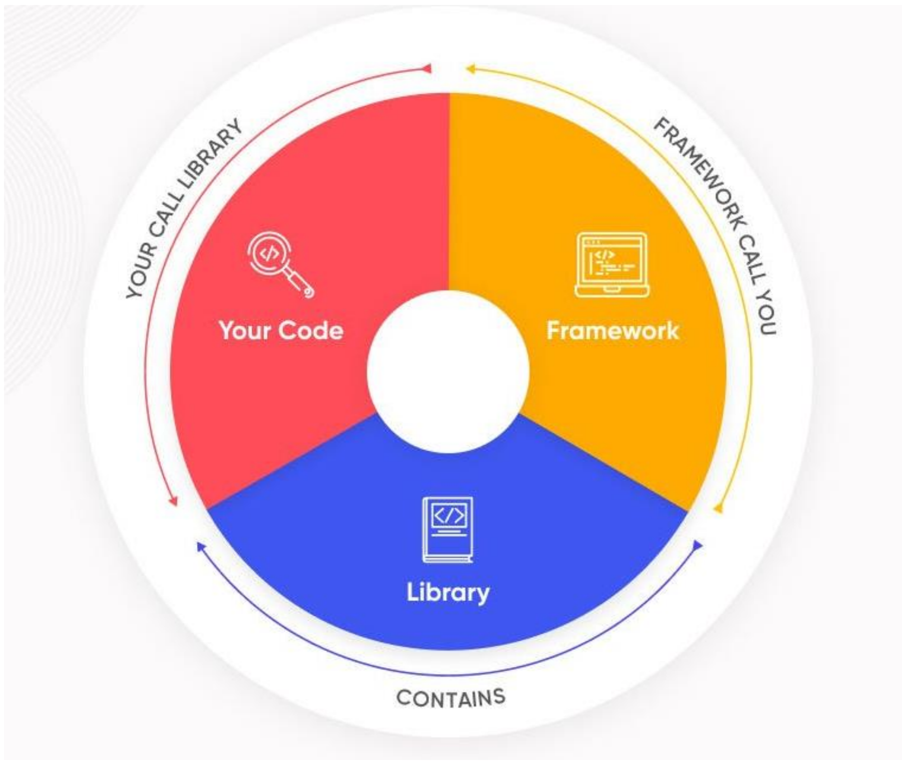
- Spring IOC:
    1. Containers
    2. DI (dependency injection)
       Setter / Constructor (primitive, bean, collections)
    3. Auto wiring (xml and annotation)
    4. Bean Lifecycle
    5. Factory Bean (static, non-static)
    6. I18N (internationalization)
    7. Lookup Method /Replace Method
- Spring JDBC:
    1. Connection pool
    2. Jdbc Template /NamedParamter Jdbc Template
- Spring ORM:
    1. Spring Hibernate Integration oracle
    2. Hibernate Template
- Spring MVC:
    1. Spring MVC architecture
    2. Controller
- Spring Boot:
    1. Spring vs Spring Boot
    2. Spring Initializer / STS
    3. Spring Boot Annotations
- Micro services:
    1. Rest Service Implementation
    2. Rest template vs web client vs Feign
    3. Load Balancing Eureka
    4. API gateway
    5. Cloud Config Server
    6. Exception Handling/ Hystrix

- Swagger / Mockito

What is framework?

- ❖ Definition: The basic structure, arrangement, or system.
- ❖ A framework in programming is a tool that provides ready-made components or solutions that are customized in order to speed up development.
- ❖ A framework can include support programs, compilers, code libraries, toolsets, and APIs to develop software and create systems. Open-source frameworks are always being updated and improved

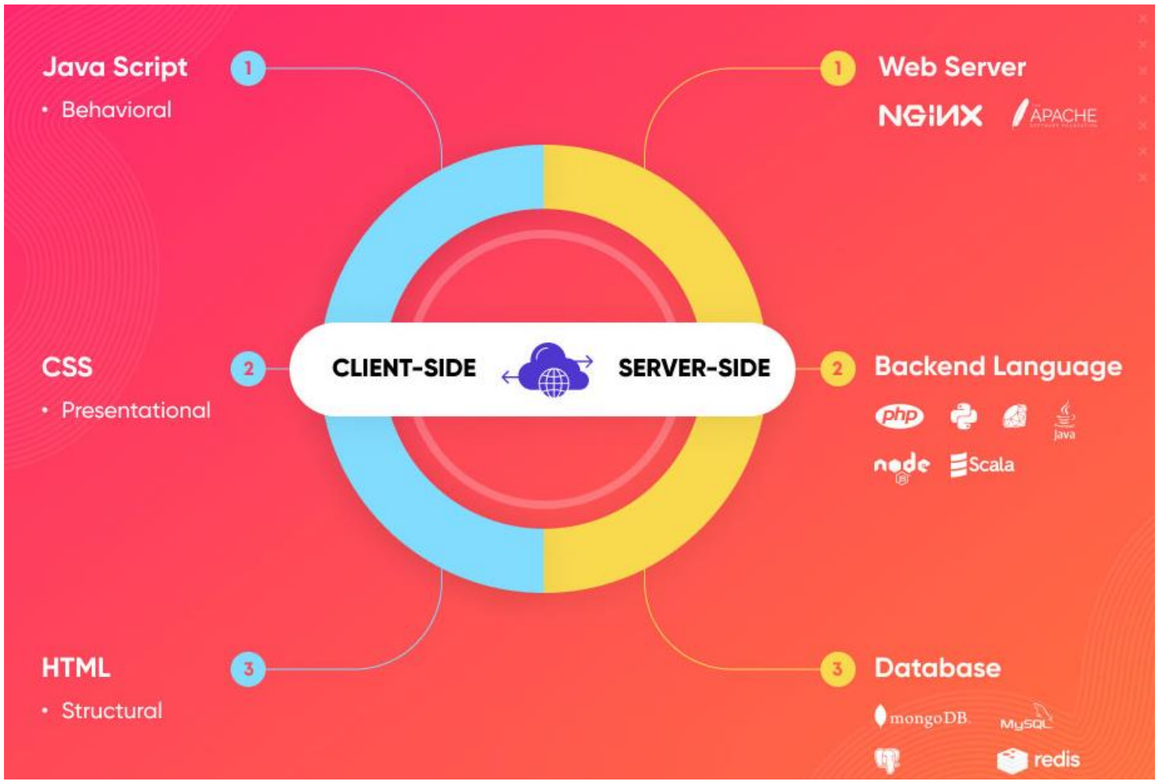


- ❖ The purpose of a framework is to assist in development, providing standard, low-level functionality so that developers can focus efforts on the elements that make the project unique

Good Framework should have below :

- ❖ **Functionality** – choose a framework that provides the functionality needed for the project at hand
- ❖ **Consistency** – a framework can assist in consistency for large or distributed teams
- ❖ **Documentation**– choose a framework that has well-documented code and that provides implementation training
- ❖ **Active Community** – frameworks are only as strong as the user base of support. Choose a framework that is well-established with an active user base

Web framework: combination of both front end and back end framework



Front-End Frameworks – Front-end frameworks (client-side frameworks) provide basic templates and components of HTML, CSS, and JavaScript for building the front-end of a website or web app.

- Angular
- React JS
- Vue JS

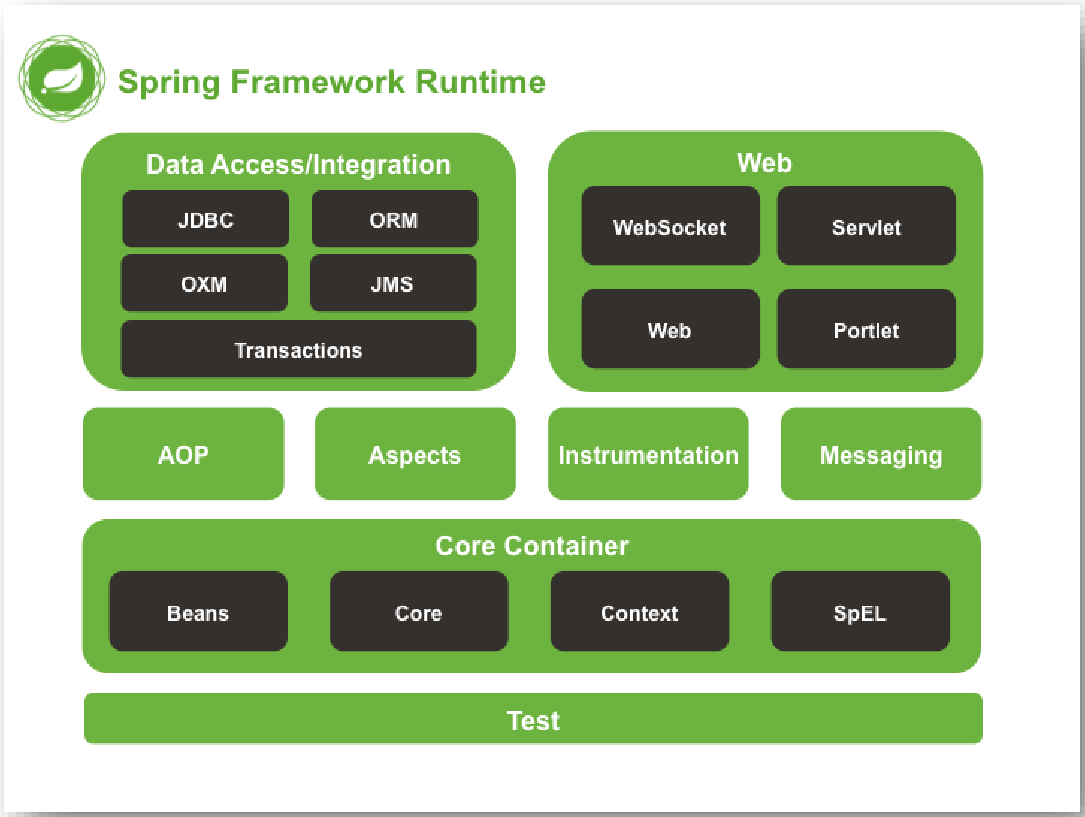
Back-End Frameworks – Back-end frameworks (server-side frameworks) provide generic functionalities that can be assembled or built upon to assist in development

- Spring
- Hibernate
- JPA
- Ibatis

Programming frameworks are an important tool in developing software for today’s marketplace in a way that is rapid, scalable, reliable, and secure. Whether the product is a website or mobile app, frameworks provide a starting point to solve common problems

Developer(s)	**Pivotal Software**
Initial release	**1 October 2002**
Developer:	**Rod Johnson, is the founder of the Spring Framework, an open source application framework for Java, Creator of Spring, CEO at SpringSource.**

<u>Modules</u>:



Modules

I. Ioc framework (inversion of control container & Dependency Injection)
II. Dao(DataAccessObject) framework
 Spring JDBC
 Spring ORM.
III. Transaction Management framework(for Global Transactions)
IV. Remote Access Framework
V. Model view Controller Framework

Why road Johnson named to his frame work name as **spring?**
Firstly the name for this framework interface21 then later changed it to spring
 Spring represented a fresh start after the "winter" of traditional J2EE.
In J2ee

i. For building Enterprise applications if you depends on traditional j2ee there are lot of configurations and container dependencies are there and all the time containers are not freeware
ii. And The code will become tightly coupled and heavyweight

IOC

IOC giving support of containers and dependency injection support

Container ?:
A basic spring container responsibilities :
i. Containers provides Dynamic Object creation support, by reading data from configuration files
ii. Containers can support Dependency lookup and Dependency Injection.
iii. Containers can manage object lifecycles
iv. Containers can provide event handlers
v. Containers can support internationalization

If you have experience of Web container (like tomcat) &   web+ Ejb Container (like Weblogic)

EX: Tomcat container by reading web.xml file from application

i. It will create objects of servlets and Filters and Based on events It will create instances of Listners
ii. And web.xml file context and Config params Tomcat container will make available to servlet to read data dynamically from web.xml file
iii. And It can manage lifecycles for Servlets
iv. And through http request header It will supply locale parameters and browser Info..

Spring is having 3 containers those 3 containers are

i. Core container(BeanFactory)
ii. J2ee Container(ApplicationContext)
iii. Web container(WebApplicationContext)

i. How to create Object ?

   Create java project:

   Add jars to class path:

   i. Spring-core.jar,
   ii. Spring-beans.jar,
   iii. commons-logging.jar

<u>HelloWorld.java</u>

```
public class HelloWorld {
public void hello(){
	System.out.println("HelloWorld..");
}
}
```

<u>Spring.xml:</u>

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
	<bean id="hw" class="HelloWorld"/>
</beans>
```

<u>Test.java</u>

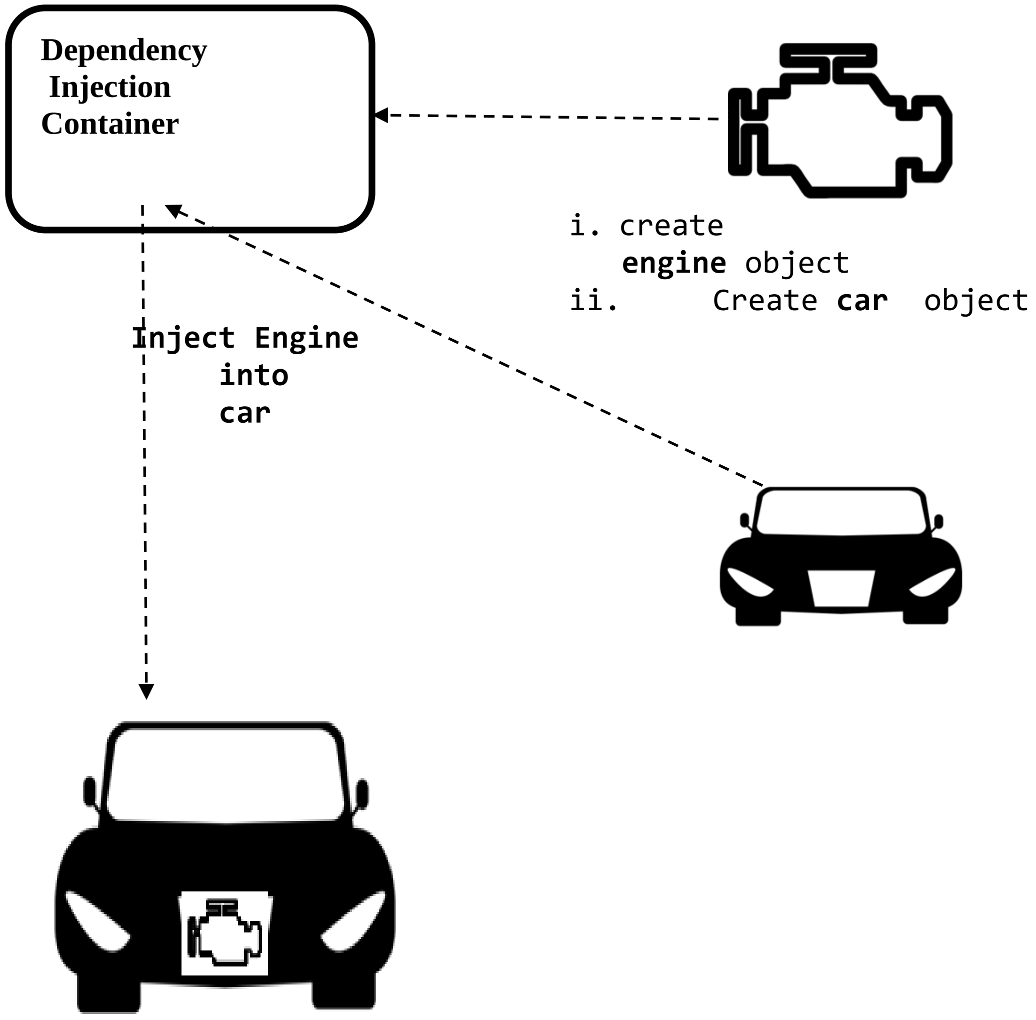
```
public class Test {
public static void main(String[] args) {
     /*loading spring.xml file into core
     container BeanFactory from class path
     */
     Resource r=new ClassPathResource("spring.xml");
     BeanFactory b=new XmlBeanFactory(r);
     /*To create HelloWorld object */
     HelloWorld h=(HelloWorld)b.getBean("hw");
     h.hello();
          }
}
```

ii. dependency injection?

Passing required parameters to A Object from configuration file through spring container Is dependency Injection

Ex:

If you have a class Car and If it Contain Dependency Engine



EX: How to do primitive and secondary types DI

There are 2 type of dependency injections
By using setter methods (or) parameterized constructors

For setter method we have to use

```
 <property name=”parametername”>
<value>value</value>
<property >
```

For parameterized constructors

```
<constructor-arg>
<value>value</value>
<constructor-arg>
```

.

```
class Engine{
private String enginename;
public void setEngineName(String enginename)
                {
               this. Enginename= enginename;
                }
}
```

```
class Car{
private Engine engine;
public void setEngine(Engine engine)
        {
        this.engine= engine;
        }
        public Engine getter(){
                return engine;          }
        }
```

<u>spring.xml</u>

```
<beans>
<bean id="eng" class="Engine" >
        <property name=" enginename">
        <value>"Honda"</value>
        </property>
<bean>
</bean>
<bean id="car" class="Car">
<property name="engine">
<ref bean="eng"/>
<bean>
</beans>
```

```
class Client{
public static void main(String… arg)
        {
        Resource r=new ClassPathResource("spring.xml");
        BeanFactory b=new XmlBeanFactory(r);
                Car c=(Car) b.getBean("car");
                System.out.println(c.getEngine().getEngineName());
        }
}
```

In above example Car is having dependency of engine
And Engine is having dependency of enginename
Here Enginename is primitive type
And Engine is secondary data type
So DI injection of

i. primitive types we can do by using **<value>"DATA"</value>**under **<property>** tag

For Secondary type we should use

ii. **<ref bean="ref"/>** under **<property>** tags

---

iii. How to solve constructor DI ambiguity
    i. By using type attribute
    ii. By using index attribute

```
class Car{
private Engine engine;
private AC ac;
public Car(Engine engine, AC ac){
this.engine= engine;
this.ac=ac;
        }
}

class AC{
private String acname;
private void setAcname(String acname)
        {
        this.acname=acname;
        }
}
```

Spring.xml

```
<beans>
<bean id="eng" class="Engine" >
          <property name=" enginename">
          <value>Honda</value>
          </property>
<bean>
```

```
<bean id="ac" class="AC">
<property name="acname">
          <value>"Auto Cool"</value>
          </property>
</bean>
```

```
<bean id="car" class="Car">
<constructor-arg  type="Engine"  index="0">
<ref bean="eng"/>
<constructor-arg >

<constructor-arg  type="AC" index="1">
<ref bean="ac"/>
<constructor-arg >
<bean>

</beans>
```

IV. How to do Dependency injection of Arrays and Collection parameters

```
class  Test
{
private String[] coursenames;
private List fruits;
private Set cricketers;
private Map countrycapital;
private Properties dbproperties;
//generate getters and setters
}
```

```
<?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:util="http://www.springframework.org/schema/util"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd">
```

```
<bean id="t" class="Test">
      <property name=" coursenames ">
       <list>
          <value>Struts</value>
          <value>Spring</value>
          <value>Hibernate</value>
```

```
<property name=" countrycapital ">
<util:map map-class="java.util.HashMap">
   <entry>
         <key><value>India</value></key>
         <value>Delhi</value>
    </entry>
```

```
class Client
{
Public static void main(String…args)
{
AplicationContext ap=new ClassPathXmlApplication("spring.xml");
Test t=(Test)ap.getBean("t");
System.out.println(t.getMethods());//cal all possible getters
}
}
```

P and C namespace:
P:propert injection
C:Constructor Injection

```
public class Car {
private String name;
private Engine engine;

public Car(String name, Engine engine) {
super();
this.name = name;
this.engine = engine;
}

public String getName() {
return name;
}

public Engine getEngine() {
return engine;
}
}
```

```
public class Engine {
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
        }
```

Spring.xml

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

   <bean id="car" class="Car" c:name="Audi" c:engine-ref="engine"/>
   <bean id="engine" class="Engine" p:name="benz"/>
    </beans>
```

```
public class Client {
public static void main(String[] args) {

        ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
        Car car=(Car)ap.getBean("car");
        System.out.println(car.getName());
        System.out.println(car.getEngine().getName());

}
}
```

Collection Merging

```
public class Person {
 private List<Address> addresses;
//generate getters and setters
}

Class Address{
Private String street;
Private String state;
Private String zip;
//generate getters and setters
}
```

```
<beans>
<bean name="baseperson" class="Person" abstract="true">
 <property name="addresses">
  <list>
   <bean class="Address" p:street="Street1" p:state="State1" p:zip="001"></bean>
   <bean class="Address" p:street="Street2" p:state="State2" p:zip="002"></bean>
   <bean class="Address" p:street="Street3" p:state="State3" p:zip="003"></bean>
  </list>
 </property>
</bean>
```

```
<bean name="person" class="Person" parent="basePerson">
 <property name="addresses">
  <list merge="true">
   <bean class="Address" p:street="Street4" p:state="State4" p:zip="004"></bean>
  </list>
 </property>
</bean>
</beans>
```

```
class Client     {
public static void main(String[] args) {
ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
Person p=(Person)ap.getBean("person");
               }
       }
```

Inner Bean:

```
class A
{
B b;
//setter
}
Class B{

}
```

simpleDI	Innter Bean DI

`<beans>`
`<bean id=”a” class=”A”>`
`<property name=”b” ref=”b”/>`
`</bean>`
`<bean id=”b” class=”B”/>`
`</beans>`
	`<beans>`
	`<bean id=”a” class=”A”>`
	`<bean class=”B”/>`
	`</bean>`
	`</beans>`

V. Auto wiring

Auto wiring is A concept doing dependency injection by using auto searching and possible parameters injecting
We can do this autowiring only for secondary types
And autowiring have many types those are

Autowiring Mode	Description
No	No autowiring is performed. All references to other beans must be explicitly injected. This is the default mode.
byName	Based on the name of a property, a matching bean name in the IoC container will be injected into this property if it exists.
byType	Based on the type of class on a setter, if only one instance of the class exists in the IoC container it will be injected into this property. If there is more than one instance of the class a fatal exception is thrown.
Constructor	Based on a constructor argument's class types, if only one instance of the class exists in the IoC container it will be used in the constructor.
Autodetect	If there is a valid constructor, the constructor autowiring mode will be chosen. Otherwise if there is a default zero argument constructor the byType autowiring mode will be chosen.

I) autowire="no" : in case of no ,user manually need to do the dependency injection

```
<beans>
<bean id="eng" class="Engine" >
        <property name=" enginename">
        <value>"Honda"</value>
        </property>
<bean>
--------------------------------------------------------------
<bean id="ac" class="AC">
<property name="acname">
        <value>"Auto Cool"</value>
        </property>
</bean>
--------------------------------------------------------------
<bean id="car" class="Car" autowire="no">
<constructor-arg  type="Engine"  index="0">
<ref bean="eng"/>
<constructor-arg >
```

<constructor-arg type="AC" index="1">
<ref bean="**ac**"/>
<constructor-arg >   <bean> **</beans>**

II) autowire="byType" :   in case of byType ,same type bean ref it will search in spring xml file if find only one possible type it container will do dependency injection if it find multiple then container will get ambiguity to decide what need to inject .

in this case for the required property autowire-candidate="true". And for the un-required property autowire-candidate="false" you need to mention then false ref will not participate in autowiring then the remaining one container will do dependency injection

i. For single possible entry

<u>spring.xml</u>
<beans>
<bean id="**eng**" class="**Engine**" >
        <property name=" enginename">
        <value>"Honda"</value>
        </property>
<bean>
</bean>
<bean id="**car**" class="**Car**" autowire="**byType**">
<!-- no need to do di- ->
</beans>

---

ii. For multiple possible entry's

<u>spring.xml</u>
<beans>
<bean id="**eng1**" class="**Engine**" autowire-candidate=**"false"**>
        <property name=" enginename">
        <value>Honda</value>
        </property>
<bean>
<bean id="**eng2**" class="**Engine**" autowire-candidate=**"true"**>
        <property name=" enginename">
        <value>BMW</value>
        </property>
<bean>

```
</bean>
<bean id="car" class="Car" autowire="byType">
<!--  auto DI- -></beans>
```

iii) autowire="byName": for autowire byName container try to search for the same parameter reference name in spring.xml file if it find a possible bean with same reference name with same type that bean it will to automatic DI

```
class Engine{
private String enginename;
public void setEngineName(String enginename)
                    {
                   this. Enginename= enginename;
                    }
}
```

```
class Car{
private Engine engine;
public void setEngine(Engine engine)
          {
          this.engine= engine;
          }
          public Engine getter(){
                    return engine;          }
          }
```

<u>spring.xml</u>

```
<beans>

</bean>
<bean id="car" class="Car" autowire="byName">
<!--  no need to do di- ->
</bean>
```

```
<bean id="engine" class="Engine">
          <property name=" enginename">
          <value>Honda</value>
          </property>
<bean>
<bean id="engine1" class="Engine">
          <property name=" enginename">
          <value>BMW</value>
          </property>
<bean>
</beans>
```

iii) autowire=”Constructor: In case of Constructor container will search for same type with single entry if it find then it will to automatic Constructor **DI**

```
class Car{
private Engine engine;
public Car(Engine engine)
                    {
this.engine= engine;
                    }
             }
```

<u>spring.xml</u>

```
<beans>

</bean>
<bean id=”car” class=”Car” autowire=”constructor”>
<!--  no need to do di- ->
</bean>
```

```
<bean id=”engine1” class=”Engine”>
          <property name=” enginename”>
          <value>Honda</value>
          </property>
<bean>
</beans>
```

---

v) autowire=”autodetect”:
If there is a valid constructor, the constructor autowiring mode will be chosen. Otherwise if there is a default zero argument constructor the byType autowiring mode will be chosen.

VI)

Singleton (vs) Prototype

Singleton:

A singleton is a class that allows only a single instance of itself to be created and gives access to that created instance. It contains static variables that can accommodate unique and private instances of itself. It is used in scenarios when a user wants to restrict instantiation of a class to only one object. This is helpful usually when a single object is required to coordinate actions across a system.

Ex:

```
class Test
{
private static Test t=null;
Private Test(){
}
public static Test getInstance()
        {
If(t==null)     {
t=new Test();
return t;
                }
else            {
return t;
                }
                }
        }

}
```

Prototype:

Prototype design pattern is used in scenarios where application needs to create number of instances of a class, which has almost same state or differs very little.

```
class Test implements Clonable
{
        @Override
          public Test clone() throws CloneNotSupportedException {
            System.out.println("Cloning Test object..");
            return (Test) super.clone();
        }
}
```

In Spring How to Make a **Bean** singleton and prototype

```
public class Car
{
public void drive(){
System.out.println("drive..");
}
}
```

For Singleton:

spring.xml

```
<beans>
</bean>
<bean id="car" class="Car"   scope="singleton"  />
</beans>
```

```
Class Test
        {
p  s v m(Str…arg)
{
ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
        Car car1=(Car)ap.getBean("car");
        Car car2=(Car)ap.getBean("car");

System.out.println(car1==car2);//true

}
}
```

For prototype:

Spring.xml

```
<beans>
<bean id="car" class="Car"   scope="prototype"  />
</beans>
```

```
Class Test
{
```

```
p  s v m(Str…arg)
{
ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
          Car car1=(Car)ap.getBean("car");
          Car car2=(Car)ap.getBean("car");

System.out.println(car1==car2);//false

}
}
```

vii) Differences between beanfactory and applicationcontext

	BeanFactory	ApplicationContext
i.	It is a Core container, and for Spring it is the base container	It is a J2ee container it is child class for BeanFactory
ii.	It will do lazy instantiation (on demand objects will create)	It will do eager instantiation(at loading it will instantiate)
iii.	Does not support the Annotation based dependency Injection.	Support Annotation based dependency Injection.- like.. @Autowired
iv.	Does not Support events listeners of spring	Application contexts can publish events to beans that are registered as listeners
v.	Does not support way to access Message Bundle(internationalization (I18N)	Support internationalization (I18N) messages.

Factory ,
Static Factory,
InstanceFactory

A factory class contains factory method to instantiate class object
That factory method can be static or instance method

i. Incase of Staticfactory static method will return other class object

Ex:
1.Class c = Class.forName("ClassName");
2. Thread t = Thread.currentThread();

3. Calendar cl = Calendar.getInstance();

```
Class Test
{         public Test(){}
}
Class TestFactory{
private static Test t;
private TestFactory(){ }
public static Test getObject()
{
if(t==null){
t=new Test();
return t;  }
          else  {   return t;  }
}
```

ii.Incase of InstanceFactory object methods will return other class object

```
class Test
{
}
class TestFactory{
private Test t;
public  Test getObject()
{
if(t==null){
                    t=new Test();
                    return t;
                    }

          else  {
                    return t;
                    }
}
```

The main aim is here to create instance of other class object, And base on requirement it need to provide support to change the object types

viii) Factory Implementation using Spring?
By using FactoryBean interface we can implement Factory .
EX:
To provide connection object to the application through a factory
FactoryBean interface is having three method

```
public Object getObject() throws Exception ;
public Class getObjectType();
public boolean isSingleton()
```

getObject method is the FactoryMethod so from this method you return Connection object to the user
when user request for factory instance from spring containers
the factory can return any object from getObject Method

```
public class DBConnFactory implements FactoryBean
{
          private String driver;
          private String url;
          private String username;
          private String password;
          //write setter methods for DI
          private Connection con;

@Override
public Object getObject() throws Exception {
          Class.forName(driver);
          con=DriverManager.getConnection(url,username,password);
          return con;
}
@Override
public Class getObjectType() {
                                        return con.getClass();
                    }
@Override
public boolean isSingleton() {
                    return true;
          }
}
```

spring.xml

```
                    <beans>
                    <bean id=”dbfactory” class=”DBConnFactory” scope=”singleton”>
           <property name=”driver” value=”oracle.jdbc.OracleDriver”/>
           <property name=”url” value=”jdbc: oracle:thin:@localhost:1521:xe”/>
           <property name=”username” value=”system”/>
           <property name=”password” value=”manager”/>
                    </bean>
                    </beans>
```

```
                class Test{
                public static void main(String args[])
                        {
                ApplicationContext ap=new
        ClassPathXmlApplicationContext("spring.xml");
Connection con1=(Connection)ap.getBean("dbfactory");
Connection con2=(Connection)ap.getBean("dbfactory");
                System.out.println(con1==con2);

                }
                }
```

Here container will return us oracle connection object from factory when try to request for the factory object and it will make it is as A singleton

This Implementation we can use for

i. Connection pool objects (datasource object lookup from JNDI)
ii. For hibernate SessionFactory
iii. And for implementing RMI Factory
iv. And For Proxy implemations like AOP proxyFactoryBean

StaticFactory Implementation in Spring

```
public class Test
{
public Test() {
System.out.println("Test");
}

}
```

```
public class TestFactory
{
```

```
public TestFactory() {
}
public static Test getObject(){
return new Test();
}
}
```

```
<beans>
<bean id="tf" class="TestFactory" factory-method="getObject"/>
</beans>
```

```
public class Client {
public static void main(String[] args) {
ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
Test  t=(Test)ap.getBean("tf");
Test  t1=(Test)ap.getBean("tf");
System.out.println(t==t1);
}
}
```

Instance Factory Implementation in Spring

Though instance methods we can return factory instance

```
public class TestFactory
{
public TestFactory() {
}
public Test getObject(){
return new Test();
}
}
```

Spring.xml

```
<beans>
<bean id="tf" class="TestFactory"/>
<bean id="tfactory" factory-bean="tf" factory-method="getObject"/>
</beans>
```

```
public class Client {
public static void main(String[] args) {
ApplicationContext ap=new ClassPathXmlApplicationContext("spring.xml");
Test  t=(Test)ap.getBean("tfactory");
Test  t1=(Test)ap.getBean("tfactory ");
System.out.println(t==t1);
}
}
```

x) **BeanLifeCycle implementations**

1. programmatic approach
2. declarative approach
3. Annotation approach

i. Programatic Approach: bean class need to implement from InitializingBean and DisposableBean interfaces

InitializingBean(I): public void **afterPropertiesSet**() throws Exception

DisposableBean(I): public void **destroy**() throws Exception

```
public class Test implements InitializingBean,DisposableBean{
private String driver,url,user,pass;
private Connection con;
        //generate setter methods
@Override
public void afterPropertiesSet() throws Exception {
Class.forName(driver);
con=DriverManager.getConnection(url,user,pass);
System.out.println("con"+con);
}
public void save(String name,String email,String address)
{       try{
PreparedStatement ps=con.prepareStatement("insert into student values(?,?,?)");
ps.setString(1, name);
ps.setString(2, email);
ps.setString(3, address);
ps.executeUpdate();
}catch(Exception e){}
}
@Override
public void destroy() throws Exception {
con.close();
System.out.println("Destroy");       }          }
```

Spring.xml

```
<bean id="t" class="Test">
<property name="driver">
<value>oracle.jdbc.OracleDriver</value>
</property>

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

<property name="user">
<value>system</value>
</property>

<property name="pass">
<value>manager</value>
</property>
</bean>
</beans>
```

```
public class Client
{
public static void main(String[] args) {
ConfigurableApplicationContext cap=
new ClassPathXmlApplicationContext("test.xml");
Test t=(Test)cap.getBean("t",Test.class);
t.save("abc", "abc@hmail.com", "HYD");
System.out.println("save..success....");
cap.close();
}
}
```

---

ii. Declarative Approach:

Bean class:

```
public class Test{
private String driver,url,user,pass;
private Connection con;
//gerenate public setter methods

public void conInit() throws Exception {
Class.forName(driver);
con=DriverManager.getConnection(url,user,pass);
System.out.println("con"+con);
}

public void save(String name,String email,String address)
{
try{
PreparedStatement ps=con.prepareStatement("insert into student values(?,?,?)");
ps.setString(1, name);
ps.setString(2, email);
ps.setString(3, address);
ps.executeUpdate();
}catch(Exception e){}
}
public void conDestroy() throws Exception {
con.close();
System.out.println("Destroy");

}


}
```

Spring.xml

```
<bean id="t" class="Test" init-method="conInit" destroy-method="conDestroy">
```

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

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

<property name="user">
<value>system</value>
</property>

<property name="pass">
<value>manager</value>
</property>

</bean>
</beans>
```

```
public class Client
{
public static void main(String[] args) {
ConfigurableApplicationContext cap=
new ClassPathXmlApplicationContext("test.xml");
Test t=(Test)cap.getBean("t",Test.class);
t.save("abc", "abc@hmail.com", "HYD");
System.out.println("save..success....");
cap.close();
}
}
```

3. Annotation Approach :

```
public class Test{
private String driver,url,user,pass;
private Connection con;
//gerenate public setter methods
@PostConstruct
public void conInit() throws Exception {
Class.forName(driver);
con=DriverManager.getConnection(url,user,pass);
System.out.println("con"+con);
}

public void save(String name,String email,String address)
{
try{
PreparedStatement ps=con.prepareStatement("insert into student values(?,?,?)");
ps.setString(1, name);
ps.setString(2, email);
ps.setString(3, address);
ps.executeUpdate();
}catch(Exception e){}
}
 @PreDestroy
public void conDestroy() throws Exception {
con.close();
System.out.println("Destroy");
}
}
```

Spring.xml

```
<beans>
<bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
</beans>
```

Instead of using AutowiredAnnotationBeanPostProcessor
You can use <context:annotation-config> and <context:component-scan>

Spring.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:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:annotation-config/>
</beans>
```

Event Handling

Spring Containers are proving Event handling support to build Event handling Mechanism
A event means an Acton to execute a particular method of a class

So Container will support 4 types events

i. Container starting event
ii. Container refresh event
iii. Container stop event
iv. Container close event

To start or execute code on this events we need implement classes here by using ContexttListnerEvent

i. Context Start event

```
public class ContextStartEH implements ApplicationListener<ContextStartedEvent>
{
@Override
	public void onApplicationEvent(ContextStartedEvent arg0) {
		System.out.println("context started");
	}
}
```

ii. Context Refresh Event

```
public class ContextRefreshEH implements
ApplicationListener<ContextRefreshedEvent> {

	@Override
	public void onApplicationEvent(ContextRefreshedEvent arg0) {
		System.out.println("Context reffreshed");

	}

}
```

iii. Context Stop event :

```
public class ContextStopEH implements
ApplicationListener<ContextStoppedEvent> {
@Override
public void onApplicationEvent(ContextStoppedEvent arg0) {

        System.out.println("CONTEXT stopped");
    }

}
```

iv. Context close Event:

```
public class ContextCloseEH implements
ApplicationListener<ContextClosedEvent> {
@Override
    public void onApplicationEvent(ContextClosedEvent arg0) {
            System.out.println("context closed");
    }
    }
```

<u>Spring.xml</u>

```
<beans>
<bean class="ContextStartEH"/>
<bean class="ContextStopEH"/>
<bean class="ContextRefreshEH"/>
<bean class="ContextCloseEH"/>
</beans>
```

```
public class Client {
public static void main(String[] args) {

    ConfigurableApplicationContext cap=new
ClassPathXmlApplicationContext("spring.xml");
    cap.start();
    cap.refresh();
    cap.stop();
    cap.close();
}
}
```

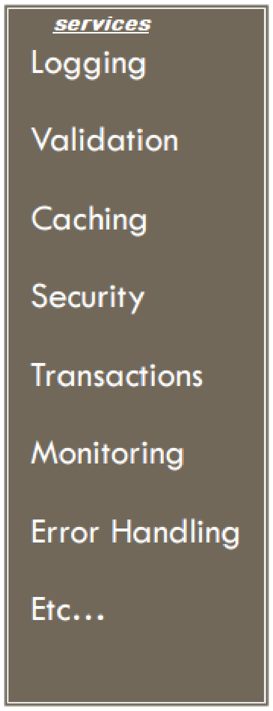
AOP

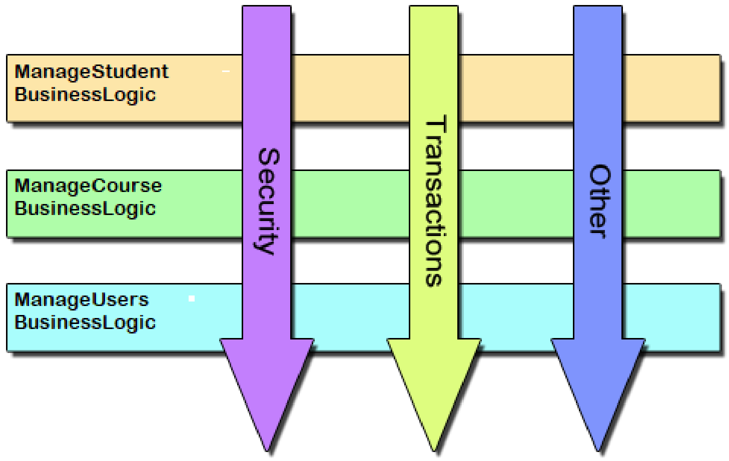
(Aspect Oriented program)

The aim of AOP is to loosely couple middleware services with spring business components. Middleware services will execute before or after our actual business component method executions. Spring AOP provides declarative enterprise/middleware services to our business components.

Aspect == Middleware service

Middleware services are ***Logging, Transaction, Security, Messaging, Testing etc.***




ManageStudent BusinessLogic
ManageCourse BusinessLogic
ManageUsers BusinessLogic
Security
Transactions
Other

Aspect - Aspect is a service. In J2EE we use services such as TX, Security, Logging, Testing etc. Writing a program to create and add services to business components is called as AOP (Aspect Oriented Programming). It is compliment to OOP but not supplement to OOP.

Joinpoint - Joinpoint is a point in execution of class when the aspect want to be executed. Such as method execution or any exception raised during method execution.

Aspects / services are implemented in a sub class of Advice classes. Advice classes are classified into 4 types i) Before Advice ii) After Advice iii) Around Advice iv) Throws Advice. But when a Advice and target objects are added to AOP, Advice will execute for every method call of target object. In advice classes we can't specify to which target object methods the Advice want to execute.

Joinpoints are expressed in Pointcut classes. Pointcut classes implement one method called matches() that returns boolean value true if required class and

method matches, otherwise returns false. Pointcut classes are used to control the Advice class execution.
Target object - Spring components are called as Target object on to whom we want to apply services/advices. For instance CustomerController, EnquiryController
AOP Processor - ProxyFactoryBean class is called as AOP processor class. AOP processor takes one taget object, many advice class objects, generate and give us one proxy object. Functions exist in this class are addAdvice(), setTarget(), addAdvisor() and getObject().
Proxy object - The outcome of AOP processor is Proxy object. Proxy is an wrapper developed around target object with all the Advice classes. Client instead of calling functions of target object, if functions are invoked on proxy object proxy will send function call through all the advices to the actual target object method @ the end of execution. That is how proxy can execute many services before and after target object method executions.

Advisors - Is a class which controls the advice execution during method calls.
Pointcut+Advice=> Advisor
Aspect == Advice
Joinpoint == Pointcut
Proxy object ! = target object
proxy object == target object+advice classes
Advice: Invoking Aspect code at a particular point of time. The point of executions are before method, after method, around method calls, and during method execution failure.
Target object: Our business components are target objects.
AOP Proxy: AOP Proxy is a sub class of business component generated by AOP container. When user invokes functions on AOP proxy object the request passes via many advice class finally to business component functions.
AOP Container: ProxyFactoryBean class is AOP container. This class takes one target object, many advice class objects, generates and give one proxy object to client.
The primary concept of AOP is not to know how implement middleware services. Our primary concern is to implement classes those runs before, after, before-after our method executions. Hence which classes we need to implement those runs before, after and before-after our method executions. To implement such classes in Spring 4 interfaces and one class are provided:

4 Advice interfaces in Spring:

```
public interface MethodBeforeAdvice {
        public void before(Method m, Object args[], Object target);
}
```

Any class sub classing from this interface and implement the above method will execute before business component execution.

```
public interface AfterReturningAdvice {
	public void afterReturning(Object retVal, Method m, Object args[], Object
target);
}
```

Any class sub classing from this interface and implement the above method will execute after business component execution.

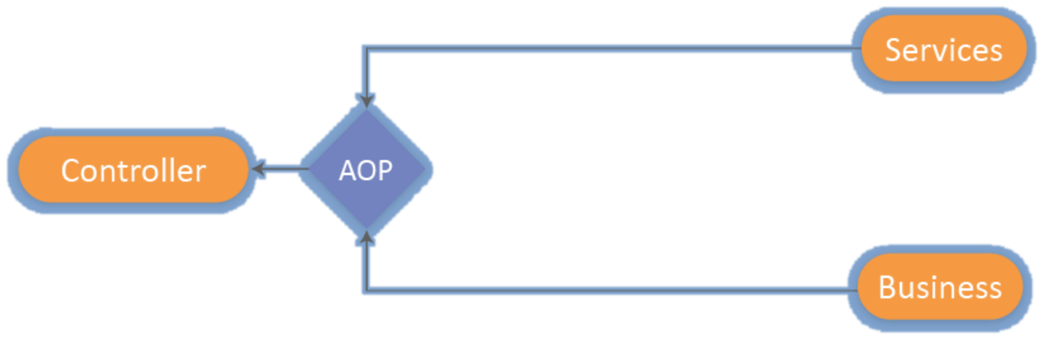
```
public interface MethodInterceptor { // we will also call this interface as
AroundAdvice
	public Object invoke(MethodInvocation);
}
```

Any class sub classing from this interface and implement the above method will both before and after business component execution. Infact this class is not called two times. While implementing the above method MethodInvocation class is having one method called proceed(). During proceed() method call the actual call goes to business method. Once after the method execution is completed the control will returned back invoke() method, the code written in invoke() method will execute as post process and then response is returned to client.

```
public interface ThrowsAdvice { // we will also call this interface as
AroundAdvice
	public void afterThrowing([Method m, Object args[], Object target],
Throwable t);
}
```

Any class sub classing from this interface and implement the above method will execute only during when exception is raised business component execution.

In Spring one class is given named ProxyfactoryBean. This ProxyFactoryBean is called as Spring AOP container. Its job is to take one target object and many advice classes, generates a proxy to target object and returns client. If client invokes methods on proxy object, the proxy will invoke all the advices before and after business method calls. Such a proxy class is generated at runtime.



+ Create a Class

Name: A

```
public class A {
  public void a() {
    System.out.println("a() method called");
  }
  public void b() {
    System.out.println("b() method called");
  }
  public void c() {
    System.out.println("c() method called");
  }
  public void d() {
    throw new NullPointerException();
  }
}
```

+ Create a Java class

Name: MBA

super interface: MethodBeforeAdvice

```
public class MBA implements MethodBeforeAdvice  {
  public void before(Method m, Object args[], Object target)   {
    System.out.println("MBA called before "+target.getClass().getName()+",
"+m.getName());
  }
}
```

+ Create a java class
Name: ARA
super interface: AfterReturningAdvice

```
public class ARA implements AfterReturningAdvice  {
  public void afterReturning(Object output, Method m, Object args[], Object target)   {
    System.out.println("ARA called after "+target.getClass().getName()+",
"+m.getName());
  }
}
```

+ Create java class
Name: AA
super interface: MethodInterceptor

```
public class AA implements MethodInterceptor {
  public Object invoke(MethodInvocation mi)   {
    System.out.println("AA called before
"+mi.getTarget().getClass().getName()+","+mi.getMethod().getName());
    Object output=mi.proceed();
    l.info("AA called after
"+mi.getTarget().getClass().getName()+","+mi.getMethod().getName());
    return output;
  }
}
```

+ Create one more class
Name: TA
super interface: ThrowsAdvice

```
public class TA implements ThrowsAdvice {
  public void afterThrowing(Method m, Object args[], Object target,
NullPointerException npe)   {
    System.out.println("TA raised during "+target.getClass().getName()+",
"+m.getName());
  }
}
```

+ Create one more java class
Name: Client

```
public class Client {
  public static void main(String rags[]) throws Exception   {
    A a=new A();
    a.a();
    ProxyFactoryBean pfb=new ProxyFactoryBean();
    pfb.setTarget(a);
    pfb.addAdvice(new MBA());
    pfb.addAdvice(new ARA());
    pfb.addAdvice(new AA());
    pfb.addAdvice(new TA());
    A aProxy=(A)pfb.getObject();
    aProxy.a();
    aProxy.b();
    aProxy.c();
    aProxy.d();
    /* programmatic way of weaving middleware services with business component.
*/
  }
}
```

configure applicationContext.xml file to weave middleware services and business component into a single proxy object.

applicationContext.xml

```
<beans>
 <bean id="a" class="A"/>
 <bean id="mba" class="MBA"/>
 <bean id="ara" class="ARA"/>
 <bean id="aa" class="AA"/>
 <bean id="ta" class="TA"/>
 <bean id="aProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="a"/>
  <property name="interceptorNames">
   <list>
    <value>mba</value>
    <value>ara</value>
    <value>aa</value>
    <value>ta</value>
   </list>
  </property>
 </bean>
</beans>
```

+ Create one more java class
Name: Client1

```
public class Client1 {
  public static void main(String rags[]) throws Exception  {
    BeanFactory f=new FileSystemXmlApplicationContext("applicationContext.xml");

    A aProxy=(A)f.getBean("aProxy");
    aProxy.a();
    aProxy.b();
    aProxy.c();
    aProxy.d();
```

```
 }
}
```

In the above example every Advice is executed for every method but we want to restrict each Advice to execute only for one-one method such as MBA for a() method, ARA for b() method, AA for c() method, TA for d() method. To restrict advices to execute for specific method. For each method we need to write one Pointcut class. Pointcut class contains one method called matches into which two arguments are passed. In that first one is Method object and second one is Class object. Hence for each method we need to write one sub class of Pointcut class implement matches method, in the method we need to check whether class name is desired class or not and method name is desired method or not. If both class name and method name are matched then we need to return true otherwise false.

Add such pointcut class and advice class objects together to ProxyFactoryBean, the PFB generates one proxy to our target object. On that if we invoke methods, each advice will be restricted to one-one method.

Add following classes:

// ASMMP.java

```
import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcut;
public class AMethodSMMP extends StaticMethodMatcherPointcut {

        public boolean matches(Method m, Class c) {
                if(m.getName().equals("a") && c.getName().equals("A"))
                        return true;
                else
                        return false;
        }
}
```

// BSMMP.java

```
import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcut;
public class BMethodSMMP extends StaticMethodMatcherPointcut {

        public boolean matches(Method m, Class c) {
                if(m.getName().equals("b") && c.getName().equals("A"))
                        return true;
                else
                        return false;
        }
}
```

```
// CSMMP.java
import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcut;
public class CMethodSMMP extends StaticMethodMatcherPointcut {

	public boolean matches(Method m, Class c) {
		if(m.getName().equals("c") && c.getName().equals("A"))
			return true;
		else
			return false;
	}
}

// DSMMP.java
import java.lang.reflect.Method;
import org.springframework.aop.support.StaticMethodMatcherPointcut;
public class DMethodSMMP extends StaticMethodMatcherPointcut {

	public boolean matches(Method m, Class c) {
		if(m.getName().equals("d") && c.getName().equals("A"))
			return true;
		else
			return false;
	}
}

// Client2.java
import org.springframework.aop.framework.ProxyFactoryBean;
import org.springframework.aop.support.DefaultPointcutAdvisor;
public class Client2 {
	public static void main(String rags[]) throws Exception {
		A a=new A();
		ProxyFactoryBean pfb=new ProxyFactoryBean();
		pfb.setTarget(a);
		pfb.addAdvisor(new DefaultPointcutAdvisor(new AMethodSMMP(),
new MBA()));
		pfb.addAdvisor(new DefaultPointcutAdvisor(new BMethodSMMP(),
new ARA()));
		pfb.addAdvisor(new DefaultPointcutAdvisor(new CMethodSMMP(),
new AA()));
		pfb.addAdvisor(new DefaultPointcutAdvisor(new DMethodSMMP(),
new TA()));
		A aProxy=(A)pfb.getObject();
		aProxy.a();
		aProxy.b();
		aProxy.c();
```

```
                aProxy.d();
        }
}
```

Instead of weaving target object, advices and pointcuts in java client program we can weave them in applicationContext.xml file.

Update applicationContext.xml file as below

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
 <!-- target object -->
 <bean id="a" class="A"/>

 <!-- advice classes -->
 <bean id="mba" class="MBA"/>
 <bean id="ara" class="ARA"/>
 <bean id="aa" class="AA"/>
 <bean id="ta" class="ta"/>

 <!-- pointcut classes -->
 <bean id="amsmmp" class="AMethodSMMP"/>
 <bean id="bmsmmp" class="BMethodSMMP"/>
 <bean id="cmsmmp" class="CMethodSMMP"/>
 <bean id="dmsmmp" class="DMethodSMMP"/>

 <!-- advisor classes -->
 <bean id="amadvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <constructor-arg ref="amsmmp"/>
        <constructor-arg ref="mba"/>
 </bean>
 <bean id="bmadvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <constructor-arg ref="bmsmmp"/>
        <constructor-arg ref="ara"/>
 </bean>
 <bean id="cmadvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <constructor-arg ref="cmsmmp"/>
        <constructor-arg ref="aa"/>
 </bean>
 <bean id="dmadvisor"
class="org.springframework.aop.support.DefaultPointcutAdvisor">
        <constructor-arg ref="dmsmmp"/>
```

```
            <constructor-arg ref="ta"/>
 </bean>

 <!-- Spring AOP container  -->
  <bean id="aProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="a"/>
  <property name="interceptorNames">
   <list>
    <value>amadvisor</value>
    <value>bmadvisor</value>
    <value>cmadvisor</value>
    <value>dmadvisor</value>
   </list>
  </property>
 </bean>

</beans>
```

```
// Client3.java
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Client3 {

          public static void main(String rags[]) throws Exception
          {
             BeanFactory f=new
FileSystemXmlApplicationContext("applicationContext.xml");

             A aProxy=(A)f.getBean("aProxy");
             // System.out.println(aProxy.getClass().getName());
             aProxy.a();
             aProxy.b();
             aProxy.c();
             aProxy.d();
          }
}
```

Instead of creating Advisor class and individual Point classes we can write pointcuts using AOP regular expression pointcuts.

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:aop="http://www.springframework.org/schema/aop"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- target object -->
 <bean id="a" class="A"/>

 <!-- advice classes -->
 <bean id="mba" class="MBA"/>
 <bean id="ara" class="ARA"/>
 <bean id="aa" class="AA"/>
 <bean id="ta" class="ta"/>

<aop:config>

  <aop:pointcut expression="execution(* A.a())" id="aa"/>
  <aop:pointcut expression="execution(* A.b())" id="ab"/>
  <aop:pointcut expression="execution(* A.c())" id="ac"/>
  <aop:pointcut expression="execution(* A.d())" id="ad"/>

  <aop:advisor advice-ref="mba" pointcut-ref="aa"/>
  <aop:advisor advice-ref="ara" pointcut-ref="ab"/>
  <aop:advisor advice-ref="aa" pointcut-ref="ac"/>
  <aop:advisor advice-ref="ta" pointcut-ref="ad"/>

</aop:config>

</beans>
```

Application on Declarative AOP with Annotations

AOP provides declarative enterprise/middleware services to spring business components. Middleware services such as Transaction, Logging, Security and Messaging.

AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. Using AspectJ compiler and weaver enables use of AspectJ language.

Enabling Aspect support

To use @AspectJ aspects in spring configuration we need to enable spring support for AOP based on @AspectJ aspects we need to add the following line in spring configuration file:

```
<aop:aspectj-autoproxy/>
```

Declaring an Aspect

```
package com.durga.aspects;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class SampleAspect {
}
```

Aspects (classes annotated with @Apsect) may have may have methods and fields just like any other class. They may also contain pointcut, advice and introduction (inner-type) declarations.

Configuring Aspect in spring xml file

```
<bean id="sampaspect" class="com. durga.aspects.SampleAspect">
  <!—perform constructor and setter injections as usual -->
</bean>
```

Declaring Pointcut

Spring AOP supports method execution join points for Spring beans, so you can think of pointcut as matching the execution of methods on spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters and a pointcut expression that determines exactly which method executions we are interested in. In the AspectJ annotation style of AOP a pointcut signature is provided by a regular method definition and the pointcut expression is indicated using the @Pointcut annotation (the method serving as the pointcut signature must have a void return type). The following example clears the difference between pointcut signature and pointcut expression.

@Pointcut (“execution(* save*(..)”) **// pointcut expression**
public void anySave() {}

Supported Pointcut designators

execution – for matching method execution join points, this is primary pointcut designator you will use when working with Spring AOP

within – limits matching to join points within certain types (simply the execution of a method declared within a matching type when using spring AOP)

this – limits matching to join points (the execution of methods when using spring AOP) where the bean reference (spring AOP proxy) is an instance of the given type

target – limits matching to join points (the execution of methods when using spring AOP) where the target object (application object being proxied) is an instance of the given type

args – limits matching to join points (the execution of methods when using spring AOP) where the arguments are instances of given type

@target – limits matching to join points (the execution of methods when using spring AOP) where the class of the executing object has an annotation of the given type(s).

@args – limits matching to join point (the execution of methods when using spring AOP) where the runtime type of actual arguments passed have annotations of the given types.

@within – limits matching join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using spring AOP)

@annotation – limit matching to join points where the student of the join point (method being executed in spring AOP) has the given annotation

Combining pointcut expressions
Pointcut expression can be combined using &&, ||, !. It is also possible to refer to pointcut expressions by name.

```
@Ponitcut("execution (public * * (..))")
public void anyPublicOperation() {}

@Ponitcut("within (com. durga.someapp.trading..*)")
public void inTrading() {}

@Ponitcut("anyPublicOperation() && inTrading()")
public void inTrading() {}
```

Example on Spring ApsectJ AOP
// FirstDAO.java
```
package com.durga.crm.dao;
public class FirstDAO {
	public void firstMethod() {
		System.out.println("firstMethod() called");
	}
}
```

// SomeBusinessService.java
```
package com. durga.crm.service;
public class SomeBusinessService {
	public void someMethod() {
		System.out.println("someMethod() executed");
	}
}
```

```
// BeforeAspect.java
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class BeforeAspect {
	@Before(value="com.
durga.crm.service.SomeBusinessService.someMethod()")
	public void doBefore1(){
		System.out.println("doBefore1() called");
	}
	@Before(value="execution(* com.durga.crm.dao.*.*(..))")
	public void doBefore2(){
		System.out.println("doBefore2() called");
	}
}
// AfterReturningAspect.java
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class AfterReturningAspect {
	@AfterReturning(pointcut="com.durga.crm.service.SomeBusinessService.so
meMethod()", returning="retVal")
	public void doBefore1(Object retVal){
		System.out.println("doBefore1() called "+retVal);
	}
	@AfterReturning(pointcut="execution(*		com.durga.crm.dao.*.*(..))",
returning="retVal")
	public void doBefore2(Object retVal){
		System.out.println("doBefore2() called "+retVal);
	}
}
// AfterAspect.java
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class AfterAspect {
	@After(value="com.durga.crm.service.SomeBusinessService.someMethod()"
)
	public void doBefore1(Object retVal){
		System.out.println("doBefore1() called "+retVal);
	}
}
// AroundAspect.java
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class AroundAspect {
	@Around(value="com.durga.crm.service.SomeBusinessService.someMethod(
)")
	public Object doSomeAround(ProceedingJoinPoint pjp) throws Throwable {
		System.out.println("doSomeAround() before");
		Object o=pjp.proceed();
		System.out.println("doSomeAround() before");
		return o;
	}
}
```

// AfterThrowingAspect.java

```
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.dao.DataAccessException;
@Aspect
public class AfterThrowingAspect {
@AfterThrowing(pointcut="com.durga.crm.service.SomeBusinessService.someMeth
od()", throwing="ex")
	public void doBefore1(DataAccessException ex){
		System.out.println("doBefore1() called "+ex);
	}
}
```

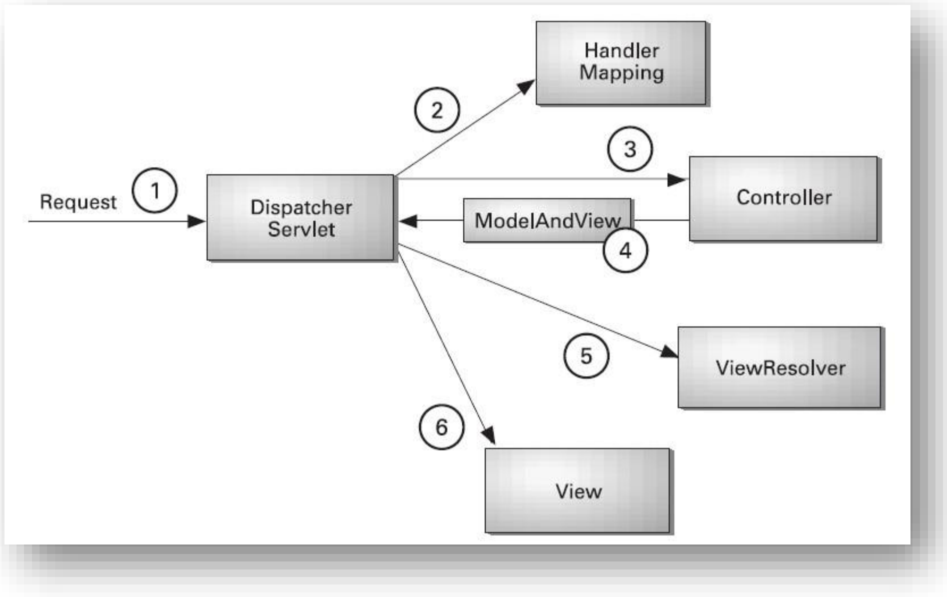
applicationContext.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
	xmlns:p="http://www.springframework.org/schema/p"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
	<aop:aspectj-autoproxy proxy-target-class="true"/>

	<bean id="sbs" class="com.durga.crm.service.SomeBusinessService"/>
	<bean id="fd" class="com.durga.crm.dao.FirstDAO"/>

	<bean id="aa" class="AfterAspect"/>
	<bean id="ara" class="AfterReturningAspect"/>
	<bean id="ata" class="AfterThrowingAspect"/>
	<bean id="arounda" class="AroundAspect"/>
	<bean id="ba" class="BeforeAspect"/>
</beans>
```

// Client.java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.durga.crm.dao.FirstDAO;
import com.durga.crm.service.SomeBusinessService;
public class Client {
	public static void main(String[] args) {
		ApplicationContext						ac=new
ClassPathXmlApplicationContext("applicationContext.xml");
		SomeBusinessService sbs=(SomeBusinessService)ac.getBean("sbs");
		sbs.someMethod();
		FirstDAO fd=(FirstDAO)ac.getBean("fd");
		fd.firstMethod();
	}
}
```

<u>**MVC-FRAMEWORK**</u>



Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet is completely integrated with Spring IoC container and allows us to use every other feature of Spring.

Following is the Request process lifecycle of Spring 3.0 MVC:

i. The client sends a request to web container in the form of http request.
ii. This incoming request is intercepted by Front controller (DispatcherServlet) and it will then tries to find out appropriate Handler Mappings.
iii. With the help of Handler Mappings, the DispatcherServlet will dispatch the request to appropriate Controller.
iv. The Controller tries to process the request and returns the Model and View object in form of ModelAndView instance to the Front Controller.
v. The Front Controller then tries to resolve the View (which can be JSP, Freemarker, Velocity etc) by consulting the View Resolver object.
vi. The selected view is then rendered back to client.

<u>Controllers:</u>
You can implement controllers form

i. Controller(I)
ii. Abstract controller
iii. SimpleFormController
iv. MultiActionController
v. By Using @Controller annotation

Handlers:

i. BeanNameUrlHandlerMapping:
   Maps the requested URL to the name of the controller.
ii. ControllerClassNameHandlerMapping:
   Uses convention to map the requested URL to Controller.
iii. SimpleUrlHandlerMapping:
   Allow developer to specify the mapping of URL patterns and handler mappings explicitly.

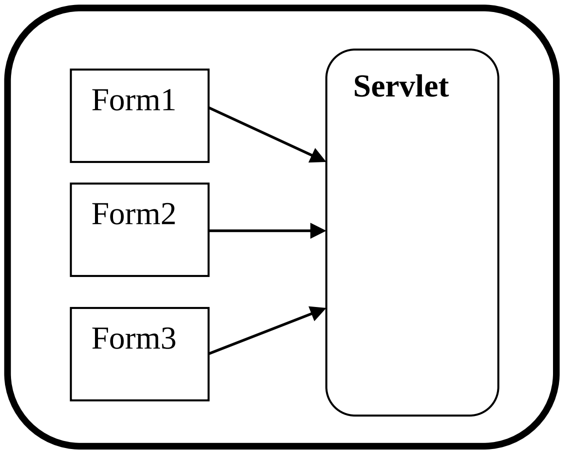
View Resolvers:

i. InternalResourceViewResolver example
ii. XmlViewResolver example
iii. ResourceBundleViewResolver
iv. Configure multiple view resolvers priority

Hanlder:
BeanNameUrlHanlderMapping
Actually, declare **BeanNameUrlHandlerMapping** is optional, by default, if Spring can't found handler mapping, the DispatcherServlet will creates a **BeanNameUrlHandlerMapping** automatically.

Spring MVC is a type of centralized mvc architecture and it is using jsp-model-I

Jsp-Model-I



To map multiple forms to one single controller under web.xml file we should use url pattern with help of expression

Ex:

```
<web-app>
<servlet>
<servlet-name> servletname  <servlet-name>
<servlet-class> servletclass</servlet-class>
</servlet>
```

We have to use dispatcher servlet as a front controller servlet and in web.xml file we have to configure it

web.xml

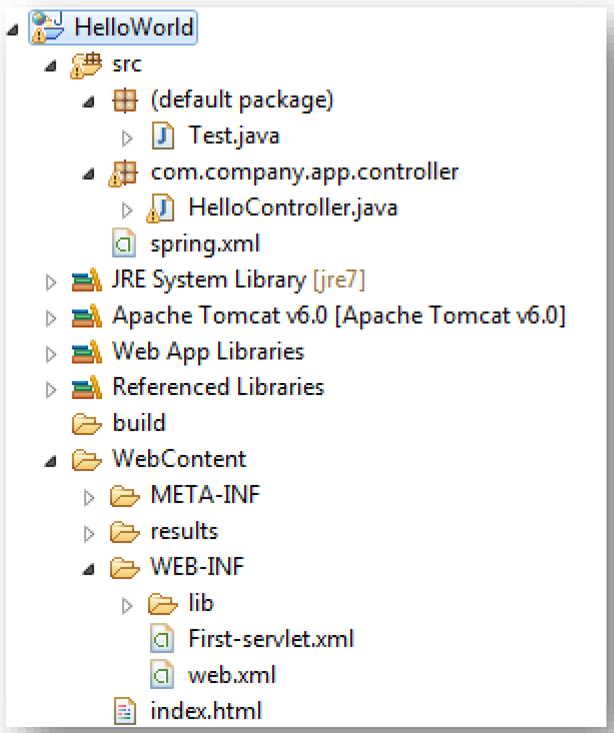
```
<web-app>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value> classpath:spring.xml</param-value>
</context-param>
<servlet>
<servlet-name>First</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<!--
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value> classpath:spring.xml /WEB-INF/First-config.xml</param-value>
</init-param> -->
</servlet>
<servlet-mapping>
```

```
<servlet-name>First</servlet-name>
<url-pattern>*.htm</url-pattern>
</servlet-mapping>
</web-app>
```

Configuration file you can configure by using

i. Servletname-servlet.xml or
ii. Using init parameter : contextConfigLocation
iii. Using Context parameter and Listner org.springframework.web.context.ContextLoaderListener

Ex:
Create dynamic web project
Add spring3.0 lib to classpath and run time:



```
HelloWorld
  src
    (default package)
      Test.java
    com.company.app.controller
      HelloController.java
    spring.xml
  JRE System Library [jre7]
  Apache Tomcat v6.0 [Apache Tomcat v6.0]
  Web App Libraries
  Referenced Libraries
  build
  WebContent
    META-INF
    results
    WEB-INF
      lib
      First-servlet.xml
      web.xml
    index.html
```

Index.html:
<form action=*"./hello.htm"*>

```
NAME:<input type="text" name="name"/><br/>
<input type="submit" value="sayHello"/>
</form>
```

results/success.jsp

```
${msg}
```

HelloController.java

```
public class HelloController implements Controller {
public HelloController() {
System.out.println("CONTROLLER");
}
	@Override
	public ModelAndView handleRequest(HttpServletRequest req,
			HttpServletResponse res) throws Exception {

		String name=req.getParameter("name");
		//to store out put data
		Map m=new HashMap();
		m.put("msg", "Hello.. mr.."+name);
		ModelAndView mav=new ModelAndView("success", m);
		return mav;
	}

}
```

```
public class Test {
public Test() {
System.out.println("TEST CLASS");
}
}
```

Spring.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 class="Test"/>
</beans>
```

First-sevlet.xml(web configuration)

```
<beans>
<!-- view to controller -->
<bean name="/hello.htm" class="com.company.app.controller.HelloController"/>
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/results/"/>
<property name="suffix" value=".jsp"/>
</bean>
</beans>
```

Web.xml:

```
<web-app>
  <listener>
    <listenerclass>
	org.springframework.web.context.ContextLoaderListener
</listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value> classpath:spring.xml</param-value>
  </context-param>
  <servlet>
    <servlet-name>First</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>First</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>
</web-app>
```

Simple Form Controller

```
public class RegController extends SimpleFormController {

private StudentDao sdao;


public void setSdao(StudentDao sdao) {
this.sdao = sdao;
}

public RegController() {
setCommandClass(RegForm.class);
setCommandName("regForm");
}

@Override
protected ModelAndView onSubmit(HttpServletRequest request,
HttpServletResponse response, Object command, BindException errors)
throws Exception {
RegForm rf=(RegForm)command;
sdao.save(rf);
return new ModelAndView("success");
}
}
```

RegForm.java (pojo classs):

```
public class RegForm {
        private String name;
        private String email;
        private String address;
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }



        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }

        }
```

RegFormValidation.java

```
public class RegFormValidator implements Validator {

@Override
public boolean supports(Class<?> clazz) {
return RegForm.class==clazz;
}

@Override
public void validate(Object command, Errors error) {
RegForm rf=(RegForm)command;
ValidationUtils.rejectIfEmptyOrWhitespace(error, "name",
"name", "Field name is required.");
ValidationUtils.rejectIfEmptyOrWhitespace(error, "email",
"email", "Field name is required.");
ValidationUtils.rejectIfEmptyOrWhitespace(error, "address",
"address", "Field name is required.");

}
}
```