Design and implement RESTful web services with Rational Software Architect

Model-driven development of RESTful web services

Sandeep Katoch

Introduction

REST (Representational State Transfer) is style of architecture that enables development of loosely coupled systems by using the HTTP protocol. Traditionally, the design of REST-based services (or RESTful web services) has been described in terms of the URIs, resources, HTTP methods supported by each resource, and their representations. Such descriptions are usually published as documentation to enable implementation of the service, as well as to enable the clients of the service to recognize and use the web service. Due to lack of any formal notation, such descriptions invariably exist in design documents as text and tables, which brings us to a few challenges in this approach:

1. How do you design your RESTful web service?
2. How do you implement this design?
3. How do you publish your RESTful services to make them available to users?
4. How do you evolve this design and implementation?

IBM® Rational® Software Architect Version 8.0.3 addresses these challenges within the paradigm of model-driven development (MDD), with the support for modeling and implementation of RESTful web services. The modeling support enables you to create UML models for your web service to describe your web service. The same UML model also serves as the source for generating documentation for those who use your web service. On the implementation side, you generate Java code to generate your web service, using the Java API for RESTful Web Services, or JAX-RS.
With this in mind, let's explore the REST modeling features of Rational Software Architect. By the end of this article, you will be able to design and implement a RESTful Bookmark service according to the UML model shown in Figure 1.

Figure 1. RESTful Bookmark web service model

Designing the web service

The design of the RESTful web service begins with a UML model. Rational Software Architect includes two template models to base your design upon (shown in Figure 2).
Choose the JAX-RS Service Model template if your final implementation of the RESTful service will be based upon JAX-RS. Or, if you are not concerned with the implementation yet or do not want to use JAX-RS, you can select the REST Service Model template. These model templates come preset with the various palettes and libraries that you will need to model the service.

After the model is created, you will normally see the main diagram of the model. On this diagram, you can see the REST palette on the left side, which you use to create REST elements, and a palette of the model elements to choose from on the right side.
You use this palette to create various elements that form a RESTful web service.

Begin by creating an application.

1. Click the **Application Class** palette entry, and then click on an empty space in the diagram.
2. Name this class **BookmarkApplication**. This application class represents the root of the RESTful web service.
3. Next, you can add the RESTful resources that this application should expose.

Each resource is exposed at an URI and it can support any of the HTTP methods. For example, for Bookmark Web Service you can expose your resources in the ways that Table 1 shows.

### Table 1. Resources and URIs

<table>
<thead>
<tr>
<th>Resource</th>
<th>URI</th>
<th>HTTP methods supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>/users</td>
<td>GET, POST, getUser, getListofUsers</td>
</tr>
<tr>
<td>User</td>
<td>/users/{username}</td>
<td>GET, deleteUser, getUser</td>
</tr>
<tr>
<td>Bookmarks</td>
<td>/users/{username}/bookmarks</td>
<td>GET, getListofBookmarks</td>
</tr>
</tbody>
</table>
From this table, you have the Users resource accessible through the `/users` URI at the root of your application.

4. To create resources, you can click on the **Resource Class** in the palette and then click on an empty space in the diagram, and name this class, Users.

5. REST URIs are modeled as *paths*. Hence, to model the `/user` URI, click the Path Dependency entry in the palette, and then click and drag from the BookmarkApplication class on the diagram to the Users class, and name this path `/users`.

Next you can add the REST methods supported by this resource.

For the Users resource, the software support HTTP GET and POST methods, so you can click the **GET operation** and then click on the **Users** class on the diagram, and name the operation `getListofUsers`.

Similarly, repeat these actions for the POST operation, and name it `createUser`.

**Note:**
From the REST perspective, these are GET and POST HTTP methods, but here you have given them more descriptive names.

Your diagram should now look something like Figure 4.

**Figure 4. BookmarkApplication and Users diagram**

6. You can now add the rest of the resources, paths, and operations.

Your final diagram will look something like Figure 5.
For the `getUser` GET operation on the User resource, notice that the URI is `/users/{username}`. Here, `{username}` is a parameter for this resource. To specify its use in the `getUser` operation, click the **Param parameter** in the palette, and then click the `getUser` operation on the main diagram. This will create a parameter for this operation.

**Note:**
After creating the parameter, it may not be visible on the diagram. This is because, by default, parameters are filtered out on diagrams. To make it visible, right-click on the **User** class in the diagram, and select **Filter > Show Signature**.

You can specify details of this parameter in the REST tab in the Properties view (Figure 6). For this parameter, select **PathParam**, because it is extracted from the path, and select **username** as the name. You can also specify that the parameter is a Header parameter or Query Parameter and so on, based upon your design. Additionally, you can specify a default value.

**Figure 6. REST Parameter tab**
For consistency, we recommend that you keep the UML name for this parameter the same as the REST name (or similar).

7. To do this, you can switch to the General tab and change the name to `username`.
8. Also in the General tab, you can specify the UML type of the parameter as `String`.

The difference in the REST type (PathParam) and UML type (String) is that the REST type is the type on the REST interface for the user of the REST service, whereas the UML type is the type of parameter that the implementation of the REST service will use.

For each REST method, you also need to specify the data format that it supports for input and output.

9. Select the `getUser` method, and switch to the REST tab in the Properties view.
10. Click inside of the Produces (or Consumes) list to specify the format.

There is a set of predefined values that you can choose from (see Figure 7), or you enter a type of your choice. For both Produces and Consumes, you can specify multiple types.

11. Select `application/xml` for the Produces section.

**Figure 7. Properties that the REST method produces and uses**

Further down in this tab you can specify the HTTP return codes that this method can return.

12. Click inside of the Return codes list to select a return code.
13. Select 200 OK as one return code and 404 Not Found as another.
The standard HTTP return codes are available in the drop-down menu, or you can also type in your custom code. For each return code, you can specify an example of the content that can be returned and add a description in the Content and ‘Description text boxes, respectively.

**Figure 8. REST method return code properties**

---

**Modeling RESTful interactions in sequence diagrams**

You can model the typical interactions with the clients of your RESTful web service by using the sequence diagrams. The calls between the client and your service are essentially HTTP calls; therefore, an HTTP interaction element is provided within the HTTTPReference library (see Figure 9).

**Figure 9. HTTP interaction elements**

---

The HTTTPReference is already imported into your models if you used the REST or JAX-RS template model to create the model.

The HTTP Interaction element is used to represent both the client and the RESTful web service.

1. To use this on a sequence diagram, click and drag the HTTP Interaction element from the Project Explorer onto a sequence diagram, and give it an appropriate name, for example **Client** for client, **Bookmark Service** for the web service.
2. You can now draw messages between the client and the web service.
3. As you draw each message, you can select a type of operation from the list of HTTP Requests and Responses for that message from the drop-down menu (Figure 10). Typically, for messages from the client to the web service, you would choose a GET, POST, DELETE request, for example.
4. And for the return, select a return code, such as 200 OK.

5. You can also detail each request or response in terms of the URI, headers, and content by using the HTTP Properties tab for a message (Figure 12).
The sequence diagram support is very useful for the clients of your web service to understand the workflow supported.

6. For the implementation side, you can further details the sequence diagram with calls to actual resource classes by simply dragging your resource classes onto the sequence diagrams and drawing messages to the diagram.

Figure 13. Sequence diagram for implementation

Generating documentation using BIRT reports

Business Intelligence and Reporting Tools (BIRT) reports are an easy way for you to share or publish your RESTful web service documentation. Rational Software Architect includes a REST report template to use with your REST Models. You can also edit and enhance the report according to your documentation needs.
1. To create a REST report, select File > New > Other > REST Modeling Reports > REST Report.

Figure 14. REST report

Implementing the web service using JAX-RS

When the model is ready, you can use the UML-to-Java transformation with the JAX-RS extension to generate the implementation for the service. However, before doing that, you need to add a few of the JAX-RS specific details to your model.

The first detail is sub-resource locator operations. In the REST model, you drew the Path dependency to provide navigation from one resource to another.

1. In the JAX-RS implementation, you need to create a sub-resource locator operation in the source resource. You can create a SubResourceLocator method from the palette.

Figure 15. JAX-RS palette
2. For the locator operation to be completely defined, you need to set its return type as the target resource and also set its locator stereotype property (SubResourceLocator) to the path value (<Path/bookmarks> in this example), as Figure 16 shows.

**Figure 16. Sub-Resource Locator properties**

3. Also, you need to add JAX-RS specific types to your operations.

For example, you might want to use the `javax.ws.rs.core.Response` object as a return type. Such types are available in the JAX-RS reference library (see Figure 17).

**Figure 17. JAX-RS reference library**

4. Now you can create a UML-to-Java transformation configuration to generate the code for the service. The source of the configuration should be the model that you created, and the target should be a dynamic web project, which is enabled with JAX-RS libraries.

5. You will also need to enable the **JAX-RS UML-to-Java5** extension (Figure 18) on the extensions tab of the transformation configuration.
This extension takes care of generating the JAX-RS annotations on the Java classes and the required web.xml file updates for the service.

6. Running this transformation will generate the required Java classes for your implementation, and there you can add the required implementation logic to your classes.

You can now package and deploy this application on an IBM® WebSphere® Application Server with Feature Pack for Web 2.0 and Mobile, which supports JAX-RS.

7. If you have the WebSphere Application Server Test Environment installed with Rational Software Architect, you can right-click on your dynamic web project and select **Debug As > Debug on Server**, and select the test server.

After that, you can test and debug your RESTful web service within the Rational Software Architect environment.

**Note:**
You can use any JAX-RS implementation to deploy your implementations (for example Apache Wink, for example). It is not necessary to use WebSphere Application Server.

The UML-to-Java transformation with the JAX-RS extension provides complete support for round-trip engineering. So you can model and generate code, and then later refine the model and update the code, or even update the code and use UML-to-Java transformation in Reconcile mode to update the model from the code.

If you already have an existing dynamic web project based upon JAX-RS, you use the Java-to-UML transformation with the JAX-RS extension to generate a UML model from your project. This can serve as good starting point for you to understand your existing JAX-RS implementations and refine them further.
Summary

This article covered the support for JAX-RS only briefly. A future article will explore JAX-RS specifics in detail.

You have learned how to model and implement RESTful web services using the model-driven development support in Rational Software Architect. A model-based approach makes design, implementation, and evolution of your RESTful web services easier.

You can download the Bookmark service model to explore further (see the Downloadable resources section). Also, you'll find more useful information in the Related topics section of this article.
# Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bookmark service UML model</td>
<td>Bookmarks_Service_Model.zip</td>
<td>6KB</td>
</tr>
</tbody>
</table>
Related topics

- Be sure to check these resources related to this article:
  - To learn more about Rational Software Architect, explore the product overview.
  - Get the WebSphere Application Server Feature Pack for Web 2.0 and Mobile
  - You might find this blog helpful, too: An example of modeling REST web services
  - Evaluate IBM software.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)