## **SPRING FRAMEWORK**

## ***What is Spring*:**

Spring is an open source framework to develop enterprise application in a declarative fashion, based on MVC design pattern.It creates the dependent objects and inject those into the bean.

## ***Inventor of Spring:***

Spring Framework is one of the most popular Java EE frameworks.

It is an open source and light weight framework created by **Rod Johnson** in June **2003**.

## ***Features of Spring:***

1. **Light weight:** Spring framework is light weight framework because of its POJO model implementation.

2. **Loose Coupling:** Because of dependency injection concept, spring objects are loosely coupled.

3. **Modular fashion:** Spring framework is designed in modular fashion. A programmer can use only needed modules and ignore the rest.

4. **Transaction management:** Spring framework provides transaction management interface for transaction management.

5. **Aspect Oriented Programming(AOP):**

It is an important part of Spring Framework. Aspect Oriented Programming is used for separating cross-cutting concerns

(for example logging, security etc.) from the business logic of the application.

6. **Integration with other frameworks:**

It just tries to integrate them with its framework which provides a solution to greater problems. Example IBATIS, Hibernate, Toplink etc

## ***How Spring Has Come:***

Earlier **Java Beans** was used in development of java applications and was intended to be used as reusable components.Complex enterprise applications that requires more functionality like security and transaction management java beans were not sufficient.

So to address the complexity of enterprise application,**EJB** was evolved.EJB provided all the intended functionality but failed to achieve the simplicity of application.EJB applications were very complex so harder to maintain.

So a framework that provides the EJB functionality and simplicity of application development was needed.**Spring** provides programming techniques like AOP and DI;these techniques used with POJOs result in implementing enterprise functionality without EJB complexity.

## ***Callback method:***

A callback method in java is a method which is called when an event occurs. Normally we can implement that by passing an implementation of a certain interface to the system which is responsible for triggering the event.

## ***Commonly used callback methods in spring:***

**Post-initialization callback methods:**

**Using InitializingBean interface:**

The InitializingBean interface provides afterPropertiesSet() method which can be used for any post-initialization task.

## ***Syntax:***

|  |
| --- |
| public class TestBean implements **InitializingBean** {  public void afterPropertiesSet() {  // post-initialization task.  }  } |

## ***Using init-method attribute:***

In XML configuration metadata we can specify the name of the method which has a void no-argument signature in init-method attribute for any post-initialization task.

## **Syntax:**

|  |
| --- |
| <bean id="testBean" class="Test" **init-method**="init"/> **In class definition:** public class Test {  public void init() {  // post-initialization task.  }  } |

**Pre-destroy callback methods:**

**Using DisposableBean interface:**

The DisposableBean interface provides destroy() method which can be used for any pre-destroy task.

***Syntax:***

|  |
| --- |
| public class TestBean implements **DisposableBean** {  public void destroy() {  // pre-destroy task.  }  } |

***Using destroy-method attribute:***

In XML configuration metadata we can specify the name of the method which has a void no-argument signature in destroy-method attribute for any pre-destroy task.

**Syntax:**

|  |
| --- |
| <bean id="testBean" class="Test" **destroy-method=**" destroy"/>  **In class definition:**  public class Test {  public void destroy() {  // pre-destroy task.  }  } |

***Spring IoC Container:***

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

***IOC(Inversion Of Controller):***   Giving control to the container to get instance of object is called **Inversion of Control** means instead of you are creating object using new operator, let the container do that for you.

**By DI, the responsibility of creating objects is shifted from our application code to Spring container, hence the phenomenon is called IOC**.

***Injection:***

Injection is a process of passing the dependency to a dependent object.

***DI(Dependency Injection):***  Way of injecting properties to an object is called **Dependency injection**.

Dependency Injection (DI) is a design pattern that implements inversion of control principle for resolving dependencies

***Types of dependency Injection:***

***1.Constructor-based Dependency Injection.***

|  |
| --- |
| **Syntax 1:**  <bean id="student" class="com.Student">  <constructor-arg index="0" value="10"/>  <constructor-arg index="1" value="Debopriyo Dutta"/>  </bean>  **Syntax 2:**  <bean id="address" class="com.Address">  **<constructor-arg index="0" value="Kolkata"></constructor-arg>**  **<constructor-arg index="1" value="WestBengal"></constructor-arg>**  </bean>  <bean id="employee" class="com.Employee">  <constructor-arg index="0" value="101"></constructor-arg>  <constructor-arg index="1" value=" Debopriyo "></constructor-arg>  **<constructor-arg index="2">**  **<ref bean="address" />**  **</constructor-arg>**  </bean>  **In class definition:**  **Employee.java**  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;  }  }  **Address.java**  public class Address {  private String city;  private String state;  public Address(String city, String state) {  this.city = city;  this.state = state;  }  } |

***2. Setter-based Dependency Injection.***

**Syntax 1:**

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

<property name="name" value="Debopriyo Dutta"></property>

</bean>

**Syntax 2:**

<bean id="employee" class="com.Employee">

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

**<property name="age" value="28" />**

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

</bean>

<bean id="address" class="com.Address">

<property name="doorNo" value="48B" />

<property name="street" value="Selimpur Lane" />

<property name="area" value="Near Milan Thirtha Club" />

</bean>

**In class definition:**

**Employee.java**

public class Employee {

private String name;

private int age;

private Address addrs;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Address getAddrs() {

return addrs;

}

public void setAddrs(Address addrs) {

this.addrs = addrs;

}

}

**Address.java**

public class Address {

private String doorNo;

private String street;

private String area;

public String getDoorNo() {

return doorNo;

}

public void setDoorNo(String doorNo) {

this.doorNo = doorNo;

}

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

public String getArea() {

return area;

}

public void setArea(String area) {

this.area = area;

}

}

Spring provides following two distinct types of containers:

***1.BeanFactory container***

***2.ApplicationContext container***

***Spring BeanFactory container:***

**BeanFactory** is a Lightweight container which loads bean definitions and manages your beans .The **BeanFactory** enables you to read bean definitions and access them using the bean factory.

When using just the **BeanFactory** you would create one and read in some bean definitions in the XML format as follows:

***How to Create XmlBeanFactory:***

ClassPathResource resource = new ClassPathResource("beans.xml");

BeanFactory factory = new XmlBeanFactory(resource);

//Get bean

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

In the Spring framework, **ApplicationContext** is the advanced container. It extends the BeanFactory interface and provides more facilities than BeanFactory such as integration with spring AOP, message resource handling for i18n etc.The Spring container is responsible for instantiating,

configuring and assembling objects known as beans, as well as managing their lifecycle.

The Spring framework provides several implementations of the **ApplicationContext** interface — **ClassPathXmlApplicationContext** and

**FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

Among the many implementations of **ApplicationContext** are three that are

commonly used:

-> **ClassPathXmlApplicationContext**—Loads a context definition from an

XML file located in the class path, treating context definition files as class

path resources.

-> **FileSystemXmlApplicationContext**—Loads a context definition from an

XML file in the filesystem.

-> **XmlWebApplicationContext**—Loads context definitions from an XML file

contained within a web application.

**ApplicationContext context =**

**new FileSystemXmlApplicationContext("c:/foo.xml");**

Similarly, you can load an application context from within the application’s class

path using ClassPathXmlApplicationContext:

**ApplicationContext context =**

**new ClassPathXmlApplicationContext("foo.xml");**

The difference between these uses of **FileSystemXmlApplicationContext** and

**ClassPathXmlApplicationContext** is that **FileSystemXmlApplicationContext** will

look for foo.xml in a specific location, whereas **ClassPathXmlApplicationContext**

will look for foo.xml anywhere in the class path.

***ContextLoaderListener:***

The **ContextLoaderListener** is a servlet listener that loads additional configuration files into spring application context along with application context

created by DispatcherServlet.

<web-app>

<servlet>

<servlet-name>disp</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>disp</servlet-name>

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

</servlet-mapping>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

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

<param-value> /WEB-INF/applicationContext.xml, /WEB-INF/spring-security.xml </param-value>

</context-param>

</web-app>

We should tell to **ContextLoaderListener** which spring configuration file(s) it should load, otherwise the context loader will look for spring

configuration file at /WEB-INF/applicationContext.xml.

***Spring Bean Life Cycle:***

The most important feature of Spring is the bean based approach. The Spring bean is created, managed and dispensed by the Spring IoC container.

Each Spring bean has a lifecycle and understanding the spring bean lifecycle enables better coding.

The life cycle of a Spring bean is very easy to understand. When a bean is instantiated, it may be required to perform some initialization to get

it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required. There are also

other activities between initialization and destruction of the bean. These activities take place behind the scenes.The following are the stages

in a bean’s lifecycle.

1.**Instantiate -** The Spring container instantiates the bean.

2. **Populate properties-** Spring IoC container injects the bean’s properties.

3. **Set Bean Name-** Spring container sets the bean name. If the bean implements **BeanNameAware**, spring container passes the bean’s id to **setBeanName()** method.

4.**Set Bean Factory-**If the bean implements **BeanFactoryAware**, Spring container passes theBeanFactory to **setBeanFactory().**

5.**Pre Initialization-**This stage is also called the bean postprocess . If there are anyBeanPostProcessors, theSpring container calls the

**postProcesserBeforeInitialization ()** method.

6.**Initialize beans-** If the bean implements IntializingBean,its **afterPropertySet()**method is called. If the bean has init method declaration, the specified

initialization method is called.

7.**Post Initialization**- If BeanPostProcessors is implemented by the bean, the Spring container calls their **postProcessAfterinitalization()** method.

8. **Ready to Use-** Now the bean is ready to be used by the application.

9.**Destroy-** The bean is destroyed during this stage. If the bean implements DisposableBean, the Spring IoC container will call the destroy() method .

If a custom destroy () method is defined, the container calls the specified method.

**Bean lifecycle in spring framework:**

1. Spring container finds the bean definition from configuration file.  
2. Spring container instantiates the bean using Java Reflection API.  
3. Spring container applies the all specified properties using DI.  
4. If the bean class implements the **BeanNameAware** interface, then spring container calls the **setBeanName()** method by passing bean’s id.  
5. If the bean class implements the **BeanClassLoaderAware** interface, then spring container calls the **setBeanClassLoader()** method by passing an instance of the ClassLoader object that loaded this bean.  
6. If the bean class implements the **BeanFactoryAware** interface, then spring container calls **setBeanFactory()** method by passing an instance of BeanFactory object.  
7. If there are any **BeanPostProcessors** object associated with the BeanFactory than spring container calls their **postProcessBeforeInitialization()** method even before setting the properties for the bean.  
8. If the bean class implements the **InitializingBean** interface, then spring container calls the **afterPropertiesSet()** method after setting bean properties.  
9. If **init-method** is specified in configuration file for the bean then spring container calls the corresponding method in the bean class.  
10. If there are any **BeanPostProcessors** associated with the bean then spring container calls the **postProcessAfterInitialization()** method.  
11. If the bean class implements the **DisposableBean** interface, then spring container calls the **destroy()** method when the application no longer needs the bean reference.  
12. If **destroy-method** is specified in the Configuration file for the bean, then spring container calls the corresponding method in the bean class.



### **Example-**

Lets write an example to implement InitalizingBean and DisposableBean interface

**Solution:**

a) Write a PersonBean which implements InitializingBean and DisposableBean interface like below

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

public class PersonBean implements InitializingBean,DisposableBean{

private String name;

public PersonBean()

{

System.out.println("Constructor of person bean is called !! ");

}

@Override

public void destroy() throws Exception

{

System.out.println("destroy method of person bean is called !! ");

}

@Override

public void afterPropertiesSet() throws Exception

{

System.out.println("afterPropertiesSet method of person bean is called !! ");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

b) Create a beans.xml file in src directory to define the PersonBean<?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="personBean" class="PersonBean" >

<property name="name" value="Dummy Person"/>

</bean>

</beans>

c) Create TestPersonBean class which will just loads the beans.xml and test the person bean life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestPersonBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

PersonBean bean = (PersonBean)context.getBean("personBean");

System.out.println(bean.getName());

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d)  Run the Program

You will see below output. Initialization and Destroy methods are getting called.

## Sample program output

*Bean Name, Factory, Application context Aware interfaces*

Several times functionality requires infrastructure or we can say application context information in a bean. To achieve such functionalities ,Spring framework  provides  a range of Aware interfaces Each interface requires us to implement a method to inject the dependency in bean. Most commonly used are –

* **BeanFactoryAware** - This interface provides setBeanFactory() method  that supplies the owning bean factory instance to the bean. Signature of the method is

void setBeanFactory(BeanFactory beanFactory) throws BeansException

* **BeanNameAware**- This interface provides setBeanName() method which sets the name of the bean in the bean factory that created this bean. Signature of the method is-

void setBeanName(String name);

* **ApplicationContextAware** -This interface provides setApplicationContext() method  that supplies the owning application context instance to the bean. Signature of the method is

void setApplicationContext(ApplicationContext applicationContext) throws BeansException

### **Example**

Lets write an example to implement Aware interfaces

**Solution:**

a) Create  a class (AwareBean) which implements ApplicationContextAware, BeanNameAware and BeanFactoryAware

import java.util.Arrays;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.BeanFactoryAware;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class AwareBean implements **ApplicationContextAware,BeanNameAware,BeanFactoryAware**{

@Override

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

System.out.println("setBeanFactory method of Aware bean is called");

System.out.println("setBeanFactory:: Aware bean singleton="

+ beanFactory.isSingleton("awareBean"));

}

@Override

public void setBeanName(String beanName) {

System.out.println("setBeanName method of Aware bean is called");

System.out.println("setBeanName:: Bean Name defined in context="

+ beanName);

}

@Override

public void setApplicationContext(ApplicationContext applicationContext)

throws BeansException {

System.out.println("setApplicationContext method of Aware bean is called");

System.out.println("setApplicationContext:: Bean Definition Names="

+ Arrays.toString(applicationContext.getBeanDefinitionNames()));

}

}

**b)** Create a beans.xml file in src directory to define the AwareBean

<?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="awareBean" class="AwareBean" >

</bean>

</beans>

c)  Create TestAwareBean class which will just loads the beans.xml and test the aware life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext

public class TestAwareBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

AwareBean bean = (AwareBean)context.getBean("awareBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d) Run the Program

You will see below output.

Sample Program output

*Custom init() and destroy() methods in bean configuration file*

Implementing InitalizingBean and DisposableBean interface is simple to use but create tight coupling with the Spring framework in our bean implementations.

Alternatively we can **init-method** and **destroy-method** attribute values for the bean in the spring bean configuration file. This is the recommended approach because of no direct dependency to spring framework and we can create our own methods.

Note: Both post-init and pre-destroy methods should have no arguments but they can throw Exceptions

<beans>

<bean id="bean\_id" class="bean.class"

init-method="customInitmethod"

destroy-method="customDestroymethod">

</bean>

</beans>

We can configure the default init-method  and destroy-method which will be applied on all the beans .They are useful when we have a pattern of defining common method names such as init() and destroy() for all your beans consistently.

<beans default-init-method=”customDefaultInitMethod” default-destroy-method=”customDefaultDestroyMethod” >

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

</bean>

</beans>

### **Example**

Write and example to show the init-method and destroy-method

**Solution**

a)Write a class CustomLifeCycleMehodBean

public class CustomLifeCycleMethodBean {

private String name;

public CustomLifeCycleMethodBean()

{

System.out.println("Constructor of  bean is called !! ");

}

public void customDestroy() throws Exception {

System.out.println("custom destroy method of  bean is called !! ");

}

public void customInit() throws Exception {

System.out.println("custom Init  method of  bean is called !! ");

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

b) Create a beans.xml file in src directory to define the CustomMethodLifeCycleBean

<?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="customLifeCycleMethodBean"

class="CustomLifeCycleMethodBean"

init-method="customInit"

destroy-method="customDestroy">

<property name="name" value="custom methods bean" ></property>

</bean>

</beans>

c) Create TestCustomMethodLifeCycleBean class which will just loads the beans.xml and test the custom methods life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCustomMethodLifeCycleBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

CustomLifeCycleMethodBean bean = (CustomLifeCycleMethodBean)context.getBean("customLifeCycleMethodBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

d)Run the Program

You will see below output and custom life cycle methods are getting called

## output to show custom life cycle methods are being called

### **Example**

Write an example to demonstrate global init and destroy methods

**Solution**

a)Write a class CustomGlobalLifeCycleMehodBean

public class CustomGlobalLifeCycleMehodBean {

public CustomGlobalLifeCycleMehodBean()

{

System.out.println("Constructor of  bean is called !! ");

}

public void globalCustomDestroy() throws Exception {

System.out.println("global custom destroy method of  bean is called !! ");

}

public void globalCustomInit() throws Exception {

System.out.println("global custom Init  method of  bean is called !! ");

}

}

**b)** Create a beans.xml file in src directory to define the CustomGlobalMethodLifeCycleBean

<?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"

**default-init-method="globalCustomInit"**

**default-destroy-method="globalCustomDestroy"**>

<bean id="customGlobalLifeCycleMethodBean"

class="CustomGlobalLifeCycleMehodBean" />

</beans>

c) Create TestCustomMethodLifeCycleBean class which will just loads the beans.xml and test the custom methods life cycle

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestCustomGlobalMethodLifeCycleBean {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext("beans.xml");

CustomGlobalLifeCycleMehodBean bean = (CustomGlobalLifeCycleMehodBean)context.getBean("customGlobalLifeCycleMethodBean");

((AbstractApplicationContext) context).registerShutdownHook();

}

}

*Spring Bean Scope:*

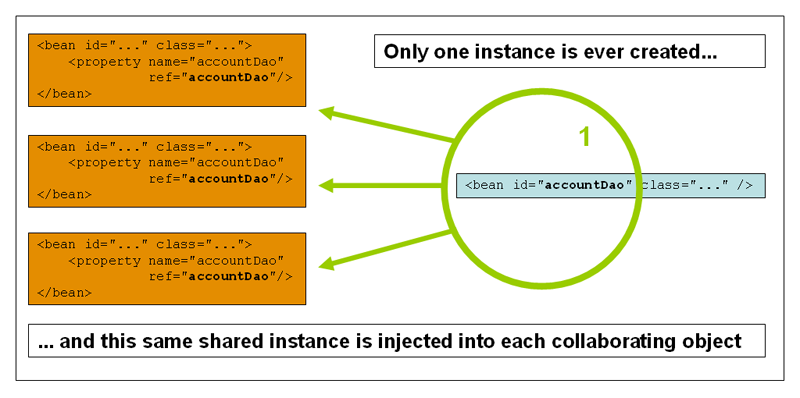
**Spring framework supports following five scopes**. Out of which three scopes are supported only in web ApplicationContext.

1. **Singleton**
2. **Prototype**
3. **Request**
4. **Session**
5. **Global Session**

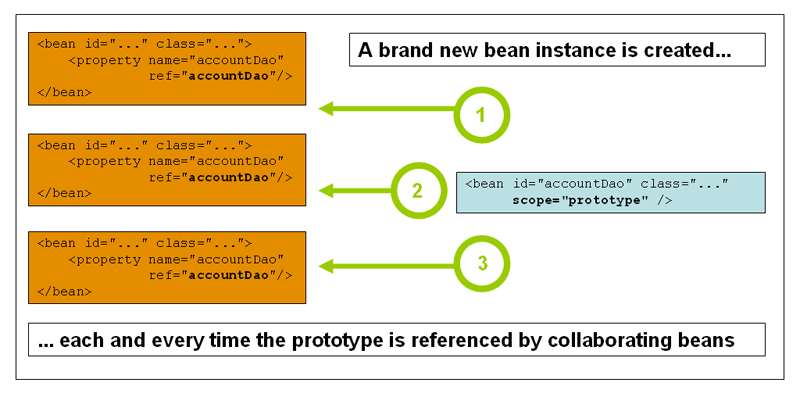
**Note:**Request, Session and Global Session scopes are valid in the context of a web-aware Spring ApplicationContext. This means that you can only use these scoped beans in an application deployed to a web server. Spring can be used in applications that run in standard JVMs along with applications that run in servlet containers (Tomcat, etc). Request, Session and Global session however, only exists in web servers so it has no meaning if the application is running in a standard desktop environment

*Singleton and Prototype:*

**The Singleton scopes the bean definition to a single instance per Spring IoC container (default)**. If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. You have to carefully understand that it is single for its own IoC container, not the JVM or your entire application. Because your application may have more than one IoC container.

[](http://d3t0dn7puh4fxw.cloudfront.net/wp-content/uploads/2013/03/singleton.png)

The **Prototype** scopes a single bean definition to have any number of object instances. If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made.

[](http://d3t0dn7puh4fxw.cloudfront.net/wp-content/uploads/2013/03/prototype.png)

*Singleton and Prototype Example:*

**HelloWorld.java**

package com;

import java.util.Date;

public class HelloWorld {

private String message;

private Date date;

public Date getDate() {

return date;

}

public void setDate(Date date) {

this.date = date;

}

public void setMessage(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

}

*MainApp.java*

package com;

import java.util.Date;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

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

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

System.out.println("\*\*\*\*\*\*\*\*\*SINGLETON SCOPE\*\*\*\*\*\*\*\*\*\*\*\*");

objA.setMessage("Message by object A");

objA.setDate(new Date());

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

System.out.println("Date : " + objA.getDate().toString());

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

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

System.out.println("Date : " + objB.getDate().toString());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

HelloWorld objC = (HelloWorld) context.getBean("prototypescope");

System.out.println("\*\*\*\*\*\*\*\*\*PROTOTYPE SCOPE\*\*\*\*\*\*\*\*\*\*\*\*");

objC.setMessage("Message by object C");

objC.setDate(new Date());

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

System.out.println("Date : " + objC.getDate().toString());

HelloWorld objD = (HelloWorld) context.getBean("prototypescope");

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

System.out.println("Your Date : " + objD.getDate());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

Step 2:

Create Beans configuration file **Beans.xml** under the src folder.

<?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="prototypescope" class="com.HelloWorld" **scope="prototype"**>

</bean>

<bean id="singletonscope" class="com.HelloWorld" **scope="singleton"**>

</bean>

</beans>

Step 3:

As a final step, let us run the application. If everything is fine with your application, the following output is printed:

\*\*\*\*\*\*\*\*\*SINGLETON SCOPE\*\*\*\*\*\*\*\*\*\*\*\*

Your Message : Message by object A

Date : Fri Mar 29 17:39:21 IST 2013

Your Message : Message by object A

Date : Fri Mar 29 17:39:21 IST 2013

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*PROTOTYPE SCOPE\*\*\*\*\*\*\*\*\*\*\*\*

Your Message : Message by object C

Date : Fri Mar 29 17:39:21 IST 2013

Your Message : null

Your Date : null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*Details of the above output:*

Here we see that in the case of singleton scope, the second retrieval by objB will display the same message and Date which was set by objA,

even though its retrieved by a new getBean() method. In singleton scope, no matter how many times you retrieve it with getBean(), it will always

return the same instance.

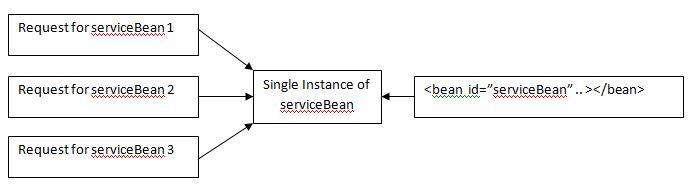
In prototype scope, you will have a new instance for each getBean() method called. Hence for the second retrieval you see that both message

and date are null.

With the help of **Spring Bean Scope** container can decide how particular bean should be created and returned to caller. **Spring Bean Scope** helps the container to decide how particular bean should be treated.   
  
Bean is the actual object of the class that is created by the Spring IoC container. We can create bean definition with the help of XML configuration metadata. When you are defining the metadata configuration for the particular bean, you are actually creating the blueprint or prototype on how IoC container should create the instance and how bean is dependent on other beans.   
  
IoC container reads those information from metadata and create the instance of that bean. There are different types of bean scope; we will discuss them one by one

|  |  |
| --- | --- |
| **Scope Name** | **Description** |
| Singleton | Default Scope, Single object instance per IoC container. |
| Prototype | Scopes a single bean definition to any number of object instances. |
| Request | Each HTTP request will have its own instance. |
| Session | Scopes bean to Life cycle of Session. |
| Global Session | Scopes bean to life cycle of Global Session. Usually in portlet context. |

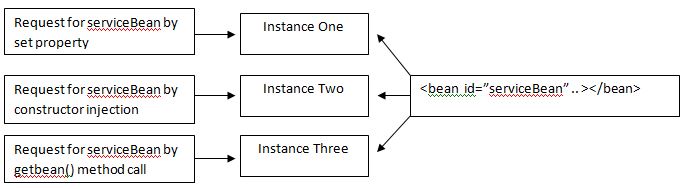
*Singleton Scope*

Singleton scope is the default scope of the bean, only a single shared instance of singleton bean is managed by IoC container. For all requests for that particular singleton bean, there will be one specific bean instance returned by IoC container.   
  
When you define a bean definition as Singleton bean, then IoC container creates only single shared instance of that bean and store it to cache.  
  
  
  [](https://www.ashtpoint.com/wp-content/uploads/2017/05/singleton.jpg) 

*XML Example*

<bean id="serviceBean" class="com.package.ServiceBean" scope="singleton"/>

*Prototype Scope*

Other bean scope is “prototype”, prototype scope of bean will create new bean instance every time a request for that been is raised. That is, whether bean is injected to another bean or request is made through getBean() method. There will be a new instance each time. It is useful for state full beans.   
  
  
  
[](https://www.ashtpoint.com/wp-content/uploads/2017/05/prototype.jpg) 

*XML Example*

<bean id="serviceBean" class="com.package.ServiceBean" scope="prototype"/>

Note\*: Spring does not manage complete life cycle of the prototype bean. Although initialization life cycle callback methods are called on all of the prototype objects but destruction life cycle callback methods are not called. The client code must clean prototype objects and release any resource that object is holding up. You can also use bean post processor as well for this particular operation.

*Singleton beans with prototype bean dependency*

One important thing to consider here with prototype and singleton scoped bean is that how they are injected to each other and how they are resolved during instantiation. As we know all Singleton scoped bean are resolved when context is starting up and prototype scoped bean are resolved when request is raised. So when you inject prototype scoped bean into Singleton bean scope then a new prototype bean is instantiated and then injected into singleton bean.

*Request, session, global session, application and Web socket scopes*

All of the scopes can only be used when web-aware spring Application context is implemented such as XMLWebApplicationContext. If you try to use them with **ClassPathXMLApplicationContext** then there will be an IlligalStateException.   
  
If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring **DispatcherServlet** or DispatcherPortlet, then no special setup is necessary:   
  
**DispatcherServlet** and DispatcherPortlet already expose all relevant state.   
  
  
**Scope Example**

<bean id="serviceBean" class="com.package.ServiceBean" scope="request"/>

<bean id="serviceBean" class="com.package.ServiceBean" scope="session"/>

<bean id="serviceBean" class="com.package.ServiceBean" scope="globalsession"/>

<bean id="serviceBean" class="com.package.ServiceBean" scope="prototype"/>

<bean id="serviceBean" class="com.package.ServiceBean" scope="prototype"/>

*Spring BeanPostProcessor:*

With the help of Spring **BeanPostProcessor**, you can easily process the instance created by the Spring IoC container. You can write your own

logic after a bean is created by the container like resource resolution, read some data from external file, validate the data member values etc.

Any bean implementing **BeanPostProcessor** is processed first by the container. You can also control the order in which more than one BeanPostProcessor

should get executed by setting order property. You can only set this property if the BeanPostProcessor implements the Ordered interface.

Important thing to remember

Any bean you write which implements BeanPostProcessor is not eligble for AOP proxies because AOP proxies are applied this way. So be careful when

writing codes, it is possible that AOP proxy may not yet have applied to bean instance. ApplicationContext automatically detects bean which

implements the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors and will be processed later by the container.

**BeanPostProcessor** provides two methods as follow :

**public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;**

**public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;**

Sample Example on how to implement BeanPostProcessor

Bean Class

package com.bpp;

public class HelloBPP {

private String name;

public void setName(String name) {

this.name = name;

}

public String getName() {

return this.name;

}

public void init() {

System.out.println("Init method called..");

}

public void destroy() {

System.out.println("Destroy method called..");

}

}

Bean implementing the BeanPostProcessor

package com.bpp;

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

import org.springframework.beans.BeansException;

public class BppProcessor implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean,

String beanName) throws BeansException {

return bean;

}

public Object postProcessAfterInitialization(Object bean,

String beanName) throws BeansException {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

return bean;

}

}

Writing **beans.xml** (Bean Definition) <?xml version="1.0" encoding="UTF-8"?>

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

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

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

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

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

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

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

<bean id="helloBPP" class="com.bpp.HelloBPP" **init-method="init"** **destroy-method="destroy"**>

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

</bean>

<bean class="com.bpp.BppProcessor"/>

</beans>

Finally calling the bean import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.bpp.HelloBPP;

public class Main {

public static void main(final String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

HelloBPP helloBPP = (HelloBPP) ctx.getBean("helloBPP");

System.out.println(helloBPP);

}

}

*Autowiring:*

Autowiring allows Spring to do the instantiation of the class.We don’t need to write any explicit injection logic.At run time you will be able to access

all methods of the class without worrying about how you got the

class.

*Advantage of Autowiring:*

It requires the less code because we don't need to write the code to inject the dependency explicitly.

•**Reduces the configuration for properties and constructor.**

•**Automatically gets updated when the configuration change.**

*How Spring Autowiring Works?*

All spring beans are managed by spring container called “application context”. The autowiring happens at the time of application starts up. When any autowiring configuration is found either by xml configuration meta data or @Autowired annotation, Spring will look for a class that matches the property in the applicationContext, and inject it automatically.

**Note: There may be the situations when more than one property matched the class spring is looking for, in that case we have to provide the qualify information to help spring for identifying which one in should use.**

*Spring Autowiring Modes:*

* **no:**  No autowiring at all. Bean references must be defined via a reference variable.
* **byName:** Autowiring by property name. This will inspect the application context and look for a bean named exactly the same as the property which needs to be autowired.
* **byType:** Allows a property to be autowired if there is exactly one bean of the property type in the application context. If there is more than one, a fatal exception is thrown.
* **constructor:** This is analogous to byType, but applies to constructor arguments.

*Example On Autowiring byname:*

Autowiring byName means whenever spring finds any property to be autowired, it will search for **exactly**one bean of given property **name** in container. If Spring find one(**unique** bean) it will autowire it. If it doesn’t find any, no auto wiring will be done(Property will not be set). If there are more than one bean of same type in container then Spring will throw Exception that byName can not be used here.

*How to Enable byType Autowiring:*

byName Autowiring can be enabled by using autowire=”byName” like below.

*Example:*

public class MyBean

{

private DemoBean db;

public void setDb(DemoBean db)

{

this.db=db;

}

}

*In the xml file*

<beans>

<bean id="id1" class="MyBean" **autowire="byName"** />

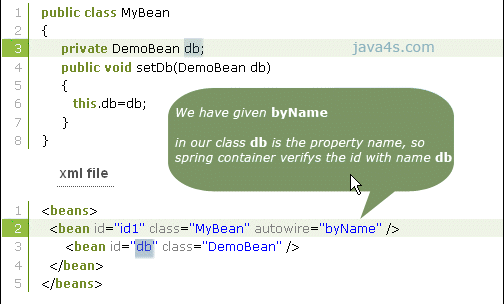
<bean id="db" class="DemoBean" />

</beans>

*Explanation:*

See line number 3 in MyBean, our class depends on DemoBean class object right, now see in the xml file line number 2 we have given autowire=”byName“, means when ever spring container notice autowire=”byName” then it will verifies whether the id in xml file is matching with the property name in the MyBean or not, if yes it will wired automatically else unwired

Am giving one figure to make you understand better



*Program to demonstrate how AutoWiring works in Spring Framework.*

In the Spring Framework bean model, beans have relationships with other beans. In order to satisfy these dependencies of injecting one bean into another we use autowire functionality of Spring Framework. In this tutorial we are taking 4 scenarios by which Spring autowires one bean to another. The Scenarios are changed based on Spring Configuration files and rest all the source code given in this tutorial is the same.

In order to run each scenario individually we just change Spring Configuration xml file and rest code remains the same. So for each scenario to run individual we need

1. Employee POJO class which has-a Address class object

2. Address POJO class which will be wired to Employee class through auto-wiring

3. Test class to test the application

4. Scenario based xml file provided below individually for each scenario

*Address POJO class -*

package com;

public class Address {

private String street;

private String city;

private String state;

public Address() {

}

public String getStreet() {

return street;

}

public void setStreet(String street) {

this.street = street;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getState() {

return state;

}

public void setState(String state) {

this.state = state;

}

}

*Employee POJO class –*

package com;

// Create a POJO class Employee which has a

// Address Object reference as instance variable

public class Employee {

private String name;

private int age;

private Address address;

public Employee( String name ) {

this.name = name;

}

public Employee( Address address) {

this.address = address;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

}

**Scenario 1 - autowire="no"**

<?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"

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

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

http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<!-- Scenario 1 -->

<bean id="employee" class="com.Employee"

**autowire="no"**>

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

<property name="age" value="34" />

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

</bean>

<bean id="address" class="com.Address">

<property name="street" value="Selimpur Lane" />

<property name="city" value="Kolkata" />

<property name="state" value="WestBengal"></property>

</bean>

</beans>

In the above scenario we are using autowire attribute value as "no". It means we are using no auto-wiring, so we have to provide ref attribute for referring any bean.

**Scenario 2 - autowire="byName"**

<?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"

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

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

http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<!-- Scenario 2 -->

<bean id="employee" class="com.Employee"

**autowire="byName"**>

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

<property name="age" value="34" />

</bean>

<bean id="address" class="com.Address">

<property name="street" value="Selimpur Lane" />

<property name="city" value="Kolkata" />

<property name="state" value="WestBengal"></property>

</bean>

</beans>

In the above scenario we are using autowire attribute value as "byName". It means we are using auto-wiring based on name of property, so here Employee has a Address property. Here we have a property in Employee by name as : Address address. So when we say that auto-wire by name Spring sees in configuration that whether there is any bean with id equal to name of property. As soon as it finds this property and its matching id, it uses its setter method to set the property .

**Scenario 3 - autowire="byType"**

<?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"

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

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

http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<!-- Scenario 3 -->

<bean id="employee" class="com.Employee"

**autowire="byType"**>

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

<property name="age" value="34" />

</bean>

<bean id="address" class="com.Address">

<property name="street" value="Selimpur Lane" />

<property name="city" value="Kolkata" />

<property name="state" value="WestBengal"></property>

</bean>

</beans>

In the above scenario we are using autowire attribute value as "byType". It means we are using auto-wiring based on type of property, so here Employee has a Address property. Here we have a property in Employee by type as : com.Address . So when we say that auto-wire by type Spring sees in configuration that whether there is any bean with class equal to type of property. As soon as it finds any matching, it uses its setter method to set the property.

**Scenario 4 - autowire="constructor"**

<?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"

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

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

http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<!-- Scenario 4 -->

<bean id="employee" class="com.Employee"

**autowire="constructor"**>

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

<property name="age" value="34" />

</bean>

<bean id="address" class="com.Address">

<property name="street" value="Selimpur Lane" />

<property name="city" value="Kolkata" />

<property name="state" value="WestBengal"></property>

</bean>

</beans>

In the above scenario we are using autowire attribute value as "constructor". It means we are using auto-wiring based on constructor arguments , so here Employee has a Address property. Here we have a constructor in Employee class which takes a argument as : Address address. So when we say that auto-wire by constructor Spring sees in configuration that whether there is any bean with id equal to constructor argument. As soon as it finds any matching , it uses its setter method to set the property .

**Test.java** -

package com;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

public static void main(String[] args) {

ApplicationContext context =

new ClassPathXmlApplicationContext(("spring.xml"));

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

System.out.println("The name of Employee is : " + employee.getName());

System.out.println("The age of Employee is : " + employee.getAge());

System.out.println("The address of Employee is : " +

employee.getAddress().getStreet() +" "+

employee.getAddress().getCity() +" "+

employee.getAddress().getState());

}

}

*Inheriting Bean in Spring*

By using the **parent** attribute of bean, we can specify the inheritance relation between the beans. In such case, parent bean values will be inherited to the current bean.

Let's see the simple example to inherit the bean.

**Employee.java**

This class contains three properties, three constructor and show() method to display the values.

package com;

public class Employee {

private int id;

private String name;

private Address address;

public Employee() {}

public Employee(int id, String name) {

super();

this.id = id;

this.name = name;

}

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

super();

this.id = id;

this.name = name;

this.address = address;

}

void show(){

System.out.println(id+" "+name);

System.out.println(address);

}

}

**Address.java**

package com;

public class Address {

private String addressLine1,city,state,country;

public Address(String addressLine1, String city, String state, String country) {

super();

this.addressLine1 = addressLine1;

this.city = city;

this.state = state;

this.country = country;

}

public String toString(){

return addressLine1+" "+city+" "+state+" "+country;

}

}

**applicationContext.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: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="e1" class="com.Employee">

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

<constructor-arg value="Sachin"></constructor-arg>

</bean>

<bean id="address1" class="com.Address">

<constructor-arg value="21,Lohianagar"></constructor-arg>

<constructor-arg value="Ghaziabad"></constructor-arg>

<constructor-arg value="UP"></constructor-arg>

<constructor-arg value="USA"></constructor-arg>

</bean>

<bean id="e2" class="com.Employee" **parent="e1"**>

**<constructor-arg ref="address1"></constructor-arg>**

</bean>

</beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the show method.

package com;

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);**

Employee e1=(Employee)factory.getBean("e2");

e1.show();

}

}

Spring framework provides the facility to inject collection values via constructor or setter method. We can use the following inside the constructor or property element.

Collection Injection  
*1.* List.  
2. Set.  
3. Map.

4. Properties

## Syntax (constructor based dependency injection):

|  |
| --- |
| <bean id="testBeanId" **class**="Test">  **<constructor-arg>**  **<list>**  **<value>value1</value>**  **<value>value2</value>**  **<value>value3</value>**  **</list>**  **</constructor-arg>**  </bean> |

## Syntax (setter based dependency injection):

|  |
| --- |
| <bean id="testBeanId" **class**="Test">  <property name="testProperty">  **<list>**  **<value>value1</value>**  **<value>value2</value>**  **<value>value3</value>**  **</list>**  </property>  </bean> |

## **Example Explanation:**

We have created two beans “Student” and “Address”. Student class requires an Address class object. In spring configuration file we define Address bean objects and pass these objects as a list in constructor-arg.

## Example:

**Student.java**

|  |
| --- |
| **package** com;    **import** java.util.List;    **public** **class** Student {  **private** String name;  **private** String rollNo;  **private** String className;  **private** List<Address> address;    **public** Student(List<Address> address){  **this**.address = (List<Address>) address;  }    **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getRollNo() {  **return** rollNo;  }  **public** **void** setRollNo(String rollNo) {  **this**.rollNo = rollNo;  }  **public** String getClassName() {  **return** className;  }  **public** **void** setClassName(String className) {  **this**.className = className;  }  **public** List<Address> getAddress() {  **return** address;  }  } |

**Address.java**

|  |
| --- |
| **package** com;  **public** **class** Address {  **private** String addLine;  **private** String city;  **private** String state;  **private** String country;    **public** String getAddLine() {  **return** addLine;  }  **public** **void** setAddLine(String addLine) {  **this**.addLine = addLine;  }  **public** String getCity() {  **return** city;  }  **public** **void** setCity(String city) {  **this**.city = city;  }  **public** String getState() {  **return** state;  }  **public** **void** setState(String state) {  **this**.state = state;  }  **public** String getCountry() {  **return** country;  }  **public** **void** setCountry(String country) {  **this**.country = country;  }  } |

**applicationContext.java**

|  |
| --- |
| <?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="student" **class**="com.Student">  <property name="name" value="Papia"/>  <property name="rollNo" value="BTech/12"/>  <property name="className" value="BTech"/>  **<constructor-arg>**  **<list>**  **<ref bean="address1"/>**  **<ref bean="address2"/>**  **</list>**  **</constructor-arg>**  </bean>    <bean id="address1" **class**="com.Address">  <property name="addLine" value="Test address1"/>  <property name="city" value="Kolkata"/>  <property name="state" value="WestBengal"/>  <property name="country" value="India"/>  </bean>    <bean id="address2" **class**="com.Address">  <property name="addLine" value="Test address2"/>  <property name="city" value="Kolkata"/>  <property name="state" value="WestBengal"/>  <property name="country" value="India"/>  </bean>    </beans> |

**Test.java**

|  |
| --- |
| **package** com;    **import** java.util.List;    **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;    **public** **class** Test {  **public** **static** **void** main(String[] args) {  *//Get ApplicationContext using spring configuration file.*  **ApplicationContext context =**  **new ClassPathXmlApplicationContext("applicationContext.xml");**    *//Get Student bean object from ApplicationContext instance.*  Student student = (Student) context.getBean("student");    *//Process Student Object.*  System.out.println("Student info: ");  System.out.println("Name: " + student.getName());  System.out.println("RollNo: " + student.getRollNo());  System.out.println("Class: " + student.getClassName());    *//Get Address from Student Object.*  List<Address> studentAddressList = student.getAddress();    *//Declare program counter.*  **int** addressCounter = 1;    *//Iterate Address List.*  **for** (Address studentAddress : studentAddressList) {  *//Process Address Object.*  System.out.println("Student Address " +addressCounter+ ": ");  System.out.println("Address Line: "+studentAddress.getAddLine());  System.out.println("City: " + studentAddress.getCity());  System.out.println("State: " + studentAddress.getState());  System.out.println("Country: " + studentAddress.getCountry());  addressCounter++;  }  }  } |

## Output:

|  |
| --- |
| Student info:  Name: Papia  RollNo: BTech/12  **Class**: BTech  Student Address 1:  Address Line: Test address1  City: Kolkata  State: WestBengal  Country: India  Student Address 2:  Address Line: Test address2  City: Kolkata  State: WestBengal  Country: India |

**spring-config.xml**

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

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<bean id="stringId" class="java.lang.String">

</bean>

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

<property name="studentProperties">

<props>

<prop key="Name">Kalai</prop>

<prop key="Course">JAVA</prop>

</props>

</property>

</bean>

</beans>

**Student.java**

package com;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

public class Student {

private Properties studentProperties = null;

public void setStudentProperties(Properties studentProperties) {

this.studentProperties = studentProperties;

}

public Properties getStudentProperties() {

return studentProperties;

}

}

**StudentMain.java:**

package com;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Properties;

import java.util.Set;

import org.springframework.beans.factory.BeanFactory;

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

import org.springframework.core.io.FileSystemResource;

public class StudentMain {

public static void main(String a[]) {

BeanFactory factory = new XmlBeanFactory(new FileSystemResource(

"config\\spring-config.xml"));

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

Properties studentProperties = student.getStudentProperties();

Iterator iterator3 = studentProperties.values().iterator();

System.out.println("Properties example in Spring");

System.out.println();

while (iterator3.hasNext()) {

Object obj = iterator3.next();

System.out.println(obj);

System.out.println();

}

}

}

Map:

Emp.java;

package com;

import java.util.List;

//bean that uses other resourse like other bean,collection framework

import java.util.Map;

public class Emp {

public Map<String,String> empMapNoRef;

public Map<String,Employee> empMapValueRef;

public Map<Employee,String> empMapKeyRef;

public Map<Employee,Employee> empMapBothRef;

public Map<String, String> getEmpMapNoRef() {

return empMapNoRef;

}

public void setEmpMapNoRef(Map<String, String> empMapNoRef) {

this.empMapNoRef = empMapNoRef;

}

public Map<String, Employee> getEmpMapValueRef() {

return empMapValueRef;

}

public void setEmpMapValueRef(Map<String, Employee> empMapValueRef) {

this.empMapValueRef = empMapValueRef;

}

public Map<Employee, String> getEmpMapKeyRef() {

return empMapKeyRef;

}

public void setEmpMapKeyRef(Map<Employee, String> empMapKeyRef) {

this.empMapKeyRef = empMapKeyRef;

}

public Map<Employee, Employee> getEmpMapBothRef() {

return empMapBothRef;

}

public void setEmpMapBothRef(Map<Employee, Employee> empMapBothRef) {

this.empMapBothRef = empMapBothRef;

}

}

**spring-config.xml**

<bean id="emp1" class="com.Employee">

<property name="id" value="1" />

<property name="name" value="Saswata Dhar" />

<property name="address" value="S.C Paul Road" />

</bean>

<bean id="emp2" class="com.Employee">

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

<property name="name" value="Asim Das" />

<property name="address" value="Dumdum Road" />

</bean>

<bean id="emp3" class="com.Employee">

<property name="id" value="3" />

<property name="name" value="Prasanta Dutta" />

<property name="address" value="Jadavpur Road" />

</bean>

<bean id="eMap" class="com.Emp">

<property name="**empMapNoRef**">

**<map>**

**<entry key="1" value="Saswata"></entry>**

**<entry key="2" value="Asim"></entry>**

**<entry key="3" value="Prasanta"></entry>**

**</map>**

</property>

<property name="**empMapValueRef**">

**<map>**

**<entry key="1" value-ref="emp1"></entry>**

**<entry key="2" value-ref="emp2"></entry>**

**<entry key="3" value-ref="emp3"></entry>**

**</map>**

</property>

<property name="**empMapKeyRef**">

**<map>**

**<entry key-ref="emp1" value="Saswata"></entry>**

**<entry key-ref="emp2" value="Asim"></entry>**

**<entry key-ref="emp3" value="Prasanta"></entry>**

**</map>**

</property>

<property name="**empMapBothRef**">

**<map>**

**<entry key-ref="emp1" value-ref="emp1"></entry>**

**<entry key-ref="emp2" value-ref="emp2"></entry>**

**<entry key-ref="emp3" value-ref="emp3"></entry>**

**</map>**

</property>

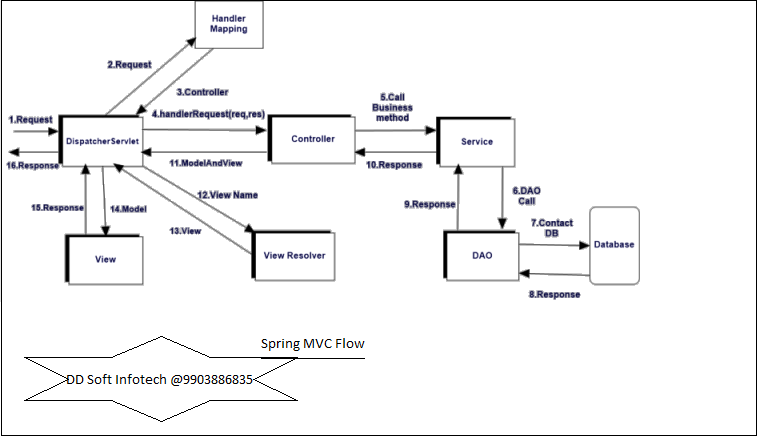
</bean>

Spring MVC Flow:

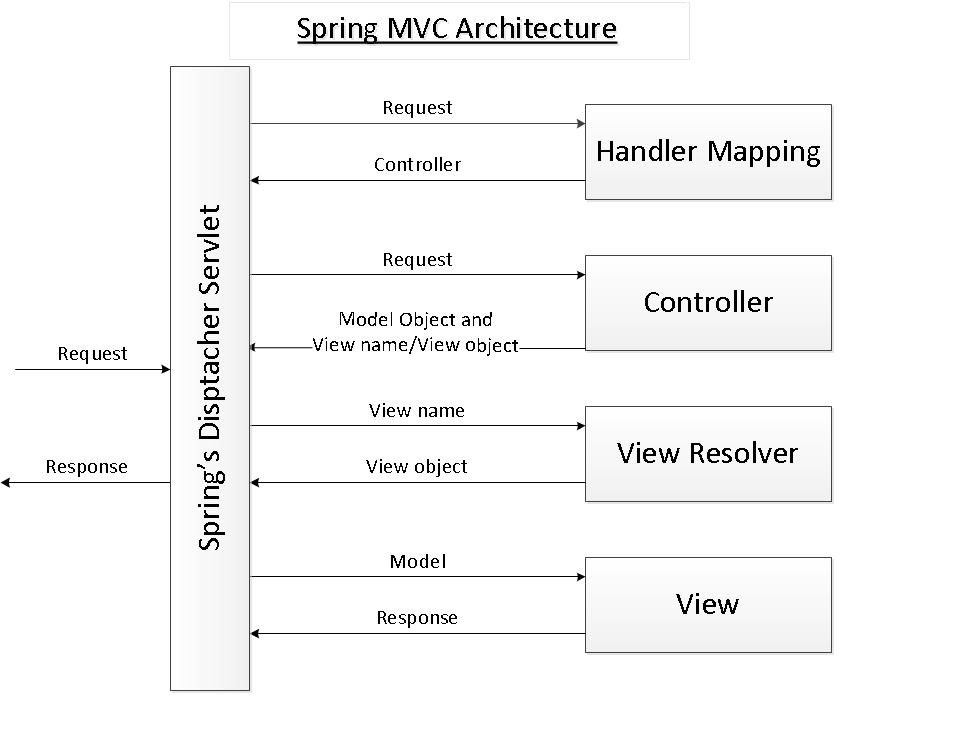
Spring’s MVC module is based on **front controller** design pattern followed by **MVC design pattern**. All the incoming requests are handled by the single servlet named **DispatcherServlet** which acts as the **front controller**in Spring’s MVC module. The **DispatcherServlet** then refers to the  **HandlerMapping** to find a controller object which can handle the request. **DispatcherServlet** then dispatches the request to the controller object so that it can actually perform the business logic to fulfil the user request. (**Controller** may delegate the responsibility to further application objects known as service objects). The controller returns an encapsulated object containing the model object and the view object (or a logical name of the view). In Spring’s MVC, this encapsulated object is represented by class **ModelAndView**. In case **ModelAndView** contains the logical name of the view, the  **DispatcherServlet** refers the **ViewResolver** to find the actual View object based on the logical name. **DispatcherServlet** then passes the model object to the view object which is then rendered to the end user.

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.

* Client(Browser) requests for a Resource in the web Application.
* The Spring front controller i.e, **DispatcherServlet** first receives the request.
* **DispatcherServlet** consults the **HandlerMapping** to identify the particular controller for the given URL.
* **HandlerMapping** identifies the controller for the given request and sends to the **DispatcherServlet**.
* **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.
* **Controller** will call the business method according to business requirement.
* **Service class** will call the **DAO class** method for business data.
* **DAO** interacts with **DB** to get data.
* **DAO** returns same data to service.
* Fetched data will be processed according to business requirement and return results to **Controller**.
* **The Controller** returns the Model and View in the form of Object back to the Controller i.e, **DispatcherServlet**.
* 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.
* **ViewResolver** selected view is rendred back to the **DispatcherServlet**.
* **DispatcherServlet** consult the particular view with the **model**.
* **View** executes and returns HTML output to the **DispatcherServlet**.
* **DispatcherServlet** will sends the output to the **Browser**.







MVC design pattern

This design pattern helps us develop loosely coupled application by segregating various concerns into different layers. MVC design pattern enforces the application to be divided into three layers, Model, View and Controller.

**Model:** This represents the application data.

**View:** This represents the application’s user interface. View takes model as the input and renders it appropriately to the end user.

**Controller:** The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

<http://www.codejava.net/frameworks/spring/understanding-spring-mvc?start=1>

https://www.dineshonjava.com/spring-mvc-form-handling-example/

<http://www.wideskills.com/spring/spring-mvc-framework>

<http://javawebtutor.com/articles/spring/spring-mvc-tutorial.php>

Defining Spring Controller

Following is the controller class which is mapped for /users. This class acts as a handler class in spring mvc. It handles the request that is mapped for /users. It processes the requests and return model and view to the front controller.

The ***@Controller*** annotation is used to mark any java class as a controller class

*@****RequestMapping*** *:* This annotation is used at both the class and method level. The **@RequestMapping** annotation is used to map web requests onto specific handler classes and handler methods. When **@RequestMapping** is used on the class level, it creates a base URI for which the controller will be used. When this annotation is used on methods, it will give you the URI on which the handler methods will be executed. From this, you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

***@*RequestMapping Basics**

Let’s discuss with a simple example – mapping an HTTP request to a method using some straightforward criteria.

**1.1 *@*RequestMapping – by Path**

***@*RequestMapping(value = "/doj/spring")**

public String getDOJSpringCourse(){

return "Get DOJ Course for Spring";

}

***1.2* @RequestMapping – the HTTP Method**

There is no by default HTTP method mapped by a @RequestMapping – so it maps to any type of HTTP request; we have to add the method :

**@RequestMapping(value = "/doj/spring", method = RequestMethod.POST)**

public String postDOJSpringCourse(){

return "Post DOJ Course for Spring";

}

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of **@RequestMapping** with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Using the ***@Autowired*** annotation the container can wire your beans automatically. By default **autowire** is done **by type**.

1. Normally we specify dependencies with the help of constructor arguments or property set. But with the help of auto wire, we don’t need to specify them they are automatically resolved.
2. It reduces the amount of code we write to specify dependencies.

DispatcherServlet and ContextLoaderListener:

**DispatcherServlet** is basically the front controller in the Spring MVC application as it loads the spring bean configuration file and initializes all the beans that have been configured. If annotations are enabled, it also scans the packages to configure any bean annotated with **@Component**, **@Controller**, **@Repository** or **@Service** annotations.

**ContextLoaderListener**, on the other hand, is the listener to start up and shut down the WebApplicationContext in Spring root.

Explain the role of InternalResourceViewResolver:

**InternalResourceViewResolver** is one of the implementations of the **ViewResolver** interface that allows you to view the **page** directory and **suffix** locations through the bean properties.

Spring Boot

**Spring Boot** is a Spring framework module which provides developers to create spring production grade applications and services with very less effort. In real-time spring application includes writing many XMLconfigurations, server setting, adding dependencies…etc. But with spring Boot we can avoid all these boilerplate code, writing XML configurations and annotations. We can create a real-time production ready applications with in minutes.

Spring Boot comes with inbuilt server, we no longer have to use any external servers like Tomcat, Glass-fish or anything else, so don’t need to deploy WAR files

We have to use either **Maven or Gradle** build tool to work with Spring Boot. Just don’t worry, I will explain about the Maven/Gradle configurations while giving the example and one more thing is, **Spring Boot** provides command line interface tool to develop/test the Spring Boot applications from the command prompt easily.

Spring Boot supports embedded containers in allowing web applications to be able to run independently without deploying on Web Server.

Spring Boot can be explained simply by the illustration below:

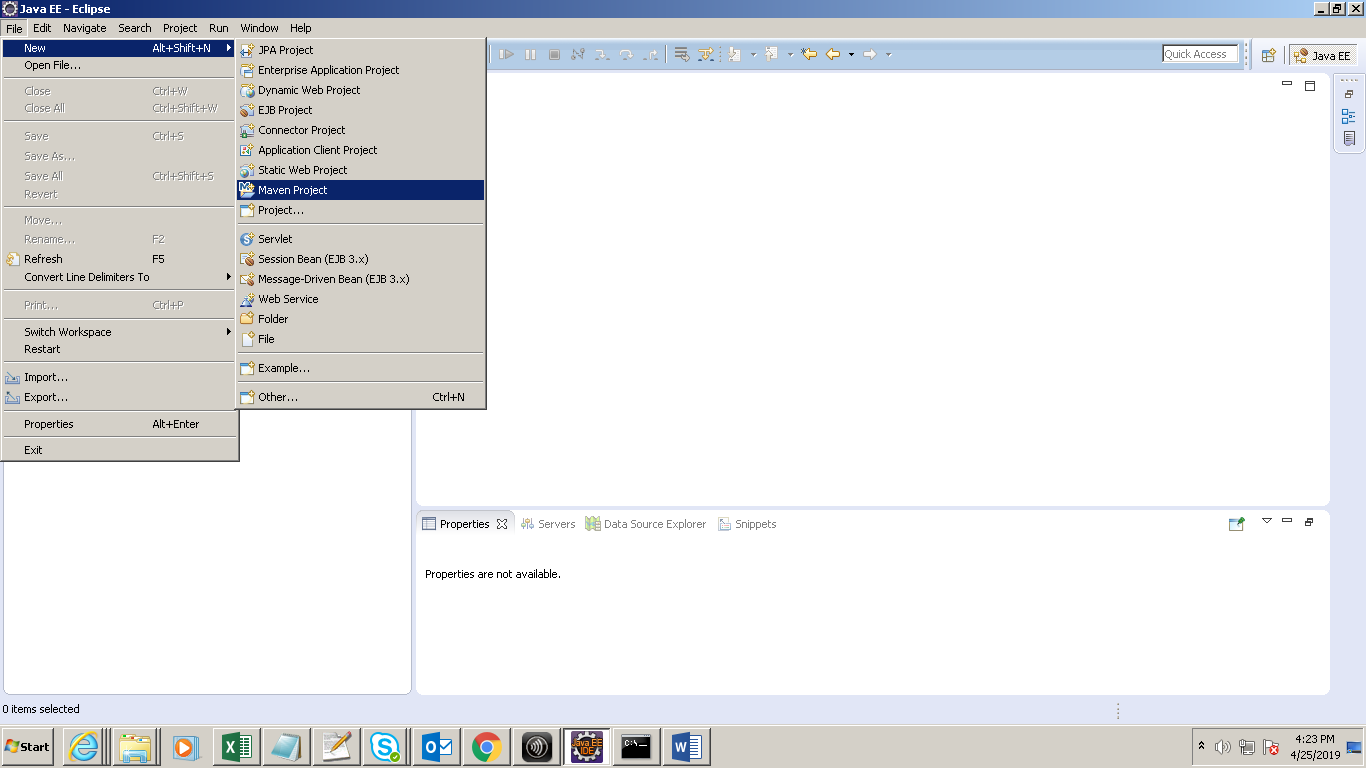


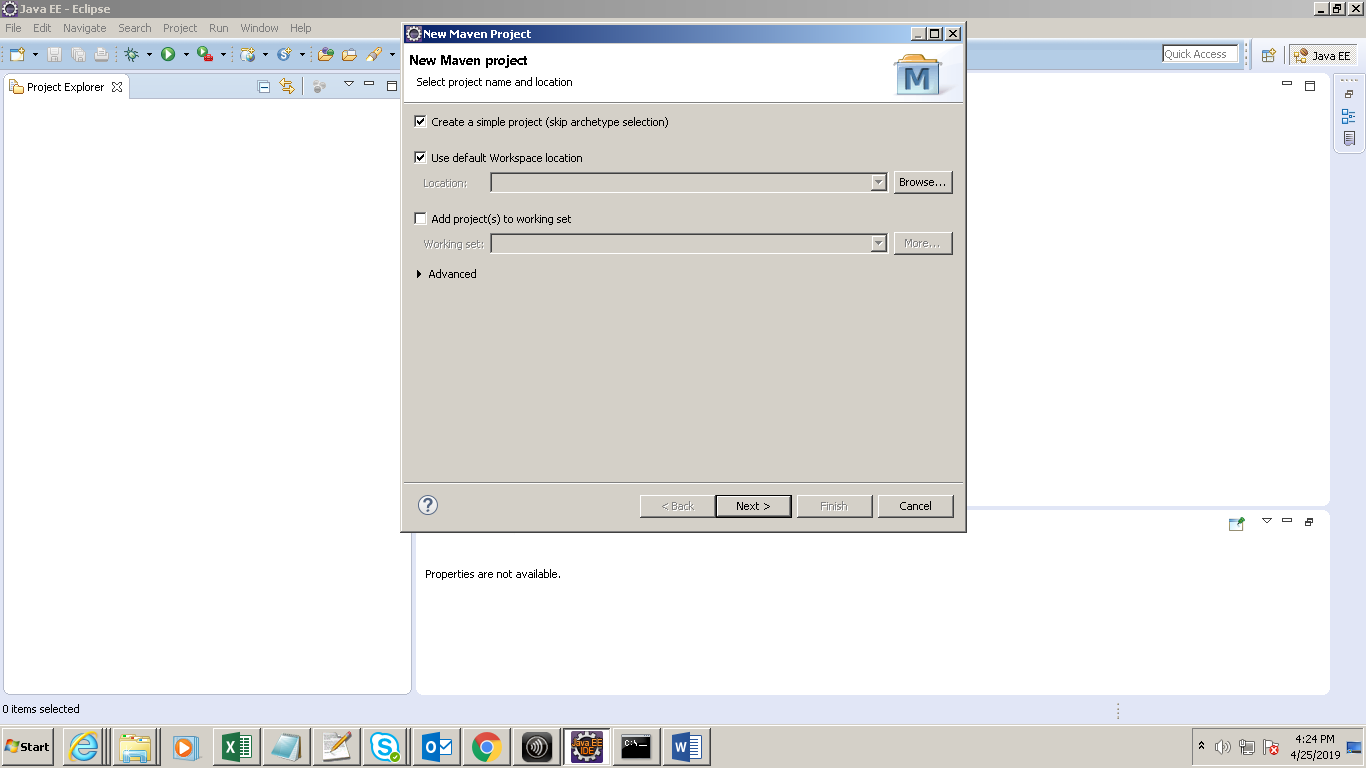
**Below is the benefits of Spring Boot:**

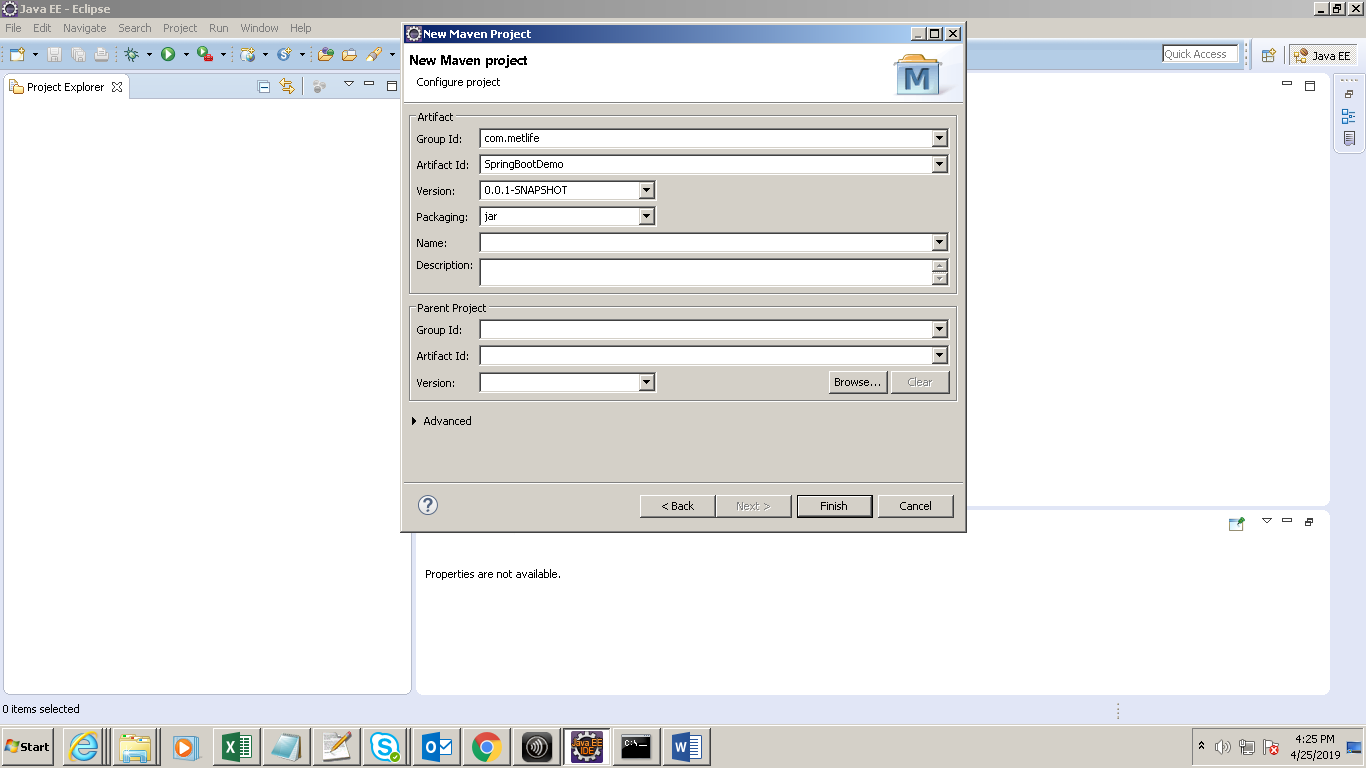
* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
* It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
* It provides lots of plugins to work with embedded and in-memory Databases very easily

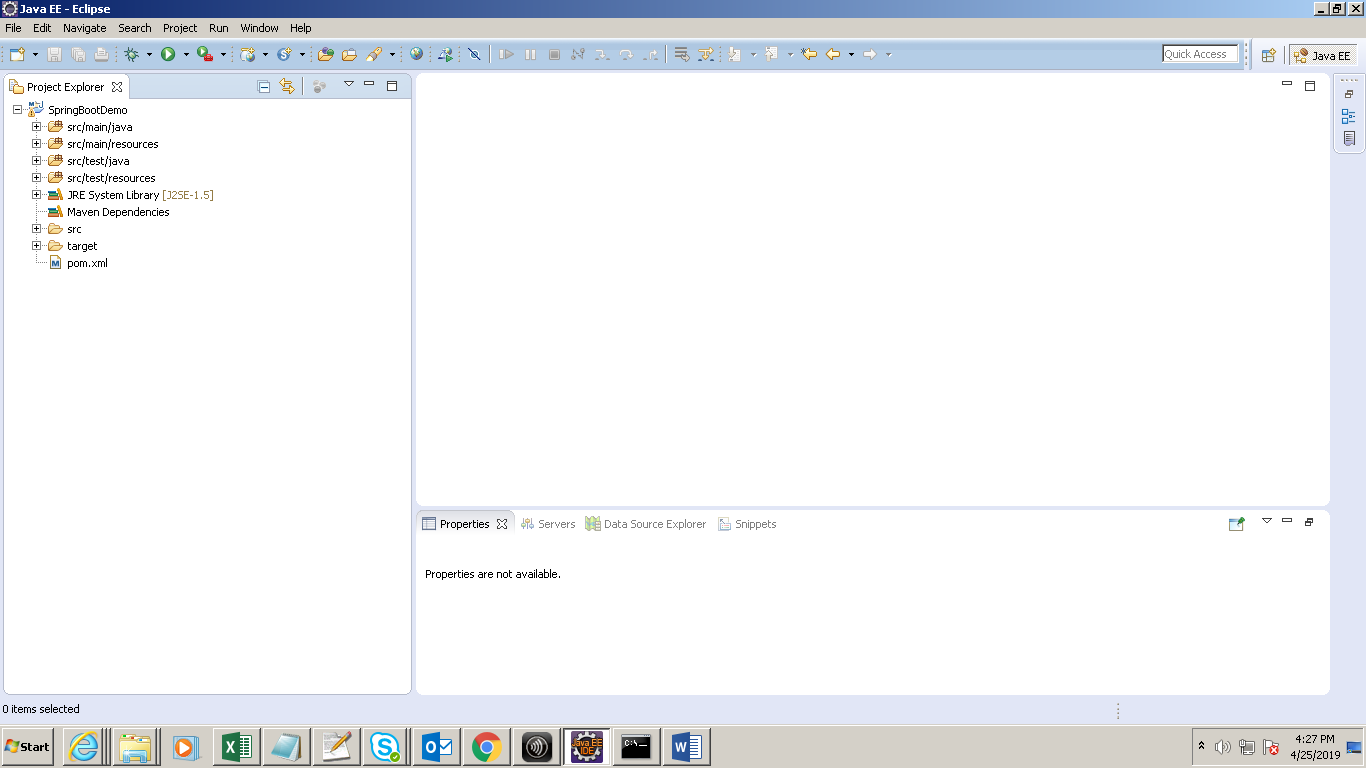
Spring Boot Steps to create project:

File-New->Maven Project









As you can see in maven project structure, the **default java compiler version ( i.e. source and target setting ) is 1.5**. To change the default settings, add the following snippet to **pom.xml**.

<properties>

<java.version>1.8</java.version>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

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

<version>3.6.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

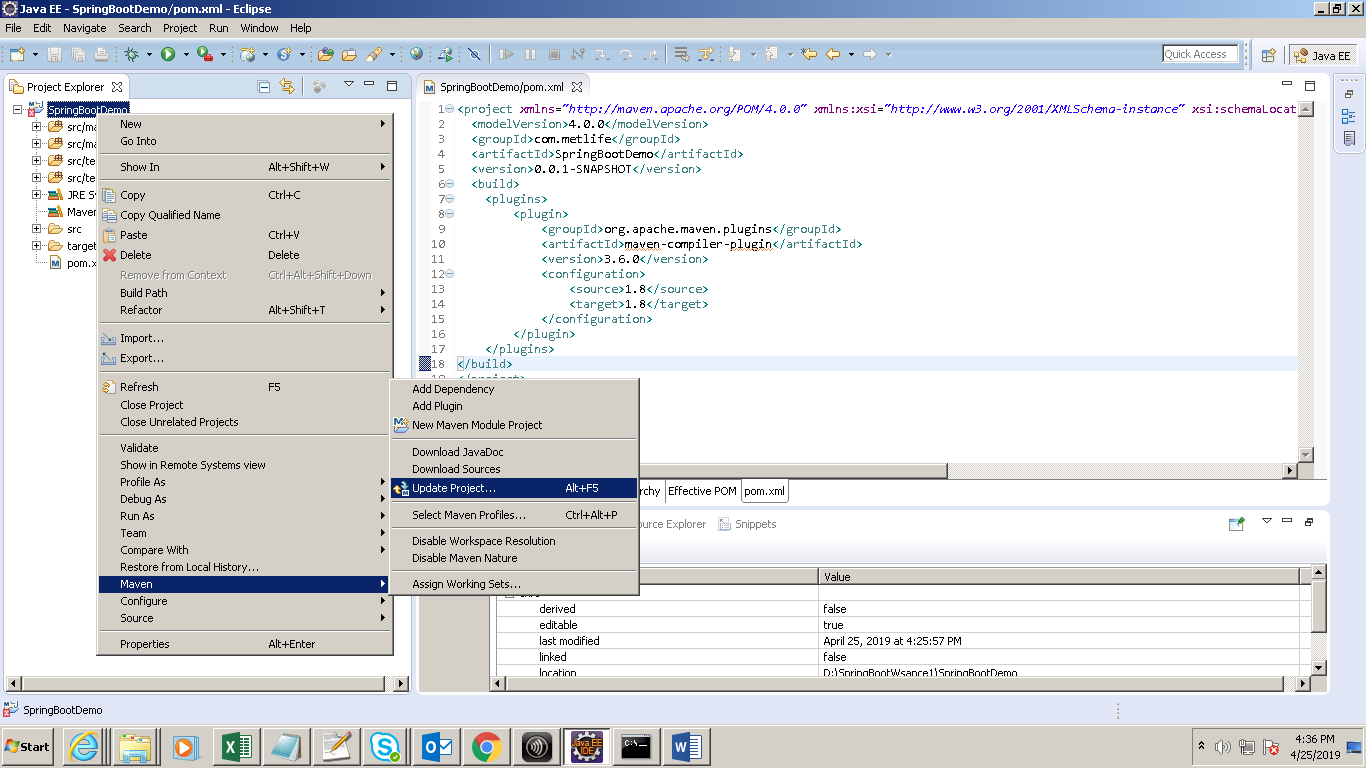
</configuration>

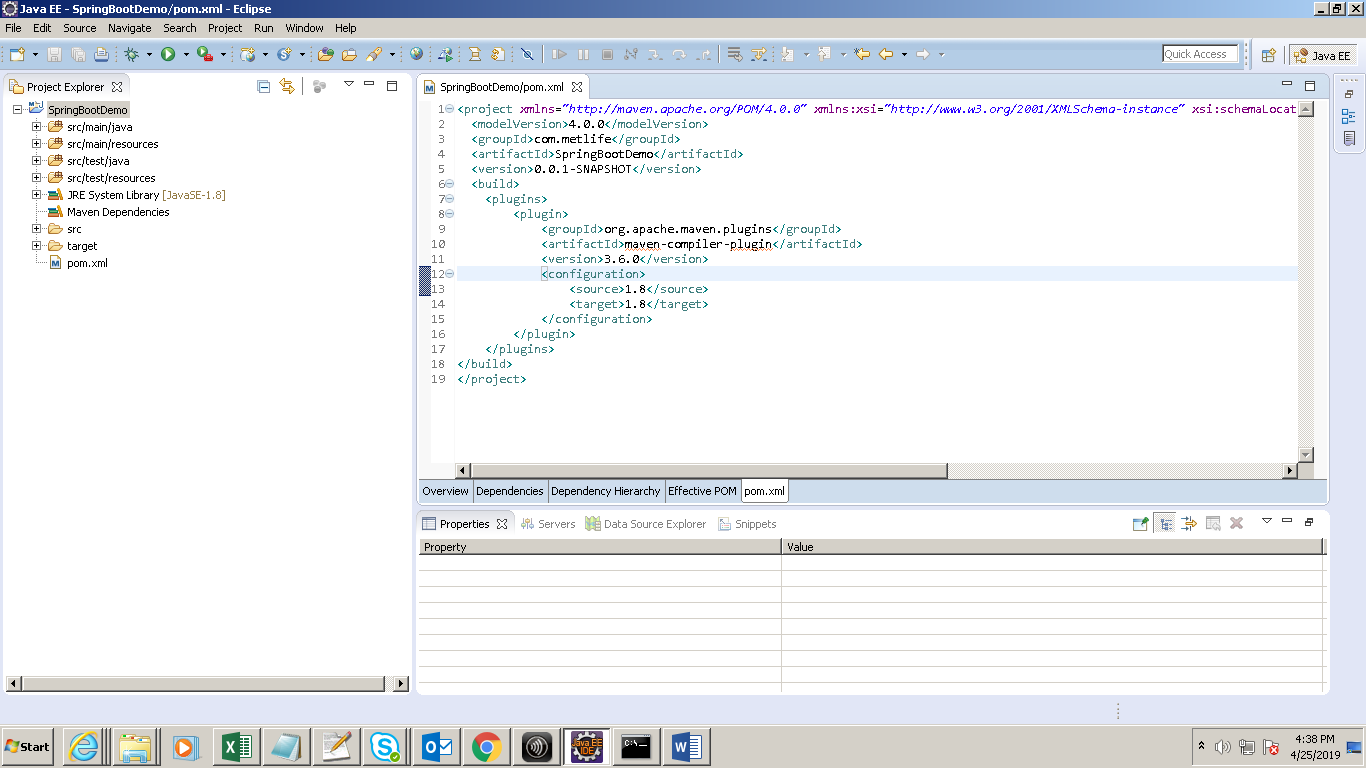
</plugin>

</plugins>

</build>

After changes in pom.xml file, update the maven project. To update maven project **right click** **on maven-project → Maven → Update Project**.





Add the below thing in pom.xml:

**<parent>**

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

**<artifactId>spring-boot-starter-parent</artifactId>**

**<version>1.5.8.RELEASE</version>**

**</parent>**

**<dependencies>**

**<dependency>**

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

**<artifactId>spring-boot-starter</artifactId>**

**</dependency>**

**</dependencies>**

**Spring Boot** provides the various ["**Starters**"](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#d0e1559) for building Spring Boot based java application.  The **spring-boot-starter-parent** is a special type of starter, which is used as a parent in pom.xml file of any kind of Spring Boot application.

Spring Boot starters are templates that contain a collection of all the relevant transitive dependenciesthat are needed to start a particular functionality. For example, If you want to create a Spring WebMVC application then in a traditional setup, you would have included all required dependencies yourself. It leaves the chances of version conflict which ultimately result in more runtime exceptions.

With String boot, to create MVC application all you need to import is **spring-boot-starter-web** dependency.

The **spring-boot-starter-parent** provides the common configuration such default java compiler level, plugin configuration, UTF-8 source encoding, dependency management etc.

There are various starters such as **spring-boot-starter-web** for building Spring web application, **spring-boot-starter-web-services** for  building Spring web service application.

I have added **spring-boot-starter-parent**, **spring-boot-starter-web** and I want to show Spring Boot tutorials in Java 8, so I have added java version

What is spring-boot-starter-parent? actually this is an existing project given by spring team which contains Spring Boot supporting configuration data (remember just configuration data, it wont download any jars), we have added this in a **<parent>** tag means, we are instructing Maven to consider our TestMain project as a child to it, wait for a second, I will show you practically why we have to add spring-boot-starter-parent as parent

In the dependencies, I have added spring-boot-starter-web for web module

Now **right click on the application > Maven > Update Project**, If you now observe the directory structure of the project, it will create a new folder named “Maven Dependencies” which contains all supporting. jars to run the Spring Boot application and the Java version also changed to 1.8

Note: put your java class in some package, it is mandatory.\*  If you haven’t created a package it gives the following error while running your Spring Boot application.

Your **ApplicationContext** is unlikely to start due to a **@ComponentScan** of the default package.

just remember, for every spring boot application we have to create a main class and that need to be annotate with **@SpringBootApplication** and bootstrap it.

**@SpringBootApplication** annotation, means this is the starting point for our Spring Boot application

To bootstrap spring boot application, create a main class annotated with **@SpringBootApplication** annotation.

**TestMain.java**

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class TestMain {

public static void main(String[] args) {

SpringApplication.run(TestMain.class, args);

}

}

**POM.xml:**

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

<modelVersion>4.0.0</modelVersion>

<groupId>com.metlife</groupId>

<artifactId>SpringBootDemo</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<java.version>1.8</java.version>

</properties>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.8.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

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

<version>3.6.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**@ComponentScan({"com.pack1.springboot.basics","com.pack1.springboot.somethingelse"})**

**@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration**

The following are the parameters accepted in the **@SpringBootApplication** annotation:

**exclude:** Exclude the list of classes from the auto configuration.

**excludeNames:** Exclude the list of fully qualified class names from the auto configuration. This parameter added since spring boot 1.3.0.

**scanBasePackageClasses:** Provide the list of classes that has to be applied for the @ComponentScan.

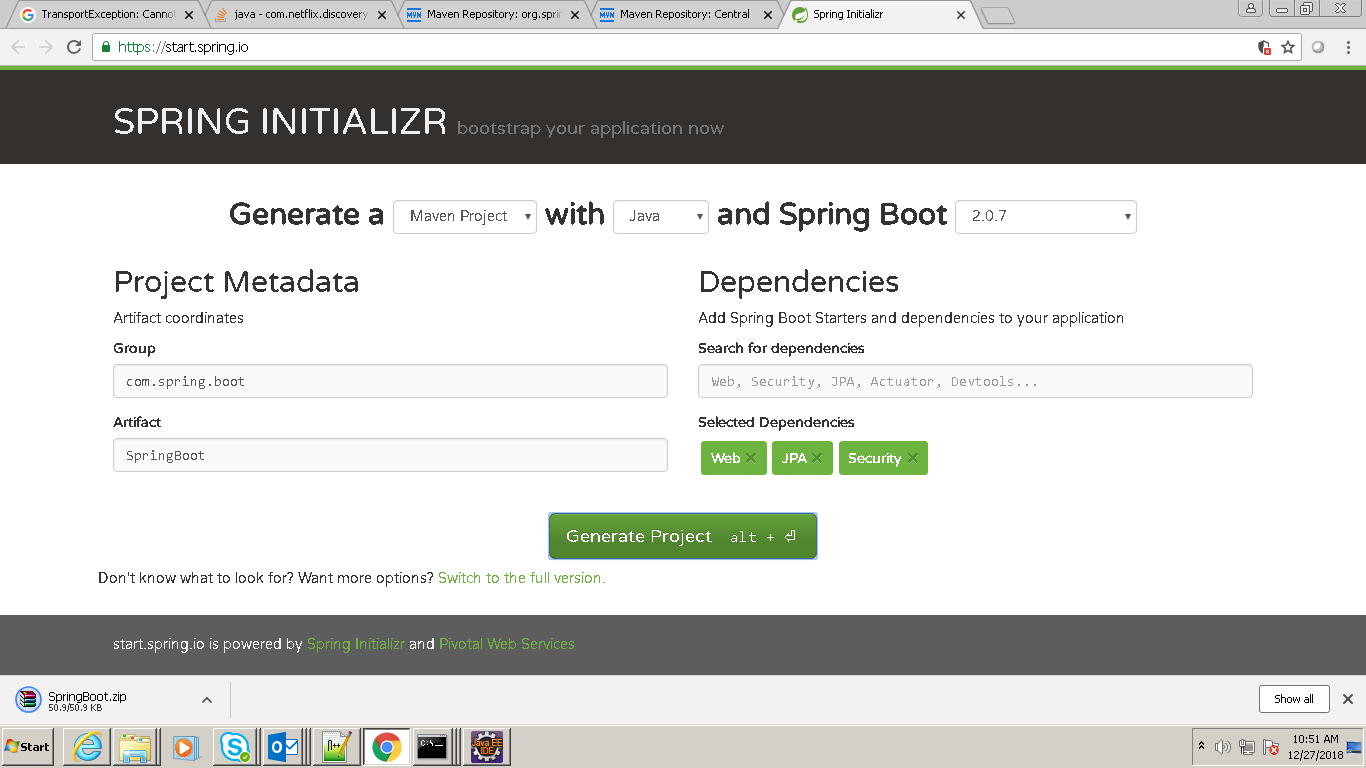
**scanBasePackages** Provide the list of packages that has to be applied for the @ComponentScan. This parameter added since spring boot 1.3.0.

**Structure of code:**

While allowing you a lot of freedom, there are some basic rules worth following then laying out your source code.

* Avoid using the default package. Make sure that everything (including your entry point) lives in a well-named package. This way you will avoid surprises related to wiring and component scan.
* **Keep your *Application.java*(your entry Class) in the top-level source directory.**
* I recommend keeping Controllers and Services together in modules that are oriented around functionality, but this is optional. Some very good developers recommend keeping all Controllers together. Stick to one style!

Spring Boot Steps: Using CLI



Go to the site ([**https://start.spring.io/**](https://start.spring.io/)) and select the version:**2.0.7** and add the dependencies

And write the group name and artifact and click on

Now **create a workspace in eclipse and extract the zip file and go to the command prompt and type the below command**:

**D:\SpringBootWsapce\SpringBatchDemo>mvn clean install**

**After that**

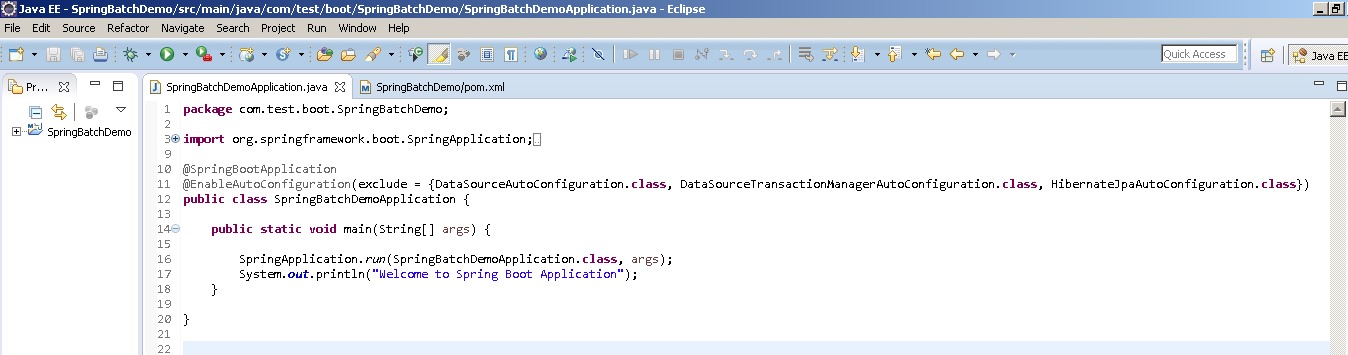
**D:\SpringBootWsapce\SpringBatchDemo>mvn eclipse:eclipse**

And then import the project as an existing maven project

**Solution for DB Issue:**

Add the following line:

**@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class, DataSourceTransactionManagerAutoConfiguration.class, HibernateJpaAutoConfiguration.class})**



**Spring Boot is mainly used to expose the service**

Core Spring Framework Annotations

### @Autowired

This annotation is applied on fields, setter methods, and constructors. The **@Autowired** annotation injects object dependency implicitly.

When you use **@Autowired** on fields and pass the values for the fields using the property name, Spring will automatically assign the fields with the passed values.

You can even use **@Autowired** on private properties, as shown below. (This is a very poor practice though!)

public class Customer {

**@Autowired**

private Person person;

private int type;

}

When you use **@Autowired** on setter methods, Spring tries to perform the by Type autowiring on the method. You are instructing Spring that it should initiate this property using setter method where you can add your custom code, like initializing any other property with this property.

public class Customer {

private Person person;

**@Autowired**

public void setPerson (Person person) {

this.person=person;

}

}

Consider a scenario where you need instance of class A, but you do not store A in the field of the class. You just use A to obtain instance of B, and you are storing B in this field. In this case setter method autowiring will better suite you. You will not have class level unused fields

When you use **@Autowired** on a constructor, constructor injection happens at the time of object creation. It indicates the constructor to autowire when used as a bean. One thing to note here is that only one constructor of any bean class can carry the **@Autowired** annotation.

**@Component**

public class Customer {

private Person person;

**@Autowired**

public Customer (Person person) {

this.person=person;

}

}

### @Qualifier

This annotation is used along with **@Autowired** annotation. When you need more control of the dependency injection process, **@Qualifier** can be used. **@Qualifier** can be specified on individual constructor arguments or method parameters. This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

Consider an example where an interface BeanInterface is implemented by two beans BeanB1 and BeanB2.

**@Component**

public class BeanB1 implements BeanInterface {

//

}

**@Component**

public class BeanB2 implements BeanInterface {

//

}

Now if BeanA **autowires** this interface, Spring will not know which one of the two implementations to inject.  
One solution to this problem is the use of the **@Qualifier** annotation.

**@Component**

public class BeanA {

**@Autowired**

**@Qualifier("beanB2")**

private BeanInterface dependency;

...

}

With the **@Qualifier** annotation added, Spring will now know which bean to autowire where beanB2 is the name of BeanB2.

**@Configuration**

This annotation is used on classes which define beans. **@Configuration** is an analog for XML configuration file – it is configuration using Java class. Java class annotated with **@Configuration** is a configuration by itself and will have methods to instantiate and configure the dependencies.

**@Configuration**

public class DataConfig{

**@Bean**

public DataSource source(){

DataSource source = new OracleDataSource();

source.setURL();

source.setUser();

return source;

}

**@Bean**

public PlatformTransactionManager manager(){

PlatformTransactionManager manager = new BasicDataSourceTransactionManager();

manager.setDataSource(source());

return manager;

}

}

### @ComponentScan

This annotation is used with **@Configuration** annotation to allow Spring to know the packages to scan for annotated components. **@ComponentScan** is also used to specify base packages using basePackageClasses orbasePackage attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

Checkout this [post](https://springframework.guru/spring-component-scan/) for an in depth look at the Component Scan annotation.

### @Bean

This annotation is used at the method level. **@Bean** annotation works with **@Configuration** to create Spring beans. As mentioned earlier, **@Configuration** will have methods to instantiate and configure dependencies. Such methods will be annotated with **@Bean**. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

Here is an example:

**@Configuration**

public class AppConfig{

**@Bean**

public Person person(){

return new Person(address());

}

**@Bean**

public Address address(){

return new Address();

}

}

### @Component

This annotation is used on classes to indicate a Spring component. The **@Component** annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

### @Controller

The **@Controller**  annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

### @Service

This annotation is used on a class. The **@Service** marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs. This annotation is a specialized form of the **@Component** annotation intended to be used in the service layer.

### @Repository

This annotation is used on Java classes which directly access the database. The **@Repository** annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the **@Repository** there is a handler for that exception and there is no need to add a try catch block.

## Spring Boot Annotations

### @EnableAutoConfiguration

This annotation is usually placed on the main application class. The **@EnableAutoConfiguration** annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

### @SpringBootApplication

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the **@SpringBootApplication** must be kept in the base package. The one thing that the **@SpringBootApplication** does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with **@SpringBootApplication** in com.example then **@SpringBootApplication** will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

The **@SpringBootApplication** is a convenient annotation that adds all the following:

* **@Configuration**
* **@EnableAutoConfiguration**
* **@ComponentScan**

### @Controller

This annotation is used on Java classes that play the role of controller in your application. The **@Controller** annotation allows auto detection of component classes in the class path and auto-registering bean definitions for them. To enable auto detection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with **@Controller** is capable of handling multiple request mappings.

This annotation can be used with Spring MVC and Spring WebFlux.

### @RequestMapping

This annotation is used both at class and method level. The **@RequestMapping** annotation is used to map web requests onto specific handler classes and handler methods. When **@RequestMapping** is used on class level it creates a base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of **@RequestMapping** with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:

**@Controller**

**@RequestMapping("/welcome")**

public class WelcomeController{

**@RequestMapping(method = RequestMethod.GET)**

public String welcomeAll(){

return "welcome all";

}

}

In this example only GET requests to /welcome is handled by the welcomeAll() method.

This annotation also can be used with Spring MVC and Spring WebFlux.

The **@RequestMapping**  annotation is very versatile

**@CookieValue**

This annotation is used at method parameter level. **@CookieValue** is used as argument of request mapping method. The HTTP cookie is bound to the **@CookieValue** parameter for a given cookie name. This annotation is used in the method annotated with **@RequestMapping**.

**@RequestParam**

This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the **@RequestMapping** annotation you can use the **@RequestParam** annotation to retrieve the URL parameter and map it to the method argument. The @**RequestParam** annotation is used to bind request parameters to a method parameter in your controller.

**@RestController**

This annotation is used at the class level. The **@RestController** annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add **@ResponseBody** to all the RequestMapping method. It means that you no more use view-resolvers or send html in response. You just send the domain object as HTTP response in the format that is understood by the consumers like JSON.

**@RestController**  is a convenience annotation which combines **@Controller**  and **@ResponseBody** .

**@SessionAttribute**

This annotation is used at method parameter level. The **@SessionAttribute** annotation is used to bind the method parameter to a session attribute. This annotation provides a convenient access to the existing or permanent session attributes.

**@Session Attributes**

This annotation is applied at type level for a specific handler. The **@SessionAtrributes** annotation is used when you want to add a JavaBean object into a session. This is used when you want to keep the object in session for short lived. **@SessionAttributes** is used in conjunction with **@ModelAttribute**.  
Consider this example.

**@ModelAttribute("person")**

public Person getPerson(){}

// within the same controller as above snippet

**@Controller**

**@SeesionAttributes(value="person", types={Person.class})**

public class PersonController{}

The **@ModelAttribute** name is assigned to the **@SessionAttributes** as value. The **@SessionAttributes** has two elements. The value element is the name of the session in the model and the types element is the type of session attributes in the model.