Build a RESTful Web service using Jersey and Apache Tomcat

Yi Ming Huang (huangyim@cn.ibm.com)
Software Engineer
IBM

Dong Fei Wu (wudongf@cn.ibm.com)
Software Engineer
IBM

Qing Guo (qingguo@cn.ibm.com)
Software Engineer
IBM

24 September 2009

Representational state transfer (REST) was introduced in early 2000 by Roy Fielding's doctoral dissertation. However, in the Java™ community, it was not standardized until the JSR 311(JAX-RS) specification was finalized in 2008. The first release of its reference implementation is even later. In this article, I introduce Jersey, which is the reference implementation of JSR 311, by describing its essential APIs and annotations. I'll also show you how you can smoothly transfer from servlet-style services to RESTful services by integrating Jersey into Apache Tomcat.

RESTful Web service introduction

Representational State Transfer, or REST, was introduced and defined in 2000 by the doctoral dissertation of Roy Fielding, one of the principal authors of the HTTP specification versions 1.0 and 1.1.

The most important concept in REST is *resources*, which are identified by global IDs— typically using URIs. Client applications use HTTP methods (`GET/ POST/ PUT/ DELETE`) to manipulate the resource or collection of resources. A RESTful Web service is a Web service implemented using HTTP and the principles of REST. Typically, a RESTful Web service should define the following aspects:

- The base/root URI for the Web service such as http://host/<appcontext>/resources.
- The MIME type of the response data supported, which are JSON/XML/ATOM and so on.
- The set of operations supported by the service. (for example, `POST, GET, PUT` or `DELETE`).

Table 1 illustrates the resource URI and HTTP methods used in typical RESTful Web services. (Resources also gives you more introduction and design considerations for RESTful Web services.)

Table 1. Example of a RESTful Web service

Method / Resource	Collection of resources, URI like: http://host/<appctx>/resources	Member resources, URI like: http://host/<appctx>/resources/1234
GET	List all the members of the collection resources.	Retrieve a representation of one resource identified as 1234.
PUT	Update (replace) the collection with another one.	Update the member resource identified as 1234.
POST	Create a member resource in the collection where the ID of it is automatically assigned.	Create a sub resource under it.
DELETE	Delete the entire collection of resources.	Delete the member resource identified as 1234.

JSR 311 (JAX-RS) and Jersey

The proposal for JSR 311 or JAX-RS (The Java API for RESTful Web Services) was started in 2007, and the release of version 1.0 was finalized in October 2008. Currently, JSR 311 version 1.1 is in the draft state. The purpose of this JSR is to provide a set of APIs that can simplify the development of REST-style Web services.

Before the JAX-RS specification there were frameworks like Restlet and RestEasy that could help you implement the RESTful Web services, but they were not intuitive. Jersey is the reference implementation for JAX-RS, and it contains three major parts.

- Core Server: By providing annotations and APIs standardized in JSR 311, you can develop a RESTful Web service in a very intuitive way.
- Core Client: The Jersey client API helps you to easily communicate with REST services.
- Integration: Jersey also provides libraries that can easily integrate with Spring, Guice, Apache Abdera, and so on.

In the following sections of the article, I introduce all of these components, but will focus more on the Core Server.

Build a RESTful Web service

I'll begin with a "hello world" application that can be integrated into Tomcat. This application will get you through setting up the environment and will cover the basics of Jersey and JAX-RS.

Then, I'll introduce a more complicated application to go deeper into the essentials and features of JAX-RS, such as multiple MIME type representations support, JAXB support, and so on. I will excerpt code snippets from the sample to introduce the important concepts.

Hello World: The first Jersey Web project

To set up the development environment you need the following artifacts (see Resources for downloads):

- IDE: Eclipse IDE for JEE (v3.4+) or IBM Rational Application Developer 7.5
- Java SE5 or above
- Web container: Apache Tomcat 6.0 (Jetty and others will also work)
- Jersey libraries: Jersey 1.0.3 archive, which includes all the necessary libraries

Setting up the environment for Jersey

First, create a server run time for Tomcat 6.0 on Eclipse. This is the Web container for your RESTful Web application. Then create a dynamic Web application named "Jersey," and specify the target run time to be Tomcat 6.0.

Finally, copy the following libraries from the Jersey archive to the lib directory under WEB-INF:

- Core Server: jersey-core.jar, jersey-server.jar, jsr311-api.jar, asm.jar
- Core Client: (Used for testing) jersey-client.jar
- JAXB support: (Used in the advanced example) jaxb-impl.jar, jaxb-api.jar, activation.jar, stax-api.jar, wstx-asl.jar
- JSON support: (Used in the advanced example) jersey-json.jar

Developing the REST service

Now that you have set up the environment you are ready to develop your first REST service, which simply says "Hello" to the client.

To do this, you need to direct all the REST requests to the Jersey container by defining a servlet dispatcher in the application's web.xml file. (See Listing 1.) Besides declaring the Jersey servlet, it also defines an initialization parameter indicating the Java package that contains the resources.

Listing 1. Define the Jersey servlet dispatcher in the web.xml file

```
<servlet>
  <servlet-name>Jersey REST Service</servlet-name>
<servlet-class>
  com.sun.jersey.spi.container.servlet.ServletContainer
</servlet-class>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>sample.hello.resources</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Now you will write a resource named HelloResource, which accepts the HTTP `GET` and responses with the cliché "Hello Jersey."

Listing 2. HelloResource in package sample.hello.resources

```
@Path("/hello")
public class HelloResource {
 @GET
 @Produces(MediaType.TEXT_PLAIN)
 public String sayHello() {
  return "Hello Jersey";
 }
}
```

There are several points in the code that need highlighting:

- Resource Class: Notice the resource class is a plain old java object (POJO) and is not restricted from implementing any interface. This adds many advantages such as reusability and simplicity.
- Annotations: They are defined in javax.ws.rs.*, which are part of the JAX-RS (JSR 311) specification.
- `@Path`: This defines the resource base URI. Formed with context root and hostname, the resource identifier will be something like http://localhost:8080/Jersey/rest/hello.
- `@GET:` This means that the following method responds to the HTTP `GET` method.
- `@Produces:` Defines the response content MIME type as plain/text.

Testing the Hello app

To test the app, open your browser and enter the URL http://<host>:<port>/<appctx>/rest/hello. You will see the response "Hello Jersey." This is quite simple, with annotations taking care of the request, response, and methods.

The following sections will cover the essential parts of the JAX-RS specification and will be introduced using some code snippets from the Contacts example application. You can find all the code for this more advanced sample in the source code package (see Download).

Resources

Resources are the key parts that compose a RESTful Web service. You manipulate resources using HTTP methods like `GET, POST, PUT`, and `DELETE`. Anything in the application can be a resource: employees, contacts, organizations, everything. In JAX-RX, resources are implemented by a POJO, with an `@Path` annotation to compose its *identifier*. A resource can also have sub resources. In this case, the parent resource is a resource collection while the sub resources are member resources.

In the sample Contacts application, you will manipulate individual contacts and collections of contacts. `ContactsResource` is the collection resource with the URI of /contacts, and `ContactResource` is the member resource with the URI of /contacts/{contactId}. The underlining JavaBean is a simple Contact class with id, name, and address as its member fields. See Listings 3 and 4 for details. You can also download the full source code package at the end of this article (see Download).

Listing 3. ContactsResource

```
@Path("/contacts")
public class ContactsResource {
 @Context
 UriInfo uriInfo;
 @Context
 Request request;

 @GET
 @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public List<Contact> getContacts() {
  List<Contact> contacts = >new ArrayList<Contact>();
  contacts.addAll( ContactStore.getStore().values() );
  return contacts;
 }

@Path("{contact}")
 public ContactResource getContact(
   @PathParam("contact") String contact) {
  return new ContactResource(uriInfo, request, contact);
 }
}
```

There are several interesting things here that you should note.

- `@Context:` Use this annotation to inject the contextual objects such as Request, Response, UriInfo, ServletContext, and so on.
- `@Path("{contact}"):` This is the `@Path` annotation combined with the root path "/contacts" that forms the sub resources' URI.
- `@PathParam("contact"):` This annotation injects the parameters into the path, contact id in this case, to the method parameter. Other available annotations are `@FormParam`, `@QueryParam`, and so on.
- `@Produces:` Multiple MIME types are supported for responses. In this and the preceding case, application/xml will be the default MIME type.

You may also notice that the `GET` methods return custom Java objects instead of a String (plain text), as is shown in the previous Hello World example. The JAX-RS specification requires that the implementation support multiple representation types like InputStream, byte[], JAXB elements, collections of JAXB elements, and so on, as well as the ability to serialize them to XML, JSON, or plain text as responses. I will provide more information on representation techniques, and especially on the JAXB element representation, later in this article.

Listing 4. ContactResource

```
public class ContactResource {
 @Context
 UriInfo uriInfo;
 @Context
 Request request;
 String contact;

 public ContactResource(UriInfo uriInfo, Request request,
   String contact) {
  this.uriInfo = uriInfo;
  this.request = request;
  this.contact = contact;
 }
```

```
 @GET
 @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
 public Contact getContact() {
  Contact cont = ContactStore.getStore().get(contact);
  if(cont==null)
   throw new NotFoundException("No such Contact.");
  return cont;
 }
}
```

The code for ContactResource is straightforward. And, note the following items:

- Representation Type Contact: Contact is a simple JavaBean annotated by `@XmlRootElement`, which makes it possible to be represented as XML or JSON.
- ContactStore: It's a HashMap-based in-memory data store whose implementation is not important for this article.

Method

HTTP methods are mapped to CRUD (create, read, update and delete) actions for a resource. Although you can make slight modifications such as making the `PUT` method to be create or update, the basic patterns are listed as follows.

- HTTP `GET`: Get/List/Retrieve an individual resource or a collection of resources.
- HTTP `POST`: Create a new resource or resources.
- HTTP `PUT`: Update an existing resource or collection of resources.
- HTTP `DELETE`: Delete a resource or collection of resources.

Because I have already introduced the `GET` method, I will start my descriptions with `POST`. I will continue using the Contact example as I explain these other methods.

POST

Usually a new contact is created by filling in a form. That is, an HTML form will be POSTed to the server, and the server creates and persists the newly created contact. Listing 5 demonstrates the server-side logic for this.

Listing 5. Accept the form submission (POST) and create a new contact

```
@POST
@Produces(MediaType.TEXT_HTML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public void newContact(
  @FormParam("id") String id,
  @FormParam("name") String name,
  @Context HttpServletResponse servletResponse
) throws IOException {
 Contact c = new Contact(id,name,new ArrayList<Address>());
 ContactStore.getStore().put(id, c);

 URI uri = uriInfo.getAbsolutePathBuilder().path(id).build();
 Response.created(uri).build();

 servletResponse.sendRedirect("../pages/new_contact.html");
}
```

Note that the following parts make this example work.

- `@Consumes:` Declares that the method consumes an HTML FORM.
- `@FormParam:` Injects the form input identified by the HTML name attribute to this method.
- `@Response.created(uri).build():` Builds a new URI for the newly created contact as `/contacts/{id}` and set the response code as `201/created`. You can access the new contact using http://localhost:8080/Jersey/rest/contacts/<id>.

PUT

I use the PUT method to update an existing resource. However, this can also be implemented as an update or by creating a resource as shown in the code snippet in Listing 6.

Listing 6. Accept PUT request and create or update a contact

```
@PUT
@Consumes(MediaType.APPLICATION_XML)
public Response putContact(JAXBElement<Contact> jaxbContact) {
 Contact c = jaxbContact.getValue();
 return putAndGetResponse(c);
}

private Response putAndGetResponse(Contact c) {
 Response res;
 if(ContactStore.getStore().containsKey(c.getId())) {
  res = Response.noContent().build();
 } else {
  res = Response.created(uriInfo.getAbsolutePath()).build();
 }
 ContactStore.getStore().put(c.getId(), c);
 return res;
}
```

I cover a number of different concepts in this example, highlighting the following concepts.

- Consume XML: The `putContact()` method accepts the request content type of APPLICATION/XML, while this input XML will bind to the Contact object using JAXB. You will find the client code in next section.
- Empty response with different status code: The response of the `PUT` request will not have any content, but will have a different status code. If the contact exists in the data store, I update this contact and return `204/no content`. If there is a new contact, I create it and return `201/created`.

DELETE

It's quite simple to implement a `DELETE` method. Take a look at Listing 7 for an example.

Listing 7. Delete a contact identified by its ID

```
@DELETE
public void deleteContact() {
 Contact c = ContactStore.getStore().remove(contact);
 if(c==null)
  throw new NotFoundException("No such Contact.");
}
```

Representation

In the previous sections, I illustrated several representation types. Now I'll briefly go through them and give you a close look at the JAXB representation. Other supported representation types are byte[], InputStream, File, and so on.

- String: Plain text.
- Response: A generic HTTP response that can contain your custom content with a different response code.
- Void: An empty response with a status code of 204/no content.
- Resource Class: Delegate the process to this resource class.
- POJO: JavaBeans that are annotated with `@XmlRootElement`, which makes it a JAXB bean, and which you can bind to XML.
- Collection of POJOs: A collection of JAXB beans.

JAX-RS supports the use of JAXB (Java API for XML Binding) to bind a JavaBean to XML or JSON and vise versa. The JavaBean must be annotated with `@XmlRootElement`. Listing 8 takes a Contact bean as an example. The fields without an explicit `@XmlElement` annotation will have the XML element named the same as themselves. Listing 9 displays the serialized XML and JSON representation for one Contact bean. Representation for a collection of contacts is much the same and has <Contacts> as the wrapper element by default.

Listing 8. Contact bean

```
@XmlRootElement
public class Contact {
 private String id;
 private String name;
 private List<Address> addresses;

 public Contact() {}

 public Contact(String id, String name, List<Address> addresses) {
  this.id = id;
  this.name = name;
  this.addresses = addresses;
 }

 @XmlElement(name="address")
 public List<Address> getAddresses() {
  return addresses;
 }

 public void setAddresses(List<Address> addresses) {
  this.addresses = addresses;
 }
 // Omit other getters and setters
}
```

Listing 9. The representation for one Contact

```
XML representation:
<contact>
  <address>
    <city>Shanghai</city>
    <street>Long Hua Street</street>
  </address>
  <address>
    <city>Shanghai</city>
    <street>Dong Quan Street</street>
  </address>
  <id>huangyim</id>
    <name>Huang Yi Ming</name>
</contact>


JSON representation:
{"contact":[{"address":[{"city":"Shanghai","street":"Long
            Hua Street"},{"city":"Shanghai","street":"Dong Quan
            Street"}],"id":"huangyim","name":"Huang Yi Ming"}]}
```

For more advanced topics using JAXB see the project home in Resources.

Clients that communicate with REST services

In the example so far, I have developed a RESTful Web service that supports CRUD contacts. Now I'll explain how to communicate with this REST service using curl and Jersey client APIs. In doing so, I'll test the server-side code and give you more information about the client-side technologies.

Use curl to communicate with the REST service

Curl is a popular command-line tool that can send requests to a server using protocols like HTTP and HTTPS. It is a good tool to communicate with RESTful Web services because it can send content by any HTTP method. Curl is already distributed with Linux® and Mac, and there is a utility that you can install for the Windows® platform (see Resources).

Now, let's initialize the first curl command that gets all the contacts. You can refer to Listing 3 for the server-side code.

```
curl http://localhost:8080/Jersey/rest/contacts
```

The response will be in XML and will contain all the contacts.

Notice the `getContacts()` method also produces an application/json MIME-type response. You can request content in this type also.

```
curl –HAccept:application/json http://localhost:8080/Jersey/rest/contacts
```

The response will be a JSON string that contains all the contacts.

Now I will `PUT` a new contact. Notice that the `putContact()` method in Listing 6 accepts XML and uses JAXB to bind the XML to the Contact object.

```
curl -X PUT -HContent-type:application/xml --data "<contact><id>foo</id>
                <name>bar</name></contact>" http://localhost:8080/Jersey/rest/contacts/foo
```

A new contact identified by "foo" is added to the contacts store. You can use URI /contacts or /contacts/foo to verify the contacts collection or individual contact.

Using the Jersey Client to communicate with the REST service

Jersey also provides a client library that helps you to communicate with the server as well as unit test the RESTful services. The library is a generic implementation that can cooperate with any HTTP/HTTPS-based Web service.

The core class for the client is the `WebResource` class. You use it to build a request URL based on the root URI, and then send requests and get responses. Listing 10 shows how to create a `WebResource` instance. Notice that `WebResource` is a heavy object, so you create it once.

Listing 10. Create the WebResource instance

```
Client c = Client.create();
WebResource r=c.resource("http://localhost:8080/Jersey/rest/contacts");
```

The first Jersey client example is to send a `GET` request to fetch all the contacts and print the response status code and response content. See Listing 11.

Listing 11. GET all contacts and print the response

```
ClientResponse response = r.get(ClientResponse.class);
System.out.println( response.getStatus() );
System.out.println( response.getHeaders().get("Content-Type") );
String entity = response.getEntity(String.class);
System.out.println(entity);
```

Listing 12 shows a second example that creates a new contact identified by "foo".

Listing 12. Create one contact

```
Address[] addrs = {
 new Address("Shanghai", "Ke Yuan Street")
};
Contact c = new Contact("foo", "Foo Bar", Arrays.asList(addrs));

ClientResponse response = r
 .path(c.getId())
 .accept(MediaType.APPLICATION_XML)
 .put(ClientResponse.class, c);
System.out.println(response.getStatus());
```

Notice the APIs for the `WebResource` instance. It builds the URI, sets the request headers, and invokes the request in one line of code. The content (Contact object) is bound to XML automatically.

Listing 13 shows the last example that retrieves the contact identified by "foo" (which I created in the last example) and then deletes it.

List 13. Retrieve "foo" contact and delete it

```
GenericType<JAXBElement<Contact>> generic = new GenericType<JAXBElement<Contact>>() {};
JAXBElement<Contact> jaxbContact = r
 .path("foo")
 .type(MediaType.APPLICATION_XML)
 .get(generic);
Contact contact = jaxbContact.getValue();
System.out.println(contact.getId() + ": " + contact.getName());

ClientResponse response = r.path("foo").delete(ClientResponse.class);
System.out.println(response.getStatus());
```

Notice here that when you want to get a response that is a JAXB bean, you need to use the generic type feature introduced in Java 2 Platform, Standard Edition (J2SE).

Play with these examples using the Jersey client. You can find more sample code in the source package too (see Download). Also refer to the Jersey Web site for more information (see Resources).

Conclusion

Jersey can be integrated with other frameworks or utility libraries using the Jersey integration libraries. Currently, Jersey can integrate with Spring, Guice, and can support ATOM representation with apache-adbera integration. You'll find APIs and guides on getting started on Jersey project home.

Downloads

Description	Name	Size
Source code	Jersey.Sample.Contact.Src.zip	10KB

Resources

- Find introduction and other related links for REST on Wikipedia.
- Download Eclipse from the project Web site
- Check out the trial for Rational Application Developer.
- Get Java SE 5.0 from the company Web site.
- Download Apache Tomcat 6.x from the project Web site.
- You can find downloads, sample code archives, users' guides, and JAX-RS API documents on the Jersey Project Home.
- Read this paper on the Jersey client APIs to get more information.
- You can get more information on JAX-RS (JSR 311) on the Java Community Process Web site.
- Get curl for Windows.
- The JAXB Reference Implementation Project gives more information on JAXB.

About the authors

Yi Ming Huang

Yi Ming Huang is a software engineer working on Lotus ActiveInsight in the China Development Lab. He has experience on Portlet/Widget-related Web development and is interested in REST, OSGi, and Spring technologies.

Dong Fei Wu

Dong Fei Wu is a staff software engineer with the IBM WebSphere Dashboard Framework development team in the IBM China Software Development Lab. He designs and develops basic builders for WebSphere Dashboard Framework. You can contact him at wudongf@cn.ibm.com.

Qing Guo

Qing Guo is the development lead of the IBM WebSphere Dashboard Framework. He is experienced in J2EE-related areas.