Unit-7 Java Web Frameworks: Spring MVC

GTU Syllabus:

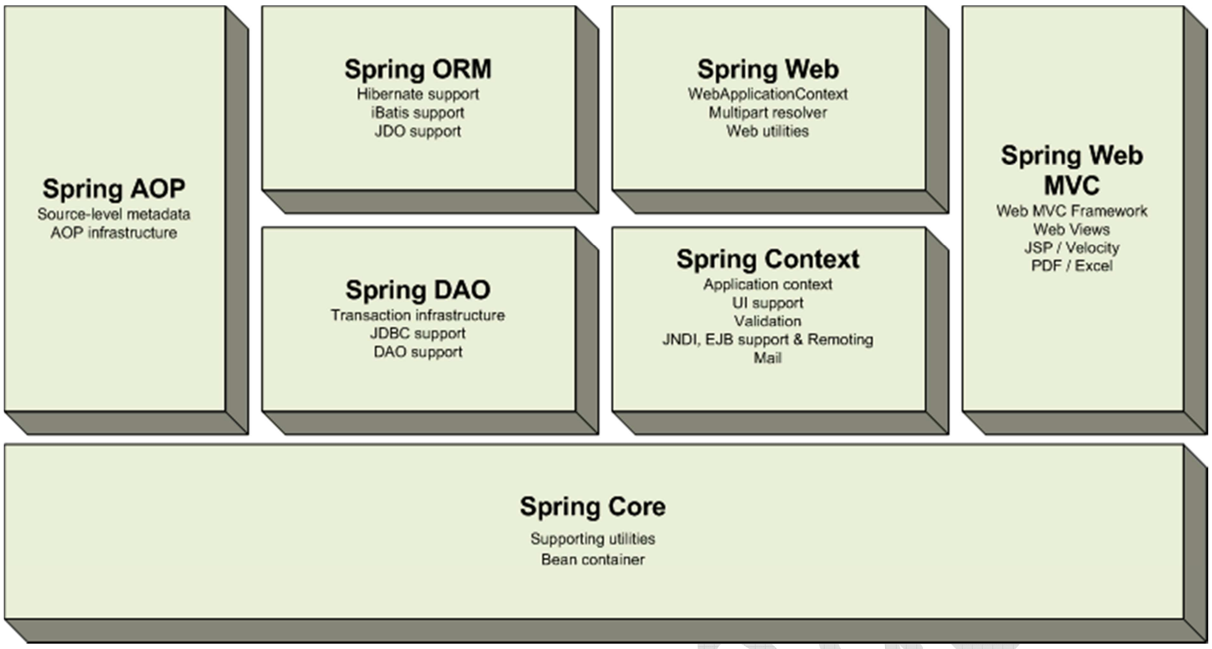
Overview of Spring, Spring Architecture, bean life cycle, XML Configuration on Spring, Aspect – oriented Spring, Managing Database, Managing Transaction

No	GTU Questions	Marks	year
1	Discuss the application design with MVC architecture.	3	W-2017
2	Briefly explain spring bean life cycle.	7	W-2017
3	Explain architecture of Spring MVC Framework. Explain all modules in brief.	7	S-2017
4	What is Dependency Injection?	3	S-2017
5	What is Spring IoC container?	4	S-2017
6	What is Spring Web MVC framework? List its key features.	7	S-2016
7	Discuss the application design with MVC architecture.With figure	7	W-2016

Overview of Spring:

Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly. Spring framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.

Modules of the Spring Framework



Very loosely coupled, components widely reusable and separately packaged.

The Spring Framework can be considered as a collection of frameworks-in-the-framework:

- **Core -** Inversion of Control (IoC) and Dependency Injection.
- **AOP :-** These modules support aspect oriented programming implementation where you can use Advices, Pointcuts etc. to decouple the code.
- **DAO -** Data Access Object support, transaction management, JDBC-abstraction
- **ORM -** Object Relational Mapping data access, integration layers for JPA, JDO, Hibernate, and iBatis
- **MVC -** Model-View-Controller implementation for web-applications

Context-This module supports internationalization (I18N), EJB,Remote Access, Authentication and Authorization, Remote Management like RMI,HTTP Invoker,Hessain, Messaging Framework, Web Services, Email, Testing

- **Expression Language:-** It is an extension to the EL defined in JSP. It provides support to setting and getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrieval of objects by name etc.

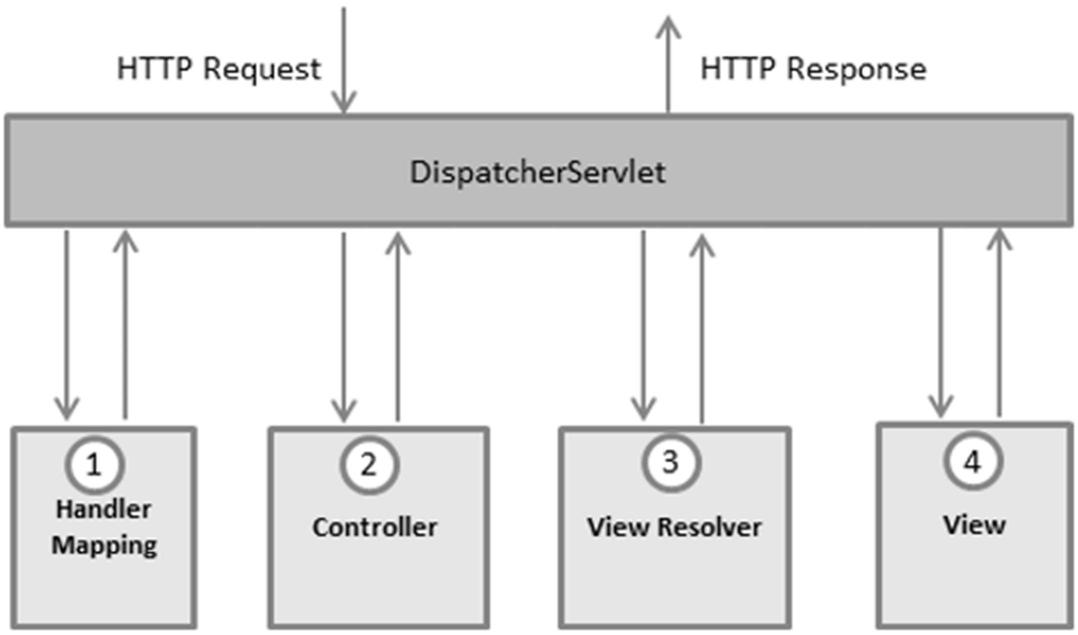
Spring - MVC Framework:

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram –



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* –

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

Required Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet* example –

```
<web-app>


  <display-name>Spring MVC Application</display-name>


  <servlet>
    <servlet-name>HelloWeb</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>


  <servlet-mapping>
    <servlet-name>HelloWeb</servlet-name>
    <url-pattern>*.jsp</url-pattern>
  </servlet-mapping>


</web-app>
```

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INFdirectory. In this case, our file will be **HelloWebservlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory –

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:context = "http://www.springframework.org/schema/context"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan base-package = "com " />

  <bean class =
"org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
  </bean>

</beans>
```

Defining a Controller

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller**annotation indicates that a particular class serves the role of a controller. The **@RequestMapping**annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
@RequestMapping("/hello")
```

```
public class HelloController {

  @RequestMapping(method = RequestMethod.GET)

  public String printHello(ModelMap model) {

    model.addAttribute("message", "Hello Spring MVC Framework!");

    return "hello";

  }

}
```

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation**@RequestMapping(method = RequestMethod.GET)** is used to declare theprintHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in *@RequestMapping* as follows –

```
@Controller

public class HelloController {

  @RequestMapping(value = "/hello", method = RequestMethod.GET)

  public String printHello(ModelMap model) {

    model.addAttribute("message", "Hello Spring MVC Framework!");

    return "hello";

  }

}
```

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above –

- You will define required business logic inside a service method. You can call another method inside this method as per requirement.
- Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".
- A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp –

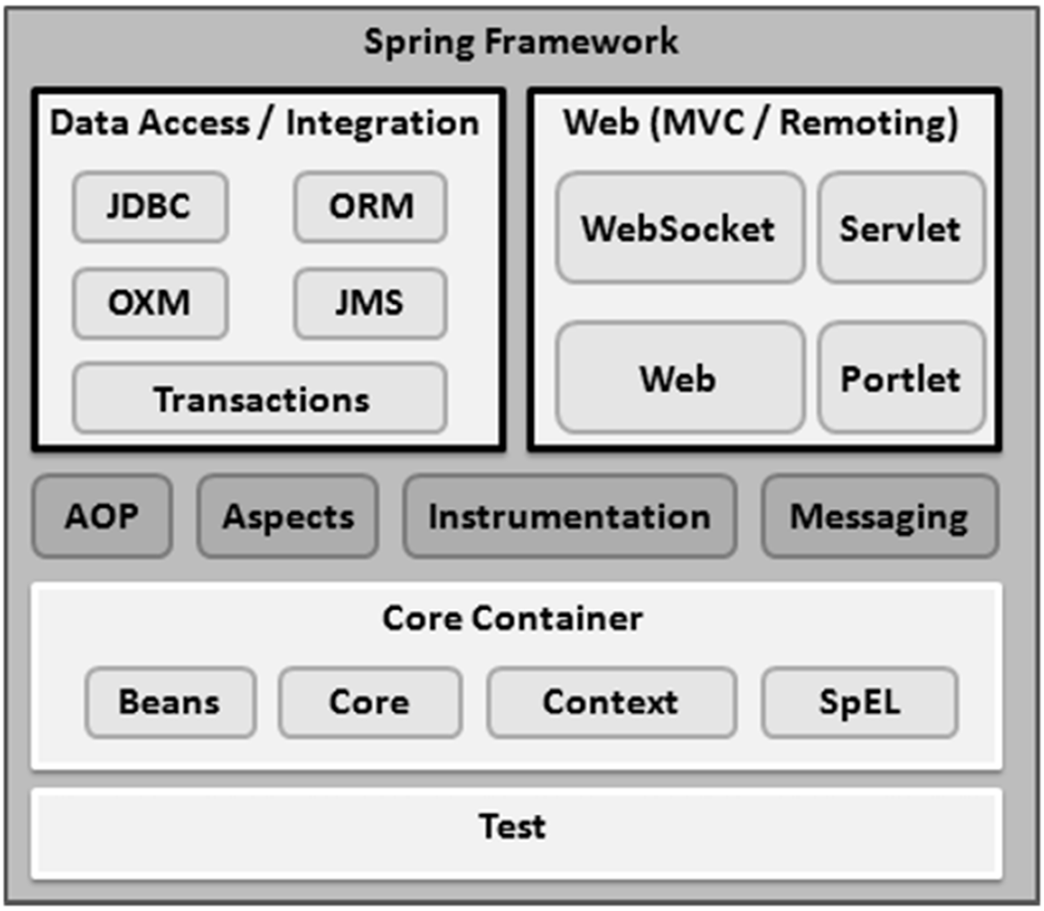
```
<html>
  <head>
    <title>Hello Spring MVC</title>
  </head>

  <body>
    <h2>${message}</h2>
  </body>
</html>
```

Here **${message}** is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

Spring Architecture:

Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows –

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.

- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows –

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows –

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.

- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows –

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

Advantages of Spring Framework Architecture:

There are many advantages of Spring Framework. They are as follows:

- **Lightweight:** Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive, Enable you to write powerful, scalable applications using POJOs.
- **Easy to develop JavaEE application:** The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

- **Easy to test:** The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.
- **Loose Coupling:** The Spring applications are loosely coupled because of dependency injection and handles injecting dependent components without a component knowing where they came from (IoC).
- **Powerful abstraction:** It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.
- **Declarative support:** It provides declarative support for caching, validation, transactions and formatting.
- **Portable :**can use server-side in web/ejb app, client-side in swing app, business logic is completely portable.
- **Cross-cutting behavior :** Resource Management is cross-cutting concern, easy to copy-and-paste everywhere.

Spring:bean life cycle

- The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.
- Though, there is lists of the activities that take place behind the scenes between the time of bean Instantiation and its destruction, here only two important bean lifecycle callback methods which are required at the time of bean initialization and its destruction.
- To define setup and teardown for a bean, we simply declare the <bean>
- with init-method and/or destroy-method parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

Initialization callbacks:

- The *org.springframework.beans.factory.InitializingBean* interface specifies a single method:

void afterPropertiesSet() throws Exception;

- So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

```
public class ExampleBean implements InitializingBean
 {
public void afterPropertiesSet() {
 // do some initialization work
} }
```

- In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example:

```
<bean id="exampleBean" class="com.ExampleBean" init-method="init"/>
```

- Following is the class definition:

```
public class ExampleBean {
 public void init()
 { // do some initialization work } }
```

- Destruction callbacks
- The *org.springframework.beans.factory.DisposableBean* interface specifies a single method:

```
  void destroy() throws Exception;
```

- So you can simply implement above interface and finalization work can be done inside destroy() method as follows:

```
public class ExampleBean implements DisposableBean {
public void destroy() {
// do some destruction work
 } }
```

- In the case of XML-based configuration metadata, you can use the **destroy -method** attribute to specify the name of the method that has a void no-argument signature. For example:

```
<bean id="exampleBean" class="com.ExampleBean" destroy-method="destroy"/>
```

Following is the class definition:

```
public class ExampleBean {
 public void destroy() {
// do some destruction work } }
```

Example:

Simple Spring Hello World Application & Handling Bean Life Cycle

1) Create a Java Bean – POJO (HelloWorld.java) in com package

```
package com;

public class HelloWorld
{
  private String message;

  public void setMessage(String message)
  {
    this.message  = message;
  }

  public void getMessage()
  {
    System.out.println("Your Message : " + message);
  }
  public void init() //Bean Life Cycle
  {
     System.out.println("Calling Init Method");
  }
  public void destroy()    //Bean Life Cycle
  {
     System.out.println("Bean is Destoyed");
  }
}
```

2) Create a class to Inject Dependency (MainApp.java) in com package

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

public class MainApp {
  public static void main(String[] args) {
    AbstractApplicationContext context = new
ClassPathXmlApplicationContext("Bean.xml");

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

    obj.getMessage();

    context.registerShutdownHook();
  }
}
```

3) Create XML Mapping File (Bean.xml) in source package (Click on source package and select New ,select Spring xml configuration file

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

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

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

     xsi:schemaLocation=http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd>

   <bean id="helloWorld" class="com.HelloWorld" init-method="init" destroy-
method="destroy">

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

  </bean>

</beans>
```

XML Configuration on Spring :

The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following –

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

Sr.No.	Properties & Description
1	**class** This attribute is mandatory and specifies the bean class to be used to create the bean.
2	**name** This attribute specifies the bean identifier uniquely. In XMLbased configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
3	**scope** This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
4	**constructor-arg** This is used to inject the dependencies and will be discussed in subsequent chapters.
5	**properties** This is used to inject the dependencies and will be discussed in subsequent chapters.

6	**autowiring mode** This is used to inject the dependencies and will be discussed in subsequent chapters.
7	**lazy-initialization mode** A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup.
8	**initialization method** A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter.
9	**destruction method** A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. Following are the three important methods to provide configuration metadata to the Spring Container –

- XML based configuration file.
- Annotation-based configuration
- Java-based configuration

Example:

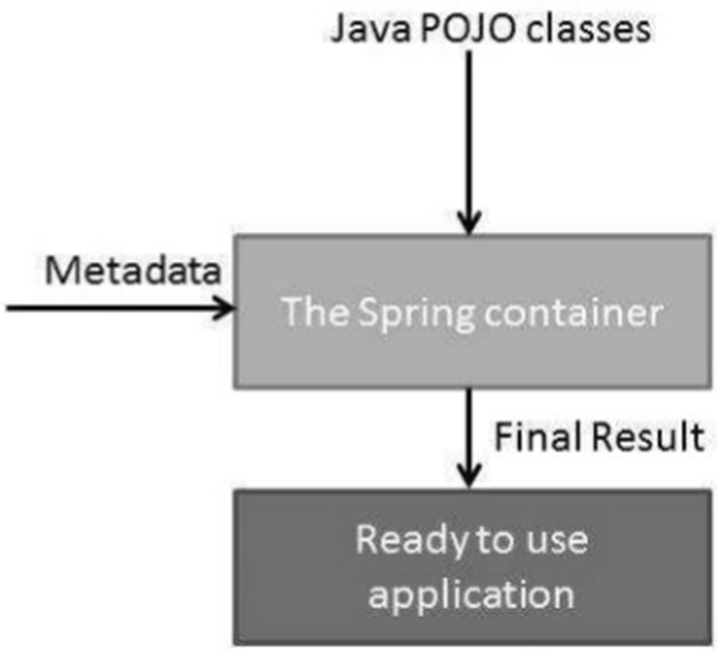
```
<?xml version="1.0" encoding="UTF-8"?>

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

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

    xsi:schemaLocation=http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd>
```

```
   <bean id="helloWorld" class="com.HelloWorld" init-method="init" destroy-
method="destroy">

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

  </bean>

</beans>
```

Spring - IoC Containers

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides the following two distinct types of containers.

Sr.No.	Container & Description
1	**Spring BeanFactory Container** This is the simplest container providing the basic support for DI and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.
2	**Spring ApplicationContext Container** This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface.

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

Spring - Dependency Injection:

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this –

```
public class TextEditor {
  private SpellChecker spellChecker;


  public TextEditor() {
    spellChecker = new SpellChecker();
  }
}
```

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this –

```
public class TextEditor {
  private SpellChecker spellChecker;


  public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }
}
```

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus the flow of control has been "inverted" by Dependency

Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

Thus, DI exists in two major variants and the following two sub-chapters will cover both of them with examples –

Sr.No.	Dependency Injection Type & Description
1	**Constructor-based dependency injection** Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.
2	**Setter-based dependency injection** Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.

Constructor based Dependency example:

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

Example

Here is the content of **TextEditor.java** file – in com package

```
package com;
```

```
public class TextEditor {
  private SpellChecker spellChecker;
```

```

  public TextEditor(SpellChecker spellChecker) {
    System.out.println("Inside TextEditor constructor." );
    this.spellChecker = spellChecker;
  }
  public void spellCheck() {
    spellChecker.checkSpelling();
  }
}
```

Following is the content of another dependent class file **SpellChecker.java in com package**

```
package com;


public class SpellChecker {
  public SpellChecker(){
    System.out.println("Inside SpellChecker constructor." );
  }
  public void checkSpelling() {
    System.out.println("Inside checkSpelling." );
  }
}
```

Following is the content of the **MainApp.java** file in com package

```
package com;
```

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;


public class MainApp {

   public static void main(String[] args) {

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


      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();

   }

}
```

Following is the configuration file **Beans.xml** which has configuration for the constructor-based injection –

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


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

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

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

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


   <!-- Definition for textEditor bean -->

   <bean id = "textEditor" class = "com.TextEditor">

      <constructor-arg ref = "spellChecker"/>
```

```
  </bean>
```

```
  <!-- Definition for spellChecker bean -->
  <bean id = "spellChecker" class = "com.SpellChecker"></bean>


</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

```
Inside SpellChecker constructor.
Inside TextEditor constructor.
Inside checkSpelling.
```

Setter based dependency Injection:

Example:

Here is the content of **TextEditor.java** file – in com package

```
package com;


public class TextEditor {
  private SpellChecker spellChecker;


  // a setter method to inject the dependency.
  public void setSpellChecker(SpellChecker spellChecker) {
    System.out.println("Inside setSpellChecker." );
    this.spellChecker = spellChecker;
  }
  // a getter method to return spellChecker
  public SpellChecker getSpellChecker() {
    return spellChecker;
  }
  public void spellCheck() {
    spellChecker.checkSpelling();
  }
}
```

Here you need to check the naming convention of the setter methods. To set a variable **spellChecker** we are using **setSpellChecker()** method which is very similar to Java POJO classes. Let us create the content of another dependent class file **SpellChecker.java** – in com package

```
package com;


public class SpellChecker {
  public SpellChecker(){
    System.out.println("Inside SpellChecker constructor." );
  }
  public void checkSpelling() {
    System.out.println("Inside checkSpelling." );
  }
}
```

Following is the content of the **MainApp.java** file – in com package

```
package com;
```

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;


public class MainApp {

   public static void main(String[] args) {

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


      TextEditor te = (TextEditor) context.getBean("textEditor");

      te.spellCheck();

   }

}
```

Following is the configuration file **Beans.xml** which has configuration for the setter-based injection – in Source package

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


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

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

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

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


   <!-- Definition for textEditor bean -->

   <bean id = "textEditor" class = "com.TextEditor">

      <property name = "spellChecker" ref = "spellChecker"/>
```

```
  </bean>


  <!-- Definition for spellChecker bean -->

  <bean id = "spellChecker" class = "com.SpellChecker"></bean>


</beans>
```

You should note the difference in Beans.xml file defined in the constructor-based injection and the setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

The second important point to note is that in case you are passing a reference to an object, you need to use **ref** attribute of <property> tag and if you are passing a **value** directly then you should use value attribute.

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message –

```
Inside SpellChecker constructor.
Inside setSpellChecker.
Inside checkSpelling.
```

XML Configuration using p-namespace

If you have many setter methods, then it is convenient to use **p-namespace**in the XML configuration file. Let us check the difference –

Let us consider the example of a standard XML configuration file with <property> tags –

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


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

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

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

  <bean id = "john-classic" class = "com.example.Person">
    <property name = "name" value = "John Doe"/>
    <property name = "spouse" ref = "jane"/>
  </bean>

  <bean name = "jane" class = "com.example.Person">
    <property name = "name" value = "John Doe"/>
  </bean>

</beans>
```

The above XML configuration can be re-written in a cleaner way using p-namespace as follows –

```
<?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 = "john-classic" class = "com.example.Person"
    p:name = "John Doe"
    p:spouse-ref = "jane"/>
  </bean>


  <bean name =" jane" class = "com.example.Person"
    p:name = "John Doe"/>
  </bean>


</beans>
```

Here, you should note the difference in specifying primitive values and object references with p-namespace. The **-ref** part indicates that this is not a straight value but rather a reference to another bean.

Aspect – oriented Spring :

Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

```
class A{
public void m1(){...}
public void m2(){...}
public void m3(){...}
public void m4(){...}
public void m5(){...}
public void n1(){...}
public void n2(){...}
public void p1(){...}
public void p2(){...}
public void p3(){...}
}
```

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.

Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

Where use AOP?

AOP is mostly used in following cases:

- to provide declarative enterprise services such as declarative transaction management.
- It allows users to implement custom aspects.

AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

- Join point
- Advice
- Pointcut
- Introduction
- Target Object
- Aspect
- Interceptor
- AOP Proxy
- Weaving

Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

- **Before Advice**: it executes before a join point.
- **After Returning Advice**: it executes after a joint point completes normally.
- **After Throwing Advice**: it executes if method exits by throwing an exception.
- **After (finally) Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return.
- **Around Advice**: It executes before and after a join point.

Pointcut

It is an expression language of AOP that matches join points.

Introduction

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

Target Object

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

Aspect

It is a class that contains advices, joinpoints etc.

Interceptor

It is an aspect that contains only one advice.

AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

AOP Implementations

AOP implementations are provided by:

1. AspectJ
2. Spring AOP
3. JBoss AOP

Spring AOP AspectJ Annotation Example

The **Spring Framework** recommends you to use **Spring AspectJ AOP implementation** over the Spring 1.2 old style dtd based AOP implementation because it provides you more control and it is easy to use.

There are two ways to use Spring AOP AspectJ implementation:

1. By annotation: We are going to learn it here.
2. By xml configuration (schema based): We will learn it in next page.

Spring AspectJ AOP implementation provides many annotations:

1. **@Aspect** declares the class as aspect.
2. **@Pointcut** declares the pointcut expression.

The annotations used to create advices are given below:

1. **@Before** declares the before advice. It is applied before calling the actual method.
2. **@After** declares the after advice. It is applied after calling the actual method and before returning result.
3. **@AfterReturning** declares the after returning advice. It is applied after calling the actual method and before returning result. But you can get the result value in the advice.
4. **@Around** declares the around advice. It is applied before and after calling the actual method.
5. **@AfterThrowing** declares the throws advice. It is applied if actual method throws exception.

Understanding Pointcut

Pointcut is an expression language of Spring AOP.

The **@Pointcut** annotation is used to define the pointcut. We can refer the pointcut expression by name also. Let's see the simple example of pointcut expression.

```
1. @Pointcut("execution(* Operation.*(..))")
2. private void doSomething() {}
```

The name of the pointcut expression is doSomething(). It will be applied on all the methods of Operation class regardless of return type.

@Before Example

The AspectJ Before Advice is applied before the actual business logic method. You can perform any operation here such as conversion, authentication etc.

Create a class that contains actual business logic.

File: Operation.java

```
  package com;
public  class Operation{

   public void msg(){

System.out.println("msg method invoked");
}
   public int m(){
System.out.println("m method invoked");
return 2;
}
   public int k(){
System.out.println("k method invoked");
return 3;
}
}
```

Now, create the aspect class that contains before advice.

File: TrackOperation.java

```
1. package com;
```

```

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TrackOperation{
   @Pointcut("execution(* Operation.*(..))")
   public void k(){}//pointcut name

   @Before("k()")//applying pointcut on before advice
   public void myadvice(JoinPoint jp)//it is advice (before advice)
   {
      System.out.println("additional concern");
      //System.out.println("Method Signature: "  + jp.getSignature());
   }
}
```

Now create the applicationContext.xml file that defines beans.

File: applicationContext.xml

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


   <bean id="opBean" class="com.Operation">   </bean>
   <bean id="trackMyBean" class="com.TrackOperation"></bean>

   <bean class="org.springframework.aop.aspectj.annotation.AnnotationAware
AspectJAutoProxyCreator"></bean>

</beans>
```

Now, let's call the actual method.

File: Test.java

```
package com;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test{
    public static void main(String[] args){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        Operation e = (Operation) context.getBean("opBean");
        System.out.println("calling msg...");
        e.msg();
        System.out.println("calling m...");
        e.m();
        System.out.println("calling k...");
        e.k();
    }
}
```

Output

```
calling msg...
additional concern
msg() method invoked
calling m...
additional concern
m() method invoked
calling k...
additional concern
k() method invoked
```

Managing Database: (Extra topic less important)

JdbcTemplate Class

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the *JdbcTemplate* class are *threadsafe* once configured. So you can configure a single instance of a *JdbcTemplate* and then safely inject this shared reference into multiple DAOs.

A common practice when using the JDBC Template class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

Configuring Data Source

Let us create a database table **Student** in our database **TEST**. We assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown in the following code snippet –

```
<bean id = "dataSource"
   class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
```

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

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

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

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

</bean>
```

Data Access Object (DAO)

DAO stands for Data Access Object, which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

Spring - Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as **ACID** –

- **Atomicity** – A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** – This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** – There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.

- **Durability** – Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows –

- Begin the transaction using *begin transaction* command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform *commit* otherwise *rollback* all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

Local vs. Global Transactions

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case, transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic vs. Declarative

Spring supports two types of transaction management –

- Programmatic transaction management – This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
- Declarative transaction management – This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Sr.No	Method & Description
1	**TransactionStatus getTransaction(TransactionDefinition definition)** This method returns a currently active transaction or creates a new one, according to the specified propagation behavior.
2	**void commit(TransactionStatus status)** This method commits the given transaction, with regard to its status.
3	**void rollback(TransactionStatus status)** This method performs a rollback of the given transaction.

Spring MVC Login Example

We can simply create login application by following the Spring MVC. We need to pass HttpServletRequest and HttpServletResponse objects in the request processing method of the Controller class. Let's see the example:

1) Controller Class :Create following controller in com package.

HelloWorldController.java

```
package com;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HelloWorldController {

   @RequestMapping("/hello")
   public ModelAndView helloWorld(HttpServletRequest request,HttpServletR
esponse res) {
        String name=request.getParameter("name");
        String password=request.getParameter("password");

        if(password.equals("admin")){
        String message = "HELLO "+name;
        return new ModelAndView("hellopage", "message", message);
        }
        else{
            return new ModelAndView("errorpage", "message","Sorry, username o
r password error");
        }
   }
     }
```

2) View components

To run this example, It must be located inside the **WEB-INF/jsp** directory.

hellopage.jsp

```
Message is: ${message}
```

errorpage.jsp

```
${message}
<jsp:include page="/index.jsp"></jsp:include>
```

3) Index page

It is the login page, that receive name and password from the user.

index.jsp

```
<form action="hello.html" method="post">
Name:<input type="text" name="name"/><br/>
Password:<input type="password" name="password"/><br/>
<input type="submit" value="login"/>
</form>
```

Note: Advance Java having 4 frameworks, Only spring framework can work without webserver.

THANK YOU