**Spring IOC**

Three types of Spring Containers

1. Core : BeanFactory
2. J2EE : ApplicationContext/ ConfigurableApplicationContext
3. Web Container :

Container will following tasks

1. Create instances for POJO classes.
2. Manage life cycles of POJO Classes.
3. Dependency Injection into POJO Classes.

Spring.xml

<DTD>

<beans>

<beans id=”” class=””/>

<beans id=”” class=””/>

</beans>

1st User

IOC

SAX Parser

getBean(“t”)

2nd User

getBean(“t”) core

SAX parser in IOC spring core will check the well formedness of spring.xml by using the DTD of file. If not found in compliance to dtd, an exception will be thrown it is not well formed. It will also check if it is using the tags that is valid according to the DTD if any tags being well formed not found valid then also SAX parser will throw an exception at the time of loading the application for invalid tag being used.

If user request a bean then an object of bean will be created by using getBean(”beanRef”) where beanRef is the name of bean to which bean class has been mapped. One instance it will create in JVM for 1st user request by BeanFactory. If one more user request for the same bean class then it will not create the bean but will return the same beans reference which was created at the time of first user request. In bean property spring will use the scope as ‘singleton’ by default. We can change it to prototype for each request for the bean it’ll create a new bean object.

If we use J2EE container ApplicationContext then it will create all the beans defined in the spring.xml at the time of loading the application if we have used the beans scope attribute as singleton which is same by default. In this config container is eager type meaning it will create all the beans present in the spring.xml at the time of application loading. If we change our bean scope to prototype then at the time of loading application it will not create beans and it will create a separate bean for each user request. In case of this ApplicationContext container will behave like lazy container.

After getBean() method is executed then IOC will create the Test Class instance using following

Class.forName(“Test”).getInstance();

Spring can access private constructors

IOC main features are:

1. Dependency Injection: For example a DAO Class have dependencies of Connection class, Driver Class, URL, User Name, Password to access the Databases. If our class have some dependencies like such then we can inject such type of dependencies using spring IOC. We can pass such dependencies from xml file to our classes

Business

DAO Class

TX | Sec| Mail | JMS | Quartz

View

Controller

JDBC

Dependencies Dependencies Dependencies Dependencies for:

1. DAO JDBC

Business Class 2. Transaction Drivers/ConnectionPool

3. Security Spring JDBC

4. Mail 1.JDBC Template

5. Quartz Hibernate/Ibatis

1.Session Factory

2.Hibernate Template

By Using IOC we can inject these type of dependencies to the required by the classes for its executions. We can inject these dependencies by passing the required parameters from the xml to POJO Classes.

There are two type of Dependency Injection

1. Setters
2. Constructors

Let’s consider a class

public class Test(){

private String gender;

public void setGender(gender){  
 this.gender=gender;

}

Public void sayHello(String name){  
 System.out.println(“Hello ” + gender + name)

}

}

Spring.xml

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<property name=”gender” value=”Mr”/> or <property name=”gender”>

</bean> <value>Mr</value>

</beans> </property>

A object of Test class is created by getBean() method by IOC by using its default constructor and hence will store the gender value as ‘Null’. After reading the property tag nested inside bean tag it will call the setter method setGender() of bean Test to pass the value Mr to the gender property of the bean class and gender will be assigned with Mr. **If same attribute is configured for two times the we’ll get the duplicate value exceptions.**

public class Test(){

private String gender;

private int age

public Test(String gender){  
 this.gender=gender;

}

public Test(int age){  
 this.age=age;

}

public Test(String gender, int age){

this.gender=gender;

this.age=age;

}

public void setGender(gender){  
 this.gender=gender;

}

Public void sayHello(String name){  
 System.out.println(“Hello ” + gender + name)

}

}

We can pass the values of bean property by parameterized constructors also. In that case we have to configure our spring.xml file as follows

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<constructor-arg value=”Mr” type=”String”/> or <constructor-arg >

<contructor-arg value=”25” type=”int”/> <value>Mr</value>

</constructor-arg>

</bean>

</beans>

Multiple parameter constructor configuration into the spring.xml

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<constructor-arg value=”25” type=”int”/> //if order is not required

<constructor-arg value=”Mr” type=”String”/>

<bean>

</beans>

<DTD>

Or we can use index also which defines the order in which the parameters are appearing in the constructors

<beans>

<bean id=”t” class=”beans.Test”>

<constructor-arg value=”25” index=”1”/> //if order is not required

<constructor-arg value=”Mr” index=”0”/>

<bean>

</beans>

public class Test(){

private String gender;

private int age;

private String email;

public Test(String gender){  
 this.gender=gender;

}

public Test(int age){  
 this.age=age;

}

public Test(String gender, int age){

this.gender=gender;

this.age=age;

}

public Test(String gender, int age , String email){

this.gender=gender;

this.age=age;

}

public void setGender(gender){  
 this.gender=gender;

}

Public void sayHello(String name){  
 System.out.println(“Hello ” + gender + name + “. Your age is “+ age+ “ and your email is ” + email )

}

}

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<constructor-arg value=”25” index=”1”/> //if order is not required

<constructor-arg value=”Mr” index=”0”/>

<constructor-arg value=”mastermasroor@gmail.com” index=”2”/>

<constructor-arg value=”masroorsdm@gmail.com” index=”2”/> //if we want override existing

<bean> //parameter in indexes only

</beans>

Types of Data we have:

1. Primitive
2. Secondary
3. Primitive Arrays
4. Secondary Arrays
5. Complex Data Types : Collections

All these types of dependencies can be injected through spring.xml and these dependencies can be injected into the POJO classes by setters and parameterized constructors of it

If we have secondary datatypes then our spring.xml file look like

For constructors

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<constructor-arg ref=” ” index=”1” type=” ” /> //if order is not required

<bean>

</beans>

For Setters

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<parameter ref=” ” type=” ” /> //if order is not required

<bean>

</beans>

We use secondary data types in our beans which is be a Class. Consider following example

Public class Car{

Private Engine engine;

Private String carname;

Setters;

public void displayCar(){  
 System.out.println(“Car name is ” + carname);

System.out.println(“Car model year is ” +engine.getModelYear);

}

}

Public class Engine{

Private String modelYear;

//setters;

//getters;

}

We can two different xmls for each bean as follows if we want to pass the reference of the bean Engine to Car beans as

Engine engine = new Engine(“2012”)

Car car = new Car(engine, “Suzuki”)

Car.xml

<DTD>

<beans>

<bean id=”t” class=”beans.Test”>

<property name=”engine” ref=”eng”/>

<property name=”carname” value=”Suzuki”/>

</bean>

</beans>

engine.xml

<DTD>

<beans>

<bean id=”eng” class=”beans.Engine”>

<property name=”modelYear” value=”Suzuki”/>

</bean>

</beans>

Or we can combine these beans config in one xml so that a new instance of Engine is passed to Car Class in its Constructors argument only i.e. as Car (new Engine (2012), Suzuki)

spring.xml

<beans>

<bean id=”eng” class=”beans.Engine”>

<property name=”engine” >

<bean id=”eng” class=”beans.Test”>

<property name=”modelyear” value=”2015”/>

</bean>

</property>

<property name=”carname” value=”Suzuki”/>

</bean>

</beans>

For passing values to the primitive arrays and collection type we use the <value> tag nested inside <list> tag which further nested inside the <property> tag as follows

For primitive arrays and we can pass as

<DTD>

<beans>

<bean id=”t” class=”beans.Test”

<property name=”arrayA” type=”int”>

<list>

<value>1</value>

<value>2</value>

<value>3</value>

<list>

</property>

<property name=”arrayB” type=”String”>

<list>

<value>ABC</value>

<value>PQR</value>

<value>XYZ</value>

<list>

</property>

</bean>

<beans>

For secondary arrays having the refrence bean object

<DTD>

<beans>

<bean id=”t” class=”beans.Test”

<property name=”Employee” type=”int”>

<list>

<ref bean= “emp1”>

<ref bean=” emp2”>

<ref bean= “emp3“>

<list>

</property>

<property name=”Dept” type=”String”>

<list>

<ref bean=”dept1”>

<ref bean = “dept2”>

<ref bean=”dept3”>

<list>

</property>

</bean>

<beans>

For Collection classes we can pass the values as follows. Consider the Class as follows

public class Test{

private List fruits; // = new ArrayList();

private Set cricketers ; // = new HashSet();

private Map countryCapital; //= new HashMap;

private Properties drivers;

//getters

//setters

}

A list can accept all types of the data. It can accept

1. Primitives
2. Primitive arrays
3. Secondary Types
4. Collection Types

list.add(primitives);

list.add(Secondary types)

list.add( PrimitiveArrays)

list.add(Collection object)

The spring xml file to inject the values to these Collection classes will be defined as follows

<DTD>

<beans>

<bean id=”t” class=”beans.Test”

//If type not set by default will instantiate the object of ArrayList

<property name=”fruits” type=”java.util.ArrayList”>

<list>

<value>Apple</value>

<value>Banana</value>

<value>Mango</value>

………………………………..

<list>

</property>

//If type not set by Default will create an instance of LinkedHashSet Object

<property name=”cricketers” type=”java.util.HashSet”>

<set>

<value>Sachin</value>

<value>Rahul</value>

<value>Virat</value>

………………………………..

<set>

</property>

//If type not set by default will instantiate the object of LinkedHashMap

<property name=”countryCapital” type=”java.util.HashMap”>

<map>

<entry value=”India” value=”New Delhi”>

<entry value=”Argentina” value=”Buenos Aeres”>

<entry value=”Brazil” value=”Brasilia”>

………………………………..

<map>

</property>

<property name=”countryCapital” >

<props>

<prop key=”India”>New Delhi</prop>

<prop key=”Argentina”> Buenos Aeres </prop>

<prop key=” Brazil”> Brasilia </prop>

………………………………..

<props>

<property>

</bean>

<beans>

Or we can use Schema instead of dtd to write spring.xml as follows

<beans xmlns=” beans-namespace”

Xmlns=”util- namespace”>

<bean id=”t” class=”beans.Test”

//If type not set by default will instantiate the object of ArrayList

<property name=”fruits” >

<util:list list-class=”java.util.ArrayList”>

<value>Apple</value>

<value>Banana</value>

<value>Mango</value>

………………………………..

<util:list>

</property>

//If type not set by Default will create an instance of LinkedHashSet Object

<property name=”cricketers” >

<util:set set-class=”java.util.HashSet”>

<value>Sachin</value>

<value>Rahul</value>

<value>Virat</value>

………………………………..

<util:set>

</property>

//If type not set by default will instantiate the object of LinkedHashMap

<property name=”countryCapital”>

<util: map map-class=”java.util.HasMap”>

<entry key=”India” value=”New Delhi”>

<entry key=”Argentina” value=”Buenos Aeres”>

<entry key=”Brazil” value=”Brasilia”>

………………………………..

<util:map>

</property>

</bean>

<beans>

In case we have a properties file present already as

drivers.properties

URL: <http://localhost/MASDB>

UserName= db2admin5

Password= Admin@123

Public class test{

private Properties drivers;

}

Spring.xml with DTD

<DTD>

<beans>

<bean id= “t” class=”beans.Test”>

<property name=”drivers” >

<props>

<prop key=”India”>New Delhi</prop>

<prop key=”Argentina”> Buenos Aeres </prop>

<prop key=” Brazil”> Brasilia </prop>

………………………………..

<props>

<property>

<bean>

</beans>

Spring.xml with XML Scema

<DTD>

<beans xmlns: beans-namespace

xmlns: util-namespace >

<bean id= “t” class=”beans.Test”>

<property name=”drivers” >

//It wull read the properties directly from drivers.properties file

<util:property location:”classpath:resources/drivers.propeties/>

<property>

<bean>

</beans>

Dependency Injection can be done in two types

1. Setters
2. Parameterized Constructors

In case of setter dependency Injection we need not be give the value for the properties but for Parameterized constructors compulsory we need to provide the values. In order that values are passed for Setter dependency injection we have a concept called dependency checking. its main aim is making setter dependency injection mandatory.

We have use attribute dependency-checking=” ” bean tag. By default this value is none. It can take values as none, simple, objects, all

1. None: Setter Injection will not be enforced to pass the value.
2. Simple: If we want primitive setter injection mandatorily pass the value for it.
3. Object: if we want Secondary Type (Object) setter Dependency injection mandatorily pass the value for it.
4. All: If we want both type of injection mandatorily pass the value for it

This approach will be useful when we have only one attribute in our class and that property value we want to be compulsorily pass through Setter Injection but if we have more attributes say 10 and we want only 2 of them to be injected through the setters injection compulsorily then this approach might not be useful. Consider following example

Public class DAO{

Private String driverclass

Private String url;

Private String user;

Private String pass;

//Setters

Public getConnection(){  
 //getting connection

}

}

Here in this example we compulsorily need driver class and url to connect to the database but user and password are not compulsory attributes as connection to the database can happen without these 2 attributes also in some cases. But it is not recommended to have a database without any user authentication.

If we write the below xml file for the ApplicationContext Container then it will ask for the other properties i.e. user and pass values

Spring.xml

<beans>

<bean id=”DAO” class=”beans.DAO” dependency-checking=”simple ”>

<property id=”driverclass” value=”com.mysql.jdbc.Driver”/>

<property id=”url” value=”jdbc:mysql:http://localhost/Employee”/>

</bean>

</beans>

To solve this problem they have given some annotations @required to annotate setters method that this field is required to be set compulsorily.

To activate this @required annotation IOC will need to generate a class in order to achieve this things and we have to instantiate the class **RequiredAnnotationBeanPostProcessor**.

For car and Engine Class example if we want the instantiation of engine bean as compulsorily in order to instantiate the Car Class then we use a concept called **Depends on.** This property will instruct the container to create the object of Engine bean before an object of Car can be instantiated.

For example we have three classes A , B and C while B depending on C’s instantiation before it is instantiated likewise A depends on B’s instantiation before it can be instantiated.

Public class A{

}

Public class B{

}

Public class C{

}

Spring.xml

<beans>

<bean id=”a” class=”beans.A” depends-on=”b”/>

<bean id=”b” class=”beans.B” depends-on=”c”/>

<bean id=”c” class=”beans.C” />

</beans>

Mutual dependencies are not possible as it may become ambiguous and JVM will throw an error

P-namespace and c-namespace:

Spring provided us to inject the values in class properties by p-namepaces to inject values using setters and c-namespace to inject the values using constructors. Consider following example

Public class Test{

private String name;

private Car car;

public Test(String name, Car car){  
 this.name=name;

this.car=car

}

//setters

}

public class Car{  
 private class carname;

//setters

}

Spring.xml using p-namespace will be like

<DTD>

<beans>

<bean id=”t” class=”beans.Test” p:name=”ABC” p:car-c c:name=”XYZ” c:car-ref=”c”/>

<bean id=”c” class=”beans.Car”/>

</beans>

**Autowiring**

There are two types of autowiring

XML Based Annotation based

Autowire: ByName **@Autowired**: it’s a property level annotation so must come just

ByType before the property of a bean. By default it will use ByType

constructor mechanism to find the objects. **@Autowired(“ByType”).** So in

autodetect case there is chance to get ambiguity. To resolve that ambiguity

None we can user **@Qualifier(“engine”)** to annotate the bean

Reference eligible for autowiring and that object ref will be only injected in the bean. To activate these two annotation we need to instantiate the class **AutowireAnnotationBeanPostProcessing**.

So in the spring xml we need to configure the **AutowireAnnotationBeanPostProcessing** so that annotations namely **@Autowired** and **@Qualifier(“engine”)** can be used in our application. In this case in our bean class we need not have the setters, getters and Constructors. Without these annotations will directly access the private property of the bean and instantiate and inject the same bean reference bean to dependent bean class object.

Consider a Banking application where we have 1000 view and for each view 1 controller is configured and each controller is again dependent on Business class to produce the output which has be presented to the View whish display the same to User. Now each controller will have the dependency of DAO Class which will connect to the database for its operation to fetch results and provide same to the Business class and further each Dao class will dependent on ConnectionPool Object. So there is 1000 controllers, 1000 business class, 1000 Doas and 1 connection Pool Class object

Dao3

Controller3

View3

View2

Controller2

Connection Pool

Controller1

Dao1

Business1

View1

Dao2

Business2

Business3

------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------------------

..........................................................................................................................................

Business1000

Dao1000

Controller1000

View1000

------------------------------------------------------------------------------------------------------------------

Here in above example Controller class will have the dependency of Business class and same way the Business will have the dependency of Dao class and Dao class will have the dependency of Connection Pool Class. Let’s consider we have a 1000 beans reference of each View, Controller, Business and Dao Class then we have configure the dependency of each bean it is property tag which is hectic task

Spring.xml

<beans>

<bean id = “controller1” class=”beans.Controller”>

<property name=”business” ref=”business1”>

</bean>

<bean id =”business1” class=”beans.Business”>

<property name=”dao” ref=”dao1”>

</bean>

<bean id=”dao1” class=”beans.Dao”>

<property name=”conPool” ref=”conPool”>

</bean>

<beans id=”conPool” class=”beans.ConnectionPool”>

</bean>

………………………………………………………………………………..

</beans>

For thousand Controller class we need to pass the beans reference of it corresponding Business class on which it is dependent on to inject the bean ref it to the controller bean and likewise for Business bean ref we need to pass the corresponding bean ref of Dao class in order to inject into the Business class. Further Connection pool reference to inject into 1000 dao bean reference.

If we configure in beans tag the default-autowire=”ByType” then for all 1000 bean ref of each Controller, Business, Dao class it will set autowire property of its bean tag as ByType. We need not to write the autowire property for each bean which will be hectic task to write for each bean. All controller class is controller package, business class is in business package and dao classes in dao package then by scanning packages spring will create the object for each controller, business and Dao classes but they need to have stereotype annotations. Then spring will create the instance of each object and will do automatic Dependency Injection thus reducing the burden of creating instances of each and also inject the dependencies manually.

There are four **stereotype annotations:**

1. **@Controller: used for Controller classes**
2. **@Repository: used for DAO classes**
3. **@Service: used for Business Classes/ Model Part**
4. **@component: classes other than above three**

**For above implementation to work we need add xml schema xmlns: context in our beans tag.**

<beans xmlns: context:” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ”>

<context:component-scan base-package=”com.ems.controller”/>

<context:component-scan base-package=”com.ems.business”/>

<context:component-scan base-package=”com.ems.dao”/>

<context:component-scan base-package=”com.ems.util”/>

</bean>

The above stereotype annotations can only be used to instantiate the classes in the package if our bean contains only the secondary data types. For primitive data types it will not work and wont

The above all config will scan all the package and spring if find any classes annotated with above four stereotype annotations in these packages then it will create the instance of each class automatically. Total class it will create is equal to 3000+1 classes. 1000 each of controller, business and DAO classes and 1 object for ConnectionFactoty classes which is a utility class annotated as @component.

For annotation like **@Requred** and **@Autowired** and **@ Qualifier** we need object of **RequiredAnnotationBeanPostProcessor** and **AutowireAnnotationBeanPostProcessor** classes in order to activate these annotations. Instead this we can configure our spring.xml document as follows to activate these annotations.

**<context:annotations-config/>**

**This config will activate all the annotations.**

**How to do static variable and static method dependency Injections(DI)**

Public class Car{  
 private static String name;

Public static void setName(String name){

Car.name=name;

}

Public static void displayCar(){  
 System.out.println(“Car Name: ” name);

}

}

To inject static variable into a bean class spring has provided a support class called **MethodInvokingFactoryBean** this class will help to pass the static variable in static methods. In this class spring have given setter methods

1. setArguments(Object [] arguments): By using this we need to pass the arguments of static method
2. setStaticMethod(String staticMethod ) By using this we need configure the static method of the bean.

Our spring.xml file for the same will be configured as

<beans>

<bean id =”c” class=”beans.Car”/>

<bean class=”MethodInvokingFactoryBean”>

<property name=”StaticMethod”>

<value>Car.setName</value>

</property>

<property name=”arguments”>

<list>

<value>Mahindra Scorpio</value>

</list>

</property>

</bean>

<beans>

**Singleton class**: A class that has been instantiated only once for entire mainstack/per main method. Only one main method will be used by a web application. Example of Singleton classes

Java.util.ResourceBundle

We can instantiate the ResourceBundle class using a new keyword as there is a static method which is used to instantiate the ResourceBundle and if we try instantiate it with same static method it will return same reference.

ResourceBundle rb = ResourceBundle.getBundle(“Messages”);

ResourceBundle rb2 = ResourceBundle.getBundle(“Message”); will return same rb object ref only

If we compare these objects then we’ll find that both have the ref variable have same object reference.

Sytem.out.println(rb==rb2) will print **“true”.**

Main objective of singleton classes is reduce memory consumption under heap and hence create only once instance and will pass the same ref if user tries to create another Object of the class. This approach is useful only when all the method and propertied are static type ie static contents only but if it contains the non-static methods then no use of creating singleton classes.

Calender c= Calender.getInstance()

But this class has a private constructor so that it can’t be allow other classes to access its constructors. But Spring can access the private constructors also. But JRE has coded to throw **IllegalAccessException** if we try to instantiate a Calendar class using constructors arguments as spring tries to access the private constructor internally and makes it public by changing its access modifier type to public from private and then instantiate it. If we also write an **IllegalAccessException** for our private constructor of class then spring also will not able to access this constructor.

If we can’t create an object through its private constructors using spring IOC then we need use the factory Method to get its instance. The **factory methods** are those which return the same class object. Factory Classes static methods will return other class instance as follows. Factory classes main aim is to return the reference of Interface whose implementation can be implemented by various clients for example **DriverManager** is a **Factory class** which return the object in the form reference of a Connection interface which just has specification which can be implemented accordingly for various databases by taking their Drivers which provides Connection Interface implementation. If we try to call DriverManager.getConnection() 3 times then it will return 3 object for each call which will be duplicate and will returned to Connection Interface reference. By using Factory Methods they are hiding the objection creation logic. But **LoggerFactory** class will create only **one object even when called for several time**. There are **ProxyFactory classes** also which will create the proxy classes.

Factory class’s main aim is:

1. Hiding instantiation logic
2. By returning instance into interface reference Factories will make the clients independent.

For example Connection Interface which will get the ref of DriverManage.getConnection()

And Connection Interface Implementation is provided by various vendor lie Oracle, MySQL, IBM DB2

**Logging l= LoggingFactory.getLogger(this.class)**

**Validator v = ValidatorFactory.getValidator();**

Here Logger and Validator instance ref respectively as l and v are created by Factory Classes LoggingFactory and ValidatorFactory respectively and getLogger and getValidator are static methods

Session s= sf.openSession();

Here sf is an instance reference of **SessionFactory** and in that instance ref we are creating a Session object by invoking method **openSession** on that instance ref of **SessionFactory.**

The spring.xml file for above types of instantiation will be as follows

<beans>

<bean id =”c” class=”java.util.Calender” factory-method=”getInstance”/>

<bean id =”lf” class=”LoggingFactory” factory-method=”getLogger”/> Logger l = getBean(“lf”)

<bean id =”vf” class=”ValidatorFactory” factory-method=”getValidator”/> Validator v= getBean(“vf”)

<bean id=”sf” class=”SessionFactory”/>

<bean id =”s” factory-bean=”sf” factory-method=”openSession”/>

</beans>

How to create a **singleton Class**

public class Test(){

private Test t;

private Test(){

}

public static Test getInstance(){  
 if (t==null){

t = new Test();

return t;

}

else{

return t;

}

}

}

**Factory Class Examples:**

We have an interface called Car and we need to provide the same for Car vendors like Benz, Maruti, Tata to implement it. For getting the reference of Car interface we need write a Factory Class called a Car Factory

Interface Car{

Public void drive(); implementation by

Maruti

} Benz

Tata

public CarFactory(){

private static carname;

//static setters

private CarFactory(){

}

public static Car getCar(String carname) throws Exception{

return (Car) Class.forName(carname).newInstance();

}

}

Spring.xml for above example

<bean class=”InvokingMethodFactoryBean”>

<property name=”setMethod” value=”CarFactory.setCarName”>

<property name=”arguments”>

<list>

<value>BENZ</value>

</list>

</property>

</bean>

<bean id =”cf” class=”CarFactory” factory-method=”getCar”/>

If our CarFacory has non-static members then our class will be

public CarFactory(){

private String carname;

//static setters

public Car getCar(String carname) throws Exception{

return (Car) Class.forName(carname).newInstance();

}

}

Spring.xml for above example for dependency Injection will be as follows

<bean id=”cf” class=”CarFactory”>

<property name=”carName” value=”Benz”>

<bean>

<bean id= “c” factory-bean=”cf” factory-method=”getCar”>

If Benz is implementing the Car Interface then they should implement the drive the method inside the their implementation

public class Benz implements Car{  
 public void drive(){

System.out.println(“Safe drive 100 km/h”)

}

}

Class Client{

public static void main(String [] args){

Car c= CarFactory.getCar(“Benz”);

c.drive();

}

}

In Spring they have given one **approach to implement factories called FactoryBean Interface**. By implementing this also we can implement a Factory Class.

**In this Interface we have three methods**

1. **getObject(): Object**
2. **getObjectType:Class**
3. **isSingleTon(): Boolean**

class CarFactory implements FactoryBean{

public String carName;

//setters

public Object getObject() throws Exception{

Car c = (Car) Class.forName(carName).getInstance

return c;

}

public Class getObjectType() throws Exception{

return Car.class;

}

Public Boolean isSingleTon(){

Return true; //If we want our Car to be Singleton Object

}

}

**Managing LifeCycle of Bean(Spring) Classes**

In case servlet we have implement our servlet classes from servlet Interface or extend from httpServlet or GenericServlet Class which implements servlet Interface only. Our servlet class will implement lifecycle methods that has been provided under servlet or override the lifecycle methods of HttpServlet or Generic Servlet class. When a servlet class is configured in web.xml file <load-on-startup>1</ load-on-startup> then that server will loaded into the container as the application is deployed on Apache Tomcat Server. After Container loads the classes then its Lifecyle method **init()** will be invoked and **destroy()** method will be called when application is undeployed.

In spring we can manage the life cycle of POJO classes like Controller, DAO and Business classes by using IOC container. That IOC container name is **ConfigurableApplicationContext** which manages the life cycle of each POJO classes. Here the **init() and destroy()** will be maintained by IOC container **ConfigurableApplicationContext.** IOC will create the object at the time of loading the bean classes as configured in spring.xml. After creation of bean object if it contains lifecycle **init()** method inside bean it will invoke that method. Init( ) method will execute only once for its entire life and hence we should write the code which we want to executed only once. For example DAO class responsibility is to take the connection from database by using that connection it may need to do database operations CRUD and finally while closing the DAO beans it need to return back the connection to the database. So in case of DAO class we can write the openConnection Logic. In Dao class we can have multiple methods called save, update, delete, find, findAll. By using the same con object we can do all CRUD operations and finally when destroy() method is invoked we must write cleanup Logic and initialization operations we need to write here. Once we’ll call the close method on **ConfigurableApplicationContext**  object then Destroy() lifecycle method of the beans will be invoked.

Business class may have methods createStudent(), updateStudent(), deleteStudent(), findStudent(), findAllStudent(). And if we want the logging of these method operations then we can create the logger object in init() lifecycle method. Finally we destroy() lifecycle methods is invoked then we write the cleanup code to remove the logger object instant from memory heap. Inside our business methods we can use logger.info(), logger.debug() methods of Logger Class.

Controller class is similar like servlet.

We have three methods to implement the Lifecycle of Spring Beans

1. **Programmatic Approach**: Spring POJO classes must **implement interfaces InitializingBeans and DisposableBeans**. While we are implementing our bean with Interface **InitializingBeans** we need to implement the abstract method **afterPropertySet()** while implementing the DisposableBean Interface we compulsorily need to implement the abstract method **destroy()**
2. **Xml File Approach**: In this approach we need not to implement any Interfaces for our POJO classes. While using **Xml approach to manage the lifecycle** method we can **write our own name for lifecycle methods like myinit() , mydestroy()** in our Bean classes. The methods we need to configure under xml file in order to get the Lifecyle mechanism for the Bean objects.

<DTD>

<beans>

<bean id=”t” class=”beans.Test” init-method=”myinit” destroy-method=”mydestroy”>

</bean>

</beans>

1. **Annotation Approach**: If we have our own Lifecycle methods say **myinit()** and **mydestroy()** in our Bean class then we have to annotate those Lifecycle methods with annotations **@PostConstruct** and **@PreDestroy** respectively. A sample is given below. To activate these annotations we need to create the object of **CommonAnnotationBeanPostProcessor** bean context. Fully qualified name of the class is **org.springframework.context.annotation.CommonAnnotationBeanPostProcessor**

@PostContruct

myinit(){

//Initilaization logic

}

----------------------------------

----------------------------------

@PreConstruct

mydestroy(){  
 //Cleanup Logic

}

Suppose we have 100 Test bean classes as follows

Test1{

init(){}

destroy(){}

}

Test2{

init(){}

destroy(){}

}

Test3{

init(){}

destroy(){}

}

-----------

-----------

Test100{

init(){}

destroy(){}

}

If we are using the same name for init() and destroy() lifecycle methods for each and every Test Classes then we can configure their name respectively default-init-method and deafault-destroy-method attributes in root beans tag.

<DTD>

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

<bean id=”T1” class=”beans.Test1”>

</bean>

<bean id=”T2” class=”beans.Test2”>

</bean>

<bean id=”T3” class=”beans.Test3”>

</bean>

-----------------------------------------------------

-----------------------------------------------------

<bean id=”T100” class=”beans.Test100”>

</bean>

</beans>

There are 3 types of Dependency Injection

1. Setter
2. Constructor
3. Lookup Method DI

**Lookup Method Dependency Injection**

public interface Test{

public void a(); These are lookup methods

public void b();

}

abstract Class Test{

abstract public void a(); this is lookup method

public void b(){ This is concrete method

}

}

Class Test{  
  
 public void a(){

//Impl

}

public void b(){

//Impl

}

}

If methods a() and b() have complete implementations then these two methods are fully implemented methods. In case we want to override the implementation part of these methods then we can call these methods like lookup methods. A lookup method we can expect from Interface or Abstract class or Concrete class. If it’s an interface then compulsorily we have to implement the methods and compulsorily we have to implement all the lookup methods. Abstract classes may or may not have abstract methods. Abstract methods may become lookup methods and in concrete class also we can have the lookup methods if we don’t want the implementation methods then we can override the method with our won implementation which will make it lookup methods. By Lookup method we can provide implementation and also can override the implementation. By using spring we can provide the implementation. Spring provides some runtime implementation classes to provide implementations to our methods. Which type of implementation we require whether its primitive, secondary, Array, Secondary array or Collection Type. We can pass the same through property of bean tag in spring.xml file.

Abstract class Car{

abstract public Engine myEngine();

}

Spring.xml

<DTD>

<beans>

<bean id=”c” class=”Car”>

<lookup-method name=”myEngine” beans=”eng”/>

</bean>

<bean id=”eng” class=”beans.Engine”/>

</beans>

When we load above the spring xml in IOC then IOC will generate one runtime proxy class CarProxy extending abstract class Car and override the lookup method that is configured for lookup in our case it is myEngine and will create the Engine bean class instance and return the reference of it

public class ProxyCar{

@override  
 public void myEngine(){

return eng;

}

}

Car c= (Car) ap.getBean(“c”)

When will call the getBean() method on ApplicationContext object then it will return the Car reference but will not create the Car object but Object created at Runtime, which is a Proxy class i.e. Child class of abstract class Car which is ProxyCar, will get the reference of Car.

We can get the name of proxy class created by Spring at runtime as follows

System.out.println(c.getClass().getCannonicalName());

Engine e= c.myEngine();

We have one more tag in spring nested under bean tag Method-Replacer tag by using it will override the existing implementation with our own implementation.

public class Bank{

Deposit(){

//Current Implementation

}

Withdraw(){

//Current Implementation

}

calculateInterest(){

//Current Implementation

}

}

If we want override our calculateInterest() method of our class then we can provide the method implementation patch using Method Replacer.

Class NewInterestCalculator implements MethodReplacer{

public Object reimplement(Object o, Method M, Object [] param){  
 //New Implement

}

}

Spring.xml config file for the same will be

<DTD>

<beans>

<bean id=”b” class=”beans.Bank”>

<replace-method name=” calculateInterest” replace=”nic”>

</bean>

<bean id=”nic” class=”beans. NewInterestCalculator”>

</bean>

</beans>

Here also IOC will provide **a proxy class** when we load the spring.xml as follows

Class BankProxy extends Bank{  
 @override

Public void calculateInterest(){

//The new Implementation

}

}

It’s useful when we want to override the implementation of a method and we don’t want to redeploy the application on the server. In we just have write a class which extends the MethodReplacer which will override the current method implementation.

**Autowiring In Spring** Continued

We have two annotations in spring for autowring

1. @Autowired
2. @Qualifier(“eligible bean for autowiring”)

In JDK also we have 2 annotations for autowiring which is equivalent to @Autowired annotations of Spring Framework and we can use these annotations in spring application also as an alternative to @Resource. The @Resource Annotation will inject the object ByName by default and @inject annotation will inject the object ByType by Default

1. @Resource
2. @Inject

Auto wiring The main objective of this is to automatically inject the secondary types. It's no applicable primitives.

We ll consider an banking example

A banking applicationn will have so many forms and fro each form there will a controrller attached to it so that it can accept the request and and pass the response back to user in order that it can be understandable by Users. For each Controller there might be Business class on which might be dependenet on likewise The business class might have the dependancy of DAO class which fetch the data from the database and produce the data to Business class.

Lets say there ara 100 Forms and 100 Controller classes are there for these form also there 100 Business and 100 DAO Classes in order to assist former in their execution.

our spring.xml file will be configured as follows

<DTD>

<beans>

<bean id="d1" class="beans.DAO">

</bean>

<bean id="b1" class="beans.Business">

<property name="dao" ref="d1">

</bean>

<bean id="c1" class="beans.Controller">

<property name="dao" ref="d1">

</bean>

<bean id="d2" class="beans.DAO">

</bean>

<bean id="b2" class="beans.Business">

<property name="dao" ref="d1">

</bean>

<bean id="c2" class="beans.Controller">

<property name="dao" ref="d1">

</bean>

//Here the above bean config we have to dove 98 more bean classes which will be tedious task for the developers

</beans>

if we use auto wiring property then we dont need to pass the refernece of the bean a taht a another is dependent on. IOC is designed to sense the object a class is dependenet on and will accordingly inject the same bean but that bean class property must be inside the bean class which is depending on it then only IOC can create the bean object.

Autowire attribute of bean tag (or can be used in beans tag as defalut-autowire) can accept following type of values

1. ByType Setter Injection the must have the setter method of the bean it is dependent on otherwise it will not inject the same because when it will setter then only IOC will consider the parameter as dependent parameter. If IOC finds more than one object for the same dependent property in the bean tag then ambiguity occurs as which to choose to inject in the bean class it is dependent on and will throw beancreationException. In such case we need to set autowire-candiate of bean tag to be false which true by default. So the object whose autowire is true only those object is eligible for the autowire candidate. ByType is the best approach because we don’t have to remember all the reference types as becomes hectic task we remember all these ref types.

2. ByName Setter Injection: If we define this in autowire then it will search the reference in the bean matching the same Type and name and if both of these things are met then IOC will inject that reference in the bean object which is depending on it. It doesn’t find the bean with same reference name a used in Bean property then no bean ref is eligible for injection. As Ids are unique so this type of autowiring is considered safe

3. Constructor: Constructor Dependency Injection is used when we have Constructor Dependency Injection If there is Constructor of a bean class then IOC will search in the parameter which needs to be injected and then by parameter type it will inject the reference of the same bean. Internally it will do the same operation as that ByType Injection and thus will have the same issue as with ByType.

4. Autodetect: Setter and Constructor D Injection after writing 100 Dao, Business and Controller classes we might not know which class has constructor and which class has setters for the its parameters. In such case this value of autowire is very useful. It'll automatically find the dependncies are in the form of setter or Constructors and hence it will find out the sane and accordingly inject the required dependencies.

5. No (Default)

<beans default autowire="ByType">

<bean id="d1" class="">

</bean>

<bean id="b1" class="">

</bean>

<bean id="c1" class="">

</bean>

//Same as above remaining 99 classes

<beans>

Above spring.xml will inject the dependencies in the beans if any automatically by checking the type of bean class it is dependent on and will inject the appropriate class.

Liecycle of dao classes

StereoType annotations also given in our J2EE container

1. @Named have same functionality as of @Component of spring

To load properties file data into the class in order to inject the dependency we use class whose name is **propertyPlaceHolderConfigurer** on object of which we have to invoke the method setLocation(Path of prop file). This method will load the file in the IOC container and IOC will get the data from properties file and store the same at context scope and we can use the expression to get the data from IOC context.

<beans>

<bean class="PropertyPlaceHolderConfigurer">

<property name="location" value="Path of properties File"/>

</bean>

<bean id="cp" class="beans.ConnectionPool">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="userName" value="${userName}"/>

<property name="password" value="${password}"/>

</bean>

</beans>

I18N Support to users so that the product can be used by different peoples with different linguistic regions

L10N Support also needed in order provide for I18N Support. Here in l10n support we need to provide business support and provide validations. For example we are building the application for Bank which can be used by India and US users. If the user from India wants deposit money then he can deposit in Rupees while user can deposit in US Dollars. For interest calculation business policy for India may differ from that of US. So in such case also we need to provide the appropriate business logic for both the countries to comply with their interest rate calculation standards.

Here in spring J2EE container ApplicationConetxt will provide for I18N support. In core Java also I18N support is provided and can be implemented as follows.

Locale l = new Local("hi") Its instructor must contain the language which you want to give support for

ResourceBundle rb = ResourceBundle.getResource("properties file", l) In its parameters we need to pass the fully eligible name of the properties file i.e. it path and the locale object we created before this line.

String value=rb.getString("key") for a specified string it will return the value the corresponding string

S.o.pln(value)

The same thing we can achieve in our AplplicationContext J2EE Container for Spring

Locale l = new Locale("hi")

ApplicationContext ap= new ClassPathXmlApplicationContext("spring.xml");

String value=ap.getMessage("key", null, l);

In spring.xml we have to configure the location of properties file by instantiating class called ResourceBundleMessage

<beans>

<bean id="messageSource" class="ResourceBundleMessage">

<property name="baseName" value="Data file location"/>

</bean>

</beans>

In spring.xml file we compulsory need write the ref name from the **ResourcBundleMesage** class as **messageResource** or spring will force us to write the name as above in order that application functions as intended

Spring MVC

1. **Form backup Support**: It will store the data of form in a bean class and we need to submit the data so that it can be validated and if any field in form is invalid then it will populate the validated data and will remove the invalid from the bean and will return the validation error against the invalid fields. to get this support we need to use spring tag libraries.

2.**Controller** (MultiAction Controller and MultiForm Controllers)

For example if we have 3 forms student registeration, student update and student search operations and all these opertion we want to execute with same controllers then we can write the controllers for the same and we can write the each event in single controller class and on button click the appropriate event will invoked.

3. Validation Support

4. I18N Support

5. Interceptor Support

6. ViewResolvers

7. ExceptionHandling

For MVC Application the container is WebApplicationContext.

Spring MVC is designed using jsp model architecture. As per Jsp model arhcitecture we have 4 models

3 Form data

Helper Controllers

StuReg Controller

StuReg Form

Login Bean

EmpReg Bean

StuReg Bean

EmpReg Controller

Login Controller

EmpReg Form

Login Form

Front Controllers

Validation Read Data

RR

Controllers

Storing valid data into the beans

3 form are there in above application which is attached to a single controller. If we want to write common set of operation then we can write the same in Controller class. After the Controller execute the common set of operations for each form then specific operation related to each form it needs to execute. In order to execute these operation specific to each form we need Helper classes(Controllers) for each form i.e. in above example there three Helper Controller (StuReg, EmpReg, Login). If we have thousand forms then we need not write thousand **validation and read Data operations** in separate controllers but we can write these common operations in a single controller thus reducing the code writing. The Helper Controller will have the access for the corresponding beans and by taking the reference of beans it will execute the form specific operations written in Helper Controllers. Each Helper controller classes can call the model part for each Helper classes to execute the Business Codes. **The common Data validation and Data read operation as well as common required operations like Internationalization, localization, Controllers support, Exception Handling Support, Form Backup Support, ViewResolvers (Templating) Support can be written in a Controller called as Front Controllers.**

Four JSP Model are there for Front Controller Approach

1. JSP (JSP-Model 1) : Chances of Manipulation
2. Servlet (JSP- Model II) : Can Protect the data as Its written as class| Struts 1.x|Spring MVC|JSF
3. Filter (JSP-Model III): It also a class| Struts 2.x
4. Tag Support (JSP-Model IV)

Servlet and Filter JSP Model are **recommended First Controller** Approach. In springMVC, the name of the front controller is **DispatcherServlet** and the name of front controller in Struts 1.x is **ActionServlet.**  For JSF the front controller name is **FacesServlet.** For Struts 2.x the front controller name is **FilterDispatcher or StrutsPrepareandExecuteFilter.**

Each from we have to map its form action to the front controller servlet below is the mapping to be configured in web.xml.

<web-app>

<servlet>

<servlet-name>ds</servlet-name>

<servlet-class>DispatchServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ds</servlet-name>

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

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

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

-------------------------------------------- if we have thousand forms then we have to configure all the them with their form action.

Or instead we can write

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

To configure the all the config action to the front controller DispatcherServlet

</servlet-mapping>

</web-app>

Tomcat will support three type of URL pattern

1. Complete character Sequence
2. /\* : If we are going for only one framework to develop an application then this pattern is recommended
3. \*.ext: If we are using 2 fameworks in our application then this pattern is recommended

<web-app>

<servlet>

<servlet-name>ds</servlet-name>

<servlet-class>DispatchServlet</servlet-class>

</servlet>

<servlet>

<servlet-name>as</servlet-name>

<servlet-class>ActionServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ds</servlet-name>

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

</servlet-mapping>

<servlet-mapping>

<servlet-name>as</servlet-name>

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

</servlet-mapping>

</web-app>

In form action we have to append the extension so that specific framework front controller can determine to which front controller has to be called for a specific form. The extension name we can use anything we can give our own custom name. For example StuReg form can be given action name as studreg.sp while EmpReg from can be given action name asempreg.do and Login from can be given action name as login.sp so that StuReg and Login forms are configured with Spring **front** **controller DIspatchServlet and likewise EmpReg** from is mapped with **struts 1.x front controller ActionServlet.**

**For struts 1.x the recommended url pattern is /\*.do and for struts 2.x the recommended url-pattern is /\*.action**

A sample springMVC example

/hello.htm Spring.xml HelloController

Controller()

output folder

dispatcher-servlet.xml User specific Controller

Success.jsp

To implement user specific controller we can use following

Programmatic Approach

1. Controller(I)
2. AbstractController
3. AbstractCommandController
4. SimpleFormController
5. AbstractWizardFormController
6. MultiActionController.

StereoType Annotations

@Controller

If we implement our class with Controller Interface then we need to compulsorily implement the following method

Public ModelAndView handleRequest(HttpRequest req, HttpResponse res) throws Exception{

String name= req.getParameter(“name”);

Map m= new HashMap();

m.put(“msg”, “Hello………..”+ name);

return new ModelAndView(“success”, m);

}

We can read data by using request object and finally we can send the response using ModelAndView object. This method will return the ModelAndView object to FrontController which will be provided to the success.jsp where we can access the map object by using **${msg}.**

web.xml config file

<web-app>

<servlet>

<servlet-name>dispatch</servlet-name>

<servlet-class>DispatchServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name> dispatch </servlet-name>

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

</servlet-mapping>

</web-app>

Here we have define an spring config name which name should start with Front Controller servlet name as configured in web.xml i.e. in this example dispatcher and then putting a hyphen(-) writing servlet.xml. **The full name of spring config file here should be dispatcher-servelt.xml**. here in this xml we have to configure the controller class for a specific form action and also the location of success page and its extension. To configure the location of our output page we need class called **InternalResourceViewResolver**  for which they have given 2 setter method setPrefix() and setSuffix(). In Prefix property we needset the value as folder location of the success page and suffix property we need to pass type of file it is eg jsp, html etc

<beans>

<bean name=”/hello.htm” class=”HelloController”/>

<bean class=”InternalResourceViewResolver”>

<property name=”suffix” value=”/output/”>

<property name=”suffix” value=”.jsp”>

</bean>

</beans>

In our success.jsp we can read the map data by using the Object Graph Notation Language (OGNL) expressions.

Controllers

Core Controller (2)

Controller(I)

Throw away Controller(1)

Command Controllers(5)

BaseCommandController

FormController(5)

SimpleFormController

AbstractWizardFormController

AbstractCommandController

AbstractFormController

ThrowAwayController

View Controllers(2)

UrlFileNameView Controller

ParameterizableView Controller

AbstractController

Multi action Controller(1)

MultiActionController

**Core Controllers**

1. **Controller**
2. **AbstractController**

**View Controller**

1. **ParameterizableViewController**
2. **UrlFileNameViewController**

**Command Conrtroller**

1. **BaseCommandController**
   1. **AbstractFormController**
      1. **SimpleFormController**
      2. **AbstractWizardFormController**
2. **AbstractCommandController**

**ThrowAway Controller**

1. **ThrowAway Controller**

**MultiAction**

1. **MultiActionController**

**Handler classes(First 3 are usefull)**

1. **BeanNameUrlHandlerMapping**
2. **SimpleUrlHandlerMapping**
3. **ControllerClassNameHandlerMapping**
4. **CommonsPathMapHandlerMapping**

**ViewResolver Classes**

1. **InternalResourceViewResolver**
2. **TileViewResolver**

<Servlet-name>-servlet.xml

Read the spring config file

DispatchServlet

Forms

1 2

handlerMapping Class

5 4 ControllerName

Success.jsp or failure.jsp

ViewResolver

ModelAndView 3

Will suggest the location of page Req, Resp

And its extension 5

User specific Controllers

**SpringMVC Architecture**

Whenever we submit any from The DispatchServlet will send the 1st request to HandlerMapping(Default Handler Class name is **BeanNameUrlHandlerMapping**) Class to know about corresponding Controller Class name. By using url pattern also they can identify the controller attached for an View.

In spring config file we have to configure the Controller class name with some url pattern as follows so that HandlerMapping can find the controller class for the action url.

**<bean name=”/urlpath.ext” class=”ControllerClassName”/>**

HandlerClass will read the above config in spring config file and will find the appropriate Controller class mapped for the url and Handler class will sugghest the DispatcherServlet the Controller name for given url then DispatcherServlet will create the instance of Controller Class and will pass the HttpRequest and HttpResponse Object to Controller Methods and Controller class will execute the code under handleRequest and will Return the ModelAndView Object and in our ModelAndView we need to configure our required Map Object and success/failure view page name. Using ModelAndView Object The DispatchServlet can’t find the output page. It need to find the view page location and its extension which may jsp, html etc and for view page location and extension it will contact viewResolver class which will suggest for the view page location and extension of the page.

We have on more Handler class named as **ControllerClassNameUrlHandlerMapping.**  If the controller name is **Reg**Controller and our form action for the same is **reg**.htm then we have to configure the controller class in spring config file as follows so that Form action action name **reg** is same as **Reg** in Controller Name.

<bean class=”ControllerClassNameUrlHandlerMapping”/>

<bean class=”RegController”>

So in this case we need to follow the **naming conventions**.

We have one more HandlerMapping class **SimpleUrlHandlerMapping** . This HanderMapping class is the recommended one. Its configuration in spring config file will be as follows

<bean class=”SimpleUrlHandlerMapping”>

<property name=”mappings”>

<props>

<prop key=”/reg.htm”>r</prop>

</props>

</property>

<bean>

<bean id=”r” class=”RegController”>

One more HandlerMapping class is there with name CommonsPathMapHandlerMapping: Maps controllers to URLs using source-level metadata placed in the controller code. The metadata is defined using Jakarta Commons Attributes

If we want to keep the name of spring config file name as spring.xml instead of <dispatcherservletName>-servlet.xml then in web.xml we have to do extra configuration ie in <init-param> tag we need to provide the fully qualified path of the spring.xml file ie /WEB-INF/spring.xml

<servlet>

<servlet-name>dispatcherservlet<servlet-name>

<servlet-class>DispatcherServlet<servlet-class>

<init-param>

<param-name>ContextConfigLocation</param-name >

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

<init-param>

</servlet>

<servlet-mapping>

<servlet-name>dispatcherservlet</servlet-name>

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

</servlet-mapping>

We can also configure the the ContextConfigLocation inside the <context-parameter> tag also outside of <servlet> tag as follows

<listener>ContextLoaderListener</listener>

<context-parameter>

<param-name>ContextConfigLocation</param-name>

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

</context-parameter>

In AbstractController class which is core Controller Abstract class the method we need to implement here is handleRequestInternal.

Eclipse Project

src

resources

test.xml

Web Content

WEB-INF

Spring.xml

To load both the xml tes.xml and spring.xml in our Application Context then in our web.xml we have to configure them as follows.

<web-app>

<servlet>

<servlet-name>ds</servlet-name>

<servlet-name>DispatcherServlet</servlet-name>

<init-param>

<param-name>ContextConfigLocation</param-name>

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

classpath:resources/test.xml

</param-value>

</init-param>

</servlet>

</web-app>

**AbstractController** class has methods as follows

1. Public ModelAndView handleRequest(HttpRequest req, HttpResponse resp) throws Exception
2. Public ModelAndView handleRequestInternal(HttpRequest req, HttpResponse resp) throws Exception

But 1st method will delegate the request to second method and hence 2nd method will carry out the operations. So business code execution happens here in 1st but only delegation to 2nd method so that 2nd method will be called.

**ParamerizableViewController** class is child class of **AbstractController** class and has property viewName which we need to set in spring file and then retrieve by getter method at controller side in order to find the appropriate success page. Configuration for the same is done as follows in spring.xml file

<bean class=”bean.HelloController”>

<property name=” viewName” value=”success”/>

</bean>

**UrlFileViewController** class is **Child class of** **AbstractController** class and used for doing forward operations. In case of simple hyperlink request and we want to map a hyperlink request to any form we can use this Controller class. They have given the implementation already and we need not override the implementation. They have implemented the handleRequestInternal(req,resp) method as follows

Public ModelAndView handleRequestInternal(HttpRequest req, HttpResponse resp) throws Exception{

//will read the action name configured for a url say (/hello.ds)

//Action name it will use as a page name

return new ModelAndView(“hello”);

}

handleRequestInternal method of UrlFileViewController will return the same name as of action name and will return ModelAndView(“hello”) object to handleRequest() method which in turn forward this object to DispatcherServlet and then it’ll ask the ViewResolver to identify the page name using the location(**prefix**) and extension(**suffix**) of file being requested for forwards.

ThrowAwayController

Name -----A-------- DispatcherServlet using Bean implement Beans imp ThrowAwayController{

Email--------B------- to ThrowawayController will take the private String name;

Address-------C----- help of this controller to set the bean private String email;

with form data private String address;

//getters

//setters

}

Following method of the ThrowAwayController Interface we need override to write our own implementation

public ModelAndView execute (){  
  
 ===========

returm mav;

}

For doing multAction we have **MultiActionController** Class which is used. This Controller class is child class of AbstractController Class. Three points we need to take care of when we using MultActionController class

1. We should’t override the handleRequest() method as well as handleRequestInternal() method.
2. We should write the custom methods for our Controller operations with handleRequest method Signatures.
3. Custom Method names should be action names.

For example

DispatcherServlet

/empsave.ds EmpCRUDController ext MultiActionController

public ModelAndView handleRequest(req, resp) throw Exception{

}

public ModelAndView handleRequestInternal(req, resp) throw Exception{

}

public ModelAndView empSave(req, resp) throws Exception {

}

public ModelAndView empUpdate(req, resp) throws Exception{

}

/empupdate.ds

Whenever a empSave form is submitted then DispatcherServlet will call handleRequest method which will delegate the task to handleRequestInternal() method which will try to find the method name which same as action name and hence will invoke that method ie empSave() method. Likewise it will call empUpdate() method when it will find the with same action name as that of method name.

In multiActionController we have a method called setMethodNameResolver(methodNameResolver mnr)

Which have suffix and prefix as properties which we can set if we want add before(preffix) and after (suffix) our action name some text to give the method name as some meaningful name.

**BaseCommandController(Depcrecated in spring 3 and later verions)** is a command controller which will store the form data into beans which need to create for each form we have for our application. If we use this Controller then we need to write our form using spring form tags using following import line

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

<html:form commandName=*"empreg"* method=*"POST"*>

<html:errors/>

<pre>

Name : <html:input path=*"name"*/>

Email : <html:input path=*"email"*/>

Address: <html:input path=*"address"*/>

<input type=*"submit"* value=*"save"*/>

</pre>

</html:form>

**Spring DAO**

**DAO(DataAccess Object)**: The name suggests as these object interacts with various source of data to provide the Business class with data required by the Business class. It may interact with RDBMS, XML Document, LDAP machines, NoSQL Machines

RDBMS

XML

LDAP

NoSQL Machines

DAO

Following are API the DAO will take advantage of to interact with different type of DATA

1. JDBC used to interact with RDBMS
2. JAX-P, JAX-B are used to interact wit XML documents
3. ORM are used to interact with databases only but have certain additional features which JDBC cant provide.
   1. EJB
   2. JPA
   3. Hibernate
   4. Ibatis
4. OGM to interact with NoSQL Databases

Spring also have given support for interacting with RDBMS and required jars are spring-dao and spring –orm. Here we have to develop our dao part as DAO interface and DAO implementation where DAO interface contains all the method of DAO operations and DAO implementation class will be implementing the 3rd party implementations like EJB, JPA, Hibernate, or Ibatis so that later point of time if we are migrating our application from one framework to another framework our Business part is not affected.

If various implementation of DAO take the reference of DAO interface in a ORM then for different DAO implementation types we can have the dao interface reference in our Busienss class by using Spring Dependency Injection

StudentDAOImpl Class

public class StudentDAOImpl implements StudentDAO{

public int save(Student st){

//implementation

}

public boolean update(Student st) {

//implementation

}

public boolean delete(Student st) {

}

public Student findByPK(int id) {

}

public int findByName(String name) {

}

public List<Student> findALL(){

}

}

Class Student{

Private String id, name, email, address;

//Getters

//Setters

}

Model Object

public interface StudentDAO{

public int save(Student st);

public boolean update(Student st);

public boolean delete(Student st);

public Student findByPK(int id);

public int findByName(String name);

public List<Student> findALL();

}

StudentDAO interface

Getting Connnection from Database is not recommended. It is recommended to maintain the ConnectionPool Object to increase the database Access Performance and hence increase the application performance. So in our DAO class we have to read the connection from connection pool Object then execute the operation on the tables and finally place the connection into the Connection Pool Object so that other users can use it. Direct connection from database is not recommended for example if for 1 lakh user try to interact with the Database then 1 lakh times it need to open the sockets and 1 lakh time it needs to close the socket. If it takes 0.1 sec to access the connection from ConnectionPool and 0.1 second to place the connection back to the ConnectionPool, then to open the socket of database will be 1 second to get connection and again for closing the socket it may take 1 second. So in case of using the Connection we will get 10 times better performance than that of getting connection from database directly.

One more disadvantage with direct connection access is that if the maximum amount of the Database connection (say 20) is used up and at a time only 20 users can interact with database and if some more user try to access the database then it will ignore the access with some exceptions that all the connection is used. But in connection pools we can put some wait time like 10 seconds which will ensure that if all the connections are used up then new user will have a wait time of 10 second before its request to connect to the database is refused. If in that time range any connection are available for use then this connection will be allocated to the data connection request.

1. Before implementing DAO class we need to provide DAO interface
2. In DAO interface create the abstract methods with Model Object as parameters wherever needed
3. In DAOImp class get the connection from ConnectionPool Object which is recommended in performance point of view.

For creating a connection pool the sun has given a standard interface **DataSource**. So our CommectioPool Class must implement this Interface. For implementation of DataSource mabny vendors have given the implementation.

1. BasicDataSource given by Apache Foundation
2. ComboPooledDataSource given by MChange for small enterprise Applications
3. DriverManagerDataSource given by Spring vendors
4. WebLogicDataSource given by WebLogic Application Server of IBM

**How to create the ConnectionPool**

BasicDataSource bds = new BasicDataSource();

bds.setDriverClassName(driverName);

bds.setUrl(url);

bds.setUrl(userName);

bds.setUrl(pwd);

bds.setMaxConnection(20);



bds.setMinIdle(5);

bds.setWait (5000); Connection Pool

Conenction con=bds.getConnection();

It will get a connection pool and from that pool if we want to get the connection we can get it by invoing method getConnection() method on BasicDataSource Object which will mark 1 one of the Connection pool as used. It will maintain 2 map objects: FreeMap which store the Used Connection Pool and ConsumedConnectionMap which will have the details of consumed connection and once any connection is closed then that pool is added back to FreeMap. Connection Pool get all the connections size at the time of deployment application and will place the connection back to it when it is undeployed.

**package** beans;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** org.apache.commons.dbcp2.BasicDataSource;

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

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

BasicDataSource bds= **new** BasicDataSource();

bds.setDriverClassName("com.mysql.jdbc.Driver");

bds.setUrl("jdbc:mysql://localhost/Mas");

bds.setUsername("root");

bds.setPassword("root");

bds.setMaxTotal(20);

bds.setMaxIdle(5);

bds.setMaxWaitMillis(1000\*5);

**for** (**int** i = 0; i < 25; i++) {

**try** {

Connection con = bds.getConnection();

System.***out***.println(con+ " : " + (i+1));

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

Above ConnectionPool object is asking for more than 25 connection but the connection pool has 20 conncetion to give but request after 20 connections will be kept in queue so with wait time 5s and after 5 seconds if it doesn’t get any unused connection in the connection pool java.util.NoSuchElementException: Timeout waiting for idle object

will be thrown for every 5 seconds.

In order to provide the connection to every new request the previous Connection in the pool which is idle for 5 minutes has to be released or connection should be closed so that it can serve any no of connections. In case of Connection Pool the getConnection() Method will interact with Database but will interact with Map in the Connection Pool and hence if there are 25000 connection need to be made then it’ll interact with connection pool to get the connection and then close the connections.

**package** beans;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.SQLException;

**import** java.util.Date;

**public** **class** Test1 {

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

System.***out***.println("JDBC connection from database");

Date d1= **new** Date();

**for** (**int** i = 0; i < 25000; i++) {

**try** {

Class.*forName*("com.mysql.jdbc.Driver");

String url="jdbc:mysql://localhost/Mas";

String user="root";

String pwd="root";

Connection con = DriverManager.*getConnection*(url, user, pwd);

System.***out***.println(con+ " : " + (i+1));

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

Date d2= **new** Date();

System.***out***.println("\nStart time: "+ d1);

System.***out***.println("Start time: "+ d2);

System.***out***.println("Time difference: "+ (d2.~~getSeconds~~()-d1.~~getSeconds~~()));

}

}

Direct Connection to the database is not recommended because it may provide maximum connection it may provide and if all the connection is used up then it will refuse all connections and will throw exceptions

com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: The driver was unable to create a connection due to an inability to establish the client portion of a socket

Caused by: java.net.BindException: Address already in use: connect

In spring JDBC they have given a class to do JDBC related tasks ie Database Operations like CRUD operations. Instead of using JDBC we can use the Spring JDBC class called **JDBCTemplate**  where methods include are

1. setDataSource(DataSource ds)
2. update(SQL sql);
3. queryForInt(SQL sql): int
4. queryForDouble(SQL sql): double
5. queryForFloat(SQL sql): Float
6. Like that multiple methods with different datatype
7. queryForObject(SQL sql): <T>
8. queryForMap(SQL sql): Map
9. queryForList(SQL sql): List

In case of Spring JBDC we can use JDBCTemplate class to implement our various CRUD methods. JDBCTemplate have the dependency of DataSource object so we can configure the dependency in the spring xml file as

<bean id=”jt” class=”JDBCTemplate” autowire=”Byname”>

<property id=”dataSource ref=”bds”/>

</bean>

<bean id=”bds” class=”BasicDataSource”/>

public int save(Student s){

Object[] params= new Object[]{s.getId(), s.getName(), s.getEmail, s.getAddress};

Int i = jt.update(“insert into Student values(?,?,?,?) ”);

}

The above statement of update which is invoked on the object JDBCTemplate will internaly run the prepared statement and will provide the result. For parameters we need to pass as Object Array. Here in above case we need not to worry about open conncetion and close connections. It is taken care by spring JDBC framework and also we need not to handle any checked Exceptions. If there are database related exceptions then it will throw the same exception at run time only. It means they have converted the database related checked exceptions into unchecked exceptions. Following are the difference between JDBC and SpringJDBC

JBDC Spring JDBC

CompileTime Exception like SQLException converted into RuntimeExceptions

SQLexception DataAccessException

Java.sql package spring-jdbc jar

User need to close the connection in order User need not to close the connection. These things are

The connection are available for other taken care by spring JDBC

Connection request

PreparedStatement is type safe ie All JDBCTemplate Methods are type safe by using **Type**

Appropriate datatype for a specific field will **Class**(used for type safe) which will check types

Be determined at Compile time only

For Any Select Query it will return ResultSet But here we have **generic select methods** if we require

Even for Select max(salary) from emp so only id then id will be returned, if required salary then user has to iterations for every result set salary will be returned. Here we need not to any type of iterarions

In case of implementing our DAO classes with Hibernate framework we need to inject the SessionFactory Classs in our DAO Class and on that session factory Object we need to obtain the connection by invoking openSession() method.

DML operations

Session s= sf.openSession();

Transaction t =s.beginTransaction();

s.save();

t.commit();

s. close();

Select Query Operation

Session s= sf.openSession();

Entity e = (Entity) s.get(Entity.class, pk) //seriazable Primary Key

s.close();

Instead of using palin hibernate code we can use the HibernateTemplate class given by Spring framework. In this also we have methods like that of Simple Hibernate implementation. Following are the methods available in this class

1. save()/ persist()
2. update()/merge()
3. delete()
4. get() methods for single DRL Operations
5. load() methods for single DRL operations
6. find(HQL): List<T>
7. findByCriteria(DetachedCriteria): List

While using simple hibernate framework we need to open session we need write code for begin transaction, close transaction and then close session but in the case of HibernateTemplate Class we need not to implement do all these things. All these things are taken care by the Spring Framework.

DAO SessionFactory

DI

HibernateTemplate

DI

A SessionFactory Object will maintain a Pool of Sessions. From those session HibernateTemplate will get 1 session and it will CRUD operation and finally it will place that session back in Pool. At the same time it will all transaction related operations like starting the transaction and ending the transactions.

DAO2

DAOI

BOI

BOI boi=

DAO 1

DAOI daoI=

Business

DAO

DAO3

Controller Class Model

Spring DAO part spring enforces us to use Interface and Implementation pattern to achieve IOC. Here also to achieve IOC between Business and Controller classes. For business also we need to use Interface and Implementation Model. Here in above example if we user interface and implementation model for DAO module. If we take the reference of DAO interface and on that reference we can inject any DAO implementation which may be from Hibernate, JDBC, or any other ORMs which will reduce the dependence of Business on DAO classes and hence it will be totally decoupled with DAO classes. It means if we migrate our application from 1 database implement to other database implement then we do not need to change the design of Business class as it is decoupled. Similarly all these things can be implemented for Business and Controller part in order to decouple them i.e. eliminating the dependency of Business class in Controller class. In the same way Controller class also can take an Interface Reference of Business class and on that reference we can inject various implementation of Business class. But we may ask question why there is need of using Interface Implementation of business class there no way there will be new technology we need to implement our business class. Then why we required this model. Business Code may change in future. Use cases may be same but Business logic we may need to change which depends on various user requirements and if we do not want that change in Business logic to affect our Controller then we must strictly follow Interface-Implementation pattern.

Model Object are recommended for simple CRUD operations while DTO Objects are recommended for complex crud operations.

**BO interface** for **basic CRUD operations**

Methods

1. use usecase name as method name.
2. In signature use model object

**BO Interface** for **Complex CRUD Operations**

Methods

1. Use usecase name as method name
2. Using usecase arguments we have to create a DTO(Data Transfer Object)
3. Use DTO object as method argument.

Let’s consider an example

Student Table

SID: \_\_\_\_\_\_\_\_\_\_\_\_\_

Name:\_\_\_\_\_\_\_\_\_\_\_\_

Email: \_\_\_\_\_\_\_\_\_\_\_\_

Course:\_\_\_\_\_\_\_\_\_\_\_

Fee:\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | Email | Address |
| 101 | Masroor | [mas@gmail.com](mailto:mas@gmail.com) | Varanasi |

Submit

|  |  |  |
| --- | --- | --- |
| Couse\_Id | Course | Fees |
| 201 | Java | 1000 |

Course Tabe

If our business classes need to do operations on two tables then our Business class compulsorily need to have transaction capabilities. If with one business class we need to talk to 2 daos then we need to have a transaction capabilities i.e. if transaction in dao is completed successfully and transaction second dao is not executed successfully then both of the transactions need to rolled back and if transactions in both daos are completed successfully then only we need to commit whole transactions. The Commit operation such case is need to be done by Business not by DAOs because the independent transaction that would be happening in the both of the daos which is interdependent on each other for consistency and integrity of data will be treated as single transaction by Business module. If transactions in both of the daos has been successfully then only commit will be applied by Business class so that to intimate both the transactions happening in different daos to commit to convey that transaction was successful. But in case the transaction in one dao is unsuccessful and in other successful then Business class will call rollback so that transactions executing in both Session are roll backed. Business in this case has to apply the **Global Transaction facilities** and if we require security we need to apply the **security** and if after successful completion of transaction we need to send the mails we need to user then we require mail support given and failure handling support given by **JMS.** If u want track user information then we have to apply Loggers and to fix timers we need quartz support. As per standards we should not write these service codes inside the business class instead we need to write the codes for services separately under services layer. In that service layer we need to provide various capabilities like TX, Sec, JMS, Mail Service and Quartz Services. These services need to execute inside our business code. To combine the business code with our service code we use AOP where we need to attach our Business and Services and AOP will finally deliver the proxies to Controller and Controller need to talk to proxies and by using that proxy if we call any method() then that method will execute the Business code along with our Services code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TXN | Sec | JMS | Mail | Quartz |

BOI boi=

Business Proxy

Method()

ProxyFactoryBean

DAO 1

DI

BDS

DAOI

BOI

SF

DAOI daoI=

Business

DAO

DI DI

HT

Controller AOP

In most of case we need to run the service code before business code for example fi in Business there is deposit code then Security services need to run to check if the user is authenticated user or not i.e. Cashier, Manger in bank etc. So we have to check the role of person doing deposit hence here security code will execute before deposit code which is inside business class. In case transaction it need to start before business code executes and after business code it need to rollback or commit transactions. If any code inside the Business logic throws exception like Not valid day for transfer then that exception will be handled by JMS and will store the same business logic to be executed so that transfer request will be committed when Valid day comes i.e. from Monday to Friday. In fund transfer is success i.e. business code executes successfully then an intimation mail is sent. For this we need to provide mailing support.

In AOP Spring have given the interfaces to implement our services

If we want to execute our service before business code the we have interface called **MethodBeforeAdvice(I).** If we want to execute our services after the business code executes we have interface **AfterReturningAdvice(I).** If we want to execute our service before and after the business code execution then we have interface called **MethodInterceptor(I).** While our business code gets some exceptions then to provide exception handling service we have interface **ThrowsAdvice(I).** We can call all these classes as **AOP Advices classes.**

In AOP they have given Proxy class to combine the service code with Business code at runtime named **ProxyFactoryBean**. ProxyFacoryBean will generate a Proxy Class which are actually child class of Business class and in that proxy class it will combine the services code also. An interface reference of BOI is injected upon an instance of proxy class of Business Class. If method on business class is invoked on business interface reference then it will execute the business code along with Services code.

**AOP (Aspect Oriented Programming)**

**Aspect**: means a service also we can this as Service Oriented Programming

**Advice**: ServiceProvicer, it will provide aspects

**Pointcut**: A point or condition to execute the aspects for Business. To execute service we need to apply conditions using point cuts. For example we need to transaction service for DML operations and that to if we are dealing with multiple DAOs i.e. Withdrwal and deposit there is no need for transaction services in case of Select Query operations i.e. findBalance(). If we apply transaction service for a specific business class then it will apply TXn services on all operations. Instead of that if we want apply to txn facilities for few operations only then we need to use pointcut to give our condtions for txn service applications.

**Advisor**: It is Pointcut with Advice combination. **(Pontcut+ Advice)**

**AOP Proxy**: is a weaver. It will attach both business and service code and provide a proxy object which will combine the codes according to our business requirements.

**Target:** Business Object

**To implement AOP** i.e. combine the codes of Business and Services code which create a Proxy object, **we have 3 approaches**

1. **Programmatic Approach**
2. **Declarative Approach(Xml)**
3. **Annotations Approach**

Consider an example

Business class

Public class bank{

private int amt=5000;

Private String ac=sbi1000;

Public int deposit(int amt, String ac){

If (ac.equals(this.ac)){

return this.amt+amt;

}

else{

return 0;

}

}

}

Public class LogBefore implements MethodBeforeAdvisor{

Public void before(Object o, Object []param, Method m ){

Logger l = LoggerFactorr.getLogger(this.Class);

l.info(“Before deposit”); //Log along with time

}

}

ProxyFacoryBean pfb = new ProxyFactoryBean();

It has following methods

1. pfb. setTarget(Business Class Reference); //To add business method to proxy class
2. pfb. setInterceptorNames(String []names); //To add multiple services to proxy class

For above example

Bank b = new Bank();

LogBefore l = new LogBefore();

Pfb.setTarget(b);

Pfb.setInterceptorNames(new String[]{l} );

Bank proxy = Pfb.getObject(); //will return the Proxy object which can be type casted to parent class i.e. Bank Class.

**Tag information about ThrowsAdvice**

Public void afterThrowing([Method, param, target], ThrowableSubClass){

}

Class LogForExceptions implements ThrowsAdvice{

Public void afterThrowing(Exception e ){  
 }

Public void afterThrowing(RemoteException e){

}

Public void afterThrowing(Method m, Object [] params, Object target, Exception){

}

Public void afterThrowing(Method m, Object [] params, Object target, ServletException){

}

}

In case we want services to be run for only few methods in our application then we may need to use pointcut Classes. We have **2 Pointcut Classes**

1. **StaticMethodMatcherPointcut**
2. **NameMatchMethodPointcut**

Using PointCut along with Services

Pointcut

Advisor

Advice/Services

Bank

AOP

DI

We can’t combine the Pointcut code to Proxy Object we have to first combine the code of Advices/services and Pointcut to some Advisor classes then Advisor class object can be injected into the ProxyFactoryBeam object. To implement Advisor we have following Advisors

1. DefaultPointcutAdvisor
   1. Void setAdvice(Advice advice)
   2. Advice getAdvice()
   3. setPointcut(Pointcut pointcut)
2. RegExMethodPointcutAdvisor