Build RESTful web services using Spring 3

Yi Ming Huang
Software Engineer
IBM

Dong Fei Wu
Software Engineer
IBM

Skill Level: Intermediate

Date: 27 Jul 2010

In the Java™ world, you can build a RESTful web service in several ways: Some folks use JSR 311(JAX-RS) and its reference implementation Jersey, others use the Restlet framework, and some might even implement from scratch. Spring, the well-known framework for building Java EE applications, now supports REST in its MVC layer. This article introduces the "Spring way" to build RESTful web services. Learn how to use Spring APIs and annotations to build RESTful web services, and see how Spring integrates this new feature seamlessly into its original framework.

Introduction

> Roy Fielding, one of the principal authors of the HTTP specification versions 1.0 and 1.1, introduced REST in his doctoral dissertation in 2000.

With the representational state transfer (REST) style architecture, requests and responses are built around the transfer of representations of resources. Resources are identified by global IDs that typically use a uniform resource identifier (URI). Client applications use HTTP methods (such as GET, POST, PUT, or DELETE) to manipulate the resource or collection of resources. Generally, a GET method is used to get or list the resource or collection of resources, POST is used to create, PUT is used to update or replace, and DELETE is for removing the resource.

For example, `GET http://host/context/employees/12345` gets the *representation* of the employee with the ID 12345. The response representation could be an XML or ATOM that contains the detailed employee information, or it could be a JSP/HTML page that gives a better UI. Which representation you will see depends on the server-side implementation and the MIME type your clients request.

A RESTful web service is a web service implemented using HTTP and the principles of REST. Generally, a RESTful web service will define the base resource URI, the representation/response MIME types it supports, and the operations it supports.

In this article, learn how to use Spring to build the Java server-side RESTful web services. The example will use a browser, curl, and the Firefox plug-in *RESTClient* for the clients that make requests. You can download the source code used in this article.

This article assumes you are familiar with REST basics. Resources has more information about REST.

Spring 3 REST support

Build RESTful web services with Java technology http://www.ibm.com/developerworks/training/kp/j-kp-rest/index.html
Before the Spring framework supported REST, people used several other implementations, such as Restlet, RestEasy, and Jersey, to help build RESTful web services in the Java world. Jersey, the most significant of the group, is the reference implementation of JAX-RS (JSR 311). Resources has more about JSR 311 and Jersey.

Spring, which is a widely used Java EE framework, added support for building RESTful web services in Release 3. Although the REST support is not an implementation of JAX-RS, it has more features than the specification defines. The REST support is integrated seamlessly into Spring's MVC layer and can be easily adopted by applications that build with Spring.

The major features of Spring REST support include:

- Annotations, such as `@RequestMapping` and `@PathVariable`, to support resource identification and URI mappings
- `ContentNegotiatingViewResolver` to support different representations with different MIME/content types
- Seamless integration into the original MVC layer with a similar programming model

Build an example RESTful web service

The example in this section walks through setting up the Spring 3 environment and creating a "Hello world" application that can be integrated into Tomcat. Then we go through a more complicated application to introduce the essentials of Spring 3 REST support, such as multiple MIME type representations support and JAXB support. Code snippets help illustrate the concepts. You can download all of the sample code for this article.

This article assumes you are familiar with the Spring framework and Spring MVC.

Hello World: Using Spring 3 REST support

To set up the development environment to follow along with the example, you need:

- An IDE: Eclipse IDE for JEE (v3.4+)
- Java SE5 or above
- A web container: Apache Tomcat 6.0 (Jetty and others will also work)
- Spring 3 framework (v3.0.3 is the latest as of this writing)
- Other libraries: JAXB 2, JSTL, commons-logging

Create a dynamic web application in Eclipse and set up Tomcat 6 as its runtime. You then need to set up the web.xml file to enable Spring WebApplicationContext. The example distributes the Spring bean configuration into two files: rest-servlet.xml will handle the MVC/REST related configuration, and rest-context.xml handles the service-level configuration (such as data source beans). Listing 1 shows the Spring configuration snippet in web.xml.

Listing 1. Enable Spring WebApplicationContext in web.xml

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
 /WEB-INF/rest-context.xml
</param-value>
</context-param>

<!-- This listener will load other application context file in addition to
            rest-servlet.xml -->
<listener>
<listener-class>
 org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>

<servlet>
<servlet-name>rest</servlet-name>
<servlet-class>
 org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
<servlet-name>rest</servlet-name>
<url-pattern>/service/*</url-pattern>
</servlet-mapping>
```

Set up the Spring MVC-related configuration (Controller, View, View Resolver) in the rest-servlet.xml file. Listing 2 shows the most important snippet.

Listing 2. Setting up Spring MVC configuration in rest-servlet.xml

```
<context:component-scan base-package="dw.spring3.rest.controller" />

<!--To enable @RequestMapping process on type level and method level-->
<bean class="org.springframework.web.servlet.mvc.annotation
  .DefaultAnnotationHandlerMapping" />
<bean class="org.springframework.web.servlet.mvc.annotation
  .AnnotationMethodHandlerAdapter" />
```

```
<!--Use JAXB OXM marshaller to marshall/unmarshall following class-->
<bean id="jaxbMarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
<property name="classesToBeBound">
 <list>
     <value>dw.spring3.rest.bean.Employee</value>
     <value>dw.spring3.rest.bean.EmployeeList</value>
 </list>
</property>
</bean>

<bean id="employees" class=
 "org.springframework.web.servlet.view.xml.MarshallingView">
<constructor-arg ref="jaxbMarshaller" />
</bean>

<bean id="viewResolver" class=
"org.springframework.web.servlet.view.BeanNameViewResolver" />
```

In the code above:

`Component-scan`
: Enables automatic scan for the class that has Spring annotations
In practice, it's used to detect the `@Controller` annotation defined in controller classes.

`DefaultAnnotationHanlderMappings` and **`AnnotationMethodHandlerAdapter`**
: Beans that will make the `@ReqeustMapping` annotation on the class or method to be processed by Spring
This annotation will be discussed in detail in the next section.

`Jaxb2Mashaller`
: Defines the marshaller/unmarshaller that uses JAXB 2 to do the object XML mapping (OXM)

`MashallingView`
: Defines an XML representation `view` that utilizes the Jaxb2Mashaller

`BeanNameViewResolver`
: Defines a view resolver using the bean name that the user specifies
The example will use the `MarshallingView` name "employees."

That completes the Spring-related configuration. The next step is to write a controller that handles the user request. Listing 3 shows the controller class.

Listing 3. EmployeeController in package dw.spring3.rest.controller

```
@Controller
public class EmployeeController {
@RequestMapping(method=RequestMethod.GET, value="/employee/{id}")
public ModelAndView getEmployee(@PathVariable String id) {
 Employee e = employeeDS.get(Long.parseLong(id));
 return new ModelAndView(XML_VIEW_NAME, "object", e);
}
}
```

The `@RequestMapping` annotation is the key to the Spring REST feature. It specifies which HTTP method (`RequestMethod.GET`) and which URI (`/employee/{id}`) should be handled by the annotated method. Note that:

- For the `{id}` placeholder, the value within the {} can be injected to the method parameter using the `@PathVariable` annotation.
- XML_VIEW_NAME equals `employees`, which is the view name defined in rest-servlet.xml.
- `employeeDS` is a simple memory-based data source whose implementation is out of the scope of this article.

Publish the web application to your Tomcat. At this point you can open the browser and enter `http://<host>:<port>/<appcontext>/service/employee/1`. The browser should display an XML view of the employee with ID equal to 1.

Read on to learn about more features of Spring REST support.

Methods

Resources are manipulated using HTTP methods such as GET, POST, PUT, and DELETE. Previously you learned how to use the `GET` method to retrieve employee information. Now we'll go through POST, PUT, and DELETE.

Using the capabilities of `@RequestMapping` annotation, the code for handling different methods is quite similar. Listing 4 shows a code snippet of the `EmployeeController` class.

Listing 4. EmployeeController in dw.spring3.rest.controller

```
@RequestMapping(method=RequestMethod.POST, value="/employee")
public ModelAndView addEmployee(@RequestBody String body) {
 Source source = new StreamSource(new StringReader(body));
 Employee e = (Employee) jaxb2Mashaller.unmarshal(source);
 employeeDS.add(e);
 return new ModelAndView(XML_VIEW_NAME, "object", e);
}

@RequestMapping(method=RequestMethod.PUT, value="/employee/{id}")
public ModelAndView updateEmployee(@RequestBody String body) {
 Source source = new StreamSource(new StringReader(body));
 Employee e = (Employee) jaxb2Mashaller.unmarshal(source);
 employeeDS.update(e);
 return new ModelAndView(XML_VIEW_NAME, "object", e);
}

@RequestMapping(method=RequestMethod.DELETE, value="/employee/{id}")
public ModelAndView removeEmployee(@PathVariable String id) {
 employeeDS.remove(Long.parseLong(id));
 List<Employee> employees = employeeDS.getAll();
 EmployeeList list = new EmployeeList(employees);
 return new ModelAndView(XML_VIEW_NAME, "employees", list);
}
```

In the code above:

- The `RequestMethod.<Method>` value identifies which HTTP method the annotated method should handle.
- With `@RequestBody`, the HTTP request's body content can be injected as a parameter.

In the example, the body is the XML data that's representing the employee. We use JAXB 2 to unmarshall the XML to Java bean and then persist it. A sample request body could be:

```
<employee><id>3</id><name>guest</name></employee>
```

- Other useful annotations that could be injected into method parameters are `@PathVariable`, `@RequestParm`, and so on. The Spring documentation has a complete list of annotations (see Resources).

Collection of resources

Typically you'd also need to manipulate a collection of resources. For example, you might want to get all the employee information instead of information for just an individual. This could be implemented similarly to the previous case; all you need to change is the URI from /employee to /employees. The plural form of employee suits the collection semantics properly. Listing 5 shows the implementation.

Listing 5. getAllEmployees in EmployeeController

```
@RequestMapping(method=RequestMethod.GET, value="/employees")
public ModelAndView getEmployees() {
 List<Employee> employees = employeeDS.getAll();
 EmployeeList list = new EmployeeList(employees);
 return new ModelAndView(XML_VIEW_NAME, "employees", list);
}
```

You need to declare a wrapper class for the Employee collection. The wrapper class is needed for JAXB 2 because it can't marshal the java.util.List class properly. Listing 6 shows the `EmployeeList` class.

Listing 6. EmployeeList class in dw.spring3.rest.bean

```
@XmlRootElement(name="employees")
public class EmployeeList {
 private int count;
 private List<Employee> employees;

 public EmployeeList() {}

 public EmployeeList(List<Employee> employees) {
  this.employees = employees;
  this.count = employees.size();
 }

 public int getCount() {
  return count;
 }
 public void setCount(int count) {
  this.count = count;
 }

 @XmlElement(name="employee")
 public List<Employee> getEmployees() {
  return employees;
 }
 public void setEmployees(List<Employee> employees) {
  this.employees = employees;
 }

}
```

Content negotiation

Another common feature of REST services is that they can produce different representations according to the request. For example, if the client requests HTML/text representation of all the employees, the server should produce a well-formed HTML page to the user. If the client requests application/XML representation of the employees, the server should produce an XML result instead. Other popular representation types are ATOM and PDF.

Spring 3 introduces a new view resolver called `ContentNegotiatingViewResolver`. It can switch view resolvers according to request content type (the `Accept` property in the request header) or URI suffix. The following example uses `ContentNegotiatingViewResolver` to implement multiple representation support.

In the rest-servlet.xml file, comment out the original `viewResolver` that was defined. Use the `ContentNegotiatingViewResolver` instead, as shown in Listing 7.

Listing 7. Define content negotiation

```
<bean class="org.springframework.web.servlet.view
  .ContentNegotiatingViewResolver">
<property name="mediaTypes">
 <map>
 <entry key="xml" value="application/xml"/>
 <entry key="html" value="text/html"/>
 </map>
</property>
<property name="viewResolvers">
 <list>
 <bean class="org.springframework.web.servlet.view
   .BeanNameViewResolver"/>
 <bean class="org.springframework.web.servlet.view
   .UrlBasedViewResolver">
  <property name="viewClass" value=
  "org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
 </bean>
 </list>
</property>
</bean>
```

The definition shows support for handling two request content types: `application/xml` and `text/html`. The code also defines two view resolvers: one `BeanNameViewResolver` to handle application/xml and one `UrlBasedViewResolver` to handle text/html.

In practice, when you enter `http://<host>:<port>/<appcontext>/service/employees` in the browser it requests `text/html` for the employees. Then `UrlBasedViewResolver` will take effect and Spring will pick /WEB-INF/jsp/employees.jsp as its view. When you add the request header `Accept:application/xml` and invoke the request, the `BeanNameViewResolver` will take effect. Per the code in Listing 5, it will use a view named `employees` to represent, which is the JAXB 2 marshaller view that was defined.

The controller code for `getAllEmployees()` won't change. The employees.jsp page will use the model object named `employees` to render itself. Listing 8 shows a code snippet for employees.jsp.

Listing 8. employees.jsp in /WEB-INF/jsp

```
<table border=1>
 <thead><tr>
  <th>ID</th>
  <th>Name</th>
  <th>Email</th>
 </tr></thead>
 <c:forEach var="employee" items="${employees.employees}">
 <tr>
  <td>${employee.id}</td>
  <td>${employee.name}</td>
  <td>${employee.email}</td>
 </tr>
 </c:forEach>
</table>
```

Clients that communicate with REST services

Thus far you've developed a simple RESTful web service that supports CRUD (create, read, update, and delete) operations for employees. This section explores how to communicate with the service. You'll test the REST service using curl.

You could also use the Firefox plug-in called RESTClient to test REST services. It's easy to use and has a good UI. See Resources for download information.

Using curl

Curl is a popular command line tool that can send requests to a server using HTTP and HTTPS protocols. It's a useful tool to communicate with RESTful web services because it can send content by any HTTP method. Curl is a built-in utility on Linux® and the Mac®. For the Windows® platform you can download the tool (see Resources).

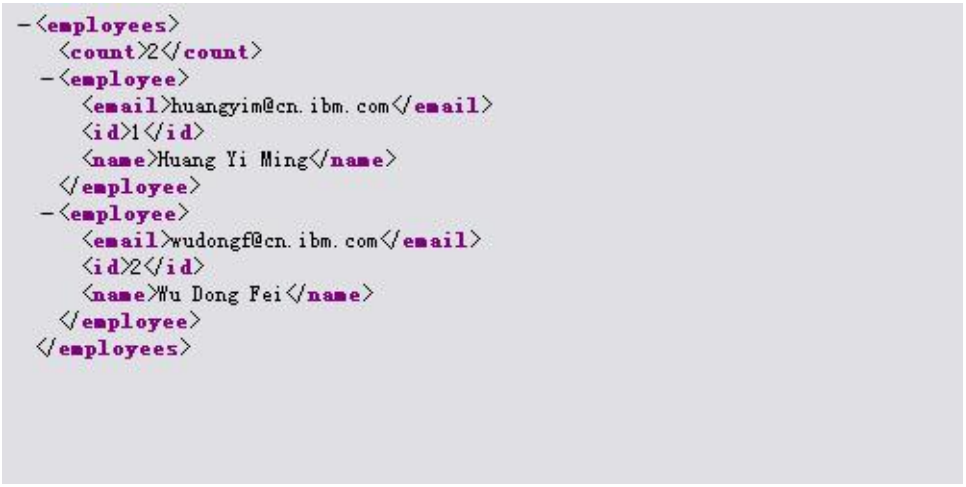
To initialize your first curl command that will get all the employees, enter:

```
curl –HAccept:application/xml

http://localhost:8080/rest/service/employees
```

The response will be in XML and will contain all the employees, as shown in Figure 1.

Figure 1. XML representation of all employees



You can also try the same URL in a browser. In this case, the Accept header specifies text/html, so a table defined in employees.jsp will display. Figure 2 shows an example.

Figure 2. HTML representation of all employees

ID	Name	Email
1	Huang Yi Ming	huangyim@cn.ibm.com
2	Wu Dong Fei	wudongf@cn.ibm.com

To POST a new employee to the server, use the code below. The `addEmployee()` code in Listing 4 will use the request body and unmarshall it to the `Employee` object.

```
curl -X POST -HContent-type:application/xml --data
"<employee><id>3</id><name>guest</name><email>guest@ibm.com</email></employee>"
http://localhost:8080/rest/service/employee
```

A new employee is added. You can use the first example to verify the employee list.

The PUT method is similar to POST.

```
curl -X PUT -HContent-type:application/xml --data
"<employee><id>3</id><name>guest3</name><email>guest3@ibm.com</email></employee>"
http://localhost:8080/rest/service/employee/3
```

The code above updates the data for the employee with the ID of 3 .

Summary

Now that Spring 3 supports REST in its MVC layer, you can use Spring APIs and annotations to build RESTful web services. In this article the examples showed you

how to use some of the new Spring 3 features that will help you easily build Java server-side RESTful web services.

Downloads

Description	Name	Size	Download method
Article source code	src.zip	9KB	HTTP

Information about download methods

Resources

Learn

- Get an introduction and other related links for REST on Wikipedia.
- Learn all about Spring 3.
- For Jersey, get downloads, sample code archives, a Users Guide, and JAX-RS API documents. Jersey is the open source (under dual CDDL+GPL license), production-quality JAX-RS (JSR 311) Reference Implementation for building RESTful web services.
- "Build a RESTful web service using Jersey and Apache Tomcat" (developerWorks, Sep 2009), by the same authors, describes how you can smoothly transfer from servlet-style services to RESTful services by integrating Jersey into Apache Tomcat.
- Read more about the JAXB Reference Implementation Project.

Get products and technologies

- Get Spring 3 downloads, documents, and tutorials.
- Get curl for Windows.
- Get the Firefox RESTClient extension, which is used to visit and test RESTful/ WebDav services.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
- Create your My developerWorks profile today and set up a watchlist on REST, web services, or Spring. Get connected and stay connected with My developerWorks.
- Find other developerWorks members interested in web development.
- Web developers, share your experience and knowledge in the Web development group.
- Share what you know: Join one of our developerWorks groups focused on web topics.
- Roland Barcia talks about Web 2.0 and middleware in his blog.
- Follow developerWorks' members' shared bookmarks on web topics.
- Get answers quickly: Visit the Web 2.0 Apps forum.

About the authors

Yi Ming Huang

Yi Ming Huang, a Software Engineer at the China Development Lab, works on Lotus Mashups. He has experience on portlet and widget-related web development. Yi is interested in REST, OSGi, and Spring technologies.

Dong Fei Wu

Dong Fei Wu is a Software Engineer at the China Development Lab and is working on Lotus Mashups. He has extensive experience on portlet and widget-related web development. Dong is also the Chief Developer for the WebSphere Dashboard Framework.