Testing RESTful Web Services made easy using the REST-assured Framework

October 23rd, 2011 by Micha Kops

There are many frameworks out there to facilitate testing RESTful webservices but there is one framework I'd like to acquaint you with my favourite framework named REST-assured.

REST-assured offers a bunch of nice features like a DSL-like syntax, XPath-Validation, Specification Reuse, easy file uploads and those features we're going to explore in the following article.

With a few lines of code and Jersey I have written a RESTful web service that allows us to explore the features of the REST-assured framework and to run tests against this service.

Prerequisites

We're going to need a JDK and Maven .. nothing more …

- Java Development Kit >= 6
- Maven 3

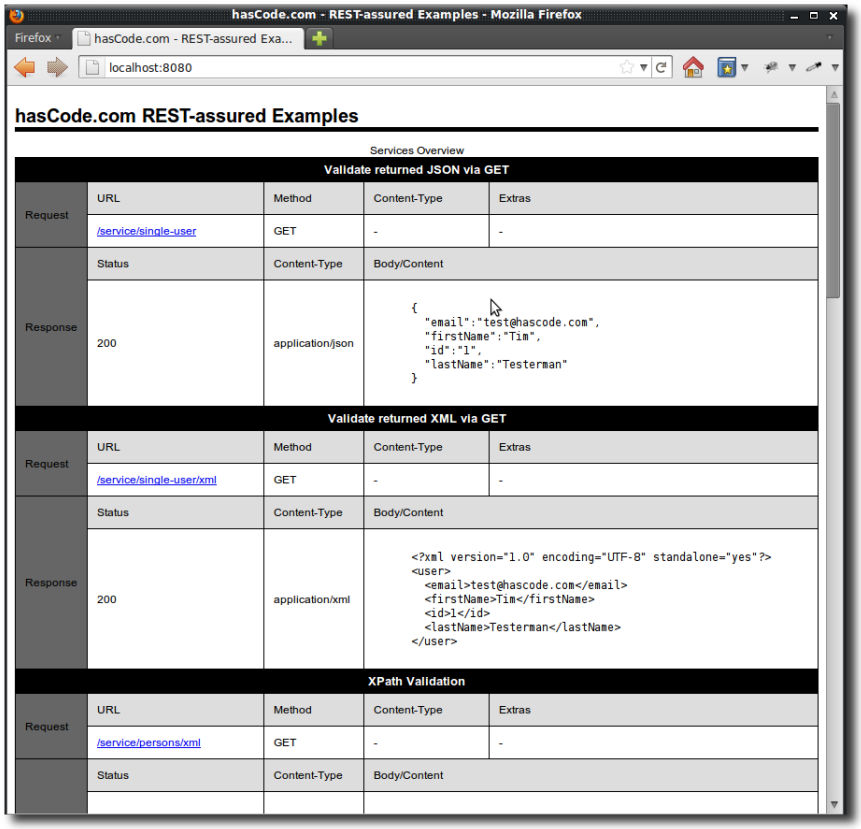
The REST Service to be tested

I have added a demo web application that exposes a RESTful service (Jersey used here) and allows us to run our tests against it. There are two possible ways to run the web app:

- Check out the tutorial sources (see chapter "Tutorial Sources Download") and run

```
mvn tomcat:run
```

- Or simply download the following war file from my Bitbucket repository and deploy it to a valid web container like Tomcat, Jetty etc..
- If you're running the app and open the URL *http://localhost:8080* in your browser you should be able to see a nice overview of the exported service methods



REST Service Overview

Adding REST-assured to your Maven project

You only need to add the following dependencies to your *pom.xml* to use REST-assured and – of course JUnit..

```
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.10</version>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>com.jayway.restassured</groupId>
 <artifactId>rest-assured</artifactId>
 <version>1.4</version>
 <scope>test</scope>
</dependency>
```

Examples

I have added some examples for different scenarios to test .. headers, status codes, cookies, file uploads etc ..

Verify JSON GET Request

We're testing a simple response containing some JSON data here ..

- Request URL: */service/single-user*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Body:

```
{
  "email":"test@hascode.com",
  "firstName":"Tim",
  "id":"1",
  "lastName":"Testerman"
}
```

And this is our test case:

```
@Test
public void testGetSingleUser() {
  expect().
    statusCode(200).
    body(
      "email", equalTo("test@hascode.com"),
      "firstName", equalTo("Tim"),
      "lastName", equalTo("Testerman"),
      "id", equalTo("1")).
    when().
    get("/service/single-user");
}
```

Using JsonPath

This time we're using JsonPath to programatically test the returned JSON structure..

- Request URL: */service/single-user*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Body

```
{
  "email":"test@hascode.com",
  "firstName":"Tim",
  "id":"1",
  "lastName":"Testerman"
}
```

And this is our test:

```
@Test
public void testGetSingleUserProgrammatic() {
  Response res = get("/service/single-user");
  assertEquals(200, res.getStatusCode());
  String json = res.asString();
  JsonPath jp = new JsonPath(json);
  assertEquals("test@hascode.com", jp.get("email"));
  assertEquals("Tim", jp.get("firstName"));
  assertEquals("Testerman", jp.get("lastName"));
  assertEquals("1", jp.get("id"));
}
```

Using Groovy Closures

JsonPath allows us to use Groovy closures to perform searches on the returned JSON structure.

- Request URL: */service/persons/json*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Body

```
{
        "person":[
        {
                "@id":"1",
                "email":"test@hascode.com",
                "firstName":"Tim",
                "lastName":"Testerman"
        },{
                "@id":"20",
                "email":"dev@hascode.com",
                "firstName":"Sara",
                "lastName":"Stevens"
        },{
                "@id":"11",
                "email":"devnull@hascode.com",
                "firstName":"Mark",
                "lastName":"Mustache"
        }
        ]
}
```

And this is our test – we're searching for a person whose email matches the pattern /*test@*/:

```
@Test
public void testFindUsingGroovyClosure() {
  String json = get("/service/persons/json").asString();
  JsonPath jp = new JsonPath(json);
  jp.setRoot("person");
  Map person = jp.get("find {e -> e.email =~ /test@/}");
  assertEquals("test@hascode.com", person.get("email"));
  assertEquals("Tim", person.get("firstName"));
  assertEquals("Testerman", person.get("lastName"));
}
```

Verifying XML

Now we're going to validate returned XML

- Request URL: */service/single-user/xml*

- Request Method: GET
- Response Content-Type: *application/xml*
- Response Body

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <email>test@hascode.com</email>
  <firstName>Tim</firstName>
  <id>1</id>
  <lastName>Testerman</lastName>
</user>
```

And this is our test:

```
@Test
public void testGetSingleUserAsXml() {
  expect().
    statusCode(200).
    body(
      "user.email", equalTo("test@hascode.com"),
      "user.firstName", equalTo("Tim"),
      "user.lastName", equalTo("Testerman"),
      "user.id", equalTo("1")).
    when().
    get("/service/single-user/xml");
}
```

XML using XPath

To validate complex XML structure XPath is way more comfortable here..

- Request URL: */service/persons/xml*
- Request Method: GET
- Response Content-Type: *application/xml*
- Response Body

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<people>
  <person id="1">
    <email>test@hascode.com</email>
    <firstName>Tim</firstName>
    <lastName>Testerman</lastName>
  </person>
  <person id="20">
    <email>dev@hascode.com</email>
    <firstName>Sara</firstName>
    <lastName>Stevens</lastName>
  </person>
  <person id="11">
    <email>devnull@hascode.com</email>
    <firstName>Mark</firstName>
    <lastName>Mustache</lastName>
  </person>
</people>
```

And this is our test:

```
@Test
public void testGetPersons() {
 expect()
 .statusCode(200)
 .body(hasXPath("//*[self::person and self::person[@id='1'] and self::person/email[text()='test@hascode.com'] and self::person/firstName[text()
 .body(hasXPath("//*[self::person and self::person[@id='20'] and self::person/email[text()='dev@hascode.com'] and self::person/firstName[text()
 .body(hasXPath("//*[self::person and self::person[@id='11'] and self::person/email[text()='devnull@hascode.com'] and self::person/firstName[te
 .when().get("/service/persons/xml");
}
```

XML verification vs a Schema

Now we're going to validate the xml returned against a XML schema file

- Request URL: */service/single-user/xml*
- Request Method: GET
- Response Content-Type: *application/xml*
- Response Body

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <email>test@hascode.com</email>
  <firstName>Tim</firstName>
  <id>1</id>
  <lastName>Testerman</lastName>
</user>
```

This is the schema we're using to validate in a file named *user.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">

	<element name="user">
		<complexType>
			<sequence>
				<element name="email">
					<simpleType>
						<restriction base="string">
							<pattern value=".+@.+"></pattern>
						</restriction>
					</simpleType>
				</element>
				<element name="firstName" type="string"></element>
				<element name="id" type="int"></element>
				<element name="lastName" type="string"></element>
			</sequence>
		</complexType>
	</element>
</schema>
```

And this is our test case:

```
@Test
public void testGetSingleUserAgainstSchema() {
  InputStream xsd = getClass().getResourceAsStream("/user.xsd");
  assertNotNull(xsd);
  expect().
  statusCode(200).
  body(
    matchesXsd(xsd)).
  when().
  get("/service/single-user/xml");
}
```

Handling Request Parameters

This is a simple example how to add some request parameters

- Request URL: */service/user/create*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Body

```
{
  "email":"test@hascode.com",
  "firstName":"Tim",
  "id":"1",
  "lastName":"Testerman"
}
```

And this is our test:

```
@Test
public void testCreateuser() {
  final String email = "test@hascode.com";
  final String firstName = "Tim";
  final String lastName = "Tester";

  given().
    parameters(
      "email", email,
      "firstName", firstName,
      "lastName", lastName).
  expect().
    body("email", equalTo(email)).
    body("firstName", equalTo(firstName)).
    body("lastName", equalTo(lastName)).
  when().
  get("/service/user/create");
}
```

HTTP Status Code

Now an example how to verify HTTP headers – in the following example, a *404 Page Not Found* is returned ..

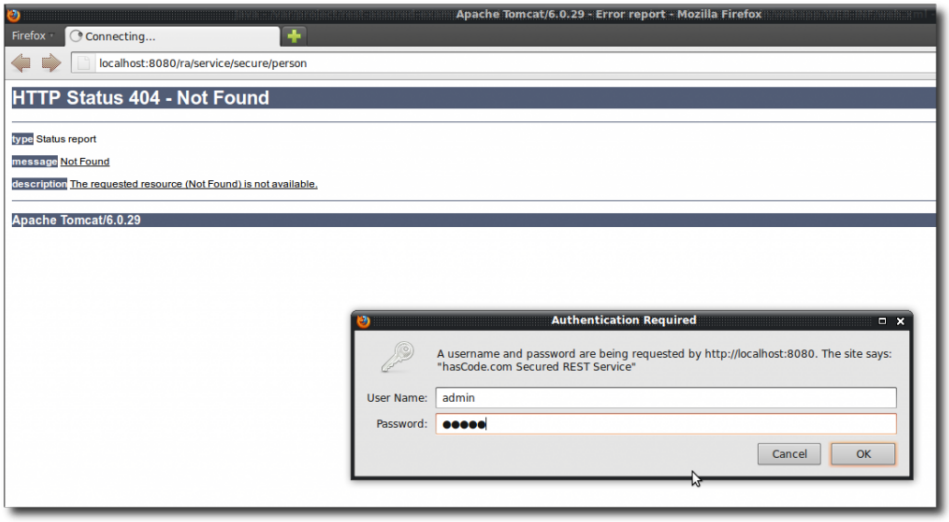
- Request URL: */service/status/notfound*
- Request Method: GET
- Response Content-Type: *text/plain*
- Response Status: 404 / Page Not Found

And this is our test:

```
@Test
public void testStatusNotFound() {
  expect().
    statusCode(404).
  when().
  get("/service/status/notfound");
}
```

Authentication

In this example we're handling basic authentication ..



Basic Authentication secured REST Service

- Request URL: */service/secure/person*
- Request Method: GET
- Response Content-Type: *text/plain*
- Response Status: 401 Unauthorized/ 200 Status Ok (when logged in with username=admin and password=admin)

And this is our test:

```
@Test
public void testAuthenticationWorking() {
  // we're not authenticated, service returns "401 Unauthorized"
  expect().
    statusCode(401).
  when().
  get("/service/secure/person");

  // with authentication it is working
  expect().
    statusCode(200).
  when().
    with().
      authentication().basic("admin", "admin").
  get("/service/secure/person");
}
```

Setting HTTP Headers

In the following example we're setting some HTTP headers. The value of the HTTP header named "myparam" is returned by the REST service in the response body..

- Request URL: */service/single-user*
- Request Method: GET
- Response Content-Type: *text/plain*
- Response Body: #value-of-myparam#

And this is our test:

```
@Test
public void testSetRequestHeaders() {
  expect().
    body(equalTo("TEST")).
  when().
    with().
    header("myparam", "TEST").
  get("/service/header/print");

  expect().
    body(equalTo("foo")).
  when().
    with().
    header("myparam", "foo").
  get("/service/header/print");
}
```

Verifying HTTP Headers

Now we're going to verify HTTP response headers

- Request URL: */service/header/multiple*
- Request Method: GET
- Response Content-Type: *text/plain*
- Response Header: *customHeader1:foo, anotherHeader:bar*

And this is our test:

```
@Test
public void testReturnedHeaders() {
  expect().
    headers("customHeader1", "foo", "anotherHeader", "bar").
  when().
  get("/service/header/multiple");
}
```

Setting Cookies

The following example shows how to set cookies. The REST service sends a 403 / Forbidden until a cookie with name=authtoken and value=abcdef is send.

- Request URL: */service/access/cookie-token-secured*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Status: 403 / 200

And this is our test:

```
@Test
public void testAccessSecuredByCookie() {
  expect().
    statusCode(403).
  when().
  get("/service/access/cookie-token-secured");

  given().
    cookie("authtoken", "abcdef").
  expect().
    statusCode(200).
  when().
  get("/service/access/cookie-token-secured");
}
```

Verifying Cookies

This is how to verify cookies set by the service. The service returns the request parameter “name” as the value of the cookie named “userName”:

- Request URL: */service/cookie/modify*
- Request Method: GET
- Request Parameter: name
- Response Cookie: userName:#value-of-name#

And this is our test:

```
@Test
public void testModifyCookie() {
  expect().
    cookie("userName", equalTo("Ted")).
  when().
    with().param("name", "Ted").
  get("/service/cookie/modify");

  expect().
    cookie("userName", equalTo("Bill")).
  when().
    with().param("name", "Bill").
  get("/service/cookie/modify");
}
```

File Uploads

The following example shows how to handle file uploads. We’re sending a text file to the REST service and the service returns the file content as a string in the response body.

- Request URL: */service/file/upload*
- Request Method: GET
- Request Content-Type: *multipart/form-data*
- Response Content-Type: *text/plain*
- Response Body: #file-content#

And this is our test:

```
@Test
public void testFileUpload() {
  final File file = new File(getClass().getClassLoader()
      .getResource("test.txt").getFile());
  assertNotNull(file);
  assertTrue(file.canRead());
  given().
    multiPart(file).
  expect().
    body(equalTo("This is an uploaded test file.")).
  when().
  post("/service/file/upload");
}
```

Registering custom parsers for MIME-types

Sometimes you've got to handle a RESTful service that returns an invalid content type so that REST-assured does not know which parser to use to process the response. This is not a real problem though because the framework allows you to register parsers for a given content type as shown in the example below:

- Request URL: */service/detail/json*
- Request Method: GET
- Response Content-Type: *text/json*
- Response Body:

```
{"test":true}
```

And this is our test:

```
@Test
public void testRegisterParserForUnknownContentType() {
  RestAssured.registerParser("text/json", Parser.JSON);
  expect().
    body("test", equalTo(true)).
  when().
  get("/service/detail/json");
}
```

Specification reuse

Another nice feature of the REST-assured framework the possibility to create specifications and reuse, modify or extend them in several tests.

- Request URL: */service/single-user* / */service/user/create*
- Request Method: GET
- Response Content-Type: *application/json*
- Response Body

```
{
  "email":"test@hascode.com",
  "firstName":"Tim",
  "id":"1",
  "lastName":"Testerman"
}
```

And this is our test:

```
@Test
public void testSpecReuse() {
  ResponseSpecBuilder builder = new ResponseSpecBuilder();
  builder.expectStatusCode(200);
  builder.expectBody("email", equalTo("test@hascode.com"));
  builder.expectBody("firstName", equalTo("Tim"));
  builder.expectBody("lastName", equalTo("Testerman"));
  builder.expectBody("id", equalTo("1"));
  ResponseSpecification responseSpec = builder.build();

  // now we're able to use this specification for this test
  expect().
    spec(responseSpec).
  when().
  get("/service/single-user");

  // now re-use for another test that returns similar data .. you may
  // extend the specification with further tests as you wish
  final String email = "test@hascode.com";
  final String firstName = "Tim";
  final String lastName = "Testerman";

  expect().
    spec(responseSpec).
  when().
    with().
    parameters(
      "email", email,
      "firstName", firstName,
      "lastName",lastName).
  get("/service/user/create");
}
```

Troubleshooting

- "*WARNING: Cannot find parser for content-type: text/json — using default parser.*" – The framework does not know for sure which parser to use. Register the corresponding parser like this: e.g. *RestAssured.registerParser("text/json", ParserJSON);*
- *"If I use the War file, the path is no longer localhost:8080, but becomes http://localhost:8080/rest-assured-example/."* - specify the changed context path in the *get()* method or – more comfortable – define it as a global setting e.g. like this one

```
@Before
 public void setUp(){
 RestAssured.basePath = "yourbasepath";
 }
```

Tutorial Sources

I have put the source from this tutorial on my Bitbucket repository – download it there or check it out using Mercurial: