**Spring Framework**

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.

Spring is light weight framework because of its POJO (Plain Old Java Object) model

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc.

The first version was written by [Rod Johnson](https://en.wikipedia.org/wiki/Rod_Johnson_%28programmer%29), The framework was first released under the [Apache 2.0 license](https://en.wikipedia.org/wiki/Apache_License) in June 2003.

Spring Framework 4.2.0 was released on 31 July 2015 and was immediately upgraded to version 4.2.1, which was released on 01 Sept 2015. It is *"compatible with Java 6, 7 and 8, with a focus on core refinements and modern web capabilities”.*

Spring Framework 4.3 has been released on 10 June 2016

**Features of Spring:**

1. **Complete and a modular framework**

spring framework can be used for all layer implementations for a real time application or spring can be used for the development of particular layer of a real time application unlike struts [ only for front end related ] and hibernate [ only for database related ], but with spring we can develop all layers.

1. **Spring is non-invasive**:

**invasive** means involving other objects and classes.

it doesn’t force a programmer to extend or implement their class from any predefined class or interface given by Spring API.

in struts we used to extend Action Class, that’s why struts is said to be invasive.

in Hadoop Frame work we used to extend Mapper and Reducer Classes, that’s why Hadoop Frame is said to be invasive.

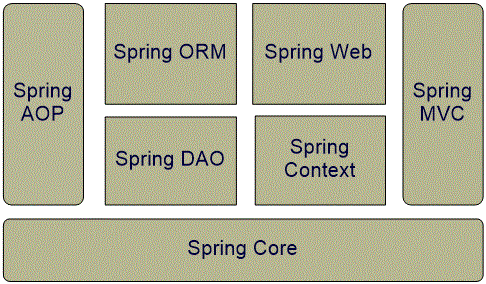
1. **Spring is light weight,** because of its POJO model. Spring Framework made J2EE application development little easier, by introducing POJO model. Spring is vast framework so spring people divide the whole spring in to different modules, they are designed in such a way that no module is dependent to other module , except Spring core module, so according to your requirement you can use a particular module, you no need to use whole total framework.
2. **End to end Development :** Spring supports all aspects of application development, Business aspects, persistence aspects, etc, so we can develop a complete application using spring.
3. **Spring supports All types of application development:** We can develop any type of applications using spring, eg: Core java, web Application, Distributed application, Enterprise application.
4. **Spring is versatile:** We can integrate any technologies with spring , so we can say spring is versatile.
5. **Spring supports dependency injection:** The dependency between classes are managed easily by the spring

## Spring Framework Modules:

## The Spring framework consists of seven well-defined modules

* Spring Core Module
* Spring Context [ J2EE ]
* Spring DAO Module [ Spring JDBC ]
* Spring ORM module
* Spring AOP [ Aspect Oriented Programming ]
* Spring WEB Module
* Spring MVC Module

The Spring modules are built on top of the core container, which defines how beans are created, configured, and managed, as shown in following figure.



## 1) core container :

## The Spring Core container contains

1. core,
2. beans,
3. context and
4. SpEL, Spring Expression Language (EL) modules.

The Core package is the most fundamental part of the framework and provides the IOC and Dependency Injection features.

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

## 2) Spring context module (J2EE) :

The Spring context is a configuration file that provides context information to the Spring framework.

The Spring context includes enterprise services such as e-mail, JNDI, EJB, internalization, validation, scheduling and applications life cycle events.

## 3. Spring AOP module :

**AOP (Aspect-Oriented Programming)**

**AOP is to cross cut code with main code.**

It’s target is to increase modularity, by permitting the separation of secondary code that cuts across the main code. The main code includes the actual business logic and the secondary code makes the business logic more robust ([**powerful**](https://www.google.co.in/search?client=firefox-b&biw=1138&bih=509&q=define+powerful&forcedict=powerful&sa=X&ved=0ahUKEwiZrvHAtJ_SAhXHMY8KHRF6AxAQ_SoIIDAA) ) like including the code that should be executed when a business logic fails. In AOP, this extra code is said to cross cut the actual code.

**What is concern in AOP?**

Aspect-oriented programming breaks the whole logic into discrete (separate) parts known as **concerns** (like marketing, finance, production, planning etc). Every programming style supports these concerns by implementing them in the code thorough functions, classes, methods and modules etc. Another style of concerns exist that come across these concerns (other way which cuts across) called **cross cutting concerns** like logging, transaction management, security etc.

The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect.

Whereas DI ( Dependency Injection ) helps you decouple your application objects from each other, AOP helps you decouple **cross-cutting concerns** from the objects that they affect.

## 4. Spring DAO module :

The Spring DAO module provides a JDBC-abstraction layer that reduces the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

## 5. Spring ORM module:

Spring provides integration with ORM mapping tools like Hibernate. Spring transaction management supports each of these ORM frameworks as well as JDBC.

## 6. Spring Web module :

The Web context module provides basic web-oriented integration features builds on top of the application context module, providing contexts for Web-based applications. As a result, the Spring framework supports integration with Jakarta Struts. The Web module also eases the tasks of handling multi-part requests and binding request parameters to domain objects.

## 7. Spring MVC framework module:

Spring's MVC package provides a Model-View-Controller (MVC) implementation for web-applications.

## Spring - Environment Setup

## Step 1 - Setup Java Development Kit (JDK)

## You can download the latest version of SDK from Oracle's Java site − [Java SE Downloads](https://www.oracle.com/technetwork/java/javase/downloads/index.html)

## <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

jdk-8u121-windows-x64

rt.click on the above software and run as administrator.

Click Next and finish.

Computer, select Properties → Advanced → Environment Variables.

System variables:

New :

Variable Name : JAVA\_HOME

Value : C:\Program Files\Java\jdk1.8.0\_121

Select PATH variable and EDIT

Name : PATH

Value : go to end of value and place ; add following

;%JAVA\_HOME%\bin

Eg:

C:\ProgramData\Oracle\Java\javapath;%PATH%;C:\app\nagaswarao\product\11.2.0\dbhome\_1\bin;%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;%SYSTEMROOT%\System32\WindowsPowerShell\v1.0;%JAVA\_HOME%\bin;

## Step 2 - Setup Eclipse IDE

Down load neon eclipse 2.0

eclipse-jee-neon-2-win32-x86\_64

extract into c: drive it generates eclipse folder.

Create shortcut on desk top then start.

**Create sample java project and test it.**

**Step 3 - Install Apache Common Logging API**

You can download the latest version of Apache Commons Logging API from [https://commons.apache.org/logging/](https://commons.apache.org/logging/download_logging.cgi)

**Apache Commons Logging 1.2**

**Binaries**

**Add these libraries to java project build path.**

**Step 4 - Setup Spring Framework Libraries**

**Down load following jars**

**Add these jars to java project build path.**

**The following using with Maven:**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-beans</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-core</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-orm</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-web</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-webmvc</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context-support</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-tx</artifactId>**

**<version>4.3.3.RELEASE</version>**

**</dependency>**

**Create a package and create following class.**

**package** com.nrit.mnrao.test;

**public** **class** Sample {

**private** String message;

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

**Create Test class for testing application as below**

**package** com.nrit.mnrao.test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("TestBean.xml");

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

String message = obj.getMessage();

System.***out***.println(message);

}

}

**Step 4 - Create Bean Configuration File (**TestBeans.xml) **under src folder ( root ).**

TestBeans.xml:

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

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

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

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

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

<bean id = *"helloWorld"* class = *"com.nrit.mnrao.test.Sample"*>

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

</bean>

</beans>

**Employee details:**

**package** com.nrit.mnrao.test;

**public** **class** Employee {

**private** **int** empId;

**private** String empName;

**private** String empDept;

**private** **double** empSalary;

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getEmpName() {

**return** empName;

}

**public** **void** setEmpName(String empName) {

**this**.empName = empName;

}

**public** String getEmpDept() {

**return** empDept;

}

**public** **void** setEmpDept(String empDept) {

**this**.empDept = empDept;

}

**public** **double** getEmpSalary() {

**return** empSalary;

}

**public** **void** setEmpSalary(**double** empSalary) {

**this**.empSalary = empSalary;

}

}

**===============================**

**package** com.nrit.mnrao.test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("TestBean.xml");

Employee employee = (Employee) context.getBean("emp1");

System.***out***.println(employee.getEmpId());

System.***out***.println(employee.getEmpName());

System.***out***.println(employee.getEmpDept());

System.***out***.println(employee.getEmpSalary());

}

}

TestBean.xml:

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

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

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

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

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

<bean id = *"emp1"* class = *"com.nrit.mnrao.test.Employee"*>

<property name = *"empId"* value = *"1001"*/>

<property name = *"empName"* value = *"MNRAO"*/>

<property name = *"empDept"* value = *"Dev"*/>

<property name = *"empSalary"* value = *"5000"*/>

</bean>

<bean id = *"emp2"* class = *"com.nrit.mnrao.test.Employee"*>

<property name = *"empId"* value = *"1002"*/>

<property name = *"empName"* value = *"MNRAO1"*/>

<property name = *"empDept"* value = *"Dev1"*/>

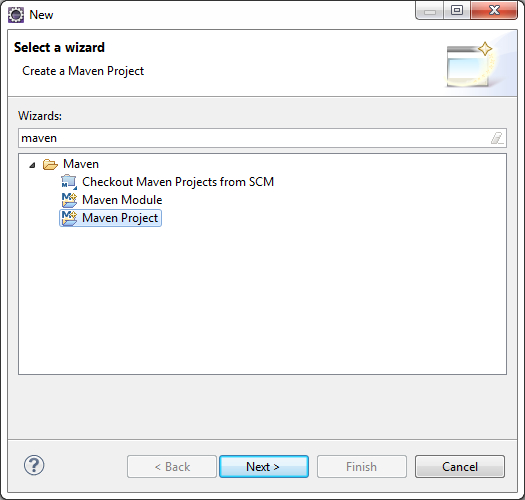
<property name = *"empSalary"* value = *"6000"*/>

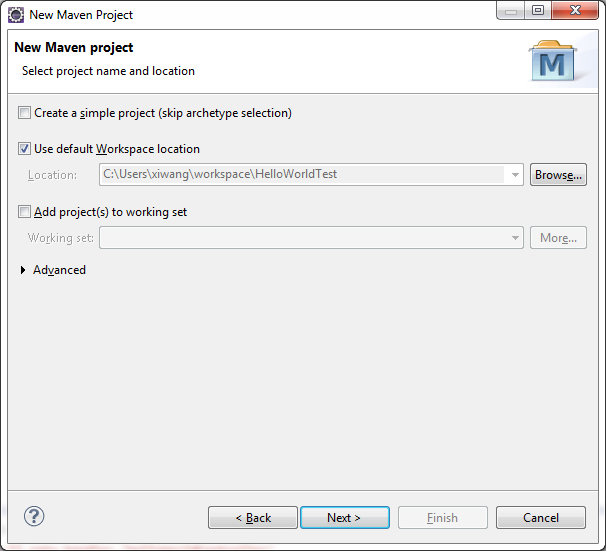
</bean>

</beans>

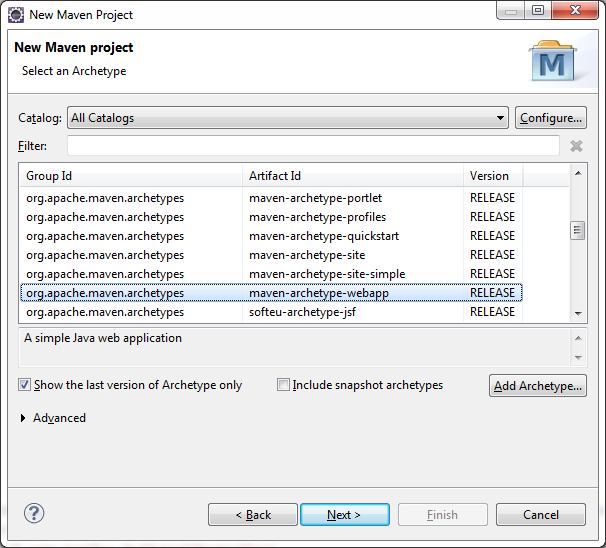
**Step 1: Create a Maven Project**

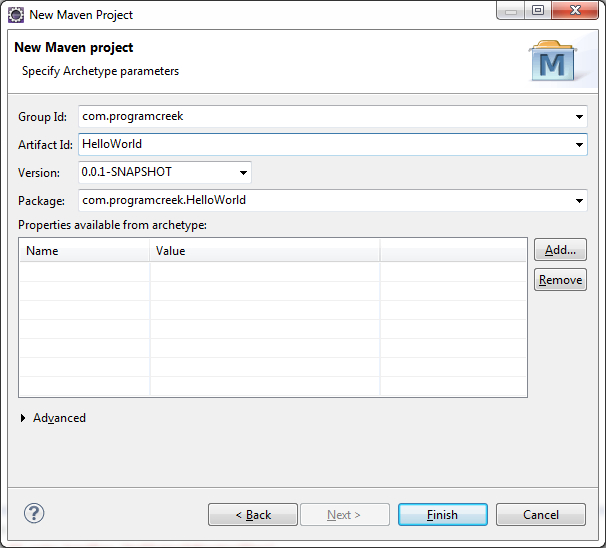
Create a Maven project by following the following steps:





Select "webapp".



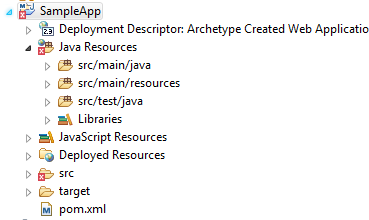


GroupId identifies the project uniquely across all projects, so we need to enforce a naming schema. ArtifactId is the name of the jar without version

After the Maven project is created, the project in the Navigator view should look like the following:

Missing src/main/java

1. Right click the Maven Project -> Build Path -> Configure Build Path
2. In Order and Export tab, you can see the message like '2 build path entries are missing'
3. Now select 'JRE System Library' and 'Maven Dependencies' checkbox
4. Click OK



How to resolve below error



As shown above, there is an error marked with red. If you open index.jsp file, you can see the error message:

The superclass "javax.servlet.http.HttpServlet" was not found on the Java

Build Path

**Installing Tomcat Server on eclipse :**

 Open **Window -> Preferences -> Server -> Installed Runtimes** to create a Tomcat installed runtime.

 Click on **Add...** to open the **New Server Runtime** dialog, then select your runtime under **Apache v8.0**

To fix the problem, right click on project -> Properties -> Java Build Path -> Add Library...-> Server Runtime -> Apache Tomcat -> Finish.

Right click RunAs 🡪 Run on Server.

Default call to the index.jsp, which is available in webapp folder.

**Step 2: Configure Spring to MVC**

To make a Spring web application, we need to configure several xml files. First of all, we need to add Spring dependencies. Edit the automatically generated pom.xml file to be the following:

pom.xml:

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.Sailotech</groupId>

<artifactId>SampleApp</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>SampleApp Maven Webapp</name>

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

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- Spring dependencies -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.3.3.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/commons-logging/commons-logging -->

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

</dependencies>

<build>

<finalName>SampleApp</finalName>

</build>

</project>

Edit the default web.xml:

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"*

id=*"WebApp\_ID"* version=*"3.1"*>

<display-name>Archetype Created Web Application</display-name>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

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

<param-value>WEB-INF/spring/mvc-context.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

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

</servlet-mapping>

</web-app>

mvc-context.xml:

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

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

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

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

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd*

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

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd*

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

<context:component-scan base-package=*"com.stech.test.\*"* >

<context:exclude-filter type=*"annotation"*

expression=*"org.springframework.stereotype.Service"* />

<context:exclude-filter type=*"annotation"*

expression=*"org.springframework.stereotype.Repository"* />

</context:component-scan>

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"*>

<value>/WEB-INF/views/</value>

</property>

<property name=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

</beans>

index.jsp:

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>Spring 4 MVC - HelloWorld Index Page</title>

</head>

<body>

<center>

<h2>Hello World</h2>

<h3>

<a href=*"hello?name=java"*>Click Here</a>

</h3>

</center>

</body>

</html></body>

</html>

**Step 3: Create Spring Controller and View**

Create the package by name “com.stech.test.controller” under directory by name “src/main/java/ “.

HelloWorldController.java

**package** com.stech.test.controller;

**import** org.springframework.stereotype.Controller;

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

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.servlet.ModelAndView;

@Controller

**public** **class** HelloWorldController {

String message = "Welcome to Spring MVC!";

@RequestMapping("/hello")

**public** ModelAndView showMessage(

@RequestParam(value = "name", required = **false**, defaultValue = "World")String name)

{

System.***out***.println("in controller");

ModelAndView mv = **new** ModelAndView("helloworld");

mv.addObject("message", message);

mv.addObject("name", name);

**return** mv;

}

}

Go to WEB-INF folder and create a folder by name “views” and helloworld.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<body>

<h1>Hello World</h1>

<h1>${message}</h1>

<h1>${name}</h1>

</body>

</body>

</html>

Fix for : DispatcherServlet Not Found

You need to add the "Maven Dependency" in the Deployment Assembly

* right click on your project and choose properties.
* click on Deployment Assembly.
* click add
* click on "Java Build Path Entries"
* select Maven Dependencies"
* click Finish.

**Spring Core Module, Introduction to Inversion of Control –**

Spring Core Module is the most important Module of Spring.

Inversion Of Control(IOC) or Dependency Injection is one of best way to implement loosely coupled application.

Using DI (Dependency Injection) ,the framework acts as an Object Factory to build service objects and inject those service objects to application based on some runtime configuration file.

IOC makes the code simpler and easy to test.The IOC container manages the pojo and its dependencies in the application. In IOC the application defines the dependency and these dependencies are then satisfied at the Runtime by the IOC container as given below.

 A caller asks the container for an object with a specific name or a specific type.

 The container injects these objects by name into other objects, via either constructors, properties or factory methods.

Objects are not required to be created. Dependencies are injected into the objects that need them.

Classes are not required to be coupled in any specific implementation which is the biggest advantage of loose coupling. Since an object knows its dependency by their interface, the dependency can be swapped out with a different implementation.

In Spring there are many ways to wire components but a common approach is via XML. Typically the container is configured by loading XML files with some runtime configuration containing bean definition which provide the information required to create the beans.

Before getting into Spring Application development, we need to understand what is tight coupling and loose coupling in java. So let us try to understand about tight and loose coupling between java objects.

## A Tightly Coupled Application

Consider the following application.

**package** com.visix;

**public** **interface** Bank {

**public** **int** getRateOfInterest();

}

**The Bank interface has a method to calculate Rate Of Interest.**

CitiBank.java:

**package** com.visix;

**public** **class** CitiBank **implements** Bank {

**public** **int** getRateOfInterest() {

**return** 15;

}

}

HdfcBank.java:

**package** com.visix;

**public** **class** HdfcBank **implements** Bank {

**public** **int** getRateOfInterest() {

**return** 13;

}

}

The CitiBank and HdfcBank classes implement the Bank interface and the classes generate interest rate related to CitiBank and HdfcBank.

**package** com.visix;

**public** **class** BankService {

Bank bank = **new** HdfcBank();

**public** **void** rateOfInterest() {

System.***out***.println("Rate of Interest is " + bank.getRateOfInterest()+ "%");

}

}

the BankService class displays the rate of interest and it holds the reference of Bank interface.

**package** com.visix;

**public** **class** BankApplication {

**public** **static** **void** main(String as[]) {

BankService bankService = **new** BankService();

bankService.rateOfInterest();

}

}

## A Loosely Coupled Application

To avoid tight coupling, use dependency injection where the component only needs a way to accept the resources and the container will deliver the resources to the components.

Following is updated version of BankService class. It demonstrates a loosely coupled application.

**BankService.java:**

**package** com.visix;

**public** **class** BankService {

Bank bank;

**public** **void** setBank(Bank bank) {

**this**.bank = bank;

}

**public** **void** rateOfInterest() {

System.***out***.println("Rate of Interest is " + bank.getRateOfInterest()

+ "%");

}

}

BankService now doesn't directly create instance of Bank interface, instead of that it will now make use of container to do this. The container will inject the required dependencies.

Dependencies are injected using the setter or Constructor injection. BankService uses the Setter Injection. The value for the Bank is set using setBank() method.

**package** com.visix;

**public** **class** BankApplication {

**public** **static** **void** main(String as[]) {

BankService bankService = **new** BankService();

Bank bank = **new** CitiBank();

bankService.setBank(bank);

bankService.rateOfInterest();

bank = **new** HdfcBank();

bankService.setBank(bank);

bankService.rateOfInterest();

}

}

**Spring IOC Container**

Java Application involve writing a large number of classes.Associating these classes with each other is complicated task.Re usability of a specific class becomes an issue with traditional approach as it is coupled with other classes.Reusing of classes can be achieved only if they are loosely coupled with each other.Spring framework takes care of wiring the classes together due to which classes becomes loosely coupled. Spring framework provides a lightweight container that injects the required functionalities into the code.

In a typical Spring based application, we need to create Simple Java Beans and wire them and let the container manage them. The container gets all the instruction to create, configure and manage the beans from the bean configuration metadata. The configurations can be done through Java code and/or annotations and/or XML files.

Spring provides two interfaces that acts as containers, namely **BeanFactory** and **ApplicationContext** .A **BeanFactory** just initiates and configures beans. An **ApplicationContext** also does that, and also provides more supporting infrastructure that enables lots of enterprise specific features such as Aspect Oriented Programming and transaction. Thus, it is better to use **ApplicationContext** rather than **BeanFactory** , in most cases.Let's discuss **BeanFactory** and **ApplicationContext** containers in detail.

## BeanFactory

The **org.springframework.beans.BeanFactory** container is the root container that loads all the beans and provides dependency injection to enterprise applications. **XmlBeanFactory** and **StaticListableBeanFactory** are some of the classes that implements **BeanFactory** interface.

One of the most popularly used implementation of **BeanFactory** is the **XMLBeanFactory** . **XMLBeanFactory** allows the representation of objects and their rich dependencies in terms of XML. The **XmlBeanFactory** takes XML configuration metadata to create the fully configured application.

To use the BeanFactory, we need to create the instance of **XmlBeanFactory** class as given below:

Resource resource=**new** ClassPathResource("applicationContext.xml");

BeanFactory factory=**new** XmlBeanFactory(resource);

The constructor of **XmlBeanFactory** class receives the **Resource** object so we need to pass the resource object to create the object of BeanFactory.

The following sample shows a simple hello world application using the **XmlBeanFactory** .

HelloWorld.java is the Bean class whose code is given below.

**package** com.visix.spring;

**public** **class** HelloWorld {

**private** String message;

**public** **void** setMessage(String message) {

**this**.message = message;

}

**public** **void** getMessage() {

System.***out***.println("Here is the output : " + message);

}

}

The below XML code shows the contents of the bean xml configuration. It has a single bean configured that has a single property by the name message. A default value is set for the property.

Bean.xml:

<?xml version=*"1.0"* encoding=*"UTF-8"*?> <beans xmlns=*"http://www.springframework.org/schema/beans"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>

<bean id=*"helloWorld"* class=*"com.visix.spring.HelloWorld"*>

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

</bean>

below is the code for the **Test.java** that uses the **XMLBeanFactory** to create the **HelloWorld** bean and invoke a method in the created Spring bean.

**package** com.visix.spring;

**import** org.springframework.beans.factory.InitializingBean;

**import** org.springframework.beans.factory.xml.XmlBeanFactory;

**import** org.springframework.core.io.ClassPathResource;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

XmlBeanFactory factory = **new** XmlBeanFactory(

**new** ClassPathResource("Beans.xml"));

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

helloObj.getMessage();

}

}

In the above code, the XmlBeanFactory loads the metadata based on the CLASSPATH variable. It uses the configuration metadata to load, assemble and dispense the beans upon request. Finally, a method on the bean **(getMessage())** is invoked to get the desired result.

The BeanFactory is preferred when there are limited resources like mobile devices or applets. BeanFactory is a subset of ApplicaitonContext and provides lesser functionalities. When we need full capabilities with respect to configuration handling then we go for ApplicationContext.

**ApplicationContext**

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

It is an advance container that extends the BeanFactory container with various enterprise-level features, given as follows:

* Ability to resolve textual messages from a properties file and helps in internationalizing the applications.
* Integrates with Spring Aspect Oriented Programming(AOP) to generate proxy classes.
* Publishes events to all registered subscribers.

The most popularly used ApplicationContext implementations are:

* **FileSystemXmlApplicationContext** - This implementation loads the definitions of the beans from an XML file. It is required to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** This container loads the definitions of the beans from an XML file. However, it is not required to provide the full path of the XML file. It does require you to set CLASSPATH properly because this container will look for bean configuration XML file in the specified CLASSPATH.
* **XmlWebApplicationContext** : This container loads the XML file with definitions of all beans from within a web application.

The **ClassPathXmlApplicationContext** class is the implementation class of **ApplicationContext** interface. We need to instantiate the **ClassPathXmlApplicationContext** class to use the **ApplicationContext** as given below:

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

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

**Constructor Injection with primitive and String-based values**

Let us see an example of injecting primitive and String based values by using constructor injection. Consider the following Employee bean class.

**Creating Bean Class**

Create a class Employee.java inside the package "com.visix.spring.bean".

Employee.java:

package com.visix.spring.bean;

**public** **class** Employee {

**private** String name;

**private** **int** age;

**private** String country;

**public** Employee(String name, **int** age, String country) {

**this**.name = name;

**this**.age = age;

**this**.country = country;

}

**public** String toString() {

**return** "Employee Name is" + name + " and employee Age is " + age

+ " years, lives in " + country;

}

}

In the above class there are three attributes viz. name, age and country. All the three attributes are set through constructor injection. The toString() method of the class is used to display the Employee information.

**Configure bean in configuration xml file**

Below is the code snippet to use the Set in the spring bean configuration file.

bean.xml:

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg value=*"MNRAO"* />

<constructor-arg value=*"30"* />

<constructor-arg value=*"Hyderabad"* />

</bean>

</beans>

## Creating Test Class

Create a class Test.java to test the program.

### Test.java:

**package** com.visix.spring.bean;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"bean.xml");

Employee employee = (Employee) context.getBean("employee");

System.***out***.println(employee);

}

}

Output:

Employee Name is MNRAO and employee Age is 30 years, lives in Hyderabad

**Constructor injection type ambiguities in Spring**

In the above example there is only one constructor in the Employee class, this code will work fine. But when your class contains multiple constructors with same number of arguments, it will cause the constructor injection argument type ambiguities problem.

Now add two constructor with the same number of arguments in our Employee bean class. Updated Employee bean class is given below.

**package** com.visix.spring.bean;

**public** **class** Employee {

**private** String name;

**private** **int** age;

**private** String city;

**public** Employee(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** Employee(**int** age, String city) {

**this**.age = age;

**this**.city = city;

}

**public** String toString() {

**return** "Employee Name is " + name + " and employee Age is " + age

+ " and employee city is " + city;

}

}

Bean configuration is given below.

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg value=*"24"* />

<constructor-arg value=*"MNRAO"* />

</bean>

</beans>

Now which constructor do you think will be invoked? The second one with the int and the String argument, right? But for your surprise it will call the first constructor with String and int argument . To avoid this confusion you need to specify the type attribute of the constructor-arg element. Now with the following bean configuration, the first constructor will be invoked.

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg type=*"int"* value=*"24"*/>

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

</bean>

</beans>

Now which constructor do you think will be called? The second constructor, right? But again for your surprise the first constructor will be called, this is because the order in which the arguments appear in the bean configuration file will not be considered while invoking the constructor. To solve this problem you can use the *index* attribute to specify the constructor argument index.

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg index=*"0"* value=*"24"* />

<constructor-arg index=*"1"* value=*"Chennai"* />

</bean>

</beans>

**Constructor injection type ambiguities in Spring**

In Spring framework, when your class contains multiple constructors with same number of arguments, it will always cause the constructor injection argument type ambiguities issue.

Let us consider the below Employee class with different types of arguments. One with *String, String* as arguments and other with *float, String* as arguments.

We can inject following types of values by using constructor injection.

**Employee.java:**

**package** com.visix.spring.bean;

**public** **class** Employee {

**private** String name;

**private** String designation;

**private** **float** salary;

**public** Employee(String name, String designation) {

**this**.name = name;

**this**.designation = designation;

}

**public** Employee(**float** salary, String name) {

**this**.salary = salary;

**this**.name = name;

}

**public** **void** displayResult() {

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

System.***out***.println("Designation: " + designation);

System.***out***.println("Salary: " + salary);

}

}

### beans.xml

Below is the code snippet to use the Set in the spring bean configuration file.

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg value=*"10000"* />

<constructor-arg value=*"MNRAO"* />

</bean>

</beans>

In the above configuration, we want to pass 10000 to *salary* and *Mukesh* to name i.e. second constructor (float,String argument) need to be called.

Now create a Test class to check the result.

**package** com.visix.spring.bean;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"beans.xml");

Employee employee = (Employee) context.getBean("employee");

employee.displayResult();

}

}

Name: 10000 Designation: MNRAO Salary: 0.0

If you observe the above output we didn't get expected result. ie. second constructor (float,String) argument was not called, instead first constructor (String,String) argument was called. This is the Constructor injection type ambiguity.

Spring container by default converts every passing value to String value. i.e. In our example 10000 converted to String. That's why second constructor with (String,String) argument was called.

**Solution:**

We can resolve this problem by using type attribute of <constructor-arg> tag. Always specify the exact datatype for constructor-arg value using type attribute in bean configuration file.

Update the bean.xml file as given below.

<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=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg type=*"float"* value=*"10000"* />

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

</bean>

</beans>

O/P:

Name: MNRO Designation: null Salary: 10000.0

Now the ambiguity is solved, but consider one more scenario by adding one more constructor in Employee class. Add one more constructor with String, float as arguments in Employee.java. Updated code is given below.

**package** com.visix.spring.bean;

**public** **class** Employee {

**private** String name;

**private** String designation;

**private** **float** salary;

**public** Employee(String name, String designation) {

**this**.name = name;

**this**.designation = designation;

System.***out***.println("1st constructor called");

}

**public** Employee(String designation, **float** salary) {

**this**.designation = designation;

**this**.salary = salary;

System.***out***.println("2nd constructor called");

}

**public** Employee(**float** salary, String name) {

**this**.salary = salary;

**this**.name = name;

System.***out***.println("3rd constructor called");

}

**public** **void** displayResult() {

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

System.***out***.println("Designation: " + designation);

System.***out***.println("Salary: " + salary);

}

}

Again Run the application but surprise you will get following output.

O/P;

Name: null Designation: MNRAO Salary: 10000.0

If you observe we didn't get expected output. ie. third constructor with (float,String) argument was not called, instead second constructor with (String,float) argument was called. This is again ambiguity in Constructor injection type.

**Solution:**

We can resolve this problem by using index attribute of <constructor-arg> tag. So while using constructor injection always specify the exact datatype for constructor-arg value using type attribute and index attribute in beans.xml.

Update the bean.xml file as given below.

<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=*"employee"* class=*"com.visix.spring.Employee"*>

<constructor-arg type=*"float"* index=*"0"* value=*"10000"* />

<constructor-arg type=*"java.lang.String"* index=*"1"*

value=*"MNRAO"* />

</bean>

</beans>

3rd constructor called

Name: MNRAO Designation: null Salary: 10000.0

**Conclusion :**

It’s always a good practice to explicitly declare the data type and index for each constructor argument, to avoid constructor injection type ambiguities issue.

**Constructor Injection with Dependent Object**

In this example we will discuss about constructor injection with dependent object using a simple program.

Let us create two bean class Address and Employee. Employee class has reference of Address.

Address.java

**package** com.visix.spring.bean;

**public** **class** Address {

**private** String city;

**private** String state;

**private** String country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

**public** String toString() {

**return** "City is " + city + " State is " + state + " and Country is "

+ country;

}

}

**Employee.java**

**package** com.visix.spring.bean;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address address;

**public** Employee(**int** id, String name, Address address) {

**this**.id = id;

**this**.name = name;

**this**.address = address;

}

**void** display() {

System.***out***.println("Employee ID: " + id + " Name: " + name);

System.***out***.println(address.toString());

}

}

### bean.xml

Below is the code snippet to use the Set in the spring bean configuration file. As discussed the ref attribute is used to define the reference of another object.

<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=*"address"* class=*"com.visix.spring.bean.Address"*>

<constructor-arg value=*"Hyderabad"*></constructor-arg>

<constructor-arg value=*"Telangan"*></constructor-arg>

<constructor-arg value=*"India"*></constructor-arg>

</bean>

<bean id=*"employee"* class=*"com.visix.spring.bean.Employee"*>

<constructor-arg value=*"101"* type=*"int"*></constructor-arg>

<constructor-arg value=*"MNRAO"*></constructor-arg>

<constructor-arg>

<ref bean=*"address"* />

</constructor-arg>

</bean>

</beans>

### Test.java

This class gets the bean from the applicationContext.xml file and calls the display() method.

**package** com.visix.spring.bean;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"bean.xml");

Employee e = (Employee) context.getBean("employee");

e.display();

}

}

O/P;

employee ID: 101 Name: MNRAO City is Hyderabad State is Telngana and Country is India

**Spring Constructor Injection with Collections :**

We can inject java collections by using constructor injection in spring framework. We can inject following collection by using constructor injection.

* List
* Set
* Map

In this example, there is a library which contains list of books.

## Files Required :

1. Library.java
2. bean.xml
3. Test.java

**package** com.visix;

**import** java.util.List;

**public** **class** Library {

**private** **int** id;

**private** String name;

**private** List<String> books;

**public** Library(**int** id, String name, List<String> books) {

**this**.id = id;

**this**.name = name;

**this**.books = books;

}

**public** String toString() {

**return** "Books Dtails " + books + "Id is: " + id + " Library Name: "

+ name;

}

}

Bean.xml:

<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=*"library"* class=*"com.visix.Library"*>

<constructor-arg value=*"101"*></constructor-arg>

<constructor-arg value=*"University Library"*></constructor-arg>

<constructor-arg>

<list>

<value>Java Complete Reference</value>

<value>Mastering Servlets</value>

<value>Mastering JSP</value>

</list>

</constructor-arg>

</bean>

</beans>

Test.java:

o/p:

Books Dtails [Java Complete Reference, Mastering Servlets, Mastering JSP]

Id is: 101 Library Name: University Library

**Constructor Injection With Collection Of Custom Object :**

We can inject collection of custom object by using the ref element inside the list, set or map tag in spring configuration file.

Let us see an example of injecting collection of custom objects by using constructor injection.In this example, department class has list of employee object.

## Files Required :

1. Department.java
2. Employee.java
3. bean.xml
4. Test.java

Department.java :

**package** com.visix.spring;

**import** java.util.List;

**public** **class** Department {

**private** **int** id;

**private** String departmentName;

List<Employee> empList;

**public** Department(**int** id, String departmentName, List<Employee> empList) {

**this**.id = id;

**this**.departmentName = departmentName;

**this**.empList = empList;

}

**public** **void** displayResult() {

System.***out***.println("Department ID is :" + id

+ " and Department Name is " + departmentName);

System.***out***.println("Employees are:");

**for** (Employee employee : empList) {

System.***out***.println(employee);

}

}

}

Employee.java:

**package** com.visix.spring;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** String city;

**public** Employee(**int** id, String name, String city) {

**this**.id = id;

**this**.name = name;

**this**.city = city;

}

**public** String toString() {

**return** "Employee ID :" + id + " Employee Name: " + name

+ " Employee City: " + city;

}

}

In the above class there are three attributes viz. id, name and city. All the three attributes are set through constructor injection. The toString() method of the class is used to display the Employee information.

## Configure bean in configuration xml file

Below is the code snippet to use the Set in the spring bean configuration file.

### bean.xml

<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=*"emp1"* class=*"com.visix.spring.Employee"*>

<constructor-arg value=*"100"*></constructor-arg>

<constructor-arg value=*"MNRAO"*></constructor-arg>

<constructor-arg value=*"Hyderabad"*></constructor-arg>

</bean>

<bean id=*"emp2"* class=*"com.visix.spring.Employee"*>

<constructor-arg value=*"101"*></constructor-arg>

<constructor-arg value=*"Ravi"*></constructor-arg>

<constructor-arg value=*"Delhi"*></constructor-arg>

</bean>

<bean id=*"department"* class=*"com.visix.spring.Department"*>

<constructor-arg value=*"100"*></constructor-arg>

<constructor-arg value=*"Finance"*></constructor-arg>

<constructor-arg>

<list>

<ref bean=*"emp1"* />

<ref bean=*"emp2"* />

</list>

</constructor-arg>

</bean>

</beans>

## Creating Test Class

Create a class Test.java to test the program.

### Test.java

**package** com.visix;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.visix.spring.Department;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"Spring-Module.xml");

Department department = (Department) context.getBean("department");

department.displayResult();

}

}

O/P:

Department ID is :100 and Department Name is Finance

Employees are:

Employee ID :100 Employee Name: MNRAO Employee City: Hyderabad

Employee ID :101 Employee Name: Ravi Employee City: Delhi

### Note:

Similarly you can use Set instead of List for that only you need to change **list to set** in the **bean.xml** file and **List to Set** in the **Department.java** file.

**Spring Constructor Injection with Map:**

this example we will discuss about constructor injection with map by using a simple program.

Here, We are using key and value pair both as a string. Like previous examples, it is the example of department where one Department can have multiple Employee.

**Department.java:**

**package** com.visix.spring;

**import** java.util.Map;

**public** **class** Department {

**private** **int** departmentId;

**private** String departmentName;

**private** Map<String, String> employee;

**public** Department(**int** departmentId, String departmentName,

Map<String, String> employee) {

**this**.departmentId = departmentId;

**this**.departmentName = departmentName;

**this**.employee = employee;

}

**public** **void** displayResult() {

System.***out***.println("Department id is : " + departmentId);

System.***out***.println("Department Name is : " + departmentName);

System.***out***.println("Employee Details....");

**for** (Map.Entry me : employee.entrySet()) {

System.***out***.println("Employee Designation :" + me.getKey() + ", Name:"

+ me.getValue());

}

}

}

bean.xml:

<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=*"department"* class=*"com.visix.spring.Department"*>

<constructor-arg value=*"101"*></constructor-arg>

<constructor-arg value=*"Finance"*></constructor-arg>

<constructor-arg>

<map>

<entry key=*"Manager"* value=*"Amit"*></entry>

<entry key=*"Senior Manager"* value=*"Ravi"*></entry>

<entry key=*"Director"* value=*"Raj"*></entry>

</map>

</constructor-arg>

</bean>

</beans>

### Test.java

This class gets the bean from the applicationContext.xml file and calls the displayResult() method of Department bean class.

**package** com.visix;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.visix.spring.Department;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"spring-config.xml.xml");

Department department = (Department) context.getBean("department");

department.displayResult();

}

}

O/P:

Department id is : 101

Department Name is : Finance

Employee Details....

Employee Designation :Manager, Name:Amit

Employee Designation :Senior Manager, Name:Ravi

Employee Designation :Director, Name:Raj

Autowiring in Spring :

In Spring usually we provide bean configuration details in the spring bean configuration file and specify the beans that will be injected in other beans using ref attribute or using an inner bean. Spring framework provides autowiring features where we don’t need to provide bean injection details explicitly and spring container will take care about injecting the dependencies.

By default autowiring is disabled in spring framework and programmer has to explicitly wire the bean properties into an xml file. If autowiring is enabled then spring container will take care about injecting the dependencies, and no need to configure into an xml file explicitly.

Autowiring can't be used to inject primitive and string values, it will only supported if the dependancies are in the form of objects.

To enable Autowiring, just define the "autowire" attribute in <bean>.

<bean id=*"employee"* class=*"com.visix.spring.Employee"* autowire=*"byName"*/>

## Autowiring Modes:

There are five autowiring modes available in Spring.

 **no** - It is the default autowiring mode. It means no autowiring by default.

 **byName**- The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.

 **byType**-The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.

 **constructor**-The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

 **autodetect**-It is deprecated since Spring 3.

**Example:**

We are going to create two java classes Department and Employee for demonstration of all the modes mentioned above

**Department.java:**

**package** com.visix.spring;

**public** **class** Department {

**private** Employee employee;

**public** Department(Employee employee) {

**this**.employee = employee;

}

**public** **void** setEmployee(Employee employee) {

**this**.employee = employee;

}

//..

}

**Employee.java:**

**package** com.visix.spring;

**public** **class** Employee {

//..

}

**1. Auto-Wiring 'no'**

This is the default mode, you need to wire your bean via 'ref' attribute.

Bean.xml:

<bean id=*"Department"* class=*"com.visix.spring.Department"*>

<property name=*"employee"* ref=*"employee"* />

</bean>

<bean id=*"employee"* class=*"com.visix.spring.Employee"* />

**2. Auto-Wiring 'byName'**

The **byName** mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. In this case, since the name of "employee" bean is same with the name of the "Department" bean’s property ("employee"), so, Spring will auto wired it via setter method – "setEmployee(Employee employee)".

<bean id=*"department"* class=*"com.visix.spring.Department"* autowire=*"byName"*/>

<bean id=*"employee"* class=*"com.visix.spring.Employee"* />

**3. Auto-Wiring 'byType'**

The byType mode bean looks for other beans with the same type as property. For instance in our demo code 'Department' class has 'employee' variable, then configuring autowire=byType on 'Department' bean makes the container look for the beans of 'Employee' class and inject into 'Department' bean. Spring will auto wired it via setter method.

<bean id=*"department"* class=*"com.visix.spring.Department"* autowire=*"byType"* />

<bean id=*"employee"* class=*"com.visix.spring.Employee"* />

**4. Auto-Wiring 'constructor'**

Autowiring by constructor is similar to byType, but in byType we used setter injection here we have to use constructor injection. The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

In this case, since the data type of "employee" bean is same as the constructor argument data type in "department" bean’s property (Employee object), so, Spring auto wired it via constructor – "Department(Employee employee)".

<bean id=*"department"* class=*"com.visix.spring.Department"* autowire=*"constructor"* />

<bean id=*"employee"* class=*"com.visix.spring.Employee"* />

Spring Autowiring by Name Example

In Autowiring by Name mode, spring framework will try to find out a bean in the configuration file, whose id is matching with the property name to be wired. If a bean found with id as property name then that class object will be injected into that property by calling setter injection. If no id is found then that property remains un-wired without throwing any exception.

The following example shows using byName autowiring mode.

You can enable this feature via autowire="byName" like below :

<bean id=*"employee"* class=*"com.visix.spring.Employee"* autowire=*"byName"*/>

# Example On Spring Autowiring byName

**Files Required.**

* Book.java
* Author.java
* spring-config.xml
* Test.java

**Book.java:**

**package** com.visix.spring;

**public** **class** Book {

**private** Author author;

**private** String isbn;

**private** String year;

**public** Author getAuthor() {

**return** author;

}

**public** **void** setAuthor(Author author) {

**this**.author = author;

}

**public** String getIsbn() {

**return** isbn;

}

**public** **void** setIsbn(String isbn) {

**this**.isbn = isbn;

}

**public** String getYear() {

**return** year;

}

**public** **void** setYear(String year) {

**this**.year = year;

}

**public** String toString() {

**return** "Book [ISBN=" + isbn + ", Author=" + author + ", Year=" + year;

}

}

**Author.java:**

**package** com.visix.spring;

**public** **class** Author {

**private** String name;

**private** String address;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** String toString() {

**return** "Author [Name=" + name + ", Address=" + address;

}

}

spring-config.xml:

<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"*>

<bean id=*"book"* class=*"com.visix.spring.Book"* autowire=*"byName"*>

<property name=*"isbn"* value=*"1478589684"* />

<property name=*"year"* value=*"2010"* />

</bean>

<bean id=*"author"* class=*"com.visix.spring.Author"*>

<property name=*"name"* value=*"MNRAO"* />

<property name=*"address"* value=*"India"* />

</bean>

</beans>

In this bean.xml,

<bean id=*"author"* class=*"com.visix.spring.Author"*>

id=*"author" ,* author is an attribute in the class “Book”,

it creates instance of an *Author* class and makes a call to setter methods, setName() and setAddress() by passing values.

Once data has been set in the object of *Author* class, makes a call to setAuthor() of Book class by passing object of *Author* class.

O/P:

Book [ISBN=1478589684, Author=Author [Name=MNRAO, Address=India, Year=2010

In this case, since the name of author bean is same with the name of the book bean's property [author], so Spring auto wires it via the setter method i,e, setAuthor(Author author).

Spring Autowiring by Type Example :

This mode allows a property to be autowired if exactly one bean of the property type exists in the container.In other words if a bean found with class as property type then that class object will be injected into that property by calling setter injection. If more than one exists a fatal exception is thrown, which indicates byType autowiring may not be used for that bean. If no class found then that property remains un-wired, but never throws any exception just like before.

The following example shows using byType autowiring mode

You can enable this feature via autowire="byType" like below :

<bean id=*"employee"* class=*"com.visix.spring.Employee"* autowire=*"byType"*/>

# Example On Spring Autowiring byType

Files Required.

* Book.java
* Author.java
* spring-config.xml
* Test.java

**Book.java**

**package** com.visix.spring;

**public** **class** Book {

**private** Author author;

**private** String isbn;

**private** String year;

**public** Author getAuthor() {

**return** author;

}

**public** **void** setAuthor(Author author) {

**this**.author = author;

}

**public** String getIsbn() {

**return** isbn;

}

**public** **void** setIsbn(String isbn) {

**this**.isbn = isbn;

}

**public** String getYear() {

**return** year;

}

**public** **void** setYear(String year) {

**this**.year = year;

}

**public** String toString() {

**return** "Book [ISBN=" + isbn + ", Author=" + author + ", Year=" + year;

}

}

**Author.java**

**package** com.visix.spring;

**public** **class** Author {

**private** String name;

**private** String address;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** String toString() {

**return** "Author [Name=" + name + ", Address=" + address;

}

}

spring-config.xml:

<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"*>

<bean id=*"book"* class=*"com.visix.spring.Book"* autowire=*"byType"*>

<property name=*"isbn"* value=*"Effective Java"* />

<property name=*"year"* value=*"2010"* />

</bean>

<bean id=*"author"* class=*"com.visix.spring.Author"*>

<property name=*"name"* value=*"MNRAO"* />

<property name=*"address"* value=*"INDIA"* />

</bean>

</beans>

**Test.java:**

**package** com.visix;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.visix.spring.Book;

**public** **class** Test {

**public** **static** **void** main(String as[]) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-config.xml");

Book book = (Book) context.getBean("book");

System.***out***.println(book);

}

}

O/P:

Book [ISBN=Effective Java, Author=Author [Name=MNRAO, Address=INDIA, Year=2010

In the above example "Book" bean has a property with data type of "Author" class, Spring container will find the bean with same data type of class "Author" and wire it automatically via setter method. And if no matching found, just do nothing.

**Spring Autowiring by Constructor Example:**

Autowiring by constructor is similar to byType,but in byType we used setter injection here we have to use constructor injection.The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

The following example shows using constructor autowiring mode .

You can enable this feature via autowire="constructor" like below :

<bean id=*"employee"* class=*"com.visix.spring.Employee"* autowire=*"constructor"*/>

# Example On Spring Autowiring by Constructor

Files Required.

* Book.java
* Author.java
* spring-config.xml
* Test.java

**package** com.visix.spring;

**public** **class** Book {

**private** Author author;

**private** String isbn;

**private** String year;

**public** Book(Author author) {

**this**.author = author;

}

**public** Author getAuthor() {

**return** author;

}

**public** **void** setAuthor(Author author) {

**this**.author = author;

}

**public** String getIsbn() {

**return** isbn;

}

**public** **void** setIsbn(String isbn) {

**this**.isbn = isbn;

}

**public** String getYear() {

**return** year;

}

**public** **void** setYear(String year) {

**this**.year = year;

}

**public** String toString() {

**return** "Book [ISBN=" + isbn + ", Author=" + author + ", Year=" + year;

}

}

**Author.java:**

**package** com.visix.spring;

**public** **class** Author {

**private** String name;

**private** String address;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getAddress() {

**return** address;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** String toString() {

**return** "Author [Name=" + name + ", Address=" + address;

}

}

spring-config.xml:

<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"*>

<bean id=*"book"* class=*"com.visix.spring.Book"* autowire=*"constructor"*>

<property name=*"isbn"* value=*"Effective Java"* />

<property name=*"year"* value=*"2010"* />

</bean>

<bean id=*"author"* class=*"com.visix.spring.Author"*>

<property name=*"name"* value=*"MNRAO"* />

<property name=*"address"* value=*"INDIA"* />

</bean>

</beans>

Test.java:

**package** com.visix;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.visix.spring.Book;

**public** **class** Test {

**public** **static** **void** main(String as[]) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("spring-config.xml");

Book book = (Book) context.getBean("book");

System.***out***.println(book);

}

}

O/P:

Book [ISBN=Effective Java, Author=Author [Name=MNRAO, Address=INDIA, Year=2010

In the above example since the data type of author bean is same as the constructor argument data type in book bean's property(Author) object, Spring auto wires it via constructor public Book(Author author).

**Spring JdbcTemplate :**

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.

## Spring Jdbc Approaches

Spring framework provides following approaches for JDBC database access:

* JdbcTemplate
* NamedParameterJdbcTemplate
* SimpleJdbcTemplate
* SimpleJdbcInsert and SimpleJdbcCall

## 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.

Methods of spring JdbcTemplate class.

**public int update(String query) 🡪** is used to insert, update and delete records.

**public int update(String query, Object... args)** 🡪 is used to insert, update and delete records using PreparedStatement using given arguments.

**public void execute(String query)🡪** is used to execute DDL query.

**public T execute(String sql, PreparedStatementCallback action)🡪** executes the query by using PreparedStatement callback.

**public T query(String sql, ResultSetExtractor rse)🡪** is used to fetch records using ResultSetExtractor.

**public List query(String sql, RowMapper rse)🡪** is used to fetch records using RowMapper.

### Example of Spring JdbcTemplate

We are assuming that you have created the following table inside the Oracle11g database.

create table employee(

id number(10),

name varchar2(100),

salary number(10)

);

Create Dynamic Web Application in eclipse:

“SpringJDBCExample”

Add spring and spring jdbc jars to project class path.

Right click on project 🡪 properties 🡪 Java Build Path 🡪 libraries 🡪 Add External Jars 🡪 Browse to Spring and select all libraries and click on OK.

And also add Jdbc connector jar for oracle (ojdbc14-1.0.jar )

And also add “commons-logging-1.1.1.jar” for logs

Create a package “com.visix.spring.jdbc” under folder “src”

Create **Employee.java inside the above package.**

**Employee.java**

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

**package** com.visix.spring.jdbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **float** salary;

**public** Employee(**int** id, String name, **float** salary)

{

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**float** salary) {

**this**.salary = salary;

}

}

**EmployeeDao.java:**

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

**package** com.visix.spring.jdbc;

**import** org.springframework.jdbc.core.JdbcTemplate;

**public** **class** EmployeeDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

**public** **int** saveEmployee(Employee e){

String query="insert into employee values("+e.getId()+",'"+e.getName()+"',"+e.getSalary()+")";

**return** jdbcTemplate.update(query);

}

**public** **int** updateEmployee(Employee e){

String query="update employee set name='"+e.getName()+"',salary="+e.getSalary()+"where id="+e.getId();

**return** jdbcTemplate.update(query);

}

**public** **int** deleteEmployee(Employee e){

String query="delete from employee where id="+e.getId();

**return** jdbcTemplate.update(query);

}

}

Create a xml file for bean configuration.

**Create “applicationContext.xml” under src folder.**

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.

**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.

**package** com.visix.spring.jdbc;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("employeedao");

**int** status=dao.saveEmployee(**new** Employee(102,"MNRAO",35000));

System.***out***.println(status);

// int status=dao.updateEmployee(new Employee(102,"MNRAO1",15000));

// System.out.println(status);

//Employee e=new Employee();

//e.setId(102);

//int status=dao.deleteEmployee(e);

//System.out.println(status);

}

}

RunAs 🡪 JavaApplication

Example of PreparedStatement in Spring JdbcTemplate:

We can execute parameterized query using Spring JdbcTemplate by the help of **execute()** method of JdbcTemplate class. To use parameterized query, we pass the instance of **PreparedStatementCallback** in the execute method.

**Syntax of execute method to use parameterized query**

public T execute(String sql,PreparedStatementCallback<T>);

**PreparedStatementCallback interface**

It processes the input parameters and output results. In such case, you don't need to care about single and double quotes.

**Method of PreparedStatementCallback interface**

It has only one method doInPreparedStatement. Syntax of the method is given below:

public T doInPreparedStatement(PreparedStatement ps)throws SQLException, DataAccessException

### Example of using PreparedStatement in Spring

We are assuming that you have created the following table inside the Oracle11g database.

create table employee1(

id number(10),

name varchar2(100),

salary number(10)

);

**Employee.java**

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

**package** com.visix.spring.jdbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **float** salary;

**public** Employee()

{

id=0;

name=**null**;

salary=0.0f;

}

**public** Employee(**int** id, String name, **float** salary)

{

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**float** salary) {

**this**.salary = salary;

}

}

**EmployeeDao.java**

It contains one property jdbcTemplate and one method saveEmployeeByPreparedStatement. You must understand the concept of annonymous class to understand the code of the method.

**package** com.visix.spring.jdbc;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** org.springframework.dao.DataAccessException;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.PreparedStatementCallback;

**public** **class** EmployeeDao

{

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate)

{

**this**.jdbcTemplate = jdbcTemplate;

}

**public** Boolean saveEmployeeByPreparedStatement(**final** Employee e)

{

String query="insert into employee1 values(?,?,?)";

**return** jdbcTemplate.execute(query, **new** PreparedStatementCallback<Boolean>()

{

@Override

**public** Boolean doInPreparedStatement(PreparedStatement ps)

**throws** SQLException, DataAccessException {

ps.setInt(1,e.getId());

ps.setString(2,e.getName());

ps.setFloat(3,e.getSalary());

**return** ps.execute();

}

});

}

}

**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.

**applicationContext.xml**

<beans

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

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

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

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

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

<bean id=*"ds"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:ORCL"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

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

</bean>

<bean id=*"employeedao"* class=*"com.visix.spring.jdbc.EmployeeDao"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>

</bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the saveEmployeeByPreparedStatement() method.

**package** com.visix.spring.jdbc;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao empDao=(EmployeeDao)ctx.getBean("employeedao");

empDao.saveEmployeeByPreparedStatement(**new** Employee(108,"MNRAO",35000));

System.***out***.println("successfully inserted ");

}

}

ResultSetExtractor Example | Fetching Records by Spring JdbcTemplate:

We can easily fetch the records from the database using **query()** method of **JdbcTemplate** class where we need to pass the instance of ResultSetExtractor.

Syntax of query method using ResultSetExtractor:

**public T query(String sql, ResultSetExtractor<T> rse)**

**ResultSetExtractor Interface**

**ResultSetExtractor** interface can be used to fetch records from the database. It accepts a ResultSet and returns the list.

**Method of ResultSetExtractor interface**

It defines only one method extractData that accepts ResultSet instance as a parameter. Syntax of the method is given below:

**public T extractData(ResultSet rs)throws SQLException,DataAccessException**

**Example of ResultSetExtractor Interface to show all the records of the table**

We are assuming that you have created the following table inside the Oracle11g database.

create table employee2(

id number(10),

name varchar2(100),

salary number(10)

);

**Employee.java**

This class contains 3 properties with constructors and setter and getters. It defines one extra method toString().

**package** com.visix.spring.jdbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **float** salary;

**public** Employee()

{

id=0;

name=**null**;

salary=0.0f;

}

**public** Employee(**int** id, String name, **float** salary)

{

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**float** salary) {

**this**.salary = salary;

}

**public** String toString(){

**return** id+" "+name+" "+salary;

}

}

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployees.

**package** com.visix.spring.jdbc;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.dao.DataAccessException;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.ResultSetExtractor;

**public** **class** EmployeeDao {

**private** JdbcTemplate template;

**public** **void** setJdbcTemplate(JdbcTemplate template) {

**this**.template = template;

}

**public** List<Employee> getAllEmployees(){

**return** template.query("select \* from employee1",**new** ResultSetExtractor<List<Employee>>(){

@Override

**public** List<Employee> extractData(ResultSet rs) **throws** SQLException,

DataAccessException {

List<Employee> list=**new** ArrayList<Employee>();

**while**(rs.next()){

Employee e=**new** Employee();

e.setId(rs.getInt(1));

e.setName(rs.getString(2));

e.setSalary(rs.getInt(3));

list.add(e);

}

**return** list;

}

});

}

}

**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.

**applicationContext.xml:**

<beans

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

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

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

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

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

<bean id=*"ds"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:ORCL"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

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

</bean>

<bean id=*"employeedao"* class=*"com.javatpoint.EmployeeDao"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>

</bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the getAllEmployees() method of EmployeeDao class.

**package** com.visix.spring.jdbc;

**import** java.util.List;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("employeedao");

List<Employee> list=dao.getAllEmployees();

**for**(Employee e:list)

System.***out***.println(e);

}

}

**RowMapper Example | Fetching records by Spring JdbcTemplate**

Like ResultSetExtractor, we can use RowMapper interface to fetch the records from the database using **query()** method of **JdbcTemplate** class. In the execute of we need to pass the instance of RowMapper now.

**Syntax of query method using RowMapper:**

**public T query(String sql,RowMapper<T>  rm)**

**RowMapper Interface**

**RowMapper** interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

**Advantage of RowMapper over ResultSetExtractor**

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

**Method of RowMapper interface**

It defines only one method mapRow that accepts ResultSet instance and int as the parameter list. Syntax of the method is given below:

**public T mapRow(ResultSet rs, int rowNumber)throws SQLException**

**Example of RowMapper Interface to show all the records of the table**

We are assuming that you have created the following table inside the Oracle11g database.

create table employee(

id number(10),

name varchar2(100),

salary number(10)

);

**Employee.java**

This class contains 3 properties with constructors and setter and getters and one extra method toString().

**package** com.visix.spring.jdbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **int** salary;

**public** Employee()

{

id=0;

name=**null**;

salary=0;

}

**public** Employee(**int** id, String name, **int** salary)

{

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** String toString(){

**return** id+" "+name+" "+salary;

}

}

**EmployeeDao.java**

It contains on property jdbcTemplate and one method getAllEmployeesRowMapper.

**package** com.visix.spring.jdbc;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.List;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.RowMapper;

**public** **class** EmployeeDao {

**private** JdbcTemplate template;

**public** **void** setJdbcTemplate(JdbcTemplate template) {

**this**.template = template;

}

**public** List<Employee> getAllEmployeesRowMapper(){

**return** template.query("select \* from employee1",**new** RowMapper<Employee>(){

@Override

**public** Employee mapRow(ResultSet rs, **int** rownumber) **throws** SQLException {

Employee e=**new** Employee();

e.setId(rs.getInt(1));

e.setName(rs.getString(2));

e.setSalary(rs.getInt(3));

**return** e;

}

});

}

}

**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.

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

<beans

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

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

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

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

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

<bean id=*"ds"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:ORCL"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

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

</bean>

<bean id=*"employeedao"* class=*"com.javatpoint.EmployeeDao"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>

</bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the getAllEmployeesRowMapper() method of EmployeeDao class.

**package** com.visix.spring.jdbc;

**import** java.util.List;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ctx=**new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao=(EmployeeDao)ctx.getBean("employeedao");

List<Employee> list=dao.getAllEmployeesRowMapper();

**for**(Employee e:list)

System.***out***.println(e);

}

}

Spring SimpleJdbcTemplate Example

Spring 3 JDBC supports the java 5 feature var-args (variable argument) and autoboxing by the help of SimpleJdbcTemplate class.

SimpleJdbcTemplate class wraps the JdbcTemplate class and provides the update method where we can pass arbitrary number of arguments.

Syntax of update method of SimpleJdbcTemplate class:

**int update(String sql,Object... parameters)**

**Example of SimpleJdbcTemplate class**

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

create table employee(

id number(10),

name varchar2(100),

salary number(10)

);

**Employee.java:**

**package** com.visix.spring.jdbc;

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** **int** salary;

**public** Employee()

{

id=0;

name=**null**;

salary=0;

}

**public** Employee(**int** id, String name, **int** salary)

{

**this**.id=id;

**this**.name=name;

**this**.salary=salary;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**int** salary) {

**this**.salary = salary;

}

**public** String toString(){

**return** id+" "+name+" "+salary;

}

}

**EmployeeDao.java**

It contains one property SimpleJdbcTemplate and one method update. In such case, update method will update only name for the corresponding id. If you want to update the name and salary both, comment the above two lines of code of the update method and uncomment the 2 lines of code given below.

**package** com.visix.spring.jdbc;

**import** org.springframework.jdbc.core.simple.SimpleJdbcTemplate;

**public** **class** EmployeeDao {

SimpleJdbcTemplate template;

**public** EmployeeDao(SimpleJdbcTemplate template) {

**this**.template = template;

}

**public** **int** update (Employee e){

String query="update employee set name=? where id=?";

**return** template.update(query,e.getName(),e.getId());

//String query="update employee set name=?,salary=? where id=?";

//return template.update(query,e.getName(),e.getSalary(),e.getId());

}

}

**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 SimpleJdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the SimpleJdbcTemplate class for the datasource property.

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

**applicationContext.xml**

<beans

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

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

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

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

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

<bean id=*"ds"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"oracle.jdbc.driver.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@localhost:1521:ORCL"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"jtemplate"* class=*"org.springframework.jdbc.core.simple.SimpleJdbcTemplate"*>

<constructor-arg ref=*"ds"*></constructor-arg>

</bean>

<bean id=*"edao"* class=*"com.visix.spring.jdbc.EmployeeDao"*>

<constructor-arg>

<ref bean=*"jtemplate"*/>

</constructor-arg>

</bean>

</beans>

Test.java:

**package** com.visix.spring.jdbc;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.beans.factory.xml.~~XmlBeanFactory~~;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Resource r=**new** ClassPathResource("applicationContext.xml");

BeanFactory factory=**new** XmlBeanFactory(r);

EmployeeDao dao=(EmployeeDao)factory.getBean("edao");

**int** status=dao.update(**new** Employee(108,"MNRAO1",35000));

System.***out***.println(status);

}

}

**Spring - MVC Framework:**

Spring MVC is used to develop the web applications that uses MVC design pattern. By using Spring MVC we can build flexible and loosely coupled web applications.The MVC design pattern helps in seperating the business logic, presentation logic and controller logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

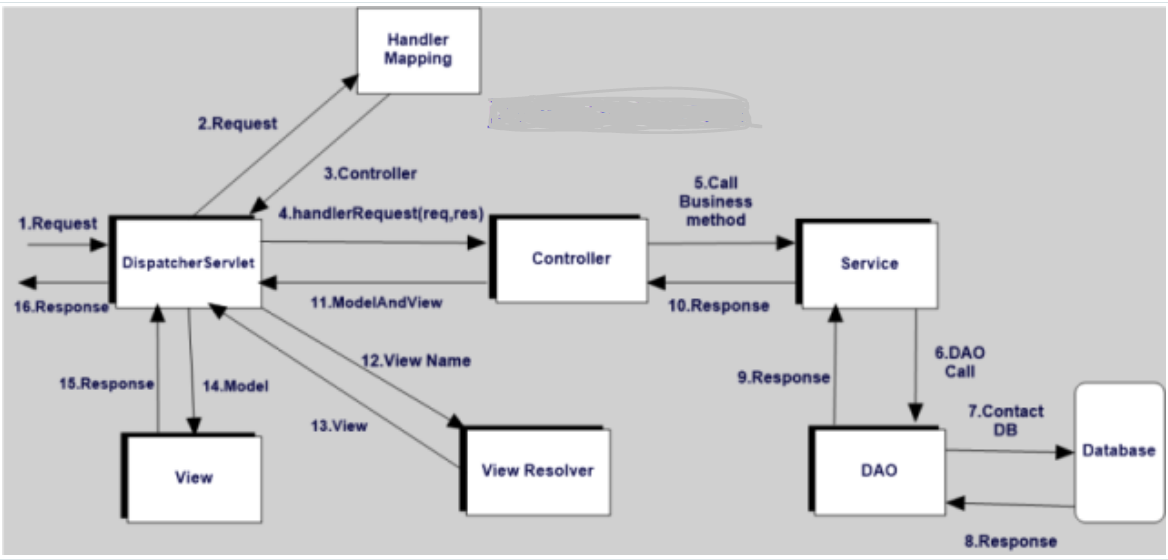
Spring MVC will make application development faster, cost effective and flexible.

## Advantage of Spring MVC Framework

* Clear separation of responsibilities because it implements MVC design pattern.
* Loose coupling among Model, View and Controller.
* Inbuilt front controller.
* Simplified Validation implementation.
* Simplified Exception Handling.
* Tiles Support.
* Internationalization logic is simplified compared to other frameworks.
* Spring MVC provides a set of custom tags which are useful to implement presentation logic easily.

**Spring MVC Flow:**





* 1. Client(Browser) requests for a Resource in the web Application.
  2. The Spring front controller i.e, DispatcherServlet first receives the request.
  3. DispatcherServlet consults the HandlerMapping to identify the particular controller for the given URL.
  4. HandlerMapping identifies the controller for the given request and sends to the DispatcherServlet
  5. DispatcherServlet will call the handleRequest(request, response) method on Controller. A Controller is developed by writing a simple java class which implements Controller interface or extends its adapter class.
  6. Controller will call the business method according to business requirement.
  7. Service class will call the DAO class method for business data.
  8. DAO interacts with DB to get data.
  9. DAO returns same data to service
  10. Fetched data will be processed according to business requirement and return results to Controller.
  11. The Controller returns the Model and View in the form of Object back to the front Controller i.e, DispatcherServlet.
  12. The front controller i.e, DispatcherServlet then tries to resolve the actual View which may be JSP, velocity or Free Marker by consulting the View Resolver Object.
  13. ViewResolver selected view is rendered back to the DispatcherServlet.
  14. DispatcherServlet consult the particular view with the model.
  15. View executes and returns HTML output to the DispatcherServlet.
  16. DispatcherServlet will sends the output to the Browser.

# Spring 4 MVC+Hibernate 4+MySQL+Maven integration example using annotations

In this case study, we will integrate Spring with Hibernate using annotation based configuration. We will develop a simple CRUD oriented web application containing a form asking user input, saving that input in MySQL database using Hibernate , retrieving the records from database and updating or deleting them within transaction, all using annotation configuration.

**Following technologies being used:**

* Spring 4.0.6.RELEASE
* Hibernate Core 4.3.6.Final
* validation-api 1.1.0.Final
* hibernate-validator 5.1.3.Final
* Oracle 11g
* Maven 3
* JDK 1.8
* Tomcat 8.0.21
* Eclipse Neon Service Release 2