Spring REST

Marten Deinum1, Daniel Rubio2 and Josh Long3

(1)Meppel, Drenthe, The Netherlands

(2)F. Bahia, Ensenada, Baja California, Mexico

(3)Apartment 205, Canyon Country, California, USA

In this chapter, you will learn how Spring addresses Representational State Transfer, usually referred to by its acronym REST. REST has had an important impact on web applications since the term was coined by Roy Fielding (<http://en.wikipedia.org/wiki/Roy_Fielding>) in 2000.

Based on the foundations of the Web’s Hypertext Transfer Protocol (HTTP), the architecture set forth by REST has become increasingly popular in the implementation of web services. Web services in and of themselves have become the cornerstone for much machine-to-machine communication taking place on the Web. It’s the fragmented technology choices (e.g., Java, Python, Ruby, .NET) made by many organizations that have necessitated a solution capable of bridging the gaps between these disparate environments. For example, how is information in an application backed by Java accessed by one written in Python? How can a Java application obtain information from an application written in .NET? Web services fill this void.

There are various approaches to implementing web services, but RESTful web services have become the most common choice in web applications. They are used by some of the largest Internet sites (e.g., Google and Yahoo) to provide access to their information, to back access to Ajax calls made by browsers, and to provide the foundations for the distribution of information such as news feeds (e.g., RSS).

In this chapter, you will learn how Spring applications can use REST so that you can both access and provide information using this popular approach.

4-1. Publish XML with REST Services

PROBLEM

You want to publish an XML-based REST service with Spring.

SOLUTION

There are two possibilities when designing REST services in Spring. One involves publishing an application’s data as a REST service; the other involves accessing data from third-party REST services to be used in an application. This recipe describes how to publish an application’s data as a REST service. Recipe 4-2 describes how to access data from third-party REST services. Publishing an application’s data as a REST service revolves around the use of the Spring MVC annotations @RequestMapping and @PathVariable. By using these annotations to decorate a Spring MVC handler method, a Spring application is capable of publishing an application’s data as a REST service.

In addition, Spring supports a series of mechanisms to generate a REST service’s payload. This recipe will explore the simplest mechanism, which involves the use of Spring’s MarshallingView class. As the recipes in this chapter progress, you will learn about more advanced mechanisms supported by Spring to generate REST service payloads.

HOW IT WORKS

Publishing a web application’s data as a REST service (or as it’s more technically known in web services parlance, “creating an endpoint”) is strongly tied to Spring MVC, which you explored in Chapter [3](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_3_Chapter.html). Since Spring MVC relies on the annotation @RequestMapping to decorate handler methods and define access points (i.e., URLs), it’s the preferred way in which to define a REST service’s endpoint.

*Use MarshallingView to Produce XML*

The following code illustrates a Spring MVC controller class with a handler method that defines a REST service endpoint:

**package** com.apress.springrecipes.court.web;  
  
**import** com.apress.springrecipes.court.domain.Members;  
**import** com.apress.springrecipes.court.service.MemberService;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Controller;  
**import** org.springframework.ui.Model;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
**public class** RestMemberController {  
  
    **private** final MemberService memberService;  
  
    @Autowired  
    **public** RestMemberController(MemberService memberService) {  
        **super**();  
        **this**.memberService=memberService;  
    }  
  
    @RequestMapping("/members")  
    **public** String getRestMembers(Model model) {  
        Members members = **new** Members();  
        members.addMembers(memberService.findAll());  
        model.addAttribute("members", members);  
        **return** "membertemplate";  
    }  
}

By using @RequestMapping("/members") to decorate a controller’s handler method, a REST service endpoint is made accessible at host\_name/[app-name]/members. You can observe that control is relinquished to a logical view named membertemplate. The following code illustrates the declaration used to define the logical view named membertemplate:

@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.apress.springrecipes.court")  
**public class** CourtRestConfiguration {  
  
    @Bean  
    **public** View membertemplate() {  
        **return new** MarshallingView(jaxb2Marshaller());  
    }  
  
    @Bean  
    **public** Marshaller jaxb2Marshaller() {  
        Jaxb2Marshaller marshaller = **new** Jaxb2Marshaller();  
        marshaller.setClassesToBeBound(Members.class, Member.class);  
        **return** marshaller;  
    }  
  
    @Bean  
    **public** ViewResolver viewResolver() {  
        **return new** BeanNameViewResolver();  
    }  
}

The membertemplate view is defined as a MarshallingView type, which is a general-purpose class that allows a response to be rendered using a marshaller. *Marshalling* is the process of transforming an in-memory representation of an object into a data format. Therefore, for this particular case, a marshaller is charged with transforming Members and Member objects into an XML data format. The marshaller used by MarshallingView belongs to one of a series of XML marshallers provided by Spring—Jaxb2Marshaller. Other marshallers provided by Spring include CastorMarshaller, JibxMarshaller, XmlBeansMarshaller, and XStreamMarshaller.

Marshallers themselves also require configuration. We opted to use the Jaxb2Marshaller marshaller because of its simplicity and Java Architecture for XML Binding (JAXB) foundations. However, if you’re more comfortable using the Castor XML framework, you might find it easier to use CastorMarshaller; if you’re more at ease using XStream, you would likely find it easier to use XStreamMarshaller; and it’s similar for the rest of the available marshallers.

The Jaxb2Marshaller marshaller needs to be configured with either a property named classesToBeBound or a property named contextPath. In the case of classesToBeBound, the classes assigned to this property indicate the class (i.e., object) structure that is to be transformed into XML. The following code illustrates the Members and Member classes assigned to the Jaxb2Marshaller marshaller:

**package** com.apress.springrecipes.court.domain;  
  
**import** javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement  
**public class** Member {  
    **private** String name;  
    **private** String phone;  
    **private** String email;  
  
    **public** String getEmail() {  
        **return** email;  
    }  
  
    **public** String getName() {  
        **return** name;  
    }  
  
    **public** String getPhone() {  
        **return** phone;  
    }  
  
    **public** void setEmail(String email) {  
        **this**.email = email;  
    }  
  
    **public** void setName(String name) {  
        **this**.name = name;  
    }  
  
    **public** void setPhone(String phone) {  
        **this**.phone = phone;  
    }  
}

Here’s the Members class:

**package** com.apress.springrecipes.court.domain;  
  
**import** javax.xml.bind.annotation.XmlAccessType;  
**import** javax.xml.bind.annotation.XmlAccessorType;  
**import** javax.xml.bind.annotation.XmlElement;  
**import** javax.xml.bind.annotation.XmlRootElement;  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.List;  
  
@XmlRootElement  
@XmlAccessorType(XmlAccessType.FIELD)  
**public class** Members {  
  
    @XmlElement(name="member")  
    **private** List<Member> members = **new** ArrayList<>();  
  
    **public** List<Member> getMembers() {  
        **return** members;  
    }  
  
    **public** void setMembers(List<Member> members) {  
        **this**.members = members;  
    }  
   
    **public** void addMembers(Collection<Member> members) {  
        **this**.members.addAll(members);  
    }  
}

Note the Member class is a POJO decorated with the @XmlRootElement annotation. This annotation allows the Jaxb2Marshaller marshaller to detect a class’s (i.e., object’s) fields and transform them into XML data (e.g., name=John into <name>john</name>, email=john@doe.com into <email>john@doe.com</email>).

To recap what’s been described, this means that when a request is made to a URL in the form http://[host\_name]//app-name]/members.xml, the corresponding handler is charged with creating a Members object, which is then passed to a logical view named membertemplate. Based on this last view’s definition, a marshaller is used to convert a Members object into an XML payload that is returned to the REST service’s requesting party. The XML payload returned by the REST service is illustrated in the following code:

**<?xml** version="1.0" encoding="UTF-8" standalone="yes"**?>**   
**<members>**   
    **<member>**   
        **<email>**marten@deinum.biz**</email>**   
        **<name>**Marten Deinum**</name>**   
        **<phone>**00-31-1234567890**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**john@doe.com**</email>**   
        **<name>**John Doe**</name>**   
        **<phone>**1-800-800-800**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**jane@doe.com**</email>**   
        **<name>**Jane Doe**</name>**   
        **<phone>**1-801-802-803**</phone>**   
    **</member>**   
**</members>**

This XML payload represents a simple approach to generating a REST service’s response. As the recipes in this chapter progress, you will learn more sophisticated approaches, such as the ability to create widely used REST service payloads such as RSS, Atom, and JSON.

If you look closely at the REST service endpoint or URL described in the previous paragraph, you’ll note that it has an .xml extension. If you try another extension—or even omit the extension—this particular REST service may not be triggered. This last behavior is directly tied to Spring MVC and how it handles view resolution. It has nothing do with REST services per se.

By default, since the view associated with this particular REST service handler method returns XML, it’s triggered by an .xml extension. This allows the same handler method to support multiple views. For example, it can be convenient for a request like http://[host\_name]/[app-name]/members.pdf to return the same information in a PDF document, as well as a request like http://[host\_name]/[app-name]/members.html to return content in HTML or a request like http://[host\_name]/[app-name]/members.xml to return XML for a REST request.

So, what happens to a request with no URL extension, like http://[host\_name]/[app-name]/members? This also depends heavily on Spring MVC view resolution. For this purpose, Spring MVC supports a process called *content negotiation*, by which a view is determined based on a request’s extension or HTTP headers.

Since REST service requests typically have HTTP headers in the form Accept: application/xml, Spring MVC configured to use content negotiation can determine to serve XML (REST) payloads to such requests even if requests are made extensionless. This also allows extensionless requests to be made in formats such as HTML, PDF, and XLS, all simply based on HTTP headers. Recipe 3-7 in Chapter [3](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_3_Chapter.html) discusses content negotiation.

*Use @ResponseBody to Produce XML*

Using MarshallingView to produce XML is one way of producing results; however, when you want to have multiple representations (JSON, for instance) of the same data (a list of Member objects), adding another view can be a cumbersome task. Instead, you can rely on the Spring MVC HttpMessageConverters to convert an object to the representation requested by the user. The following code shows the changes made to RestMemberController:

@Controller  
**public class** RestMemberController {  
...  
    @RequestMapping("/members")  
    @ResponseBody  
    **public** Members getRestMembers() {  
        Members members = **new** Members();  
        members.addMembers(memberService.findAll());  
        **return** members;  
    }  
}

The first change is that you have now, additionally , annotated the controller method with @ResponseBody. This annotation tells Spring MVC that the result of the method should be used as the body of the response. Because you want XML, this marshalling is done by the Jaxb2RootElementHttpMessageConverter class provided by Spring. The second change is that because of the @ResponseBody annotation, you don’t need the view name anymore but can simply return the Members object.

**Tip**

When using Spring 4 or higher instead of annotating the method with @ResponseBody, you can also annotate your controller with @RestController instead of @Controller, which would give the same result. This is especially convenient if you have a single controller with multiple methods.

These changes also allow you to clean up your configuration, as you don’t need MarshallingView and Jaxb2Marshaller anymore.

**package** com.apress.springrecipes.court.web.config;  
  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;  
  
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.apress.springrecipes.court")  
**public class** CourtRestConfiguration {}

When the application is deployed and you do request the members from http://localhost:8080/court/members.xml, it will yield the same results as before.

**<?xml** version="1.0" encoding="UTF-8" standalone="yes"**?>**   
**<members>**   
    **<member>**   
        **<email>**marten@deinum.biz**</email>**   
        **<name>**Marten Deinum**</name>**   
        **<phone>**00-31-1234567890**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**john@doe.com**</email>**   
        **<name>**John Doe**</name>**   
        **<phone>**1-800-800-800**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**jane@doe.com**</email>**   
        **<name>**Jane Doe**</name>**   
        **<phone>**1-801-802-803**</phone>**   
    **</member>**   
**</members>**

*Use @PathVariable to Limit the Results*

It’s common for REST service requests to have parameters. This is done to limit or filter a service’s payload. For example, a request in the form http://[host\_name]/[app-name]/member/353/ can be used to retrieve information exclusively on member 353. Another variation can be a request like http://[host\_name]/[app-name]/reservations/07-07-2010/ to retrieve reservations made on the date July 7, 2010.

To use parameters for constructing a REST service in Spring, you use the @PathVariable annotation. The @PathVariable annotation is added as an input parameter to the handler method, per Spring’s MVC conventions, for it to be used inside the handler method body. The following snippet illustrates a handler method for a REST service using the @PathVariable annotation:

**import** org.springframework.web.bind.annotation.PathVariable;  
  
@Controller  
**public class** RestMemberController {  
...  
    @RequestMapping("/member/{memberid}")  
    @ResponseBody  
    **public** Member getMember(@PathVariable("memberid") long memberID) {  
        **return** memberService.find(memberID);  
    }  
}

Notice the @RequestMapping value contains {memberid}. Values surrounded by { } are used to indicate that URL parameters are variables. Further note that the handler method is defined with the input parameter @PathVariable("memberid") long memberID. This last declaration associates whatever memberid value forms part of the URL and assigns it to a variable named memberID that can be accessible inside the handler method. Therefore, REST endpoints in the form /member/353/ and /member/777/ will be processed by this last handler method, with the memberID variable being assigned values of 353 and 777, respectively. Inside the handler method, the appropriate queries can be made for members 353 and 777—via the memberID variable—and returned as the REST service’s payload.

A request to http://localhost:8080/court/member/2 will result in an XML representation of the member with an ID of 2.

**<?xml** version="1.0" encoding="UTF-8" standalone="yes"**?>**   
**<member>**   
    **<email>**john@doe.com**</email>**   
    **<name>**John Doe**</name>**   
    **<phone>**1-800-800-800**</phone>**   
**</member>**

In addition to the supporting { } notation, it’s also possible to use a wildcard (\*) notation for defining REST endpoints. This is often the case when a design team has opted to use expressive URLs (often called *pretty URLs*) or opts to use search engine optimization (SEO) techniques to make a REST URL search engine friendly. The following snippet illustrates a declaration for a REST service using the wildcard notation:

@RequestMapping("/member/\*/{memberid}")  
@ResponseBody  
**public** Member getMember(@PathVariable("memberid") long memberID) { ... }

In this case, the addition of a wildcard doesn’t have any influence over the logic performed by the REST service. But it will match endpoint requests in the form of /member/John+Smith/353/ and /member/Mary+Jones/353/, which can have an important impact on end user readability and SEO.

It’s also worth mentioning that data binding can be used in the definition of handler methods for REST endpoints. The following snippet illustrates a declaration for a REST service using data binding:

@InitBinder  
**public** void initBinder(WebDataBinder binder) {  
    SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");  
    binder.registerCustomEditor(Date.class, **new** CustomDateEditor(dateFormat, **false**));  
}  
  
@RequestMapping("/reservations/{date}")  
**public** void getReservation(@PathVariable("date") Date resDate) { ... }

In this case, a request in the form http://[host\_name]/[app-name]/reservations/07-07-2010/ is matched by this last handler method, with the value 07-07-2010 passed into the handler method—as the variable resDate—where it can be used to filter the REST web service payload.

*Use ResponseEntity to Inform the Client*

The endpoint for retrieving a single Member instance returns either a valid member or nothing at all. Both lead to a request that will send the HTTP response code 200, which means OK, back to the client. However, this is probably not what your users will expect. When working with resources, you should inform them of the fact that a resource cannot be found. Ideally, you would want to return the HTTP response code 404, which indicates “not found.” The following code snippet shows the modified getMember method :

**package** com.apress.springrecipes.court.web;  
  
**import** org.springframework.http.HttpStatus;  
**import** org.springframework.http.ResponseEntity;  
...  
@Controller  
**public class** RestMemberController {  
...  
    @RequestMapping("/member/{memberid}")  
    @ResponseBody  
    **public** ResponseEntity<Member> getMember(@PathVariable("memberid") long memberID) {  
        Member member = memberService.find(memberID);  
        **if** (member != **null**) {  
            **return new** ResponseEntity<Member>(member, HttpStatus.OK);  
        }  
        **return new** ResponseEntity(HttpStatus.NOT\_FOUND);  
    }  
}

The return value of the method has been changed to ResponseEntity<Member>. The ResponseEntity class in Spring MVC acts as a wrapper for an object to be used as the body of the result together with an HTTP status code. When you find a Member, it is returned with HttpStatus.OK, which corresponds to an HTTP status code of 200. When there is no result, you return HttpStatus.NOT\_FOUND, corresponding to the HTTP status code 404, which means “not found.”

4-2. Publish JSON with REST Services

PROBLEM

You want to publish a JavaScript Object Notation (JSON)–based REST service with Spring.

SOLUTION

JSON has blossomed into a favorite payload format for REST services. However, unlike most REST service payloads, which rely on XML markup, JSON is different in the sense that its content is a special notation based on the JavaScript language. For this recipe, in addition to relying on Spring’s REST support, you will also use the MappingJackson2JsonView class that forms part of Spring to facilitate the publication of JSON content.

**Note**

The MappingJackson2JsonView class depends on the presence of the Jackson JSON processor library, version 2, which can be downloaded at <http://wiki.fasterxml.com/JacksonDownload>. If you are using Maven or Gradle you can simply add the Jackson library as a dependency in your projects build file.

If your Spring applications incorporate Ajax designs, it’s likely that you’ll find yourself designing REST services that publish JSON as their payload. This is mainly because of the limited processing capabilities in browsers. Although browsers can process and extract information from REST services that publish XML payloads, it’s not very efficient. By instead delivering payloads in JSON, which is based on a language for which browsers have a native interpreter—JavaScript—the processing and extraction of data becomes more efficient. Unlike RSS and Atom feeds, which are standards, JSON has no specific structure it needs to follow—except its syntax, which you’ll explore shortly. Therefore, a JSON element’s payload structure is likely to be determined in coordination with the team members charged with an application’s Ajax design.

HOW IT WORKS

The first thing you need to do is determine the information you want to publish as a JSON payload. This information can be located in an RDBMS or text file, be accessed through JDBC or ORM, or inclusively be part of a Spring bean or some other type of construct. Describing how to obtain this information would go beyond the scope of this recipe, so we will assume you’ll use whatever means you deem appropriate to access it. In case you’re unfamiliar with JSON, the following snippet illustrates a fragment of this format:

{  
    "glossary": {  
        "title": "example glossary",  
        "GlossDiv": {  
            "title": "S",  
            "GlossList": {  
                "GlossEntry": {  
                    "ID": "SGML",  
                    "SortAs": "SGML",  
                    "GlossTerm": "Standard Generalized Markup Language",  
                    "Acronym": "SGML",  
                    "Abbrev": "ISO 8879:1986",  
                    "GlossDef": {  
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",  
                        "GlossSeeAlso": ["GML", "XML"]  
                    },  
                    "GlossSee": "markup"  
                }  
            }  
        }  
    }  
}

As you can observe, a JSON payload consists of text and separators such as { , } ,[ , ] , :, and ". We won’t go into details about using one separator over another, but it suffices to say this type of syntax makes it easier for a JavaScript engine to access and manipulate data than if it was to process it in an XML-type format.

*Use MappingJackson2JsonView to Produce XML*

Since you’ve already explored how to publish data using a REST service in recipes 4-1 and 4-3, we’ll cut to the chase and show you the actual handler method needed in a Spring MVC controller to achieve this process.

@RequestMapping("/members")  
**public** String getRestMembers(Model model) {  
  
    Members members = **new** Members();  
    members.addMembers(memberService.findAll());  
    model.addAttribute("members", members);  
    **return** "jsonmembertemplate";  
}

You probably notice that it is quite similar to the controller method mentioned in recipe 4-1. The only difference is that you return a different name for the view. The name of the view you are returning here, jsonmembertemplate, is different and maps to a MappingJackson2JsonView view. You need to configure this view in your configuration class.

@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.apress.springrecipes.court")  
**public class** CourtRestConfiguration {  
    ...  
    @Bean  
    **public** View jsonmembertemplate() {  
        MappingJackson2JsonView view = **new** MappingJackson2JsonView();  
        view.setPrettyPrint(**true**);  
        **return** view;  
    }  
}

The MappingJackson2JsonView view uses the Jackson2 library to convert objects to and from JSON. It uses a Jackson2 ObjectMapper instance for the conversion. When a request is made to http://localhost:8080/court/members.json, the controller method will be invoked, and a JSON representation will be returned.

{  
    "members" : {  
        "members" : [ {  
            "name" : "Marten Deinum",  
            "phone" : "00-31-1234567890",  
            "email" : "marten@deinum.biz"  
        }, {  
            "name" : "John Doe",  
            "phone" : "1-800-800-800",  
            "email" : "john@doe.com"  
        }, {  
            "name" : "Jane Doe",  
            "phone" : "1-801-802-803",  
            "email" : "jane@doe.com"  
        } ]  
    }  
}

Actually, this JSON will be produced by each call to /members or /members.\* (for instance, /members.xml will also produce JSON). Let’s add the method and view from recipe 4-1 to the controller.

@Controller  
**public class** RestMemberController {  
...  
    @RequestMapping(value="/members", produces=MediaType.APPLICATION\_XML\_VALUE)  
    **public** String getRestMembersXml(Model model) {  
        Members members = **new** Members();  
        members.addMembers(memberService.findAll());  
        model.addAttribute("members", members);  
        **return** "xmlmembertemplate";  
    }  
  
    @RequestMapping(value="/members", produces= MediaType.APPLICATION\_JSON\_VALUE)  
    **public** String getRestMembersJson(Model model) {  
        Members members = **new** Members();  
        members.addMembers(memberService.findAll());  
        model.addAttribute("members", members);  
        **return** "jsonmembertemplate";  
    }  
}

You have now a getMembersXml method and a getMembersJson method; both are basically the same with the distinction that they return a different view name. Notice the produces attribute on the @ RequestMapping annotation. This is used to determine which method to call: /members.xml will now produce XML, whereas /members.json will produce JSON.

Although this approach works, duplicating all the methods for the different supported view types isn’t a feasible solution for enterprise applications. You could create a helper method to reduce the duplication, but you would still need a lot of boilerplate because of the differences in the @RequestMapping annotations.

*Use @ResponseBody to Produce JSON*

Using a MappingJackson2JsonView to produce JSON is one way of producing results; however, as mentioned in the previous section, it can be troublesome, especially with multiple supported view types. Instead, you can rely on the Spring MVC HttpMessageConverters to convert an object to the representation requested by the user. The following code shows the changes made to RestMemberController:

@Controller  
**public class** RestMemberController {  
...  
    @RequestMapping("/members")  
    @ResponseBody  
    **public** Members getRestMembers() {  
        Members members = **new** Members();  
        members.addMembers(memberService.findAll());  
        **return** members;  
    }  
}

The first change is that you have now, additionally, annotated the controller method with @ResponseBody. This annotation tells Spring MVC that the result of the method should be used as the body of the response. Because you want JSON, this marshalling is done by the Jackson2JsonMessageConverter class provided by Spring. The second change is that because of the @ResponseBody annotation, you don’t need the view name anymore but can simply return the Members object .

**Tip**

When using Spring 4 or higher instead of annotating the method with @ResponseBody, you can also annotate your controller with @RestController instead of @Controller, which would give the same result. This is especially convenient if you have a single controller with multiple methods.

These changes also allow you to clean up your configuration because you don’t need MappingJackson2JsonView anymore.

**package** com.apress.springrecipes.court.web.config;  
  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;  
  
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.apress.springrecipes.court")  
**public class** CourtRestConfiguration {}

When the application is deployed and you request the members from http://localhost:8080/court/members.json, it will give the same results as before.

{  
    "members" : {  
        "members" : [ {  
            "name" : "Marten Deinum",  
            "phone" : "00-31-1234567890",  
            "email" : "marten@deinum.biz"  
        }, {  
            "name" : "John Doe",  
            "phone" : "1-800-800-800",  
            "email" : "john@doe.com"  
        }, {  
            "name" : "Jane Doe",  
            "phone" : "1-801-802-803",  
            "email" : "jane@doe.com"  
        } ]  
    }  
}

You probably noticed that RestMemberController and CourtRestConfiguration are now the same as in recipe 4-1. When calling http://localhost:8080/court/members.xml, you will get XML.

How is this possible without any additional configuration? Spring MVC will detect what is on the classpath; when it automatically detects JAXB 2, Jackson, and Rome (see recipe 4-4), it will register the appropriate HttpMessageConverter for the available technologies.

*Use GSON to Produce JSON*

Up until now you have been using Jackson to produce JSON from your objects; another popular library is GSON, and Spring has out-of-the-box support for it. To use GSON, you will need to add it to your classpath (instead of Jackson), and then it will be used to produce the JSON.

When using Maven, add the following dependency:

**<dependency>**   
    **<groupId>**com.google.code.gson**</groupId>**   
    **<artifactId>**gson**</artifactId>**   
    **<version>**2.8.0**</version>**   
**</dependency>**

When using Gradle, add the following:

compile 'com.google.code.gson:gson:2.8.0'

This, just like when using Jackson, is all you need to implement JSON serialization with GSON. If you start the application and call http://localhost:8080/court/members.json, you will still receive JSON but now through GSON instead.

4-3. Access a REST Service with Spring

PROBLEM

You want to access a REST service from a third party (e.g., Google, Yahoo, or another business partner) and use its payload inside a Spring application.

SOLUTION

Accessing a third-party REST service inside a Spring application revolves around the use of the Spring RestTemplate class. The RestTemplate class is designed on the same principles as many other Spring \*Template classes (e.g., JdbcTemplate, JmsTemplate), providing a simplified approach with default behaviors for performing lengthy tasks. This means the processes of invoking a REST service and using its returning payload are streamlined in Spring applications.

HOW IT WORKS

Before describing the particularities of the RestTemplate class, it’s worth exploring the life cycle of a REST service so you’re aware of the actual work the RestTemplate class performs. Exploring the life cycle of a REST service can best be done from a browser, so open your favorite browser on your workstation to get started. The first thing that’s needed is a REST service endpoint. You are going to reuse the endpoint you created in recipe 4-2. This endpoint should be available at http://localhost:8080/court/members.xml (or .json). If you load this last REST service endpoint on your browser, the browser performs a GET request, which is one of the most popular HTTP requests supported by REST services. Upon loading the REST service, the browser displays a responding payload like the following:

**<?xml** version="1.0" encoding="UTF-8" standalone="yes"**?>**   
**<members>**   
    **<member>**   
        **<email>**marten@deinum.biz**</email>**   
        **<name>**Marten Deinum**</name>**   
        **<phone>**00-31-1234567890**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**john@doe.com**</email>**   
        **<name>**John Doe**</name>**   
        **<phone>**1-800-800-800**</phone>**   
    **</member>**   
    **<member>**   
        **<email>**jane@doe.com**</email>**   
        **<name>**Jane Doe**</name>**   
        **<phone>**1-801-802-803**</phone>**   
    **</member>**   
**</members>**

This last payload represents a well-formed XML fragment, which is in line with most REST services’ responses. The actual meaning of the payload is highly dependent on a REST service. In this case, the XML tags (<members>, <member>, etc.) are definitions set forth by yourself, while the character data enclosed in each XML tag represents information related to a REST service’s request.

It’s the task of a REST service consumer (i.e., you) to know the payload structure—sometimes referred to as the *vocabulary*—of a REST service to appropriately process its information. Though this last REST service relies on what can be considered a custom vocabulary, a series of REST services often relies on standardized vocabularies (e.g., RSS), which make the processing of REST service payloads uniform. In addition, it’s worth noting that some REST services provide Web Application Description Language (WADL) contracts to facilitate the discovery and consumption of payloads.

Now that you’re familiar with a REST service’s life cycle using your browser, you can take a look at how to use the Spring RestTemplate class to incorporate a REST service’s payload into a Spring application. Given that the RestTemplate class is designed to call REST services, it should come as no surprise that its main methods are closely tied to REST’s underpinnings, which are the HTTP protocol’s methods: HEAD, GET, POST, PUT, DELETE, and OPTIONS. Table [4-1](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_4_Chapter.html#Tab1) contains the main methods supported by the RestTemplate class.

***Table 4-1.*RestTemplate Class Methods Based on HTTP Protocol’s Request Methods**

| **Method** | **Description** |
| --- | --- |
| headForHeaders(String, Object...) | Performs an HTTP HEAD operation |
| getForObject(String, Class, Object...) | Performs an HTTP GET operation and returns the result as a type of the given class |
| getForObject(String, Class, Object...) | Performs an HTTP GET operation and returns a ResponseEntity |
| postForLocation(String, Object, Object...) | Performs an HTTP POST operation and returns the value of the location header |
| postForObject(String, Object, Class, Object...) | Performs an HTTP POST operation and returns the result as a type of the given class |
| postForEntity(String, Object, Class, Object...) | Performs an HTTP POST operation and returns a ResponseEntity |
| put(String, Object, Object...) | Performs an HTTP PUT operation |
| delete(String, Object...) | Performs an HTTP DELETE operation |
| optionsForAllow(String, Object...) | Performs an HTTP OPTIONS operation |
| execute(String, HttpMethod, RequestCallback, ResponseExtractor, Object...) | Can perform any HTTP operation with the exception of CONNECT |

As you can observe in Table [4-1](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_4_Chapter.html#Tab1), the RestTemplate class methods are prefixed with a series of HTTP protocol methods that include HEAD, GET, POST, PUT, DELETE, and OPTIONS. In addition, the execute method serves as a general-purpose method that can perform any HTTP operation, including the more esoteric HTTP protocol TRACE method, except the CONNECT method, which is not supported by the underlying HttpMethod enum used by the execute method. Note that by far the most common HTTP method used in REST services is GET since it represents a safe operation to obtain information (i.e., it doesn’t modify any data). On the other hand, HTTP methods such as PUT, POST, and DELETE are designed to modify a provider’s information, which makes them less likely to be supported by a REST service provider. For cases in which data modification needs to take place, many providers opt for the SOAP protocol, which is an alternative mechanism to using REST services.

Now that you’re aware of the RestTemplate class methods, you can move on to invoking the same REST service you did with your browser previously, except this time using Java code from the Spring Framework. The following code illustrates a class that accesses the REST service and returns its contents to System.out:

**package** com.apress.springrecipes.court;  
  
**import** org.springframework.web.client.RestTemplate;  
  
**public class** Main {  
  
    **public** static void main(String[] args) **throws** Exception {  
        final String uri = "http://localhost:8080/court/members.json";  
        RestTemplate restTemplate = **new** RestTemplate();  
        String result = restTemplate.getForObject(uri, String.class);  
        System.out.println(result);  
    }  
}

**Caution**

Some REST service providers restrict access to their data feeds depending on the requesting party. Access is generally denied by relying on data present in a request (e.g., HTTP headers or IP address). So, depending on the circumstances, a provider can return an access denied response even when a data feed appears to be working in another medium (e.g., you might be able to access a REST service in a browser but get an accessed denied response when attempting to access the same feed from a Spring application). This depends on the terms of use set forth by a REST provider.

The first line declares the import statement needed to access the RestTemplate class within a class’s body. First you need to create an instance of the RestTemplate class. Next, you can find a call made to the getForObject method that belongs to the RestTemplate class, which as described in Table [4-1](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_4_Chapter.html#Tab1) is used to perform an HTTP GET operation—just like the one performed by a browser to obtain a REST service’s payload. There are two important aspects related to this last method: its response and its parameters.

The response of calling the getForObject method is assigned to a String object. This means the same output you saw in your browser for this REST service (i.e., the XML structure) is assigned to a String. Even if you’ve never processed XML in Java, you’re likely aware that extracting and manipulating data as a Java String is not an easy task. In other words, there are classes better suited for processing XML data (and with it a REST service’s payload) than a String object. For the moment, just keep this in mind; other recipes in the chapter illustrate how to better extract and manipulate the data obtained from a REST service.

The parameters passed to the getForObject method consist of the actual REST service endpoint. The first parameter corresponds to the URL (i.e., endpoint) declaration. Notice the URL is identical to the one used when you relied on a browser to call it.

When you execute this, the output will be the same as in the browser except that it is now printed in the console.

*Retrieve Data from a Parameterized URL*

The previous section showed how you can call a URI to retrieve data , but what about a URI that requires parameters? You don’t want to hard-code parameters into the URL. With the RestTemplate class, you can use a URL with placeholders, and these placeholders will be replaced with actual values upon execution. Placeholders are defined using { and }, just as with request mapping (see recipes 4-1 and 4-2).

The URI http://localhost:8080/court/member/{memberId} is an example of such a parameterized URI. To be able to call this method, you need to pass in a value for the placeholder. You can do this by using a Map and passing that as the third parameter to the getForObject method of the RestTemplate class.

**public class** Main {  
  
    **public** static void main(String[] args) **throws** Exception {  
        final String uri = "http://localhost:8080/court/member/{memberId}";  
        Map<String, String> params = **new** HashMap<>();  
        params.put("memberId", "1");  
        RestTemplate restTemplate = **new** RestTemplate();  
        String result = restTemplate.getForObject(uri, String.class, params );  
        System.out.println(result);  
    }  
}

This last snippet makes use of the HashMap class —part of the Java collections framework—and creates an instance with the corresponding REST service parameters, which is later passed to the getForObject method of the RestTemplate class. The results obtained by passing either a series of String parameters or a single Map parameter to the various RestTemplate methods are identical.

*Retrieve Data as a Mapped Object*

Instead of returning a String to be used in your application, you can also (re)use your Members and Member classes to map the result. Instead of passing in String.class as the second parameter, pass Members.class, and the response will be mapped onto this class.

**package** com.apress.springrecipes.court;  
  
**import** com.apress.springrecipes.court.domain.Members;  
**import** org.springframework.web.client.RestTemplate;  
  
**public class** Main {  
  
    **public** static void main(String[] args) **throws** Exception {  
        final String uri = "http://localhost:8080/court/members.xml";  
        RestTemplate restTemplate = **new** RestTemplate();  
        Members result = restTemplate.getForObject(uri, Members.class);  
        System.out.println(result);  
    }  
}

The RestTemplate class makes use of the same HttpMessageConverter infrastructure as a controller with @ResponseBody marked methods. As JAXB 2 (as well as Jackson) is automatically detected, mapping to a JAXB-mapped object is quite easy.

4-4. Publish RSS and Atom Feeds

PROBLEM

You want to publish an RSS or Atom feed in a Spring application.

SOLUTION

RSS and Atom feeds have become a popular means by which to publish information. Access to these types of feeds is provided by means of a REST service, which means building a REST service is a prerequisite to publishing RSS and Atom feeds. In addition to relying on Spring’s REST support, it’s also convenient to rely on a third-party library especially designed to deal with the particularities of RSS and Atom feeds. This makes it easier for a REST service to publish this type of XML payload. For this last purpose, you will use Project Rome, an open source library available at <http://rometools.github.io/rome/>.

**Tip**

Even though RSS and Atom feeds are often categorized as news feeds, they have surpassed this initial usage scenario of providing just news. Nowadays, RSS and Atom feeds are used to publish information related to blogs, weather, travel, and many other things in a cross-platform manner (i.e., using XML). Hence, if you require publishing information of any sort that’s to be accessible in a cross-platform manner, doing so as RSS or Atom feeds can be an excellent choice given their wide adoption (e.g., many applications support them, and many developers know their structure).

HOW IT WORKS

The first thing you need to do is determine the information you want to publish as an RSS or Atom news feed. This information can be located in an RDBMS or text file, be accessed through JDBC or ORM, or inclusively be part of a Spring bean or some other type of construct. Describing how to obtain this information would go beyond the scope of this recipe, so we will assume you’ll use whatever means you deem appropriate to access it. Once you’ve pinpointed the information you want to publish, it’s necessary to structure it as an RSS or Atom feed, which is where Project Rome comes into the picture.

In case you’re unfamiliar with an Atom feed’s structure , the following snippet illustrates a fragment of this format:

**<?xml** version="1.0" encoding="utf-8"**?>**   
**<feed** xmlns="http://www.w3.org/2005/Atom"**>**   
    **<title>**Example Feed**</title>**   
    **<link** href="http://example.org/"**/>**   
    **<updated>**2010-08-31T18:30:02Z**</updated>**   
    **<author>**   
        **<name>**John Doe**</name>**   
    **</author>**   
    **<id>**urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6**</id>**   
    **<entry>**   
        **<title>**Atom-Powered Robots Run Amok**</title>**   
        **<link** href="http://example.org/2010/08/31/atom03"**/>**   
        **<id>**urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a**</id>**   
        **<updated>**2010-08-31T18:30:02Z**</updated>**   
        **<summary>**Some text.**</summary>**   
    **</entry>**   
**</feed>**

The following snippet illustrates a fragment of an RSS feed’s structure :

**<?xml** version="1.0" encoding="utf-8"**?>**   
**<rss** version="2.0"**>**   
    **<channel>**   
        **<title>**RSS Example**</title>**   
        **<description>**This is an example of an RSS feed**</description>**   
        **<link>**http://www.example.org/link.htm**</link>**   
        **<lastBuildDate>**Mon, 28 Aug 2006 11:12:55 -0400 **</lastBuildDate>**   
        **<pubDate>**Tue, 31 Aug 2010 09:00:00 -0400**</pubDate>**   
        **<item>**   
            **<title>**Item Example**</title>**   
            **<description>**This is an example of an Item**</description>**   
            **<link>**http://www.example.org/link.htm**</link>**   
            **<guid** isPermaLink="false"**>** 1102345**</guid>**   
            **<pubDate>**Tue, 31 Aug 2010 09:00:00 -0400**</pubDate>**   
        **</item>**   
    **</channel>**   
**</rss>**

As you can observe from these last two snippets, RSS and Atom feeds are just XML payloads that rely on a series of elements to publish information. Though going into the finer details of either an RSS or Atom feed structure would require a book in itself, both formats possess a series of common characteristics ; chief among them are these:

* They have a metadata section to describe the contents of a feed (e.g., the <author> and <title> elements for the Atom format and the <description> and <pubDate> elements for the RSS format).
* They have recurring elements to describe information (e.g., the <entry> element for the Atom feed format and the <item> element for the RSS feed format). In addition, each recurring element has its own set of elements with which to further describe information.
* They have multiple versions. RSS versions include 0.90, 0.91 Netscape, 0.91 Userland, 0.92, 0.93, 0.94, 1.0, and 2.0. Atom versions include 0.3 and 1.0. Project Rome allows you to create a feed’s metadata section, recurring elements, and any of the previously mentioned versions, from the information available in the Java code (e.g., Strings, Maps, or other such constructs).

Now that you’re aware of the structure of RSS and Atom feeds, as well as the role Project Rome plays in this recipe, let’s take a look at a Spring MVC controller charged with presenting a feed to an end user:

*//FINAL*   
**package** com.apress.springrecipes.court.web;  
  
**import** com.apress.springrecipes.court.feeds.TournamentContent;  
**import** org.springframework.stereotype.Controller;  
**import** org.springframework.ui.Model;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
  
**import** java.util.ArrayList;  
**import** java.util.Date;  
**import** java.util.List;  
  
@Controller  
**public class** FeedController {  
    @RequestMapping("/atomfeed")  
    **public** String getAtomFeed(Model model) {  
        List<TournamentContent> tournamentList = **new** ArrayList<>();  
        tournamentList.add(TournamentContent.of("ATP", **new** Date(), "Australian Open", "www.australianopen.com"));  
        tournamentList.add(TournamentContent.of("ATP", **new** Date(), "Roland Garros", "www.rolandgarros.com"));  
        tournamentList.add(TournamentContent.of("ATP", **new** Date(), "Wimbledon", "www.wimbledon.org"));  
        tournamentList.add(TournamentContent.of("ATP", **new** Date(), "US Open", "www.usopen.org"));  
        model.addAttribute("feedContent", tournamentList);  
  
        **return** "atomfeedtemplate";  
    }  
  
    @RequestMapping("/rssfeed")  
    **public** String getRSSFeed(Model model) {  
        List<TournamentContent> tournamentList;  
        tournamentList = **new** ArrayList<TournamentContent>();  
        tournamentList.add(TournamentContent.of("FIFA", **new** Date(), "World Cup", "www.fifa.com/worldcup/"));  
        tournamentList.add(TournamentContent.of("FIFA", **new** Date(), "U-20 World Cup", "www.fifa.com/u20worldcup/"));  
        tournamentList.add(TournamentContent.of("FIFA", **new** Date(), "U-17 World Cup", "www.fifa.com/u17worldcup/"));  
        tournamentList.add(TournamentContent.of("FIFA", **new** Date(), "Confederations Cup", "www.fifa.com/confederationscup/"));  
        model.addAttribute("feedContent", tournamentList);  
  
        **return** "rssfeedtemplate";  
    }  
}

This Spring MVC controller has two handler methods. One is called getAtomFeed(), which is mapped to a URL in the form http://[host\_name]/[app-name]/atomfeed, and the other is called getRSSFeed(), which is mapped to a URL in the form http://[host\_name]/[app-name]/rssfeed.

Each handler method defines a List of TournamentContent objects, where the backing class for a TournamentContent object is a POJO. This List is then assigned to the handler method’s Model object for it to become accessible to the returning view. The returning logical views for each handler methods are atomfeedtemplate and rssfeedtemplate, respectively. These logical views are defined in the following manner inside a Spring configuration class:

**package** com.apress.springrecipes.court.web.config;  
  
**import** com.apress.springrecipes.court.feeds.AtomFeedView;  
**import** com.apress.springrecipes.court.feeds.RSSFeedView;  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.web.servlet.HandlerMapping;  
**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;  
**import** org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping;  
  
@Configuration  
@EnableWebMvc  
@ComponentScan(basePackages = "com.apress.springrecipes.court")  
**public class** CourtRestConfiguration {  
  
    @Bean  
    **public** AtomFeedView atomfeedtemplate() {  
        **return new** AtomFeedView();  
    }  
  
    @Bean  
    **public** RSSFeedView rssfeedtemplate() {  
        **return new** RSSFeedView();  
    }  
...  
}

As you can observe, each logical view is mapped to a class. Each of these classes is charged with implementing the necessary logic to build either an Atom or RSS view. If you recall from Chapter [3](https://learning.oreilly.com/library/view/spring-5-recipes/9781484227909/A314861_4_En_3_Chapter.html), you used an identical approach (i.e., using classes) for implementing PDF and Excel views.

In the case of Atom and RSS views, Spring comes equipped with two classes specially equipped and built on the foundations of Project Rome. These classes are AbstractAtomFeedView and AbstractRssFeedView. Such classes provide the foundations to build an Atom or RSS feed , without dealing in the finer details of each of these formats.

The following code illustrates the AtomFeedView class that implements the AbstractAtomFeedView class and is used to back the atomfeedtemplate logical view:

**package** com.apress.springrecipes.court.feeds;  
  
**import** com.rometools.rome.feed.atom.Content;  
**import** com.rometools.rome.feed.atom.Entry;  
**import** com.rometools.rome.feed.atom.Feed;  
**import** org.springframework.web.servlet.view.feed.AbstractAtomFeedView;  
  
**import** javax.servlet.http.HttpServletRequest;  
**import** javax.servlet.http.HttpServletResponse;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.stream.Collectors;  
  
**public class** AtomFeedView **extends** AbstractAtomFeedView {  
  
    @Override  
    **protected** void buildFeedMetadata(Map model, Feed feed, HttpServletRequest request) {  
        feed.setId("tag:tennis.org");  
        feed.setTitle("Grand Slam Tournaments");  
  
        List<TournamentContent> tournamentList = (List<TournamentContent>) model.get("feedContent");  
  
        feed.setUpdated(tournamentList.stream().map(TournamentContent::getPublicationDate).sorted().findFirst().orElse(**null**));  
  
    }  
  
    @Override  
    **protected** List buildFeedEntries(Map model, HttpServletRequest request, HttpServletResponse response)  
        **throws** Exception {  
        List<TournamentContent> tournamentList = (List<TournamentContent>) model.get("feedContent");  
        **return** tournamentList.stream().map(**this**::toEntry).collect(Collectors.toList());  
    }  
  
    **private** Entry toEntry(TournamentContent tournament) {  
        Entry entry = **new** Entry();  
        String date = String.format("%1$tY-%1$tm-%1$td", tournament.getPublicationDate());  
        entry.setId(String.format("tag:tennis.org,%s:%d", date, tournament.getId()));  
        entry.setTitle(String.format("%s - Posted by %s", tournament.getName(), tournament.getAuthor()));  
        entry.setUpdated(tournament.getPublicationDate());  
  
        Content summary = **new** Content();  
        summary.setValue(String.format("%s - %s", tournament.getName(), tournament.getLink()));  
        entry.setSummary(summary);  
        **return** entry;  
    }  
}

The first thing to notice about this class is that it imports several Project Rome classes from the com.sun.syndication.feed.atom package, in addition to implementing the AbstractAtomFeedView class provided by the Spring Framework. In doing so, the only thing that’s needed next is to provide a feed’s implementation details for two methods inherited from the AbstractAtomFeedView class: buildFeedMetadata and buildFeedEntries.

buildFeedMetadata has three input parameters: a Map object that represents the data used to build the feed (i.e., data assigned inside the handler method, in this case a List of TournamentContent objects), a Feed object based on a Project Rome class that is used to manipulate the feed itself, and an HttpServletRequest object in case it’s necessary to manipulate the HTTP request.

Inside the buildFeedMetadata method, you can observe several calls are made to the Feed object’s setter methods (e.g., setId, setTitle, setUpdated). Two of these calls are made using hard-coded strings, while another is made with a value determined after looping over a feed’s data (i.e., the Map object). All these calls represent the assignment of an Atom feed’s metadata information.

**Note**

Consult Project Rome’s API if you want to assign more values to an Atom feed’s metadata section, as well as specify a particular Atom version. The default version is Atom 1.0.

The buildFeedEntries method also has three input parameters: a Map object that represents the data used to build the feed (i.e., data assigned inside the handler method, in this case a List of TournamentContent objects), an HttpServletRequest object in case it’s necessary to manipulate the HTTP request, and an HttpServletResponse object in case it’s necessary to manipulate the HTTP response. It’s also important to note that the buildFeedEntries method returns a List of objects, which in this case corresponds to a List of Entry objects based on a Project Rome class and containing an Atom feed’s recurring elements.

Inside the buildFeedEntries method, you can observe that the Map object is accessed to obtain the feedContent object assigned inside the handler method. Once this is done, an empty List of Entry objects is created. Next, a loop is performed on the feedContent object, which contains a list of a List of TournamentContent objects, and for each element, an Entry object is created that is assigned to the top-level List of Entry objects. Once the loop is finished, the method returns a filled List of Entry objects.

**Note**

Consult Project Rome’s API if you want to assign more values to an Atom feed’s recurring elements section.

Upon deploying the previous class, in addition to the previously cited Spring MVC controller , accessing a URL in the form http://[host\_name]/[app-name]/atomfeed.atom (or http://[host\_name]/atomfeed.xml) would result in the following response:

**<?xml** version="1.0" encoding="UTF-8"**?>**   
**<feed** xmlns="http://www.w3.org/2005/Atom"**>**   
    **<title>**Grand Slam Tournaments**</title>**   
    **<id>**tag:tennis.org**</id>**   
    **<updated>**2017-04-19T01:32:52Z**</updated>**   
    **<entry>**   
        **<title>**Australian Open - Posted by ATP**</title>**   
        **<id>**tag:tennis.org,2017-04-19:5**</id>**   
        **<updated>**2017-04-19T01:32:52Z**</updated>**   
        **<summary>**Australian Open - www.australianopen.com**</summary>**   
    **</entry>**   
    **<entry>**   
        **<title>**Roland Garros - Posted by ATP**</title>**   
        **<id>**tag:tennis.org,2017-04-19:6**</id>**   
        **<updated>**2017-04-19T01:32:52Z**</updated>**   
        **<summary>**Roland Garros - www.rolandgarros.com**</summary>**   
    **</entry>**   
    **<entry>**   
        **<title>**Wimbledon - Posted by ATP**</title>**   
        **<id>**tag:tennis.org,2017-04-19:7**</id>**   
        **<updated>**2017-04-19T01:32:52Z**</updated>**   
        **<summary>**Wimbledon - www.wimbledon.org**</summary>**   
    **</entry>**   
    **<entry>**   
        **<title>**US Open - Posted by ATP**</title>**   
        **<id>**tag:tennis.org,2017-04-19:8**</id>**   
        **<updated>**2017-04-19T01:32:52Z**</updated>**   
        **<summary>**US Open - www.usopen.org**</summary>**   
    **</entry>**   
**</feed>**

Turning your attention to the remaining handler method— getRSSFeed—from the previous Spring MVC controller charged with building an RSS feed, you’ll see that the process is similar to the one just described for building Atom feeds. The handler methods also creates a List of TournamentContent objects, which is then assigned to the handler method’s Model object for it to become accessible to the returning view. The returning logical view in this case, though, now corresponds to one named rssfeedtemplate. As described earlier, this logical view is mapped to a class named RssFeedView.

The following code illustrates the RssFeedView class , which implements the AbstractRssFeedView class:

**package** com.apress.springrecipes.court.feeds;  
  
**import** com.rometools.rome.feed.rss.Channel;  
**import** com.rometools.rome.feed.rss.Item;  
**import** org.springframework.web.servlet.view.feed.AbstractRssFeedView;  
  
**import** javax.servlet.http.HttpServletRequest;  
**import** javax.servlet.http.HttpServletResponse;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.stream.Collectors;  
  
**public class** RSSFeedView **extends** AbstractRssFeedView {  
  
    @Override  
    **protected** void buildFeedMetadata(Map model, Channel feed, HttpServletRequest request) {  
        feed.setTitle("World Soccer Tournaments");  
        feed.setDescription("FIFA World Soccer Tournament Calendar");  
        feed.setLink("tennis.org");  
  
        List<TournamentContent> tournamentList = (List<TournamentContent>) model.get("feedContent");  
        feed.setLastBuildDate(tournamentList.stream().map( TournamentContent::getPublicationDate).sorted().findFirst().orElse(**null**) );  
    }  
   
  
    @Override  
    **protected** List<Item> buildFeedItems(Map model, HttpServletRequest request, HttpServletResponse response)  
        **throws** Exception {  
        List<TournamentContent> tournamentList = (List<TournamentContent>) model.get("feedContent");  
  
        **return** tournamentList.stream().map(**this**::toItem).collect(Collectors.toList());  
    }  
  
    **private** Item toItem(TournamentContent tournament) {  
        Item item = **new** Item();  
        item.setAuthor(tournament.getAuthor());  
        item.setTitle(String.format("%s - Posted by %s", tournament.getName(), tournament.getAuthor()));  
        item.setPubDate(tournament.getPublicationDate());  
        item.setLink(tournament.getLink());  
        **return** item;  
    }  
}

The first thing to notice about this class is that it imports several Project Rome classes from the com.sun.syndication.feed.rss package, in addition to implementing the AbstractRssFeedView class provided by the Spring Framework. Once it does so, the only thing that’s needed next is to provide a feed’s implementation details for two methods inherited from the AbstractRssFeedView class: buildFeedMetadata and buildFeedItems. The buildFeedMetadata method is similar in nature to the one by the same name used in building an Atom feed. Notice the buildFeedMetadata method manipulates a Channel object based on a Project Rome class, which is used to build RSS feeds, instead of a Feed object, which is used to build Atom feeds. The setter method calls made on the Channel object (e.g., setTitle, setDescription, setLink) represent the assignment of an RSS feed’s metadata information. The buildFeedItems method, which differs in name from its Atom counterpart buildFeedEntries, is so named because an Atom feed’s recurring elements are called *entries* and an RSS feed’s recurring elements are *items*. Naming conventions aside, their logic is similar.

Inside the buildFeedItems method, you can observe that the Map object is accessed to obtain the feedContent object assigned inside the handler method. Once this is done, an empty List of Item objects is created. Next, a loop is performed on the feedContent object, which contains a List of TournamentContent objects, and for each element, an Item object is created that is assigned to the top-level List of Item objects. Once the loop is finished, the method returns a filled List of x§x§ objects.

**Note**

Consult Project Rome’s API if you want to assign more values to an RSS feed’s metadata and recurring element sections, as well as specify a particular RSS version. The default version is RSS 2.0.

When you deploy the previous class, in addition to the previously cited Spring MVC controller , accessing a URL in the form http://[host\_name]/rssfeed.rss (or http://[host\_name]/rssfeed.xml) results in the following response:

**<?xml** version="1.0" encoding="UTF-8"**?>**   
**<rss** version="2.0"**>**   
    **<channel>**   
        **<title>**World Soccer Tournaments**</title>**   
        **<link>**tennis.org**</link>**   
        **<description>**FIFA World Soccer Tournament Calendar**</description>**   
        **<lastBuildDate>**Wed, 19 Apr 2017 01:32:31 GMT**</lastBuildDate>**   
        **<item>**   
            **<title>**World Cup - Posted by FIFA**</title>**   
            **<link>**www.fifa.com/worldcup/**</link>**   
            **<pubDate>**Wed, 19 Apr 2017 01:32:31 GMT**</pubDate>**   
            **314861\_4\_En**FIFA**</author>**   
        **</item>**   
        **<item>**   
            **<title>**U-20 World Cup - Posted by FIFA**</title>**   
            **<link>**www.fifa.com/u20worldcup/**</link>**   
            **<pubDate>**Wed, 19 Apr 2017 01:32:31 GMT**</pubDate>**   
            **314861\_4\_En**FIFA**</author>**   
        **</item>**   
        **<item>**   
            **<title>**U-17 World Cup - Posted by FIFA**</title>**   
            **<link>**www.fifa.com/u17worldcup/**</link>**   
            **<pubDate>**Wed, 19 Apr 2017 01:32:31 GMT**</pubDate>**   
            **314861\_4\_En**FIFA**</author>**   
        **</item>**   
        **<item>**   
            **<title>**Confederations Cup - Posted by FIFA**</title>**   
            **<link>**www.fifa.com/confederationscup/**</link>**   
            **<pubDate>**Wed, 19 Apr 2017 01:32:31 GMT**</pubDate>**   
            **314861\_4\_En**FIFA**</author>**   
        **</item>**   
    **</channel>**   
**</rss>**

Summary

In this chapter, you learned how to develop and access REST services using Spring. REST services are closely tied to Spring MVC, whereby a controller acts to dispatch requests made to REST services, as well as access third-party REST services to use this information for application content.

You learned how REST services leverage annotations used in Spring MVC controllers, which included @RequestMapping to indicate service endpoints, as well as @PathVariable to specify access parameters for filtering a service’s payload. In addition, you learned about Spring’s XML marshallers, such as Jaxb2Marshaller, which allow application objects to be transformed into XML and be output as a REST service’s payload. You also learned about Spring’s RestTemplate class and how it supports the series of HTTP methods that include HEAD, GET, POST, PUT, and DELETE—all of which allow you to access and perform operations on third-party REST services directly from the context of a Spring application.