Spring Framework 3.0 MVC

Aaron Schram

Me

- University of Colorado PhD Student
  - Software Engineering
- Previously employment
  - Mocapay, Inc (mobile payments)
  - Rally Software (agile tooling)
  - BEA (Weblogic Portal)
  - Lockheed Martin

What's Spring MVC?

- A model-view-controller framework for Java web application
- Made to simplify the writing and testing of Java web applications
- Fully integrates with the Spring dependency injection (Inversion of Control) framework
- Open Source
- Developed and maintained by Interface21, recently purchased by VMWare

Project Goals

- J2EE should be easier to use
- It is best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.
- JavaBeans offer a great way of configuring applications.
- OO design is more important than any implementation technology, such as J2EE.
- Checked exceptions are overused in Java. A platform shouldn't force you to catch exceptions you're unlikely to be able to recover from.
- Testability is essential, and a platform such as Spring should help make your code easier to test.

Why Use Spring MVC?

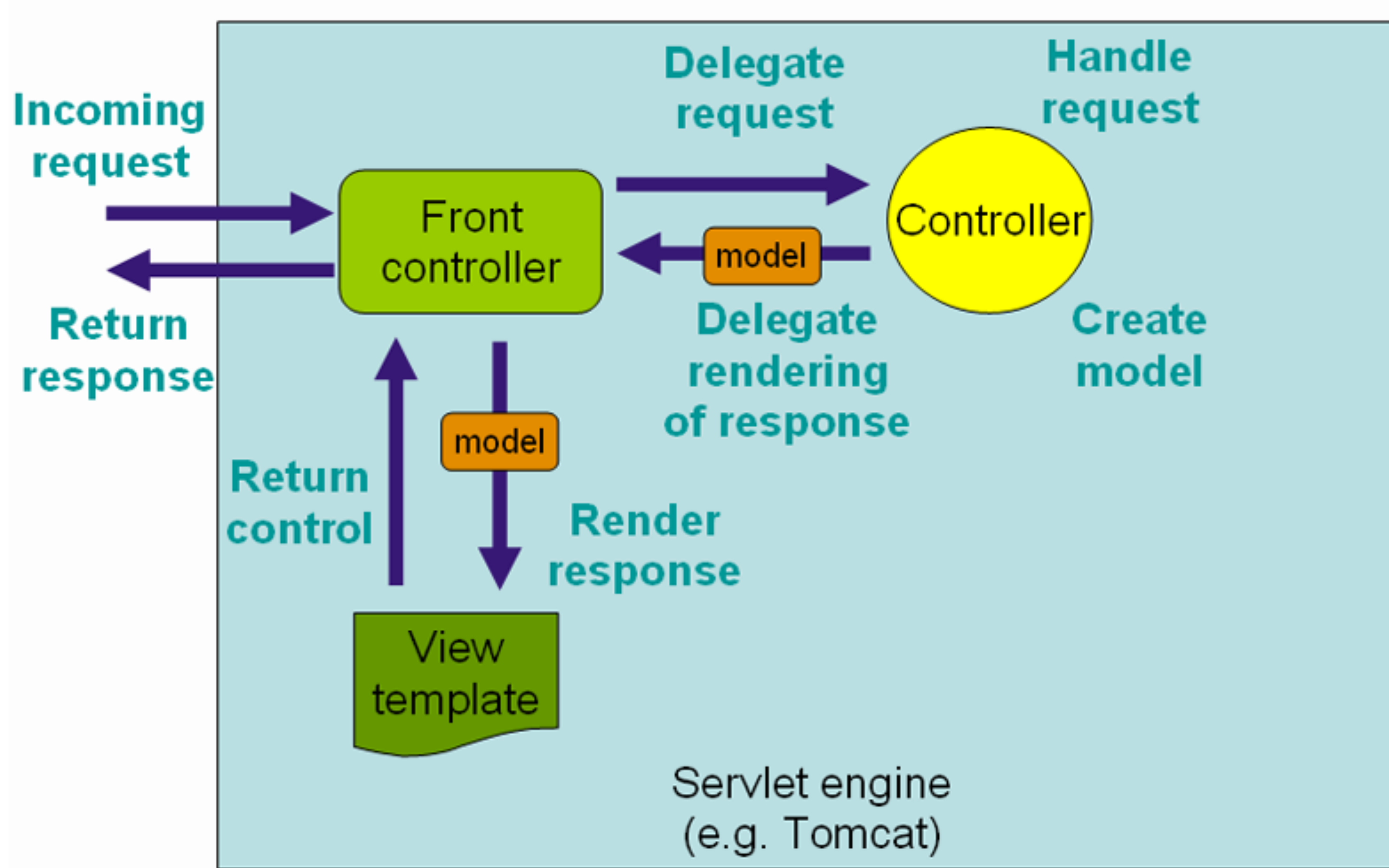
- For most purposes you only have to define one Servlet in web.xml
- Capable of Convention over Configuration
  - Similar to Ruby on Rails or other popular web frameworks that work with dynamic languages
- Normal business objects can be used to back forms
  - No need to duplicate objects just to implement an MVC's command object interface
- Very flexible view resolvers
  - Can by used to map *.json, *.xml, *.atom, etc to the same logic code in one controller and simply output the type of data requested
- Enforces good Software Engineering principles like SRP and DRY

Let's Get Started!

Dispatcher Servlet

- Used to handle all incoming requests and route them through Spring
- Uses customizable logic to determine which controllers should handle which requests
- Forwards all responses to through view handlers to determine the correct views to route responses to
- Exposes all beans defined in Spring to controllers for dependency injection

Dispatcher Servlet Architecture



Uses the **Front Controller Design Pattern**

Defining The Dispatcher Servlet

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Defining a Dispatcher Servlet named **"spring"** that will intercept all urls to this web application

Spring Configuration

- By default Spring looks for a ***servletname*** **-servlet.xml** file in /WEB-INF
- For the previous example we would need to create a file in /WEB-INF named **spring-servlet.xml**

spring-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress UnparsedCustomBeanInspection -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

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

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


</beans>
```

spring-servlet.xml cont.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress UnparsedCustomBeanInspection -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

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

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


</beans>
```

<mvc:annotation-driven /> tells Spring to support annotations like **@Controller**, **@RequestMapping** and others that simplify the writing and configuration of controllers

spring-servlet.xml cont.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress UnparsedCustomBeanInspection -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

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

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


</beans>
```

Define a simple view resolver that looks for JSPs that match a given view name in the director **/WEB-INF/jsp**

spring-servlet.xml cont.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--suppress UnparsedCustomBeanInspection -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- JSR-303 support will be detected on classpath and enabled automatically -->
    <mvc:annotation-driven/>

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

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


</beans>
```

Tell Spring where to automatically detect controllers

Configuration Done!

Woo Hoo!

So What's a Controller Look Like?

Example: Classroom Controller

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
    public static final String URL_MAPPING = "/classroom";
    public static final String VIEW_NAME = "classroom";

    public static final Logger logger = Logger.getLogger(ClassroomController.class);

    private ClassroomService classroomService;

    @Autowired
    public ClassroomController(ClassroomService classroomService)
    {
        this.classroomService = classroomService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String handleRequest(ModelMap model) throws Exception
    {
        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
    public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("classroom", classroomService.getClassroom(classroomId));

        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}/students", method = RequestMethod.GET)
    public String getStudentsForClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("students", classroomService.getStudentsInClassroom(classroomId));

        return VIEW_NAME;
    }
}
```

A Controller that gets a class or all the students in the class

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
```

Mark this class as a controller

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
```

Define what default URLs this class should respond to

Side Note: Autowiring

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
    public static final String URL_MAPPING = "/classroom";
    public static final String VIEW_NAME = "classroom";

    public static final Logger logger = Logger.getLogger(ClassroomController.class);

    private ClassroomService classroomService;

    @Autowired
    public ClassroomController(ClassroomService classroomService)
    {
        this.classroomService = classroomService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String handleRequest(ModelMap model) throws Exception
    {
        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
    public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("classroom", classroomService.getClassroom(classroomId));

        return VIEW_NAME;
    }
```

Autowiring allows Spring to do the instantiation of the class you want to make use of for you. At run time you will be able to access all methods of the class without worrying about how you got the class. This is known as **Dependency Injection**.

Back To Classroom Example

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
    public static final String URL_MAPPING = "/classroom";
    public static final String VIEW_NAME = "classroom";

    public static final Logger logger = Logger.getLogger(ClassroomController.class);

    private ClassroomService classroomService;

    @Autowired
    public ClassroomController(ClassroomService classroomService)
    {
        this.classroomService = classroomService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String handleRequest(ModelMap model) throws Exception
    {
        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
    public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("classroom", classroomService.getClassroom(classroomId));

        return VIEW_NAME;
    }
```

This method is the default method called when **/classroom** or / is hit from a client. It simply forwards to a jsp named **classroom.jsp** located in **/WEB-INF/jsp**

Side Note: Restful URLs

- Spring like many other popular frameworks can make use of RESTful URLs
  - They come in the style of **/users/*user_id***
  - Commonly without any extension such as **.html**
- Popularized by **Ruby on Rails**
- Collections are accessed like:
  - **/users**
- Individual entries are accessed like:
  - **/users/*user_id***
- *CRUD operations are done via HTTP methods*
  - *PUT, POST, GET, DELETE*

Classroom RESTful URLs

```java
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
    public static final String URL_MAPPING = "/classroom";
    public static final String VIEW_NAME = "classroom";

    public static final Logger logger = Logger.getLogger(ClassroomController.class);

    private ClassroomService classroomService;

    @Autowired
    public ClassroomController(ClassroomService classroomService)
    {
        this.classroomService = classroomService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String handleRequest(ModelMap model) throws Exception
    {
        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
    public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("classroom", classroomService.getClassroom(classroomId));

        return VIEW_NAME;
    }
```

The highlighted section above demonstrates how to accomplish RESTful URLs in the Spring MVC Framework. Using the **@PathVariable** annotation you can gain access to the variable passed in on the URI. This is commonly referred to as URI Templating.

What's a Model?

- A ***Model*** is used in Spring MVC to pass objects from the controller tier up into the view
- A ***Model*** is really just a ***java.util.Map***
- You can add attributes to a ***Model*** and they will be put on the request as attributes and available in the applications ***PageContext***.
- In Spring you can simply pass back a ***Map*** or one of two Spring specific classes; ***ModelMap*** or ***Model***

ModelMap Example

```
@Controller
@RequestMapping(value = {ClassroomController.URL_MAPPING, "/"})
public class ClassroomController
{
    public static final String URL_MAPPING = "/classroom";
    public static final String VIEW_NAME = "classroom";

    public static final Logger logger = Logger.getLogger(ClassroomController.class);

    private ClassroomService classroomService;

    @Autowired
    public ClassroomController(ClassroomService classroomService)
    {
        this.classroomService = classroomService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String handleRequest(ModelMap model) throws Exception
    {
        return VIEW_NAME;
    }

    @RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
    public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
    {
        model.addAttribute("classroom", classroomService.getClassroom(classroomId));

        return VIEW_NAME;
    }
```

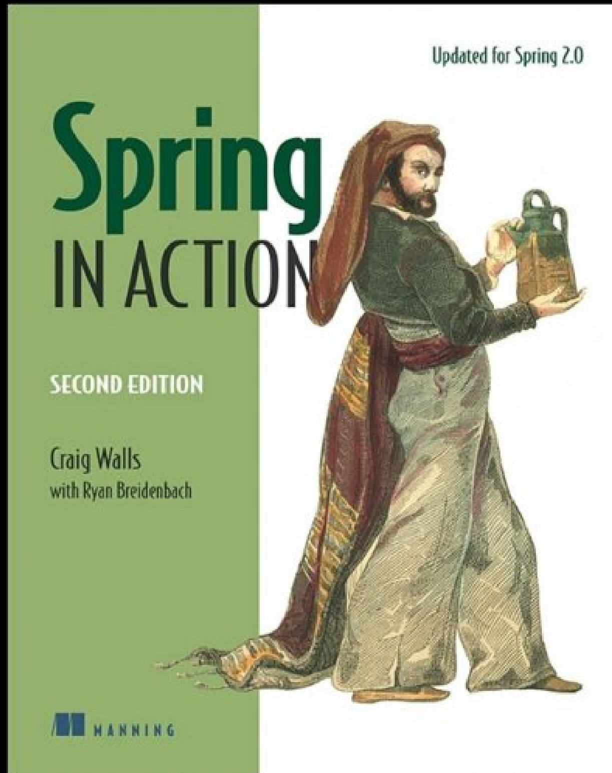
In the above example we use a service method to read and return a **Classroom** object. We make that **Classroom** object available to the view under the key ***"classroom"*** by calling ***addAttribute()*** on the **ModelMap**

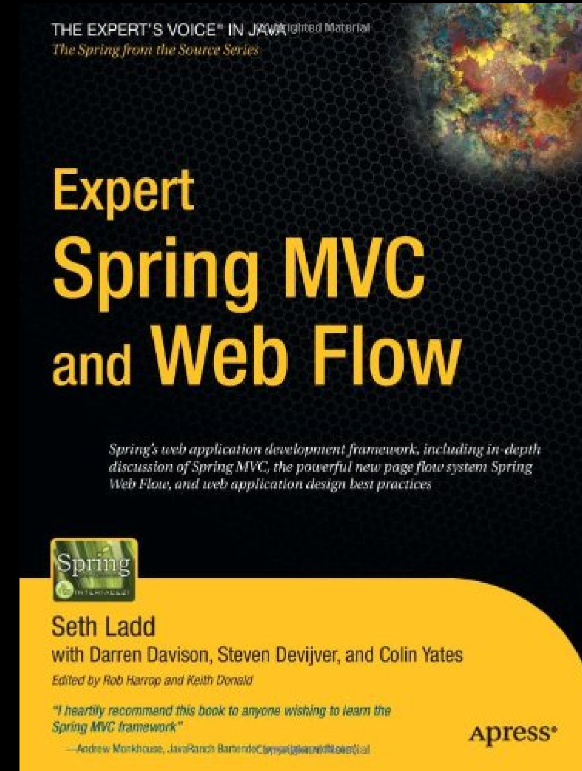
Getting All Students In A Classroom

```
@Autowired
public ClassroomController(ClassroomService classroomService)
{
    this.classroomService = classroomService;
}

@RequestMapping(method = RequestMethod.GET)
public String handleRequest(ModelMap model) throws Exception
{
    return VIEW_NAME;
}

@RequestMapping(value = "/classroom/{classroomId}", method = RequestMethod.GET)
public String getClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
{
    model.addAttribute("classroom", classroomService.getClassroom(classroomId));

    return VIEW_NAME;
}

@RequestMapping(value = "/classroom/{classroomId}/students", method = RequestMethod.GET)
public String getStudentsForClassroom(@PathVariable("classroomId") Long classroomId, ModelMap model)
{
    model.addAttribute("students", classroomService.getStudentsInClassroom(classroomId));

    return VIEW_NAME;
}
```

Above you can see that how to get all the students in a given classroom by requesting the URL **/classroom/*{id}* /students**. A Java **List<Student>** will be available to the **classroom.jsp** view for display

More Helpful Information





- SpringSource.org Chapter 15
- RESTful URLs

Aaron Schram

aaron.schram@colorado.edu