Getting Started with Spring MVC

Building Basic Applications

Version 1.0

Date July 11, 2017

Table of Contents

Appendices

1 Introduction

This tutorial takes a basic approach to Spring MVC development. You will not use any fancy tools or IDEs to build this tutorial. You will do bare-knuckled programming using Eclipse as the Sun Java 1.5 as the development platform!

1.1 Scope of this document

In this tutorial, we are covering enough essentials of the Spring MVC with just enough theory to keep the discussion going and keep you productively learning to use Spring MVC to build web applications.

You might be surprised that it is easier to develop web applications with Spring MVC, than other java frameworks. Of course, there are better, but we will not talk about any others in this tutorial. I will also not be discussing Inversion of control (IoC) and Dependency injection (DI) in spring.

1.2 Objective

In this tutorial, you get an overview of the Spring MVC framework and learn how to write basic web applications. You build a simple calculator application. We hope that in subsequent iterations, we get to some more advanced aspects like styling etc.

1.3 Intended Audience

If you are new to Spring MVC, this tutorial is or you.

1.4 Prerequisites

This tutorial is written for Java developers whose experience is at beginner to intermediate level. You should have general familiarity with the Java language, Eclipse as the IDE. Having some experience in web applications will be extremely helpful.

1.5 System Requirements

To run the example in this tutorial, you need a Java development environment (JDK). It helps to have an IDE (eclipse), but is not necessary. The sample has all the needed libraries needed and you do not need to download anything.

2 Spring MVC

2.1 Lifecycle of a request in Spring MVC

Before we look into individual components, let us look at the lifecycle of a request in Spring MVC:

1. A request leaves the browser asking for a URL and optionally request parameters
2. The request is first examined by *DispatcherServlet*
3. *DispatcherServlet* consults handler-mappings defined in a configuration file and selects an appropriate controller and delegates the request
4. The controller applied appropriate logic to process the request that results in some information. This information is associated with a logical name of a result rendering entity. This object is returned as an object of type *ModelAndView*
5. *DispatcherServlet* then consults the logical view name with a view resolver to determine the actual view
6. *DispatcherServlet* delivers the model, request to the view implementation, renders an output and send it back to the browser.

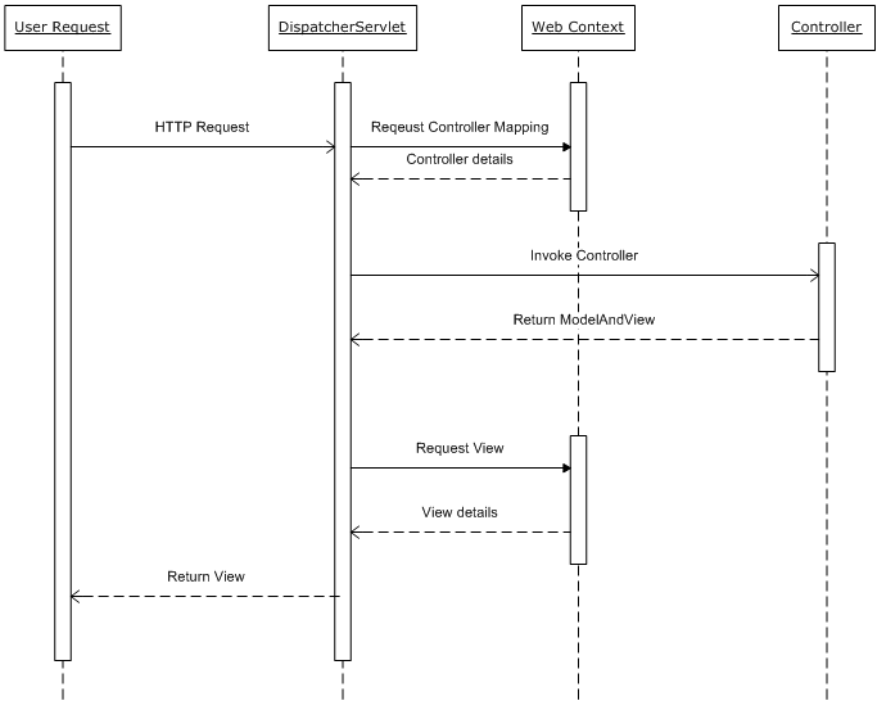

Figure 1: Request Flow

3 The DispatcherServlet

You will notice that we spoke about *DispatcherServlet* in the request flow. This *DispatcherServlet* is an actual servlet (inherits from HttpServlet base class) and has to be declared in deployment descriptors.

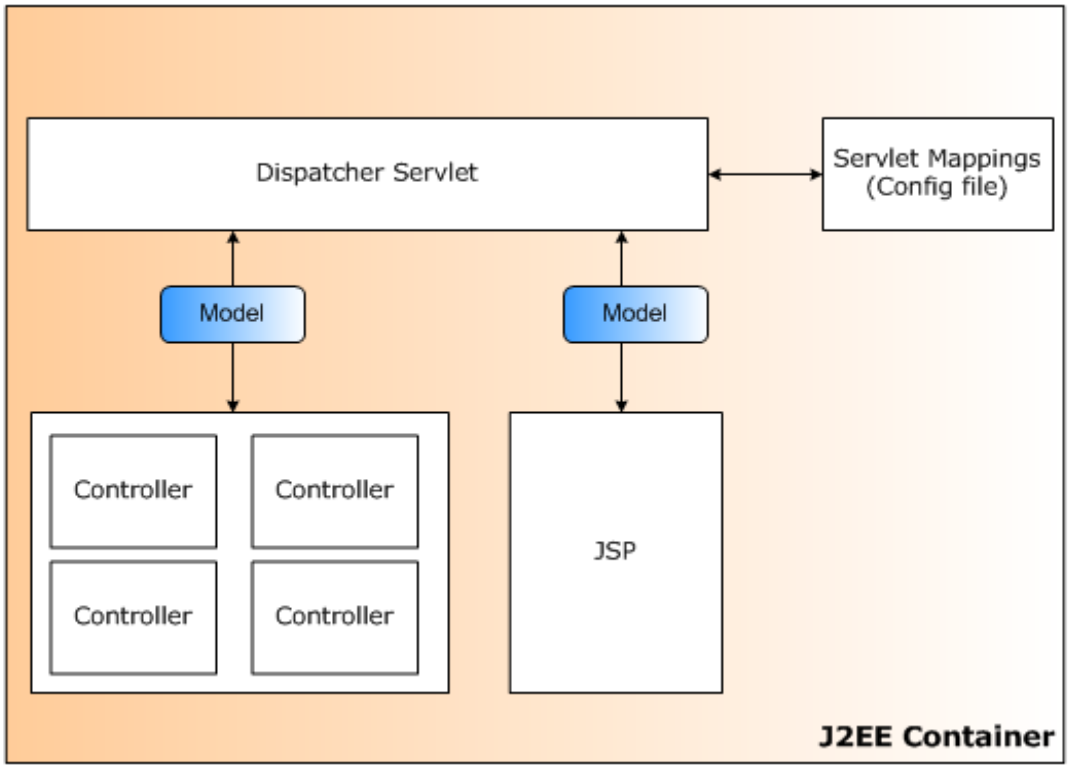

Figure 2: Spring Context and DispatcherServlet

The more pattern-savvy reader will recognize that DispatcherServlet is an implementation of the Core J2EE design pattern *FrontController* design pattern. Spring MVC shares this pattern with other frameworks too. You can setup a *DispatcherServlet* in Spring MVC as under:

```
<servlet>
	<description>Spring MVC Dispatcher Servlet</description>
	<servlet-name>simplecalculator</servlet-name>
	<servlet-class>
		org.springframework.web.servlet.DispatcherServlet
	</servlet-class>
	<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
	<servlet-name>dispatcherservlet</servlet-name>
	<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

With servlet in place, the nest step is to configure the spring meta-data. By default, the DispatcherServlet looks for a configuration file names [servlet-name]-servlet.xml in WEB-INF directory of the web application. For example, the servlet defined above will look for the configuration file */WEB-INF/simplecalculator-servlet.xml*.

The DispatcherServlet has a couple of spring beans it uses in order to be able to process the request and render appropriate responses. These beans are included in the framework itself and can be configured using the *WebApplicationContext*. Each of these beans is described in more detail below.

Bean type	Description
Controller	Controllers are the components that form the 'C' part of the MVC.
Handler mappings	Handler mappings handle the execution of a list of pre- and post-processors and controllers that will be executed if they match the criteria (for instance a matching URL specified with the controller)
View resolver	Offers capability of resolving view names to views
Locale resolver	Offers capability of resolving a client locale
Multipart file resolver	Offers the functionality of file uploads
Handle Exception resolver(s)	Offers functionality to map exceptions to views

4 Controller Basics

As pointed out in the lifecycle of a Spring MVC request, there are multiple things needed to realize a complete round-trip from request to response. You will do the following:

1. Write a controller and map it to a URL in the simplecalculator-servlet.xml
2. Write the view rendering JSP and map it though a view resolver in simplecalculator-servlet.xml
3. You will write concise JSP that will redirect the browser to the home page URL. The web server does not understand Spring MVC and needs a JSP to begin.

4.1 Coding the Controller

```
package com.phoenix.springweb.core;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.*;

/**
 * Controller to generate the Home Page basics to be rendered by a view.
It extends the convenience class
 * AbstractController that encapsulates most of the drudgery involved
in handling HTTP requests.
 */
public class HomePageController extends AbstractController
{
    protected ModelAndView handleRequestInternal(HttpServletRequest
httpServletRequest, HttpServletResponse httpServletResponse) throws
Exception
    {
        // the time at the server
        Calendar cal = Calendar.getInstance(Locale.UK);
        Date now = cal.getTime();

        // time-of-day dependent greeting
        String greeting = "Morning";
        int hour = cal.get(Calendar.HOUR_OF_DAY);
        if (hour == 12)
        {
            greeting = "Day";
        }
        else if (hour > 18)
        {
            greeting = "Evening";
        }
        else if (hour > 12)
        {
            greeting = "Afternoon";
        }
```

```
        // Set the objects in the modelAndView. These will be used by
the view (JSP) to render results
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("time", now);
        modelAndView.addObject("greeting", greeting);
        modelAndView.setViewName("home");

        return modelAndView;
    }
}
```

The controller class extends the *AbstractController* that deals with the intricacies of Java Servlets. The only method that is required is the handleRequestInternal. You put your controller logic in this method. After setting up some data for the view to render, a ModelAndView object is used to pack the data and specify a logical view name. The data object names are tagged that a view can use to pull them out.

In the example, we are putting the following:

```
       // Set the objects in the modelAndView. These will be used by
the view (JSP) to render results
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("time", now);
        modelAndView.addObject("greeting", greeting);
        modelAndView.setViewName("home");
```

The controller completes it work by returning the object. From this point onwards, the DispatcherServlet takes over and resolves the view to be rendered.

Now that the controller is written, you need to tell Spring MVC in *simplecalculator-servlet.xml* to use it:

```
   <bean id="homePage" name="/home.do"
         class="com.phoenix.springweb.co.HomePageController" />
```

When the request is submitted to the server, *DispatcherServer* looks in simplecalculator-servlet.xml for a mapping of the specified URL to a controller bean. This is done by using a BeanNameUrlHandlerMapping by default. You can specify other mapping handlers instead of using a default.

4.2 Writing the view (JSP) and resolver

After the controller has returned, DispatcherServlet looks for a resolver to resolve the view name that it got from the ModelAndView object. You will use the InternalResourceViewResolver to resolve the JSP view name by default.

```
   <bean id="viewResolver"

   class="org.springframework.web.servlet.view.InternalResourceViewResol
ver">
          <property name="prefix" value="/jsp/" />
          <property name="suffix" value=".jsp" />
   </bean>
```

This resolver does is take the view name, add the prefix and append the suffix, and look for the resource with the produced name. So, if you put all your JSP files in the /jsp directory and use their main filename as the logical view names then you have an automatic mapping. Now the JSP itself:

```
<%@ page contentType="text/html;charset=UTF-8" language="java"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
   <head>
   <title>Welcome :: Spring MVC Calculator</title>
   </head>
   <body>
          <h1><c:out value="${greeting}" />! Welcome to Spring
Calculator</h1>
          <p align="center">The time on the server is <c:out
value="${time}" />
          </p>
   </body>
</html>
```

4.3 Landing page

Last but not the least; you need to point your browser to something like http://localhost:8080/CalculatorWeb/home.do to get to the home page. This defeats the purpose of having a home page altogether. You can fix easily.

Add the following entry in web.xml:

```
   <welcome-file-list>
          <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
```

This tells the web server to use index.jsp as the default home page in case no specific page is requested. index.jsp will redirect the browser to the actual home page:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<c:redirect url="/home.do"/>
```

5 Handling Forms

AbstractController is a good for hiding Java Servlet specifics but they do not free you from dealing with raw HTTP requests and responses. Spring MVC gives you AbstractFormController and SimpleFormController that can deal with form display, submission and processing.

The only difference is that SimpleFormController allows you to configure various aspects using a XML file, while AbstractFormController does not.

To understand form processing in Spring MVC, you will now add the calculation capabilities to your web application. For this, you need:

1. A page to be displayed where the user inputs the required parameters (form view)
2. A Java Bean where the input parameters will be mapped (form object)
3. A page to display the results for a successful execution with valid inputs (success view)

5.1 SimpleFormController

It is better that you understand the workflow of the SimpleFormController. Here is the workflow adapted from Spring API documentation in relevance to this tutorial:

1. Receive a GET request for the form input page
2. Instantiate the form input object, i.e. the binding object for parameter mapping
3. The form view is sent to the browser for user input
4. User submits form (using a GET with parameter or a POST)
5. Populate the form object based on the request parameters (applying necessary conversions); in case of binding errors they are reported through Error object
6. If binding is ok and a validator is attached then it is called to validate the form object; in case of errors, they are reported back through Error object
7. If errors are present associated with the model, then form view is resent to the browser
8. If validation passes then a chain of onSubmit() methods is called (one of which should be overridden) that returns the success view by default.

There are multiple submit processing methods of which one you must override depending on the task. In general, override:

1. ModelAndView onSubmit(HttpServletRequest, HttpServletResponse, Object, BindException) if you want to complete control on what to return
2. ModelAndView onSubmit(Object, BindException) if you want custom submission handling but do not need to work with the request and response objects
3. ModelAndView onSubmit(Object) if you need nothing more than the input but may return a custom view
4. void onSubmit(Object) if you do not need to determine the success or error

One point to note is that the SimpleFormController does not force you to override any of these methods. Any kind of overload-mismatch will not show any errors, but will not show desired output.

5.2 Writing your Form Controller

Let us start by defining the Form controller bean

```
<bean id="calculator"
      class="com.phoenix.springweb.core.CalculatorController">
      <property name="formView" value="calculator" />
      <property name="successView" value="calculator" />
      <property name="commandName" value="calculatorDTO" />
      <property name="commandClass"
            value="com.phoenix.springweb.core.CalculatorDTO" />
</bean>
```

- The formView defines the name of the view used to collect the data and to return to in case of errors.
- The successView is returned by default in case no error occurs.
- commandName if the name by which the Data model is referred
- commandClass is the fully qualified name of the class for holds the data

You now need to add the appropriate mappings:

```
<bean id="simpleUrlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMappin
g">
      <property name="mappings">
            <props>
                  <prop key="/home.do">homePage</prop>
                  <prop key="/calculator.do">calculator</prop>
            </props>
      </property>
</bean>
```

Note that the first form request and form submission is handled by the same URI. If there is no data in the request then a new view is loaded, else the request chain is processed.

The CalculatorController implementation:

```
package com.phoenix.springweb.core;

import org.springframework.web.servlet.mvc.SimpleFormController;

public class CalculatorController extends SimpleFormController
{
    protected void doSubmitAction(Object o) throws Exception
    {
        CalculatorDTO calc = (CalculatorDTO)o;

        calc.setResult(calc.getFirstNumber() + calc.getSecondNumber());
    }
}
```

Could it be simpler? Of course, for some of the more complex pieces this implementation will be difficult. However, in our case we process the CalculatorDTO and modify the values in the same object

5.3 Input and Result JSP files

You will use the same JSP file to read user inputs and render results. Let us look at some of code snippets of the JSP file.

Code snippet for the reading user input:

```
                <form:form commandName="calculatorDTO" method="POST"
action="calculator.do">
                        First Number: <form:input path="firstNumber" />
                        <br/>
                        Second Number: <form:input path="secondNumber" />
                        <br/>
                        <input type="submit" title="Do Addition"
value="Do Addition"/>
                </form:form>
```

The <form:form/> tag binds the model and the form objects. You specify the form object name using the commandName attribute. This should match the name provided in the mappings in the simpleCalculator-servlet.xml.

In addition, no additional tag is needed to render the html inputs and submit buttons. When we submit the form, the request is send to the same URI.

Code snippet for rendering results:

```
                <h2>
                        <c:if test="${calculatorDTO.result != null}">
                                The Result is: <c:out
value="${calculatorDTO.result}" />
                        </c:if>
```

```
                </h2>
```

You had put the result in the controller earlier. You will also use the JSTL tag libraries to check for a valid result and render the result only after the form is submitted.

6 Finale

Once you have deployed the application the following screen will be displayed:

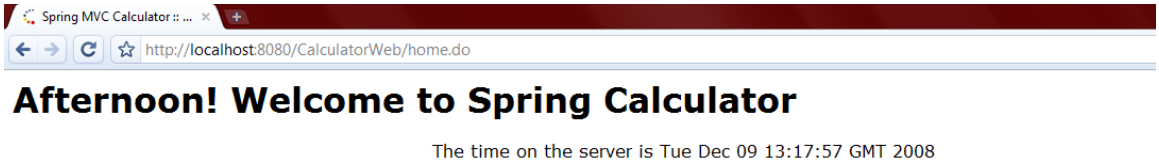

Figure 3: Home Page

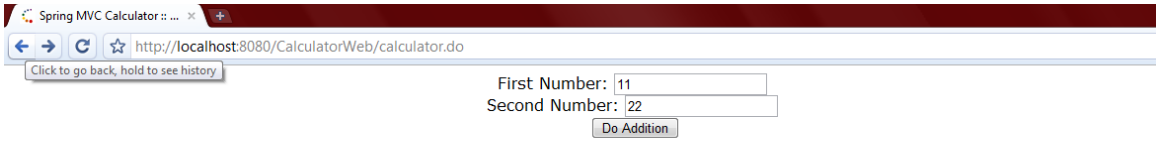

Figure 4: Calculator input page

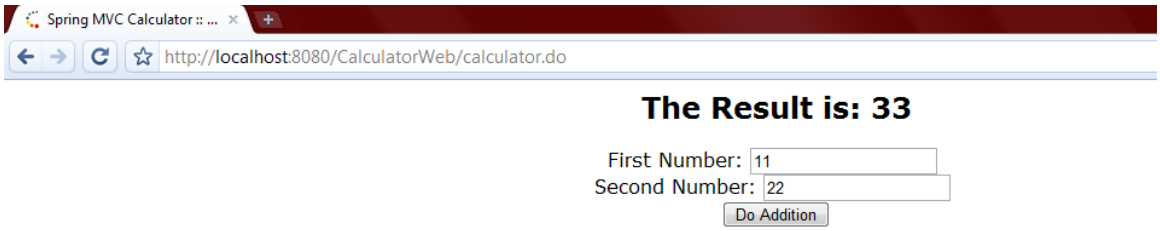

Figure 5: Results page

Appendix

A Libraries used

Library	Description	Version	Vendor
Common-collections	A utility framework built on top of Java Collections Framework	Latest	Apache
Commons-lang		Latest	Apache
Commons-logging	A utility framework built on top of Log4J	Latest	Apache
JSTL	JSP tag libs	1.1.2	Jakarta
Log4j	Used for logging purposes	1.2.15	Apache
Spring	Used for managing POJOs as managed beans	2.5.6	Spring
Spring-mvc	Used for UI	Latest	Spring
Standard	JSP tag libs	1.1.2	Jakarta
Server Runtime	This is needed to execute the application to a web server. **Please note that you may have to change this during deployment on basis of what server you use for deployment (Tomcat, JBoss, Weblogic or Websphere)**	JBoss 4.0.x	JBoss

B Reference Links

Reference Code for Sample Application

http://unitedteams.uhc.com/corporate/gs/india_uhg_captive/ITO/SA/SharedDocuments/Core%20Java%20and%20J2EE/Trainings/Getting%20Started%20with%20Spring%20MVC/Getting%20Started%20with%20Spring%20MVC.zip

Spring Framework

http://www.springframework.org/

Spring MVC

http://static.springframework.org/spring/docs/2.0.x/reference/mvc.html