Testing RESTful WebServices Made Easy

Introducing the WizTools.org RESTClient, a cross-platform tool to test RESTful WebServices and HTTP communications.

WizTools.org RESTClient is a Java application to test RESTful WebServices. It is also used for testing POX-WebServices (POX: Plain Old XML) over HTTP and other HTTP communications.

History

In early 2007, while working on a series of integration projects, we were using RESTful WebServices. To test our WebServices, I had started the project WizTools.org RESTClient (*rest-client.googlecode.com*). RESTful WebServices itself was born out of disillusionment with its more complex predecessor, the WS-* stack. The bigger technology companies, sitting in their ivory towers had designed the backbone of SOA (Service Oriented Architecture) using a series of specifications collectively called the WS-* stack. These include: SOAP, WSDL, UDDI, WS-Security and WS-Policy, among others [*en.wikipedia.org/wiki/WS-**]. The hacker community, dissatisfied with the introduced complexity of WS-* stack, named a new architecture based on HTTP: RESTful WebServices. The word RESTful WebServices was coined by Roy Fielding in his PhD thesis, "*Architectural Styles and the Design of Network-based Software Architectures*" [*www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*].

RESTful WebServices did not aim at protocol independence. It leverages some lesser-known features of the HTTP protocol. For example, using normal Web browsers, we can make HTTP GET and POST requests. In addition to the common GET and POST, RESTful WebServices use the following:

- PUT
- DELETE

Note: The HTTP specification also defines other request types like HEAD, OPTIONS and TRACE.

Back in 2007, we did not find decent clients to test these types of HTTP requests. Thus WizTools.org RESTClient was born.

Usage

RESTClient requires Java 6. To run the program, use the following command:

```
$ java -jar restclient-ui-2.3-jar-with-dependencies.jar
```

Features

The initial idea behind RESTClient was to acquire the capability to make different kinds of HTTP requests, including GET, POST, PUT and DELETE. The latest version of RESTClient also supports other features like SSL, support for adding custom HTTP headers and body, HTTP BASIC and DIGEST authentication.

One prominent feature of RESTClient is the ability to save requests, responses and the response body. This is often required for regression testing and proof-of-failure. These options are available inside the *File* menu. The related file extensions are:

1. *.rcq*—The request format. This is a RESTClient-specific XML format storing the request details.
2. *.rcs*—The response format. This also is an XML.
3. *.rcr*—This is the archive of both request and response XMLs compressed as a zip file.

Jetty Servlet Container is embedded inside RESTClient. A servlet that verbosely prints out the request details is attached to it. To start the server, use the *Tools* menu. The default listening port of this server is 10101. This can be changed during RESTClient start-up using the system property *rc:trace-server-port*.

The most powerful feature of RESTClient is its integrated support for tests. RESTClient has the Groovy programming language embedded (Figure 2). So test classes can be written in Groovy. Test classes are based on JUnit 3.x and tests are attached to each request. For example, a simple test would look like what's shown below:

```
public class SampleClassTest
    extends org.wiztools.restclient.RESTTestCase{

  // Test method names should start with `test':
  public void testStatus(){
    if(response.getStatusCode() != 200){
      fail("Response status is not 200!");
    }
  }

}
```

As you can observe, the test classes need to extend *org.wiztools.restclient.RESTTestCase*. RESTTestCase internally extends *junit.framework.TestCase*. The instance of RESTTestCase has two predefined instance variables available: *request* and *response*. These instances have various convenient methods to access the *request* and *response* details.

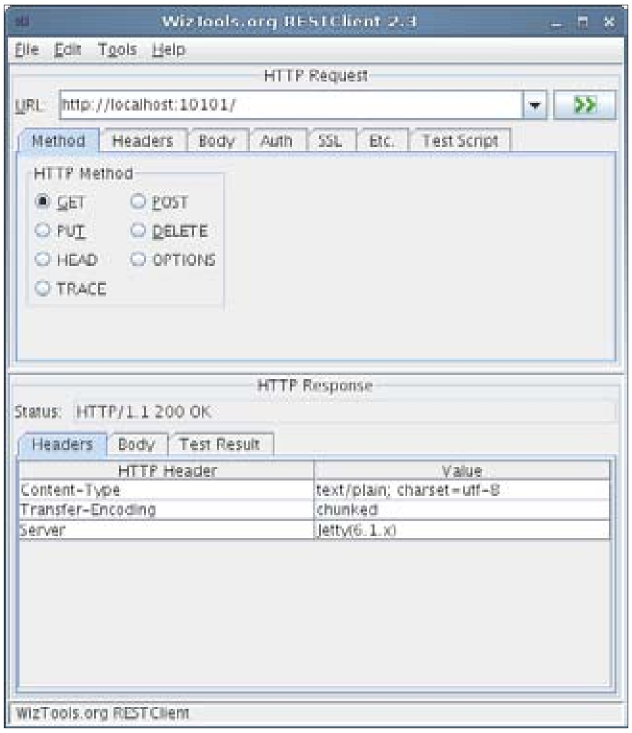
request is of type *org.wiztools.restclient.RoRequestBean*. Useful methods that may be invoked:

- org.wiztools.restclient.HTTPVersion getHttpVersion()
- java.net.URL getUrl()
- java.util.Map<String, String> getHeaders()
- org.wiztools.restclient.RoReqEntityBean getBody()

response is of type *org.wiztools.restclient.RoResponseBean*. Some common methods that can be invoked on this, are:

- int getStatusCode()
- java.lang.String getStatusLine()
- java.util.Map<String, String> getHeaders()
- java.lang.String getResponseBody()

Both the lists are not exhaustive. Please refer to API docs for a complete list of methods that may be invoked.

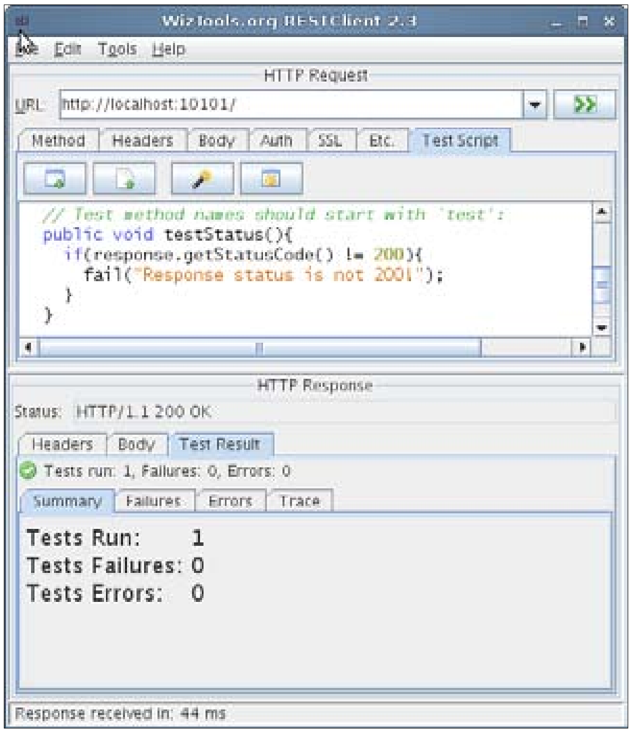


URL: http://localhost:10101/

Status: HTTP/1.1 200 OK

HTTP Header	Value
Content-Type	text/plain; charset=utf-8
Transfer-Encoding	chunked
Server	Jetty(6.1.x)

Figure 1: RESTClient Swing interface


WizTools.org RESTClient 2.3
File Edit Tools Help
HTTP Request
URL:
Method Headers Body Auth SSL Etc. Test Script
HTTP Response
Headers Body Test Result
Summary Failures Errors Trace

URL: http://localhost:10101/

```
// Test method names should start with `test':
public void testStatus(){
  if(response.getStatusCode() != 200){
    fail("Response status is not 200!");
  }
}
```

Status: HTTP/1.1 200 OK

Tests run: 1, Failures: 0, Errors: 0

```
Tests Run:      1
Tests Failures: 0
Tests Errors:   0
```

Response received in: 44 ms

Figure 2: Writing a Groovy test case in RESTClient

The command line

From version 2.3, RESTClient has two binaries: one GUI and one command-line. The command-line tool is used for running requests in a batch and logging their test results. A typical usage is as follows:

```
java -jar restclient-cli-2.3-jar-with-dependencies.jar -o /path/to/responseDir *.rcq
```

This will run all the requests in **.rcq* files in the current working directory, and save the responses (**.rcs*) in the */path/to/responseDir*. The command line client will also print a summary of the test executions.

Extending RESTClient

During the development of version 2.3, the code was re-factored to a more modular form for extensibility. Now the code is organised into various modules managed by Maven. The *restclient-lib* module has the core functionality of RESTClient. Having this as the dependency, various interfaces have been developed (the RESTClient GUI, CLI and Ant plug-in being examples). To demonstrate the ease of the API, I will show you how to write the code to execute a request and write the response-body in the console. First add the dependency for *restclient-lib* in your Maven project [for a detailed discussion on setting up the environment, refer to the Cook Book: *code.google.com/p/rest-client/wiki/Cookbook*]:

```
<dependency>
  <groupId>org.wiztools.restclient</groupId>
  <artifactId>restclient-lib</artifactId>
  <version>2.3</version>
</dependency>
```

Next, write the following code to execute the request:

```
import org.wiztools.restclient.Request;
import org.wiztools.restclient.RequestBean;
import org.wiztools.restclient.HTTPMethod;
import org.wiztools.restclient.View;
import org.wiztools.restclient.Implementation;
import org.wiztools.restclient.RequestExecuter;


// Step 1: Create the request:
RequestBean requestBean = new RequestBean();
requestBean.setUrl(new java.net.URL("http://wiztools.org/"));
requestBean.setMethod(HTTPMethod.GET);

Request request = requestBean;

// Step 2: Write the handler
View view = new View(){
 @Override
 public void doStart(Request request){
   // do nothing!
 }

 @Override
 public void doResponse(Response response){
   System.out.println(response.getResponseBody());
 }

 @Override
 public void doCancelled(){
   // do nothing!
 }

 @Override
 public void doEnd(){
   // do nothing!
 }

 @Override
 public void doError(final String error){
   System.err.println(error);
 }
};

// Step 3: Execute:
RequestExecuter executer = Implementation.of(RequestExecuter.class);
executer.execute(request, view);
```

This example is taken from the *RESTClient Cook Book* [*code.google.com/p/rest-client/wiki/Cookbook*]. The Cook Book has more details on extending.

What next?

Well, there's Ant and Maven integration. Ant integration work has begun but what about any contributors for Maven?

The team

Various people have contributed to RESTClient through suggestions, ideas, testing efforts, documentation and code contribution. I will not be able to list all of them. But the major contributors, besides me, are:

- Ravi Subramaniam: He was a young and energetic lad. Tragedy struck when we lost him in an accident in 2008. He had contributed the initial persistence code.
- Jacky Chan: No, he is not the actor! Jacky is from China, and contributed various bug fixes and modularisation ideas. But his biggest contribution is in making a RESTClient plug-in for IntelliJ IDEA.
- Velrajan: He has contributed some bug fixes and re-wrote the persistence code using XOM.

Other people who have made substantial contributions are Balasubramani S D and Avi Flax. As I said before, there have been other significant contributors, and I am thankful to all of them. END

By: Subhash Chandran

Subhash is a software developer working in the innovation department of the Chennai-based software house, Sella Synergy India Ltd. He has contributed to various Open Source projects, and publishes his contributions on his site WizTools.org. He also maintains a technical blog at indiWiz.com. He may be contacted at: subwiz@gmail.com.