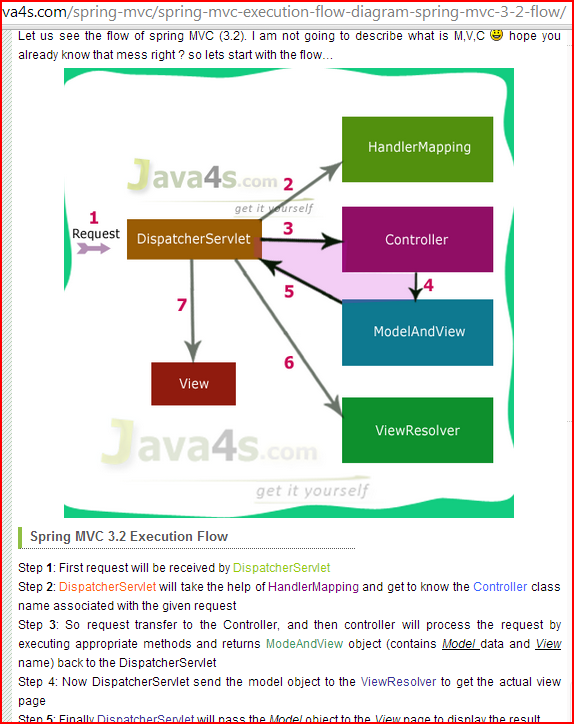
[HelloWorldController](#HelloWorldController)

[ExceptionsHandler](#ExceptionsHandler)

Spring MVC

Java4s.com



<http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/mvc.html>



<http://stackoverflow.com/questions/5055358/need-help-with-undestand-modelandview-in-java-spring>

have this basic function

protected ModelAndView handleRequestInternal (http.. , http..)

{

return new ModelAndView ("welcomePage","WelcomeMessage",message);

}

I know that this will return modelandView. I know that "welcomePage" is my viewname so that means something like welcomepage.jsp will get called.

But I am confused with what is Model part . what is WelcomeMesaage and message and how Model work in that scenario

The model presents a placeholder to hold the information you want to display on the view. It could be a string, which is in your above example, or it could be an object containing bunch of properties.

**Example 1**

If you have...

return new ModelAndView("welcomePage","WelcomeMessage","Welcome!");

... then in your jsp, to display the message, you will do:-

Hello Stranger! ${WelcomeMessage} // displays Hello Stranger! Welcome!

**Example 2**

If you have...

MyBean bean = new MyBean();

bean.setName("Mike");

bean.setMessage("Meow!");

return new ModelAndView("welcomePage","model",bean);

... then in your jsp, you can do:-

Hello ${model.name}! {model.message} // displays Hello Mike! Meow!

the second argument represents the logical name of your model. You can call it whatever name you want, but this is the name you will use when referencing your model in your JSP. In my first example, I called it WelcomeMessage, in the second example, I called it model. Use a name that makes sense to you. The third argument holds the information... it could be a simple string, or a POJO or a map... basically any useful information you want to display in the view

Another explanation:

new ModelAndView("welcomePage", "WelcomeMessage", message);

is shorthand for

ModelAndView mav = new ModelAndView();

mav.setViewName("welcomePage");

mav.addObject("WelcomeMessage", message);

Looking at the code above, you can see the view name is "welcomePage". Your ViewResolver (usually setup in .../WEB-INF/spring-servlet.xml) will translate this into a View. The last line of the code sets an attribute in your model (addObject("WelcomeMessage", message)). That's where the model comes into play.

*File :* *HelloWorldController.java*

**package** com.mkyong.common.controller;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

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

**import** org.springframework.web.servlet.mvc.AbstractController;

**public** **class** HelloWorldController **extends** AbstractController{

@Override

**protected** ModelAndView handleRequestInternal(HttpServletRequest request,

HttpServletResponse response) **throws** Exception {

ModelAndView model = **new** ModelAndView("HelloWorldPage");

model.addObject("msg", "hello world");

**return** model;

}

}

1. **ModelAndView(“HelloWorldPage”)** – The “HelloWorldPage” will pass to Spring’s viewResolver later, to indentify which view should return back to the user. (see step 6)
2. **model.addObject(“msg”, “hello world”)** – Add a “hello world” string into a model named “msg”, later you can use JSP EL ${msg} to display the “hello world” string.

A spring MVC example with handler class.

<http://www.mkyong.com/spring-mvc/spring-mvc-hello-world-example/>

How it works?

1. <http://localhost:8080/SpringMVC/welcome.htm> is requested.
2. URL is end with “.htm” extension, so it will redirect to “**DispatcherServlet**” and send request to the default **BeanNameUrlHandlerMapping**.
3. **BeanNameUrlHandlerMapping** return **HelloWorldController** to the **DispatcherServlet**.
4. **DispatcherServlet** forward request to the **HelloWorldController**.
5. **HelloWorldController** process it and return a **ModelAndView** object named “*HelloWorldPage*”.
6. **DispatcherServlet** received the **ModelAndView** and call the **viewResolver** to process it.
7. **viewResolver** return the *“/WEB-INF/pages/HelloWorldPage.jsp*” back to the **DispatcherServlet**.
8. **DispatcherServlet** return the “HelloWorldPage.jsp” back to user.

**Note**  
Actually, you don’t need to define the **BeanNameUrlHandlerMapping** in the web.xml, by default, if no handler mapping can be found, the DispatcherServlet will creates a **BeanNameUrlHandlerMapping** automatically. See this article – [BeanNameUrlHandlerMapping example](http://www.mkyong.com/spring-mvc/spring-mvc-beannameurlhandlermapping-example/) for detail.

 am getting a bit confused here. In our application we are having a few servlets defined. Here is the excerpt from the web.xml for one of the servlets:

<servlet>

<servlet-name>AxisServlet</servlet-name>

<display-name>Apache-Axis Servlet</display-name>

<servlet-class>com.cisco.framework.axis2.http.FrameworkServlet</servlet-class>

<load-on-startup>0</load-on-startup>

</servlet>

As per my understanding the value for the <load-on-startup> has to be a positive integer in order for it to get loaded automatically. I looked up on google but the responses I came across only added to my confusion.

*Load-on-startup* can specify an (optional) integer value. If the value is 0 or greater, it indicates an order for servlets to be loaded, servlets with higher numbers get loaded after servlets with lower numbers.

# [Difference between Spring MVC and Struts MVC](http://stackoverflow.com/questions/6173009/difference-between-spring-mvc-and-struts-mvc)

The major difference between Spring MVC and Struts is: Spring MVC is loosely coupled framework whereas Struts is tightly coupled. For enterprise Application you need to build your application as loosely coupled as it would make your application more reusable and robust as well as distributed.

Although Spring MVC and Struts are two popular web application frameworks used for developing Java EE web applications, they have their differences. In fact, Spring MVC was developed in order to address few limitations in Struts (version 1). But Struts2 is a highly improved framework than version 1 (they don’t even share the same code base), and therefore, the Spring MVC and Structs2 are highly comparable.

One of the main advantages of Spring MVC is that it is possible to have seamless integration with many view options such as JSP/JSTL, Tiles, FreeMaker, Excel, PDF and JSON. But, unlike Struts, Spring MVC does not provide built-in AJAX support (need to use third-party AJAX library).

Ultimately, both of them are considered highly mature frameworks, and choosing between the two comes down to the personal preference. It is important to note here that if there are any negative feelings towards struts, they are only due to the deficiencies that were found in Struts version 1 (which is now considered obsolete).

Read more: <http://www.differencebetween.com/difference-between-struts-and-vs-spring-mvc/#ixzz326VKYW00>

# [Spring 3 MVC : move/share model from Controler to another Controler](http://stackoverflow.com/questions/5834558/spring-3-mvc-move-share-model-from-controler-to-another-controler)

# <http://stackoverflow.com/questions/5834558/spring-3-mvc-move-share-model-from-controler-to-another-controler>

Here I want User user to be known from last controller,but it's making a new one instead.

@Controller

public class Controller2 {

@RequestMapping(value = "/home")

public String home(ModelMap model, User user) {

...

...

}

LoginPage.jsp

$.get("loginSubmit.html", form1Var.serialize(), function(data){

var isSucess = data.charAt(0) == "1" ? true : false;

if ( isSucess == true) {

alert("ok...");

window.location = "home";

}

My Solution :

@SessionAttributes("user")

on both controllers

and

@ModelAttribute("user") User user

as param in the method - worked

I'v also added

@ExceptionHandler(HttpSessionRequiredException.class)

public String SessionException(HttpSessionRequiredException ex) {

logger.info(ex.getMessage());

return "redirect:LogIn";

}

to catch Exception and the user will go to LoginPage instead of a exception error page

As Donal Boyle pointed , conclusion : use @SessionAttributes to share models between Controllers

### How to handle exceptions in Spring MVC Framework?

Spring MVC Framework provides following ways to help us achieving robust exception handling.

* 1. **Controller Based** – We can define exception handler methods in our controller classes. All we need is to annotate these methods with @ExceptionHandler annotation.
  2. **Global Exception Handler** – Exception Handling is a cross-cutting concern and Spring provides **@ControllerAdvice** annotation that we can use with any class to define our global exception handler.
  3. **HandlerExceptionResolver implementation** – For generic exceptions, most of the times we serve static pages. Spring Framework provides **HandlerExceptionResolver** interface that we can implement to create global exception handler. The reason behind this additional way to define global exception handler is that Spring framework also provides default implementation classes that we can define in our spring bean configuration file to get spring framework exception handling benefits.

For a complete example, please read [Spring Exception Handling Example](http://www.journaldev.com/2651/spring-mvc-exception-handling-exceptionhandler-controlleradvice-handlerexceptionresolver-json-response-example).

@ExceptionHandler(EmployeeNotFoundException.class)

    public ModelAndView handleEmployeeNotFoundException(HttpServletRequest request, Exception ex){

        logger.error("Requested URL="+request.getRequestURL());

        logger.error("Exception Raised="+ex);

        ModelAndView modelAndView = new ModelAndView();

        modelAndView.addObject("exception", ex);

        modelAndView.addObject("url", request.getRequestURL());

        modelAndView.setViewName("error");

        return modelAndView;

    }

}

Spring Restful Web Service Example with JSON, Jackson and Client Program

<http://www.journaldev.com/2552/spring-restful-web-service-example-with-json-jackson-and-client-program>

Building a RESTFul Service with Spring MVC

<http://www.leveluplunch.com/java/tutorials/010-building-restful-webservice-spring-mvc-boot/>

A RESTful service is a way to expose your data through a URL. Let's use spring mvc to implement a RESTFul webservice to retrieve a listing of agencies.

### Detailed Video Notes

## Getting started

SOAP based web service while still serve a purpose are bloated, hard to consume directly from the client side and are bound by a hard contract. REST services on the other hand have been a go to choice for developing services consumed by web applications due to the lightweight JSON responses and soft contract. What I mean by soft contract is, REST services aren’t bound to a WSDL but still have a contract in which you need to abide by or have a versioning strategy in place. Using [**spring boot**](http://projects.spring.io/spring-boot/) as a foundation lets build a simple REST web service.

## What we will build

[[**0:37**](http://www.youtube.com/embed/bdTsY_7SEXw?start=37&autoplay=1)]

While not overly exciting lots of folks are familiar with insurance, agencies sell policies to policyholders. Lets say we want to get all the possible agencies we can purchase a policy from. We will make a request tohttp://localhost:8080/agencies which should return the following:

[

{

"id":1,

"name":"All State",

"ein":"123"

},

{

"id":2,

"name":"FCCI Insurance Group",

"ein":"456"

},

{

"id":3,

"name":"Farmers",

"ein":"789"

},

{

"id":4,

"name":"Met life",

"ein":"167"

}

]

## Project set up

[[**0:54**](http://www.youtube.com/embed/bdTsY_7SEXw?start=54&autoplay=1)]

We will use [**spring STS**](http://spring.io/tools) to create project from a new spring starter project. We will select web project which will include the necessary starter projects in our pom.xml

## Create resource representation class

[[**1:4**](http://www.youtube.com/embed/bdTsY_7SEXw?start=64&autoplay=1)]

Depending on your philosophy of building services, you could return domain object from your business logic layer or create a class that represents the resource a client is requesting. In our case lets create an agency resource object that contains a small subset of an agency object.

public class AgencyResource {

private Integer id;

private String name;

private String EIN;

public AgencyResource(Integer id, String name, String eIN) {

super();

this.id = id;

this.name = name;

EIN = eIN;

}

//...

}

## Create resource controller

[[**1:21**](http://www.youtube.com/embed/bdTsY_7SEXw?start=81&autoplay=1)]

Next, we need something to handle the request and direct it to a method for processing. Within spring, HTTP requests are handled by a controller. These components are identified by the @Controller annotation. The@RequestMapping annotation ensures that HTTP requests to /agencies are mapped to the getAgencies() method.

A key difference between a traditional MVC controller and the RESTful web service controller is the way that the HTTP response is created. Rather than delegating to the view layer we want the object data written directly to the HTTP response as JSON. To accomplish this, the @ResponseBody annotation on the getAgencies() method tells Spring MVC that it does not need to look for a view and the list of agencies can be returned directly to the response.

Before the java object can be sent to the response it needs to be converted to JSON. This is done by an[**HttpMessageConverter**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html) specifically Jackson’s [**MappingJackson2HttpMessageConverter**](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/json/MappingJackson2HttpMessageConverter.html) that it picks up on the class path. If you are not familiar with jackson, jackson is a java library to marshal and unmarshal json.

@Controller

public class AgencyController {

@RequestMapping("/agencies")

@ResponseBody

public List<AgencyResource> getAgencies() {

List<AgencyResource> agencies = getListing();

return agencies;

}

//...

}

Spring 4 introduced a @RestController which is a @Controller and a @ResponseBody in one which eliminates the need to specify @ResponseBody on each method. Notice below that the @Controller has been changed to @RestController and the @ResponseBody has been removed.

@RestController

public class AgencyController {

@RequestMapping("/agencies")

public List<AgencyResource> getAgencies() {

List<AgencyResource> agencies = getListing();

return agencies;

}

//..

}

## Creating mock data

Before we get to running the application, we will create some sample data that we can send in the response.

//...

private List<AgencyResource> getListing() {

List<AgencyResource> resources = new ArrayList<>();

resources.add(new AgencyResource(1, "All State", "123"));

resources.add(new AgencyResource(2, "FCCI Insurance Group", "456"));

resources.add(new AgencyResource(3, "Farmers", "789"));

resources.add(new AgencyResource(4, "Met life", "167"));

return resources;

}

//...

## Running the application

[[**2:51**](http://www.youtube.com/embed/bdTsY_7SEXw?start=171&autoplay=1)]

Running the application the default starter project was nice enough to create an Application.java class. Lets break down what is happening in spring.

The main() method defers to the SpringApplication helper class, providing Application.class as an argument to its run() method. This tells Spring to read the annotation metadata from the Application and to manage it as a component in the Spring application context. In other words it is used to bootstrap and launch the Spring application from a java main method. This might look a bit different than the traditional way to deploy to an application server.

The @ComponentScan annotation is used to look and locate other spring beans within the demo package. This directive ensures that Spring finds and registers the AgencyController as it is marked with a with @Controllerannotation which is a type of @Component.

The @EnableAutoConfiguration annotation is class that enables spring boots convention over configuration which enables default behaviors by scanning the classpath. For example, as we noted above a java object will be marshalled to json by jackson. This is because spring boot found jackson in the class path. In addition, since the project depends on Spring MVC (spring-webmvc.jar), a Spring MVC DispatcherServlet is configured and registered for you through the DispatcherServletAutoConfiguration class.

Right clicking on the project and running the class will start the application in a tomcat container. Navigating to localhost:8080/agencies we will perform a GET and the list of agencies is returned.

Thanks for joining in today’s level up lunch, have a great day!