualifier(“beanName”)**](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation.

We apply to at the injection point along with [**@Autowired**](http://www.javaguides.net/2018/09/spring-autowired-annotation-with-example.html). In our case, we select the beans at the configuration phase so [**@Qualifier**](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) can’t be applied here. We can learn more about [**@Qualifier**](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html) annotation by following the link.

To resolve this issue Spring offers the @Primary annotation. The following example shows how to use *@Primary*annotation in a spring application.

The *@Primary* annotation may be used on any class directly or indirectly annotated with *@Component* or on factory methods annotated with [**@Bean**](http://www.javaguides.net/2018/09/spring-bean-annotation-with-example.html). In this example, we will use *@Primary* annotation with *@Component*annotation.

Read more about the importance of *@Primary* annotation on [**Spring - @Primary Annotation Example**](http://www.javaguides.net/2018/10/spring-primary-annotation-example.html)

### @Qualifier

We use **@Qualifier annotation** to resolve ambiguous dependencies.

1. This annotation helps fine-tune annotation-based autowiring. There may be scenarios when we create more than one bean of the same type and want to wire only one of them with a property. This can be controlled using @Qualifier annotation along with the *@Autowired* annotation.
2. The *@Qualifier* is used to resolve ambiguous dependencies i.e, it helps @Autowired annotations to choose one of the dependency.

If there are multiple implementations for a single interface then we can use *@Qualifier* to choose required implementation at runtime.

Example :

@Autowired

@Qualifier("datasource")

private DataSource datasource;

@Autowired

@Qualifier("datasource1")

private DataSource datasource;

Read more about the importance of *@Qualifier* annotation on [**Spring @Qualifier Annotation Example**](http://www.javaguides.net/2018/06/spring-qualifier-annotation-example.html)

### @Resource

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

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Resource(name="myMovieFinder")

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

If no name is specified explicitly, the default name is derived from the field name or setter method.

### @PostConstruct and @PreDestroy

Spring supports JSR-250 lifecycle annotations.

Consider *DatabaseInitiaizer* bean, whose *init()* and *destroy()* methods are annotated with *@PostConstruct* and *@PreDestroy* annotations respectively.

### DatabaseInitiaizer.java

package net.javaguides.spring;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import org.springframework.stereotype.Component;

@Component

public class DatabaseInitiaizer {

private List < User > listOfUsers = new ArrayList < > ();

@PostConstruct

public void init() {

User user = new User(1, "User");

User user1 = new User(2, "Admin");

User user2 = new User(3, "SuperAdmin");

listOfUsers.add(user);

listOfUsers.add(user1);

listOfUsers.add(user2);

System.out.println("-----------List of users added in init() method ------------");

for (Iterator < User > iterator = listOfUsers.iterator(); iterator.hasNext();) {

User user3 = (User) iterator.next();

System.out.println(user3.toString());

}

// save to database

}

@PreDestroy

public void destroy() {

// Delete from database

listOfUsers.clear();

System.out.println("-----------After of users removed from List in destroy() method ------------");

for (Iterator < User > iterator = listOfUsers.iterator(); iterator.hasNext();) {

User user3 = (User) iterator.next();

System.out.println(user3.toString());

}

}

}

Refer complete example of *@PostConstruct* and *@PreDestroy* annotations on [**Spring @PostConstruct and @PreDestroy Example**](http://www.javaguides.net/2018/10/spring-postconstruct-and-predestroy-example.html)

### Are annotations better than XML for configuring Spring?

I think the answer is it depends like each approach has its pros and cons, and usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It’s worth pointing out that through its JavaConfig option, Spring allows annotations to be used in a non-invasive way, without touching the target components source code and that in terms of tooling, all configuration styles are supported by the Spring Tool Suite.

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

## Spring Web Model-View-Controller

* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

## Understanding the flow of Spring Web MVC

* As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
* The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
* The controller returns an object of ModelAndView.
* The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

## Advantages of Spring MVC Framework

Let's see some of the advantages of Spring MVC Framework:-

* **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
* **Light-weight** - It uses light-weight servlet container to develop and deploy your application.
* **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
* **Rapid development** - The Spring MVC facilitates fast and parallel development.
* **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
* **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
* **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

## Spring Web MVC Framework Example

Let's see the simple example of a Spring Web MVC framework. The steps are as follows:

* Load the spring jar files or add dependencies in the case of Maven
* Create the controller class
* Provide the entry of controller in the web.xml file
* Define the bean in the separate XML file
* Display the message in the JSP page
* Start the server and deploy the project

## Directory Structure of Spring MVC

## Directory Structure of Spring MVC using Maven

## Required Jar files or Maven Dependency

To run this example, you need to load:

* Spring Core jar files
* Spring Web jar files
* JSP + JSTL jar files (If you are using any another view technology then load the corresponding jar files).

**Download Link:** [Download all the jar files for spring including JSP and JSTL](https://static.javatpoint.com/src/sp/springjars.zip).

If you are using Maven, you don't need to add jar files. Now, you need to add maven dependency to the pom.xml file.

### 1. Provide project information and configuration in the pom.xml file.

**pom.xml**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**SpringMVC**</artifactId>**
6. **<packaging>**war**</packaging>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<name>**SpringMVC Maven Webapp**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<dependencies>**
11. **<dependency>**
12. **<groupId>**junit**</groupId>**
13. **<artifactId>**junit**</artifactId>**
14. **<version>**3.8.1**</version>**
15. **<scope>**test**</scope>**
16. **</dependency>**
18. <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
19. **<dependency>**
20. **<groupId>**org.springframework**</groupId>**
21. **<artifactId>**spring-webmvc**</artifactId>**
22. **<version>**5.1.1.RELEASE**</version>**
23. **</dependency>**
25. <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
26. **<dependency>**
27. **<groupId>**javax.servlet**</groupId>**
28. **<artifactId>**servlet-api**</artifactId>**
29. **<version>**3.0-alpha-1**</version>**
30. **</dependency>**
32. **</dependencies>**
33. **<build>**
34. **<finalName>**SpringMVC**</finalName>**
35. **</build>**
36. **</project>**

### 2. Create the controller class

To create the controller class, we are using two annotations @Controller and @RequestMapping.

The @Controller annotation marks this class as Controller.

The @Requestmapping annotation is used to map the class with the specified URL name.

**HelloController.java**

1. **package** com.javatpoint;
2. **import** org.springframework.stereotype.Controller;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. @Controller
5. **public** **class** HelloController {
6. @RequestMapping("/")
7. **public** String display()
8. {
9. **return** "index";
10. }
11. }

### 3. Provide the entry of controller in the web.xml file

In this xml file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the html file will be forwarded to the DispatcherServlet.

**web.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<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" id="WebApp\_ID" version="3.0"**>**
3. **<display-name>**SpringMVC**</display-name>**
4. **<servlet>**
5. **<servlet-name>**spring**</servlet-name>**
6. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
7. **<load-on-startup>**1**</load-on-startup>**
8. **</servlet>**
9. **<servlet-mapping>**
10. **<servlet-name>**spring**</servlet-name>**
11. **<url-pattern>**/**</url-pattern>**
12. **</servlet-mapping>**
13. **</web-app>**

### 4. Define the bean in the xml file

This is the important configuration file where we need to specify the View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

This xml file should be located inside the WEB-INF directory.

**spring-servlet.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:mvc="http://www.springframework.org/schema/mvc"
6. xsi:schemaLocation="
7. http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans.xsd
9. http://www.springframework.org/schema/context
10. http://www.springframework.org/schema/context/spring-context.xsd
11. http://www.springframework.org/schema/mvc
12. http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>**
14. <!-- Provide support for component scanning -->
15. **<context:component-scan** base-package="org.Pro" **/>**
17. <!--Provide support for conversion, formatting and validation -->
18. **<mvc:annotation-driven/>**
20. **</beans>**

### 5. Display the message in the JSP page

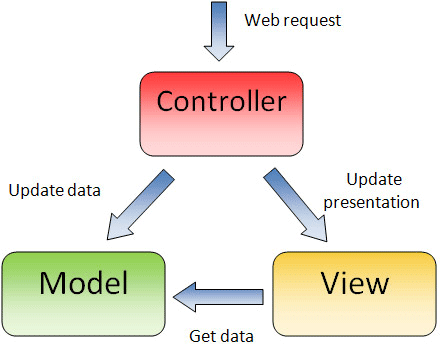
This is the simple JSP page, displaying the message returned by the Controller.

**index.jsp**

1. **<html>**
2. **<body>**
3. **<p>**Welcome to Spring MVC Tutorial**</p>**
4. **</body>**
5. **</html>**

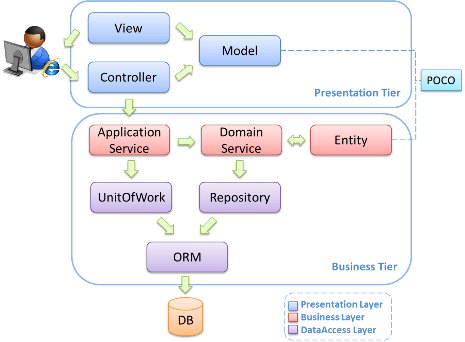
## What is MVC Framework?

[**Model-view-controller (MVC)**](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller) is a well known [**design pattern**](https://howtodoinjava.com/category/design-patterns/) for designing UI based applications. It mainly decouples business logic from UIs by separating the roles of **model, view, and controller** in an application. Usually, models are responsible for encapsulating application data for views to present. Views should only present this data, without including any business logic. And controllers are responsible for receiving requests from users and invoking back-end services (manager or dao) for business logic processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.



In a Spring MVC application, models usually consist of POJO objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with [Java Standard Tag Library (JSTL)](https://jstl.java.net/). Controller part is played by dispatcher servlet which we will learn about in this tutorial in more detail.

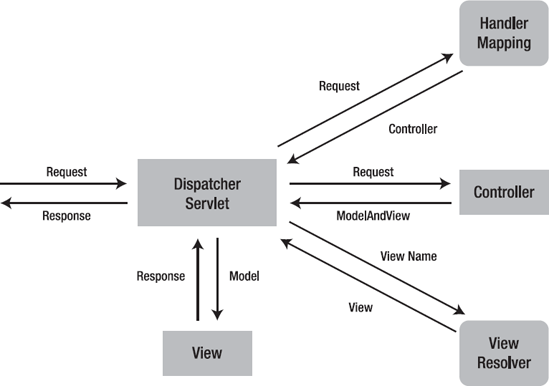
Some developers consider the service layer and DAO layers classes as part of model component in MVC. I have a different opinion on this. I do not consider service and DAO layers classes the part of MVC framework. Usually a web application is 3-tier architecture i.e. data-service-presentation. MVC is actually part of presentation layer.



## Dispatcher Servlet (Spring Controller)

In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the web.xml file). A Spring MVC controller—often referred to as a [Dispatcher Servlet](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) implements [front controller](https://en.wikipedia.org/wiki/Front_Controller_pattern) design pattern and every web request must go through it so that it can manage the entire request life cycle.

When a web request is sent to a Spring MVC application, dispatcher servlet first receives the request. Then it organizes the different components configured in Spring’s web application context (e.g. actual request handler controller and view resolvers) or annotations present in the controller itself, all needed to handle the request.

Spring dispatcher servlet

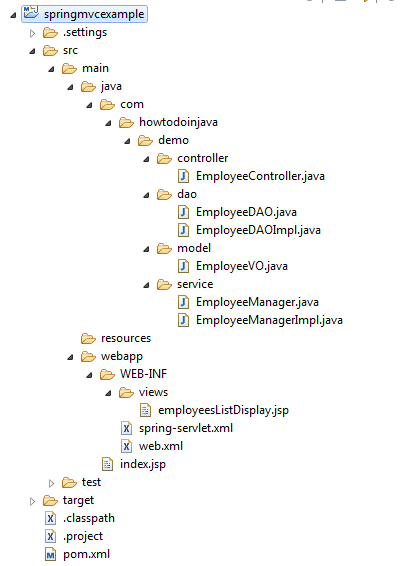
To define a controller class in Spring 3.0, a class has to be marked with the [@Controller](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Controller.html) annotation. When a @Controller annotated controller receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class’s methods are decorated with the [@RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) annotation, making them handler methods.

After a handler method has finished processing the request, it delegates control to a view, which is represented as handler method’s return value. To provide a flexible approach, a handler method’s return value doesn’t represent a view’s implementation but rather a logical view i.e. without any file extension. You can map these logical views to right implementation into applicationContext file so that you can easily change your view layer code without even touching request handler class code.

To resolve the correct file for a logical name is the responsibility of [view resolvers](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ViewResolver.html). Once the controller class has resolved a view name into a view implementation, per the view implementation’s design, it renders the objects.

## Spring MVC Hello World Example

In this application, I am creating most simple **employee management application** demo having only one feature i.e. list all available employees in system. Let’s note down the directory structure of this application.

Spring mvc hello world directory structure

Now let’s write all the files involved into this hello world application.

**pom.xml**

Below pom.xml file contains dependencies for spring mvc and taglibs support for writing jsp files.

|  |
| --- |
| <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/maven-v4_0_0.xsd>">      <modelVersion>4.0.0</modelVersion>      <groupId>com.howtodoinjava.demo</groupId>      <artifactId>springmvcexample</artifactId>      <packaging>war</packaging>      <version>1.0-SNAPSHOT</version>      <name>springmvcexample Maven Webapp</name>      <url>[http://maven.apache.org](http://maven.apache.org/)</url>      <dependencies>            <dependency>              <groupId>junit</groupId>              <artifactId>junit</artifactId>              <version>4.12</version>              <scope>test</scope>          </dependency>            <!-- Spring MVC support -->            <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-webmvc</artifactId>              <version>4.1.4.RELEASE</version>          </dependency>            <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-web</artifactId>              <version>4.1.4.RELEASE</version>          </dependency>            <!-- Tag libs support for view layer -->            <dependency>              <groupId>javax.servlet</groupId>              <artifactId>jstl</artifactId>              <version>1.2</version>              <scope>runtime</scope>          </dependency>            <dependency>              <groupId>taglibs</groupId>              <artifactId>standard</artifactId>              <version>1.1.2</version>              <scope>runtime</scope>          </dependency>        </dependencies>        <build>          <finalName>springmvcexample</finalName>      </build>  </project> |

**web.xml**

This minimum web.xml file declares one servlet (i.e. dispatcher servlet) to receive all kind of requests. Dispatcher servlet here acts as front controller.

|  |
| --- |
| <web-app id="WebApp\_ID" version="2.4"      xmlns="<http://java.sun.com/xml/ns/j2ee>"      xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"      xsi:schemaLocation="<http://java.sun.com/xml/ns/j2ee>  <http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd>">        <display-name>Spring Web MVC Hello World Application</display-name>        <servlet>          <servlet-name>spring</servlet-name>              <servlet-class>                  org.springframework.web.servlet.DispatcherServlet              </servlet-class>          <load-on-startup>1</load-on-startup>      </servlet>        <servlet-mapping>          <servlet-name>spring</servlet-name>          <url-pattern>/</url-pattern>      </servlet-mapping>    </web-app> |

**spring-servlet.xml (You can have applicationContext.xml as well)**

We are using annotated classes at request handler, service and dao layer so I have enabled annotation processing for all class files in base package “com.howtodoinjava.demo“.

|  |
| --- |
| <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:component-scan base-package="com.howtodoinjava.demo" />        <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />      <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />        <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">          <property name="prefix" value="/WEB-INF/views/" />          <property name="suffix" value=".jsp" />      </bean>    </beans> |

**EmployeeController.java**

Annotation @RequestMapping at class level and method level determine the URL at which method will be invoked.

|  |
| --- |
| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.ui.Model;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;    import com.howtodoinjava.demo.service.EmployeeManager;    @Controller  @RequestMapping("/employee-module")  public class EmployeeController  {      @Autowired      EmployeeManager manager;        @RequestMapping(value = "/getAllEmployees", method = RequestMethod.GET)      public String getAllEmployees(Model model)      {          model.addAttribute("employees", manager.getAllEmployees());          return "employeesListDisplay";      }  } |

**Read More :**[**How to use @Component, @Repository, @Service and @Controller Annotations?**](https://howtodoinjava.com/spring/spring-core/how-to-use-spring-component-repository-service-and-controller-annotations/)

**EmployeeVO.java**

This class act as model for MVC pattern.

|  |
| --- |
| package com.howtodoinjava.demo.model;    import java.io.Serializable;    public class EmployeeVO implements Serializable  {      private static final long serialVersionUID = 1L;        private Integer id;      private String firstName;      private String lastName;        //Setters and Getters        @Override      public String toString() {          return "EmployeeVO [id=" + id + ", firstName=" + firstName                  + ", lastName=" + lastName + "]";      }  } |

**EmployeeDAO.java**

The classes at third tier in 3-tier architecture. Responsible for interacting with underlying DB storage.

|  |
| --- |
| import java.util.List;    import com.howtodoinjava.demo.model.EmployeeVO;    public interface EmployeeDAO  {      public List<EmployeeVO> getAllEmployees();  } |

**EmployeeDAOImpl.java**

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;    import org.springframework.stereotype.Repository;    import com.howtodoinjava.demo.model.EmployeeVO;    @Repository  public class EmployeeDAOImpl implements EmployeeDAO {        public List<EmployeeVO> getAllEmployees()      {          List<EmployeeVO> employees = new ArrayList<EmployeeVO>();            EmployeeVO vo1 = new EmployeeVO();          vo1.setId(1);          vo1.setFirstName("Lokesh");          vo1.setLastName("Gupta");          employees.add(vo1);            EmployeeVO vo2 = new EmployeeVO();          vo2.setId(2);          vo2.setFirstName("Raj");          vo2.setLastName("Kishore");          employees.add(vo2);            return employees;      }  } |

**EmployeeManager.java**

The classes at second tier in 3-tier architecture. Responsible for interacting with DAO Layer.

|  |
| --- |
| import java.util.List;    import com.howtodoinjava.demo.model.EmployeeVO;    public interface EmployeeManager  {      public List<EmployeeVO> getAllEmployees();  } |

**EmployeeManagerImpl.java**

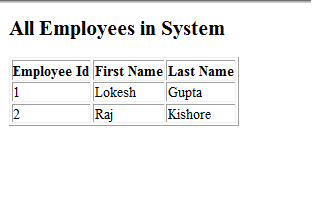
|  |
| --- |
| import java.util.List;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;    import com.howtodoinjava.demo.dao.EmployeeDAO;  import com.howtodoinjava.demo.model.EmployeeVO;    @Service  public class EmployeeManagerImpl implements EmployeeManager {        @Autowired      EmployeeDAO dao;        public List<EmployeeVO> getAllEmployees()      {          return dao.getAllEmployees();      }  } |

**employeesListDisplay.jsp**

This jsp is used to display all the employees in system. It iterates the collection of employees in loop, and print their details in a table. This fits into view layer of MVC pattern.

|  |
| --- |
| <%@ taglib prefix="c" uri="<http://java.sun.com/jsp/jstl/core>"%>  <%@ taglib prefix="fmt" uri="<http://java.sun.com/jsp/jstl/fmt>"%>    <html>  <head>      <title>Spring MVC Hello World</title>  </head>    <body>      <h2>All Employees in System</h2>        <table border="1">          <tr>              <th>Employee Id</th>              <th>First Name</th>              <th>Last Name</th>          </tr>          <c:forEach items="${employees}" var="employee">              <tr>                  <td>${employee.id}</td>                  <td>${employee.firstName}</td>                  <td>${employee.lastName}</td>              </tr>          </c:forEach>      </table>    </body>  </html> |

Now deploy the application in your application server (i am using tomcat 7). And hit the URL “**http://localhost:8080/springmvcexample/employee-module/getAllEmployees**“. You will see below screen if you have configured everything correctly.

Application front UI

**Spring with JDBC**

Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

Problems of JDBC API

The problems of JDBC API are as follows:

* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* We need to perform exception handling code on the database logic.
* We need to handle transaction.
* Repetition of all these codes from one to another database logic is a time consuming task.

Advantage of Spring JdbcTemplate

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

JdbcTemplate class

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of excepion classes defined in the **org.springframework.dao** package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

### Example of Spring JdbcTemplate

We are assuming that you have created the following table inside the Oracle10g database.

1. create table employee(
2. id number(10),
3. name varchar2(100),
4. salary number(10)
5. );

**Employee.java**

This class contains 3 properties with constructors and setter and getters.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
7. //no-arg and parameterized constructors
8. //getters and setters
9. }

**EmployeeDao.java**

It contains one property jdbcTemplate and three methods saveEmployee(), updateEmployee and deleteEmployee().

1. **package** com.javatpoint;
2. **import** org.springframework.jdbc.core.JdbcTemplate;
4. **public** **class** EmployeeDao {
5. **private** JdbcTemplate jdbcTemplate;
7. **public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {
8. **this**.jdbcTemplate = jdbcTemplate;
9. }
11. **public** **int** saveEmployee(Employee e){
12. String query="insert into employee values(
13. '"+e.getId()+"','"+e.getName()+"','"+e.getSalary()+"')";
14. **return** jdbcTemplate.update(query);
15. }
16. **public** **int** updateEmployee(Employee e){
17. String query="update employee set
18. name='"+e.getName()+"',salary='"+e.getSalary()+"' where id='"+e.getId()+"' ";
19. **return** jdbcTemplate.update(query);
20. }
21. **public** **int** deleteEmployee(Employee e){
22. String query="delete from employee where id='"+e.getId()+"' ";
23. **return** jdbcTemplate.update(query);
24. }
26. }

**applicationContext.xml**

The **DriverManagerDataSource** is used to contain the information about the database such as driver class name, connnection URL, username and password.

There are a property named **datasource** in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are passing it by the setter method but you can use constructor also.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="ds" **class**="org.springframework.jdbc.datasource.DriverManagerDataSource">
10. <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
11. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
12. <property name="username" value="system" />
13. <property name="password" value="oracle" />
14. </bean>
16. <bean id="jdbcTemplate" **class**="org.springframework.jdbc.core.JdbcTemplate">
17. <property name="dataSource" ref="ds"></property>
18. </bean>
20. <bean id="edao" **class**="com.javatpoint.EmployeeDao">
21. <property name="jdbcTemplate" ref="jdbcTemplate"></property>
22. </bean>
24. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the saveEmployee() method. You can also call updateEmployee() and deleteEmployee() method by uncommenting the code as well.

1. **package** com.javatpoint;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
5. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
8. ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");
10. EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");
11. **int** status=dao.saveEmployee(**new** Employee(102,"Amit",35000));
12. System.out.println(status);
14. /\*int status=dao.updateEmployee(new Employee(102,"Sonoo",15000));
15. System.out.println(status);
16. \*/
18. /\*Employee e=new Employee();
19. e.setId(102);
20. int status=dao.deleteEmployee(e);
21. System.out.println(status);\*/
23. }
25. }

# Hibernate and Spring Integration

We can simply integrate **hibernate application with spring application**.

In hibernate framework, we provide all the database information hibernate.cfg.xml file.

But if we are going to integrate the hibernate application with spring, we don't need to create the hibernate.cfg.xml file. We can provide all the information in the applicationContext.xml file.

### Advantage of Spring framework with hibernate

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

So **it saves a lot of code**.

**Understanding problem without using spring:**

Let's understand it by the code of hibernate given below:

1. //creating configuration
2. Configuration cfg=**new** Configuration();
3. cfg.configure("hibernate.cfg.xml");
5. //creating seession factory object
6. SessionFactory factory=cfg.buildSessionFactory();
8. //creating session object
9. Session session=factory.openSession();
11. //creating transaction object
12. Transaction t=session.beginTransaction();
14. Employee e1=**new** Employee(111,"arun",40000);
15. session.persist(e1);//persisting the object
17. t.commit();//transaction is commited
18. session.close();

As you can see in the code of sole hibernate, you have to follow so many steps.

**Solution by using HibernateTemplate class of Spring Framework:**

Now, you don't need to follow so many steps. You can simply write this:

1. Employee e1=**new** Employee(111,"arun",40000);
2. hibernateTemplate.save(e1);

## Steps

Let's see what are the simple steps for hibernate and spring integration:

1. **create table in the database** It is optional.
2. **create applicationContext.xml file** It contains information of DataSource, SessionFactory etc.
3. **create Employee.java file** It is the persistent class
4. **create employee.hbm.xml file** It is the mapping file.
5. **create EmployeeDao.java file** It is the dao class that uses HibernateTemplate.
6. **create InsertTest.java file** It calls methods of EmployeeDao class.

### Example of Hibernate and spring integration

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of spring and hibernate example.

**1) create the table in the database**

In this example, we are using the Oracle as the database, but you may use any database. Let's create the table in the oracle database

1. CREATE TABLE  "EMP558"
2. (    "ID" NUMBER(10,0) NOT NULL ENABLE,
3. "NAME" VARCHAR2(255 CHAR),
4. "SALARY" FLOAT(126),
5. PRIMARY KEY ("ID") ENABLE
6. )
7. /

**2) Employee.java**

It is a simple POJO class. Here it works as the persistent class for hibernate.

1. **package** com.javatpoint;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** salary;
8. //getters and setters
10. }

**3) employee.hbm.xml**

This mapping file contains all the information of the persistent class.

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.Employee" table="emp558">
8. <id name="id">
9. <generator **class**="assigned"></generator>
10. </id>
12. <property name="name"></property>
13. <property name="salary"></property>
14. </**class**>
16. </hibernate-mapping>

**4) EmployeeDao.java**

It is a java class that uses the **HibernateTemplate** class method to persist the object of Employee class.

1. **package** com.javatpoint;
2. **import** org.springframework.orm.hibernate3.HibernateTemplate;
3. **import** java.util.\*;
4. **public** **class** EmployeeDao {
5. HibernateTemplate template;
6. **public** **void** setTemplate(HibernateTemplate template) {
7. **this**.template = template;
8. }
9. //method to save employee
10. **public** **void** saveEmployee(Employee e){
11. template.save(e);
12. }
13. //method to update employee
14. **public** **void** updateEmployee(Employee e){
15. template.update(e);
16. }
17. //method to delete employee
18. **public** **void** deleteEmployee(Employee e){
19. template.delete(e);
20. }
21. //method to return one employee of given id
22. **public** Employee getById(**int** id){
23. Employee e=(Employee)template.get(Employee.**class**,id);
24. **return** e;
25. }
26. //method to return all employees
27. **public** List<Employee> getEmployees(){
28. List<Employee> list=**new** ArrayList<Employee>();
29. list=template.loadAll(Employee.**class**);
30. **return** list;
31. }
32. }

**5) applicationContext.xml**

In this file, we are providing all the informations of the database in the **BasicDataSource** object. This object is used in the **LocalSessionFactoryBean** class object, containing some other informations such as mappingResources and hibernateProperties. The object of **LocalSessionFactoryBean** class is used in the HibernateTemplate class. Let's see the code of applicationContext.xml file.

*File: applicationContext.xml*

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

10. <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource">
11. <property name="driverClassName"  value="oracle.jdbc.driver.OracleDriver"></property>
12. <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>
13. <property name="username" value="system"></property>
14. <property name="password" value="oracle"></property>
15. </bean>
17. <bean id="mysessionFactory"  **class**="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
18. <property name="dataSource" ref="dataSource"></property>
20. <property name="mappingResources">
21. <list>
22. <value>employee.hbm.xml</value>
23. </list>
24. </property>
26. <property name="hibernateProperties">
27. <props>
28. <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
29. <prop key="hibernate.hbm2ddl.auto">update</prop>
30. <prop key="hibernate.show\_sql">**true**</prop>
32. </props>
33. </property>
34. </bean>
36. <bean id="template" **class**="org.springframework.orm.hibernate3.HibernateTemplate">
37. <property name="sessionFactory" ref="mysessionFactory"></property>
38. </bean>
40. <bean id="d" **class**="com.javatpoint.EmployeeDao">
41. <property name="template" ref="template"></property>
42. </bean>

45. </beans>

**6) InsertTest.java**

This class uses the EmployeeDao class object and calls its saveEmployee method by passing the object of Employee class.

1. **package** com.javatpoint;
3. **import** org.springframework.beans.factory.BeanFactory;
4. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
5. **import** org.springframework.core.io.ClassPathResource;
6. **import** org.springframework.core.io.Resource;
8. **public** **class** InsertTest {
9. **public** **static** **void** main(String[] args) {
11. Resource r=**new** ClassPathResource("applicationContext.xml");
12. BeanFactory factory=**new** XmlBeanFactory(r);
14. EmployeeDao dao=(EmployeeDao)factory.getBean("d");
16. Employee e=**new** Employee();
17. e.setId(114);
18. e.setName("varun");
19. e.setSalary(50000);
21. dao.saveEmployee(e);
23. }
24. }