RESTful Web Services

Restlet Framework version 2.3

From this tutorial you will learn how to:

• implement a simple RESTful Web Service using the Restlet Framework,
• use some of the Linux commands and other software useful for debugging RESTfull Web Services (e.g., lwp-request, cURL, RESTClient).

During the tutorial, a teacher will be present to help with any practical issues that may arise. This tutorial is voluntary and your work will not be graded.

1. Environment configuration

During this tutorial it is recommended to program in the Eclipse IDE. If you prefer, you can use any tools desired, however that can reduce the chances that the teacher is familiar with that tools and is able to help you if problems occur.

First, create a new Java project in the Eclipse IDE.

Second, create a new directory inside of the project and call it lib.

Next, download the Restlet Framework from http://restlet.com/downloads/current/ (this tutorial was prepared for version 2.3.5 Java SE), and extract the content into the lib directory.

Add org.restlet.jar to the Build Path of your project.

2. Restlet’s Hallo world

In the first part of this tutorial we will work with three Java classes:

1. DemoServer.java – the main class of the server application,
2. DemoServerResource.java – which implements the server logic (handling client’s requests),
3. DemoClient.java – the main class of the client application.

For convenience we will put them all in a single package called demo.

2.1. A server main program

Let’s start with implementation of the DemoServer class. First, create a class with a public static void main method (in Eclipse IDE you may simply check the checkbox in the New Java Class wizard to automate the process).
Inside of the main method create an instance of a `org.restlet.Server` and instantiate it using a following constructor: `Server(Protocol protocol, int port, Class<? extends ServerResource> nextClass)`. It allows you to specify:

- which protocol shall be used – choose `Protocol.HTTP`,
- which port the server should run on – choose for example 8111,
- what class will implement the logic of the server – choose `DemoServerResource.java`, which we will implement in the next step.

Finally, start the server using `demoServer.start()`.

The complete implementation of the server main program should look similarly to the one below:

```java
package demo;

import org.restlet.Server;
import org.restlet.data.Protocol;

public class DemoServer {
    public static void main(String[] args) throws Exception {
        Server demoServer = new Server(Protocol.HTTP, 8111,
                                        DemoServerResource.class);
        demoServer.start();
    }
}
```

### 2.2. A server logic

Now, when we have the main class running the server ready, it’s time to implement the logic of the server.

First, create a new class `DemoServerResource` that extends `org.restlet.resource.ServerResource` base class (in Eclipse IDE you can specify a superclass in a New Java Class wizard).

In a second step, create a new `public` method `Read` that returns a `String`, e.g. a static text "read!\n". Annotate it with `@Get` to specify the type of requests that the method should handle.

The initial implementation of the `DemoServerResource` class should look similar to:
```java
package demo;
import org.restlet.resource.Get;
import org.restlet.resource.ServerResource;

public class DemoServerResource extends ServerResource {
    @Get
    public String Read() {
        return "read!\n";
    }
}
```

If you launch your DemoServer application in the current state you should already be able to see that it’s working. There are several ways to send a GET request to your server:

- You can go to the localhost:8111 in an internet browser what will send a GET request to the server (note that not all browsers display the response properly).
- You can use a command GET in a terminal: `GET http://localhost:8111` (which is a short version of `lwp-request -m GET http://localhost:8111`).
- You can use an extension to your internet browser, e.g. RESTClient for Mozilla Firefox (https://addons.mozilla.org/firefox/addon/restclient/).

- You can also use a standalone application with GUI, e.g. RESTClient (http://code.fosshub.com/WizToolsorg-RESTClient/downloads) to send a GET request to a specific address.
2.3. A client main program
We will now implement a client application that will allow us to send requests to the server and receive responses.

First, let’s create a DemoClient class with a public static void main method.

Inside of the main method create an instance of org.restlet.resource.ClientResource and call it demoClientResource. Instantiate it using the ClientResource(String uri) constructor, that allows to specify the address of the server (http://localhost:8111).

Now you can use demoClientResource.get() method to send a GET request to the server and get a response (as a Representation object). Representation.write(OutputStream outputStream) method allows you to write the response into an arbitrary outputStream. We will use these two methods to write the response to the standard system output (System.out).

The initial implementation of the client application should look similar to the presented below:
package demo;

import org.restlet.resource.ClientResource;

public class DemoClient {

    public static void main(String[] args) throws Exception {

        demoClientResource.get().write(System.out);
    }
}

Output from the client’s console:

Starting the internal HTTP client
read!

3. Using HTTP methods for RESTful services

Let’s now extend the implementations of the DemoServerResource and DemoClient classes to cover all four CRUD (Create, Read, Update and Delete) operations using HTTP methods (POST, GET, PUT and DELETE respectively).

3.1. A server logic

In the DemoServerResource class add three methods:

- String Update()
- String Create()
- String Delete()

that return different messages to be able to distinguish them, and annotate them with proper HTTP methods.

Now, the DemoServerResource should look like below:

package demo;

import org.restlet.resource.Get;
import org.restlet.resource.Post;
import org.restlet.resource.Put;
import org.restlet.resource.Delete;
import org.restlet.resource.ServerResource;

public class DemoServerResource extends ServerResource {

    @Get
    public String Read() {
        return "read!\n";
    }
}
@Put
public String Update() {
    return "update!\n";
}

@Post
public String Create() {
    return "create!\n";
}

@Delete
public String Delete() {
    return "delete!\n";
}

You can now run the extended version of the server and test it using another program: cURL. In a terminal run:

- curl -X PUT http://localhost:8111
- curl -X POST http://localhost:8111
- curl -X DELETE http://localhost:8111

3.2. A client main program
Now, extend the client with calls of `post()`, `put()` and `delete()` methods. After this changes `DemoClient` should look similar to:

```java
package demo;

import org.restlet.resource.ClientResource;

public class DemoClient {

    public static void main(String[] args) throws Exception {
        ClientResource demoClientResource = new
            ClientResource("http://localhost:8111/");

        demoClientResource.get().write(System.out);
        demoClientResource.post(null).write(System.out);
        demoClientResource.put(null).write(System.out);
        demoClientResource.delete().write(System.out);
    }

}
```

If you run the client application, the output from the client’s console should look like below:

```
Starting the internal HTTP client
read!
create!
update!
```
4. Passing arguments in HTTP requests
Till now, we were sending “empty” HTTP requests (without body) and the server was reacting differently just based on the type of the request (GET, POST, etc.). But what if we would like to pass some arguments in HTTP methods, e.g. an identifier of a resource we would like to read (GET) or the value of the resource we would like to store (POST)?

In Restlet there are two main ways of passing the arguments inside of the HTTP requests:

• in the request’s body (for POST and PUT requests),
• using request’s query value (for all types of requests).

Let’s now reimplement DemoServerResource and DemoClient classes to enable passing arguments in the HTTP requests.

4.1. A server logic
In DemoServerResource we need to:

• introduce an object (e.g., String) into the parameters list of methods handling POST and PUT requests,
• read the query parameter’s value using String Resource.getQueryValue(String name) method for GET and DELETE requests,
• update the returned values by including new parameters (to be able to see that they were passed correctly).

```java
package demo;

import org.restlet.resource.Get;
import org.restlet.resource.Post;
import org.restlet.resource.Put;
import org.restlet.resource.Delete;
import org.restlet.resource.ServerResource;

public class DemoServerResource extends ServerResource {

    @Get
    public String Read() {
        String item = getQueryValue("queryItem");
        return "read " + item + "!\n";
    }

    @Put
    public String Update(String body) {
        return "update " + body + "!\n";
    }
}
```
@Post
public String Create(String body) {
    return "create " + body +"!
";
}

@Delete
public String Delete() {
    String item = getQueryValue("queryItem");
    return "delete " + item + "!
";
}
}

You can test the new version of the server using for example cURL. In a terminal run:
- curl -X DELETE http://localhost:8111?queryItem=xyz
- curl -X POST --data "abc" http://localhost:8111

You can also try testing it using the RESTClient plugin for Mozilla Firefox.

4.2. A client main program
In DemoClient we need to:
- pass an object (e.g., String) when calling
  ClientResource.post(Object entity) and
  ClientResource.put(Object entity) methods,
- use Resource.setQueryValue(String name, String value) method to specify the query parameter’s name and value.

```java
package demo;

import org.restlet.resource.ClientResource;

public class DemoClient {

    public static void main(String[] args) throws Exception {

        demoClientResource.setQueryValue("queryItem", "xyz");
        demoClientResource.get().write(System.out);
        demoClientResource.post("abc").write(System.out);
        demoClientResource.put("abc").write(System.out);
        demoClientResource.delete().write(System.out);
    }
}
```

The output of the client’s console should look similar to the one below.

```
Starting the internal HTTP client
read xyz!
create abc!
update abc!
```
5. **Using routing to serve multiple resources**

Up to this point, we were considering a server implementing a single method for each request type (GET, POST, etc.). But what, if we would like to build a more complex server that provides multiple resources and for each of them has separate CRUD methods?

5.1. **A server main program**

In the Restlet Framework for Java SE we can use `org.restlet.Component` class to handle multiple resources by a single server application. So let's create `DemoServerComponent` class extending the `org.restlet.Component` class (you can start from scratch or modify the previously implemented `DemoServer` class).

In the main method use `org.restlet.util.ServerList.add(Protocol protocol, int port)` method to add a new server supporting the given protocol (HTTP) on the specified port (8111).

In the constructor use `org.restlet.routing.Router.attach(String pathTemplate, Class<? extends ServerResource> targetClass)` method to route requests to the proper server resources (`DemoServerResource.class` and `SecondServerResource.class`) based on the URI path (`"/first/"` and `"/second/"` respectively).

```java
package demo;

import org.restlet.Component;
import org.restlet.data.Protocol;

public class DemoServerComponent extends Component {

    public static void main(String[] args) throws Exception {
        new DemoServerComponent().start();
    }

    public DemoServerComponent() {
        getServers().add(Protocol.HTTP, 8111);

        getDefaultHost().attach("/first/", DemoServerResource.class);
        getDefaultHost().attach("/second/", SecondServerResource.class);
    }
}
```
5.2. A server logic
Now we need to implement the logic of SecondServerResource class. You can simply copy the DemoServerResource class and modify its name and the returned values of each method to be able to distinguish the responses generated by the new class from those of the old one.

The implementation of the SecondServerResource should look similar to the one below:

```java
package demo;

import org.restlet.resource.Get;
import org.restlet.resource.Post;
import org.restlet.resource.Put;
import org.restlet.resource.Delete;
import org.restlet.resource.ServerResource;

public class SecondServerResource extends ServerResource {

    @Get
    public String Read() {
        String item = getQueryValue("queryItem");
        return "SecondServerResource: read " + item + "!
";
    }

    @Put
    public String Update(String body) {
        return "SecondServerResource: update " + body + "!
";
    }

    @Post
    public String Create(String body) {
        return "SecondServerResource: create " + body + "!
";
    }

    @Delete
    public String Delete() {
        String item = getQueryValue("queryItem");
        return "SecondServerResource: delete " + item + "!
";
    }
}
```

5.3. A client main program
Modify the DemoClient class by changing the target URI to http://localhost:8111/first/. Copy the content of the main method, rename the second instance of the ClientResource to
secondClientResource and change the target URI in the constructor of secondClientResource to http://localhost:8111/second/.

Now, the DemoClient class should look similar to the one below.

```java
package demo;

import org.restlet.resource.ClientResource;

public class DemoClient {
    public static void main(String[] args) throws Exception {
        demoClientResource.setQueryValue("queryItem", "xyz");
        demoClientResource.get().write(System.out);
        demoClientResource.post("abc").write(System.out);
        demoClientResource.put("abc").write(System.out);
        demoClientResource.delete().write(System.out);

        secondClientResource.setQueryValue("queryItem", "xyz");
        secondClientResource.get().write(System.out);
        secondClientResource.post("abc").write(System.out);
        secondClientResource.put("abc").write(System.out);
        secondClientResource.delete().write(System.out);
    }
}
```

The console output of the client’s application should look alike:

```
Starting the internal HTTP client
read xyz!
create abc!
update abc!
delete xyz!
Starting the internal HTTP client
SecondServerResource: read xyz!
SecondServerResource: create abc!
SecondServerResource: update abc!
SecondServerResource: delete xyz!
```

6. Using interfaces

Till now we were using the standard methods of org.restlet.resource.ClientResource like get() or post() to explicitly choose a type of request we want to send to the server. However, the Restlet Framework provides a more convenient way of interacting with the server – interfaces.
In the last part of this tutorial, let’s create an interface that will allow us to call some of the methods of `SecondServerResource` class from the `DemoClient` using their names.

### 6.1. Interface
First, we have to create an interface `ISecondResource` with headers of provided methods (e.g. `Read` and `Create`) and annotate them with appropriate HTTP methods.

```java
package demo;

import org.restlet.resource.Get;
import org.restlet.resource.Post;

public interface ISecondResource {
    @Get
    public String Read();

    @Post
    public String Create();
}
```

### 6.2. A server logic
The only change that we need to make to the `SecondServerResource` class is to indicate that it implements `ISecondResource interface`.

```java
public class SecondServerResource extends ServerResource implements ISecondResource {
    ...
}
```

### 6.3. A client main program
In the `DemoClient` we need to add a new way of calling `SecondServerResource` methods. First, we need to create an instance of `ISecondResource interface` and initiate it using `org.restlet.resource.ClientResource.wrap(Class<?> resourceInterface)` method. Next, we can call methods provided by the interface in a "standard way", e.g., `secondResource.Read()`. Please note that we still can use `setQueryValue` method to set request parameters.

```java
...
ISecondResource secondResource =
    secondClientResource.wrap(ISecondResource.class);

secondClientResource.setQueryValue("queryItem", "klm");
String responseGet = secondResource.Read();
System.out.print(responseGet);

String responsePost = secondResource.Create("abc");
...
System.out.print(responsePost);

---

**Bibliography**

2. Restlet 2.3 Tutorial - [http://restlet.com/technical-resources/restlet-framework/tutorials/2.3](http://restlet.com/technical-resources/restlet-framework/tutorials/2.3)