



Rapise® | Web Service Testing Tutorials
Inflectra Corporation

Date: May 4th, 2017



Contents

Introduction	1
1. Testing REST Web Services	2
1.1. What is REST and what is a RESTful web service?.....	2
1.2. Overview	2
1.3. Using the REST Query Builder ..	2
1.4. Saving the REST Requests as Objects	9
1.5. Generating REST Test Scripts	11
1.6. Writing REST Test Scripts	16
2. Testing SOAP Web Services	19
2.1. What is SOAP and what is a SOAP web service?	19
2.2. Overview	19
2.3. Inspecting the SOAP WSDL Endpoint	19
2.4. Invoking the SOAP Actions.....	22
2.5. Generating the Rapise Test Script	24

Introduction

Rapise® is a next generation software test automation tool that leverages the power of open architecture to improve application quality and reduce time to market.

This guide provides a quick step-by-step tutorial for creating a sample Rapise tests that can test the two main different types of web service – SOAP and REST.

For further information on using Rapise, please refer to the more comprehensive *Rapise User Guide*. For information on using Rapise in conjunction with our SpiraTest test management system, please refer to the *Using Rapise with SpiraTest Guide*.

Rapise contains a built-in web service module that can currently test the following types of web service:

1. **REST Web Services** - Rapise contains a built-in REST definition builder and object library that allows you to prototype out your REST web service requests, inspect the returned HTTP headers and HTTP response body and then convert into a parameterized set of Rapise objects that can be scripted against in the main Rapise JavaScript editor. It also includes built-in support for verifying the data returned as Rapise checkpoints.
2. **SOAP Web Services** - Rapise contains a built-in SOAP request tester and object library that allows you to prototype out your SOAP web service requests, inspect the returned HTTP headers and SOAP response body and then convert into a parameterized set of Rapise objects that can be scripted against in the main Rapise JavaScript editor. It also includes built-in support for verifying the data returned as Rapise checkpoints.

1. Testing REST Web Services

In this section you shall learn how to test a RESTful web services API using Rapise. We shall be using a demo application called **Library Information System** that has a dummy RESTful web service API available for learning purposes. You can access this sample application at <http://www.libraryinformationssystem.org>, and its RESTful web service API can be found at: www.libraryinformationssystem.org/Services/RestService.aspx.

1.1. What is REST and what is a RESTful web service?

REpresentational State Transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged as a web API design model that offers greater simplicity over other web service protocols such as SOAP and XML-RPC.

A RESTful web API (also called a RESTful web service) is a web API implemented using HTTP and REST principles. Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs. This is because REST is an architectural style, unlike SOAP, which is a protocol.

1.2. Overview

Creating a REST web service test in Rapise consists of the following steps:

1. Using the REST query builder to create the various REST web service requests and verify that they return the expected data in the expected format.
2. Parameterizing these REST web service requests into reusable templates and saving as Rapise learned objects.
3. Generating the test script in Javascript that uses the learned Rapise web service objects.

We shall discuss each of these steps in turn.

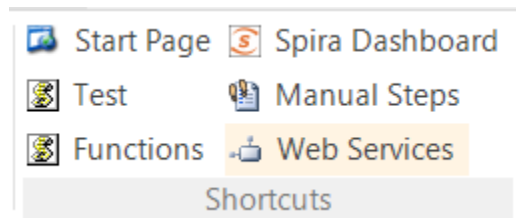
1.3. Using the REST Query Builder

Create a new test in Rapise called MyRestTest1.sstest.

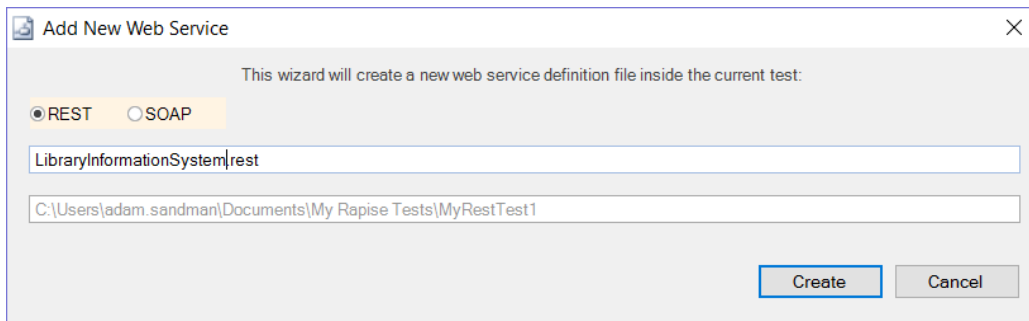
- For **Methodology**, choose **Basic: Windows Desktop Application** and Rapise will create a new blank test project. If you plan on using a combination of Web or Mobile UI tests in the same script, you could choose one of the other types.
- For **Scripting Language**, choose **JavaScript**. The scriptless Rapise Visual Language (RVL) can be used with web service tests, but it means that all the web service tests need to be in a JavaScript subroutine / scenario that is called from the RVL test.

Rapise will create a new blank test project.

Once you have created it, click on the "Web Services" icon in the Test ribbon to add a new web service definition to your test project:

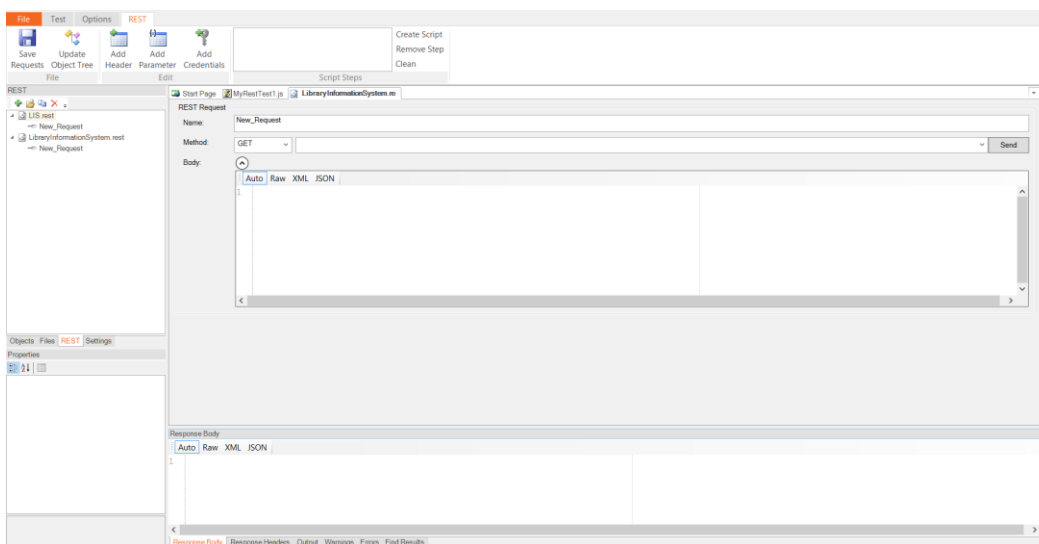


This will display the Add New Web Service dialog box:



Choose **REST** as the type of web service you want to create.

Then, enter the name of the web service that you're going to add, in this case enter "**LibraryInformationSystem.rest**" and click "Create". This will add the REST web services definition file to your test project:



You will see on the right hand side, there is a new document editor for the .rest file. This is the REST web services query form. It lets you send test HTTP requests to the web service under test and inspect the output being returned.

If you open up API documentation for our sample application (www.libraryinformationsystem.org/Services/RestService.aspx) you will see that it exposes several operations for retrieving, adding, updating and deleting books and authors in the system. For this tutorial we shall perform the following operations:

1. Get the special SessionID to identify our test session
2. Get a list of books in the system
3. Add a new book to the system and verify that it was added

According to the documentation that means we will need to send the following requests:

(i) Get a Unique Session

URL: `http://www.libraryinformationsystem.org/Services/RestService.svc/session`

Method: GET

Returns: Unique session ID that is passed to other requests to keep data separate for different demo

users

(ii) Get this list of books

URL: `http://www.libraryinformationssystem.org/Services/RestService.svc/book?session_id={session_id}`

Method: GET

Returns
: Array of book objects

(iii) Add a new book to the list

URL: `http://www.libraryinformationssystem.org/Services/RestService.svc/book?session_id={session_id}`

Method: POST

Pass a populated book object:

Body:

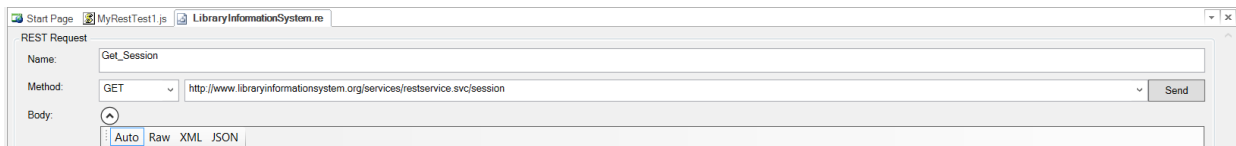
```
{
  "Name": "Book Name",
  "AuthorId": 1,
  "GenreId": 1,
}
```

Returns
: Single book object that has its BookId populated

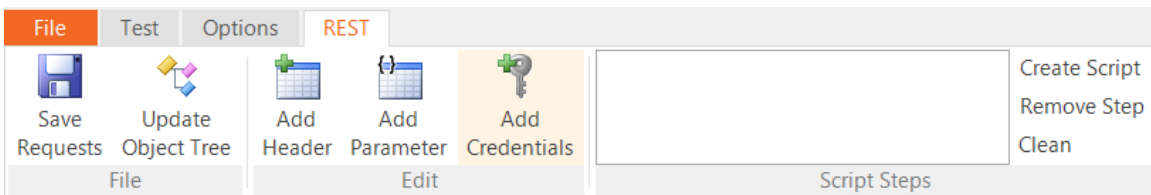
The first request will be to get the unique session ID that we will need to pass to the other requests. This is needed by our sample application to prevent testing by different users interfering with each other. To create this request, simply enter the following information on the REST Request form:

- **Name:** Get_Session
- **Method:** GET
- **URL:** `http://www.libraryinformationssystem.org/Services/RestService.svc/session`

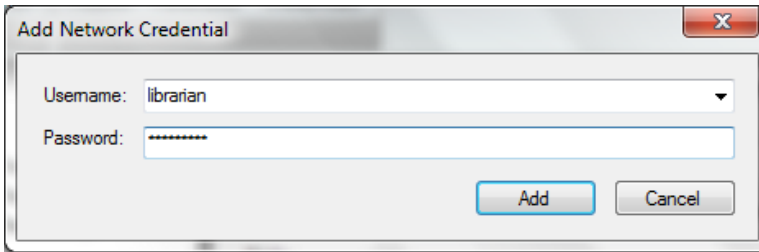
You should now have it populated as illustrated below:



This web service request requires that we pass credentials by means of HTTP Basic authentication. So click on the "REST" tab in the Rapise ribbon and click on the "Add Credentials" button.



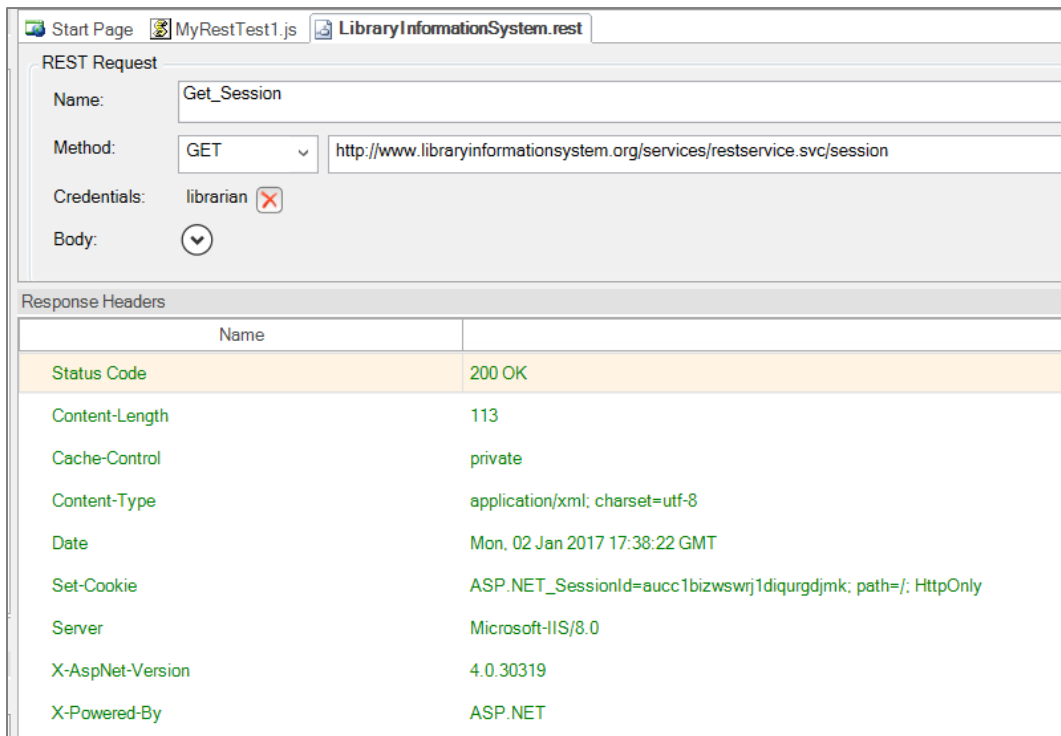
This will display the "Add Credentials" dialog box:



The "Add Network Credential" dialog box contains two input fields: "Username:" with a dropdown menu showing "librarian" and "Password:" with a masked field of seven asterisks. At the bottom right, there are two buttons: "Add" and "Cancel".

Enter **librarian** as both the username and password and click "Add".

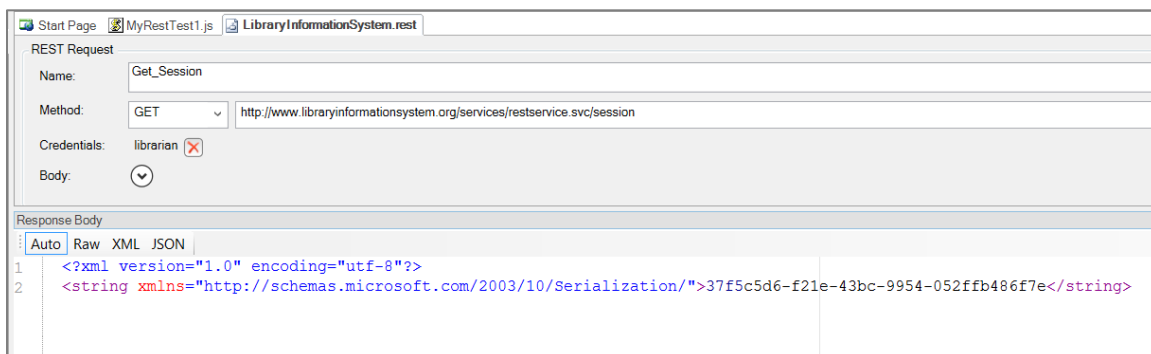
Now click the "Send" button and the request will get sent to the web service:



The REST client interface shows the "Response Headers" tab selected. The "REST Request" section displays: Name: Get_Session, Method: GET, URL: http://www.libraryinformationsystem.org/services/restservice.svc/session, Credentials: librarian, and Body: (empty). The "Response Headers" table is as follows:

Name	Value
Status Code	200 OK
Content-Length	113
Cache-Control	private
Content-Type	application/xml; charset=utf-8
Date	Mon, 02 Jan 2017 17:38:22 GMT
Set-Cookie	ASP.NET_SessionId=aucc1bizswrj1diqurgdjm; path=/; HttpOnly
Server	Microsoft-IIS/8.0
X-AspNet-Version	4.0.30319
X-Powered-By	ASP.NET

The Response Header tab will display the headers coming back from the web service. The Status Code **200 OK** means that the request succeeded and that data was returned. If you click on the "**Response Body - XML**" tab, you will see the XML serialized data returned from the web service:



The REST client interface shows the "Response Body" tab selected. The "REST Request" section is the same as in the previous screenshot. The "Response Body" section shows the XML data:

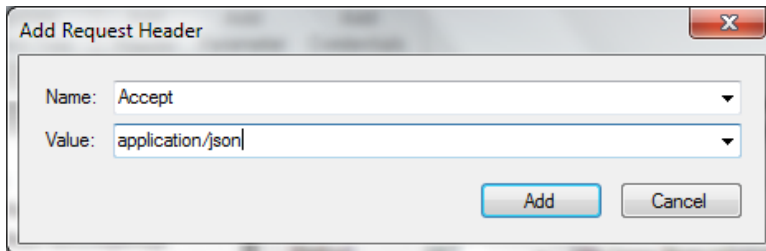
```
1 <?xml version="1.0" encoding="utf-8"?>
2 <string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">37f5c5d6-f21e-43bc-9954-052ffb486f7e</string>
```

Since Rapise uses JavaScript as its scripting language, it is usually easier to work with JSON (JavaScript Object Notation) serialized data rather than XML. In the case of the sample Library Information System

web service, you can change the format that it accepts and retrieves by sending two special HTTP headers:

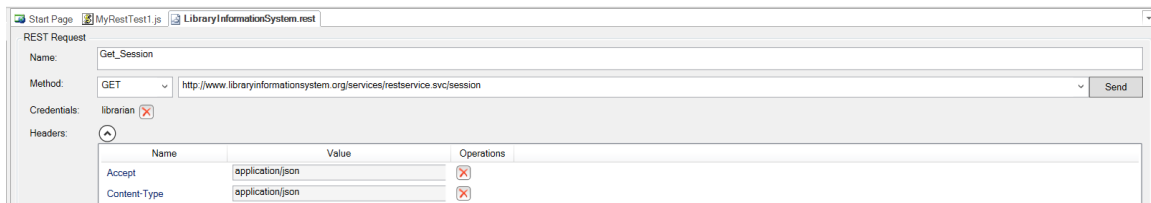
- **Content-Type:** application/json
- **Accept:** application/json

To add these headers to the request, simply click on the "Add Header" button in the REST ribbon tab. This will display the following dialog box:



The dialog box is titled "Add Request Header" and has a close button (X) in the top right corner. It contains two input fields: "Name:" with a dropdown menu showing "Accept" and "Value:" with a text box containing "application/json". At the bottom, there are two buttons: "Add" and "Cancel".

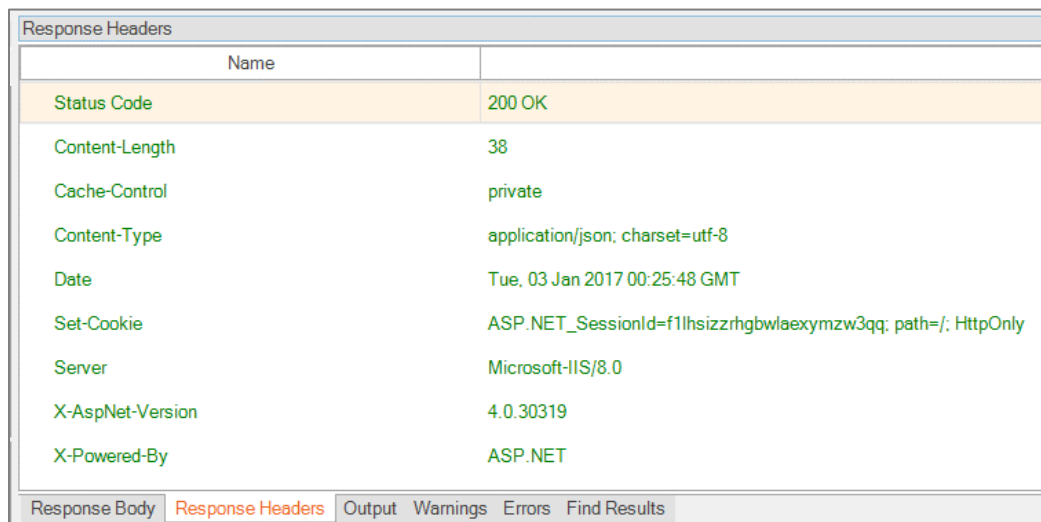
Choose the HTTP Header "**Accept**" from the list and enter "**application/json**" as the value. Repeat for the "**Content-Type**" header. You should now have the following populated request:



The screenshot shows the REST client interface. The "REST Request" section is active. The "Name" field contains "Get_Session". The "Method" is set to "GET" and the "URL" is "http://www.libraryinformationssystem.org/services/restservice.svc/session". The "Credentials" field is set to "librarian". The "Headers" section is expanded, showing a table with two headers added:

Name	Value	Operations
Accept	application/json	[X]
Content-Type	application/json	[X]

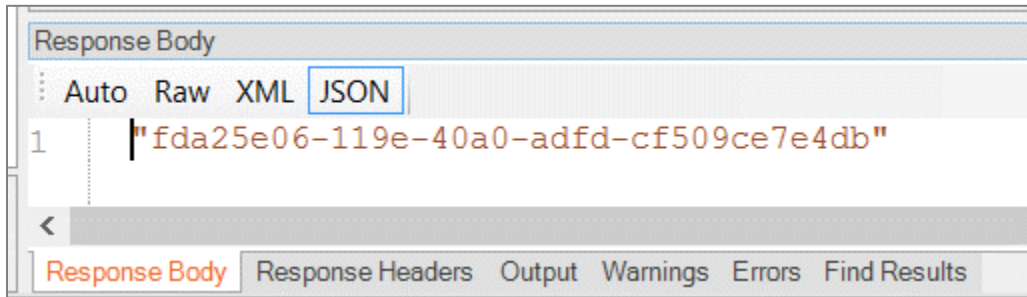
Now click the "Send" button and the request will get sent to the web service:



The screenshot shows the "Response Headers" tab of the REST client. It displays a table of response headers:

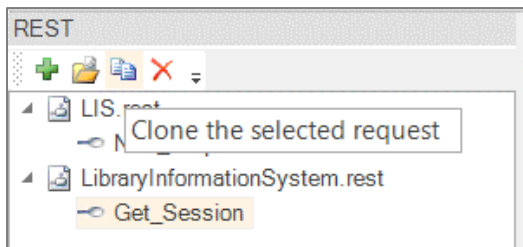
Name	Value
Status Code	200 OK
Content-Length	38
Cache-Control	private
Content-Type	application/json; charset=utf-8
Date	Tue, 03 Jan 2017 00:25:48 GMT
Set-Cookie	ASP.NET_SessionId=f1lhsizzrhgbwlaexymzw3qq; path=/; HttpOnly
Server	Microsoft-IIS/8.0
X-AspNet-Version	4.0.30319
X-Powered-By	ASP.NET

The Response Header tab will display the headers coming back from the web service. Note that the returned Content-Type is listed as "application/json" as requested. If you click on the "Formatted JSON" tab, you will see the JSON serialized data returned from the web service:

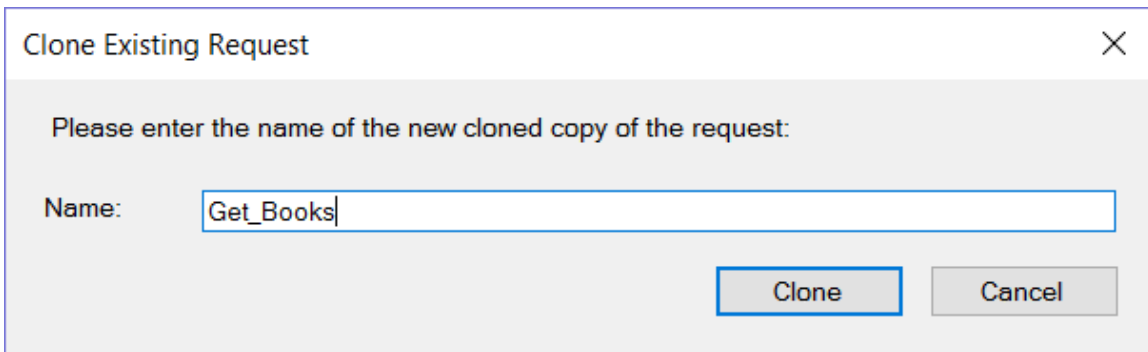


We have now completed the creation of our first test operation. Click on the "Save Requests" button in the Rapise REST Ribbon to make sure our changes have been saved.

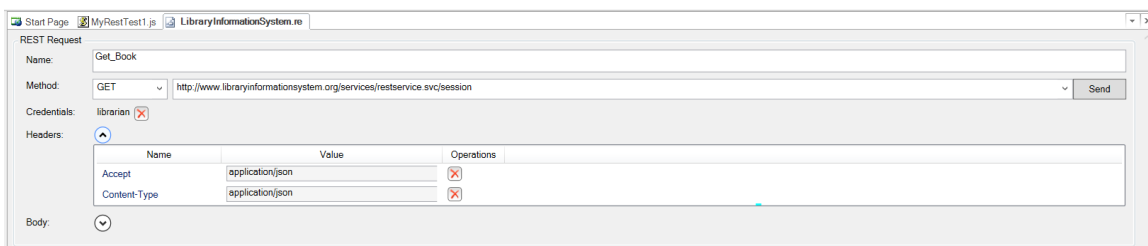
Now click on the "Clone request" icon in the REST request explorer in the left-hand side of the screen:



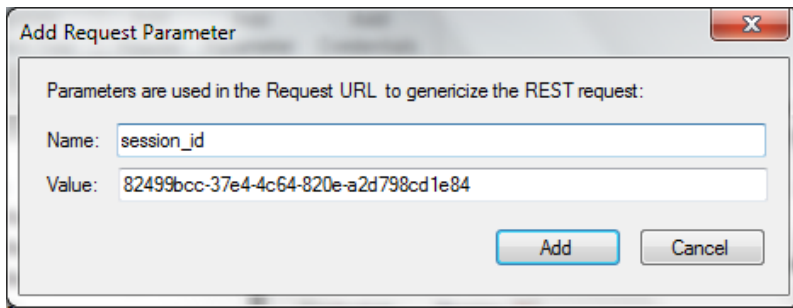
This will display the Clone Request dialog box. This lets us create a new REST request that contains the headers and authentication already defined on our existing request. This will save time over creating a new REST request from scratch:



Enter the name "Get_Books" in the dialog box and click the "Clone" button. This will create a new REST request with this name:



For this request, we need to pass through the SessionID in the querystring. Rather than hardcoding it in the URL, we can make use of the parameterization feature of Rapise. Click on the "Add Parameter" button in the Rapise REST Ribbon. This will display the "Add Request Parameter" dialog box:



Enter in the following:

- o **Name: session_id**

Value: 82499bcc-37e4-4c64-820e-a2d798cd1e84

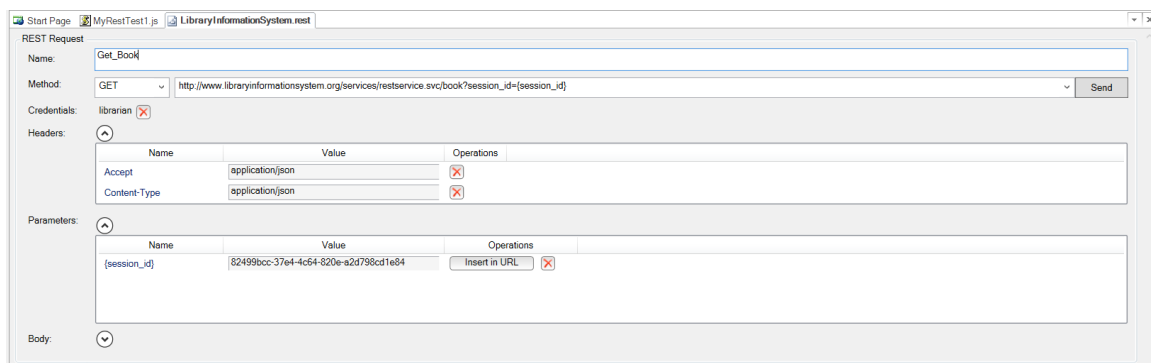
(you can also copy and paste the value returned by the Get_Session command)

Now, click the "Add" button and the parameter will be added to the request.

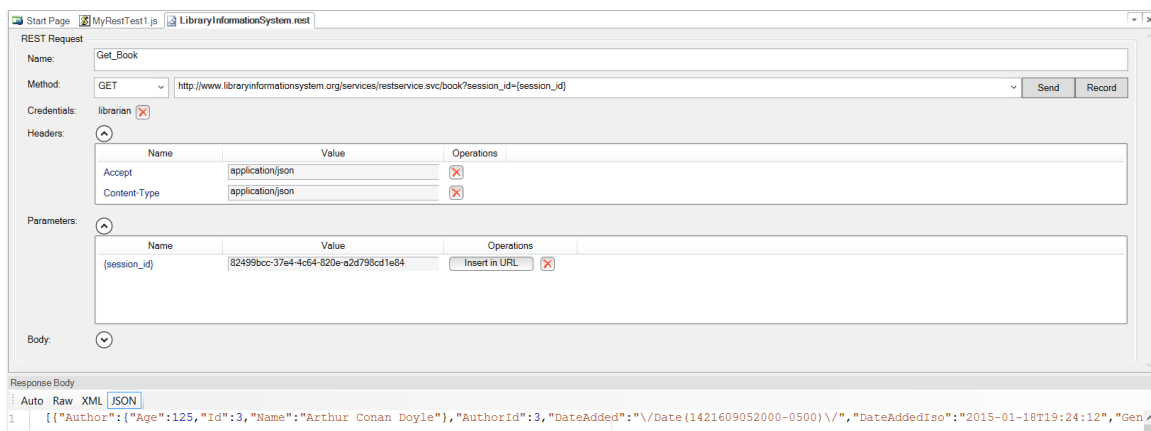
Next, change the URL to:

`http://www.libraryinformationssystem.org/Services/RestService.svc/book?session_id=`

Then position the caret at the end of this URL and click the "Insert in URL" button. This will insert the parameter token in the URL at the specified point:

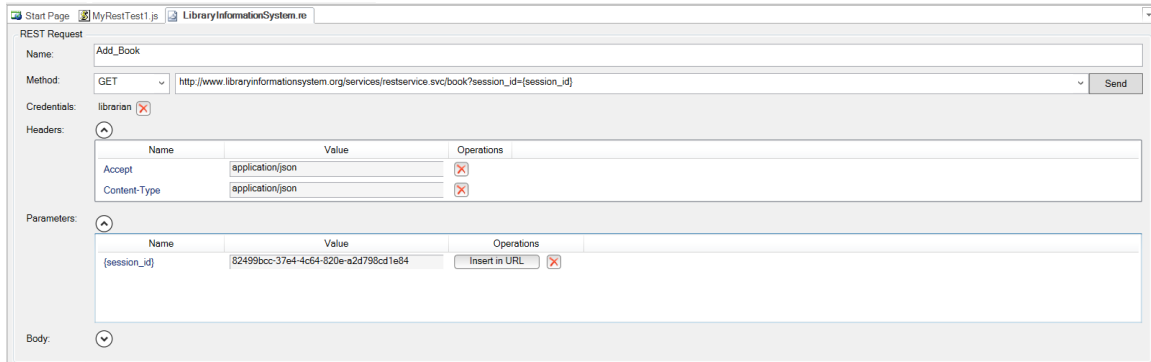


Now click the "Send" button and the request will get sent to the web service. This will return the list of books serialized as a JSON array of objects:



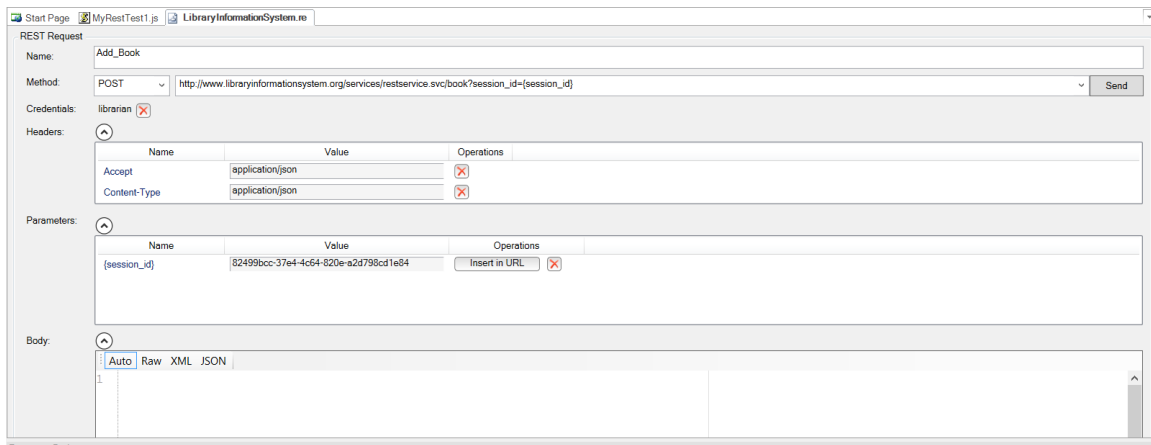
We have now completed the creation of our second test operation. Click on the "Save Requests" button in the Rapise REST Ribbon to make sure our changes have been saved.

Now click on the "Clone request" icon in the REST request explorer in the right-hand side of the screen. Enter the name "Add_Book" in the dialog box and click the "Clone" button. This will create a new REST request with this name:



This operation will add a new book to the system, so it's a POST request. Change the Method type in the dropdown list from "GET" to "POST".

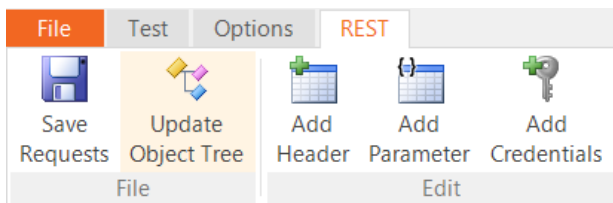
Expand the "Body" field on the form. This is where you can enter in an XML or JSON serialized Book record that will get added to the system. For now we'll leave this blank and let Rapise serialize the body for us later on when we actually write our test script. So we should now have:



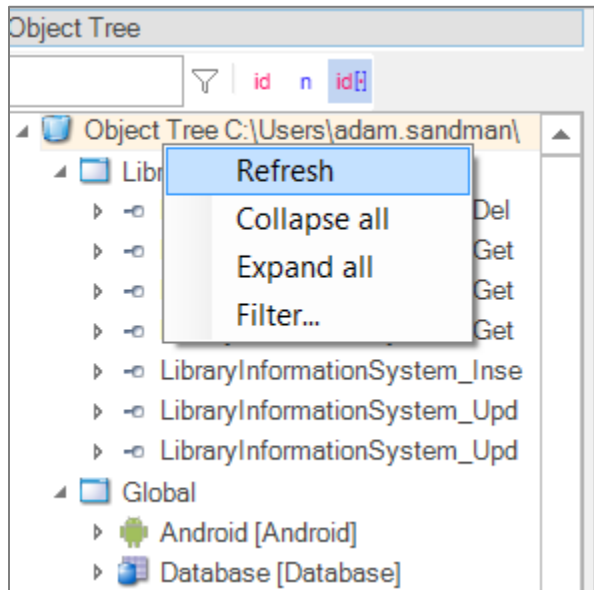
We have now completed the creation of our third test operation. Click on the "Save Requests" button in the Rapise REST Ribbon to make sure our changes have been saved.

1.4. Saving the REST Requests as Objects

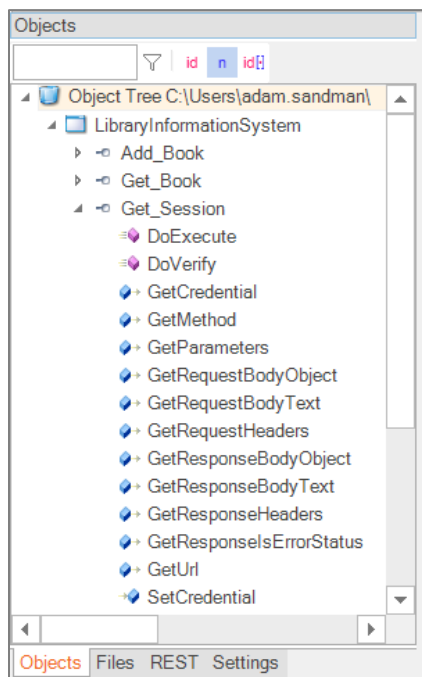
Now that we have created our three REST requests, the next step is to actually create the Rapise objects that we can use in our JavaScript test scripts. Click on the "Update Object Tree" button in the Rapise REST Ribbon to tell Rapise to update the Object Tree with our new requests:



Rapise will open a command prompt window in the background and then display a confirmation message once the Object Tree has been updated. Click on the "Object" tab of the main Rapise explorer, then right-click on the Object-Tree root node to bring up the context menu:



Click on the Refresh icon and you will see the "LibraryInformationSystem" heading displayed, with the three saved REST request listed underneath:



If you expand one of the REST requests (e.g. Add_Book), you'll see that it has a single operation "**DoExecute**" that executes the web services and a series of properties available for inspecting or updating any part of the REST request prior to it being sent to the server.

In the next section we shall illustrate how you can write a test script using these learned objects:

- a) You can either have Rapise **generate test scripts** and verification points automatically (described in section 1.5), or
- b) You can manually **write the test scripts** using the objects and the Rapise code editor (described in section 1.6)

1.5. Generating REST Test Scripts

Inside the REST request explorer, double-click on the **Get_Session** function to open up the request:

REST Request

Name: Get_Session

Method: GET http://www.libraryinformationssystem.org/services/restservice.svc/session Send

Credentials: librarian

Headers:

Name	Value	Operations
Accept	application/json	X
Content-Type	application/json	X

Click on the **Send** button to send the sample request. Once that has succeeded, you will see the **Record** button appear to the right:

REST Request

Name: Get_Session

Method: GET http://www.libraryinformationssystem.org/services/restservice.svc/session Send Record

Credentials: librarian

Headers:

Name	Value	Operations
Accept	application/json	X
Content-Type	application/json	X

Click that button and the request will get added to the list of recorded steps:

Get_Session GET

Create Script
Remove Step
Clean

Script Steps

Now open up the **Get_Books** request and follow the same procedure:

1. Click on the **'Send'** button to execute the request
2. Click on the **'Record'** button to record the action as a script step

This time we also want to verify the result. You will see a list of books returned in the **Verify** box underneath the Body section:

Get_Books

GET http://www.libraryinformationssystem.org/services/restservice.svc/book?session_id={session_id} Send Record Verify

librarian

response[14]

- 0
 - Author
 - Age=125
 - Id=3
 - Name=Arthur Conan Doyle
 - AuthorId=3
 - DateAdded=1/18/2015 2:24:12 PM
 - DateAddedIso=1/18/2015 7:24:12 PM
 - Genre
 - GenreId=2

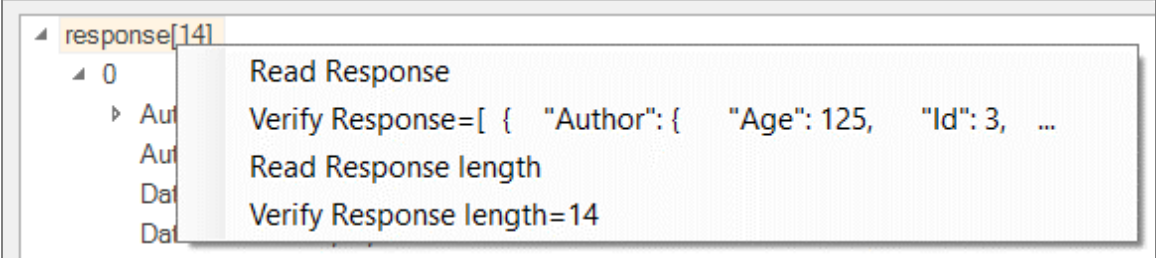
If you select the overall array **response[14]** and click the main **'Verify'** button next to the Record button, the system will automatically add a verification step that verifies all of the values. To try this, click the **Verify** button. This will add a bold verification step to the recorded script:

```
Get_Session GET
*Get_Books GET
```

You will see a script step recorded with a verification test added (it's shown in bold with an asterisk*):

However, in many cases you only want to verify certain properties. For example, we might want to just verify that 14 books are returned, and that the first book has the right name.

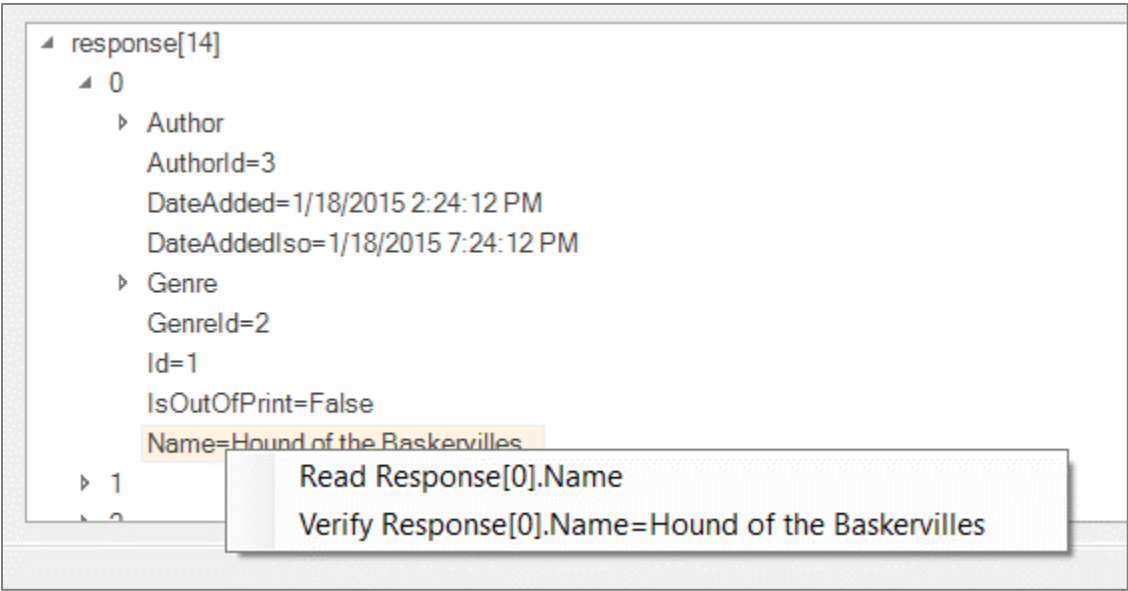
To do this, right-click on the **response[14]** entry to display the verification content menu:



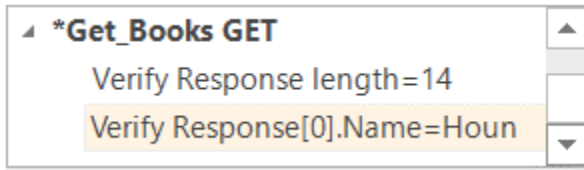
Choose the option 'Verify Response length=14'. This adds the following step to the recorded script:

```
Get_Session GET
*Get_Books GET
  Verify Response length=14
```

Now we want to verify the name of the first book returned. To do that, expand the "0" index entry and then right-click on the "Name" property returned:



Choose the option to **Verify Response[0].Name = Hound of the Baskervilles**. This will add a verification step for this specific property:

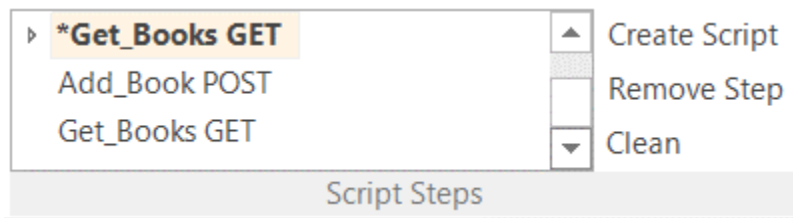


Now we add the last two requests - adding a book and verifying that it was added. To do that open up the two requests and click **Send** then **Record**:

- Add_Book (POST)
- Get_Book (GET)

The **Add_Book** won't actually work at this point because we've not populated the body, but it will be good enough to create the test script. For the second instance of Get_Books don't use the Verify option since we will want to code that by hand to match the book we actually added.

Once you are done, you should have:



Now click on the **Create Script** option and Rapise will generate the following code for you:

```
function Test()
{
    var
    LibraryInformationSystem_Get_Session=SeS('LibraryInformationSystem_Get_Session');
    LibraryInformationSystem_Get_Session.SetRequestHeaders([{"Name":"Accept", "Value":"application/json"}, {"Name":"Content-Type", "Value":"application/json"}]);
    LibraryInformationSystem_Get_Session.DoExecute();

    var
    LibraryInformationSystem_Get_Books=SeS('LibraryInformationSystem_Get_Books');
    LibraryInformationSystem_Get_Books.SetRequestHeaders([{"Name":"Accept", "Value":"application/json"}, {"Name":"Content-Type", "Value":"application/json"}]);
    LibraryInformationSystem_Get_Books.DoExecute();

    Tester.Assert('Compare call result for
http://www.libraryinformationsystem.org/services/restservice.svc/book?session_id={session_id}', LibraryInformationSystem_Get_Books.GetResponseBodyText(), "...");
    LibraryInformationSystem_Get_Books.DoVerify('LibraryInformationSystem_Get_Books Response', "length", 14);
    LibraryInformationSystem_Get_Books.DoVerify('LibraryInformationSystem_Get_Books Response', "[0].Name", "Hound of the Baskervilles");
    var
    LibraryInformationSystem_Add_Book=SeS('LibraryInformationSystem_Add_Book');
```

```

        LibraryInformationSystem_Add_Book.SetRequestHeaders ([{"Name": "Accept", "
Value": "application/json"}, {"Name": "Content-
Type", "Value": "application/json"}]);
        LibraryInformationSystem_Add_Book.DoExecute ();

        var
LibraryInformationSystem_Add_Book=SeS ('LibraryInformationSystem_Add_Book');
        LibraryInformationSystem_Add_Book.SetRequestHeaders ([{"Name": "Accept", "
Value": "application/json"}, {"Name": "Content-
Type", "Value": "application/json"}]);
        LibraryInformationSystem_Add_Book.DoExecute ();

        LibraryInformationSystem_Get_Books.SetRequestHeaders ([{"Name": "Accept",
"Value": "application/json"}, {"Name": "Content-
Type", "Value": "application/json"}]);
        LibraryInformationSystem_Get_Books.DoExecute ();
    }

```

If you click **Play** on this script as written, you will see that the tests to retrieve the books work correctly, but the test of adding a new book fails:

#	Type	Start	Name	Status	Comment	Iteration
	Message	17:52:55.623	Starting scenario: Test	Info		
	Assert	17:52:56.511	Get_Session.DoExecute(1)	Pass	Returned Value: true	0
	Assert	17:52:56.978	Get_Book.DoExecute(1)	Pass	Returned Value: true	0
	Assert	17:52:56.985	Compare call result for http://www.libraryinformationssystem.org/servi	Pass	[{"Author": "Age":125,"Id":3,"Name": "Arthur Conan	0
	Assert	17:52:57.210	Add_Book.DoExecute(1)	Fail	Returned Value: false	0
	Assert	17:52:57.793	Get_Book.DoExecute(1)	Pass	Returned Value: true	0
	Test	17:52:57.798	MyRestTest1	Fail	Passed:4 Failed:1	

Test Fail
Total: 7 Pass: 4 Fail: 2 Info: 1

This is as we'd expect since we've not populated the new book yet!

To make the template test script more useful, we should make the following changes:

- o Add comments to each of the sections to describe the purpose
- o Add code to get the **session ID** from the first call and pass to the subsequent calls
- o Create a JavaScript object to contain the new book information, and pass that to the **Add Book** function
- o Get the new book ID from the result of the **Add Book** function and use it later on.
- o Remove the check for the entire returned book array and just keep the check for the individual properties.

The complete updated test script looks like the following. We have highlighted the new/changed lines in yellow:

```

        //First get the session
        var
LibraryInformationSystem_Get_Session=SeS ('LibraryInformationSystem_Get_Sessio
n');
        LibraryInformationSystem_Get_Session.SetRequestHeaders ([{"Name": "Accept
", "Value": "application/json"}, {"Name": "Content-
Type", "Value": "application/json"}]);
        LibraryInformationSystem_Get_Session.DoExecute ();
        var sessionId =
LibraryInformationSystem_Get_Session.GetResponseBodyObject ();
        Tester.Message ('Session ID: ' + sessionId);

```

```

    //Get the list of books
    var
LibraryInformationSystem_Get_Books=SeS('LibraryInformationSystem_Get_Books');
    LibraryInformationSystem_Get_Book.SetRequestHeaders([{"Name":"Accept", "
Value":"application/json"}, {"Name":"Content-
Type", "Value":"application/json"}]);
    LibraryInformationSystem_Get_Books.DoExecute({"session_id": sessionId
});

    //Verify the data
    LibraryInformationSystem_Get_Books.DoVerify('LibraryInformationSystem_G
et_Books Response', "length", 14);
    LibraryInformationSystem_Get_Books.DoVerify('LibraryInformationSystem_G
et_Books Response', "[0].Name", "Hound of the Baskervilles");

    //Add a book
    var newBook = {
        Name: "A Christmas Carol",
        AuthorId: 2,
        GenreId: 3
    };

    var
LibraryInformationSystem_Add_Book=SeS('LibraryInformationSystem_Add_Book');
    LibraryInformationSystem_Add_Book.SetRequestHeaders([{"Name":"Accept", "
Value":"application/json"}, {"Name":"Content-
Type", "Value":"application/json"}]);
    LibraryInformationSystem_Add_Book.SetRequestBodyObject(newBook)
    LibraryInformationSystem_Add_Book.DoExecute({"session_id": sessionId
});

    //Get the ID of the new book
    newBook = LibraryInformationSystem_Add_Book.GetResponseBodyObject();
    Tester.Message("New Book ID: " + newBook.Id);

    //Verify the data
    LibraryInformationSystem_Get_Book.SetRequestHeaders([{"Name":"Accept", "
Value":"application/json"}, {"Name":"Content-
Type", "Value":"application/json"}]);
    LibraryInformationSystem_Get_Books.DoExecute({"session_id": sessionId
});
    LibraryInformationSystem_Get_Books.DoVerify('LibraryInformationSystem_G
et_Books Response', "length", 15);

```


1.6. Writing REST Test Scripts

Open up the main **MyRestTest1.js** file in the Rapise editor. It will initially consist of a single empty function `Test()`:

```
1 //##### Script Steps #####
2
3
4 function Test()
5 {
6 |
7 }
8
9 g_load_libraries=["Web Service"];
10
11
12
13
```

The first task is to get a new SessionId from the server using the **Get_Session** operation. To do this, drag the **"DoExecute"** operation from under the **"Get_Session"** object into the script editor, in between the opening and closing braces of the `Test()` function:


```
1 //##### Script Steps #####
2
3
4 function Test()
5 {
6 > SeS('LibraryInformationSystem_Get_Session').DoExecute(null);
7 |
8 }
9 g_load_libraries=["Web Service"];
10
11
12
13
```

This will execute the web serviced and return the SessionId. To actually access the retrieved value, you need to drag the **"GetResponseBodyObject"** property to the script editor, under the previous line. Then add the JavaScript code `var sessionId =` to actually store the value. We will also add a `Tester.Message(sessionId);` line afterwards to write out the value of the sessionId to the test report. This will help us make sure we are getting back a valid response from the web service. You should now have the following code:

```
function Test()
{
> SeS('LibraryInformationSystem_Get_Session').DoExecute(null);
> var sessionId = SeS('LibraryInformationSystem_Get_Session').GetResponseBodyObject();
> Tester.Message(sessionId);
}

g_load_libraries=["Web Service"];
```

Save this test and click "Play" to execute the test. You should now see a report similar to the following:

#	Name	Start	Type	Status	Comment	Iteration
	Starting scenario: Test	13:37:16.020	Message	Info		
	Get_Session.DoExecute([null])	13:37:17.486	Assert	Pass	Returned Value: true	0
	d51f97ea-d879-4eb1-b585-55469b88cef7	13:37:17.486	Message	Info		0
	MyRestTest1	13:37:17.486	Test	Pass	Passed:1 Failed:0	
 Test Pass Total:4 Pass:2 Fail:0 Info:2						

Now we need to add the code to get the list of books. To do that, simply drag the "DoExecute" operation from under the "Get_Books" object into the script editor. Then change the `(null)` argument to instead provide the session id as a Javascript dictionary:

```
SeS('LibraryInformationSystem_Get_Books').DoExecute({"session_id":sessionId});
```

To get the list of books as a JavaScript array, drag the "GetResponseBodyObject" property to the script editor, under the previous line. Then assign the value of this property to a variable such as "books":

```
var books = SeS('LibraryInformationSystem_Get_Books').GetResponseBodyObject();
```

Now we can add code to test that the number of books returned matches the expected value. Type in the following code:

```
Tester.AssertEqual('Book count matches', 14, books.length);
```

You should now have the following code:

```
function Test()
{
  > SeS('LibraryInformationSystem_Get_Session').DoExecute(null);
  > var sessionId = SeS('LibraryInformationSystem_Get_Session').GetResponseBodyObject();
  > Tester.Message(sessionId);
  > SeS('LibraryInformationSystem_Get_Books').DoExecute({"session_id":sessionId});
  > var books = SeS('LibraryInformationSystem_Get_Books').GetResponseBodyObject();
  > Tester.AssertEqual('Book count matches', 14, books.length);
}

g_load_libraries=["Web Service"];
```

Finally we need to add the code to add a new book to the system. To do that, simply drag the "DoExecute" operation from under the "Add_Book" object into the script editor. Then change the `(null)` argument to instead provide the session id as a Javascript dictionary:

```
SeS('LibraryInformationSystem_Add_Book').DoExecute({"session_id":sessionId});
```

To provide the data for a new book, we will need to drag the "SetRequestBodyObject" property of the "Add_Book" object to the line **above** the DoExecute and pass in a populated JavaScript object:

```
var newBook = {};
newBook.Name = 'A Christmas Carol';
newBook.AuthorId = 2;
newBook.GenreId = 3;

SeS('LibraryInformationSystem_Add_Book').SetRequestBodyObject(newBook);
```

Finally Add code to test that our new book was added correctly and the count has increased by one:

```

        SeS('LibraryInformationSystem_Get_Books').DoExecute({"session_id":sessionId});
        books =
        SeS('LibraryInformationSystem_Get_Books').GetResponseBodyObject();
        Tester.AssertEqual('Book count matches', 15,
        books.length);

```

You should now have the following code:

```

function Test()
{
    >> SeS('LibraryInformationSystem_Get_Session').DoExecute(null);
    >> var sessionId = SeS('LibraryInformationSystem_Get_Session').GetResponseBodyObject();
    >> Tester.Message(sessionId);
    >>
    >> SeS('LibraryInformationSystem_Get_Books').DoExecute({"session_id":sessionId});
    >> var books = SeS('LibraryInformationSystem_Get_Books').GetResponseBodyObject();
    >> Tester.AssertEqual('Book count matches', 14, books.length);
    >>
    >> var newBook = {};
    >> newBook.Name = 'A Christmas Carol';
    >> newBook.AuthorId = 2;
    >> newBook.GenreId = 3;
    >> SeS('LibraryInformationSystem_Add_Book').SetRequestBodyObject(newBook);
    >> SeS('LibraryInformationSystem_Add_Book').DoExecute({"session_id":sessionId});
    >>
    >> SeS('LibraryInformationSystem_Get_Books').DoExecute({"session_id":sessionId});
    >> books = SeS('LibraryInformationSystem_Get_Books').GetResponseBodyObject();
    >> Tester.AssertEqual('Book count matches', 15, books.length);
}

```

Save this test and click "Play" to execute the test. You should now see a report similar to the following:

#	Name	Start	Type	Status	Comment	Iteration
	Starting scenario: Test	14:49:03.725	Message	Info		
	Get_Session.DoExecute([null])	14:49:04.334	Assert	Pass	Returned Value: true	0
	c3d8dcd4-6125-427d-939a-0dd181b3cce1	14:49:04.334	Message	Info		0
	Get_Books.DoExecute(["session_id":"c3d8dcd4-6125-427d-939a-0dd181b3cce1"])	14:49:05.051	Assert	Pass	Returned Value: true	0
	Book count matches	14:49:05.051	Assert	Pass		0
	Add_Book.DoExecute(["session_id":"c3d8dcd4-6125-427d-939a-0dd181b3cce1", "name":"A Christmas Carol", "authorId":2, "genreId":3])	14:49:05.379	Assert	Pass	Returned Value: true	0
	Get_Books.DoExecute(["session_id":"c3d8dcd4-6125-427d-939a-0dd181b3cce1"])	14:49:05.597	Assert	Pass	Returned Value: true	0
	Book count matches	14:49:05.597	Assert	Pass		0
	MyRestTest1	14:49:05.597	Test	Pass	Passed:6 Failed:0	

Test Pass
 Total:9 Pass:7 Fail:0 Info:2

Congratulations! You have just created your first test script that tests a RESTful web service.

2. Testing SOAP Web Services

In this section you shall learn how to test a SOAP web services API using Rapise. We shall be using a demo application called **Library Information System** that has a dummy SOAP web service API available for learning purposes. You can access this sample application at <http://www.libraryinformationsystem.org>, and its SOAP web service API can be found at:

www.libraryinformationsystem.org/Services/SoapService.aspx

2.1. What is SOAP and what is a SOAP web service?

SOAP is the Simple Object Access Protocol, and allows you to make API calls over HTTP/HTTPS using specially formatted XML. SOAP web services make use of the Web Service Definition Language (WSDL) and communicate using HTTP POST requests. They are essentially a serialization of RPC object calls into XML that can then be passed to the web service. The XML passed to the SOAP web services needs to match the format specified in the WSDL.

SOAP web services are fully self-describing, so most clients do not directly work with the SOAP XML language, but instead use a client-side proxy generator that creates client object representations of the web service (e.g. Java, .NET objects). The web service consumers interact with these language-specific representations of the SOAP web service. However when these SOAP calls fail you need a way of testing them that includes being able to inspect the raw SOAP XML that is actually being sent.

2.2. Overview

Creating a SOAP web service test in Rapise consists of the following steps:

1. Using the SOAP web services studio to inspect the SOAP WSDL
2. Invoke the various SOAP operations and verify that they return the expected data in the expected format.
3. Generating the test script in JavaScript that uses the learned Rapise web service objects based on the WSDL.

We shall discuss each of these steps in turn.

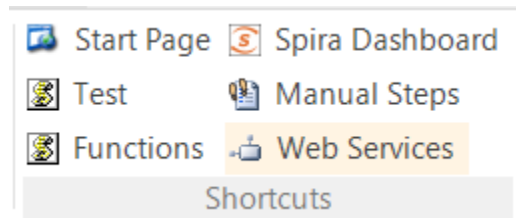
2.3. Inspecting the SOAP WSDL Endpoint

Create a new test in Rapise called MySoapTest1.sstest.

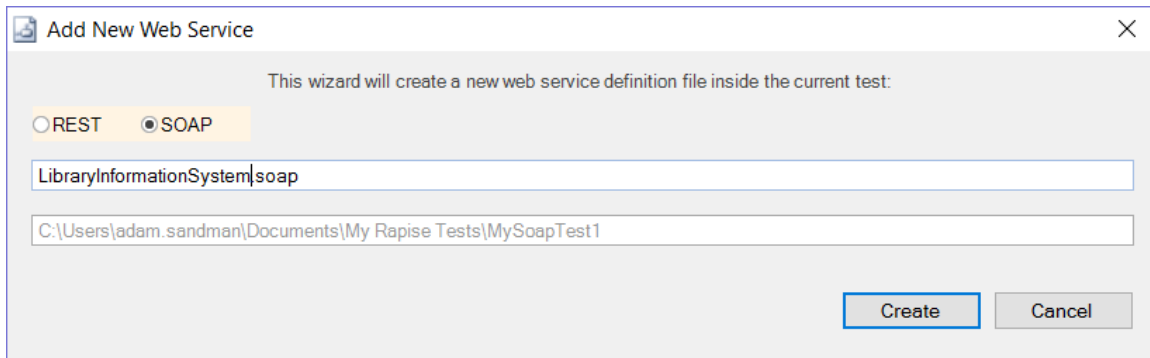
- For **Methodology**, choose **Basic: Windows Desktop Application** and Rapise will create a new blank test project. If you plan on using a combination of Web or Mobile UI tests in the same script, you could choose one of the other types.
- For **Scripting Language**, choose **JavaScript**. The scriptless Rapise Visual Language (RVL) can be used with web service tests, but it means that all the web service tests need to be in a JavaScript subroutine / scenario that is called from the RVL test.

Rapise will create a new blank test project.

Once you have created it, click on the "Web Services" icon in the Test ribbon to add a new web service definition to your test project:



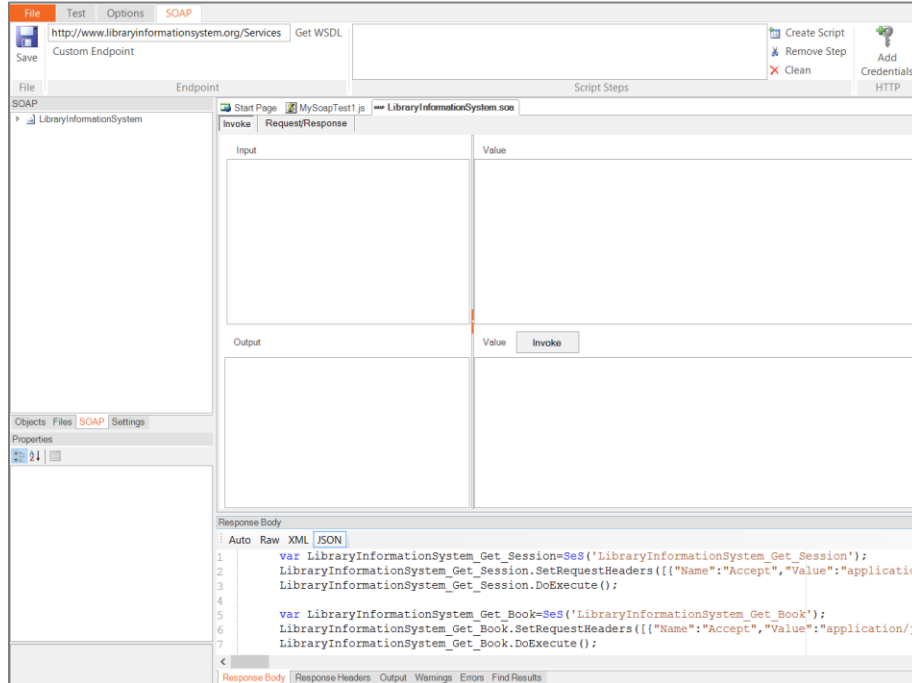
This will display the Add New Web Service dialog box:



Choose **SOAP** as the type of web service you want to create.

Then, enter the name of the web service that you're going to add, in this case enter "**LibraryInformationSystem.soap**" and click "Create".

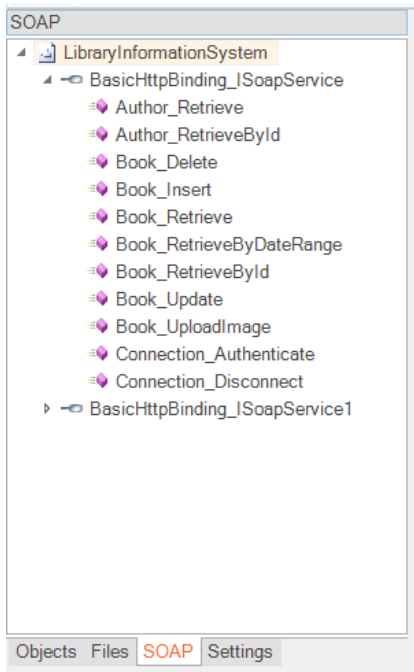
This will add the SOAP web services definition file to your test project:



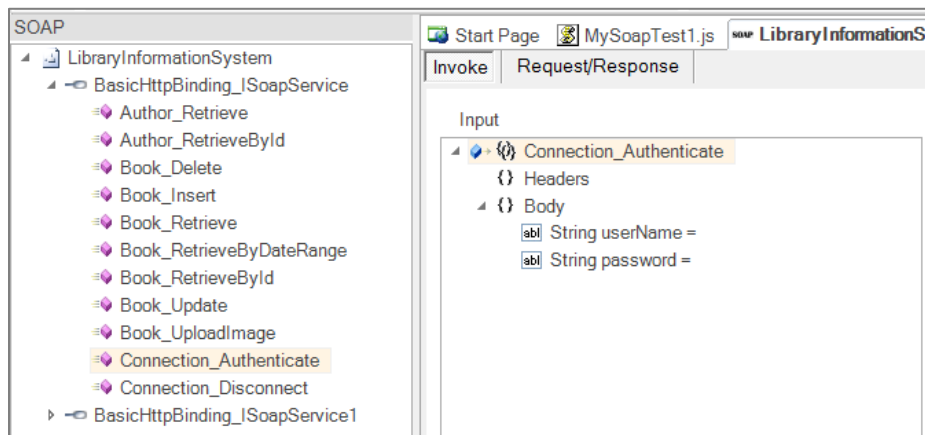
In the **Endpoint** section of the SOAP ribbon, enter the following URL to the sample application's WSDL file:

- o `http://www.libraryinformationsystem.org/Services/SoapService.svc?wsdl`

then click the **Get WSDL** to load the list of SOAP operations:



Now click on the **Connection_Authenticate** operation in the SOAP explorer:



This is the first operation we will need to invoke since it is used to authenticate with the online library system before calling the other functions.

You can click on each of the different SOAP operations (e.g. for inserting, retrieving, deleting or updating a book) and the SOAP studio will display the expected input and output parameters as well as any headers.

In the next section we shall be performing the following actions:

- Authenticating as a specific user
- Viewing the list of books
- Inserting a new book
- Viewing the updated list of books
- Disconnecting

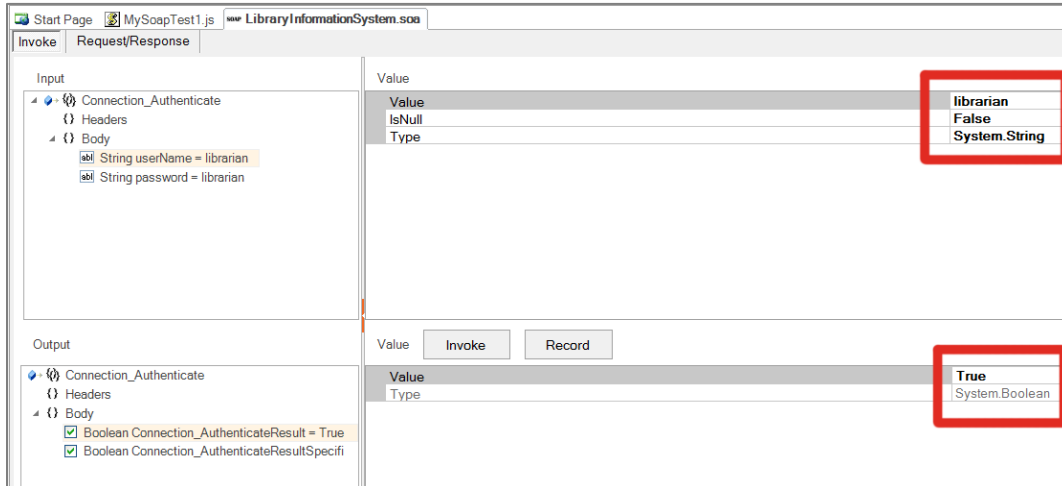
Each one will involve calling a specific SOAP operation with some input parameters, viewing the data returned and adding a verification step if appropriate.

2.4. Invoking the SOAP Actions

Starting with the **Connection_Authenticate** operation that we had selected, click on the two Input parameters in turn and enter values:

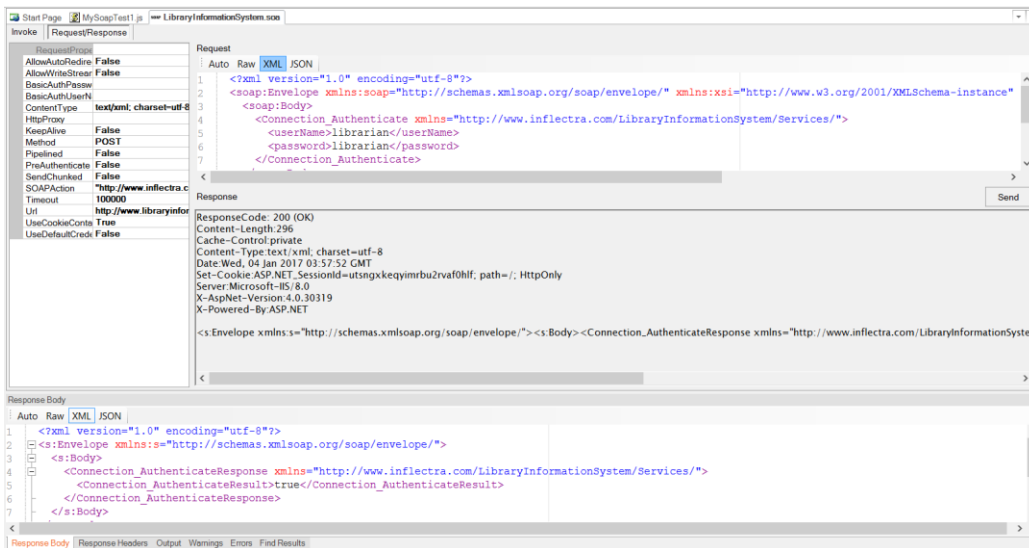
- o userName = librarian
- o password = librarian

Then click the **Invoke** button underneath:



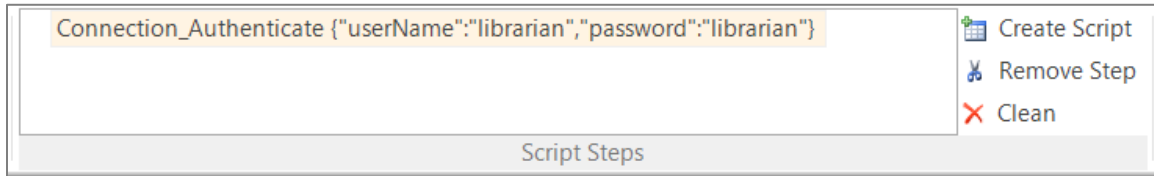
You can see that the response to our Invoked operation as a simple boolean value of **True** returned. That indicated that we authenticated correctly. If you try putting in an incorrect login/password, you'll get back **False** instead.

If you have a SOAP web service that doesn't behave as expected, you may want to view the raw SOAP XML that is being sent to/from the web service. To view this, click on the **Request/Response** tab of the SOAP studio editor and the following will be displayed:

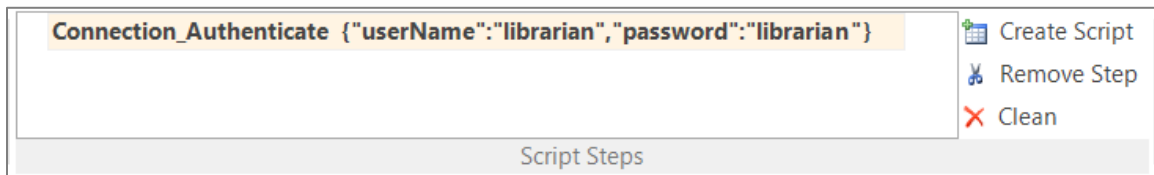


This view lets you see the Request and Response HTTP headers body, with the body displayed in a friendly, easy to read color-coded XML format. That way you can easily invoke the SOAP operations using the Rapise SOAP studio GUI and view the raw SOAP XML being sent to/from the server. This is invaluable when debugging a failing SOAP web service.

In the case of our test of **Connection_Authenticate**, we can now click the **Record** button (next to Send) to add this operation to our list of recorded test steps:

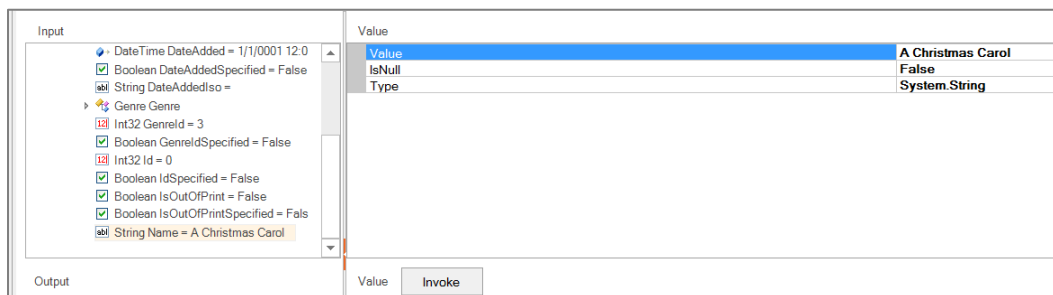


Once you have added the operation to the list of recorded steps, you can go one step further and ask Rapise to verify the data returned. To do that, click on the **Verify** button that is displayed next to the **Record** button. The step will now switch to **bold** to indicate that a verification step is also included.



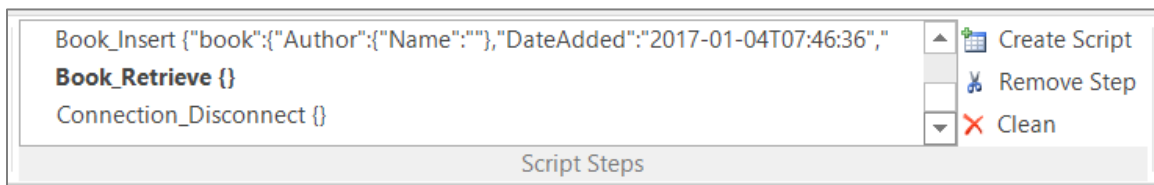
Now we need to repeat this process for the following additional operations:

- **Book_Retrieve**
 - No Input Parameters
 - Press **Invoke** to test the retrieve
 - Press **Record** to record the test script
 - Click **Verify** to add a verification step
- **Book_Insert**
 - Populate the **Book** input object with these values:
 - AuthorId = 2
 - GenreId = 3
 - Name = 'A Christmas Carol'
 - DateAdded = (pick a date using the date picker)
 - DateAddedIso = 2017-01-04T07:46:36
 - Press **Invoke** to test the insert
 - Press **Record** to record the test script



- **Book_Retrieve**
 - No Input Parameters
 - Press **Invoke** to test the retrieve
 - Press **Record** to record the test script
 - Click **Verify** to add a verification step
- **Connection_Disconnect**
 - No Input Parameters
 - Press **Invoke** to test the retrieve
 - Press **Record** to record the test script

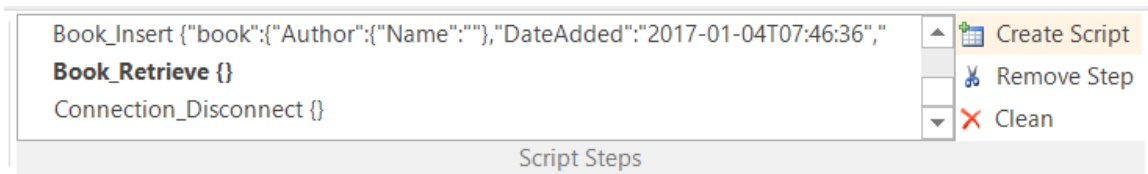
Once you have completed all these steps, you will see the following recorded in the **Script Steps** box:



Now that we have recorded the operations and verifications, we can proceed to generate the test script in Rapise that will regression test the web service.

2.5. Generating the Rapise Test Script

In the SOAP ribbon, click on the **Create Script** button to generate the initial test script:



Click on the **Test** shortcut in the main test ribbon, and Rapise will display the **MySoapTest.js** file.

In the main Rapise test script file, you will see the following generated:

```
function Test ()
{
    var LibraryInformationSystem=SeS ('LibraryInformationSystem');
    LibraryInformationSystem.DoExecute ('Connection_Authenticate',
{"userName":"librarian","password":"librarian"});
    Tester.Assert ('Connection_Authenticate Response',
LibraryInformationSystem.GetResponseObject (),
{"Body":{"Connection_AuthenticateResult":true,"Connection_AuthenticateResults
pecified":true},"Headers":{}});
    LibraryInformationSystem.DoExecute ('Book_Retrieve', {});
    Tester.Assert ('Book_Retrieve Response',
LibraryInformationSystem.GetResponseObject (), {...}],"Headers":{}});
    LibraryInformationSystem.DoExecute ('Book_Insert',
{"book":{"Author":{"Name":"","DateAdded":"2017-01-
04T07:46:36","DateAddedSpecified":true,"DateAddedIso":"2017-01-
04T07:46:36","Genre":{"Name":""},"Name":"A Christmas Carol"}});
```

```

    LibraryInformationSystem.DoExecute('Book_Retrieve', {});
    Tester.Assert('Book_Retrieve Response',
LibraryInformationSystem.GetResponseObject(), {...}, "Headers":{}));
    LibraryInformationSystem.DoExecute('Connection_Disconnect', {});
}

```

You will see each of the SOAP functions called in turn, with verification code automatically added.

We can add some comments to make it easier to read:

```

//Authenticate
var LibraryInformationSystem=SeS('LibraryInformationSystem');
LibraryInformationSystem.DoExecute('Connection_Authenticate',
{"userName":"librarian","password":"librarian"});
Tester.Assert('Connection_Authenticate Response',
LibraryInformationSystem.GetResponseObject(),
{"Body":{"Connection_AuthenticateResult":true,"Connection_AuthenticateResults
pecified":true},"Headers":{}});

//Verify the initial list of books
LibraryInformationSystem.DoExecute('Book_Retrieve', {});
Tester.Assert('Book_Retrieve Response',
LibraryInformationSystem.GetResponseObject(), {...}],"Headers":{}));
LibraryInformationSystem.DoExecute('Book_Insert',
{"book":{"Author":{"Name":""},"DateAdded":"2017-01-
04T07:46:36","DateAddedSpecified":true,"DateAddedIso":"2017-01-
04T07:46:36","Genre":{"Name":""},"Name":"A Christmas Carol"}});

//Verify the updated list of books and disconnect
LibraryInformationSystem.DoExecute('Book_Retrieve', {});
Tester.Assert('Book_Retrieve Response',
LibraryInformationSystem.GetResponseObject(), {...}, "Headers":{}));
LibraryInformationSystem.DoExecute('Connection_Disconnect', {});

```

When you click the **Play** button in the main test ribbon, you will see the following result:

#	Type	Start	Name	Status	Comment
	Message	08:46:21.701	Starting scenario: Test	Info	
	Assert	08:46:22.539	LibraryInformationSystem.DoExecute(["Connection_Authenticate",{"us	Pass	Returned Value: true
	Assert	08:46:22.546	Connection_Authenticate Response	Pass	
	Assert	08:46:22.918	LibraryInformationSystem.DoExecute(["Book_Retrieve",{}])	Pass	Returned Value: true
	Assert	08:46:22.926	Book_Retrieve Response	Pass	
	Assert	08:46:23.135	LibraryInformationSystem.DoExecute(["Book_Insert",{"book":{"Author	Pass	Returned Value: true
	Assert	08:46:23.353	LibraryInformationSystem.DoExecute(["Book_Retrieve",{}])	Pass	Returned Value: true
	Assert	08:46:23.358	Book_Retrieve Response	Pass	
	Assert	08:46:23.556	LibraryInformationSystem.DoExecute(["Connection_Disconnect",{}])	Pass	Returned Value: true
	Test	08:46:23.561	MySoapTest1	Pass	Passed:8 Failed:0

Test Pass
Total:10 Pass:9 Fail:0 Info:1

Congratulations! You have recorded and executed a SOAP web service test.

Legal Notices

This publication is provided as is without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

This publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information contained herein; these changes will be incorporated in new editions of the publication. Inflectra Corporation may make improvements and/or changes in the product(s) and/or program(s) and/or service(s) described in this publication at any time.

The sections in this guide that discuss internet web security are provided as suggestions and guidelines. Internet security is constantly evolving field, and our suggestions are no substitute for an up-to-date understanding of the vulnerabilities inherent in deploying internet or web applications, and Inflectra cannot be held liable for any losses due to breaches of security, compromise of data or other cyber-attacks that may result from following our recommendations.

SpiraTest®, SpiraPlan®, SpiraTeam®, Rapise® and Inflectra® are either trademarks or registered trademarks of Inflectra Corporation in the United States of America and other countries. Microsoft®, Windows®, Explorer® and Microsoft Project® are registered trademarks of Microsoft Corporation. All other trademarks and product names are property of their respective holders.

Please send comments and questions to:

Technical Publications

Inflectra Corporation

8121 Georgia Ave, Suite 504

Silver Spring, MD 20910-4957

U.S.A.

support@inflectra.com