**Spring *WEB-MVC***

***Search for index.jsp in main/webapp***

***Create java file inside main/java/p1/AddController***

***Create Spring file One-servlet.xml inside Web-INF***

***Create config file main/resources/Spconfig.xml***

Pom.xml **EX 20**

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.8</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.8</version>

</dependency>

**For Error:** **Could not initialize class org.apache.maven.plugin.war.util.WebappStructureSerializer**

<build>

<finalName>MabenWebPro</finalName>

<plugins>

<plugin>

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

<artifactId>maven-war-plugin</artifactId>

<version>3.3.1</version>

</plugin>

</plugins>

</build>

</project>

Index.jsp

<h2>Hello World!</h2>

<form action=*"add"*>

<input type=*"text"* name=*"t1"*>

<input type=*"text"* name=*"t2"*>

<input type=*"submit"* value=*"Click Me"*>

</form>

Web.xml

<servlet>

<servlet-name>one</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>one</servlet-name>

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

</servlet-mapping>

**Now to convert our class into controller we have this controller classes this are called Programmed Approaches**

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

**Or Else we can implement our App by using Stereotype Approaches Annotation**

**@Controller**

AddController.java

@Controller

**public** **class** AddController {

@RequestMapping(value="/add")

**public** ModelAndView add(HttpServletRequest request,HttpServletResponse response){

**int** a = Integer.*parseInt*(request.getParameter("t1"));

**int** b = Integer.*parseInt*(request.getParameter("t2"));

**int** c = a + b;

ModelAndView mv = **new** ModelAndView();

mv.setViewName("success");which file to call

mv.addObject("result",c);

**return** mv;

}

}

One-servlet.xml

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

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

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.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc.xsd*

*http://www.springframework.org/schema/context*

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

<context:annotation-config></context:annotation-config>

<context:component-scan base-package=*"p1"*></context:component-scan>

**We are just passing name like successs on above suppose we have to page success.html and success.jsp which to call and where to search with the help of this it will come to**

**Create a Success.jsp inside *webapp/OutPut/Success.jsp***

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/OutPut/"*></property>

<property name=*"suffix"* value=*".jsp"*></property>

</bean>

</beans>

Success.jsp

Result is <%= request.getAttribute("result") %>

**@RequestParam** **EX 22**

***Just make changes is AddController.java***

@Controller

public class AddController {

@**RequestMapping**(value="/add")

public ModelAndView add(**@RequestParam("t1")int i,@RequestParam("t2")int j,**HttpServletRequest request,HttpServletResponse response){

int c = i + j;

ModelAndView mv = new ModelAndView();

mv.setViewName("success.jsp");

mv.addObject("result",c);

return mv

**@RequestParam(name = "eid",required = false) required = false** means its became an optional parameter

**It can get bind directly also without using @RequestParam and HttpServletRequest**

**@PathVariable**  
IF our URL like this add/SAM add is action name and SAM is value and I want this value when it’s come with URL so we can use @**PathVariable** for that   
**EX....**   
**@RequestMapping**(value = "add/{x}")  
**public** ModelAndView add**(@RequestParam**("t1")**int** a,**@PathVariable**("x")**String** b) {}

**@SessionAttributes**  
@Controller

@SessionAttributes({"result","key"})

**public** **class** MyController {

@RequestMapping(value = "add")

**public** ModelAndView add(@RequestParam("t1")**int** a,@RequestParam("t2")**int** b) {

ModelAndView mv = **new** ModelAndView();

mv.setViewName("Success");

**int** c = a + b;

mv.addObject("result",c);

mv.addObject("key","SAM");

**return** mv;  
Access this session value with <%=session.getAttribute("result") %>

**Now we can remove value from Session Scope**

@RequestMapping(value = "del",method = RequestMethod.***GET***)

**public** ModelAndView deleteSession(WebRequest request,SessionStatus status) {

status.setComplete();

request.removeAttribute("eid", WebRequest.***SCOPE\_SESSION***);

request.removeAttribute("eName",WebRequest.***SCOPE\_SESSION***);

ModelAndView mv = **new** ModelAndView();

mv.setViewName("Success");

**return** mv;

***If it show BAD REQUEST change ../del in anchor tag***

***status.setComplete()***

The method **setComplete()** removes all the attributes marked with **@SessionAtributes**. If you want to keep a session attribute, you must add it to the **HttpSession**

**@PropertySource**Used to read properties file from directly from classpath means

**it should be under src/main/resource/some.properties  
EX....** create file db.properties with entries ......username=root  
password=sam **@PropertySource**("classpath:db.properties") if our file is not in classpath somewhere in different location we can do

file :db.properties **Enter Location  
@Value**("${username}") //  
String test = **null**;  
  
**Another way......**  
@Autowired  
Environment env;  
**Access** >>> System.out.println(env.getProperty("username"));

**Another way......**

**@ConfigurationProperties** on top of our POJO class

If we want to bind all properties of property file into POJO class we can use this to test this we required some Spring boot dependency

**Update for @Controller, @RestController, @RequestBody, @ResponseBody**

whenever we are using **@Controller** in our class if we want to produce response in **JSON** format then we have to use **@ResponseBody** like this  
  
 **@RequestMapping**(value = "add",method = RequestMethod.***POST***)

**public** **@ResponseBody Student** add(@RequestBody Student s) {

System.***out***.println("S >>>>>>>>> " +s);

**return** s;

}  
but from Spring-4 we can use **@RestController** it will directly produce response in **JSON** format we don’t need to use **@ResponseBody** in return type  
  
**@RequestMapping**(value = "add",method = RequestMethod.***POST***)

**public** **Student** add(@RequestBody Student s) {

System.***out***.println("S >>>>>>>>> " +s);

**return** s;

}  
**Example of Rest API in Spring**  
create a class **Student** with variable **t1** and **t2** both String  
Pom.xml

<dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.8</version>

</dependency>

one-servlet.xml or Spring.xml  
**<mvc:annotation-driven /> annotation of this @EnableWebMvc**

MyController.java  
**@RestController**

**public** **class** MyController {

@RequestMapping(value = "add",method = RequestMethod.***POST***)

**public** Student add(@RequestBody Student s) {

System.***out***.println("S >>>>>>>>> " +s);

**return** s;

}

}  
Use POSTMAN to hit select ROW and JSON(application/json) under body tab  
**{  
“t1”:”1”,  
“t2”:”2”  
}**  
t1 and t2 should be same as we defined in Student class  
FULL VIEW OF one-servlet.xml or Spring.xml if NEEDED  
  
<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

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.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc.xsd*

*http://www.springframework.org/schema/context*

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

<ctx:annotation-config></ctx:annotation-config>

<ctx:component-scan base-package=*"p1"*></ctx:component-scan>

<mvc:annotation-driven />

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"/output/"*></property>

<property name=*"suffix"* value=*".jsp"*></property>

</bean>

</beans>

**What if our method return void** in Controller then we cannot return **@ResponseBody**

**@RequestMapping**(**value** = "/updateSomeData" **method** = RequestMethod.POST)

**@ResponseStatus**(**value** = HttpStatus.OK)

**public** **void** updateDataThatDoesntRequireClientToBeNotified(...)

you can return void, then you have to mark the method **@ResponseStatus(value = HttpStatus.OK)** you don't need **@ResponseBody**

**Return response as XML**

**Pom.xml**

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

<version>2.9.8</version>

</dependency>

**Employee.java**

@JacksonXmlRootElement ***on top of class***

**EmployeeController.java**

@RequestMapping(value = "addEmployee",produces = MediaType.***APPLICATION\_XML\_VALUE***)

Multiple also we can return we need to pass Array in produces

**Return response as XML and JSON on single method**

**One-servlet.xml**

<mvc:annotation-driven content-negotiation-manager=*"contentNegotiationManager"*></mvc:annotation-driven>

<bean id=*"contentNegotiationManager"*

class=*"org.springframework.web.accept.ContentNegotiationManagerFactoryBean"*>

<property name=*"favorParameter"* value=*"true"* />

<property name=*"parameterName"* value=*"type"* />

<property name=*"mediaTypes"*>

<map>

<entry key=*"json"* value=*"application/json"*></entry>

<entry key=*"xml"* value=*"application/xml"* />

</map>

</property>

</bean>

**URLs Call**

[http://localhost:8080/SpringMavenWeb/addEmployee?type=***xml***](http://localhost:8080/SpringMavenWeb/addEmployee?type=xml) ***OR json***

**EmployeeController.java**

@RequestMapping(value = "addEmployee",method = RequestMethod.***POST***,

produces = {MediaType.***APPLICATION\_JSON\_VALUE***, MediaType.***APPLICATION\_XML\_VALUE***})

**@ModelAttribute**

But **@RestController** and **@RequestBody** will bind data whenever we send JSON Request so If we want to bind data of HTML or JSP form in our Object we have to use something Called... **@ModelAttribute**

@RequestMapping(value = "getStudent", method = RequestMethod.***POST***)

**public** **void** getStudent(@ModelAttribute("student") Student s) {

System.***out***.println("Called " + s);

}

student in ROUND BRACKET is same as mv.addObject("student",s); or else

mv.addObject("msg","Welcome to Spring MVC"); to Call in JSP <%=request.getAttribute("msg")%>

Now **@ModelAttribute** at method level... Suppose I have two methods in which I added same

mv.addObject("msg","Welcome to Spring MVC");

EX....

@RequestMapping(value = "getStudent", method = RequestMethod.***POST***)

**public** ModelAndView getStudent(@ModelAttribute("student") Student s) {

ModelAndView mv = **new** ModelAndView();

mv.addObject("msg","Welcome to Spring MVC");

mv.setViewName("Success");

**return** mv;

}

@RequestMapping(value = "getTeacherData", method = RequestMethod.***POST***)

**public** ModelAndView getTeacherData() {

ModelAndView mv = **new** ModelAndView();

mv.addObject("msg","Welcome to Spring MVC");

mv.setViewName("Success");

**return** mv;

}

So, it will lead to some extra code instead of this we can do like this

@ModelAttribute

**public** **void** addCommanMsg(Model m) {

m.addAttribute("msg","Welcome to Spring MVC");

}

So, whenever we have method like this Spring MVC will call this method first before calling any request

NOW

If we want @ModelAttribute should bind data in our class of another data type Like List, Long etc

Add this in Student.jsp

<tr><td>Course Type<td><td>

<select name=*"course"* multiple=*"multiple"*>

<option value=*"Java"*>Java</option>

<option value=*".NET"*>.NET</option>

<option value=*"PHP"*>PHP</option>

<option value=*"C++"*>C++</option>

</select>

<td></tr>

Student.java

List<String> course; Getter and Setter and call it from Controller......

Now Suppose we have one more class Called Address.java

String city;String state;String zipCode; Getter and Setter

And Ref available in Student.java like Address add Getter and Setter

***Ex....***

Address.java

String city;

String state;

String zipCode;

Student.java

**int** sid;

String sName;

**int** roll;

List<String> course;

Address add;

Student.jsp

<tr><td>Address Details</td></tr>

<tr><td>City</td><td><input type=*"text"* name=*"add.city"*></td></tr>

<tr><td>State</td><td><input type=*"text"* name=*"add.state"*></td></tr>

<tr><td>ZipCode</td><td><input type=*"text"* name=*"add.zipCode"*></td></tr>

Success.jsp // **Enable Expression language** <%@ page isELIgnored=*"false"* %>

<h1>${msg}</h1>

<table>

<tr><td>${msg}</td></tr>

<tr><td>${student.sid}</td></tr> <!-- student is key defined in @ModelAttribute("student") -->

<tr><td>${student.sName}</td></tr>

<tr><td>${student.roll}</td></tr>

<tr><td>${student.course}</td></tr>

<tr><td>Address Details</td></tr>

<tr><td>${student.add.city}</td></tr>

<tr><td>${student.add.state}</td></tr>

<tr><td>${student.add.zipCode}</td></tr>

</table>

**@InitBinder**

So, by Default @ModelAttribute("student") will bind all data in Model class i.e., Student but I want name should not get banded so for that we have to use something called **@InitBinder**

@InitBinder

**public** **void** initBinder(WebDataBinder binder) {

binder.setDisallowedFields(**new** String[] {"sName"});

}

Method name can be anything but Annotation and parameter must be there.......

So now Suppose we have one more field called Date in our Student.jsp and Student.java

Ex....

Student.jsp <tr><td>Date</td><td><input type=*"text"* name=*"date"*></td></tr>

Student.java Date date; from java.util and Getter and Setter.....

Success.jsp <tr><td>${student.date}</td></tr>

So by Default it will accept date like this 10/10/2010 but I want it should accept date like this

10\*\*10\*\*2010 in My Custom Format

@InitBinder

**public** **void** initBinder(WebDataBinder binder){

binder.setDisallowedFields(**new** String[] {"sName"});

SimpleDateFormat dateFormat = **new** SimpleDateFormat("dd\*\*mm\*\*yyyy");

binder.registerCustomEditor(Date.**class**, "date",**new** CustomDateEditor(dateFormat,**false**));

}

CustomDateEditor we called this classes are property editor class....

Like we have so many predefined Custom Editor Ex... CustomDateEditor,ClassEditor,FileEditor,

CustomNumberEditor or else we can create our OWN also...

Ex..... (If we do not enter Mr or Ms in name field it will say NOT FOUND!)

StudentNameEditor.java

**import** java.beans.PropertyEditorSupport;

**public** **class** StudentNameEditor **extends** PropertyEditorSupport{

@Override

**public** **void** setAsText(String studentName) **throws** IllegalArgumentException {

**if**(studentName.contains("Mr") || studentName.contains("Ms")) {

setValue(studentName);

}**else** {

studentName = "Not Found ";

setValue(studentName);

@InitBinder

**public** **void** initBinder(WebDataBinder binder){

binder.setDisallowedFields(**new** String[] {"sName"});

SimpleDateFormat dateFormat = **new** SimpleDateFormat("dd\*\*mm\*\*yyyy");

binder.registerCustomEditor(Date.**class**, "date",**new** CustomDateEditor(dateFormat,**false**));

binder.registerCustomEditor(String.**class**, "sName",**new** StudentNameEditor());

}

@RequestMapping(value = "getStudent", method = RequestMethod.***POST***)

**public** ModelAndView getStudent(@ModelAttribute("student") Student s,BindingResult result) {

ModelAndView mv = **new** ModelAndView();

**if**(result.hasErrors()) {

mv.setViewName("../Student");

**return** mv;

}

mv.setViewName("Success");

**return** mv;

}

**@ExceptionHandler**

Use to handle Exception in Spring, **now if sName is empty it should throw exceptions**

Output/Error.jsp

<%= (String)request.getAttribute("error") %>

InvalidFieldException.java

**public** **class** InvalidFieldException **extends** RuntimeException{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

String message; **//Getter Setter and Constructor**

}

StudentController.java

@RequestMapping(value = "getStudent", method = RequestMethod.***POST***)

**public** ModelAndView getStudent(@ModelAttribute("student") Student s,BindingResult result) {

ModelAndView mv = **new** ModelAndView();

**if**(e.geteName() == **null** || e.geteName().equals("")) {

**throw** **new** InvalidFieldException("Field are Invalid");

}

**return** mv;

}

@ExceptionHandler

**public** ModelAndView handleInvalidField(InvalidFieldException exception) {

ModelAndView mv = **new** ModelAndView("Error");

mv.addObject("error", exception.getMessage());

**return** mv;

}

**@ControllerAdvice**

Now the above approach for exception is for StudentController only so if we want something which is global for all like instead of having separate handleInvalidField method for every controller we should make it generic

GenericHandler.java ***Remove below method from Controller class***

@ControllerAdvice

**public** **class** GenericHandler {

@ExceptionHandler

**public** ModelAndView handleInvalidField(InvalidFieldException exception) {

ModelAndView mv = **new** ModelAndView("Error");

mv.addObject("error", exception.getMessage());

**return** mv;

}

}

Now if you hit your service with **POSTMAN,** you will get JSON String with **timestamp, status, message ETC.** BUT if you want only your exception message in Response so you have to use **@ResponseBody** below **@ExceptionHandler** Annotation

Now if you want to use Single Annotation instead of **@ResponseBody** and **@ControllerAdvice** you can use **@RestControllerAdvice**

**@Conditional and Condition**

Initially spring will create all bean on startup but I wanted to create a bean when IF-Condition get satisfied then only so we can use this.

SchoolIncome.java

@Component

@Conditional(VerifyIncome.**class**)

**public** **class** SchoolIncome {

**public** **void** display() {

System.***out***.println("SchoolIncome is $100000");

}

}

VerifyIncome.java

**public** **class** VerifyIncome **implements** Condition{

**public** **boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

String status = "allow";

**if**(status.equalsIgnoreCase("allow")) {

**return** **true**;

}

**return** **false**;

}

}

StudentController.java

@Autowired(required = **false**)

SchoolIncome com;

@RequestMapping(value = "getStudent", method = RequestMethod.***POST***)

**public** ModelAndView getStudent(@ModelAttribute("student") Student s,BindingResult result) {

ModelAndView mv = **new** ModelAndView();

**if**(**this**.com == **null**) {

System.***out***.println("NULL value Inserted");

}**else** {

**this**.com.display();

}

**return** mv;

}

**Difference between @Profile and @Conditional annotation is that @Profile is restricted to write conditional checking based on predefined properties whereas in @Conditional you can define your custom logic also**

***Spring MVC Configuration without web.xml and*** ***Servlet.xml***

Pom.xml

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

One-Servlet.xml Replace

**import** org.springframework.context.annotation.ComponentScan.Filter;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.context.annotation.FilterType;

**import** org.springframework.web.servlet.ViewResolver;

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

**import** org.springframework.web.servlet.view.InternalResourceViewResolver;

**import** org.springframework.web.servlet.view.JstlView;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "p1")

**public** **class** AppConfig **implements** WebMvcConfigurer {

@Bean

**public** ViewResolver viewResolver() {

InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.**class**);

viewResolver.setPrefix("/OutPut/");

viewResolver.setSuffix(".jsp");

**return** viewResolver;

}

}

**If you want to exclude any annotation**

@ComponentScan(basePackages = "p1",

excludeFilters = {

@Filter(type = FilterType.***ANNOTATION***, value = Configuration.**class**)

})

web.xml Replace

**import** javax.servlet.ServletContext;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletRegistration;

**import** org.springframework.web.WebApplicationInitializer;

**import** org.springframework.web.context.ContextLoaderListener;

**import** org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

**import** org.springframework.web.context.support.GenericWebApplicationContext;

**import** org.springframework.web.servlet.DispatcherServlet;

**public** **class** WebServletConfiguration **implements** WebApplicationInitializer{

**public** **void** **onStartup**(ServletContext servletContext) **throws** ServletException {

AnnotationConfigWebApplicationContext **applicationContext** = **new** AnnotationConfigWebApplicationContext();

**applicationContext**.register(AppConfig.**class**);

**applicationContext**.scan("p1");

servletContext.addListener(**new** ContextLoaderListener(**applicationContext**));

**applicationContext**.setServletContext(servletContext);

ServletRegistration.Dynamic servlet =

servletContext.addServlet("dispatcher", **new** DispatcherServlet(**new** GenericWebApplicationContext()));

servlet.setLoadOnStartup(1);

servlet.addMapping("/");

**}**

**}**

**@Lazy annotation**  
Now it’s time to go a bit deeper. The bean initialization is eager by default. Spring will try to initializeall the beans and wire them all together. You can mark a **@Bean or @Component** with

**@Lazy** to have them be initialized on demand. It can save some startup time!

***1)Create a maven project with QuickStart* Ex 23**

***2)Then copy and paste Spring-context dependency***

***In pom.xml file***

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.3.RELEASE</version>

</dependency>

***Now take one class***

Samsung.class

**public** **class** Samsung {

**public** **void** config(){

System.***out***.println("OctaCore , 12Mp Camera, 4gb Ram");

}

}

App.java

**public** **class** App {

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

Samsung s = **new** Samsung();

s.config();

}

}

***This is how we are doing normally***

***Now for Spring fw we need to create Spring.xml file but write now we are doing via Annotation for that we don’t required that file we just need to one class called AppConfig***

AppConfig.java

@Configuration ***we need to declare this if we are not using Spring config file***

**public** **class** Appconfig {

@Bean ***in Spring.xml we r using bean tag we need to declare like this over here***

**public** Samsung getPhone(){

**return** **new** Samsung();

}

}

Samsung.java

**public** **class** Samsung {

**public** **void** config(){

System.***out***.println("OctaCore , 12Mp Camera, 4gb Ram");

}

}

App.java

**public** **class** App {

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

ApplicationContext fac = **new** AnnotationConfigApplicationContext(Appconfig.**class**);

Samsung s = fac.getBean(Samsung.**class**);

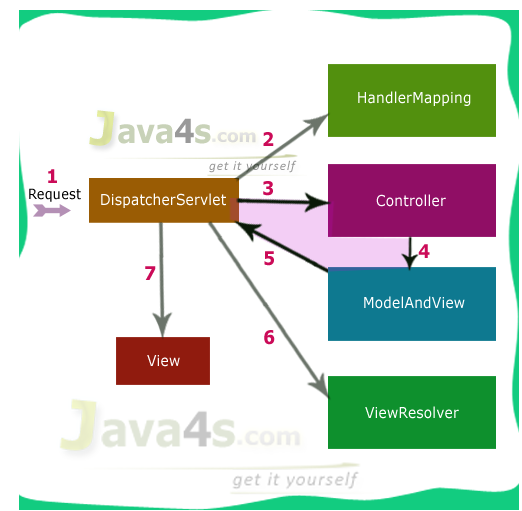
s.config();

}

}

Spring MVC Execution Flow Diagram

Let us see the flow of spring MVC (3.2). I am not going to describe what is MVC :-) hope you already know that mess right? so let’s start with the flow…



## Spring MVC 3.2 Execution Flow

Step **1**: First request will be received by DispatcherServlet

Step **2**: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request HandlerMapping will scan all controller with the help of URL pattern.

We have total 4 handlerMapping class   
1.**BeanNameUrlHandlerMapping**(default)

Suppose if we configure like this in Spring.xml file

***<bean name=”/add” class="p1.AddController"/>***

***The reuest we are getting via URL called add it should handle by AddController***

2.**SimpleUrlHandlerMapping:** whenever we want to map any URL pattern to particular bean id

**<bean class = “SimpleUrlHandlerMapping”>**

**<property>**

**<props>**

**<prop key=”/add”>id of our controller class</prop> some URL Pattern**

**</props>**

**</property>**

**</bean>**

3.**ControllerClassNameURLHandlerMapping** automatically it will find Controller class based on your URL pattern

Our Controller Class name and Url pattern should be same

***<bean class="* ControllerClassNameURLHandlerMapping** ***"/>***

***<bean class="* p1.add** ***"/>***

4.**CommanPathMapHandlerMapping** Not More Use Full

Step **3**: So, request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModeAndView object (contains Model data and View name) back to the DispatcherServlet  
Step **4**: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page  
Step **5**: Finally, DispatcherServlet will pass the Model object to the View page to display the result

That’s it :-)

Just remember this diagram for the interview purpose, i will explain you the practical flow in the first example.