

Mastering Spring MVC 3

And its @Controller programming model

Get the code for the demos in this presentation at

- <http://src.springsource.org/svn/spring-samples/mvc-showcase>

Topics

- Getting started
- Introduction to the MVC programming model
- Mapping HTTP requests
- Obtaining request input
- Generating responses
- Rendering views
- Type conversion and validation
- Exception handling
- Testing

Getting started

- **Create a new Spring MVC project from a template**

- Most use Roo to do this, either from an IDE like STS or the command-line

Typical setup:

- **One DispatcherServlet registered in web.xml**

- FrontController that dispatches web requests to your application logic
- Generally the “default servlet” mapped to “/”

- **Two Spring Containers (or ApplicationContexts) instantiated**

- 1 “root” context to host “shared resources” required by Servlets / Filters
- 1 “web” context to host local application components delegated to by the DispatcherServlet
 - Your application components are typically discovered via classpath scanning

Demo

Typical Spring MVC project structure

Introduction to the MVC programming model

- **DispatcherServlet requests are mapped to @Controller methods**
 - @RequestMapping annotation used to define mapping rules
 - Method parameters used to obtain request input
 - Method return values used to generate responses
- **Simplest possible @Controller**

```
@Controller
public class HomeController {
    @RequestMapping("/")
    public @ResponseBody String home() {
        return "hello world";
    }
}
```

Demo

Simplest possible @Controller

Mapping requests

■ By path

- `@RequestMapping("path")`

■ By HTTP method

- `@RequestMapping("path", method=RequestMethod.GET)`
 - POST, PUT, DELETE, OPTIONS, and TRACE are also supported

■ By presence of query parameter

- `@RequestMapping("path", method=RequestMethod.GET, params="foo")`
- Negation also supported: `params={ "foo", "!bar" }`
- Parameter value also supported: `params="foo=bar"`

■ By request header

- `@RequestMapping("path", header="content-type=text/*")`
- Negation also supported

Mapping requests (2)

- Simplest possible @Controller revisited

```
@Controller
public class HomeController {
    @RequestMapping("/", method=RequestMethod.GET,
        headers="Accept=text/plain")
    public @ResponseBody String home() {
        return "hello world";
    }
}
```


Demo

Mapping requests

Request mapping at the class level

- **@RequestMapping** can be used at the class level
 - Concise way to map all requests *within* a path to a @Controller

```
@Controller
@RequestMapping("/accounts/*")
public class AccountsController {

    @RequestMapping("active")
    public @ResponseBody List<Account> active() { + }

    @RequestMapping("inactive")
    public @ResponseBody List<Account> inactive() { + }

}
```

Request mapping at the class level (2)

- The same rules expressed with method-level mapping only:

```
@Controller
public class AccountsController {

    @RequestMapping("/accounts/active")
    public @ResponseBody List<Account> active() { + }

    @RequestMapping("/accounts/inactive")
    public @ResponseBody List<Account> inactive() { + }

}
```

Demo

@RequestMapping at the class level

Obtaining request data

- **Obtain request data by declaring method arguments**
 - **A query parameter value**
 - `@RequestParam("name")`
 - **A group of query parameter values**
 - A custom `JavaBean` with a `getName()/setName()` pair for each parameter
 - **A path element value**
 - `@PathVariable("var")`
 - **A request header value**
 - `@RequestHeader("name")`
 - **A cookie value**
 - `@CookieValue("name")`
 - **The request body**
 - `@RequestBody`
 - **The request body and any request header**
 - `HttpEntity<T>`

Demo

Obtaining request data

Injecting standard objects

- A number of “standard arguments” can also be injected
 - Simply declare the argument you need
- **HttpServletRequest** (or its more portable **WebRequest** wrapper)
- **Principal**
- **Locale**
- **InputStream**
- **Reader**
- **HttpServletResponse**
- **OutputStream**
- **Writer**
- **HttpSession**

Injecting custom objects

- **Custom object injectors can also be defined**
 - Implement the WebArgumentResolver extension point
 - Register with the AnnotationMethodHandlerAdapter

```
public interface WebArgumentResolver {  
  
    Object resolveArgument(MethodParameter param,  
        NativeWebRequest request);  
  
}
```


Demo

Injecting standard and custom objects

Generating responses

- **Return a POJO annotated with `@ResponseBody`**

- POJO marshaled as the body of the response

or

- **Return a new `ResponseEntity<T>` object**

- More powerful; allows for setting custom response headers and status code

Demo

Generating responses

HttpMessageConverters

- Behind the scenes, a `HttpMessageConverter` underpins reading the request body and generating the response
- Multiple converters may be registered for different content types
- For `@RequestBody`, the first converter that can read the POSTed “Content-Type” into the desired method parameter type is used
- For `@ResponseBody`, the first converter that can write the method return type into one of the client’s “Accept”ed content types is used
 - Also applies to `HttpResponseBodyEntity<T>` (can also force the content-type)
- Default set of `HttpMessageConverters` registered for you
- Can write your own

Default `HttpMessageConverters`

■ `StringHttpMessageConverter`

- Reads “text/*” into Strings; writes Strings as “text/plain”

■ `FormHttpMessageConverter`

- Reads “application/x-www-form-urlencoded” into `MultiValueMap<String, String>`
- Writes `MultiValueMap<String, String>` into “application/x-www-form-urlencoded”

■ `ByteArrayMessageConverter`

- Reads “*/*” into a `byte[]`; writes Objects as “application/octet-stream”

■ `Jaxb2RootElementHttpMessageConverter`

- Reads “text/xml” || “application/xml” into Objects annotated by JAXB annotations
- Writes JAXB-annotated Objects as “text/xml” or “application/xml”
- Only registered by default if JAXB is present on the classpath

Default `HttpMessageConverters` (2)

■ `MappingJacksonHttpMessageConverter`

- Reads “application/json” into Objects; writes Objects as “application/json”
- Delegates to the Jackson JSON Processing Library
- Only registered by default if Jackson API is in your classpath

■ `SourceHttpMessageConverter`

- Reads “text/xml” or “application/xml” into `javax.xml.transform.Source`
- Writes `javax.xml.transform.Source` to “text/xml” or “application/xml”

■ `ResourceHttpMessageConverter`

- Reads/writes `org.springframework.core.io.Resource` objects

■ `AtomFeed/RssChannelHttpMessageConverter`

- Reads/writes Rome Feed and RssChannels (`application/atom+xml` | `rss+xml`)
- Only registered by default if Rome is present in your classpath

Demo

Default HttpMessageConverters

Other `HttpMessageConverters` options available to you

- **`BufferedImageHttpMessageConverter`**

- Reads/writes mime-types supported by Java Image I/O into `BufferedImage`

- **`MarshallingHttpMessageConverter`**

- Reads/writes XML but allows for pluggability in Marshalling technology

- **Register your own or customize existing ones by setting the “`messageConverters`” property of the `AnnotationMethodHandlerAdapter` bean**

- Easy to override the defaults using a `BeanPostProcessor`

Demo

Customizing the HttpMessageConverters

Rendering views

- **A DispatcherServlet can also render Views**

- Alternative to having a `HttpMessageConverter` write the response body
- Designed for generating text/* content from a template

- **Declare a Model parameter to export data to the view**

- Call `model.setAttribute("name", value)` for each item to export

- **Select the view by to render by returning a String**

- Do not use `@ResponseBody` annotation in this case
- Configured `ViewResolver` maps name to a View instance

- **Default ViewResolver forwards to internal servlet resources**

- Many other options: JSP, Tiles, Freemarker, Velocity, iText PDF, JExcel, Jasper Reports, and XSLT are all supported out of the box
- Can also write your own View integrations

Demo

Rendering views

Type conversion

- **Type conversion happens automatically**
- **A common “ConversionService” underpins the places where type conversion is required**
 - Always used with `@RequestParam`, `JavaBean`, `@PathVariable`, and `@RequestHeader`, and `@CookieValue`
 - `HttpMessageConverter` may use when reading and writing objects
 - for `@RequestBody`, `@ResponseBody`, `HttpEntity`, `ResponseEntity`
- **All major conversion requirements satisfied out-of-the-box**
 - Primitives, Strings, Dates, Collections, Maps, custom value objects
- **Can declare annotation-based conversion rules**
 - `@NumberFormat`, `@DateTimeFormat`, your own custom `@Format` annotation
- **Elegant SPI for implementing your own converters**

Demo

Type Conversion System

- **Trigger validation by marking a JavaBean parameter as `@Valid`**
 - The JavaBean will be passed to a Validator for validation
 - JSR-303 auto-configured if a provider is present on the classpath
- **Binding and validation errors can be trapped and introspected by declaring a `BindingResult` parameter**
 - *Must* follow the JavaBean parameter in the method signature
 - Errors automatically exported in the model when rendering views
 - *Not supported* with other request parameter types (`@RequestBody`, etc)

Demo

Validation

Exception Handling

■ Two-levels of Exception Handling

- @Controller level
- DispatcherServlet level

■ @Controller level

- Annotate a separate method in your @Controller as a @ExceptionHandler
- Or simply catch the Exception yourself in your handler method

■ DispatcherServlet level

- Rely on the DefaultHandlerExceptionResolver
 - Maps common exceptions to appropriate status codes
- Supplement with your own custom HandlerExceptionResolver as needed

Demo

Exception Handling

Testing

■ Unit Testing

- Controllers are just POJOs - just new them up and test them!
- Inject mock dependencies using your favorite mocking library (Mockito)

■ HttpServlet Mocks

- Useful when you have a Servlet API dependency in your @Controller
- MockHttpServletRequest, MockHttpServletResponse, MockServletContext

■ Integration Testing

- Selenium-based acceptance tests great way to exercise end-to-end behavior

Demo

Testing

Resources

■ Reference Manual

- <http://www.springsource.org/documentation>

■ Samples

- <http://src.springsource.org/svn/spring-samples/mvc-showcase>
- <http://src.springsource.org/svn/spring-samples>

■ Forum

- <http://forum.springframework.org>

■ Issue Tracker

- <http://jira.springsource.org/browse/SPR>

■ Blog

- <http://blog.springsource.com>

■ Twitter

- Follow @kdonald, @poutsma, @springrod, @benalexau

Enjoy being an application developer!
Questions?