Spring FrameWork

Agenda :

- Spring Framework Introduction
- Features of Spring framework
- Spring Architecture
- Modules of Spring framework
    1. Core Module
    2. DAO Module
    3. ORM Module
    4. JEE/Application Context Module
    5. AOP Module
    6. Web MVC Module
- Terminology

Framework:

- It is a special software that is capable of developing applications based on ceratian architecture with the ability to generate common logics of the application.
- It is given based on core technologies.
- It provides an abstraction layer on core technologies.

Struts: It is a framework to develop MVC-II architecture based web applications only.

Hibernate: Its a ORM Framework which is used only for database operations.

Spring:

Its a framework software which can be used to develop any kind of Java/J2EE applications.
Spring framework provides abstraction layer on:

- java & J2EE core technologies
- ORM Tools like:
    1. Hibernate
    2. JDO
    3. iBastis etc..,
    4. AOP Framework etc..,

Introduction:

1. Spring framework is developed to simplify the development of enterprise applications in Java technologies.
2. It is an open source framework begin developed by Interface21.
3. The Spring provides light weight IoC Container and AOP framework.
4. It provides support for JPA, Hibernate, Web services, Schedulers, Ajax, Struts, JSF and many other frameworks.
5. The Spring MVC components can be used to develop MVC based web applications.
6. The Spring 3.0 framework has been released with major enhancements and support for the Java 5 [JDK1.5].
7. Spring can be used to configure declarative transaction management, remote access to your logic using RMI or web services, mailing facilities and various options in persisting your data to a database.
8. Spring framework can be used in modular fashion, it allows to use in parts and leave the other components which is not required by the application.

- Spring is an open source framework, which is very flexible while developing any kind of application. i.e. stand alone application/web-application/enterprise applications.
- Struts is used only for web applications, where as Spring can be used for developing stand-alone application, applet based application, web application and enterprise application.
- Spring framework is created to address the complexity of enterprise application development.
- One of the chief advantages of the Spring framework is its layered architecture
- Spring is a light-weight framework.
- Spring is having total 6 modules.
- We can use the modules independently or combine depending on the type of application and requirement.
- Main core of Spring is IOC module.

The Advantages of spring framework

The advantages of spring are as follows:

- Spring has layered architecture. Use what you need and leave you don't need now.
- Spring Enables POJO Programming. There is no behind the scene magic here. POJO programming enables continuous integration and testability.
- Dependency Injection and Inversion of Control Simplifies JDBC
- Open source and no vendor lock-in.

The Features of Spring framework

The features of spring framework are as follows

- **Lightweight:** Spring is lightweight when it comes to size and transparency. The basic version of spring framework is around 1MB. And the processing overhead is also very negligible.
- **Inversion of control (IOC):** Loose coupling is achieved in spring using the technique Inversion of Control. The objects give their dependencies instead of creating or looking for dependent objects.
- **Aspect oriented (AOP):** Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services.
- **Container:** Spring contains and manages the life cycle and configuration of application objects.
- **MVC Framework:** Spring comes with MVC web application framework, built on core Spring functionality. This framework is highly configurable via strategy interfaces, and accommodates multiple view technologies like JSP, Velocity, Tiles, iText, and POI. But other frameworks can be easily used instead of Spring MVC Framework.
- **Transaction Management:** Spring framework provides a generic abstraction layer for transaction management. This allowing the developer to add the pluggable transaction managers, and making it easy to demarcate transactions without dealing with low-level issues. Spring's transaction support is not tied to J2EE environments and it can be also used in container less environments.
- **JDBC Exception Handling:** The JDBC abstraction layer of the Spring offers a meaningful exception hierarchy, which simplifies the error handling strategy. Integration with Hibernate, JDO, and iBATIS: Spring provides best Integration services with Hibernate, JDO and iBATIS

Spring framework came up with every module for every tier. No other framework contains these many modules.

- **Tier:** It is a clear separation of two or more Systems arranged by one after another.

Possible Tiers while developing an application.

UI Tier	Presentation Tier	Business Tier	Integration Tier	DataBase Tier
Web browser	HTML/JSP	J2EE	JDBC/DAO	EIS

- **UI Tier:** User Interface Tier. Like a web browser
- **Presentation Tier:** CSS, HTML, JavaScript, Images, Servlet, Jsp, Web MVC Module of spring,Struts.
- **Business Tier:** we can use plain java class, java beans, ejb, messaging services. For this Spring is providing J2EE Module.
- **Integration Tier:** JDBC or any ORM tools like Hibernate, IBatis. For this Spring is providing JDBC& DAO Module.
- **DataBase Tier:** under this Enterprise Information Services will come.

If you are using any ORM related tools like Hibernate, for this Spring provides one module ORM.

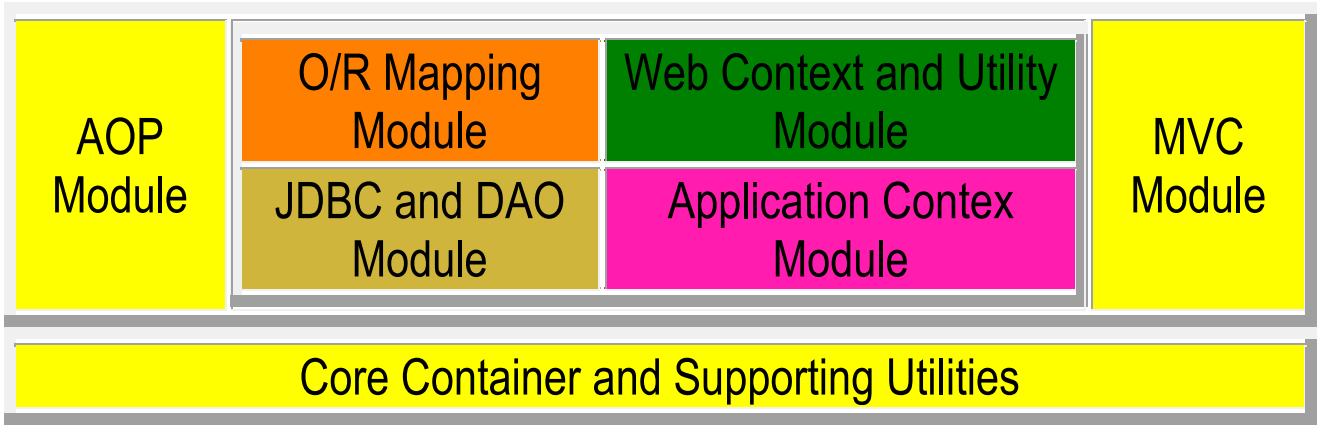
Spring Architecture:

```
Spring is well-organized architecture consisting of
seven modules.
Modules in the Spring framework are:
```

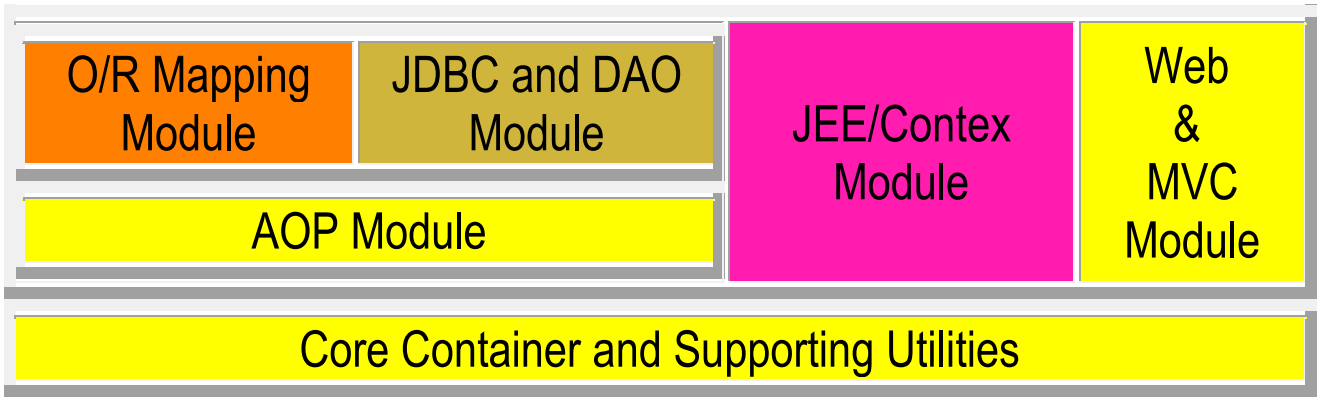
Spring1.x: It has 7 modules. They are:

1. Core

2. DAO
3. ORM
4. JEE/Context
5. Web
6. MVC
7. AOP



Spring2.x: It has 6 modules only. Because Web & Web MVC modules were clubbed together & given as Web MVC module.



Core Module:

1. It is base module for all modules.
2. Provides BeanFactory container which makes Spring as a container

3. provides the fundamental functionality of the Spring framework.
4. The Core package is the most import component of the Spring Framework.
5. This component provides the Dependency Injection [DI] features.

This module contains IOC container.

```
IOC container is a piece of s/w, Which creates the
objects and establish the dependencies.
```

Core module is sub-divided into 4 modules : They are

1. bean
2. core
3. context
4. Expression Language

- The beans & core modules contains the code of IOC container.
- Context module is dependent on beans & core module. When we use context module, it takes care of developing I18N applications, and validations.
- By using context module also we can create IOC container.
- If we created IOC container based on beans and core modules, If we want to take care of I18N applications we have to provide our own code.
- If we created IOC container based on context module, It will take care of I18N applications on it's own.

Expression Language : This module is an extension to the JSPEL expression.

We can use Expression Language as part of configuration files as well as java programs and etc.,

DAO Module:

DAO stands for Data Access Object
This module contains set of classes & interfaces, which takes care of JDBC code by using DAO module, we no need to provide a tedious JDBC code.

1. It provides abstraction layer on JDBC & allows us to do database operations.
2. Using it we can focus only on database operations instead of doing repeated work by writing common code again and again.
3. Here we can make use of JdbcTemplate class where we can call methods to do required database operation.
4. We can connect to database either by manual code same like jdbc or we can do easily by using JdbcTemplate class.
5. Using JdbcTemplate class we can retrieve data from database in the form of serializable objects.

ORM Module:

1. It provides an abstraction layer on ORM Framework software's like Hibernate, iBatis, JDO etc..,
2. Spring doesn't attempt to implement its own ORM solution
3. Spring's transaction management supports each of these ORM frameworks as well as JDBC.
4. As part of ORM module, Spring guys given the classes like HibernateTemplate, JPATemplate etc.,

JEE/Application Context Module:

1. It provides abstraction layer on Java/J2EE technologies like Java Mail, JMS, JMX, EJB etc..,
2. The context module is what makes spring as a framework.
3. This module extends the concept of BeanFactory
4. Adds support for internationalization (I18N) messages, application life cycle events, and validation.
5. Supplies many enterprise services such as e-mail, JNDI access, EJB integration, remoting, and scheduling.

AOP [Aspect Oriented Programming] Module:

1. One of the key components of Spring is the AOP framework.
2. It provides a methodology to configure middle-ware services or to perform any authentication, validations or to perform any pre processing logic or post processing logic on our business logic methods.
3. Without disturbing any kind of spring resources we can integrate our spring application with any kind of middle-ware service.
4. Spring provides a separate layer for middle-Ware configuration.
5. This module serves as the basis for developing your own aspects for your Spring-enabled application.

- This module takes care of staring the AOP & ending the AOP.
- This AoP programming module is integrated with AspectJ.
- This AOP model, address of all the problems of object orientation(OOP)
- A programming Language AspectJ is built based on AOP programming model.

- The spring guys has integrated AspectJ programming language as part of spring.
- Spring guys has developed their own aspects & placed inside the AOP module.

Ex: TransactionManager is predefined aspect given by the spring guys.

Web MVC Module:

It is given for two operations:

1. To integrate spring application with any other web based framework like Struts, JSF etc..,
2. It provides its own web framework named as Spring MVC to develop MVC-II architecture based web applications. It provides the MVC implementations for the web applications.

- In the web based applications we generate reports, Spring guys has integrated the report generation tools like Jasper, IReports generation of excel sheets as part of spring.
- The web module is divided into a 4 sub-modules : They are-
    1. Web
    2. Spring Servlet
    3. Spring Struts
    4. Spring Portlets
- From Spring 3.0 onwards, they are removed a support of integrating Spring with Struts.
  According to Spring documentation, using struts 2 is same as Spring MVC.

Note: We can develop a spring application either by using all modules or only by using few modules of spring. We can also

integrate our spring module with any kind of a java application. So spring is loosely coupled.

Terminology :

Spring is a open source, light weight, loosely coupled, aspect oriented & dependency injection based framework software which can be used to develop any kind of java/j2ee application.

Open Source: Spring software can be down loaded and used at free of cost. Spring developed source code can referred by any one who is interested.

Container is a special software which can manage complete life cycle of a given resource.

Loosely Coupled: Need not to inherit any class or interface as we do for our servlet programming. Instead of having "is-a" relationship here we can have "has-a" relationship.

Lightweight:

1. Spring is lightweight in terms of both size and overhead.
2. The entire Spring framework can be distributed in a single JAR file that weighs in at just over 1 MB.
3. Spring is non-intrusive: objects in a Spring-enabled application typically have no dependencies on Spring specific classes.
4. Mainly spring is used for business logic. The same business logic if we try to develop using EJB, then its compulsory to have support of any application server like weblogic. But the same logic can be executed in Spring without help of any web server or application server. Because Spring provides its own containers in the form of pre defined classes.

Note: Servlet Container & EJB Container are heavy weight because unless we start web server or application server we cannot activate either servlet container or EJB Container.

Inversion Of Control (IOC) and Spring Core Container

Agenda :

1. Inversion of Control (IoC)
2. Dependency Injection (DI)
3. Spring Containers
    - BeanFactroy container
    - ApplicationContext container
    - ApplicationContext container advantages
4. Initializing Spring Core Container
5. Types of Dependency Injection
    - Constructor Injection
    - Setter Method Injection
6. abstract attribute
7. spring container based on FileSystemResource
8. purpose of getBean() method
9. lazy-init attribute
10. What is Wiring
    - auto-wire attribute
    - auto-wire="byName"
    - auto-wire="byType"
11. Difference between id and name attributes
12. ref tag
    - bean attribute

Inversion of Control (IoC)

The advantage of Spring instead of developer create the object and establish the dependencies, the spring container create the object and establish the dependencies. This process is called as Inversion of Control(IOC).

- IOC describes that a Dependency Injection needs to be done by an external entity(i.e., container) instead of creating the dependencies by the component itself.
- Dependency Injection is a process of Injecting(Pushing) the dependencies into an object.

- In general object is controlled by our java programmer, otherwise we can store the state of object using Serialization or Externalization.
- If controlling of object is done by configuration files i.e. creating of object, storing state of object (i.e. somewhat inverse of way), this is called Inversion of Control.(IOC).

- This IOC concept is base for every module of spring.
- IOC is core in case of spring.
- IOC is also called as Dependence Injection.
- Here we are achieving dependency injection depending on object whatever the data requires, that will be injected through configuration file.
- IOC container provides objects bases on the configuration in xml files. We are not creating object by using new operator for an entity. We will get object from IOC container. It goes to configuration file, based on that it generates object.
- IOC container generates objects and it is giving those objects to our programmer.
- IOC is maintains Singleton Object

Dependency Injection (DI)

Dependency Injection is a process which takes place when IOC getting perform because this reason we call Dependency Injection & IOC are same.

- In developing huge systems using the object oriented programming methodology, we generally divide the system into objects where each of the objects represents some functionality.
- In this case, the objects in the system use some other objects to complete the given response.
- The objects with which our object collaborates to provide the services are known as its dependencies.

The traditional ways of obtaining the dependencies are by creating the dependencies or by pulling the dependencies using some factory classes and methods or form the naming registry. But these approaches result in some problems which are

- The complexity of the application increases.
- The development time-dependency increases.
- The difficulty for unit testing increases.

To solve these problem we have to use Push Model.
i.e., inject the dependent objects into our object instead of creating or pulling the dependent objects.

The process of injecting(pushing) the dependencies into an object is known as Dependency Injection (DI)

Spring Containers :

Core container or IOC container or BeanFactroy container

1. Its an implementation class of an interface BeanFactory
2. It supports lazy loading.
    - It means container will not create objects of configured pojo classes immediately.
    - It waits till it receives the first request.

- Once the object is created the same object will be maintained till end of the application.

3. By default it considers all the configured classes as singleton classes.
4. So that the same object will be given to all requests which belongs to same class.
5. We cannot change singleton scope for core container.
6. provides the fundamental functionality of the Spring framework.
7. BeanFactory, the heart of any Spring-based application.

Advanced container or ApplicationContext container

1. It is advanced container.
2. By default it supports early loading.
   - It means without receiving any request from client program immediately it will create object for every singleton scoped class.
   - When ever the container gets activated immediately it creates objects of all singleton classes.
3. Has many advantages over core container.

ApplicationContext container is enhancement of bean factory container with some advantages.

Its features are as follows:

1. Pre initialization of beans by default. [Early loading]. It means in our spring configuration file we may have any no. of bean tags. Objects will be created sequentially for each and every bean class as specified in the sequence of <bean> tags in xml file.
2. Ability to read values of bean properties from properties file.

3. supports Internationalization (I18n)
4. Gives the ability to work with events & listeners.
5. **org.springframework.context.ApplicationContext** interface is the sub interface of**org.springframework.beans.factory.BeanFactory** interface.
6. Activating application context interface is nothing but creating object of a class that implements **org.springframework.context.ApplicationContext** interface.
7. There are three regularly used implementation classes of application context interface. By creating object for any of these three classes we can activate ApplicationContext container.
8. This container is available in JEE module.
9. **org.springframework.context.support.FileSystemXmlApplicationContext:** It activates application context container by locating given spring configuration file in the specified path.
   Ex:

```
10. FileSystemXmlApplicationContext ctx = new
11.
   FileSystemXmlApplicationContext("c:\f1\spriong.
   cfg.xml")
```

12. **org.springframework.context.support.ClassPathXmlApplicationContext:** It activates application context container by locating spring configuration file in the same working directory or from jar files added in the classpath.
    Ex:

```
13. ClassPathXmlApplicationContext ctx = new
14.
   ClassPathXmlApplicationContext("spring.cfg.xml"
   );
```

15. **org.springframework.context.support.XmlWebApplicationContext:** this class activates application context container by locating spring configuration file in deployment directory structure of web application by default in WEB-INF folder.
    Ex:

```
16. XmlWebApplicationContext ctx =
17.     new
    XmlWebApplicationContext("spring.cfg.xml");
```

18. In real time applications we can find regularly working with Application Context container rather than working with Bean Factory container.
19. Application context container can perform all modes of dependency injection like bean factory container.
20. Application context container performs pre instantiation on all singleton scoped on spring bean classes of spring configuration file at the moment of application context container gets activated.
21. Pre-instantiation means creating spring bean class objects immediately after the application context container activation.
22. BeanFactory cannot perform this pre instantiation on spring bean classes.
23. when we call factory.getBean(_) then bean factory container immediately creates the object and uses the same for further requests on same id-value.
24. When we call ctx.getBean(_) then application context container gives access to bean id related spring bean class object which is created at pre-instantiation process.
25. Application context container can perform pre-instantiation only on singleton scoped spring bean classes that are configured in spring configuration file.

26. If we give scope value as prototype then object will not be created for bean at the time of pre-instantiation.
27. If the object is not created at pre-instantiation then it will be created after receiving the call along with its dependent objects.
28. If singleton scope bean class property has prototype scoped bean class object as dependent value then the application context container also creates prototype scoped spring bean class object during pre instantiation process along with singleton scope bean class object.
29. It happens to satisfy dependency injection needs done on singleton scope bean class properties.

Initializing Spring Core Container & Accessing Spring Beans :

```
//Using BeanFactory to instantiate spring core
container

BeanFactory beans= new XmlBeanFactory(
          new FileSystemResource("mybeans.xml"));
```

We are using XmlBeanFactory implementation for instantiating the spring container with 'mybeans.xml' file as configuration file.

```
 //Using ApplicationContext to instantiate spring
core container
ApplicationContext context=
      new
ClassPathXmlApplicationContext("mybeans.xml");
```

We are using ClassPathXmlApplicationContext implementation for instantiating the spring container with 'mybeans.xml' file as configuration file.

Example 1:

HelloService.java

```
 package com.core;

public class HelloService {
 String message;
 public HelloService(){
  message="msg from default Constructor";
 }

 public HelloService(String msg){
  message=msg;
 }

 public String getMessage(){
  return message;
 }
}//class
```

mybeans.xml

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

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

http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd">

<bean id="helloService1"
class="com.core.HelloService"/>

<bean id="helloService2"
class="com.core.HelloService">
```

```
 <constructor-arg>
  <value>hello from configuration</value>
 </constructor-arg>
</bean>

</beans>
```

HelloServiceTestCase.java

```
package com.core;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

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

 BeanFactory beans=
       new XmlBeanFactory(new
FileSystemResource("mybeans.xml"));

 HelloService
hello1=(HelloService)beans.getBean("helloService1")
;
 System.out.println(hello1.getMessage());

 HelloService
hello2=(HelloService)beans.getBean("helloService2")
;
 System.out.println(hello2.getMessage());
}//main

}//class
```

output :

```
msg from default Constructor
hello from configuration
```

Types of Dependency Injection :

We have 2 types of dependency injection options

1. Constructor Injection
2. Setter Method Injection

Constructor Injection

Injecting the dependencies of an object through its constructor arguments

If the constructor is establishing the dependencies between objects, we call it as Constructor Injection.

constructor Injection example :

Employee.java

```
package com.core;

public class Employee {

private String no;
private String name;
private String address;

Employee(String no,String name,String address){
 this.no=no;
 this.name=name;
 this.address=address;
```

```
 System.out.println("We are in Parameter
Constructor");
}

}
```

applicationContext.xml

```
<bean id="emp" class="com.core.Employee">
 <constructor-arg type="java.lang.String"
index="0">
   <value type="java.lang.String">25</value>
 </constructor-arg>
  <constructor-arg type="java.lang.String"
index="1">
   <value type="java.lang.String">Ashok</value>
 </constructor-arg>
  <constructor-arg type="java.lang.String"
index="2">
   <value type="java.lang.String">HYD</value>
 </constructor-arg>
</bean>
```

- The spring container find the constructors based on number of parameters.
- When we configure constructor injection, we know need to supply index attribute, if we doesn't supply index attribute by default the tag is considered as parameter zero, 2nd tag is considered as parameter index '1' etc.,
- If we don't specify type default spring uses the data types of spring bean.
- Even though index and type attributes are optional its always recommended to use index and type attributes.
- These resolve the conflict of overloaded constructors instead of using index attribute we can use name attribute to specify to which variable we are supplying the value.

```
<bean id="emp" class="com.core.Employee">
 <constructor-arg name="no" type="int" value="2"/>
 <constructor-arg name="name" type="String"
value="Ashok"/>
 <constructor-arg name="address" type="String"
value="HYD"/>
</bean>
```

- When ever mandatory properties are available , when the object is created we have to initialize them, in this scenario we use constructor injection.
- Based on the user choice we want to initialize the properties in this scenario we use setter method injection.

Setter Method Injection

If the setter method is establishing the dependencies between objects, we call it as Setter Method Injection.

Requirement :
create a spring bean address with 3 properties and establish the dependencies and ask the spring container to create the object

spring bean : In spring, java beans are considered as spring-beans.

Address.java

```
package com.core;

public class Address {

private String street;
private String city;
private String state;
```

```
public String getStreet() {
	return street;
}
public void setStreet(String street) {
	this.street = street;
}
public String getCity() {
	return city;
}
public void setCity(String city) {
	this.city = city;
}
public String getState() {
	return state;
}
public void setState(String state) {
	this.state = state;
}

}//class
```

- If we want spring container to create the object and establish the dependencies, we must configure spring configuration file.
- Instead of manual configuration, we take a help of view(i.e., spring explorer)

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"
```

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

http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd">

<bean id="addr" class="com.core.Address"
      abstract="false" lazy-init="default"
autowire="default">

<property name="street">
 <value type="java.lang.String">Apet</value>
</property>
<property name="city">
 <value type="java.lang.String">HYD</value>
</property>
<property name="state">
 <value type="java.lang.String">IND</value>
</property>

</bean>

</beans>
```

- When we configure spring bean into spring bean configuration file we must specified spring type.
- The spring type is based on the property data type.
  spring types : **ref, idref, value, null, set, list, map, props**
- When we configure spring bean into spring bean configuration file, **to reduce the spring bean configuration file** Spring 3.0 has provided an option P-namespace.

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

```
<bean id="addr" class="com.core.Address"
      p:street="APet" p:city="HYD"
p:state="IND" />
```

- Spring container consider attributes as properties.
  If We doesn't to specify type attribute by default spring uses the property type of spring bean.
  To resolve the configuration is to spring bean configuration file instead of value tags, we can use value attribute.

```
<bean id="addr" class="com.core.Address" >

  <property name="city" value="Bangalore" />
  <property name="state" value="Karnataka" />

</bean>
```

- **ClassPathResource** can be used to read the context of resources which is available in side ClassPath of our ClassPath.
- **FileSystemResource** can be used to read the content of resource which is available in our computer.

MyApp.java

```
package com.core;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class MyApp {
```

```
public static void main(String[] args) {
 Resource resource=
    new
ClassPathResource("com/core/applicationContext.xml"
);
 BeanFactory container=new
XmlBeanFactory(resource);
 System.out.println("container : "+container);

 Object o=container.getBean("addr");
 Address a=(Address)o;
 System.out.println("Street : "+a.getStreet());
 System.out.println("City : "+a.getCity());
 System.out.println("State : "+a.getState());
}//main

}
```

Output :

```
container :
org.springframework.beans.factory.xml.XmlBeanFactor
y@64883c:
             defining beans [addr]; root of factory
hierarchy

Street : Apet
City : HYD
State : IND
```

- When we create a spring container object based on XmlBeanFactory immediately it is not creating spring bean object and establish the dependencies.
- When ever we call a method **getBean()**, then only we create the object and perform dependency.

When we try to remove configuration file from classpath and try to run the program, we get an exception saying **java.io.FileNotFoundException**

When the spring container create the object, it has initialized with default values.

abstract attribute :

- We can use an abstract attribute, if we don't want to allow any body to create an object by default **abstract="false"**
- If we make as **abstract="true"**, spring container can't create the object to the spring bean **id**.
- specify **abstract="true"**, we can not saying the spring bean class also abstract.

The following is an example of creating a spring container based on FileSystemResource :

```
package com.core;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;

public class MyApp {

public static void main(String[] args) {
 Resource resource=
    new
FileSystemResource("D:/work/applicationContext.xml"
);
```

```
 BeanFactory container=new
XmlBeanFactory(resource);
 System.out.println("container : "+container);

}//main

}
```

What is the purpose of getBean() method :

- getBean() is used to ask the spring container and establish dependencies.
- When ever we call getBean() method every time it is not creating spring bean object.

In the most of the projects we may use singleton design pattern (OR) We may create multiple objects we can control this based on scope.

- When we create the spring container object based on the XmlBeanFactory until & unless we call getBean() it will not create spring bean object and establish the dependencies
- `Object o=container.getBean("addr");`
- When the above code is executed, it will take the id and check whether the id is available, it will take the class name and create the object and establish the dependencies. Now the getBean() converts the object into super-class object.
- If we call getBean() with an id, which is not available we will get an Exception **org.springframework.beans.factory.NoSuchBeanDefinitionException**.
- Instead of using getBean() we can directly get the object by using our loaded method getBean()
- `syntax :`
-

```
T  getBean(String name, class<T> required type)

Address addr=container.getBean("addr",Address.class);
```

Every time when we call a getBean(), is it creating a new spring bean and establishing dependencies ?

- **getBean()** behavior is based on the scope attribute in the spring bean configuration file.
- If no scope attribute is specified by-default it uses **scope="singleton"** it will create the object only once and it will returns the same object every time, when we call getBean()

```
Address addr1=container.getBean("addr",Address.class);
Address addr2=container.getBean("addr",Address.class);

System.out.println(addr1);
System.out.println(addr2);  //same hashcode
```

- Core module we have to bean package & context package.
- By using context package we can create IOC container.
- By using context IOC package it takes care of I18N applications.
- The context package contains an interface "ApplicationContext" (org.springframework.context)
- This interface internally inherits the properties of "BeanFactory" interface
- Any class which provide the implementation of ApplicationContext also can be called as spring container.
- ClassPathXmlApplicationContext, FileSystemXmlApplicationContext provide the

implementation of ApplicationContext.(org.springframework.context.support)

- When ever we create spring container object based on ApplicationContext and if the scope="singleton" immediately spring container create the object and establish the dependencies.
- When ever we configure spring bean into spring bean configuration file and scope="singleton", it will create the objects the order of object creation is **top to bottom**.

lazy-init attribute :

- lazy-init attribute is used to specified when the spring container has to create spring bean object.
- lazy-init attribute work only if scope="singleton".
- spring container will not consider lazy-init if the scope value is prototype/request/session.
- lazy-init="default" means lazy-init="false"
- If lazy-init="false" and scope="singleton" whenever spring container creates the object immediately.

```
<bean name="addr" class="com.core.Address"
            lazy-init="false" scope="singleton"/>
```

Requirement :
Implement dependency between an address object and employee object

Address.java

```
package com.core;
```

```
public class Address {

private String street;
private String city;
private String state;

public String getStreet() {
     return street;
}
public void setStreet(String street) {
     this.street = street;
}
public String getCity() {
     return city;
}
public void setCity(String city) {
     this.city = city;
}
public String getState() {
     return state;
}
public void setState(String state) {
     this.state = state;
}

}//class
```

Employee.java

```
package com.core;

public class Employee {

private int no;
private String name;
private Address address;

public int getNo() {
     return no;
```

```
}
public void setNo(int no) {
     this.no = no;
}
public String getName() {
     return name;
}
public void setName(String name) {
     this.name = name;
}
public Address getAddress() {
     return address;
}
public void setAddress(Address address) {
     this.address = address;
}

}
```

Employee spring bean dependent on Address spring bean.

We have to configure the above 2 spring beans into spring configuration file.

applicationContext.xml

```
<bean id="emp" class="com.core.Employee" >
 <property name="no" value="1"/>
 <property name="name" value="Ashok"/>
 <property name="address">
    <ref bean="addr"/>
 </property>
</bean>

<bean id="addr" class="com.core.Address" >
  <property name="street" value="Abc"/>
  <property name="city" value="Bangalore"/>
  <property name="state" value="Chennai"/>
</bean>
```

When we create a spring container object and call the getBean() method, it will create the Employee object as well as Address object.Because Employee is dependent on Address.

What is Wiring :

- Connecting 2 different objects is called Wiring (OR) Establishing dependencies between the objects is called as Wiring.
- By default spring uses manual wiring the meaning of it is, if we doesn't specify the dependencies spring container will not establish the dependencies.

auto-wire attribute :

By default auto-wire attribute take values

- byName
- byType

By default auto-wire="no" , the meaning of it is spring container is looking for manual wiring.

auto-wire="byName"

in this case the spring container automatically establish the wiring/dependencies, if the **property name** and **bean id** are same.

```
<bean id="emp" class="com.core.Employee"
                lazy-init="true" autowire="byName">
 <property name="no" value="1"/>
 <property name="name" value="Ashok"/>
</bean>
```

```
<bean id="address" class="com.core.Address" lazy-
init="true">
  <property name="street" value="Abc"/>
  <property name="city" value="Bangalore"/>
  <property name="state" value="Chennai"/>
</bean>
```

Note : If manual wiring is specify, auto-wire will not considered by spring container.

auto-wire="byType"

in this case the spring container check is there any beans are configured with that **data type** , if it is configured it will establish the dependency.

```
<bean id="emp" class="com.core.Employee"
               lazy-init="true" autowire="byType">
 <property name="no" value="1"/>
 <property name="name" value="Ashok"/>
</bean>

<bean id="addr" class="com.core.Address" lazy-
init="true">
  <property name="street" value="Abc"/>
  <property name="city" value="Bangalore"/>
  <property name="state" value="Chennai"/>
</bean>
```

The dis-advantage of "byType" is if multiple spring beans are configured for the same type it throws an Exception "UnsatisfiedLinkError".

While we are configuring spring bean into spring bean configuration file we use **id** or **name**.

Difference between id and name attributes :

- **id** attribute doesn't allow to take special character as input. where as **name** attribute allow as to take special character.

```
<bean id="/addr" />   //invalid
<bean name="/addr" />  //valid
```

- By using name attribute to a single name (or) two names to a bean.

```
<bean name="addr1, addr2" class="com.Address"
scope="prototype" />
```

Note : We can use space also as a character for the names in bean type.

```
<bean name="addr1 addr2" />
```

- To the spring beans we can assign the alias names also. The advantage of alias names is different people can use sane bean with different names.

```
<beans>
<bean id="addr1" class="com.core.Address"
scope="singleton"/>
<alias name="addr1" alias="addr2"/>
</beans>
```

- We can ask the spring container creates the object based on alias names also.

```
container.getBean("addr2");
```

Requirement :
In the spring based project we always use multiple spring bean configuration files because of the maintanance of the project becomes easy

configure Employee bean into applicationContext.xml and Address bean into customerContext.xml

We have to supply both these beans parameter to container object, we can use any of the following two approaches

Approach 1:

```
ApplicationContext container=new
ClassPathXmlApplicationContext(
       new
String[]{"com/core/applicationContext.xml",

"com/core/customerContext.xml"});

 container.getBean("addr");
 container.getBean("emp");
```

Approach 2:

```
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/core/applicationContext.xml",

"com/core/customerContext.xml");
 container.getBean("addr");
 container.getBean("emp");
```

In the above configuration we are hard coding the configuration file names instead of hard coding we use **wild character search**

```
ApplicationContext container=
   new
ClassPathXmlApplicationContext("com/core/*.xml");
                            //not working
BeanFactory

 container.getBean("addr");
 container.getBean("emp");
```

We can use an import tag to import one spring bean into another spring bean.

applicationContext.xml

```
<beans>
 <import resource="customerContext.xml"/>
</beans>
```

---

ref tag :

ref tag takes multiple attributes they are : bean, parent, local.

<ref bean="addr"/>

When we specified **<ref bean="addr"/>** spring container checks whether the reference is available in current configuration file, if not available it checks in all the imported configuration file and create the object.

<ref local="addr"/>

When we specified **<ref local="addr"/>** spring container always checks in the current configuration file only.
If it is not available in the current configuration file spring container throws an exception.

<ref parent="addr"/>

When we specified **<ref parent="addr"/>** spring container checks for reference in parent spring container.

In spring we can create multiple spring containers, we can make some container are parent containers, some containers are child containers.

The advantage of using child containers is we can access the beans from child containers as well as parent containers.

The following configuration of parent and child containers

```
ApplicationContext pContainer=new
ClassPathXmlApplicationContext(

"com/core/applicationContext.xml");

ApplicationContext cContainer=new
ClassPathXmlApplicationContext(
     new
String[]{"com/core/customerContext.xml"},pContainer
);

 cContainer.getBean("addr");
 cContainer.getBean("emp");
```

Developing spring based application by using manual procedure :

1. To develop the manual procedure we have to set the class path to following jar files :
    1. beans.jar
    2. core.jar
    3. context.jar
    4. context-support.jar
    5. expression.jar
    6. asm.jar

    As spring internally uses log4j we have to set the classpath to "common-logging.jar"

2. Develop spring bean configuration file.(EX: applicationContext.xml)
3. Develop spring bean and configure in spring configuration file.
4. Develop the java application to call the spring container.

We have to supply the values to properties according to the datatypes of the property.

Data Types of property :

We have to classified data types into 5 categories

1. String type
2. Reference type
3. Primitive type
4. Arrays
5. Collections

null :

We use this tag to supply null value to a property

```
<property name="name">
  <null/>
</property>
```

According to the above configuration name property hold the "null" value.

primitive :

For primitive datatypes we use a tag "value".
We can specify the data type by using "type" attribute.

```
 <property name="no">
  <value type="int">10</value>
 </property>
```

```
 <property name="male">
  <value type="boolean">true</value>
 </property>

 <property name="salary">
  <value type="double">1000.00</value>
 </property>
```

Arrays :

If we have any property whose type is an array to supply the value, we use a tag value with comma(,) separator.

Employee.java

```
package com.core;

public class Employee {

String[] parents;

public String[] getParents() {
     return parents;
}
public void setParents(String[] parents) {
     this.parents = parents;
}

}
```

applicationContext.java

```
<bean id="emp" class="com.core.Employee">
 <property name="parents">
  <value>father,mother</value>
 </property>
</bean>
```

Collections :

ArrayList :

To supply a value to arraylist property we use a tag "list"

Employee.java

```
package com.core;
import java.util.ArrayList;

public class Employee {

ArrayList projects;
public ArrayList getProjects() {
     return projects;
}
public void setProjects(ArrayList projects){
     this.projects = projects;
}

}
```

applicationContext.xml

```
<bean id="emp" class="com.core.Employee">
 <property name="projects">
  <list>
   <value>BMS project</value>
   <value>ZING project</value>
  </list>
 </property>
</bean>
```

MyApp.java

```
package com.core;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
```

```
public class MyApp {

public static void main(String[] args) {

ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/core/applicationContext.xml");

 Employee e=(Employee)container.getBean("emp");
 System.out.println(e.getProjects());

}//main

}

output :

[BMS project, ZING project]
```

Ex 2 :

We are created a project bean, the project bean is responsible for holding the project information.

Project.java

```
package com.core;

public class Project {
 String projectName;
 String customerName;

public String getProjectName() {
     return projectName;
}
public void setProjectName(String projectName) {
```

```
        this.projectName = projectName;
}
public String getCustomerName() {
        return customerName;
}
public void setCustomerName(String customerName) {
        this.customerName = customerName;
}

}
```

An Employee can have multiple projects to store the project information
We are using the following bean with ArrayList property.

Employee.java

```
package com.core;
import java.util.ArrayList;

public class Employee {

ArrayList projects;
public ArrayList getProjects() {
        return projects;
}
public void setProjects(ArrayList projects){
        this.projects = projects;
}

}
```

To supply ArrayList values with object, we use following tags

```
<bean name="bms" class="com.core.Project">
 <property name="projectName" value="BMS project"/>
 <property name="customerName" value="Ashok"/>
</bean>
```

```
<bean name="zing" class="com.core.Project">
 <property name="projectName" value="Zing
project"/>
 <property name="customerName" value="Arun"/>
</bean>

<bean id="emp" class="com.core.Employee">
 <property name="projects">
  <list>
    <ref bean="bms"/>
    <ref bean="zing"/>
  </list>
 </property>
</bean>
```

Set :

Employee.java

```
package com.core;
import java.util.HashSet;

public class Employee {

HashSet projects;
public HashSet getProjects() {
     return projects;
}
public void setProjects(HashSet projects) {
     this.projects = projects;
}

}
```

To configure setProperty to use **set tag** as shown below

applicationContext.xml

```
<bean id="emp" class="com.core.Employee">
```

```
 <property name="projects">
  <set>
   <value>LIC</value>
   <value>KSRTC</value>
  </set>
 </property>
</bean>
```

Map :

Employee.java

```
package com.core;
import java.util.HashMap;

public class Employee {

HashMap projects;
public HashMap getProjects() {
     return projects;
}
public void setProjects(HashMap projects) {
     this.projects = projects;
}

}
```

The following configuration for HashMap

```
<bean id="emp" class="com.core.Employee">
 <property name="projects">
  <map>
   <entry key="p1" value="BMS project"/>
   <entry key="p2" value="Zing project"/>
  </map>
 </property>
</bean>
```

Props :

Employee.java

```
package com.core;
import java.util.Properties;

public class Employee {

Properties projects;
public Properties getProjects() {
     return projects;
}
public void setProjects(Properties projects) {
     this.projects = projects;
}

}
```

applicationContext.xml (spring configuration file)

```
<bean id="emp" class="com.core.Employee">
 <property name="projects">
  <props>
   <prop key="p1">BMS project</prop>
   <prop key="p2">Zing project</prop>
  </props>
 </property>
</bean>
```

Getting the data from spring bean configuration file and storing the data into object is called "Data binding process"
When the data binding process is happening we get "Bind Exception".

We would like to create a spring bean **Student** with sno, name, fatherName, motherName properties

Student.java

```
package com.core;

public class Student {

int no;
String name;
String fatherName;
String motherName;

public int getNo() {
	return no;
}
public void setNo(int no) {
	this.no = no;
}
public String getName() {
	return name;
}
public void setName(String name) {
	this.name = name;
}
public String getFatherName() {
	return fatherName;
}
public void setFatherName(String fatherName) {
	this.fatherName = fatherName;
}
public String getMotherName() {
	return motherName;
}
public void setMotherName(String motherName) {
	this.motherName = motherName;
}

}
```

When we want to use above spring bean to configure into 2 students properties like fatherName, motherName are getting related.

To resolve this problem we have configured the spring bean for configuring fatherName & motherName in one place declared the class as abstract.

Parent.java

```
package com.core;

public abstract class Parent {
 String fatherName;
 String motherName;

public String getFatherName() {
     return fatherName;
}
public void setFatherName(String fatherName) {
     this.fatherName = fatherName;
}
public String getMotherName() {
     return motherName;
}
public void setMotherName(String motherName) {
     this.motherName = motherName;
}

}
```

Student.java

```
package com.core;

public class Student extends Parent{

int no;
String name;
```

```
public int getNo() {
	return no;
}
public void setNo(int no) {
	this.no = no;
}
public String getName() {
	return name;
}
public void setName(String name) {
	this.name = name;
}

}
```

configuration file :

```
<bean id="parents" class="com.core.Parent"
abstract="true">
 <property name="fatherName" value="abc"/>
 <property name="motherName" value="xyz"/>
</bean>

<bean id="students" class="com.core.Student"
parent="parents">
 <property name="no" value="1"/>
 <property name="name" value="SaiCharan"/>
</bean>
```

Java application :

```
package com.core;

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

public class MyApp {

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

```
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/core/applicationContext.xml");

Student s=(Student)container.getBean("students");
 System.out.println(s.getNo());
 System.out.println(s.getMotherName());
}//main

}
```

The advantage of above approach is we can inherit the properties of abstract bean and use it.

Dependency Injection Example :

EmployeeServices.java

```
package com.core;

public interface EmployeeServices {
 public boolean incrementSalary(int eno,double
amount);
}
```

EmployeeServicesImpl.java

```
package com.core;

public class EmployeeServicesImpl implements
EmployeeServices{
private    EmployeeDao employeeDao;

public void setEmployeeDao(EmployeeDao dao){
     employeeDao=dao;
}
```

```
 public boolean incrementSalary(int eno,double
amount){
      double sal=employeeDao.getSal(eno);
      sal+=amount;
      employeeDao.setSal(eno,sal);
      System.out.println("salary :"+ sal);

      return true;
  }
}
```

EmployeeDao.java

```
package com.core;

public interface EmployeeDao {

void setSal(int eno,double amount);
double getSal(int eno);
}
```

EmployeeDaoImpl.java

```
package com.core;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class EmployeeDaoImpl implements
EmployeeDao{

private DataSource dataSource;
public EmployeeDaoImpl(DataSource ds) {
     dataSource=ds;
}
private Connection con;
```

```
public void setSal(int eno,double sal){
 try{
      con=dataSource.getConnection();
      PreparedStatement pstmt=
            con.prepareStatement("update emp set
sal=? where eno=?");
      pstmt.setDouble(1, sal);
      pstmt.setInt(2,eno);
      int count=pstmt.executeUpdate();
      System.out.println("update records :"
+count);
 }
 catch(Exception e){
      e.printStackTrace();
 }
 finally{
     try {
      con.close();
     } catch (Exception e) {
          e.printStackTrace();
     }
 }//finally

}//setSal
public double getSal(int eno){
 try{
  con=dataSource.getConnection();
  PreparedStatement pstmt=
          con.prepareStatement("select sal from
emp where eno=?");
  pstmt.setInt(1,eno);
  ResultSet rs=pstmt.executeQuery();
     if(rs.next())
      return rs.getDouble(1);
      throw new RuntimeException("Employee not
found");
      }//try
```

```
  catch(Exception e){
      e.printStackTrace();
      throw new RuntimeException();
  }
 finally{
      try {
       con.close();
      } catch (SQLException e) {
           e.printStackTrace();
      }
 }//finally
}

}
```

mybeans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
      xsi:schemaLocation="http://www.springframework
.org/schema/beans
      http://www.springframework.org/schema/beans/sp
ring-beans-2.5.xsd">

<bean id="empservices"
class="com.core.EmployeeServicesImpl">
 <property name="employeeDao">
    <ref local="dao"/>
 </property>
</bean>

<bean id="dao" class="com.core.EmployeeDaoImpl">
 <constructor-arg>
   <ref local="ds"/>
 </constructor-arg>
</bean>
```

```
<bean id="ds"
class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName">
  <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url">

<value>jdbc:oracle:thin:@localhost:1521:xe</value>
 </property>
 <property name="username">
   <value>system</value>
 </property>
 <property name="password">
  <value>tiger</value>
 </property>
</bean>

</beans>
```

EmployeeServicesTestCase.java

```
package com.core;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

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

 BeanFactory beans=
     new XmlBeanFactory(new
FileSystemResource("mybeans.xml"));
 EmployeeServices empservices=
```

```
(EmployeeServices)beans.getBean("empservices");

empservices.incrementSalary(Integer.parseInt(args[0
]), 1000d);
}//main

}//class
```

Configuring Beans :

- The Spring Core Container supports the bean configurations for different types of instantiations.
- This is one of the advantage of spring framework allowing us to configure any existing java class to be instantiated and managed by the spring container.
- Spring even supports the configuration of a static inner class for instantiation.

Spring supports 3 types of instantiations :

1. Using constructor
2. Static factory method
3. Non-static factory method

Instantiating Bean using Constructor :

```
<!--instantiating bean with no argument
constructor-->
<bean id="bean1" class="MyBeanClass1"/>

<!--instantiating bean with one String argument
constructor-->
<bean id="bean2" class="MyBeanClass2">
 <constructor-arg type="java.lang.String">
   <value>Hello</value>
```

```
 </constructor-arg>
</bean>

<!--instantiating bean with two argument
constructor-->
<bean id="bean3" class="MyBeanClass3">
 <constructor-arg index="1">
  <value>Hello</value>
 </constructor-arg>

 <constructor-arg index="0">
  <ref local="mybean"/>
 </constructor-arg>
</bean>
```

Instantiating Bean using Static factory method :

syntax :

```
<bean id="mybean" class="com.core.MyFactoryBean"
                    factory-method="getInstance"/>
<bean id="con" class="java.sql.DriverManager"
                                  factory-
method="getConnection">
<constructor-arg>

<value>jdbc:oracle:thin:@localhost:1521:xe</value>
</constructor-arg>
<constructor-arg>
  <value>scott</value>
</constructor-arg>
<constructor-arg>
  <value>tiger</value>
</constructor-arg>

<property name="autoCommit">
 <value type="boolean">false</value>
</property>
```

```
</bean>
```

Instantiating Bean using Non-static factory method :

```
<bean id="bean1" class="com.core.MyFactoryBean"/>

<bean id="bean2" factory-bean="bean1"
                 factory-method="getConnection"/>
```

Bean Scopes and LifeCycle

Scopes	Description
singleton	▪ This is default scope. ▪ A bean defination configured with this scope is instantiated **only once per container instance.** ▪ And all the requests for this bean or references created for this bean in the context will be given with the single object reference.
prototype	▪ A bean defination configured with this scope is instantiated **every time it is requested or referenced**.
request	▪ This scope is applicable only when using a web aware spring ApplicationContext. ▪ A bean defination configured with this scope is instantiated **for each HTTP request**.
session	▪ This scope is applicable only when using a web aware spring ApplicationContext. ▪ A bean defination configured with this scope is instantiated **for each HTTP session**.

	▪ That is, the bean scoped with a session is shared between all the session's requests.
global session	▪ This scope is applicable only when using a web aware spring ApplicationContext. ▪ A bean defination configured with this scope is same as session scope. ▪ Beans defind as the global session scope are scoped to the lifetime of the global portlet session.

Initialization :

- To listen for the initialization lifecycle event the bean can implement **org.springframework.beans.factory.InitializingBean** interface.
- This interface declares only one method **afterPropertiesSet()**

```
<bean id="mybean" class="com.core.BeanLifeCycle"
                        init-method="initialize"/>
```

BeanLifeCycle.java

```
package com.core;

import
org.springframework.beans.factory.InitializingBean;

public class BeanLifeCycle implements
InitializingBean{

public BeanLifeCycle(){
 //invoke before the bean properties are set
//can perform initializations, but bean properties
are not available
```

```
}
public void afterPropertiesSet()throws Exception{
 //invoke after the bean properties are set
successfully
//can perform initializations, even using the bean
properties
}

}//class
```

Spring allows us to configure the default initialization method for all the beans configured in an XML configuration file.

```
<beans default-init-method="initialize">
 <bean id="mybean1"
class="com.core.BeanLifeCycle1"/>
 <bean id="mybean2"
class="com.core.BeanLifeCycle2"/>

 <bean id="mybean3" class="com.core.BeanLifeCycle3"
                                        init-
method="init"/>
</beans>
```

Destruction :

- To listen for the destruction lifecycle event the bean can implement **org.springframework.beans.factory.DisposableBean** interface.
- This interface declares only one method **destroy()**

```
<bean id="mybean" class="com.core.BeanLifeCycle"
                                destroy-
method="close"/>
```

BeanLifeCycle.java

```
package com.core;
```

```
import
org.springframework.beans.factory.DisposableBean;

public class BeanLifeCycle implements
DisposableBean{

public void destroy() throws Exception {
 //invoked just before the bean is put out of
service
//can perform finalizations
}

}//class
```

Spring allows us to configure the default destruction method for all the beans configured in an XML configuration file.

```
 <beans default-destroy-method="close">
  <bean id="mybean1"
class="com.core.BeanLifeCycle1"/>
  <bean id="mybean2"
class="com.core.BeanLifeCycle2"/>
  <bean id="mybean3"
class="com.core.BeanLifeCycle3"
                                destroy-
method="destroy"/>
 </beans>
```

Method Injection

LookUp Method Injection :

BusinessObject1.java

```
package com.core;

public abstract class BusinessObject1 {

public void service(){
```

```
 BusinessObject2 bo2=getBusinessObject2();

}

public abstract BusinessObject2
getBusinessObject2();
}
```

mybeans.xml

```
<bean id="bean1" class="com.core.BusinessObject1"
scope="singleton">
 <lookup-method name="getBusinessObject2"
bean="businessObject2"/>
</bean>

<bean id="businessObject2"
class="com.core.BusinessObject2"

scope="prototype"/>
```

I18N applications in core :

To deal with I18N applications spring guys are provided an interface **org.springframework.context.MessageSource**

This interface contains the methods which are used to deal with I18N applications.

The following UML diagram shows the important methods of interface

MessageSource
getMessage(key, arguments, locale)

'ApplicationContext' interface inherits the properties of 'MessageSource' because of this reason when ever we

create spring container object , it is taking care of I18N applications as part
of **org.springframework.context.support** package.

The following classes provides implementation of MessageSource inteface

- ResourceBundleMessageSource
- ReloadableResourceBundleMessageSource
- StaticMessageSource

By using above classes we provide the information about our property files to spring container.

Procedure to use I18N applications:

1. create the property files based on the number of languages we would like to support.
   (src folder)

```
2.one = one in English
3.two = two in English
4.three = three in English
```

resOne_en_US.properties

```
one = one in French
two = two in French
three = three in French
```

resOne_fr_CA.properties

5. configure **ResourceBundleMessageSource** in spring bean configuration file.

```
<bean id="messageSource" class=
"org.springframework.context.support.ResourceBundleMessageSource">
 <property name="basename" value="resOne"/>
</bean>
```

11. create a spring container object and call a method **getMessage()**

```
package com.core;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.*;


public class MyApp {

public static void main(String[] args) {

ApplicationContext container=new ClassPathXmlApplicationContext(
"com/core/applicationContext.xml");


System.out.println(container.getMessage("one", null, null));
}//main

}
```

28. If we are retrieving a key which is not available we get an exception **NoSuchMessageException**.

29. As part of spring property file we can use configurable messages for example

```
one = one in English {0} and {1}
```

resOne_en_US.properties

31. To supply the values to above parameters we use the following syntax :

```
32. package com.core;
33. import java.util.Locale;
34.
35. import
  org.springframework.context.ApplicationContext;
36. import org.springframework.context.support.*;
37.
38.
39. public class MyApp {
40.
41. public static void main(String[] args) {
42.
43. ApplicationContext container=new
  ClassPathXmlApplicationContext(
44.
  "com/core/applicationContext.xml");
45.
46.
  System.out.println(container.getMessage("one",
47.              new String[]{"valueOne",
  "valueTwo"},
48.
  Locale.getDefault()));
49. }//main
50.
51. }
```

MyApp.java

Using multiple bundles in spring :

To use multiple bundles we have to use a property basenames by using a comma(,) separated values of basenames.

```
<bean id="messageSource" class=

"org.springframework.context.support.ResourceBundle
MessageSource">

 <property name="basenames" value="resOne,resTwo"/>
</bean>
```

1. Aspect Oriented Programming (AOP)

Terminology :

1. Concern
2. Joinpoint
3. Advice
4. Pointcut
5. Aspect
6. Weaving

Spring AOP

Spring Advice API

- Before advice
- After returning advice
- Throws advice
- Around advice

Before Advice

public void before(Method m,)

Working with before advice :

AccountServices.java

```
package com.aop;

public interface AccountServices {
 boolean deposit(int accno,double amt) throws
MyException;
 boolean withdraw(int accno,double amt) throws
MyException;

}
```

AccountServicesImpl.java

```
package com.aop;

public class AccountServicesImpl implements
AccountServices {

private AccountDAO accountDAO;

public AccountServicesImpl(){ }

public AccountServicesImpl(AccountDAO accountDAO){
 this.accountDAO=accountDAO;
}

public boolean deposit(int accno, double amt)
throws MyException {
 System.out.println("in deposit method");
 double bal=accountDAO.getBalance(accno);
 bal+=amt;
 accountDAO.setBalance(accno,bal);
 return true;
}

public boolean withdraw(int accno, double amt)
throws MyException {
 System.out.println("in withdraw method");
```

```
 double bal=accountDAO.getBalance(accno);
 bal-=amt;

 if(bal>=1000){
      accountDAO.setBalance(accno,bal);
      return true;
 }
 return false;
}

}
```

MyException.java

```
package com.aop;

public class MyException extends Exception {

 public MyException(){ }

 public MyException(String message){
  super(message);
 }

}
```

AccountDAO.java

```
package com.aop;

public interface AccountDAO {

 double getBalance(int accno)throws MyException;
 void setBalance(int accno,double amt)throws
MyException;

}
```

AccountDAOTestImpl.java

```
package com.aop;
```

```
public class AccountDAOTestImpl implements
AccountDAO {

 public double getBalance(int accno)throws
MyException {
  return 5000;
 }

 public void setBalance(int accno, double
amt)throws MyException {

 }

}
```

LoggingAdvice.java

```
package com.aop;

import java.lang.reflect.Method;

import org.apache.log4j.Logger;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAdvice implements
MethodBeforeAdvice {

 public void before(Method m, Object[] args, Object
target){
  System.out.println("Logging Advice applied for
:"+m.getName());
  Logger
logger=Logger.getLogger(target.getClass());
  logger.info("Method :"+m.getName()+"invoked
with"+

args.length+"arguments");
 }//before
```

```
}
```

log4j.properties

```
log4j.rootLogger=info, myapp
#log4j.appender.myapp=org.apache.log4j.ConsoleAppen
der

log4j.appender.myapp=org.apache.log4j.FileAppender
log4j.appender.myapp.file=mylog.html
log4j.appender.myapp.layout=org.apache.log4j.HTMLLa
yout
log4j.appender.myapp.append=false
```

mybeans.xml

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

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

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

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

<bean id="accDAO"
class="com.aop.AccountDAOTestImpl"/>

<bean id="accServices"
class="com.aop.AccountServicesImpl">
 <constructor-arg>
  <ref local="accDAO"/>
 </constructor-arg>
</bean>
```

```
<bean id="logging" class="com.aop.LoggingAdvice"/>
<bean id="accountServices"

class="org.springframework.aop.framework.ProxyFacto
ryBean">
 <property name="targetName">
  <value>accServices</value>
 </property>
 <property name="proxyInterfaces">
  <list>
   <value>com.aop.AccountServices</value>
  </list>
 </property>
 <property name="interceptorNames">
  <list>
   <value>logging</value>
  </list>
 </property>
</bean>

</beans>
```

AccountServicesTestCase.java

```
package com.aop;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

public class AccountServicesTestCase {

public static void main(String args[])throws
Exception{
 BeanFactory beans=
```

```
   new XmlBeanFactory(new
FileSystemResource("src/mybeans.xml"));
 AccountServices
as=(AccountServices)beans.getBean("accountServices"
);
 System.out.println(as.withdraw(1, 1000));
 System.out.println("\n");
 System.out.println(as.deposit(1, 1000));
}
}
```

result :

```
Logging Advice applied for : withdraw
in withdraw method
true

Logging Advice applied for : deposit
in deposit method
true
```

After Returning Advice

Working with AfterReturningAdvice :

MyBusinessObject.java

```
package com.aop;

public class MyBusinessObject{
 public String businessService(int id)throws
MyException{
  System.out.println("in business service");
  if(id==0){
   System.out.println("returning test1...");
   return "Test1";
  }
  else if(id==1){
   System.out.println("returning test2...");
   return "Test2";
```

```
  }
  else {
     System.out.println("returning Hello...");
     return "Hello from business service";
  }
 }
}
```

MyAfterReturningAdvice.java

```
package com.aop;

import java.lang.reflect.Method;

import
org.springframework.aop.AfterReturningAdvice;

public class MyAfterReturningAdvice implements
AfterReturningAdvice{

 public void afterReturning(Object return_value,
Method m,
                   Object[] args,Object
target)throws Throwable {
  System.out.println("In MyAfterReturningAdvice
:"+m.getName());
  System.out.println("Accessing return value...");
  if(return_value.equals("Test1")){
   System.out.println("return value found Test1");
   System.out.println("advice throwing
MyException");
   throw new MyException("MyException from
AfterReturningAdvice");
  }
  if(return_value.equals("Test2")){
   System.out.println("return value found Test2");
   System.out.println("advice throwing
MyException");
```

```
   throw new MyException("MyException from
AfterReturningAdvice");
  }
  System.out.println("return value found
:"+return_value);
  System.out.println("advice ending without
throwing exception...");
 }//afterReturning

}//class
```

MyException.java

```
package com.aop;

public class MyException extends Exception {

 public MyException(){ }

 public MyException(String message){
  super(message);
 }

}
```

MyBusinessObjectTestCase.java

```
package com.aop;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

public class MyBusinessObjectTestCase {
```

```
 public static void main(String[] args)throws
Exception{
  BeanFactory beans=
       new XmlBeanFactory(new
FileSystemResource("src/mybeans.xml"));
  MyBusinessObject
bo=(MyBusinessObject)beans.getBean("myServices");
  System.out.println("Testing normal flow ...");
  System.out.println(bo.businessService(2));
  System.out.println("\n");

  try{
   System.out.println("Testing flow when
MyException is thrown by advice");
   System.out.println(bo.businessService(0));
  }
  catch (MyException e) {
   System.out.println(e);
  }
  System.out.println("\n");
  try{
   System.out.println("Testing flow when
MyException is thrown by advice");
   System.out.println(bo.businessService(1));
  }
  catch (Exception e) {
   System.out.println(e);
  }

 }//main
}//class
```

mybeans.xml

```
<beans>

<bean id="myser" class="com.aop.MyBusinessObject"/>
```

```
<bean id="afterReturning"
class="com.aop.MyAfterReturningAdvice"/>

<bean id="myServices"

class="org.springframework.aop.framework.ProxyFacto
ryBean">
 <property name="targetName" value="myser"/>
 <property name="interceptorNames">
  <list>
   <value>afterReturning</value>
  </list>
 </property>
</bean>

</beans>
```

result :

```
Testing normal flow ...
in business service
returning Hello...
In MyAfterReturningAdvice :businessService
Accessing return value...
return value found :Hello from business service
advice ending without throwing exception...
Hello from business service


Testing flow when MyException is thrown by advice
in business service
returning test1...
In MyAfterReturningAdvice :businessService
Accessing return value...
return value found Test1
advice throwing MyException
com.aop.MyException: MyException from
AfterReturningAdvice
```

```
Testing flow when MyException is thrown by advice
in business service
returning test2...
In MyAfterReturningAdvice :businessService
Accessing return value...
return value found Test2
advice throwing MyException
com.aop.MyException: MyException from
AfterReturningAdvice
```

THROWS ADVICE

Working with throws advice :

MyBusinessObject.java

```
package com.aop;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

public class MyBusinessObjectTestCase {

 public static void main(String[] args)throws
Exception{
  BeanFactory beans=
       new XmlBeanFactory(new
FileSystemResource("src/mybeans.xml"));
  MyBusinessObject
bo=(MyBusinessObject)beans.getBean("myServices");

  try{
```

```
   System.out.println("Testing flow when
MyException1 is
                        thrown by
businessService()...");
   bo.businessService(0);
  }
  catch (MyException1 e){
   System.out.println(e);
  }

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

  try{
   System.out.println("Testing flow when
MyException2
               is thrown by businessService()...");
   bo.businessService(1);
  }
  catch (Exception e) {
   System.out.println(e);
  }

  System.out.println("\n");
  System.out.println("Testing flow when there is
         no MyException thrown by
businessService()...");
  bo.businessService(2);
 }//main
}//class
```

MyException1.java

```
package com.aop;

public class MyException1 extends Exception {

 public MyException1(){ }

 public MyException1(String message){
```

```
  super(message);
 }

}
```

MyException2.java

```
package com.aop;

public class MyException2 extends Exception {

 public MyException2(){ }

 public MyException2(String message){
  super(message);
 }

}
```

MyThrowsAdvice.java

```
package com.aop;

import org.springframework.aop.ThrowsAdvice;

public class MyThrowsAdvice implements
ThrowsAdvice{

 public void afterThrowing(MyException1 me)throws
Throwable{
 //Exception handling logic as per our system
requirement test logic
  System.out.println("In
afterThrowing(MyException1)");
 }

 public void afterThrowing(MyException2 me)throws
Throwable{
 //Exception handling logic as per our system
requirement test logic
```

```
  System.out.println("In
afterThrowing(MyException2)");
 }

}
```

mybeans.xml

```
<beans>

<bean id="myser" class="com.aop.MyBusinessObject"/>

<bean id="exceptionHandler"
class="com.aop.MyThrowsAdvice"/>

<bean id="myServices"

class="org.springframework.aop.framework.ProxyFacto
ryBean">
 <property name="targetName" value="myser"/>
 <property name="interceptorNames">
  <list>
   <value>exceptionHandler</value>
  </list>
 </property>
</bean>

</beans>
```

MyBusinessObjectTestCase.java

```
package com.aop;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;
```

```
public class MyBusinessObjectTestCase {

 public static void main(String[] args)throws
Exception{
  BeanFactory beans=
       new XmlBeanFactory(new
FileSystemResource("src/mybeans.xml"));
  MyBusinessObject
bo=(MyBusinessObject)beans.getBean("myServices");

  try{
   System.out.println("Testing flow when
MyException1 is
                thrown by businessService()...");
   bo.businessService(0);
  }
  catch (MyException1 e){
   System.out.println(e);
  }

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

  try{
   System.out.println("Testing flow when
MyException2 is
                          thrown by
businessService()...");
   bo.businessService(1);
  }
  catch (Exception e) {
   System.out.println(e);
  }

  System.out.println("\n");
  System.out.println("Testing flow when there is
```

```
        no MyException thrown by
businessService()...");
  bo.businessService(2);
 }//main
}//class
```

output :

```
Testing flow when MyException1 is thrown by
businessService()...
in business service
throwing MyException1...
In afterThrowing(MyException1)
com.aop.MyException1: MyException1 from
MyBusinessObject

Testing flow when MyException2 is thrown by
businessService()...
in business service
throwing MyException2...
In afterThrowing(MyException2)
com.aop.MyException2: MyException2 from
MyBusinessObject

Testing flow when there is no MyException thrown by
businessService()...
in business service
ending without throwing any exception
```

AROUND ADVICE

Working with around advice :

MyBusinessObject.java

```
package com.aop;

public class MyBusinessObject{
 public String businessService(int id)throws
MyException1, MyException2{
```

```
  System.out.println("in business service");

  if(id==0){
   System.out.println("throwing MyException1...");
   throw new MyException1("MyException1 from
MyBusinessObject");
  }
  else if(id==1){
   System.out.println("throwing MyException2...");
   throw new MyException2("MyException2 from
MyBusinessObject");
  }
  else if(id==2){
   System.out.println("returning Test1...");
   return "Test1";
  }
  else{
   System.out.println("returning Test2...");
   return "Test2";
  }

 }//businessService
}
```

MyAroundAdvice.java

```
package com.aop;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class MyAroundAdvice implements
MethodInterceptor {

 public Object invoke(MethodInvocation mi) throws
Throwable {
  //all the pre processing (before advice)
 try{
```

```
  System.out.println("invoke() method before
calling proceed()");
  Object o= mi.proceed();

  if(o.equals("Test1")){
   System.out.println("invoke() after executing
proceed()");
   return "Method returned Test1 but advice has
changed";
  }
   return o;
 }//try
 catch (MyException1 e) {
  System.out.println("invoke(), proceed() thrown
MyException1");
  return "Hello, there was an Exception";
 }

 }//invoke

}
```

MyException1.java

```
package com.aop;

public class MyException1 extends Exception {

 public MyException1(){ }

 public MyException1(String message){
  super(message);
 }

}
```

MyException2.java

```
package com.aop;
```

```
public class MyException2 extends Exception {

 public MyException2(){ }

 public MyException2(String message){
  super(message);
 }

}
```

log4j.properties

```
log4j.rootLogger=info, myapp
#log4j.appender.myapp=org.apache.log4j.ConsoleAppen
der

log4j.appender.myapp=org.apache.log4j.FileAppender
log4j.appender.myapp.file=mylog.html
log4j.appender.myapp.layout=org.apache.log4j.HTMLLa
yout
log4j.appender.myapp.append=false
```

mybeans.xml

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

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

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

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

<bean id="myser" class="com.aop.MyBusinessObject"/>
```

```
<bean id="around" class="com.aop.MyAroundAdvice"/>

<bean id="myServices"

class="org.springframework.aop.framework.ProxyFacto
ryBean">
 <property name="targetName" value="myser"/>
 <property name="interceptorNames">
  <list>
   <value>around</value>
  </list>
 </property>
</bean>

</beans>
```

MyBusinessObjectTestCase.java

```
package com.aop;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactor
y;
import
org.springframework.core.io.FileSystemResource;

public class MyBusinessObjectTestCase {

 public static void main(String[] args)throws
Exception{
  BeanFactory beans=
        new XmlBeanFactory(new
FileSystemResource("src/mybeans.xml"));
  MyBusinessObject
bo=(MyBusinessObject)beans.getBean("myServices");
```

```
  System.out.println("Testing flow when
MyException1 is" +
                          " thrown by
businessService()...");
  System.out.println(bo.businessService(0));
  System.out.println("\n");
  try{
   System.out.println("Testing flow when
MyException2 is" +
                          " thrown by
businessService()...");
   System.out.println(bo.businessService(1));
  }
  catch (MyException2 e){
   System.out.println(e);
  }

  System.out.println("\n");
  System.out.println("Testing flow when
businessService() returns Test1...");
  System.out.println(bo.businessService(2));

  System.out.println("\n");
  System.out.println("Testing flow when
businessService() returns Test2...");
  System.out.println(bo.businessService(3));
 }//main
}//class
```

output :

```
Testing flow when MyException1 is thrown by
businessService()...
invoke() method before calling proceed()
in business service
throwing MyException1...
invoke(), proceed() thrown MyException1
Hello, there was an Exception
```

```
Testing flow when MyException2 is thrown by
businessService()...
invoke() method before calling proceed()
in business service
throwing MyException2...
com.aop.MyException2: MyException2 from
MyBusinessObject


Testing flow when businessService() returns
Test1...
invoke() method before calling proceed()
in business service
returning Test1...
invoke() after executing proceed()
Method returned Test1 but advice has changed


Testing flow when businessService() returns
Test2...
invoke() method before calling proceed()
in business service
returning Test2...
Test2
```

Working with Spring Pointcut and Advisors

Agenda :

1. Introduction to Spring Framework

Types of Pointcuts

1. Static Pointcut

2. Dynamic Pointcut

Static Pointcut

NameMatchMethodPointcut :

mybeans.xml

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

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

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

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

<bean id="accdao"
class="com.pointcut.AccountDAOTestImpl"/>

<bean id="accServices"
class="com.pointcut.AccountServicesImpl">
 <constructor-arg>
  <ref local="accdao"/>
 </constructor-arg>
</bean>

<bean id="logging"
class="com.pointcut.LoggingAdvice"/>

<bean id="mypointcut"

class="org.springframework.aop.support.NameMatchMet
hodPointcut">
```

```
 <property name="mappedName">
  <value>withdraw</value>
 </property>
</bean>

<bean id="loggingAdvisor"

class="org.springframework.aop.support.DefaultPoint
cutAdvisor">
 <property name="advice">
  <ref local="logging"/>
 </property>
 <property name="pointcut">
  <ref local="mypointcut"/>
 </property>
</bean>

<bean id="accountServices"

class="org.springframework.aop.framework.ProxyFacto
ryBean">
 <property name="targetName" value="accServices"/>
 <property name="interceptorNames">
  <list>
   <value>loggingAdvisor</value>
  </list>
 </property>
</bean>

</beans>
```

Regular Expression Method Pointcut :

```
<bean id="mypointcut"

class="org.springframework.aop.support.JdkRegexpMet
hodPointcut">
 <property name="patterns">
  <list>
```

```
    <value>.*get.*</value>
    <value>.*set.*</value>
   </list>
  /property>
</bean>
```

Dynamic Pointcuts

Control Flow Pointcut :

```
<bean id="mypointcut"

class="org.springframework.aop.support.ControlFlowP
ointcut">
 <constructor-arg>
  <value>com.aop.WithDrawService</value>
 </constructor-arg>
</bean>
```

Spring 2.0 AOP Support

DAO (Data Access Object)

Agenda :

1. Introduction to Spring Framework DAO Module

Introduction :

- As part of DAO module of spring, we can write regular JDBC code but the developer will not find any advantage of using Spring.
- When we write the JDBC code developer has to write all traditional code as well as developer has to take care of all the exceptions.
- The advantage of using DAO module is we no need to provide huge amount of code as well as we no need to handle error.
- When we use DAO module we can use the pre-defined classes given by the spring because of this we can deliver the project quickly.
- Spring uses "Template Design Pattern"
- As part of **org.springframework.jdbc.core** package contains **JDBCTemplate** class,
  As part of this class the common code which is used in all the projects available.

The following is example of JDBC Template :

JDBCTemplate.java

```
package com.dao;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import javax.sql.DataSource;

public class JDBCTemplate {

private DataSource dataSource;

public void setDataSource(DataSource dataSource){
     this.dataSource=dataSource;
}
```

```
public DataSource getDataSource(){
      return dataSource;
}

public int update(String query) throws
SQLException{
Connection con=dataSource.getConnection();
Statement stmt=con.createStatement();
int result=stmt.executeUpdate(query);
stmt.close();
con.close();
return result;
}//update

}
```

JDBCTemplate is dependent on DataSource object.

- In spring we can use 3 categories of dataSource objects, they are default the spring guys are develops so many dummy connection pool programs.
  They are available in a
  package **org.springframework.jdbc.dataSource**.
- By default spring is integrated with **DBCP connection pool** and **C3P connection pool**
- We can use **Weblogic connection pool**.

Procedure to use DAO module in Spring :

- To the Project add spring capabilities, to add DAO module we need to check the checkbox spring 3.0 persistence JDBC library.
- We try to use JDBCTemplate class by creating the object manually.

```
package com.dao;
```

```
import org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.jdbc.datasource.DriverManagerDa
taSource;

public class MyApp {
public static void main(String[] args) {
DriverManagerDataSource ds=new
DriverManagerDataSource();
ds.setDriverClassName("oracle.jdbc.driver.OracleDri
ver");
ds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
ds.setUsername("scott");
ds.setPassword("tiger");

JdbcTemplate jt=new JdbcTemplate();
jt.setDataSource(ds);
jt.update("insert into product
values(1,'pone',2015)");
}

}
```

- In the above example developer is creating the object and establish the dependencies instead of developer doing this work spring container can create the object establish dependencies.

Procedure to use JdbcTemplate in a project :

- JDBCTemplate is dependent on dataSource, configure **org.springframework.jdbc.datasource.DriverManagerDataSource** into spring bean configuration file supply **driver class, url, userName, password** as dependencies.

applicationContext.xml

```
<bean id="ds"

class="org.springframework.jdbc.datasource.DriverMa
nagerDataSource">

 <property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="username" value="scott"/>
 <property name="password" value="tiger"/>
</bean>

<bean id="jt"
class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource">
  <ref bean="ds"/>
 </property>
</bean>
```

The following java code to insert a record into DataBase server.

```
package com.dao;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(
```

```
"com/core/applicationContext.xml");
JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);
int no=jt.update("insert into product
values(23,'pone',3450)");
System.out.println(no);
}

}
```

Procedure to use DBCP connection pool :

1. create DataBase driver from myeclipse Database explorer.
2. create java project and add spring capabilities.
3. configure DataSource **(spring explorer --> beans --> new datasource --> choose db server)**, this will populate all the required field.
4. configure JdbcTemplate
5. Get the JdbcTemplate object and call the **update()**

applicationContext.xml

```
 <bean id="ds"
class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="username" value="scott"/>
 <property name="password" value="tiger"/>
</bean>

<bean id="jt"
class="org.springframework.jdbc.core.JdbcTemplate">
```

```
 <property name="dataSource">
  <ref bean="ds"/>
 </property>
</bean>
package com.dao;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");
JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);
int no=jt.update("insert into product
values(2,'pone', 3450)");
System.out.println("success");
}

}
```

Using Weblogic connection pool in spring :

1. configure connection pool in weblogic server.
   for JNDI Connection Pool
2. create the project and add spring capabilities.
3. configure JNDI object factory bean

```
4. <bean id="jndiPool"
```

```
5.
  class="org.springframework.jndi.JndiObjectFacto
  ryBean">
6.  <property name="jndiName" value="myPool"/>
7.  <property name="jndiEnvironment">
8.   <props>
9.    <prop key="java.naming.factory.initial">
10.     weblogic.jndi.WLInitialContextFactory
11.    </prop>
12.    <prop key="java.naming.provider.url">
13.     t3://localhost:7001/
14.    </prop>
15.   </props>
16.  </property>
17.</bean>
18.
19.<bean id="ds" class=
20.
  "org.springframework.jdbc.datasource.UserCreden
  tialsDataSourceAdapter">
21.  <property name="targetDataSource"
  ref="jndiPool"/>
22.  <property name="username" value="weblogic"/>
23.  <property name="password" value="weblogic"/>
24.</bean>
25.
26.
27.<bean id="jt"
  class="org.springframework.jdbc.core.JdbcTempla
  te">
28.  <property name="dataSource" ref="ds"/>
29.</bean>
```

30. Get the spring container object call a method **update()** to perform any curd operations.

```
31. package com.dao;
```

```
32.
33.import
  org.springframework.context.ApplicationContext;
34.import org.springframework.context.support.*;
35.import
  org.springframework.jdbc.core.JdbcTemplate;
36.
37.
38.public class MyApp {
39.public static void main(String[] args) {
40.ApplicationContext container=
41.  new
  ClassPathXmlApplicationContext("com/dao/applica
  tionContext.xml");
42.JdbcTemplate
  jt=container.getBean("jt",JdbcTemplate.class);
43.int no=jt.update("insert into product
  values(24,'pone',3450)");
44.}
45.
46.}
```

set the class path to **weblogic.jar**

Requirement : Write a program JDBC application in DAO module to insert, update, delete the record.

The following example to demonstrate how to use preparedStatement in spring DAO module.

```
<bean id="ds"
class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="username" value="scott"/>
```

```
 <property name="password" value="tiger"/>
</bean>

<bean id="jt"
class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="ds"/>
</bean>
package com.dao;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");
JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);
String query="insert into product values(?,?,?)";

Object obj[]={4,"four",4000};
int no=jt.update(query,obj); //OR

int no1=jt.update(query,"5","five","5000");

System.out.println("success");
}

}
```

Callback Mechanism :

String internally uses callback mechanism our java application call the methods of spring.
The internal code of spring call the methods of java application, this technique is called as **callback** mechanism

To retrieve the data from database server in DAO module, We have to take the help of callback interface **ResultSetExtractor**. This interface is having a method **extractData()**

```
syntax :

Object extractData(ResultSet rs);
```

We need to develop a class which provides the implementation of **ResultSetExtractor**

Develop a class which provides the implementation of ResultSetExtractor.

GetData.java

```
package com.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.dao.DataAccessException;
import
org.springframework.jdbc.core.ResultSetExtractor;

public class GetData implements ResultSetExtractor
{

public Object extractData(ResultSet rs) throws
SQLException,

DataAccessException {
 while(rs.next()){
  System.out.println(rs.getString(1));
```

```
  System.out.println(rs.getString(2));
  System.out.println(rs.getString(3));
 }//while
 return null;
}//extractData

}
```

We use a method **query()** to retrive the records and display it.

```
package com.dao;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");
JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);
String query="select * from product";
int no=jt.query(query, new GetData());

System.out.println("success");
}

}
```

1. When ever we call the query() it sends the query to DataBase and DataBase Server return a ResultSet object.

2. Now the query() will call the extractData() by supplying ResultSet object.

- In the above example extractData(), we are displaying the records in the project, we will never display records in DAO component, we will always display output in view component.
- To achieve this we represent every record in the form of object in extractData() method.

Example

Product.java

```
package com.dao;

public class Product {
String pid;
String pname;
String price;
public String getPid() {
     return pid;
}
public void setPid(String pid) {
     this.pid = pid;
}
public String getPname() {
     return pname;
}
public void setPname(String pname) {
     this.pname = pname;
}
public String getPrice() {
     return price;
}
public void setPrice(String price) {
     this.price = price;
```

```
}

}
```

GetData.java

```
package com.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import org.springframework.dao.DataAccessException;
import
org.springframework.jdbc.core.ResultSetExtractor;

public class GetData implements ResultSetExtractor
{

public Object extractData(ResultSet rs) throws
SQLException,

DataAccessException {
 ArrayList list=new ArrayList();
 while(rs.next()) {
  Product p=new Product();
  p.setPid(rs.getString(1));
  p.setPname(rs.getString(2));
  p.setPrice(rs.getString(3));

  list.add(p);
}//while

 return list;
}//extractData

}
```

MyApp.java

```
package com.dao;
```

```
import java.util.ArrayList;

import
org.springframework.context.ApplicationContext;
import org.springframework.context.support.*;
import org.springframework.jdbc.core.JdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");
JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);
String query="select * from product";
ArrayList list=(ArrayList)jt.query(query, new
GetData());

java.util.Iterator i=list.iterator();

while(i.hasNext()){
Product p= (Product) i.next();
System.out.println(p.getPid());
System.out.println(p.getPname());
System.out.println(p.getPrice());
System.out.println("------");
}//while

System.out.println("success");
}//main

}
```

Spring guys are provided the following Template classes as part of DAO module.

1. JdbcTemplate
2. NamedParameterJdbcTemplate
3. SimpleJdbcTemplate
4. SimpleJdbcInsert
5. SimpleJdbcCall

NamedParameterJdbcTemplate :

- The purpose of NamedParameterJdbcTemplate is instead of using treditional '?' place holders we use variable names.
- The advantage of using NamedParameterJdbcTemplate is it improves readability of queries.
- Ex:
-
- insert into product values(:pid,:pname,:price);
- NamedParameterJdbcTemplate is dependent on DataSource object.
- NamedParameterJdbcTemplate is uses constructor injection.

Procedure to work with NamedParameterJdbcTemplate :

configure DataSource and NamedParameterJdbcTemplate by using constructor injection.

```
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="username" value="scott"/>
 <property name="password" value="tiger"/>
</bean>
```

```
<bean id="npjt"
class="org.springframework.jdbc.core.namedparam.Nam
edParameterJdbcTemplate">
 <constructor-arg type="javax.sql.DataSource"
index="0">
   <ref bean="dataSource"/>
 </constructor-arg>
</bean>
```

MyApp.java

```
package com.dao;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import
org.springframework.jdbc.core.namedparam.MapSqlPara
meterSource;
import
org.springframework.jdbc.core.namedparam.NamedParam
eterJdbcTemplate;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");
NamedParameterJdbcTemplate npjt=

container.getBean("npjt",NamedParameterJdbcTemplate
.class);

String query="insert into product
values(:pid,:pname,:price)";
```

```
MapSqlParameterSource map=new
MapSqlParameterSource();
map.addValue("pid", "10");
map.addValue("pname", "scoda");
map.addValue("price", "500000");

npjt.update(query, map);

System.out.println("success");
}//main

}
```

SimpleJdbcTemplate :

SimpleJdbcTemplate class clubbed functionality of JdbcTemplate and NamedParameterJdbcTemplate.

- In the projects the code which is responsible to interact with DataBase must be provided in DAO class.
- The DAO class contains set of methods which are responsible to interact with table to perform all curd operations.

DAO class is dependent on JdbcTemplate the following code :

Product.java

```
package com.dao;

public class Product {
String pid;
String pname;
String price;
public String getPid() {
     return pid;
```

```
}
public void setPid(String pid) {
	this.pid = pid;
}
public String getPname() {
	return pname;
}
public void setPname(String pname) {
	this.pname = pname;
}
public String getPrice() {
	return price;
}
public void setPrice(String price) {
	this.price = price;
}

}
```

ProductDAO.java

```
package com.dao;

import java.util.ArrayList;
import org.springframework.jdbc.core.JdbcTemplate;

public class ProductDAO {
 JdbcTemplate jdbcTemplate;

public JdbcTemplate getJdbcTemplate() {
	return jdbcTemplate;
}

public void setJdbcTemplate(JdbcTemplate
jdbcTemplate) {
	this.jdbcTemplate = jdbcTemplate;
}
```

```
public int storeProductDetails(Object... values){
 String query="insert into product values(?,?,?)";
 return jdbcTemplate.update(query,values);
}

public int updateProductDetails(Object... values){
 String query="update product set pid=? where
pname=?";
 return jdbcTemplate.update(query,values);
}

public int deleteProductDetails(Object... pid){
 String query="delete from product where pid=?";
 return jdbcTemplate.update(query,pid);
}

public ArrayList getAllProductRecords(){
 String query="select * from product";
 return (ArrayList)jdbcTemplate.query(query,new
ProductRowMapper());
}

}
```

ProductRowMapper.java

```
package com.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class ProductRowMapper implements RowMapper
{

public Object mapRow(ResultSet rs, int rowNum)
throws SQLException {

 Product p=new Product();
```

```
 p.setPid(rs.getString(1));
 p.setPname(rs.getString(2));
 p.setPrice(rs.getString(3));
 return p;
}

}
```

Configure DataSource, JdbcTemplate, DAO in spring bean configuration file.

```
<bean id="ds"
class="org.apache.commons.dbcp.BasicDataSource">
 <property name="driverClassName">
   <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
 <property name="username" value="scott"/>
 <property name="password" value="tiger"/>
</bean>

<bean id="jt"
class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="ds"/>
</bean>

<bean id="productDao" class="com.dao.ProductDAO">
 <property name="jdbcTemplate">
  <ref bean="jt"/>
 </property>
</bean>
```

MyApp.java

```
package com.dao;

import java.util.ArrayList;
```

```
import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(

"com/dao/applicationContext.xml");

ProductDAO
productDAO=container.getBean("productDao",ProductDA
O.class);

productDAO.storeProductDetails(6,"six",60000);
productDAO.updateProductDetails(66,"six");
productDAO.deleteProductDetails(66);

ArrayList list=productDAO.getAllProductRecords();
java.util.Iterator i=list.iterator();

while(i.hasNext()){
Product p= (Product) i.next();
System.out.println(p.getPid());
System.out.println(p.getPname());
System.out.println(p.getPrice());
System.out.println("------");
}//while

System.out.println("success");
}//main

}
```

When we use RowMapper (or) ResultSetExtractor as seperate classes the no. of programs in project are becoming more because of this we get maintanance related problems.

We have developed a class COne with a method which take an interface implementation class object.

- Who ever call doWork() method they must supply implementation class object of MyInterface.
- Instead of developing a seperate implementation class we can use an anonymous inner class as shown below.

The following example uses anonymous inner class in spring instead of developing our own RowMapper() object

```
package com.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApp
licationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

public class MyApp {
public static void main(String[] args) {
ApplicationContext container=new
ClassPathXmlApplicationContext(
```

```
"com/dao/applicationContext.xml");

JdbcTemplate
jt=container.getBean("jt",JdbcTemplate.class);

String query="select * from Product";
ArrayList list=(ArrayList)jt.query(query, new
RowMapper(){

public Object mapRow(ResultSet rs,int rowNum)throws
SQLException{
 Product p=new Product();
 p.setPid(rs.getString(1));
 p.setPname(rs.getString(2));
 p.setPrice(rs.getString(3));
 return p;
}//mapRow

}//RowMapper
 );

java.util.Iterator i=list.iterator();

while(i.hasNext()){
Product p= (Product) i.next();
System.out.println(p.getPid());
System.out.println(p.getPname());
System.out.println(p.getPrice());
System.out.println("------");
}//while

System.out.println("success");
}//main

}
```

The following example demonstrate use batchUpdate in Spring :

```
public void batchUpdates(final ArrayList products){
 String query="insert into product values(?,?,?)";
jdbcTemplate.batchUpdate(query,new
BatchPreparedStatementSetter() {

 public void setValues(PreparedStatement ps,int
i)throws SQLException{
  Product p=(Product) products.get(i);
  ps.setString(1, p.getPid());
  ps.setString(2, p.getPname());
  ps.setString(3, p.getPrice());
 }

 public int getBatchSize() {
  return products.size();
 }

}); //batchUpdate


}//batchUpdates
```

1. Spring Web MVC Introduction

Spring Web Module

Agenda :

1. Introduction :

- Spring Web module is uses MVC2 architecture.
- The advantage of spring, it is clean separation of model, view, controller and DAO components.

Procedure to develop Web based applications by using Spring

1. create web-based application by using IDE.
2. add spring capabilities and make sure that spring bean configuration file is placed in 'WEB-INF' folder.
3. add ContextLoaderListener in web.xml

```
4.<web-app>
5.<listener>
6. <listener-class>
7.
  org.springframework.web.context.ContextLoaderLi
  stener
8. </listener-class>
9.</listener>
10.</web-app>
```

11. Provide the information of spring bean configuration file in web.xml by using context parameter.

```
12.<web-app>
13.<context-param>
14. <param-name>contextConfigLocation</param-
  name>
15. <param-value>/WEB-
  INF/applicationContext.xml</param-value>
16.</context-param>
17.</web-app>
```

18. add DispatcherServlet in web.xml

```
19.<web-app>
20.<servlet>
21. <servlet-name>spring</servlet-name>
22. <servlet-class>
```

```
23.
  org.springframework.web.servlet.DispatcherServl
  et
24.  </servlet-class>
25.  <load-on-startup>1</load-on-startup>
26.</servlet>
27.<servlet-mapping>
28.  <servlet-name>spring</servlet-name>
29.  <url-pattern>*.htm</url-pattern>
30.</servlet-mapping>
31.</web-app>
```

32. Create another spring bean configuration file, the name of the file must be spring-servlet.xml
33. Create a folder whose folder name is **pages** (this folder is used to hold the jsp's of a project)
34. configure **InternalResourceViewResolver** into spring bean configuration file

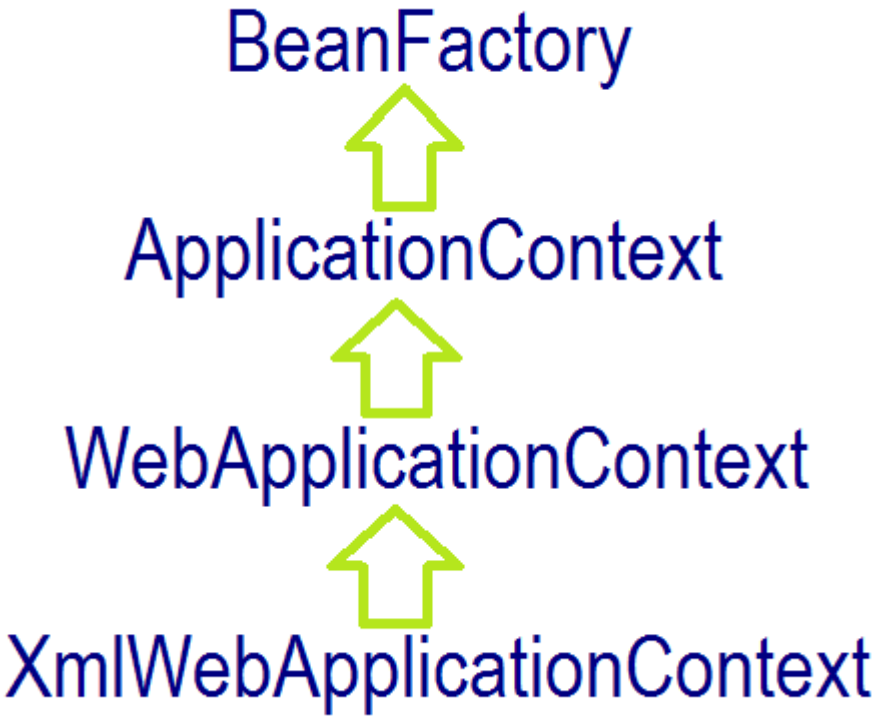
```
35.<bean id="jspView"
36.
  class="org.springframework.web.servlet.view.Int
  ernalResourceViewResolver">
37.  <property name="prefix" value="/pages/"/>
38.  <property name="suffix" value=".jsp"/>
39.</bean>
```

spring-servlet.xml

The following steps are carried out when we deployed spring based application

1. Server reads the contents from web.xml
2. Server creates ServletContext object
3. Server raises an event and check for appropriate event handler in web.xml (Listener)

4. Now the server create object to ContextLoaderListener and execute contextInitialized() method.
5. contextInitialized() method get the Servlet Context object and get the spring bean configuration file name.
6. The Listener uses XmlWebApplicationContext to create the spring container object.
7. In the web-based application we use WebApplicationContext interface to create the spring container object.
8. We use a class XmlWebApplicationContext to create the spring container.



9. The following is the internal code of ContextLoaderListener

```
10.import javax.servlet.ServletContext;
11.import javax.servlet.ServletContextEvent;
12.import javax.servlet.ServletContextListener;
13.import
  org.springframework.web.context.WebApplicationC
  ontext;
```

```
14. import
  org.springframework.web.context.support.XmlWebA
  pplicationContext;
15.
16.
17. public class ContextLoaderListener implements
  ServletContextListener {
18.
19. public void
  contextInitialized(ServletContextEvent sce) {
20.  ServletContext
  application=sce.getServletContext();
21.  String
  fileName=application.getInitParameter("contextC
  onfigLocation");
22.  WebApplicationContext container=new
  XmlWebApplicationContext(fileName);
23. }
24.
25. public void
  contextDestroyed(ServletContextEvent sce) {
26.  //close the spring container object
27. }
28.
29. }//class
```

Note : We call the spring container created by the Listener class as parent spring container object.

What will happened the client send the request whose url-pattern ends with *.htm

1. Server creates DispatcherServlet object
2. server called 2nd init() method
3. It will find the name of the servlet configured in web.xml file

4. Dynamically it will find the spring bean configuration file and create child spring container object.

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.http.HttpServlet;
import
org.springframework.web.context.WebApplicationConte
xt;
import
org.springframework.web.context.support.XmlWebAppli
cationContext;


public class DispatcherServlet extends HttpServlet
{

public void init(){
 ServletConfig config=getServletConfig();
 String name=config.getServletName();
 String fileName=name+"-servlet.xml";
 WebApplicationContext context=new
XmlWebApplicationContext(fileName);
}

}//class
```

In spring, if we want to carryout any work we must develop controller classes.

The following is the UML diagram of Controller interface

Controller

ModelAndView handleRequest(HttpServletRequest,HttpServletResponse)

- Container is a java program, which provide the implementation of Controller interface directly or indirectly.
- Develop a controller configure into spring bean configuration file display it and test it.
- All the Controller classes must be configured in child spring bean (spring-servlet.xml)
- The DAO classes and business service classes must be configured in parent spring container.

Develop a Controller class :

```
package com.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.Controller;

public class FirstController implements Controller
{

public ModelAndView
handleRequest(HttpServletRequest request,
                    HttpServletResponse
response)throws Exception {
System.out.println("we are in handleRequest()");
ModelAndView mav=null;
return mav;
```

```
}

}//class
```

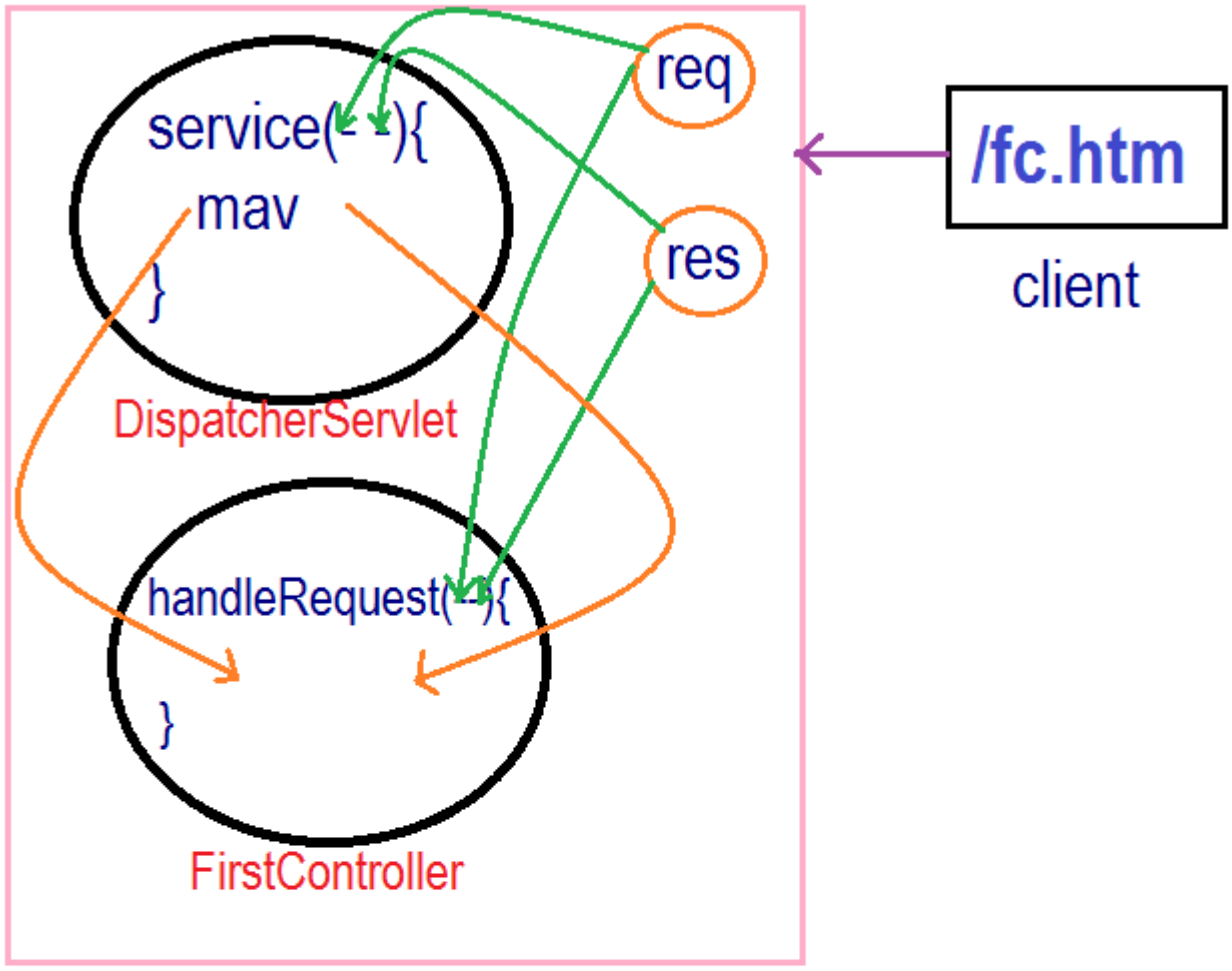
Configure Controller in spring bean configuration file

```
<bean name="/fc.htm"
class="com.mvc.FirstController"/>
```

spring-servlet.xml

The following steps are carried out when the client send the request to server, whose url-pattern ends with **.htm**

1. Server creates request and response object,if required server create DispatcherServlet object and call init() method
2. This method creates spring container object call to spring bean whose scope is singleton, DispatcherServlet service() calls handleRequest() it returns ModelAndView object.
3. The service() responds ModelAndView contains null value, DispatcherServlet service() method stop the execution of request, (i.e., because ModelAndView is holding null value)



controller > javabean > DAO > DB > objects >
model > controller > view (sessions)

In spring view object hold the name of view component, in spring we can use different views. (Ex : JSP or PDF or ExcelSheet are act as View)

```
spring model is an object it contains data, model
objects are used by view component to display the
output.

When ever a ModelAndView object is hand over into
DispatcherServlet it
call InternalResourceViewResolver , this bean forms
the JSP name by getting Prefix & Suffix.
```

It is the responsibility of DispatcherServlet to dispatch the request to the Jsp.

We can develop controller class based on AbstractController :

The Advantages of using AbstractController when compared with Controller interface

1. We can make sure that the controller work only for specific methods(GET or POST)
2. We can control the cache by specified expire date and time
3. Based on the availability of the session object we can execute the controller.

FirstController.java

```
package com.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

```
import
org.springframework.web.servlet.mvc.AbstractControl
ler;

public class FirstController extends
AbstractController {

public ModelAndView
handleRequestInternal(HttpServletRequest request,
               HttpServletResponse response)throws
Exception {
System.out.println("we are in
handleRequestInternal()");
ModelAndView mav=new ModelAndView("one");
return mav;
}

}//class
```

The following is internal code of AbstractController

```
package com.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.Controller;

public abstract class AbstractController implements
Controller{

public ModelAndView
handleRequest(HttpServletRequest request,
```

```
          HttpServletResponse response) throws
Exception {

     handleRequestInternal(request,response);
     //..............
 return null;
}

public abstract ModelAndView
handleRequestInternal(HttpServletRequest
                      request, HttpServletResponse
response);

}//class
```

- When the client send the request to above FirstController server will call **handleRequest()**method.
- As it is not available in FirstController server checks in AbstractController.
- AbstractController will call **handleRequestInternal()**.

controller has to work only POST request :

To achieve above requirement, we are use a method **setSupportedMethods(-)** in constructor as shown below.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.AbstractControl
ler;

public class SecondController extends
AbstractController{
```

```
public SecondController(){
 String[] methods=new String[]{"POST"};
 this.setSupportedMethods(methods);
}

public ModelAndView
handleRequestInternal(HttpServletRequest request,
          HttpServletResponse response) throws
Exception {
          //.................
 return null;
}

}//class
```

It is not recommended to use setSupportedMethods() to achieving by using spring bean configuration file as shown below

```
<bean name="/sc.htm"
class="com.mvc.SecondController">
 <property name="supportedMethods">
  <value>POST</value>
 </property>
</bean>
```

spring-servlet.xml

- All modern browsers try to catch the website data , the dis-advantage of this approach is if any modification is done in server , the browser will not send the request , it's not recommended to catch the data.
- In servlets, we are remove the browser cache by using cache Control Header to achieve the same in spring , we use a method **setCacheSeconds(-)**.

```
public class SecondController extends
AbstractController{
```

```
public SecondController(){
 this.setCacheSeconds(10);//disable cache memory
}

public ModelAndView
handleRequestInternal(HttpServletRequest request,
          HttpServletResponse response) throws
Exception {
          //.................
 return null;
}

}//class
```

When we specified **setCacheSeconds(-)** browser catch the data from 2 mins, if we don't want to allow browser to catch the data we specified the value **zero**.

If we want to execute the controller if the session object is available then we have to specify **requireSession** property.

```
<bean name="/sc.htm"
class="com.mvc.SecondController">
 <property name="requireSession">
  <value>true</value>
 </property>
</bean>
```

spring-servlet.xml

Reports :

- In every project we required reports, with out reports we will never implement the project reports gives the information about the health of the project.
- Generally the management uses reports to understand the business.

- Generally the reports are placed in side the PDF document or EXCEL.

There are so many Reporting Tools are available in the market some of them are...

1. Jasper
2. I-Reports
3. I-Text
4. FOP
5. BO Reports //non-java

Spring guys simplifies development of reports spring guys are given so many packages.

PDF :

The following an example to generate PDF report based on **i-Text**

```
package com.mvc;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.pdf.PdfPTable;
import com.lowagie.text.pdf.PdfWriter;

public class GeneratePDF{

public static void main(String[] args) throws
FileNotFoundException,

DocumentException{
 Document doc=new Document();
```

```
 PdfWriter.getInstance(doc, new
FileOutputStream("report.pdf"));
 doc.open();

 PdfPTable table=new PdfPTable(3);
 table.addCell("first");
 table.addCell("second");
 table.addCell("third");

 doc.add(table);
 doc.close();
 System.out.println("success");
}

}//class
standalone iText.jar
add jar file to IDE
iText in Action book download
iText in Action in PDF
```

The meaning of using jasper reports is create the object to a class and call the methods.

ExcelSheet :

- By using spring we want to develop ExcelSheet by using reports.
- To develop Excel report we can use a predefined class AbstractExcelView.
- This class internally uses POI API (apache) to generate Excel report.
- POI is an OpenSource API release on apache (add POI.jar file)

The following is an example of generating an Excel Sheet based on ExcelView

```
package com.mvc;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import
org.springframework.web.servlet.view.document.Abstr
actExcelView;

public class ExcelView extends AbstractExcelView{

@Override
protected void buildExcelDocument(Map mm,
                  HSSFWorkbook book,
                  HttpServletRequest request,
                  HttpServletResponse response)
                                    throws Exception
{

HSSFSheet sheet=book.createSheet("dummySheet");
HSSFRow row=sheet.createRow(0);

HSSFCell cell=row.createCell(0);
 cell.setCellValue("Eno");

 cell=row.createCell(1);
 cell.setCellValue("Name");

 cell.setCellValue(2);
 cell.setCellValue("Salary");
}
```

```
}//class
```

Spring Web MVC Framework

Agenda :

1. Introduction

First Spring Web Application :

Login.html

```
 <body>
 <form action="login.spring" >
 User Name : <input type="text" name="userName">
<br>
 Password: <input type="password" name="password">
<br>
 <input type="submit" value="Login">
 </form>
</body>
```

LoginController.java

```
package com.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.Controller;
```

```
public class LoginController implements Controller
{

LoginModel loginModel;
public void setLoginModel(LoginModel lm){
 loginModel=lm;
}

public ModelAndView
handleRequest(HttpServletRequest request,
                HttpServletResponse response)
throws Exception {
String userName=request.getParameter("userName");
String password=request.getParameter("password");
System.out.println("userName : "+userName);
System.out.println("password : "+password);
String type =
loginModel.getValidate(userName,password);

System.out.println("type : "+type);

if(type==null)
     return new ModelAndView("/login.html");
else if(type.equals("admin"))
     return new ModelAndView("/pages/admin.jsp");
else
     return new ModelAndView("/pages/user.jsp");

} //handleRequest
}
```

LoginModel.java

```
package com.controller;
```

```
import
org.springframework.dao.EmptyResultDataAccessExcept
ion;
import org.springframework.jdbc.core.JdbcTemplate;

public class LoginModel {

private JdbcTemplate jdbcTemplate;

public LoginModel(JdbcTemplate jdbcTemplate) {
this.jdbcTemplate=jdbcTemplate;
}

public String getValidate(String userName, String
password) {

 String sql="select type from UserDetails where
username=\'"
               + userName+ "\' and
password=\'"+password+ "\' ";

 try{
     return jdbcTemplate.queryForObject(sql,
String.class);
 }catch (EmptyResultDataAccessException e) {
      return null;
}

} //getValidate()

}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
     xmlns="http://java.sun.com/xml/ns/javaee"
```

```
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns
/javaee
     http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">

<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

ds-servlet.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">
```

```
<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
 <value>oracle.jdbc.driver.OracleDriver</value>
</property>
<property name="url">
 <value>jdbc:oracle:thin:@localhost:1521:xe</value>
</property>
<property name="username">
 <value>lms</value>
</property>
<property name="password">
 <value>scott</value>
</property>
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="loginModel"
class="com.controller.LoginModel">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>

<bean id="loginController"
class="com.controller.LoginController">
 <property name="loginModel">
  <ref local="loginModel"/>
```

```
 </property>
</bean>

<bean id="myUrlMapping"

class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
 <property name="mappings">
  <props>
    <prop
key="/login.spring">loginController</prop>
  </props>
 </property>
</bean>

</beans>
```

adminHome.jsp

```
Wel come to Admin page <br/>

User Name : <%= request.getParameter("userName") %>
```

userHome.jsp

```
Wel come to User page <br/>

User Name : <%= request.getParameter("userName") %>
```

Data Base Code snoppet :

```
SQL> create table userdetails (username
varchar2(15),
                  password varchar2(15),type
varchar2(15));

SQL> insert into userdetails
values('ashok','scott','admin');
```

Under Standing DispatcherServlet and Request Processing WorkFlow

Agenda :

I. DispatcherServlet
   - Initialization Parameters of DispatcherServlet
   - DispatcherServlet Initialization Stage
   - Spring WEB MVC Framework special objects

II. Spring WEB MVC Request Processing Work flow (Phases : 8)
   1. Prepare the request context
   2. Locate The Handler
      - HandlerMappings
        i. BeanNameUrlHandlerMapping
        ii. SimpleUrlHandlerMapping
        iii. ControllerClassNameHandlerMapping
        iv. CommonsPathMapHandlerMapping
      - Configuring multiple handler mappings
   3. Execute Interceptors preHandle methods
      - Using HandlerInterceptor
   4. Invoke Handler
      - HandlerAdapter
        i. SimpleControllerHandlerAdapter
        ii. ThrowawayControllerHandlerAdapter
        iii. HttpRequestHandlerAdapter
        iv. SimpleServletHandlerAdapter
      - ModelAndView
   5. Execute Interceptors postHandle methods
   6. Handle Exceptions
   7. Render the View

- The 'Resolve View Name' process

8. Execute Interceptors afterCompletion methods

DispatcherServlet :

- The DispatcherServlet of Spring Web MVC framework is an implementation of FrontController and is a Java Servlet component. i.e., it is a servlet front for spring Web Mvc application.
- DispathcherServlet is the FrontController classes that receives all incoming HTTP client request for the spring Web mvc appliaction.
- DispatcherServlet is responsible for initializing spring web mvc framework for our application, and is a servlet implemented as a sub-type of HttpServlet just like any other Servlet.
- DispatcherServlet also required to be configured in our web-application like any other Servlet i.e., web application deployment descriptor(web.xml)

DispatcherServlet

```
<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>
</servlet>
```

We can configure the initialization parameters to alter the behavior of the DispatcherServlet with respect to locating the spring Beans XML configuration files and initializing the application context.

Initialization Parameters of DispatcherServlet :

1. **contextClass :**
2. **namespace :**
3. **contextConfigLocation :**
4. **publishContext :**
5. **publishEvents :**
6. **detectAllHandlerMappings :**
7. **detectAllHandlerAdapters :**
8. **detectAllHandlerExceptionResolvers :**
9. **detectAllViewResolvers :**
10. **cleanupAfterInclude :**

```
<servlet>

<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>

<init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/applicationContext.xml
  /WEB-INF/myconfigs/applicationControllers.xml
 </param-value>
</init-param>

<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>
```

- All the request for this context whose servletPath ends with **.spring** extension are displayed by the web container to the DispatcherServlet.
- It is not mandatory to configure only .spring, instead of configure any other extension.

```
Spring Web MVC framework does not force to use
specific extensions or patterns in URL's;
Spring frame work is not bonded to match the handlers
only based on URLs.Alternatively we can define a
mapping based on parameters or HTTPSession state
etc.,
This future is not supported by most of Web MVC
framework which includes Struts.
```

DispatcherServlet Initialization Stage :

- While initializing the DispatcherServlet it creates WebApplicationContext implementing class instance, either user configured custom context class (or) the default XmlWebApplicationContext.
- The WebApplicationContext is responsible to locate the Spring Beans XML configuration file, then read, validate the configurations and load the details into configuration objects.
- The DispatcherServlet uses the WebApplicationContext for accessing various framework objects that are used to execute the WebApplicationContext.

Spring WEB MVC Framework special objects :

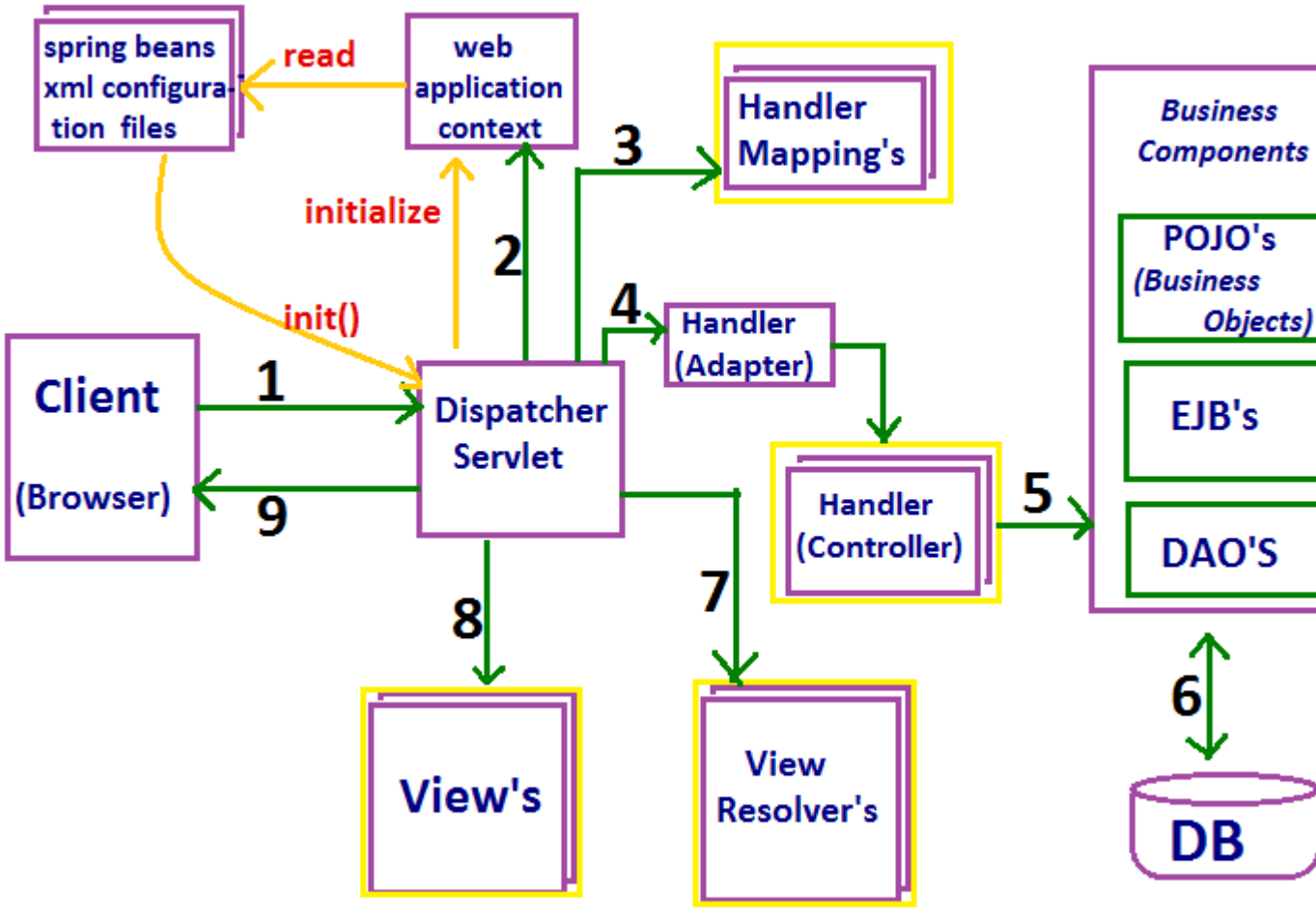
1. MultipartResolver
2. LocaleResolver
3. ThemeResolver
4. HandlerMapping

5. HandlerAdapter
6. HandlerExceptionResolver
7. RequestToViewNameTranslator
8. ViewResolver

- All the special frame work objects initialized by the DispatcherServlet using the WebApplicationContext and based on the detectAllXXX initialization parameters configured to this servlet.
- Once if all these objects are initialized successfully, the DispatcherServlet instance is put into service, which provides an entry point to serve the client requests using the special frame work objects.

Spring WEB MVC Request Processing Work flow :

- According to spring MVC , DispatcherServlet is the front controller for the spring Web MVC application, providing a centralized access for various requests to the application and collaborating with various other objects to complete the request handling and present the response to client.
- DispatcherServlet uses the WebApplicationContext object to locate the various objects configured in the Spring Bean XML configuration file.
- The WebApplicationContext is instantiated and initialized as a part of the DispatcherServlet's initialization process.
- The WebApplicationContext object is responsible to locate the spring beans XML configuration file and load its details to prepare the context for handling the requests.



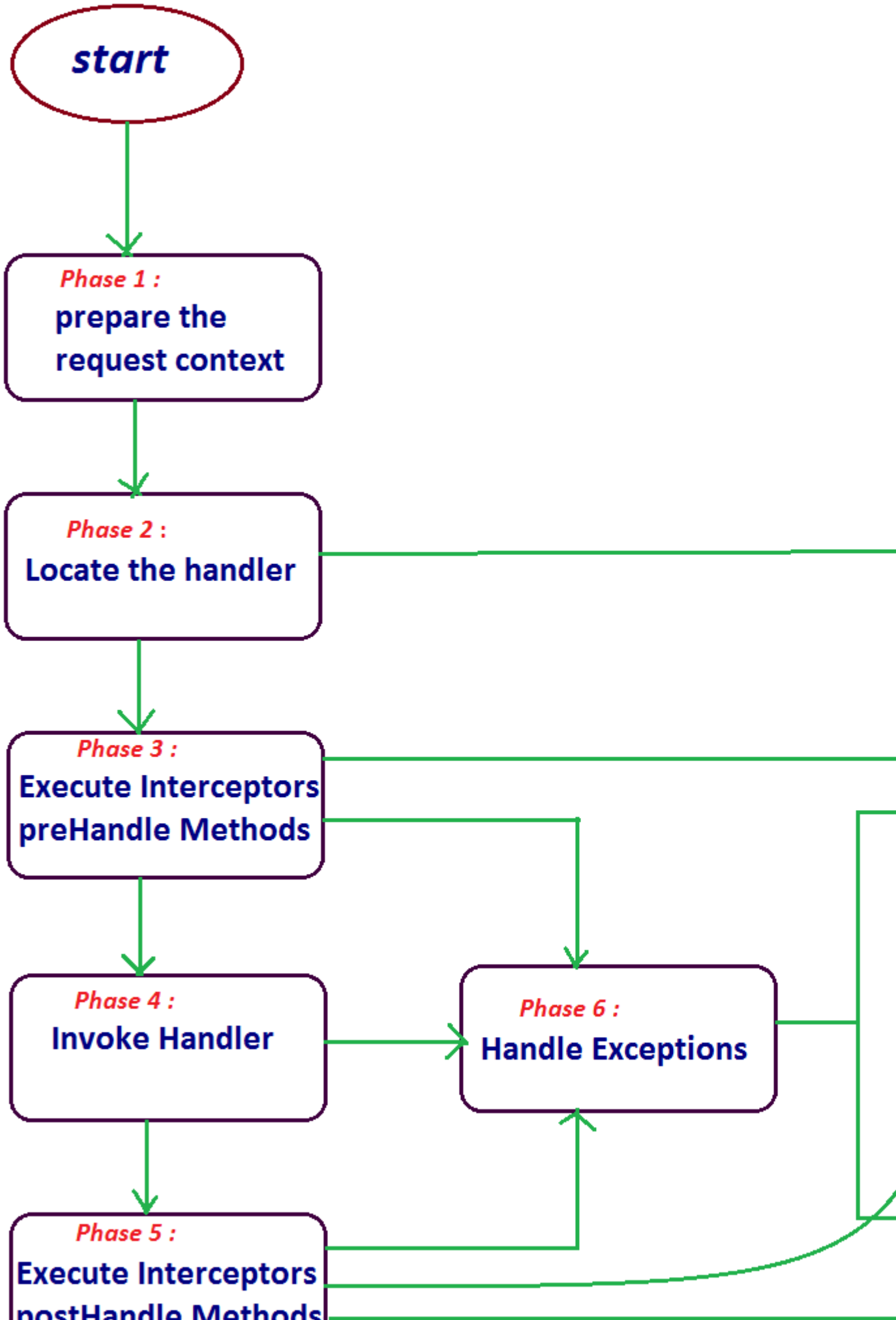
Now, when a client request is given to DispatcherServlet, it performs the following operations:

Types of Phases :

1. Prepare the request context
2. Locate The Handler
3. Execute Interceptors preHandle methods
4. Invoke Handler
5. Execute Interceptors postHandle methods

6. Handle Exceptions
7. Render the View
8. Execute Interceptors afterCompletion methods

Karrasankar158@gmail.com

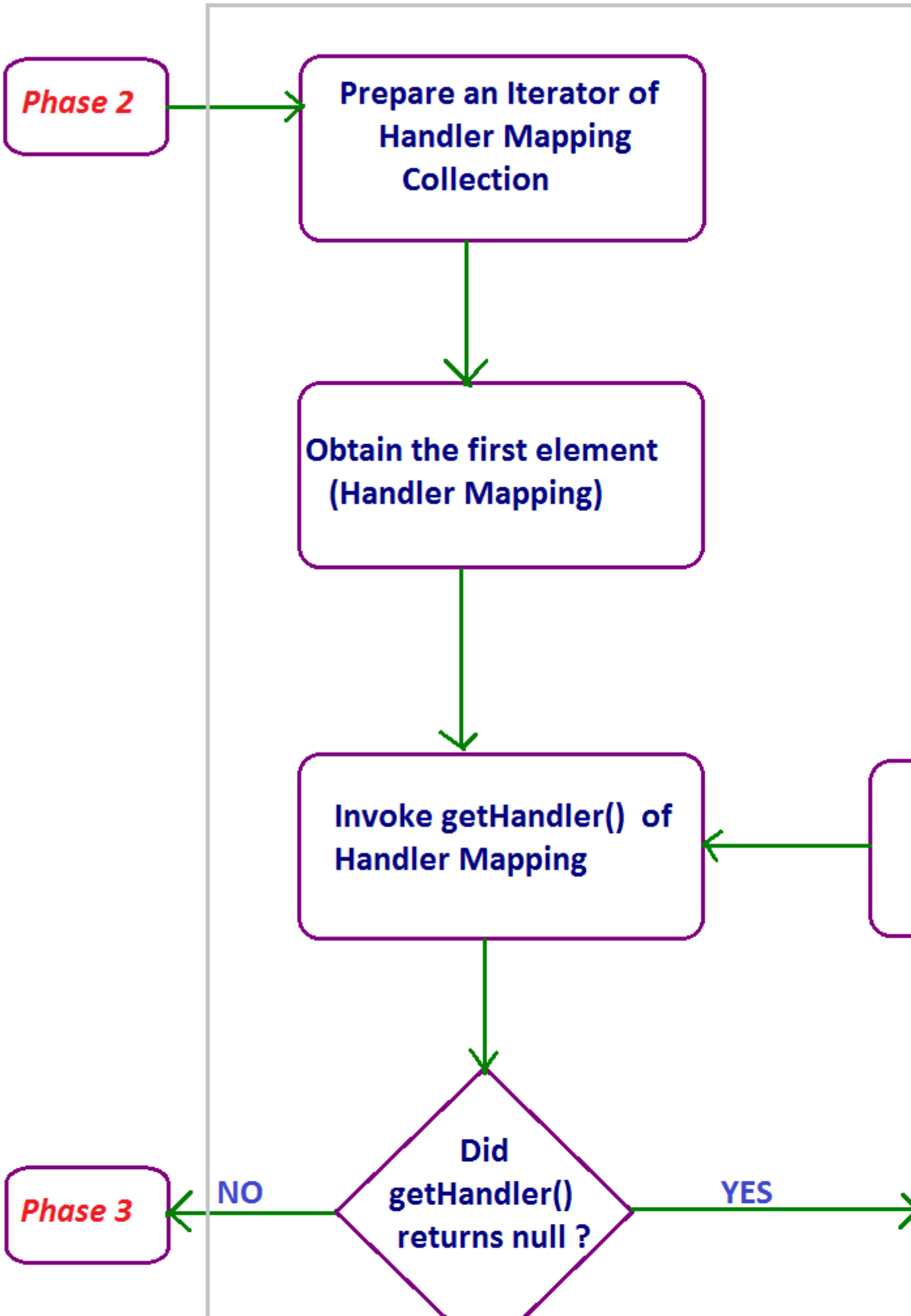


Phase 1 : Prepare the request context

1. DispatcherServlet prepare the request context, by setting the framework objects into the request scope.
2. Here framework objects are WebApplicationContext, LocaleResolver, ThemeResolver and ThemeSource.
3. These objects are set into the request scope to make them available to handler and view objects.
   so that the handler or view objects can use these objects to communicate with the framework and collect runtime details.
4. Apart from preparing and setting the framework objects into the request scope, DispatcherServlet resolves the request using MultipartResolver , so that if the request contains contains multi part data , then it wraps the request in a MultipartHttpRequest type object.
5. In case , if there is any problem in this process the request processing is terminated by throwing an Exception.
6. Once if this process is done successfully the request process workflow continues to the next phase, i.e., Locate the Handler.

Phase 2 : Locate The Handler

Karrasankar158@gmail.com



1. After preparing the request context , the DispatcherServlet locates handler that can handle this request.
2. The DispatcherServlet uses the registered HandlerMapping's and collects the HandlerExecutionChain object.
3. The HandlerExecutionChain object encapsulates the HandlerIntercepter's and the Handler object (i.e., controller).
4. Preparing an Iterator object of the Collection, storing the HandlerMapping objects.
5. This Collection object is created while the DispatcherServlet is being initialized, i.e., in its initialization phase.
6. Thereafter, the first element is received from the iterator. This can't be an empty collection since if there is no HandlerMapping configured in the context the DispatcherServlet uses BeanNameUrlHandlerMapping as default HandlerMapping.
7. Invoke the getHandler() of the current HandlerMapping object in the iteration.
    - If getHandler() returns null, get the next element is available the next HandlerMapping. Otherwise set response error code(i.e., HttpServletResponse_SC_NOT_FOUND) and terminate request processing.
    - If getHandler() returns a valid HandlerExecutionChain object reference then the control delegates to next phase.(i.e., Execute Interceptors preHandle methods)

The HandlerMappings :

1. The HandlerMapping is responsible for mapping the incoming request to the handler that can handle the request.

2. As discussed in the above section, When the DispatcherServlet receives the request it delegates the request to the HandlerMapping, which identifies the appropriate HandlerExecutionChain that can handle the request.
3. The Spring Web MVC framework provides customizable navigation strategies. Spring provides built-in navigation strategies as determining the handler based on the request URL mapping , which is again based on the bean name.
4. A part from the built-in navigation strategies, Spring allows system-specific built strategy, which can be done by writing a class implementing the HandlerMapping interface.

The Spring built-in HandlerMapping implementations are :

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

BeanNameUrlHandlerMapping

- The **org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping** is one of the implementation of HandlerMapping interface.
- This implementation defines the navigation strategy that maps the request URL's servlet-path to the bean names.
- This handler mapping strategy is very simple but powerful.

```
The BeanNameUrlHandlerMapping is the default handler
mapping when no handler mapping is configured in the
application context.
```

The BeanNameUrlHandlerMapping explicitly configure in 2 cases :

1. When we want to configure multiple handler mapping
2. When we want to configure handler interceptors

ds-servlet.xml

```
 <beans>

 <bean id="handlerMapping"

class="org.springframework.web.servlet.handler.Bean
NameUrlHandlerMapping"/>

<bean name="/addEmployee.spring"
class="com.spring.AddEmployeeController">
 <!-- set the dependencies -->
</bean>

<bean name="/removeEmployee.spring"
class="com.spring.RemoveEmployeeController">
 <!-- set the dependencies -->
</bean>


</beans>
```

SimpleUrlHandlerMapping

- The **org.springframework.web.servlet.handler.SimpleUrl HandlerMapping** is one of the implementation of HandlerMapping interface.
- This implementation defines the navigation strategy that maps the request URL's servlet-path to the **mapping configured**. i.e., It locates the handler(controller) by

matching the request URL's servlet-path with the key of the given properties or Map.

- The SimpleUrlHandlerMapping supports the configure 2 types :
  1. bean names
  2. bean instances

```
<beans>

<bean id="addEmp"
class="com.spring.AddEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="removeEmp"
class="com.spring.RemoveEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="searchEmp"
class="com.spring.SearchEmployeeController">
 <!-- configure dependencies -->
</bean>


<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
 <property name="mappings">
  <props>
    <prop key="/addEmployee.spring">addEmp</prop>
    <prop
key="/removeEmployee.spring">removeEmp</prop>
    <prop
key="/searchEmployee.spring">searchEmp</prop>
  </props>
```

```
 </property>
</bean>

</beans>
```

This configuration is suitable for configuring non-singleton beans

OR

```
<beans>

<bean id="addEmp"
class="com.spring.AddEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="removeEmp"
class="com.spring.RemoveEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="searchEmp"
class="com.spring.SearchEmployeeController">
 <!-- configure dependencies -->
</bean>


<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
 <property name="mappings">
  <map>
    <entry key="/addEmployee.spring">
     <ref local="addEmp"/>
    </entry>

    <entry key="/removeEmployee.spring">
```

```
     <ref local="removeEmp"/>
    </entry>

    <entry key="/searchEmployee.spring">
     <ref local="searchEmp"/>
    </entry>
  </map>
 </property>
</bean>

</beans>
```

This configuration is suitable for configuring singleton beans

ControllerClassNameHandlerMapping

- The **org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping** is one of the implementation of HandlerMapping interface.
- This handler mapping implementation is newly introduced in spring 2.0
- The ControllerClassNameHandlerMapping follows a simple convention for generating URL path mappings.
- The convention for simple Controller implementations is to take the short names of the controller class.

Ex : If the controller class name is **com.emp.AddEmployeeController** then the path is **addemployee***

```
<beans>

<bean id="handlerMapping"
class="org.springframework.web.servlet.mvc.

support.ControllerClassNameHandlerMapping"/>
```

```
<bean id="addEmp"
class="com.spring.AddEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="removeEmp"
class="com.spring.RemoveEmployeeController">
 <!-- configure dependencies -->
</bean>

<bean id="searchEmp"
class="com.spring.SearchEmployeeController">
 <!-- configure dependencies -->
</bean>

</beans>
```

CommonsPathMapHandlerMapping

- The **org.springframework.web.servlet.handler.CommonsPathMapHandlerMapping** is one of the implementation of **HandlerMapping** interface.
- The **org.springframework.web.servlet.handler.CommonsPathMapHandlerMapping** is designed to recognize Commons attributes meta data attributes of type **PathMap** defined in the application controller and automatically wires them in to the current DiapatcherServlet's Web-application context.
- To use this HandlerMapping the controller class must have a class level metadata of the form **@org.springframework.web.servlet.handler.commonsattribures.PathMap("/mypath.spring").**
- We can configure multiple path maps for a single controller.

```
@org.springframework.web.servlet.handler.commonsatt
ribures.PathMap(

"/mypath.spring")

public class FirstController implements Controller
{

public ModelAndView
handleRequest(HttpServletRequest request,
                    HttpServletResponse
response)throws Exception {
System.out.println("we are in handleRequest()");
ModelAndView mav=null;
return mav;
}
```

Note : To use commons attributes path mapping , we must compile application classes with Commons Attributes, and run the Commons attributes indexer tool on the application classes, which must be in a Jar rather than in WEB-INF/classes as an individual classes.

Configuring multiple handler mappings :

- We can configure multiple handler mappings in an application context.
- In such a case we need to configure an additional property 'order' that takes 'int' value on each of the handler mapping.

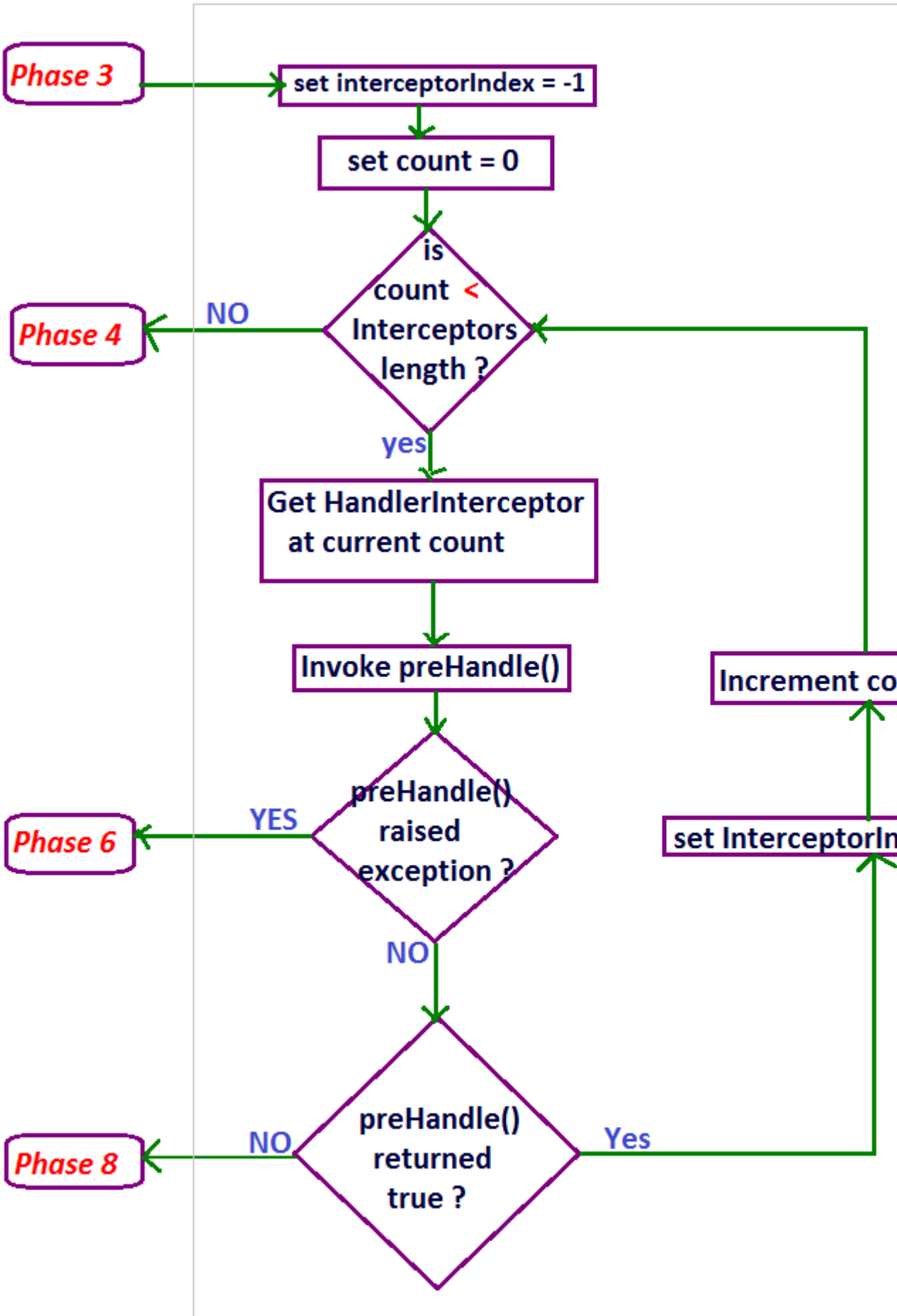
Phase 3 : Execute Interceptors preHandle methods

1. After successfully locating the HandlerExcecutionChain the DispatcherServlet executes the HandlerInterceptors

described by the HandlerExecutionChain returned by the HandlerMapping.

2. The HandlerInterceprors gives us an opportunity to add common pre and post-processing behavior without needing to modify each handler implementation.
3. The HandlerInterceptors is basically similar to a Servlet Filter(2.3v) .

Karrasankar158@gmail.com



4. The HandlerInterceptors can be used for implementing pre-processing aspects, for example , for authorization checks, or common handler behavior like locale or theme changes.

Using HandlerInterceptor

Using the HandlerInterceptor includes the following 2 steps :

1. Write an HandlerInterceptor implementation
2. Configure the interceptor

Write an HandlerInterceptor implementation :

- The HandlerInterceptor can be used for implementing pre-processing aspects, for example, for authorization checks, or common handler behavior like locale or theme changes.
- The Spring framework includes built-in HandlerInterceptor's implementing the most common pre-processing concerns like locale and theme change.

The following are the built-in HandlerInterceptor implementations

- org.springframework.web.servlet.i18n.LocaleChangeInterceptor
- org.springframework.web.servlet.theme.ThemeChangeInterceptor

Apart from the built-in HandlerInterceptors we can write HandlerInterceptor implementations encapsulating the custom pre- and post-processing logics.

```
package com.spring;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import
org.springframework.web.servlet.HandlerInterceptor;
import
org.springframework.web.servlet.ModelAndView;

public class MyHandlerInterceptor implements
HandlerInterceptor {

public boolean preHandle(HttpServletRequest
request,
               HttpServletResponse response,
                Object handler) throws Exception {

System.out.println("preHandle() executes at phase
3");

/*
Do some preprocessing like check the security etc

return boolean value accordingly,
if we want to continue the interceptor chain
returns true
*/

return false;
}

public void postHandle(HttpServletRequest request,
         HttpServletResponse response,
          Object handler, ModelAndView mav) throws
Exception {

System.out.println("afterCompletion() executes at
phase 5");
      //Do some postprocessing
}
```

```
public void afterCompletion(HttpServletRequest
request,
          HttpServletResponse response,
          Object handler, Exception ex)throws
Exception {

System.out.println("afterCompletion() executes at
phase 5");
        //Do finalizations
}

}
```

Configuring the interpretor :

```
<beans>
<bean id="myInterceptor"
class="com.spring.MyHandlerInterceptor"/>

<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.Bean
NameUrlHandlerMapping">
 <property name="interceptors">
  <list>
   <ref local="myInterceptor"/>
  </list>
 </property>
</bean>
</beans>
```

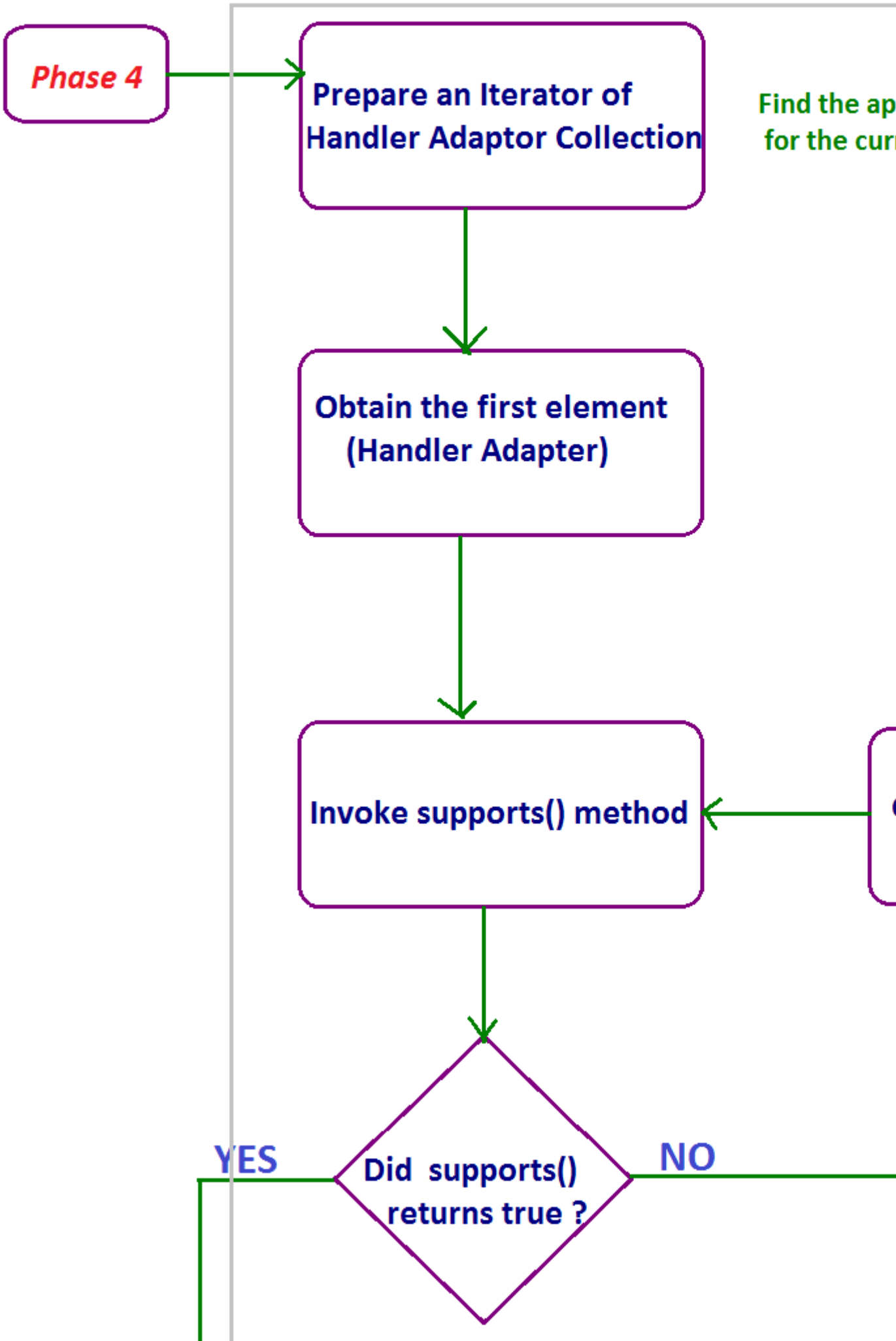
Phase 4 : Invoke Handler

1. In this phase of Spring web MVC request processing workflow the DispatcherServlet delegates the request to the handler that is located by the HandlerMapping in Phase 2.
2. DispatcherServlet uses HandlerAdapter to delegate the request to the handler located to handle this request.

- **step 1 : Prepare a iterator of HandleAdapter collection :**
  i. In this Phase workflow starts with preparing an iterator of the collection representing the HandlerAdapter objects configured in the application.
  ii. The handlerAdapters collection is initialized in the initialization phase of the DispatcherServlet where it finds all HandlerAdapters in the ApplicationContext.
  iii. If no HandlerAdapter beans are defined in the application then the default is considered as SimpleControllerHandlerAdapter. Thus the handlerAdapters collection contains at least one element.
- **step 2 : Get HandlerAdapter :**
  i. In this step the workflow obtains the next element from the iterator in step 1 and casts it into HandlerAdapter type reference.
- **step 3 : Find is HandlerAdapter compatible :**
  i. After getting the HandlerAdapter of the current iteration the workflow continues with finding whether this HandlerAdapter is suitable for the handler located in phase 2.
  ii. This is done by invoking supports() method on the HandlerAdapter.
  iii. If it returns "true" then it indicates that this HandlerAdapter supports the handler. In such a case the workflow continues to the next step.
  iv. If the supports() returns "false" then the work flow continues finding whether there is a next element in the handlerAdapter Collection , if found then it moves to step 2. If not then ServletException is thrown, delegating the workflow to phase 6.
- **step 4 : Execute handler :**

i. After successfully locating the HandlerAdapter that supports the handler that is located in phase 2, the workflow proceeds to invoke handle() method of HandlerAdapter which further delegates the request to the handler(may be controller).
ii. If the handle() method throws any exception then the workflow proceeds to phase 6 , if not it proceeds to phase 5 after collecting the ModelAndView object reference returned by the handle() method.

Karrasankar158@gmail.com



The HandlerAdapter :

1. The HandlerAdapter implementation takes the responsibility of identifying the type of handler and invokes its appropriate methods.
2. The use of HandlerAdapter facilitates us to use Plain Old Java Objects (POJOs) with any method encapsulating the handler behavior , as a handler.
3. Spring provides the built-in HandlerAdapter implementations supporting different types of handlers to work with.
4. The various types of handlers supported by the Spring built-in HandlerAdapter are controller types, HttpRequestHandler types, Servlet types, and ThrowawayController types.
5. The org.springframework.web.servlet.HandlerAdapter interface declares three methods (supports(), handle(), getLastModified()) that have to be implemented by the HandlerAdapter implementations to help DiapatcherServlet in delegating the request to the handlers.

Spring built-in HandlerAdapter implementations :

1. SimpleControllerHandlerAdapter
2. ThrowawayControllerHandlerAdapter
3. HttpRequestHandlerAdapter
4. SimpleServletHandlerAdapter

Note :

- By default only the SimpleControllerHandlerAdapter and ThrowawayControllerHandlerAdapter handler adapters are available to the DispatcherServlet.

- Other handler adapters need to be explicitly configured, as normal as other beans in the context Spring Beans XML configuration file of DispatcherServlet.
- Even though Spring supports implementing the custom handler adapters that enables handling the request using user defined type of handlers (without making them depend on the Spring API), however it is recommended to use the Spring Controller infrastructure to implement the handler.
- The custom handler adapter option is given to support some situation where we want to use the existing handlers without rewriting them as spring defined controllers.

The HandlerAdapter uses handler to handle the request and results returning the ModelAndView object.

```
 package com.spring;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

public class MyHandlerAdapter implements
HandlerAdapter {

public boolean supports(Object handler) {
return (handler instanceof com.spring.MyHandler);
}

public ModelAndView handle(HttpServletRequest
request,
```

```
          HttpServletResponse response, Object
handler) throws Exception {

MyHandler myHandler=(MyHandler) handler;
String view=myHandler.process(request,response);
Map model=(Map)request.getAttribute("model");

if(view==null)
     return null;
else if(model==null)
     return new ModelAndView(view);
else
     return new ModelAndView(view,model);

}

public long getLastModified(HttpServletRequest
request, Object handler) {

return -1;
}

}
package com.spring;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface MyHandler {

String process(HttpServletRequest request,
     HttpServletResponse response) throws
Exception;
}
```

ModelAndView :

- The ModelAndView is a value object designed to hold model and view making it possible for a handler to return both model and view in a single value.
- The ModelAndView object represents a model and view specified by the handler, which is resolved by the DispatcherServlet using the special framework objects as ViewResolver and View.
- The view is an object that can describe a view name in the form of String which will be resolved by a ViewResolver object to locate View object, alternatively, a View object directly.
- The Model is a Map, enabling to specify multiple objects.

Phase 5 : Execute Interceptors postHandle methods

- HandlerInterceptor gives us an opportunity to add common pre- and post-processing behavior without needing to modify each handler implementation.
- The post-processing operations like changing the logical view name in the ModelAndView based on some inputs/output to support different types of views.

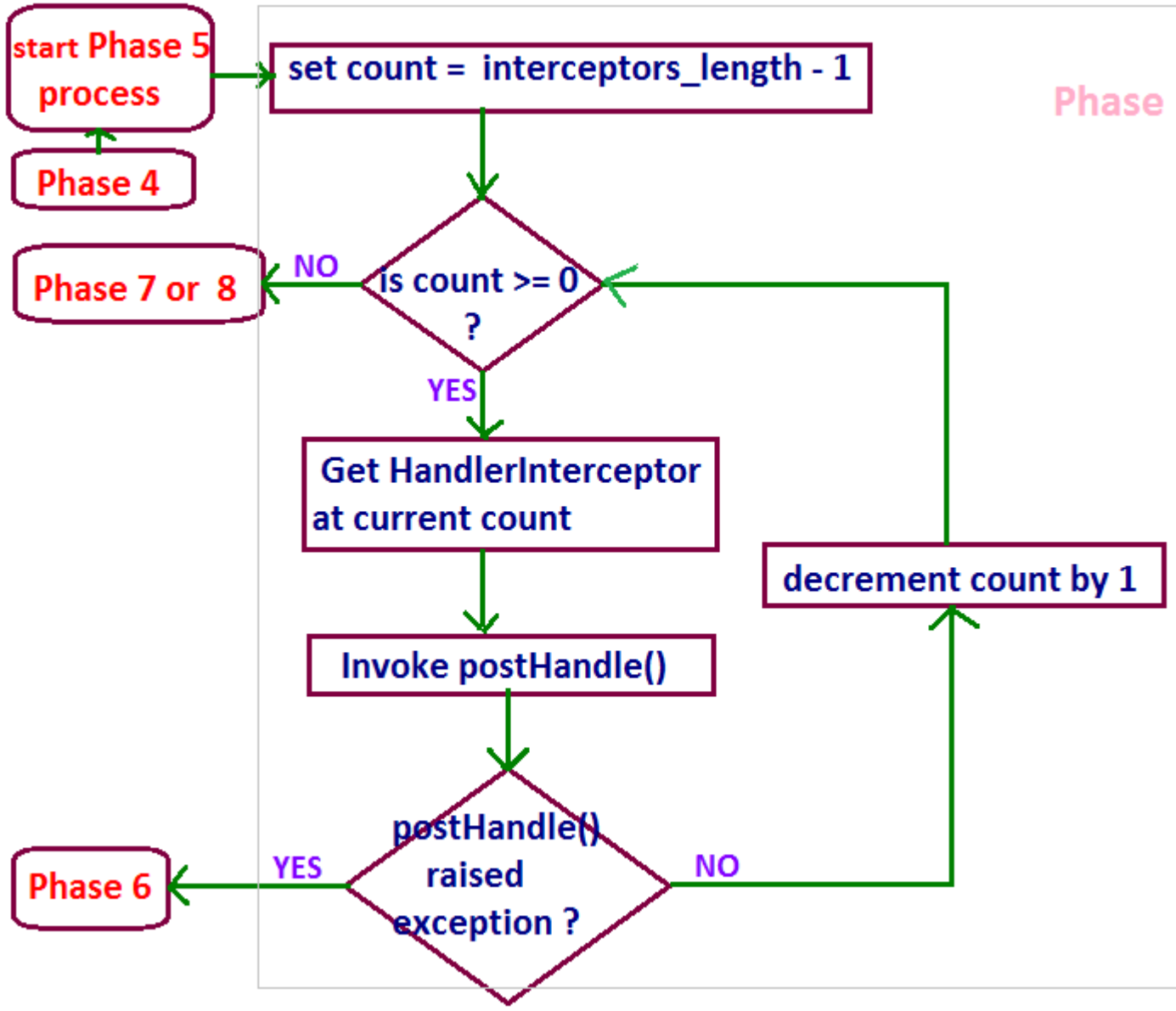
Step 1 : Prepare the Counter :

- In this phase workflow starting with setting the count to one minus the interceptor's length so that the postHandle method can be invoked on the interceptor's in the reverse order.
- If the interceptors length is 0(zero) then the workflow proceeds to Phase 7 or 8 based on whether the the handler in Phase 4 has returned a valid ModelAndView object reference or null.

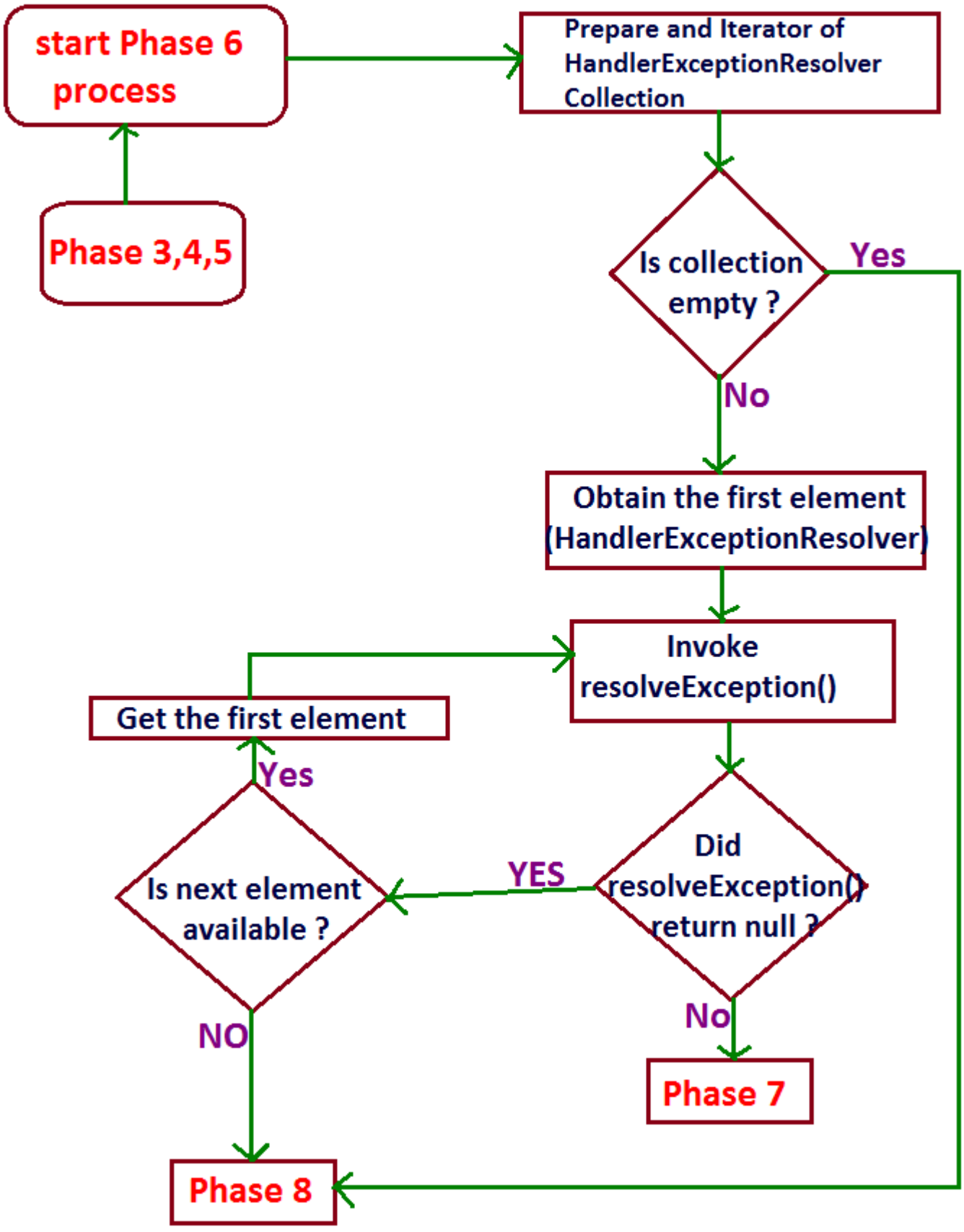
- If there are any interceptors configured then the workflow proceeds to the next step of this phase.

Step 2 : Invoke postHandle() method :

- Here it obtains the HandlerInterceptor at the current count and uses it to invoke postHandle() method.
- If the postHandle() method throws any exception then the worlflow proceeds to Phase 6 to handle the exception.
- If the postHandle() method execution is successful (i.e., normal termination) then count is decremented by 1, if the count is grater than or equal to 0(zero) then repeat step2 (i.e., this step) once again.
- If the count is less than 0(zero) then the workflow proceeds to Phase 7 or Phase 8 based on the result of Phase 4.
- That is, if the handler in Phase 4 has returned a valid ModelAndView object reference then workflow proceeds to Phase 7 to render the view, If the handler had returned null then the workflow proceeds to Phase 8 considering that the handler had prepared the response.



Phase 6 : Handle Exceptions


start Phase 6 process
Prepare and Iterator of HandlerExceptionResolver Collection
Phase 3,4,5
Is collection empty ?
Yes
No
Obtain the first element (HandlerExceptionResolver)
Invoke resolveException()
Get the first element
Yes
Is next element available ?
YES
Did resolveException() return null ?
NO
No
Phase 7
Phase 8

- This Phase of the Spring Web MVC request processing flow is executed only when there is any exception raised while executing interceptors preHandle() or postHandle() methods or the handler.
- DispatcherServlet uses HandlerExceptionResolver to handle the exceptions thrown by the HandlerInterceptor or handler.

```
Note : Spring does not take a default type for
HandlerExceptionResolver
```

Step 1 : Prepare an iterator of exception resolvers :

- Preparing an iterator of the Collection representing HandlerExceptionResolver objects configured in the application.
- The handlerExceptionResolvers Colletion is initialized in the initialization phase of the DispatcherServlet where it finds all HandlerExceptionResolver's in the ApplicationContext.
- If no HandlerExceptionResolver beans are defined in the application then this collection will be empty.
- If the handlerExceptionResolvers collection is empty then it considers that there are no exception handlers for the application, thus the workflow proceeds to Phase 8 which then ends the request processing by throwing the same exception.
- If the collection is not empty then the workflow continues to the next step of this phase.

Step 2 : Get HandlerExceptionResolver :

In this step the work flow obtains the next element from the iterator prepared in step 1, and casts into HandlerExceptionResolver type reference.

Step 3 : Invoke resolveException() method :

- After getting the HandlerExceptionResolver of the current iteration the resolveException() method is invoked to handle the exception.
- If the resolveException() method returns a valid ModelAndView object reference then the workflow proceeds to phase 7 for rendering a response.
- If the resolveException() method returns null then it finds whether the next element is available in the iterator.
- If so, then the work flow proceeds to step2 of this phase. If the next element is not available, then the workflow proceeds to phase 8 which then ends the request processing by throwing the same exception.

```
<beans>
<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.Simp
leMappingExceptionResolver">
 <property name="exceptionMappings">
  <props>
   <prop
key="org.springframework.dao.EmptyResultDataAccessE
xception">
    /sqlError.html
   </prop>

   <prop
key="org.springframework.web.HttpRequestMethodNotSu
pportedException">
    /myError.html
```

```
   </prop>
  </props>
 </property>
</bean>
</beans>
```

login.html

```
<body>
 <form action="/login.spring" > <br> // / is
optional
 User Name : <input type="text" name="userName">
<br>
 Password: <input type="password" name="password">
<br>
 <input type="submit" value="Login">
 </form>
</body>
</html>
```

LoginController.java

```
package com.spring;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.Controller;


public class LoginController implements Controller
{

LoginModel loginModel;
public void setLoginModel(LoginModel lm){
```

```
 loginModel=lm;
}

public ModelAndView
handleRequest(HttpServletRequest request,
                HttpServletResponse response)
throws Exception {
String userName=request.getParameter("userName");
String password=request.getParameter("password");
System.out.println("userName : "+userName);
System.out.println("password : "+password);
String type =
loginModel.getValidate(userName,password);

System.out.println("type : "+type);

if(type==null)
     //return new ModelAndView("/login.html");
     throw new MyException("User details are not
valid");
else if(type.equals("admin"))
     return new ModelAndView("/pages/admin.jsp");
else
     return new ModelAndView("/pages/user.jsp");

} //handleRequest
}
```

LoginModel.java

```
package com.spring;

import
org.springframework.dao.EmptyResultDataAccessExcept
ion;
import org.springframework.jdbc.core.JdbcTemplate;

public class LoginModel {
```

```
private JdbcTemplate jdbcTemplate;

public LoginModel(JdbcTemplate jdbcTemplate) {
this.jdbcTemplate=jdbcTemplate;
}

public String getValidate(String userName, String
password) {

 String sql="select type from UserDetails where
username=\'"
               + userName+ "\' and
password=\'"+password+ "\' ";

 try{
     return jdbcTemplate.queryForObject(sql,
String.class);
 }catch (EmptyResultDataAccessException e) {
      return null;
}

} //getValidate()

}
```

admin.jsp

```
Wel come to Admin page <br/>

User Name : <%= request.getParameter("userName") %>
```

user.jsp

```
Wel come to User page <br/>

User Name : <%= request.getParameter("userName") %>
```

myError.html

```
<body>
A simple test error page <br>
User details are not valid
```

```
</body>
```

ds-servlet.xml

```
<beans>

<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
 <value>oracle.jdbc.driver.OracleDriver</value>
</property>
<property name="url">
 <value>jdbc:oracle:thin:@localhost:1521:xe</value>
</property>
<property name="username">
 <value>lms</value>
</property>
<property name="password">
 <value>scott</value>
</property>
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="loginModel"
class="com.spring.LoginModel">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>
```

```
<bean id="loginController"
class="com.spring.LoginController">
 <property name="loginModel">
  <ref local="loginModel"/>
 </property>
</bean>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
<property name="mappings">
 <props>
  <prop key="/login.spring">loginController</prop>
 </props>
</property>
</bean>


<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.Simp
leMappingExceptionResolver">
 <property name="exceptionMappings">
  <props>
   <prop key="com.spring.MyException">
    /myError.html
   </prop>
  </props>
 </property>
</bean>

</beans>
```

web.xml

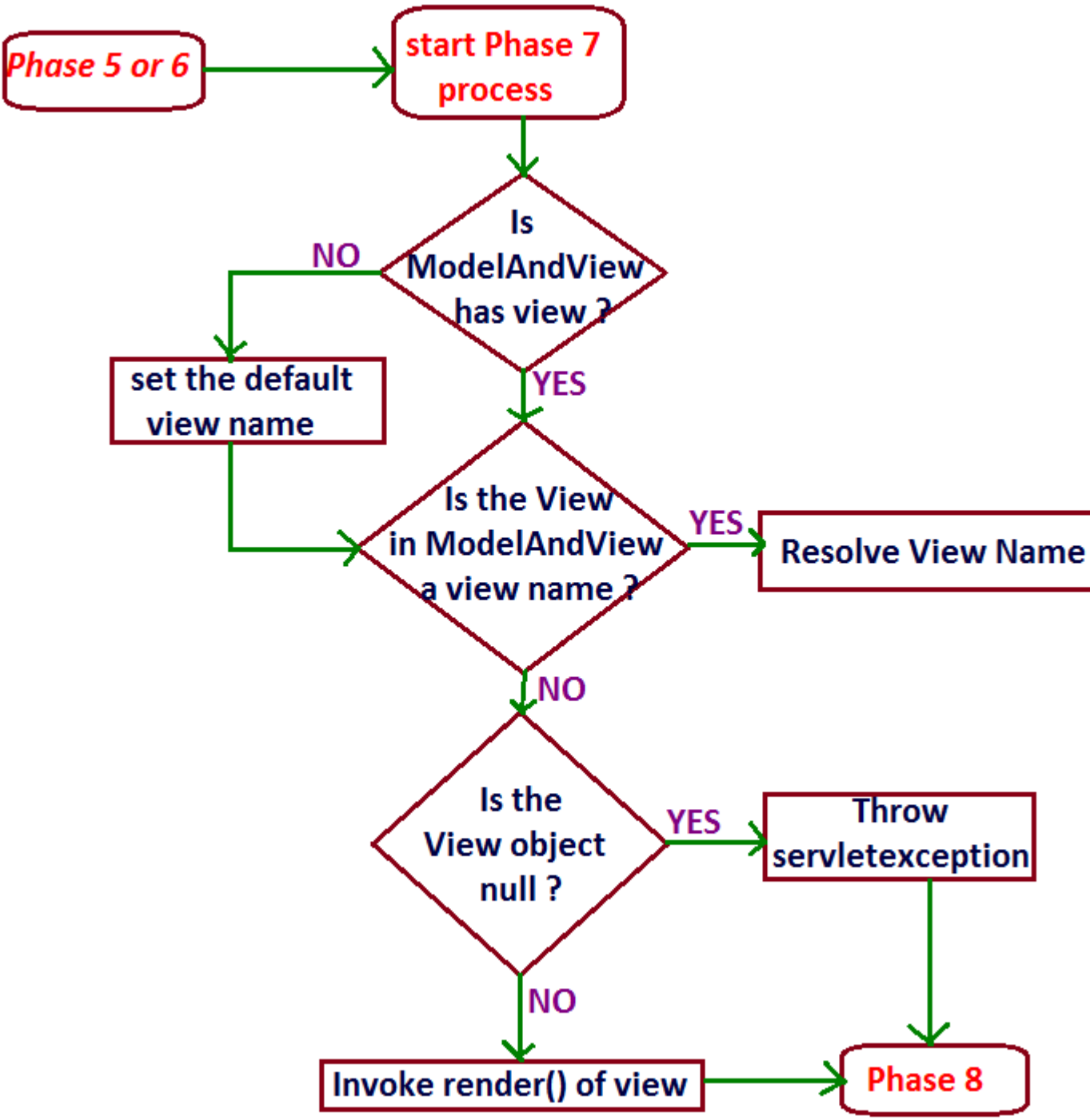
```
<web-app>

<servlet>
```

```
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

Phase 7 : Render the View

The Spring Web Mvc request processing workflow executes this phase only in the following two cases :

1. If the handler in Phase 4 retuns a valid(not null) ModelAndView object reference, and phase 5 is executed successfully (i.e., postHandle() method of HandlerInterceptors).
2. If the HandlerExceptionResolver in phase 6 returns a valid (not null) ModelAndView object reference.



Step 1 : Find the View :

- The work flow of Phase 7 of Spring Web Mvc request processing work flow starts with finding for the view in the

ModelAndView object produced in the previous phases (Phase 4 or 6).

- This is done using the hasView() method of ModelAndView. If hasView() method returns false i.e., the ModelAndView object does not contain a view , then a default view name is set to the ModelAndView object.
- The default view name is obtained by DispatcherServlet using the RequestToViewNameTranslator type object configured in this context.
- If there is no RequestToViewNameTranslator type bean configured in the context then null is set as view name.

Step 2 : Find the ModelAndView contains View reference :

- After the successful execution of step 1 the work flow continues with finding whether the view in the ModelAndView object is holding a View object reference or a view name. This is done using the isReference() method of ModelAndView.
- If isReference() method returns 'false' and if the view reference is not null then the work flow proceeds to the next step.
- If isReference() method returns 'true' then the work flow proceeds to resolve the view name. The 'Resolve View Name' process is explained in the next section. If the 'Resolve View Name' process successfully resolves the view name to View object then the workflow proceeds to the next step.
- If the view reference is null or the ViewResolver's fails to resolve the view name(i.e., if 'Resolve View Name' process returns null), the workflow proceeds to phase 8 by throwing a ServletException.

Step 3 : Delegate to View object for rendering :

- After successfully locating the View object the request is delegated to the view object invoking render() method of View.
- The view object takes the responsibility to prepare the response, i.e., presentation for the client.
- Thereafter, the workflow proceeds to Phase 8 to perform the finalizations.

The 'Resolve View Name' process :

- As described in the preceding section if the ModelAndView object describes a view name instead of View object, Dispatcherservlet uses the configured ViewResolver object to resolve the view name to locate View object.
- As an overview at this point we can simply consider that the ViewResolver object is responsible to locate the View object that can render a view for this request.

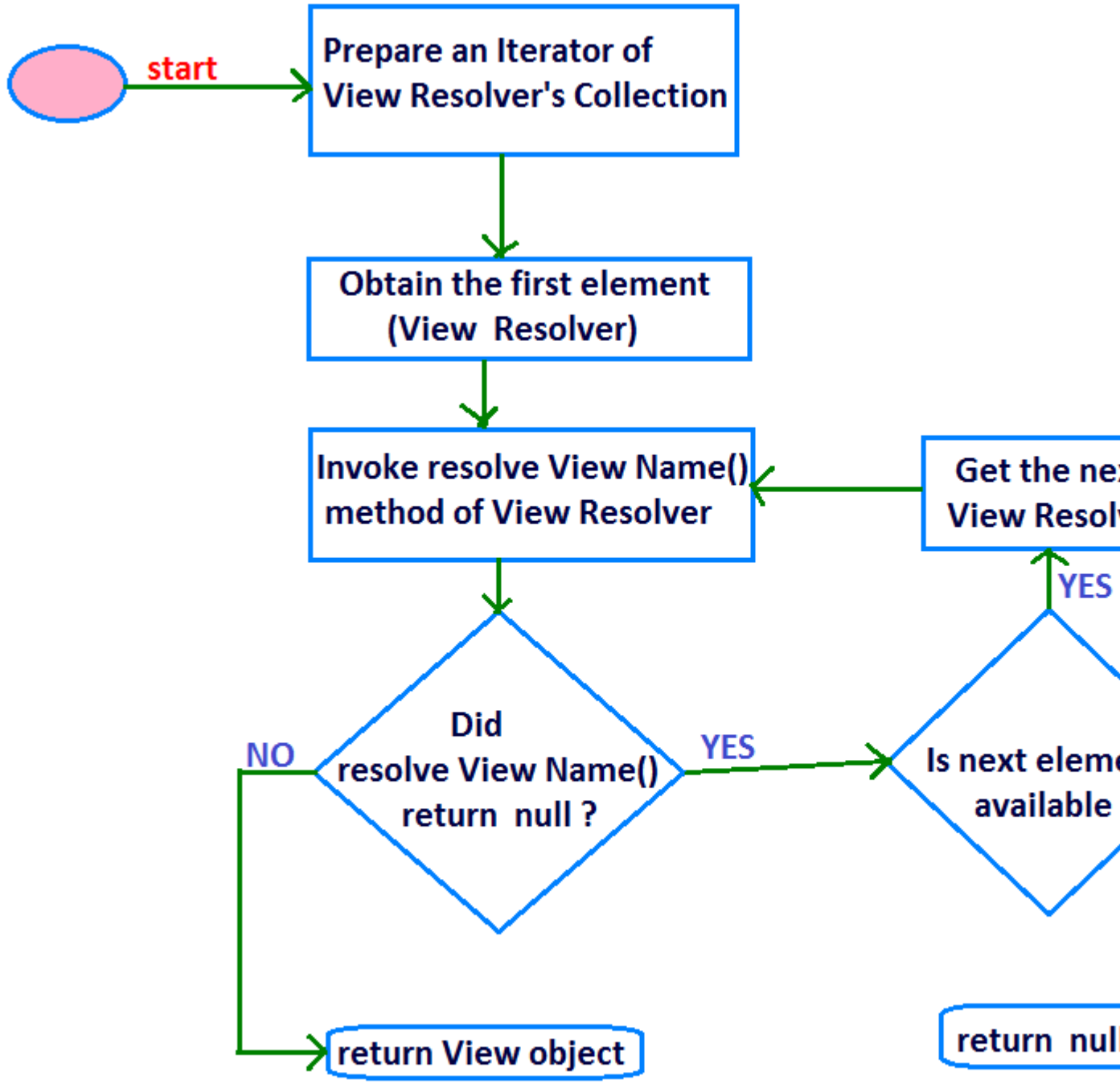
Step 1 : Prepare a ViewResolver Iterator

- The wokflow starts with the preparation of an iterator of the collection storing the ViewResolver objects. This collection is created while the DispatcherServlet is being initialized. Thereafter the first element is retrieved from the iterator.

Step 2 : Resolve View Name

- After step 1 the workflow proceeds to invoke the resolveViewName() method of the current ViewResolver object in the iteration.

- If the resolveViewName() method returns null, which indicates that the current ViewResolver failed to resolve the view name.
- In this case the workflow proceeds to the next step. If the resolveViewName() method returns a valid View object reference then the workflow completes returning the same, which is further used to render the view as described in the preceding section.

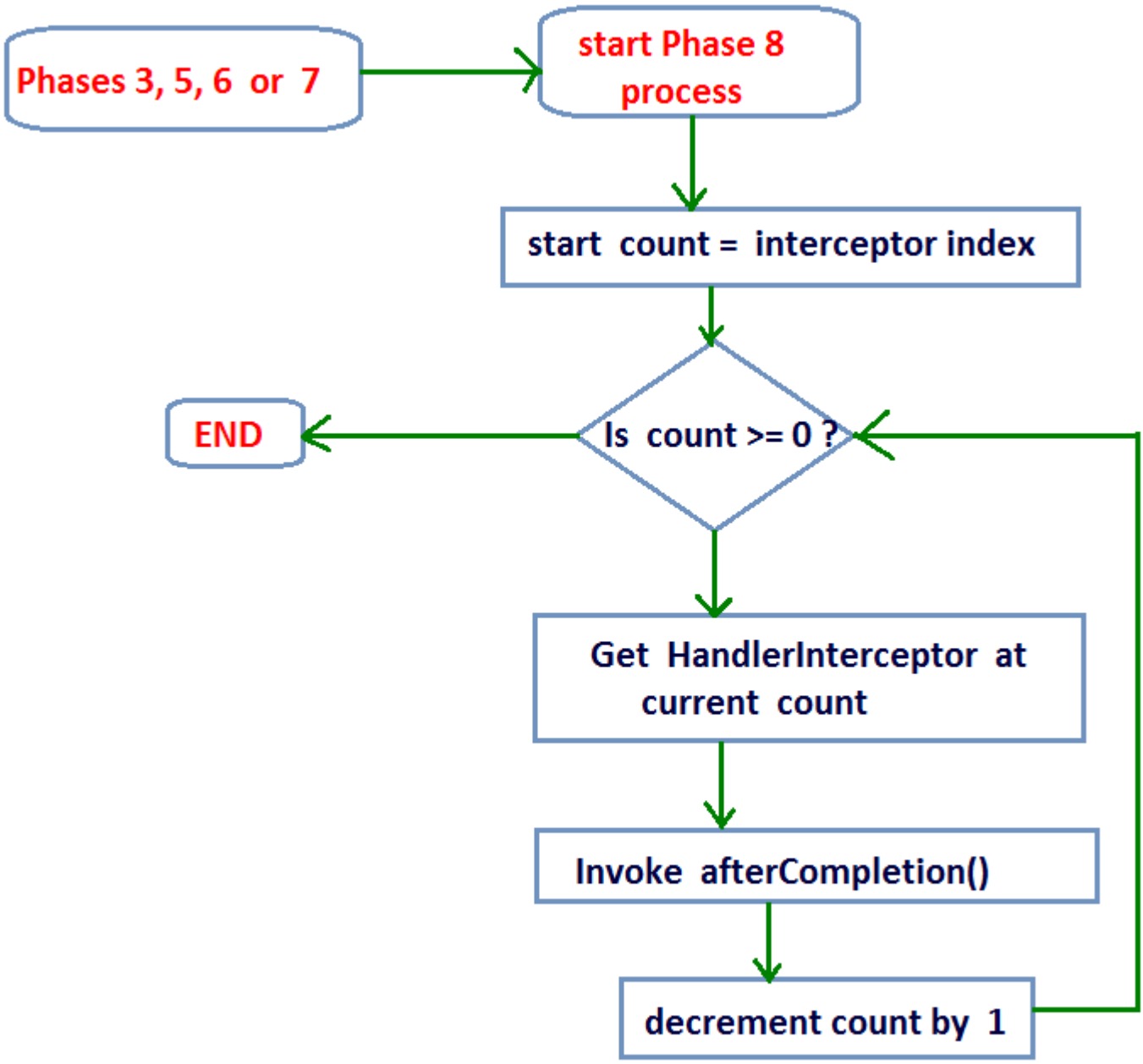


Step 3 : Move to next element

- If the ViewResolver in the preceding step fails to resolve the view name then the workflow proceeds to get the next ViewResolver from the iterator prepared in step 1.
- If the next element is available then the process moves to step 2, if not the workflow completes returning null.

Phase 8 : Execute Interceptors afterCompletion methods

- As explained in Phase 3, the HandlerInterceptor gives us an opportunity to add common pre- and post-processing behavior without needing to modify each handler implementation.
- While in Phase 3, the preHandle() method of handler interceptors are executed,
  And in Phase 5, the postHandle() methods are executed.
- In this phase of workflow the afterCompletion() method is executed on the Handler Interceptors whose preHandle() method has successfully executed and returned 'true'(i.e., in Phase 3).
- The afterCompletion() method is called on any outcome of handler execution, thus allowing us to do the resource cleanup.



Step 1 : Prepare counter :

- The workflow of Phase 8 starts with setting the count to interceptorIndex value.
- The interceptorIndex is set in the preprocessing phase, i.e., Phase 3 used to track the index of the interceptor in the list that have completed the preHandle processing successfully

so that in this phase the afterCompletion methods can be executed only on those HandlerInterceptors.

- That is, if the workflow reaches to this phase from Phase 5 then the interceptorIndex will be equal to the interceptors length minus 1(interceptorsLength - 1) since the preHandle() of all the interceptors are executed successfully.
- Thus the afterCompletion() will be executed on all interceptors in reverse order as compared to the order in which they are configured.
- If the workflow reaches to this phase because of any exception raised in the preHandle() or if preHqndle() returns 'false' then the interceptor chain stops execution the preHandle methods and delegates to this phase via Phase 6 or directly.
- After the counter is prepared if there are any interceptors configured, i.e., if the count is greater than or equal to 0(zero) then the workflow proceeds to step 2, if not it ends the request processing doing the necessary cleanup operations.

Step 2 : Obtain HandlerInterceptor :

After the counter is prepared successfully DispatcherServlet obtains the HandlerInterceptor at the current index from the interceptors list.

Step 3 : Invoke interceptors afterCompletion() method :

After obtaining the HandlerInterceptor DispatcherServlet invokes afterCompletion() method.

Step 4 : Change the counter :

- In the step of Phase 8 workflow the count is decrease by one and then verifies whether the current count is greater than or equal to 0(zero).
- If so then the workflow proceeds to step 2 of this phase, if not it ends the request processing doing the necessary cleanup operations.

```
With in this phase the Spring Web MVC request
processing workflow ends.
```

Describing Spring Controllers and Validations

Agenda :

1. Introduction
2. Types of Controllers (5) :
   i. Controller
   ii. AbstractCommandController
   iii. SimpleFormController
   iv. WizardFormController
   v. MultiActionController
3. Understanding Validators

The most common way to implement the handler in Spring Web MVC application is as a controller.

Types of Controllers

The most commonly used built-in controller types provided by the Spring Web MVC framework are :

1. Controller
2. AbstractCommandController
3. SimpleFormController
4. WizardFormController
5. MultiActionController

Controller (I)

The Controller interface declares only one method, i.e., handleRequest(-,-).

```
public ModelAndView
handleRequest(HttpServletRequest request,
              HttpServletResponse response) throws
Exception
```

AbstractCommandController (C)

```
protected ModelAndView handle(HttpServletRequest
request,
                              HttpServletResponse
response,
                         Object command,
                    BindException errors)throws
Exception
```

Working with AbstractCommandController :

login.html

```
<body>
 <form action="/login.spring" > <br> // / is
optional
 User Name : <input type="text" name="userName">
<br>
 Password: <input type="password" name="password">
<br>
 <input type="submit" value="Login">
```

```
 </form>
</body>
```

LoginController.java

```
package com.spring;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.validation.BindException;
import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.AbstractCommand
Controller;

public class LoginController extends
AbstractCommandController {

LoginModel loginModel;
public void setLoginModel(LoginModel lm){
 loginModel=lm;
}

protected ModelAndView handle(HttpServletRequest
request,
           HttpServletResponse response,
           Object command,
           BindException errors) throws Exception {

String type =
loginModel.getValidate((UserDetails)command);

System.out.println("type : "+type);

if(type==null)
     //return new ModelAndView("/login.html");
```

```
     throw new MyException("User details are not
valid");
else if(type.equals("admin"))
     return new ModelAndView("/pages/admin.jsp");
else
     return new ModelAndView("/pages/user.jsp");

} //handleRequest

}
```

LoginModel.java

```
package com.spring;

import
org.springframework.dao.EmptyResultDataAccessExcept
ion;
import org.springframework.jdbc.core.JdbcTemplate;

public class LoginModel {

private JdbcTemplate jdbcTemplate;

public LoginModel(JdbcTemplate jdbcTemplate) {
this.jdbcTemplate=jdbcTemplate;
}

public String getValidate(UserDetails user) {

 String sql="select type from UserDetails where
username=\'"
  +user.getUserName()+"\' and
password=\'"+user.getPassword()+"\' ";

 try{
     return jdbcTemplate.queryForObject(sql,
String.class);
 }catch (EmptyResultDataAccessException e) {
```

```
      return null;
}

} //getValidate()

}
```

UserDetails.java

```
package com.spring;

public class UserDetails {

private String userName;
private String password;

public String getUserName() {
     return userName;
}
public void setUserName(String userName) {
     this.userName = userName;
}
public String getPassword() {
     return password;
}
public void setPassword(String password) {
     this.password = password;
}

}
```

ds-servlet.xml

```
<beans>

<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
 <value>oracle.jdbc.driver.OracleDriver</value>
```

```
</property>
<property name="url">
 <value>jdbc:oracle:thin:@localhost:1521:xe</value>
</property>
<property name="username">
 <value>lms</value>
</property>
<property name="password">
 <value>scott</value>
</property>
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="loginModel"
class="com.spring.LoginModel">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>

<bean id="loginController"
class="com.spring.LoginController">
 <property name="loginModel">
  <ref local="loginModel"/>
 </property>

<!-- configure the command class name -->
 <property name="commandClass">
```

```
  <value
type="java.lang.Class">com.spring.UserDetails</valu
e>
 </property>

<!-- configure the command name, the name to use
when binding the instantiated command class to the
request -->

 <property name="commandName">
  <value>UserDetails</value>
 </property>

</bean>

<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
<property name="mappings">
 <props>
  <prop key="/login.spring">loginController</prop>
 </props>
</property>
</bean>

<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.Simp
leMappingExceptionResolver">
 <property name="exceptionMappings">
  <props>
   <prop key="com.spring.MyException">
    /myError.html
   </prop>
  </props>
 </property>
</bean>
```

```
</beans>
```

web.xml

```
<web-app>

<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

admin.jsp

```
Wel come to Admin page <br/>

User Name : <%= request.getParameter("userName") %>
```

user.jsp

```
Wel come to User page <br/>

User Name : <%= request.getParameter("userName") %>
```

myError.html

```
<body>
A simple test error page <br>
User details are not valid
</body>
```

AbstractCommandController can bind the request data to the command object properties.
We are using UserDetails class as a command class.

Understanding Validators

step 1 : write a validator class

```
public boolean supports(Class c)
public void validate(Object target, Errors errors)
public boolean supports(Class c) {
return c.equals(UserDetails.class);
}
```

step 2 : Associate the Validator to the Controller

```
<bean id="loginController"
class="com.spring.LoginController">
 <property name="loginModel">
  <ref local="loginModel"/>
 </property>

<!-- configure the command class name -->
 <property name="commandClass">
  <value
type="java.lang.Class">com.spring.UserDetails</valu
e>
 </property>

<!-- configure the command name, the name to use
when binding the instantiated command class to the
request -->

 <property name="commandName">
  <value>UserDetails</value>
 </property>

 <property name="validator">
 <bean class="com.spring.UserValidator"/>
 </property>
```

```
</bean>
```

Working with Validators

login.html

```
<form action="login.spring" > <br>
 User Name : <input type="text" name="userName">
<br>
 Password: <input type="password" name="password">
<br>
 <input type="submit" value="Login">
 </form>
```

UserValidator.java

```
package com.spring;

import org.springframework.validation.Errors;
import
org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class UserValidator implements Validator {

public boolean supports(Class c) {
return c.equals(UserDetails.class);
}

public void validate(Object target, Errors errors)
{
UserDetails user=(UserDetails)target;


ValidationUtils.rejectIfEmptyOrWhitespace(errors,
"userName",
     "field.required", "The userName field cannot
be empty<br>");
```

```
ValidationUtils.rejectIfEmptyOrWhitespace(errors,
"password",
          "field.required", "The password field
cannot be empty<br>");

if(user.getUserName()!= null &&
!user.getPassword().equals("")
                                &&
user.getPassword().length()<5){

  errors.rejectValue("password", "field.minlength",
                  new Object[]{Integer.valueOf(5)},
                 "The password must contain minimum
5 characters.<br>");
}//if

} //validate
} //class
```

LoginController.java

```
package com.spring;

import java.io.PrintWriter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.validation.BindException;
import org.springframework.validation.FieldError;
import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.AbstractCommand
Controller;

public class LoginController extends
AbstractCommandController {
```

```
LoginModel loginModel;
public void setLoginModel(LoginModel lm){
 loginModel=lm;
}

protected ModelAndView handle(HttpServletRequest
request,
          HttpServletResponse response,
          Object command,
          BindException errors) throws Exception {


if(errors.hasErrors()){
 System.out.println("Errors in validation");
 PrintWriter out=response.getWriter();
 out.println("errors in data submitted : <br>");
 out.println("no of errors :
"+errors.getErrorCount()+"<br>");

 out.println("userName fields errors : <br>");
 java.util.List<FieldError>
errorList=errors.getFieldErrors("userName");
 for(FieldError error:errorList)
  out.println(error.getDefaultMessage());

 out.println("password fields errors : <br>");
 errorList=errors.getFieldErrors("password");
 for(FieldError error:errorList)
 out.println(error.getDefaultMessage());

 return null;
}
```

```
String type =
loginModel.getValidate((UserDetails)command);

System.out.println("type : "+type);

if(type==null)
     //return new ModelAndView("/login.html");
     throw new MyException("User details are not
valid ..");
else if(type.equals("admin"))
     return new ModelAndView("/pages/admin.jsp");
else
     return new ModelAndView("/pages/user.jsp");

} //handleRequest

}
```

LoginModel.java

```
package com.spring;

import
org.springframework.dao.EmptyResultDataAccessExcept
ion;
import org.springframework.jdbc.core.JdbcTemplate;

public class LoginModel {

private JdbcTemplate jdbcTemplate;

public LoginModel(JdbcTemplate jdbcTemplate) {
this.jdbcTemplate=jdbcTemplate;
}

public String getValidate(UserDetails user) {

 String sql="select type from UserDetails where
username=\'"
```

```
 +user.getUserName()+"\' and
password=\'"+user.getPassword()+"\' ";

 try{
     return jdbcTemplate.queryForObject(sql,
String.class);
 }catch (EmptyResultDataAccessException e) {
      return null;
}

} //getValidate()

}
```

UserDetails.java

```
package com.spring;

public class UserDetails {

private String userName;
private String password;

public String getUserName() {
     return userName;
}
public void setUserName(String userName) {
     this.userName = userName;
}
public String getPassword() {
     return password;
}
public void setPassword(String password) {
     this.password = password;
}

}
```

MyException.java

```
package com.spring;

public  class MyException extends RuntimeException
{

     public MyException(String s) {
          super(s);
     }
}
```

admin.jsp

```
Wel come to Admin page <br/>

User Name : <%= request.getParameter("userName") %>
```

user.jsp

```
Wel come to User page <br/>

User Name : <%= request.getParameter("userName") %>
```

myError.html

```
A simple test error page <br>
User details are not valid
```

ds-servlet.xml

```
<beans>

<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
 <value>oracle.jdbc.driver.OracleDriver</value>
</property>
<property name="url">
 <value>jdbc:oracle:thin:@localhost:1521:xe</value>
</property>
<property name="username">
 <value>lms</value>
</property>
<property name="password">
```

```
 <value>scott</value>
</property>
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="loginModel"
class="com.spring.LoginModel">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>

<bean id="loginController"
class="com.spring.LoginController">
 <property name="loginModel">
  <ref local="loginModel"/>
 </property>

<!-- configure the command class name -->
 <property name="commandClass">
  <value
type="java.lang.Class">com.spring.UserDetails</valu
e>
 </property>

<!-- configure the command name, the name to use
when binding the instantiated command class to the
request -->

 <property name="commandName">
```

```
  <value>UserDetails</value>
 </property>

 <property name="validator">
 <bean class="com.spring.UserValidator"/>
 </property>

</bean>

<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.Simp
leUrlHandlerMapping">
<property name="mappings">
 <props>
  <prop key="/login.spring">loginController</prop>
 </props>
</property>
</bean>


<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.Simp
leMappingExceptionResolver">
 <property name="exceptionMappings">
  <props>
   <prop key="com.spring.MyException">
    /myError.html
   </prop>
  </props>
 </property>
</bean>

</beans>
```

web.xml

```
<web-app>
```

```
<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

result :

```
errors in data submitted :
no of errors : 2
userName fields errors :
The userName field cannot be empty
password fields errors :
The password field cannot be empty
```

SimpleFormController

home.jsp

```
<html>
<body>
<a href="addEmployee.spring">Add Employee</a>
</body>
</html>
```

addEmployee.jsp

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

<html>
```

```
<body>
<form:errors path="EmpDetails.name"/> <br> <br>
<form:errors path="EmpDetails.eno"/>

<form action="addEmployee.spring" method="post">
<pre>
Employee Number : <input type="text" name="eno"/>
Employee Name : <input type="text" name="name"/>
Employee Salary : <input type="text"
name="salary"/>
Employee Address : <input type="text"
name="address"/>
Employee JoinDate : <input type="text" name="doj"/>
 <input type="submit" value="Add Employee"/>
</pre>
</form>

</body>
</html>
```

addEmployeeSuccess.jsp

```
<html>
<body>
Employee details added Successfully
</body>
</html>
```

AddEmployeeController.java

```
package com.form;

import
org.springframework.web.servlet.mvc.SimpleFormContr
oller;

public class AddEmployeeController extends
SimpleFormController {

EmployeeServices employeeServices;
```

```
public void setEmployeeServices(EmployeeServices
employeeServices){
 this.employeeServices=employeeServices;
 System.out.println("setEmployees from
controller");
}
public void doSubmitAction(Object command) throws
Exception{
     employeeServices.create((EmpDetails)command);
     System.out.println("doSubmitAction from
controller");
}

}
```

EmpValidator.java

```
package com.form;

import org.springframework.validation.Errors;
import
org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class EmpValidator implements Validator {

public boolean supports(Class c) {
 boolean flag=c.equals(EmpDetails.class);
 return flag;
}

public void validate(Object target, Errors errors)
{
 EmpDetails empDetails=(EmpDetails)target;

 ValidationUtils.rejectIfEmptyOrWhitespace(errors,
   "name", "fiels.required","The name field cannot
be empty");
```

```
if(empDetails.getEno()<10){
 errors.rejectValue("eno", "field.minvalue",
  new Object[]{Integer.valueOf(9)},"The deptno
should be greater than 9");
 }
 else if(empDetails.getEno() > 99){
     errors.rejectValue("eno", "field.maxvalue",
     new Object[]{Integer.valueOf(99)},"The deptno
should be less than 99");
 }

 System.out.println("Emp Validator");
}//validate

} //class
```

EmployeeServices.java

```
package com.form;

import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.jdbc.core.PreparedStatementSett
er;

public class EmployeeServices {

private JdbcTemplate jdbcTemplate;
public EmployeeServices(JdbcTemplate jdbcTemplate){
 this.jdbcTemplate=jdbcTemplate;
}

public void create(final EmpDetails user) {
```

```
jdbcTemplate.update("insert into emp
values(?,?,?,?,?)",
 new PreparedStatementSetter(){

public void setValues(PreparedStatement ps) throws
SQLException {
ps.setInt(1, user.getEno());
ps.setString(2, user.getName());
ps.setString(3, user.getAddress());
ps.setDouble(4, user.getSalary());
ps.setDate(5, new
Date(System.currentTimeMillis()));

System.out.println("Emp services update");
}

} //PreparedStatementSetter
); //update()

}//create

}
```

EmpDetails.java

```
package com.form;

import java.sql.Date;

public class EmpDetails {

private int eno;
private String name,address;
private double salary;
private Date date;

public int getEno() {
     return eno;
}
```

```
public void setEno(int eno) {
     this.eno = eno;
}
public String getName() {
     return name;
}
public void setName(String name) {
     this.name = name;
}
public String getAddress() {
     return address;
}
public void setAddress(String address) {
     this.address = address;
}
public double getSalary() {
     return salary;
}
public void setSalary(double salary) {
     this.salary = salary;
}
public Date getDate() {
     return date;
}
public void setDate(Date date) {
     this.date = date;
}

}
```

ds-servlet.xml

```
<beans>

<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName">
 <value>oracle.jdbc.driver.OracleDriver</value>
```

```
</property>
<property name="url">
 <value>jdbc:oracle:thin:@localhost:1521:xe</value>
</property>
<property name="username">
 <value>lms</value>
</property>
<property name="password">
 <value>scott</value>
</property>
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="empServices"
class="com.form.EmployeeServices">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>

<bean name="/addEmployee.spring"
class="com.form.AddEmployeeController">
 <property name="employeeServices">
  <ref local="empServices"/>
 </property>

 <property name="commandClass">
  <value
type="java.lang.Class">com.form.EmpDetails</value>
 </property>
```

```
 <property name="commandName">
  <value>EmpDetails</value>
 </property>

 <property name="validator">
  <bean class="com.form.EmpValidator"/>
 </property>

 <property name="sessionForm" value="false"/>

 <property name="formView">
  <value>/addEmployee.jsp</value>
 </property>

 <property name="successView">
  <value>/addEmployeeSuccess.jsp</value>
 </property>

</bean>

</beans>
```

web.xml

```
<web-app>

<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>
```

```
</web-app>
```

http://localhost:8001/spring/home.jsp

Result :

```
setEmployees from controller
Emp Validator
Emp services update
doSubmitAction from controller
```

WizardFormController

registration1.html

```
<html>
<body>
<form name="form1"  action="registration.spring"
method="post">
<table border="0" align="left">
<tr>
<td colspan="2">
<div align="center"><strong>Registration Step 1
:</strong> </div>
</td>
</tr>
<tr><td>  </td><td>   </td></tr>
<tr>
<td>User Name :</td>
<td><input type="text" name="uname"
id="uname"></td>
</tr>
<tr>
<td>Password :</td>
<td><input type="password" name="pass"
id="pass"></td>
</tr>
<tr>
<td>Re-Password :</td>
```

```
<td><input type="text" name="repass"
id="repass"></td>
</tr>
<tr>
<td>Email :</td>
<td><input type="text" name="email"
id="email"></td>
</tr>
<tr><td>  </td><td>   </td></tr>

<tr>
<td colspan="2">
<div align="center">
<input type="submit" name="Submit" value="Next >">
</div></td>
</tr>
</table>
<input type="hidden" name="_page" value="1">
</form>
</body>
</html>
```

registration2.html

```
<html>
<body>
<form name="form1"  action="registration.spring"
method="post">
<table border="0" align="left" width="45%">
<tr>
<td colspan="2">
<div align="center"><strong>Registration Step 2
:</strong> </div>
</td>
</tr>
<tr><td>  </td><td>   </td></tr>
<tr>
<td>First Name :</td>
```

```
<td><input type="text" name="fname"
id="fname"></td>
</tr>
<tr>
<td>LastName :</td>
<td><input type="text" name="lname"
id="lname"></td>
</tr>
<tr>
<td>Middle Name :</td>
<td><input type="text" name="mname"
id="mname"></td>
</tr>
<tr>
<td>DOB :</td>
<td><input type="text" name="dob" id="dob"></td>
</tr>
<tr>
<td>Mobile :</td>
<td><input type="text" name="mobile"
id="mobile"></td>
</tr>
<tr>
<td>Phone :</td>
<td><input type="text" name="phone"
id="phone"></td>
</tr>
<tr><td>  </td><td>   </td></tr>

<tr>
<td colspan="2">
<div align="center">
<input type="submit" name="Submit" value="Next >">
</div></td>
</tr>
</table>
<input type="hidden" name="_page" value="2">
```

```
</form>
</body>
</html>
```

registrationFinal.html

```
<html>
<body>
<form name="form1"  action="registration.spring"
method="post">
<table border="0" align="left" width="45%">
<tr>
<td colspan="3" align="center">
<div><strong>Registration Final Step  :</strong>
</div>
</td>
</tr>
<tr><td>  </td><td>   </td></tr>
<tr>
<td colspan="2"><strong>Address : </strong></td>
</tr>
<tr>
<td>   Address1 :</td>
<td><input type="text" name="addr" id="addr"></td>
</tr>
<tr>
<td>   Street :</td>
<td><input type="text" name="street"
id="street"></td>
</tr>
<tr>
<td>   City :</td>
<td><input type="text" name="city" id="city"></td>
</tr>
<tr>
<td>    State :</td>
<td><input type="text" name="mobile"
id="mobile"></td>
</tr>
```

```
<tr>
<td>   Country :</td>
<td><input type="text" name="phone"
id="phone"></td>
</tr>
<tr><td>  </td><td>   </td></tr>

<tr>
<td colspan="2">
<div align="center">
<input type="submit" name="Submit" value="Finish">
</div></td>
</tr>
</table>
<input type="hidden" name="_finish" value="Finish">
</form>
</body>
</html>
```

RegistrationContoller.java

```
package com.wizard;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.validation.BindException;
import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.AbstractWizardF
ormController;

@SuppressWarnings("deprecation")
public class RegistrationController extends
AbstractWizardFormController {

private UserDAO userDAO;
```

```
public RegistrationController(UserDAO userDAO){
 this.userDAO=userDAO;
}

@Override
protected ModelAndView
processFinish(HttpServletRequest request,
                HttpServletResponse response,
                Object command,
                BindException exceptions) throws
Exception {
UserDetails userDetails=(UserDetails)command;
boolean flag=userDAO.create(userDetails);
if(flag)
 return new
ModelAndView("registrationSuccess.jsp");
return new ModelAndView("registrationFail.jsp");
}//processFinish

}
```

UserDetails.java

```
package com.wizard;

public class UserDetails {

private String uname,pass,repass,email;
private String fname,lname,mname,dob,mobile,phone;
private String addr,street,city,state,country;
public String getUname() {
     return uname;
}
public void setUname(String uname) {
     this.uname = uname;
}
public String getPass() {
     return pass;
}
```

```
public void setPass(String pass) {
     this.pass = pass;
}
public String getRepass() {
     return repass;
}
public void setRepass(String repass) {
     this.repass = repass;
}
public String getEmail() {
     return email;
}
public void setEmail(String email) {
     this.email = email;
}
public String getFname() {
     return fname;
}
public void setFname(String fname) {
     this.fname = fname;
}
public String getLname() {
     return lname;
}
public void setLname(String lname) {
     this.lname = lname;
}
public String getMiddle() {
     return mname;
}
public void setMiddle(String mname) {
     this.mname = mname;
}
public String getDob() {
     return dob;
}
public void setDob(String dob) {
```

```
        this.dob = dob;
}
public String getMobile() {
        return mobile;
}
public void setMobile(String mobile) {
        this.mobile = mobile;
}
public String getPhone() {
        return phone;
}
public void setPhone(String phone) {
        this.phone = phone;
}
public String getAddr() {
        return addr;
}
public void setAddr(String addr) {
        this.addr = addr;
}
public String getStreet() {
        return street;
}
public void setStreet(String street) {
        this.street = street;
}
public String getCity() {
        return city;
}
public void setCity(String city) {
        this.city = city;
}
public String getState() {
        return state;
}
public void setState(String state) {
        this.state = state;
```

```
}
public String getCountry() {
     return country;
}
public void setCountry(String country) {
     this.country = country;
}

}
```

UserDAO.java

```
package com.wizard;

public interface UserDAO {

 boolean create(UserDetails userDetails);

}
```

UserDAOImpl.java

```
package com.wizard;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

import org.springframework.jdbc.core.JdbcTemplate;
import
org.springframework.jdbc.core.PreparedStatementSett
er;

public class UserDAOImpl implements UserDAO {

private JdbcTemplate jdbcTemplate;
public UserDAOImpl(JdbcTemplate jdbcTemplate){
 this.jdbcTemplate=jdbcTemplate;
}
```

```
public boolean create(final UserDetails
userDetails) {
System.out.println("create");
int count= jdbcTemplate.update(
 "insert into users
values(?,?,?,?,?,?,?,?,?,?,?,?,?,?)",
 new PreparedStatementSetter(){
  public void setValues(PreparedStatement ps)
throws SQLException {

   ps.setString(1,userDetails.getUname());
   ps.setString(2,userDetails.getPass());
   ps.setString(3,userDetails.getEmail());
   ps.setString(4,userDetails.getFname());
   ps.setString(5,userDetails.getLname());
   ps.setString(6,userDetails.getMiddle());
   ps.setString(7,userDetails.getDob());
   ps.setString(8,userDetails.getMobile());
   ps.setString(9,userDetails.getPhone());
   ps.setString(10,userDetails.getAddr());
   ps.setString(11,userDetails.getStreet());
   ps.setString(12,userDetails.getCity());
   ps.setString(13,userDetails.getState());
   ps.setString(14,userDetails.getCountry());

  }//setValues()

} //PreparedStatementSetter()
); //update
 return (count==1 ||
count==Statement.SUCCESS_NO_INFO );
} //create

}
```

RegistrationFail.html

```
<html>
<body>
```

```
<b>Registration Process Failed</b>
</body>
</html>
```

RegistrationSuccess.html

```
<html>
<body>
<b>Registration Process Success </b>
</body>
</html>
```

applicationControllers.xml

```
<beans>

<bean name="/registration.spring"
class="com.wizard.RegistrationController">
<constructor-arg>
  <ref bean="userDAO"/>
</constructor-arg>

<property name="commandClass">
 <value
type="java.lang.Class">com.wizard.UserDetails</valu
e>
</property>

<property name="commandName">
 <value>UserDetails</value>
</property>

<property name="sessionForm" value="true"/>
<property name="pages">
<list>
 <value>/registration1.html</value>
 <value>/registration2.html</value>
 <value>/registrationFinal.html</value>
</list>
</property>
```

```
<property name="allowDirtyForward">
 <value>false</value>
</property>
</bean>

</beans>
```

applicationDAOs.xml

```
<beans>

<!-- configure DataSource -->
<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
<property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver" />
<property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe" />
<property name="username" value="lms" />
<property name="password" value="scott" />
</bean>

<!-- configure JdbcTemplate -->
<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
 <constructor-arg>
  <ref local="dataSource" />
 </constructor-arg>
</bean>

<bean id="userDAO" class="com.wizard.UserDAOImpl">
 <constructor-arg>
  <ref local="jdbcTemplate"/>
 </constructor-arg>
</bean>

</beans>
```

web.xml

```
<web-app>

<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/applicationControllers.xml
  /WEB-INF/applicationDAOs.xml
 </param-value>
</init-param>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

create users table in DataBase :

```
create table users(uname varchar2(15), pass
varchar2(15),
                   email varchar2(15), fname
varchar2(15),
                   lname varchar2(15),mname
varchar2(15),
                   dob varchar2(15), moble
varchar2(15),
                   phone varchar2(15),addr
varchar2(15),
                   street varchar2(15),city
varchar2(15),
```

```
                    state varchar2(15),country
varchar2(15));
```

result:

http:///localhost:8001/spring/registration.spring

MultiActionController

ArithmeticController.java

```
package com.multi;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import
org.springframework.web.servlet.ModelAndView;
import
org.springframework.web.servlet.mvc.multiaction.Mul
tiActionController;

public class ArithmeticController extends
MultiActionController {

public ModelAndView add(HttpServletRequest request,
                HttpServletResponse response)
throws Exception{
int
op1=Integer.parseInt(request.getParameter("operand1
"));
int
op2=Integer.parseInt(request.getParameter("operand2
"));
int result=op1+op2;
return new
ModelAndView("/home.jsp","result",result+"");
}
```

```
public ModelAndView subtract(HttpServletRequest
request,
          HttpServletResponse response) throws
Exception{
int
op1=Integer.parseInt(request.getParameter("operand1
"));
int
op2=Integer.parseInt(request.getParameter("operand2
"));
int result=op1-op2;
return new
ModelAndView("/home.jsp","result",result+"");
}

}
```

home.jsp

```
<html>
<body>
<% if(request.getAttribute("result")!=null) { %>
Result of previous  request (<%=
request.getParameter("submit") %>) :
<b>
<%= request.getParameter("operand1")%>,
<%= request.getParameter("operand2")%>
is :  <%= request.getAttribute("result")%>
</b>
<%} %>

<form method="post" action="myPath.spring">
Operand 1 : <input type="text" name="operand1"/>
<br>
Operand 2 : <input type="text" name="operand2"/>
<br>
<input type="submit" name="submit" value="add"/>
<input type="submit" name="submit"
value="subtract"/>
```

```
</form>

</body>
</html>
<beans>

<bean id="myMethodResolver"
class="org.springframework.web.servlet.mvc.multiact
ion.ParameterMethodNameResolver">
 <property name="paramName" value="submit"/>
</bean>

<bean name="/myPath.spring"
class="com.multi.ArithmeticController">
 <property name="methodNameResolver"
ref="myMethodResolver"/>
</bean>
</beans>
```

web.xml

```
<web-app>

<servlet>
<servlet-name>ds</servlet-name>
<servlet-
class>org.springframework.web.servlet.DispatcherSer
vlet</servlet-class>
<load-on-startup>0</load-on-startup>

</servlet>
<servlet-mapping>
<servlet-name>ds</servlet-name>
<url-pattern>*.spring</url-pattern>
</servlet-mapping>

</web-app>
```

Describing Spring View-Resolver and View

Agenda :

1. Types of View Resolver
    1. UrlBasedViewResolver
    2. InternalResourceViewResolver
    3. ResourceBundleViewResolver
    4. BeanNameViewResolver
    5. XmlViewResolver

View Resolver

1. UrlBasedViewResolver
2. InternalResourceViewResolver
3. ResourceBundleViewResolver
4. BeanNameViewResolver
5. XmlViewResolver

UrlBasedViewResolver

applicationContext.xml

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBase
dViewResolver">
 <property name="prefix" value="/WEB-INF/jsps"/>
 <property name="suffix" value=".jsp"/>
 <property name="viewClass"

value="org.springframework.web.servlet.view.Interna
lResourceView"/>
</bean>
```

InternalResourceViewResolver

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.Interna
lResourceViewResolver">
 <property name="prefix" value="/WEB-INF/jsps"/>
 <property name="suffix" value=".jsp"/>
 <property name="viewClass"

value="org.springframework.web.servlet.view.Interna
lResourceView"/>
</bean>
```

ResourceBundleViewResolver

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.Resourc
eBundleViewResolver">
 <property name="basename" value="myViews"/>
</bean>
```

myViews.properties

BeanNameViewResolver

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.BeanNam
eViewResolver"/>
```

XmlViewResolver

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.XmlView
Resolver">
```

```
 <property name="location" value="/WEB-
INF/views/MYViews.xml"/>
</bean>
```

Configuring Multiple ViewResolvers :

```
<bean
class="org.springframework.web.servlet.view.XmlView
Resolver">
  <property name="location" value="/WEB-
INF/views/MYViews.xml"/>
  <property name="order" value="0"/>
</bean>

<bean
class="org.springframework.web.servlet.view.BeanNam
eViewResolver">
 <property name="order" value="1"/>
</bean>

<bean
class="org.springframework.web.servlet.view.Interna
lResourceViewResolver">
  <property name="prefix" value="/"/>
  <property name="suffix" value=".jsp"/>
  <property name="viewClass"
value="org.springframework.web.servlet.view.JstlVie
w"/>
  <property name="order" value="2"/>
</bean>
```

Understanding View :

MyGifView.java

```
 package com.view;

import java.util.Map;
```

```
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.View;

public class MyGifView implements View {

public String getContentType() {
 return "image/gif";
}

public void render(Map model,
              HttpServletRequest request,
              HttpServletResponse response) throws
Exception {
 byte[]   content=(byte[])
model.get("ResponseContent");
 ServletOutputStream
sos=response.getOutputStream();
 sos.write(content);
 sos.flush();
 sos.close();
}//render()

}
```

Working with JSTL View :

- JSP Standard Tag Library(JSTL) is a collection of custom tag libraries, which provide core functionality used for JSP document.
- JSTL reduces the use of scriptlets in a JSP page.
- The use of JSTL tags allows developers to use predefined tags instead of writing the Java code.

JSTL provides 4 types of Tag Libraries :

1. **Core Tags :** used to process core operations in a JSP page.
2. **XML Tags :** used for parsing, selecting, and transforming XML data in a JSP page.
3. **Format Tags :** used for formatting the data used in a JSP page according to locale.
4. **SQL Tags :** used to access the relational database used in a JSP page.

You Want more JSTL

Example for JSTLView

SerachEmployeeController.java

EmpDAO.java

EmpDAOImplDB.java

EmpRowMapper.java

EmpDetails.java

services-contect.xml

webConfig-context.xml

search.jsp

employeeDetails.jsp

employeeList.jsp

dbError.jsp

notANumberError.jsp

ApplicationResources_en.properties

web.xml

Generating Views using Velocity :