18.2 Creating RESTful services

Spring's annotation-based MVC framework serves as the basis for creating RESTful Web Services. As such, you configure your servlet container as you would for a Spring MVC application using Spring's DispatcherServlet.

18.2.1 URI Templates

RESTful services use URIs to name resources. To facilitate accessing the information contained in a URI, its structure follows conventions so that it can easily be described in a parameterized form. The proposed RFC for URI Templates defines how an URI is parameterized. For example, the URI Template

```
http://www.example.com/users/{userid}
```

contains the variable 'userid'. If we assign the variable the value "fred", then 'expanding' the URI Template gives.

```
http://www.example.com/users/fred
```

When processing a request the URI can be compared to an expected URI Template in order to extract a collection of variables.

Spring uses the `@RequestMapping` method annotation to define the URI Template for the request. The `@PathVariable` annotation is used to extract the value of the template variables and assign their value to a method variable. A Spring controller method to process above example is shown below;

```
@RequestMapping("/users/{userid}", method=RequestMethod.GET)
public String getUser(@PathVariable String userId) {
  // implementation omitted...
}
```

The request `http://www.example.com/users/fred` will bind the userId method parameter to the String value 'fred'.

18.2.1.1 Mapping RESTful URLs with the @PathVariable annotation

The `@PathVariable` method parameter annotation is used to indicate that a method parameter should be bound to the value of a URI template variable.

The following code snippet shows the use of a single `@PathVariable` in a controller method:

```
@RequestMapping("/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
```

```
  model.addAttribute("owner", owner);
  return "displayOwner";
}
```

The URI Template "/owners/{ownerId}" specifies the variable name ownerId. When the controller handles this request, the value of ownerId is set the value in the request URI. For example, when a request comes in for /owners/fred, the value 'fred' is bound to the method parameter `String ownerId`.

The matching of method parameter names to URI Template variable names can only be done if your code is compiled with debugging enabled. If you do have not debugging enabled, you must specify the name of the URI Template variable name to bind to in the @PathVariable annotation. For example

```
@RequestMapping("/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String ownerId, Model model)
  // implementation omitted
}
```

The name of the method parameter does not matter in this case, so you may also use a controller method with the signature shown below

```
@RequestMapping("/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable("ownerId") String theOwner, Model model)
  // implementation omitted
}
```

Multiple @PathVariable annotations can be used to bind to multiple URI Template variables as shown below:

```
@RequestMapping("/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId, @PathVariable String petId,
  Owner owner = ownerService.findOwner(ownderId);
  Pet pet = owner.getPet(petId);
  model.addAttribute("pet", pet);
  return "displayPet";
}
```

The following code snippet shows the use of path variables on a relative path

```
@Controller
@RequestMapping("/owners/{ownerId}/**")
public class RelativePathUriTemplateController {

  @RequestMapping("/pets/{petId}")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId,
    // implementation omitted
  }
}
```

Tip

method parameters that are decorated with the `@PathVariable` annotation can be of **any simple type** such as int, long, Date... Spring automatically converts to the appropriate type and throws a `TypeMismatchException` if the type is not correct.

18.2.1.2 Mapping the request body with the @RequestBody annotation

The `@RequestBody` method parameter annotation is used to indicate that a method parameter should be bound to the value of the HTTP request body. For example,

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException
  writer.write(body);
}
```

The conversion of the request body to the method argument is done using a `HttpMessageConverter`. `HttpMessageConverter` is responsible for converting for converting from the HTTP request message to an object and converting from an object to the HTTP response body. `DispatcherServlet` supports annotation based processing using the `DefaultAnnotationHandlerMapping` and `AnnotationMethodHandlerAdapter`. In Spring 3 the `AnnotationMethodHandlerAdapter` has been extended to support the `@RequestBody` and has several `HttpMessageConverters` registered by default, these are

- `ByteArrayHttpMessageConverter` - converts byte arrays
- `StringHttpMessageConverter` - converts strings
- `FormHttpMessageConverter` - converts form data to/from a MultiValueMap<String, String>
- `SourceHttpMessageConverter` - convert to/from a javax.xml.transform.Source;
- `MarshallingHttpMessageConverter` - convert to/from an object using the `org.springframework.oxm` package.

More information on these converters can be found in the section Message Converters.

The `MarshallingHttpMessageConverter` requires a `Marshaller` and `Unmarshaller` from the `org.springframework.oxm` package to be configured on an instance of `AnnotationMethodHandlerAdapter` in the application context. For example

```
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHar
    <property name="messageConverters">
      <util:list id="beanList">
        <ref bean="stringHttpMessageConverter"/>
        <ref bean="marshallingHttpMessageConverter"/>
      </util:list>
    </property>
</bean>

<bean id="stringHttpMessageConverter"
       class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"
      class="org.springframework.http.converter.xml.MarshallingHttpMessageConve
  <property name="marshaller" ref="castorMarshaller" />
  <property name="unmarshaller" ref="castorMarshaller" />
</bean>
```

```
<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarsha
```

18.2.2 Returning multiple representations

A RESTful architecture may expose multiple representations of a resource. There are two strategies for a client to inform the server of the representation it is interested in receiving.

The first strategy is to use a distinct URI for each resource. This is typically done by using a different file extension in the URI. For example the URI `http://www.example.com/users/fred.pdf` requests a PDF representation of the user fred while `http://www.example.com/users/fred.xml` requests an XML representation.

The second strategy is for the client to use the same URI to locate the resource but set the `Accept` HTTP request header to list the media types that it understands. For example, a HTTP request for `http://www.example.com/users/fred` with an `Accept` header set to `application/pdf` requests a PDF representation of the user fred while `http://www.example.com/users/fred` with an `Accept` header set to `text/xml` requests an XML representation. This strategy is known as content negotiation.

Note

One issue with the Accept header is that is impossible to change it in a browser, in HTML. For instance, in Firefox, it's fixed to

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
```

For this reason it is common to see the use of a distinct URI for representation.

To support multiple representations of a resource Spring provides the `ContentNegotiatingViewResolver` to resolve a view based on the file extension or `Accept` header of the HTTP request. `ContentNegotiatingViewResolver` does not perform the view resolution itself, but instead delegates to a list of view resolvers set using the bean property `ViewResolvers`.

The `ContentNegotiatingViewResolver` selects an appropriate `View` to handle the request by comparing the request media type(s) with the media type (a.k.a. `Content-Type`) supported by the `View` associated with each of its `ViewResolvers`. The first `View` in the list that has a compatible `Content-Type` is used to return the representation to the client. The `Accept` header may include wild cards, for example 'text/*', in which case a `View` whose Context-Type was 'text/xml' is a compatible match.

To support the resolution of a view based on a file extension, `ContentNegotiatingViewResolver`'s bean property `MediaTypes` is used to specify a mapping of file extensions to media types. For more information on the algorithm to determine the request media type, refer to the API documentation for `ContentNegotiatingViewResolver`..

Here is an example configuration of a `ContentNegotiatingViewResolver`

```
    <bean class="org.springframework.web.servlet.view.ContentNegotiatingViewRes
        <property name="mediaTypes">
            <map>
                <entry key="atom" value="application/atom+xml"/>
                <entry key="html" value="text/html"/>
            </map>
        </property>
        <property name="viewResolvers">
            <list>
                <bean class="org.springframework.web.servlet.view.BeanNameViewR
                <bean class="org.springframework.web.servlet.view.InternalResou
                    <property name="prefix" value="/WEB-INF/jsp/"/>
                    <property name="suffix" value=".jsp"/>
                </bean>
            </list>
        </property>
    </bean>


    <bean id="content" class="com.springsource.samples.rest.SampleContentAtomV
```

The `InternalResourceViewResolver` handles the translation of view names and JSP pages while the `BeanNameViewResolver` returns a view based on the name of a bean. (See "Resolving views - the ViewResolver interface" for more details on how Spring looks up and instantiates a view.) In this example, the `content` bean is a class that inherits from `AbstractAtomFeedView` which returns an Atom RSS feed. For more information on creating an Atom Feed representation see the section 'Atom Views'.

In this configuration, if a request is made with a .html extension the view resolver will look for a view that matches the text/html media type. The `InternalResourceViewResolver` provides the matching view for text/html. If the request is made with the file extension .atom, the view resolver will look for a view that matches the application/atom+xml media type. This view is provided by the `BeanNameViewResolver` that maps to the `SampleContentAtomView` if the view name returned is 'content'. Alternatively, client requests could be made without a file extension and setting the Accept header to the preferred media-type and the same resolution of request to views would occur.

Note

If `ContentNegotiatingViewResolver`'s list of ViewResolvers is not configured explicitly, then it will automatically use any ViewResolvers defined in the application context.

The corresponding controller code that returns an Atom RSS feed for a URI of the form `http://localhost/content.atom` or `http://localhost/content` with an `Accept` header of application/atom+xml is shown below

```
@Controller
public class ContentController {

    private List<SampleContent> contentList = new ArrayList<SampleContent>();

    @RequestMapping(value="/content", method=RequestMethod.GET)
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
```

```
        return mav;
    }

}
```

18.2.3 Views

Several views were added in Spring 3 to help support creating RESTful services. They are:

- `AbstractAtomFeedView` - return an Atom feed
- `AbstractRssFeedView` - returns a RSS feed
- `MarshallingView` - returns an XML representation using Spring's Object to XML mapping (OXM) functionality

Note

Available separately is the `JacksonJsonView` included as part of the Spring JavaScript project.

18.2.3.1 Feed Views

Both `AbstractAtomFeedView` and `AbstractRssFeedView` inherit from the base class `AbstractFeedView` and are used to provide Atom and RSS Feed views respectfully. They are based on java.net's ROME project and are located in the package `org.springframework.web.servlet.view.feed`.

`AbstractAtomFeedView` requires you to implement the `buildFeedEntries` method and optionally override the `buildFeedMetadata` method (the default implementation is empty), as shown below

```
public class SampleContentAtomView extends AbstractAtomFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Feed feed,
                                   HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Entry> buildFeedEntries(Map<String, Object> model,
                                         HttpServletRequest request,
                                         HttpServletResponse response) throws I

   // implementation omitted
  }
}
```

Similar requirements apply for implementing `AbstractRssFeedView`, as shown below

```
public class SampleContentAtomView extends AbstractRssFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Channel feed,
                                   HttpServletRequest request) {
    // implementation omitted
  }
```

```
  @Override
  protected List<Item> buildFeedItems(Map<String, Object> model,
                                      HttpServletRequest request,
                                      HttpServletResponse response) throws Exce
    // implementation omitted
  }

}
```

The `buildFeedItems` and `buildFeedEntires` pass in the HTTP request in case you need to access the Locale. The HTTP response in passed in only for the setting of cookies or other HTTP headers. The feed will automatically be written to the response object after the method returns.

For an example of creating a Atom view please refer to Alef Arendsen's SpringSource TeamBlog entry.

18.2.3.2 XML Marshalling View

The `MarhsallingView` uses a XML `Marshaller` defined in the `org.springframework.oxm` package to render the response content as XML. The object to be marshalled can be set explicitly using `MarhsallingView`'s modelKey bean property. Alternatively, the view will iterate over all model properties marhsall only those types that are supported by the `Marshaller`.

18.2.4 HTTP Method Conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP method: GET, PUT, POST, and DELETE. For each methods, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring's `HiddenHttpMethodFilter` does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden _method parameter will be converted into the corresponding HTTP method request.

18.2.4.1 Supporting Spring form tags

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

This will actually perform an HTTP POST, with the 'real' DELETE method

hidden behind a request parameter, to be picked up by the `HiddenHttpMethodFilter`. The corresponding @Controller method is shown below

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

18.2.5 ETag support

An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. It can be considered to be the more sophisticated successor to the `Last-Modified` header. When a server returns a representation with an ETag header, client can use this header in subsequent GETs, in a `If-None-Match` header. If the content has not changed, the server will return `304: Not Modified`.

Support for ETags is provided by the servlet filter `ShallowEtagHeaderFilter`. Since it is a plain Servlet Filter, and thus can be used in combination any web framework. The `ShallowEtagHeaderFilter` filter creates so-called shallow ETags (as opposed to a deep ETags, more about that later). The way it works is quite simple: the filter simply caches the content of the rendered JSP (or other content), generates a MD5 hash over that, and returns that as a ETag header in the response. The next time a client sends a request for the same resource, it use that hash as the `If-None-Match` value. The filter notices this, renders the view again, and compares the two hashes. If they are equal, a `304` is returned. It is important to note that this filter will not save processing power, as the view is still rendered. The only thing it saves is bandwidth, as the rendered response is not sent back over the wire.

Deep ETags are a bit more complicated. In this case, the ETag is based on the underlying domain objects, RDMBS tables, etc. Using this approach, no content is generated unless the underlying data has changed. Unfortunately, implementing this approach in a generic way is much more difficult than shallow ETags. Spring may provide support for deep ETags in a later release by relying on JPA's @Version annotation, or an AspectJ aspect.

18.2.6 Exception Handling

The `@ExceptionHandling` method annotation is used within a controller to specify which method will be invoked when an exception of a specific type is thrown during the execution of controller methods. For example

```
@Controller
public class SimpleController {

 // other controller method omitted

 @ExceptionHandler(IOException.class)
 public String handleIOException(IOException ex, HttpServletRequest request) {
  return ClassUtils.getShortName(ex.getClass());
 }
}
```

will invoke the 'handlerIOException' method when a `java.io.IOException` is thrown.

The `@ExceptionHandler` value can be set to an array of Exception types. If an exception is thrown matches one of the types in the list, then the method annotated with the matching `@ExceptionHandler` will be invoked. If the annotation value is not set then the exception types listed as method arguments are used.

Much like standard controller methods annotated with a `@RequestMapping` annotation, the method arguments and return values of `@ExceptionHandler` methods are very flexible. For example, the `HttpServletRequest` can be access in Servlet environments and the `PortletRequest` in Portlet environments. The return type can be a `String`, which is interpreted as a view name or a `ModelAndView` object. Please refer to the API documentation for more details.

Prev | Up | Next
18. REST support | Home | 18.3 Accessing RESTful services on the Client