Spring MVC
Overview of MVC paradigm
The components of Spring MVC
MVC and Dependency Injection
Implementing a basic Controller
Creating a simple View
Configuring a Spring MVC application
  Configuring URL mappings
  Mapping views
Grouping request handling logic with MultiActionController
Handling form posts
  Adding validation
  Using data binding
  Adding error reporting
  Configuring form and success views
MVC Overview

MVC = Model-View-Controller
Clearly separates business, navigation and presentation logic
Proven mechanism for building a thin, clean web-tier
MVC Components

Three core collaborating components

Controller

- Handles navigation logic and interacts with the service tier for business logic

Model

- The contract between the Controller and the View
- Contains the data needed to render the View
- Populated by the Controller

View

- Renders the response to the request
- Pulls data from the model
Motivation for MVC

Eases maintenance burden
Changes to business logic are less likely to break the presentation logic
Vice versa

Facilitates multi-disciplined team development

Developers can focus on creating robust business code without having to worry about breaking the UI
Designers can focus on building usable and engaging UIs without worrying about Java
Motivation for MVC

Use the best tool for the job
Java is especially suited to creating business logic code
Markup or templating languages are more suited to creating HTML layouts

Ease testability
Business and navigation logic are separated from presentation logic meaning they can be tested separately
Practically: you can test more code outside the servlet container
MVC in Spring

A single **Front Controller** servlet that dispatches requests to individual Controllers

- Proven pattern shown in Struts and Core J2EE Patterns

Request routing is completely controlled by the Front Controller

- Individual Controllers can be used to handle many different URLs

Controllers are POJOs

Controllers are managed exactly like any other bean in the Spring ApplicationContext
Core Components of Spring MVC

DispatcherServlet

Spring’s Front Controller implementation

Controller

User created component for handling requests
Encapsulates navigation logic
Delegates to the service objects for business logic

View

Responsible for rendering output
Core Components of Spring MVC

ModelAndView

- Created by the Controller
- Stores the Model data
- Associates a View to the request
  - Can be a physical View implementation or a logical View name

ViewResolver

- Used to map logical View names to actual View implementations

HandlerMapping

- Strategy interface used by DispatcherServlet for mapping incoming requests to individual Controllers
MVC and Dependency Injection

All MVC components are configured in the Spring ApplicationContext
As such, all MVC components can be configured using Dependency Injection

Example:

```xml
<bean id="springCheersController" class="com....web.SpringCheersController">
  <property name="methodNameResolver" ref="springCheersMethodResolver"/>
  <property name="service" ref="service"/>
</bean>
```
Creating a Basic Controller

Goals

Create a thin-wrapper around the business functionality
Keep all business processing out of the web tier
Handle only navigation logic

Process

Create the Controller class

• Implement the Controller interface
• Or extend one of the pre-built Controller implementations

Create a setter to inject the service object
Implement the handleRequest() method
Creating a Basic Controller

```java
public class BeerListController implements Controller {
    private SpringCheersService service;

    public void setService(SpringCheersService service) {
        this.service = service;
    }

    public ModelAndView handleRequest(
        HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse)
        throws Exception {
        List beers = this.service.findAllBeers();
        return new ModelAndView("beerList", "beers", beers);
    }
}
```
Creating a Basic Controller

What did we do?
Create a class that implements the Controller interface

What’s left?
Configure the Spring MVC infrastructure
  • Once per application
Configure the Controller
Map the Controller to one or more URLs
Create a view
Map the view name to the view
Views in Spring MVC

Extensive support for many different view technologies

JSP, JSTL, Velocity, FreeMarker, JasperReports, PDF, Excel

Views are represented using logical view names which are returned by the Controller

Can return an actual `View` class from the Controller if needed
View Resolution in Spring MVC

View names are mapped to actual view implementations using `ViewResolvers`

`ViewResolvers` are configured in the `web-tier` `ApplicationContext`

Automatically detected by `DispatcherServlet`

Can configure multiple, ordered `ViewResolvers`
ViewResolver Implementations

- **InternalResourceViewResolver**
  Uses RequestDispatcher to route requests to internal resources such as JSPs
  Model data is placed in request scope for access in the view

- **FreeMarkerViewResolver**
  Uses FreeMarkerView to render the response using the FreeMarker template engine

- **VelocityViewResolver**
  Uses VelocityView to render the response using the FreeMarker template engine

- **BeanNameViewResolver**
  Maps the view name to the name of a bean in the ApplicationContext.
  Allows for view instances to be explicitly configured
Creating a View with JSP and JSTL

```html
<html>
    <head><title>Beer List</title></head>
    <body>
        <table border="0">
            <c:forEach items="${beers}" var="beer">
                <tr>
                    <td><c:out value="${beer.id}" /></td>
                    <td><c:out value="${beer.brand}" /></td>
                </tr>
            </c:forEach>
        </table>
    </body>
</html>
```
Configuring a Spring MVC Application

Configure the DispatcherServlet in web.xml

Configure ContextLoaderListener or ContextLoaderServlet to load the business tier and data tier ApplicationContexts

Create the web-tier ApplicationContext configuration file

Configure Controllers

Map URLs to Controllers

Map logical view names to view implementations
Configuring a Spring MVC Application

DispatcherServlet
(Awaiting incoming HttpServletRequests)

WebApplicationContext
(Containing controllers, view resolvers, locale resolvers and other web-related beans)

- Controllers
- HandlerMapping
- ViewResolver

WebApplicationContext(s)
(Containing middle-tier services, datasources, etcetera)
Configuring DispatcherServlet

```xml
<servlet>
  <servlet-name>springcheers</servlet-name>
  <servlet-class>o.s.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>springcheers</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>
    o.s.web.context.ContextLoaderListener
  </listener-class>
</listener>
Configuring a Spring MVC Application

Creating the web-tier ApplicationContext configuration:

Naming is important – follows the pattern /WEB-INF/<servlet_name>-servlet.xml

DispatcherServlet will automatically load this file when setting up its ApplicationContext

In our example this would be /WEB-INF/springcheers-servlet.xml
<bean id="beerListController"
    class="com.springcheers.web.BeerListController">
    <property name="service" ref="service"/>
</bean>
Mapping URLs to Controllers

Mapping request (URLs) to Controller

Controlled by implementations of the HandlerMapping interface

Useful out-of-the-box implementations

- BeanNameUrlHandlerMapping
  - Uses the Controller bean name as the URL mapping
- SimpleUrlHandlerMapping
  - Define a set of URL pattern to bean mappings

Most out of the box implementations support Ant-style path matching
Configure a HandlerMapping

```xml
<bean id="urlMapping"
    class="o.s.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/list.htm">springCheersController</prop>
            <prop key="/view.htm">springCheersController</prop>
            <prop key="/edit.htm">customerForm</prop>
            <prop key="/create.htm">customerForm</prop>
            <prop key="/beer/list.htm">beerListController</prop>
        </props>
    </property>
</bean>
```
<bean id="viewResolver" class="o.s.w.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
Understanding MultiActionController

One controller to handle different tasks

Multiple handler methods

• Each method handles a different request

MethodNameResolver determines method

• Based on parameter or other criteria

Can use a delegate to come up with ModelAndView

Good for grouping related tasks into a single class
public class SpringCheersController extends MultiActionController {

    private SpringCheersService service;

    /** setter ommitted */

    public ModelAndView handleCustomerList(
        HttpServletRequest request, HttpServletResponse response) {
        return new ModelAndView("customerList",
                "customers", this.service.getCustomerList());
    }

    public ModelAndView handleViewCustomer(
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        long id = RequestUtils.getRequiredLongParameter(request, "customerId");
        return new ModelAndView("viewCustomer",
                "customer", this.service.getCustomer(id));
    }
}
Configuring a MultiActionController

```xml
<bean id="springCheersController"
     class="com.springcheers.web.SpringCheersController">
    <property name="methodNameResolver"
              ref="springCheersControllerResolver"/>
    <property name="service" ref="service"/>
</bean>

<bean id="springCheersControllerResolver"
     class="o.s.w.servlet.mvc.multiaction.PropertiesMethodNameResolver">
    <property name="mappings">
        <props>
            <prop key="/list.htm">handleCustomerList</prop>
            <prop key="/view.htm">handleViewCustomer</prop>
        </props>
    </property>
</bean>
```
Unit Testing a Controller

Test with mock request, response and service

Glass-box testing
  Ensure that the service is invoked as desired
  Fits well with a TDD approach

Test a variety of interactions
  Controller with the request and response
  Controller with the service
private SpringCheersController controller;
private SpringCheersService service;
private MockControl serviceControl;

public void setUp() {
    this.controller = new SpringCheersController();
    this.serviceControl =
        MockControl.createControl(SpringCheersService.class);
    this.service =
        (SpringCheersService) this.serviceControl.getMock();
    
    this.controller.setService(this.service);
}
public void testHandleViewCustomer() throws Exception{
    MockHttpServletRequest request = new MockHttpServletRequest();
    MockHttpServletResponse response = new MockHttpServletResponse();

    request.addParameter("customerId", "1");

    Customer dummyCustomer = new Customer();
    this.service.getCustomer(1);
    this.serviceControl.setReturnValue(dummyCustomer);
    this.serviceControl.replay();

    ModelAndView mv = this.controller.handleViewCustomer(request, response);

    assertNotNull("ModelAndView should not be null", mv);
    assertEquals("Invalid view name", "viewCustomer", mv.getViewName());
    Customer customer = (Customer)mv.getModel().get("customer");

    assertNotNull("Customer should not be null", customer);
    assertEquals("Invalid customer returned", dummyCustomer, customer);
}
public class BeerListControllerIntegrationTests 
        extends AbstractControllerIntegrationTests {
    private BeerListController beerListController;

    public void setBeerListController(BeerListController beerListController) {
        this.beerListController = beerListController;
    }

    public void testListBeers() throws Exception {
        MockHttpServletRequest request = new MockHttpServletRequest();
        MockHttpServletResponse response = new MockHttpServletResponse();

        ModelAndView mv = this.beerListController.handleRequest(request, response);

        assertEquals("Incorrect view name", "beerList", mv.getViewName());

        List beers = (List) mv.getModel().get("beers");
        assertNotNull("Beer list not in model", beers);

        int count = jdbcTemplate.queryForInt("select count(0) from beers");
        assertEquals("Incorrect number of beers in list", count, beers.size());
    }
}
Handling Form Posts with SimpleFormController

- Create the custom SimpleFormController
- Create the form view
- Adding data binding logic to the form view
- Add error display logic to the form view
- Create the success view
- Define a command object for the form
- Add on submit logic
- Optionally
  - Add validation logic
  - Hook in custom data binding logic
Request Workflow of SimpleFormController

GET request displays the form
POST request submits the form
Both have distinct workflow
  GET does not need validation
  POST does not need form view
...
Implement template methods to customize behavior
GET request – Form Display

formBackingObject()

Retrieve the command object
Allows for pre-population of the form

initBinder()

Register custom editors

referenceData()

Load reference data needed for displaying the form

showForm()

Completes ModelAndView and returns
Command object stored in session if configured
Renders the actual form
POST request – form submission

formBackingObject()
  Retrieve the command object
  • Maybe from session, maybe from database
initBinder()
  Register custom editors
  Binding of request parameters to form
onBind()
  Called after bind but before validation
  Allows you to manually bind request parameters to the command object before validation
Validation done using Validators
onBindAndValidate()
  Called after bind and validate
  Allows you to bind parameters to the command that don’t need validation
If validation fails then add errors to the ModelAndView and show the form again
If validation succeeds call onSubmit() callbacks and show the success view
Creating the Form View

<html>
<head>
  <title>Spring Cheers</title>
</head>
<body>
<h1>Update Customer</h1>
<form name="editCustomer" method="POST">
  <table border="0">
    <tr>
      <td>Name: </td>
      <td>
        <input type="text" size="30" name="command.name"/>
      </td>
    </tr>
    <tr>
      <td colspan="2">&nbsp;</td>
      <td><input type="submit" value="Save"/></td>
    </tr>
  </table>
</form>
</body>
</html>
Adding Data Binding to the Form

```xml
<spring:bind path="command.name">
  <td>
    <input type="text" size="30"
           name="<c:out value='${status.expression}'/>">
    value="<c:out value='${status.displayValue}' />
  </td>
</spring:bind>
```
Adding Error Handling to the Form

<spring:bind path="command.name">
   <td>
      <input type="text" size="30"
             name="<c:out value='${status.expression}'/>">
      <input type="text" size="30"
             name="<c:out value='${status.displayValue}' />">
   </td>
   <td>
      <c:if test="${status.error}">
         <div class="error">
            <c:forEach items="${status.errorMessages}" var="error">
               <c:out value="${error}"/>
            </c:forEach>
         </div>
      </c:if>
   </td>
</spring:bind>
Creating the CustomerForm Controller

public class CustomerForm extends SimpleFormController {

    private SpringCheersService service;

    public void setService(SpringCheersService service) {
        this.service = service;
    }

    protected Object formBackingObject(HttpServletRequest request)
            throws Exception {
        long id = RequestUtils.getLongParameter(request, "customerId", -1);
        return (id > 0) ? this.service.getCustomer(id) : new Customer();
    }

    protected void doSubmitAction(Object customer) throws Exception {
        this.service.saveCustomer((Customer) customer);
    }
}
Validation Architecture

Not tied to the HttpServletRequest
Not tied to the web-tier
  • Validation of domain objects
  • Input from remote clients also needs validation
  • Can easy be tested outside of the container

Implementation independence
Conversion errors are non-fatal
  • java.lang.Long property
    • Typing in nothing (converts to null)
    • Typing in ‘foo’
    • No difference with respect to validation!!
Creating a Validator

```java
public class CustomerValidator implements Validator {

    public boolean supports(Class cls) {
        return (cls == Customer.class);
    }

    public void validate(Object obj, Errors errors) {
        Customer customer = (Customer) obj;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name", "required", "required");
    }
}
```
<bean id="customerForm"
    class="com.springcheers.web.CustomerForm">
    <property name="formView" value="editCustomer"/>
    <property name="successView" value="redirect:list.htm"/>
    <property name="service" ref="service"/>
    <property name="validator" ref="customerValidator"/>
</bean>
Summary

Spring MVC provides a sophisticated MVC implementation

- Interface-based for easy testing
- Fully integrated with Spring IOC
- Comprehensive view technology integration
  - JSP & JSTL
  - Velocity
  - FreeMarker
  - PDF
  - Excel