



**Rapise® User Guide**

Version 1.7

Inflectra Corporation

Date: June 21st, 2013



## About this Guide

[Top](#) [Previous](#) [Next](#)

The Rapise User's Guide is divided into four sections: Getting Started; Features; Dialogs, Views, and Menus; HowTos.

### Getting Started

The **Getting Started** section is for new Rapise users. It has the following subsections:

- (1) An [Overview](#) of Rapise: what it's for and how to use it.
- (2) [Test Samples](#), where the sample projects included with Rapise are described.
- (3) [TwoDialogs Sample](#), a step-by-step tutorial for creating your first test with Rapise
- (4) [Tutorial: Record and Playback](#), a slightly more advanced tutorial in using Rapise to test a web page.
- (5) [Tutorial: Testing REST Web Services](#), a tutorial in using Rapise to test a RESTful web service API.

### Features

The features of Rapise are many. Features have been designed to make all aspects of test automation as easy as possible.

Most of the features of Rapise fall into one of five categories:

- (1) Building test scripts with little or no manual scripting.
- (2) Reading and interpreting results and reports.
- (3) Additional features and capabilities for sophisticated testing.
- (4) Writing more involved or complicated tests using scripting.
- (5) Extending Rapise to learn new or extended libraries of capabilities.

Depending on the application set being tested, not all of these features are necessarily needed for every situation.

For each feature, this document attempts to present:

- (1) The reason you might use a given feature.
- (2) A summary of the basic value of the feature.
- (3) An overview of how the feature works from the perspective of using it.
- (4) At least one useful sample that demonstrates how to use the feature.

### Dialogs, Views, and Menus

This section details the Rapise GUI. Each subsection describes the function of a particular Dialog, View, or Menu. The purpose and consequences of all buttons, options, lists, and check boxes are listed.

### HowTos

This section focuses on specific tasks that a Rapise user might want to accomplish.

## Glossary

[Top](#) [Previous](#) [Next](#)

This glossary presents a list of terms and their definitions as they are used in this guide.

API - Application Programming Interface  
AUT - Application Under Test  
DOM - Document Object Model  
GUI - Graphical User Interface  
GWT - Google Web Toolkit  
IDE - Integrated Development Environment  
JSON - JavaScript Object Notation  
REST - REpresentation State Transfer  
SOAP - Simple Object Access Protocol  
UI - User Interface  
XML - eXtensible Markup Language  
YUI - Yahoo! User Interface (library)

## Getting Started

[Top](#) [Previous](#) [Next](#)

The **Getting Started** section is for new Rapise users. It has the following subsections:

- (1) An [Overview](#) of Rapise: what it's for and how to use it.
- (2) [Test Samples](#), where the sample projects included with Rapise are described.
- (3) [TwoDialogs Sample](#), a step-by-step tutorial for creating your first test with Rapise
- (4) [Tutorial: Record and Playback](#), a slightly more advanced tutorial in using Rapise to test a web page.
- (5) [Tutorial: Testing Adobe Flex Applications](#), a slightly more advanced tutorial in using Rapise to test an Adobe Flex/Flash/AIR application.
- (6) [Tutorial: Testing REST Web Services](#), a tutorial in using Rapise to test a RESTful web service API.

## Overview

[Top](#) [Previous](#) [Next](#)

Rapise was created to make software testing easy and manageable without being prohibitively expensive.

Rapise was made easy for software test professionals, developers and professionals concerned with quality assurance to simply and quickly write a test to cover an application, a web page, or a single bug to prevent regression.

Consider for a moment what it is you do to test your software today. Most likely it has some form of user interface (UI), probably a graphic user interface (GUI). So you will run the application, click around, perhaps in some way that gives you complete coverage of all the features (but probably not if it's a large application or web). Then you will login, if appropriate, and you will fetch some data and modify some data, test some more controls - edit boxes, buttons, drop-down lists, links, etc. If you have just fixed a bug then you will focus on the area of the application where the bug occurred. You will enter data that causes the bug, or go through the control sequence that causes the bug.

Next time you come to fix a bug in this application, you will do the same thing again.. Once again, you will focus on the area where the bug was.

Rapise presents you with two methods for capturing specific tests, and it will keep the test as a solo test or as part of a suite of tests that help you to qualify the application for release or a more formal QA process. Rapise is designed to allow the developer or the test professional to add new tests quickly and so to build up a library of tests.

There are two methods for capturing tests:

- **Record** and **playback**. In this type of test creation, you turn on the recorder and perform the actions needed to execute the test. Each test is saved to its own directory. A test consists of a javascript [test script](#) (.js), a meta-data file (\*.sstest), and any number of additional supplementary scripts and data files. The test script is automatically generated after recording; simple modifications are required to make the test [data driven](#). [Checkpoints](#) can be added during recording, or manually into the script.
- **Object-driven learning**. Rapise considers each item on the page or within the application window to be an object. Examples are buttons, edit boxes, links, etc. To create a test using this technique, you have Rapise "[learn](#)" each control, and it will keep a miniature [database](#) of all the controls you "teach" it. To create a test, you write a script to instruct Rapise to perform a particular action on each object in the prescribed order. As any point along the way, the script you write can instruct Rapise to look inside an object and read its data and compare that value or content with what you expect it to be.

There are many methodologies with their own recommended approaches for designing testing strategies to ensure that application coverage is complete and meets the business requirements specification of the work being accomplished. Inflectra in general, recommends that you [create a new test](#) for each software requirement (to track progress) and for each issue in your issue tracking system (to test for regressions).

To help you manage the requirements and issue tracking processes and to ensure that you have adequate **test coverage**, Inflectra recommend that you use Rapise with a test management system such as [SpiraTest](#). That way you can maintain all your requirements, test cases and defects in a single place.

Once you have created the test, you can [playback](#) your test from within Rapise, run it from the [command-line](#) or execute it remotely using [RapiseLauncher](#) in conjunction with [SpiraTest](#). A [report](#) detailing the outcome of each step of the test will be automatically generated.

[Recording](#), [playback](#), the [report](#), and the Rapise [engine](#) are all customizable.

## Samples Index

[Top](#) [Previous](#) [Next](#)

Rapise includes several sample tests that you are free to read, modify, copy and use. They are located in: *RapiseDataDirectory\Samples*. Unless you specified otherwise, the *RapiseDataDirectory* will be:

C:\Users\Public\Documents\Rapise.

The sample tests are described below.

### AdobeFlex3

This is a set of regression tests for [Adobe Flex](#) 3.x controls.

### AdobeFlex4

This is a set of regression tests for [Adobe Flex](#) 4.x controls.

### AnalogRecorder

This sample demonstrates [Analog Recording](#).

### CrossBrowser

A simple recorded test is modified to run on both Firefox and Internet Explorer. Read the [Cross Browser Testing](#) HowTo for more details.

### DotNet20

This sample tests a .NET 2.0 application. This sample demonstrates the capabilities of the **.NET** library. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, menu, etc.

### Java

This sample tests a Java AWT/SWING application. This sample demonstrates the capabilities of the **Java** library. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, menu, etc.

### Java SWT

This sample tests a Java SWT/RCP application. This sample demonstrates the capabilities of the **SWT** and **UIAutomation** libraries. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, menu, etc.

### Library Information System

These tests can be used to test the sample library information system web application hosted at <http://www.libraryinformationsystem.net>. This is the same sample application used by [SpiraTest](#) and illustrates how you can use [SpiraTest](#) to manage and execute automated Rapise tests. A copy of these tests is also available in new installations of [SpiraTest](#) v3.2+.

### Managed

This sample tests a .NET 2.0 application. This sample demonstrates the capabilities of the **Managed** library. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, grid, listbox, listview, menu, etc.

### QtFramework

This sample tests a sample **QT Framework** cross-platform application. This sample demonstrates the capabilities of the **QtFramework** library. The application under test contains various standard Qt widgets, such as: button, edit, tree, combo box, etc.

### Silverlight

This sample tests a Silverlight web application. This sample demonstrates the capabilities of the **UIAutomation** library. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, menu, etc.

### SimulatedObject

This sample opens **MS Paint** and draws on its canvas. It uses [Simulated Objects](#) and several related methods: **DoMouseMove(X,Y)**, **DoLButtonDown()**, **DoLButtonUp()**, and **DoSendKeys(text)**.

### SampleATM

This sample tests an **MFC** application. You will also learn how to organize your test script in modular form, how to launch the AUT from your test script and how to execute various actions on GUI

controls.

### UsingCustomStrings

This sample demonstrates how to integrate Rapise tests with other tools using [Custom Strings](#). **TestFinish()** is used to analyze and save test results. For more details, see: [Custom Strings](#).

### UsingDLLHandlerManaged

This sample shows how to [unit test managed DLLs](#). You'll see how to use methods **CreateClassInstance()** and **InvokeMember()**.

### UsingDLLHandlerUnManaged

This sample shows how to [unit test unmanaged DLL code](#). You'll learn how to register (**UserWrap.Register**) and execute (**UserWrap.ShellExecute**) a function.

### UsingImageCheckPoint

This example shows how to create image [checkpoints](#).

### UsingInclude

This sample demonstrates two ways to include external files/functions:

1. **eval(g\_helper.Include(...))**: include a file with utility functions.
2. **SeSRunJScript(...)**: include and execute external function with its own object map.

### UsingMSAccess, UsingMSExcel, UsingMSWord

These samples demonstrate how you can work with **Microsoft Word, Excel, and Access** from scripts. You'll learn how to test applications that expose a [COM interface](#).

### UsingRegistry

This sample demonstrates usage of the windows registry. The registry is queried to determine the OS (XP/2003/Vista, etc) and owner.

### UsingReporting

This sample illustrates various [reporting](#) features:

1. Regular reporting (**Tester.Assert**, **Tester.Message**)
2. Custom attributes (**Tester.SetReportAttribute**, **Tester.ResetReportAttribute**)
3. Stacked attributes (**Tester.PushReportAttribute**, **Tester.PopReportAttribute**)
4. Nested Tests (**Tester.BeginTest**, **Tester.EndTest**)
5. Inserting Links, Text and Images into the report (tags parameter, **SeSReportText**, **SeSReportLink**, **SeSReportImage**)

### UsingSpreadSheet

This example shows how you can use **Excel** spreadsheets to create [Data-Driven](#) tests. This script reads test case data from an XLS spreadsheet to test **Calculator**.

### UsingXML

This sample demonstrates how to read, modify and write XML files.

### WebServicesREST

This sample demonstrates how you can use the Rapise [Web-Services](#) module to write and execute automated web service tests against an HTTP [RESTful web service](#).

### Wpf

This sample tests a Windows Presentation Foundation (WPF) application. This sample demonstrates the capabilities of the **UIAutomation** library. The application under test contains various standard GUI controls, such as: button, edit, tree, combo box, menu, etc.

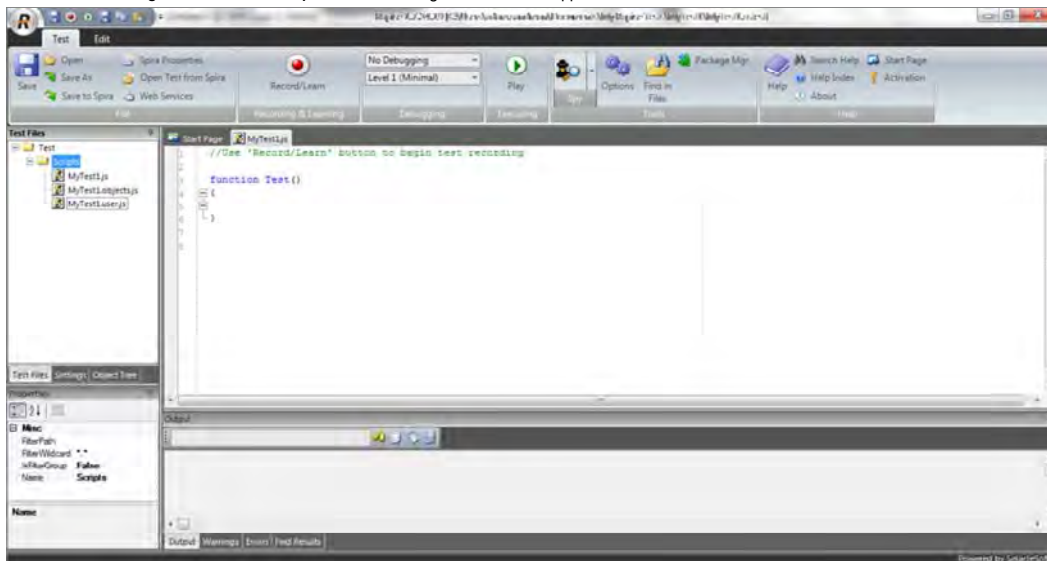
## Tutorial: Record and Playback

[Top](#) [Previous](#) [Next](#)

In this section, you will learn how to record and execute a Rapise script against a web application. We will be using a demo application called **Library Information System**. Our test will be simple. It will log on to the library catalog, navigate to the main menu, and click on all of the menu options to make sure the links are working.

### 1. Open Rapise

Go to **Start > All Programs > Inflectra > Rapise**. The following window should appear.



2. Open the AUT (Application Under Test)

Open up Internet Explorer. You will find it in **Start > All Programs > Internet Explorer**. In Internet Explorer, navigate to: <http://www.libraryinformationsystem.org>:

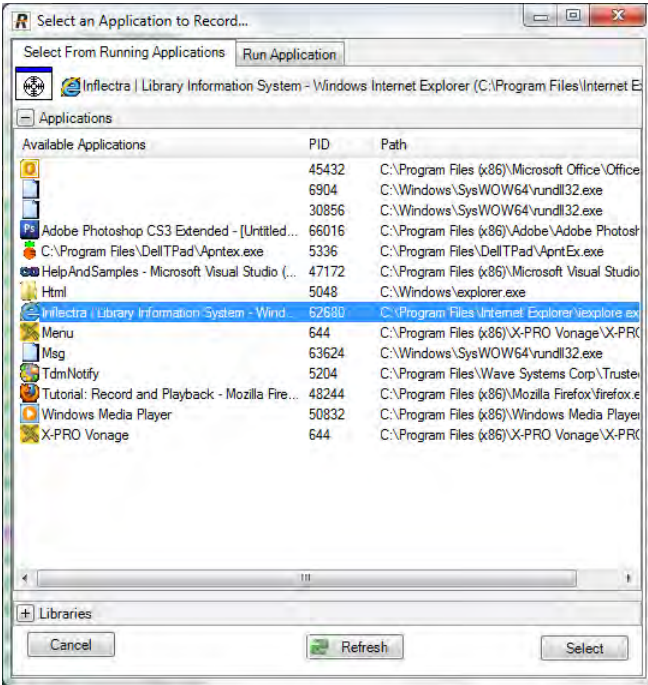


3. The Select an Application to Record Dialog

In the Rapise window, press the **Record/Learn** button on the Ribbon.



The Select an Application to Record... Dialog (SAR dialog) will open.



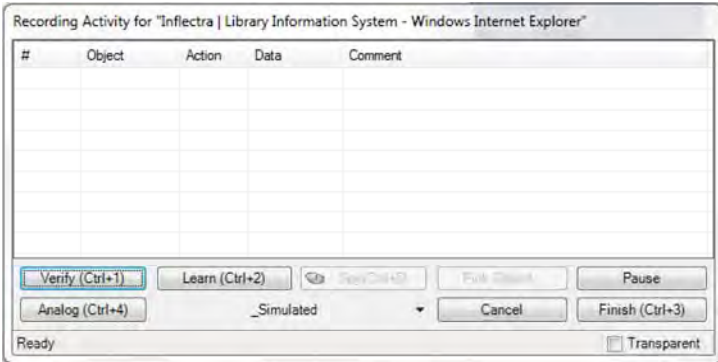
There are two sections to the **SAR dialog**. In the bottom section, you select which Rapise library will be used during the recording session. Because we will be recording our interactions with Internet Explorer, make sure that the **Internet Explorer HTML** library is checked. No other libraries should be selected. See below:

Library	Description
<input type="checkbox"/> Auto	Detect library automatically
<input type="checkbox"/> .NET	.NET 1.1, 2.0, 3.0, 3.5 with Accessibility
<input checked="" type="checkbox"/> Internet Explorer HTML	HTML DDM-based recorder for Internet Explorer
<input type="checkbox"/> Firefox HTML	HTML DDM-based recorder for Mozilla Firefox
<input type="checkbox"/> Generic	Generic library contains basic definitions for most commo...

In the top section of the SAR dialog, we choose which application to record. Scroll down the available applications and click once on SampleATM, so that it is highlighted. Now, press the **Select** button near the bottom right of the dialog.

Available Applications	PID	Path
	45432	C:\Program Files (x86)\Microsoft Office\Office
	6904	C:\Windows\SysWOW64\rundll32.exe
	30856	C:\Windows\SysWOW64\rundll32.exe
	66016	C:\Program Files (x86)\Adobe\Adobe Photosh
	5336	C:\Program Files\DellTPad\AprntEx.exe
	47172	C:\Program Files (x86)\Microsoft Visual Studio
	5048	C:\Windows\explorer.exe
	62680	C:\Program Files\Internet Explorer\explore.ex
	644	C:\Program Files (x86)\X-PRO Vonage\X-PRC
	63624	C:\Windows\SysWOW64\rundll32.exe
	5204	C:\Program Files\Wave Systems Corp\Truste
	48244	C:\Program Files (x86)\Mozilla Firefox\firefox.e
	50832	C:\Program Files (x86)\Windows Media Playe

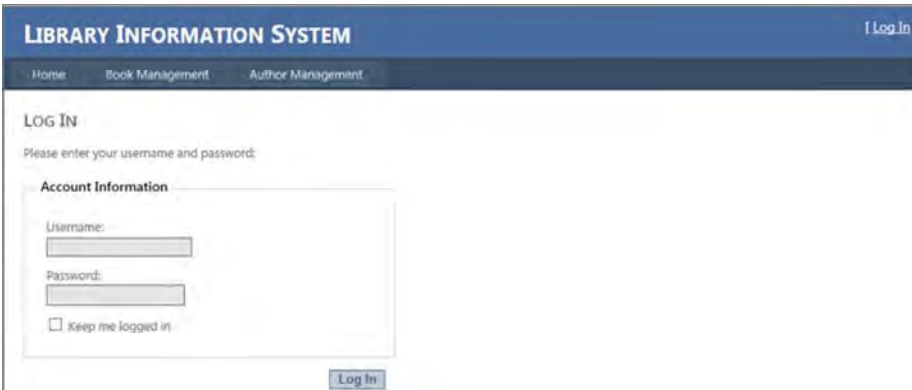
The [Recording Activity Dialog](#) (RA dialog) will appear:



The **RA dialog** has a grid. As you interact with the sample Library Information System program, the grid will automatically populate with your actions.

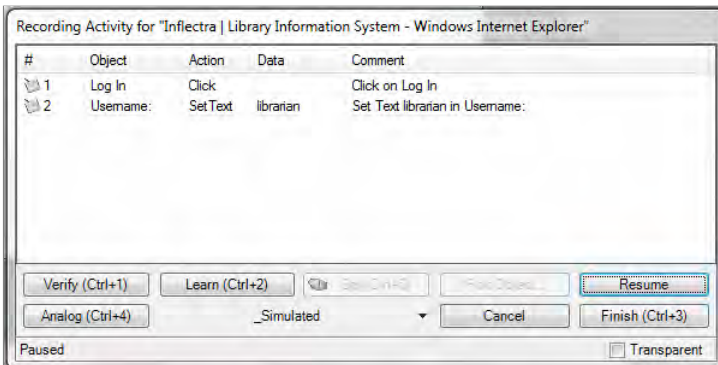
#### 4. Recording

Let's begin creating the test. On the library information system login page, click on the **Log In** link in the top-right of the screen.



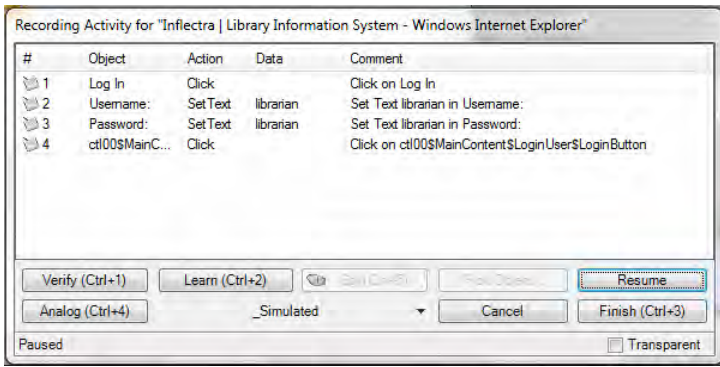
In the username text box, type *librarian*

Press the tab key. You'll notice that the **RA dialog** has changed. Your actions, clicking Log-In and entering a username, are listed in the grid:



The password for user *librarian* is also *librarian*. Type the password in and then press the **Log-In** button.

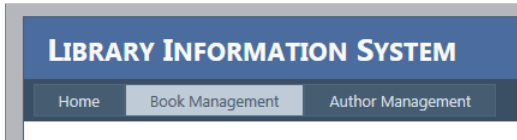
Two more rows should appear in the **RA dialog**: one to represent the password entry, and one to represent the button click:



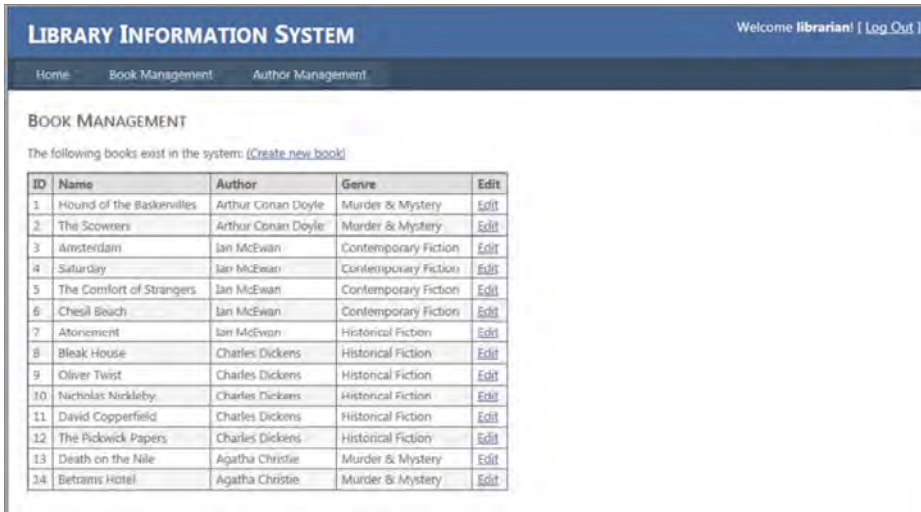
You should now be on the main menu of the Library Information System with the user's name listed in the top-right:



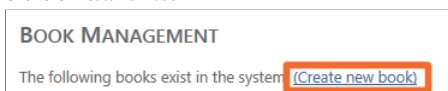
Click the **Book Management** button. It is highlighted in the next screenshot:



You should now be on the Book Management page (see the below image). Click the **Home** button to go back to the main menu.



Click the **Create new book** link:



You should now be on the Create New Book page (see image below). Click the **HOME** button to go back to the main menu.

**LIBRARY INFORMATION SYSTEM** Welcome librarian! [ Log Out ]

Home Book Management Author Management

### CREATE NEW BOOK

Please enter the book information and click Insert:

**Book Information**

Name:

Author:

Genre:

Now, click the **Author Management** button:

**LIBRARY INFORMATION SYSTEM**

Home Book Management **Author Management**

You should now be on the Author Management page (see image below):

**LIBRARY INFORMATION SYSTEM** Welcome librarian! [ Log Out ]

Home Book Management Author Management

### AUTHOR MANAGEMENT

The following authors exist in the system: [[Create new author](#)]

ID	Name	Age	Edit
1	Ian McEwan	42	<a href="#">Edit</a>
2	Charles Dickens	105	<a href="#">Edit</a>
3	Arthur Conan Doyle	125	<a href="#">Edit</a>
4	Agatha Christie	98	<a href="#">Edit</a>

Click the **Create New Author** link:

**AUTHOR MANAGEMENT**

The following authors exist in the system: [[Create new author](#)]

ID	Name	Age	Edit
1	Ian McEwan	42	<a href="#">Edit</a>
2	Charles Dickens	105	<a href="#">Edit</a>
3	Arthur Conan Doyle	125	<a href="#">Edit</a>
4	Agatha Christie	98	<a href="#">Edit</a>

You should now be on the Create New Author page (see below). Click the **Home** button to go back to the main menu.

**LIBRARY INFORMATION SYSTEM** Welcome librarian! [ Log Out ]

Home Book Management Author Management

### CREATE NEW AUTHOR

Please enter the author information and click Insert:

**Author Information**

Name:

Age:

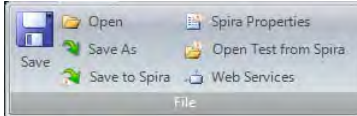
At this point, there should be 11 rows in the **RA dialog grid**.

You are now back on the Main Menu. Click **Log Out** (top-right).





To end the recording session, you can either press **CTRL+3** or press the **Stop** button on the Record dialog. End the recording session now. You will see a script created from your recording session in the Rapise window. Let's save our test. Press the **Save** button at the top left of the Rapise window.



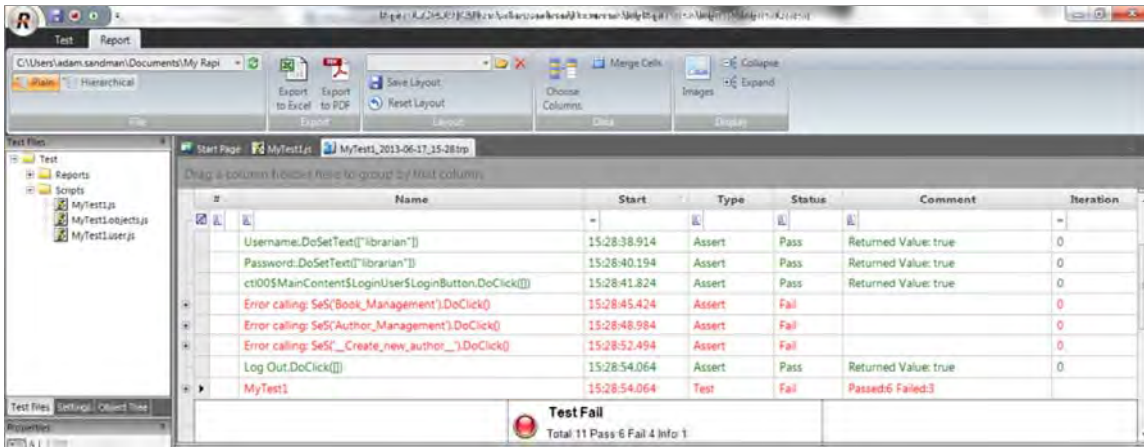
## 5. Playback

Let's execute the test we just created. First, close Internet explorer. Rapise will open a new instance of Internet Explorer to the correct url ([www.libraryinformationsystem.org](http://www.libraryinformationsystem.org)) when the test begins.

To execute the script, press the Play button at the top middle of the Rapise window.



After execution, a screen like the one below will appear. Each row represents a step in the test. The rows with green text are steps which passed, whereas the rows with red text are the steps which failed.



For more information on the report, see [Automated Reporting](#).

## Tutorial: TwoDialogs Sample

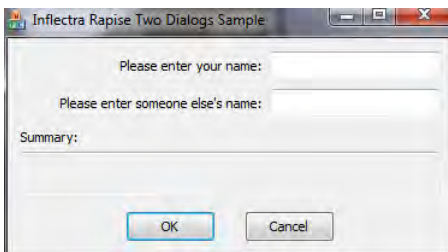
[Top](#) [Previous](#) [Next](#)

This section outlines the usage of Rapise for testing a very simple [AUT](#).

Please run the application now. You will find it in the samples directory where you installed Rapise.

By default, that will be `C:\Users\Public\Documents\Rapise\Samples\TwoDialogs\TwoDialogs.exe`.

You will see the following:



Please run the application a few times and observe its behaviour. If you press the OK button with the first edit box empty, the application will complain and return you to the dialog box.

If you put text in the first edit box but not the second, you will be shown a single line of text in a read-only edit box..

If you enter text in the second edit box as well as the first, pressing OK will put two lines of summary information in the read-only edit box.

An adequate testing strategy for this over-simple application might be to:

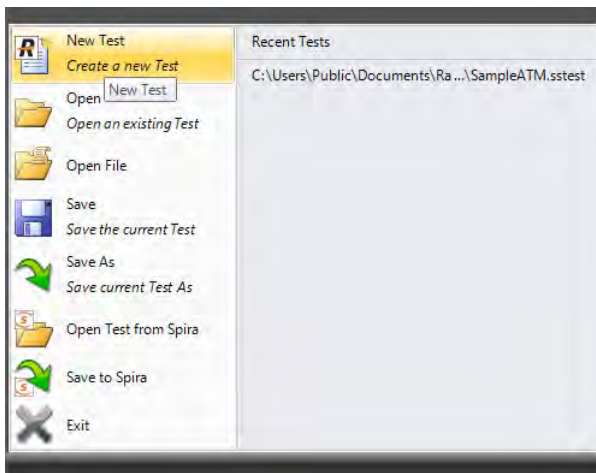
1. Put data in the first text box but not the second, and verify that the summary information is correct.
2. Press the OK button with no data in either text box, and verify that a message box is displayed.
3. Verify that if the success "Thank You" message is displayed the edit box input fields are cleared (but not the summary information).

If at this point you do not understand what the application is supposed to do, or the application is not behaving as described here, please contact [support](#) and clarify the details before proceeding.

Now, let's use Rapise to implement the first of these tests.

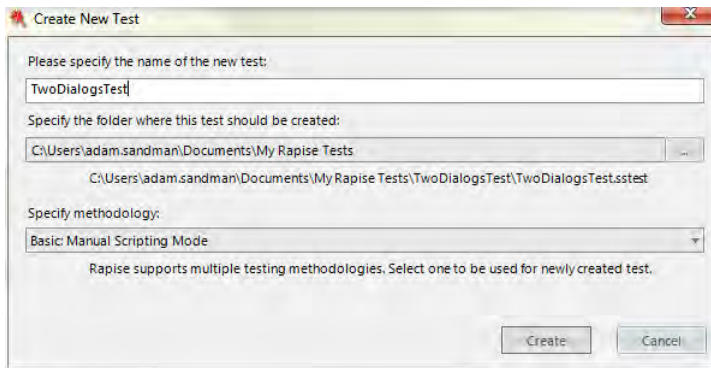
Step 1. Run the TwoDialogs application and leave it in its default start state.

Step 2. Start Rapise and make the window a conveniently large size. Click on the  button (top left). Choose the first option there, "New Test."

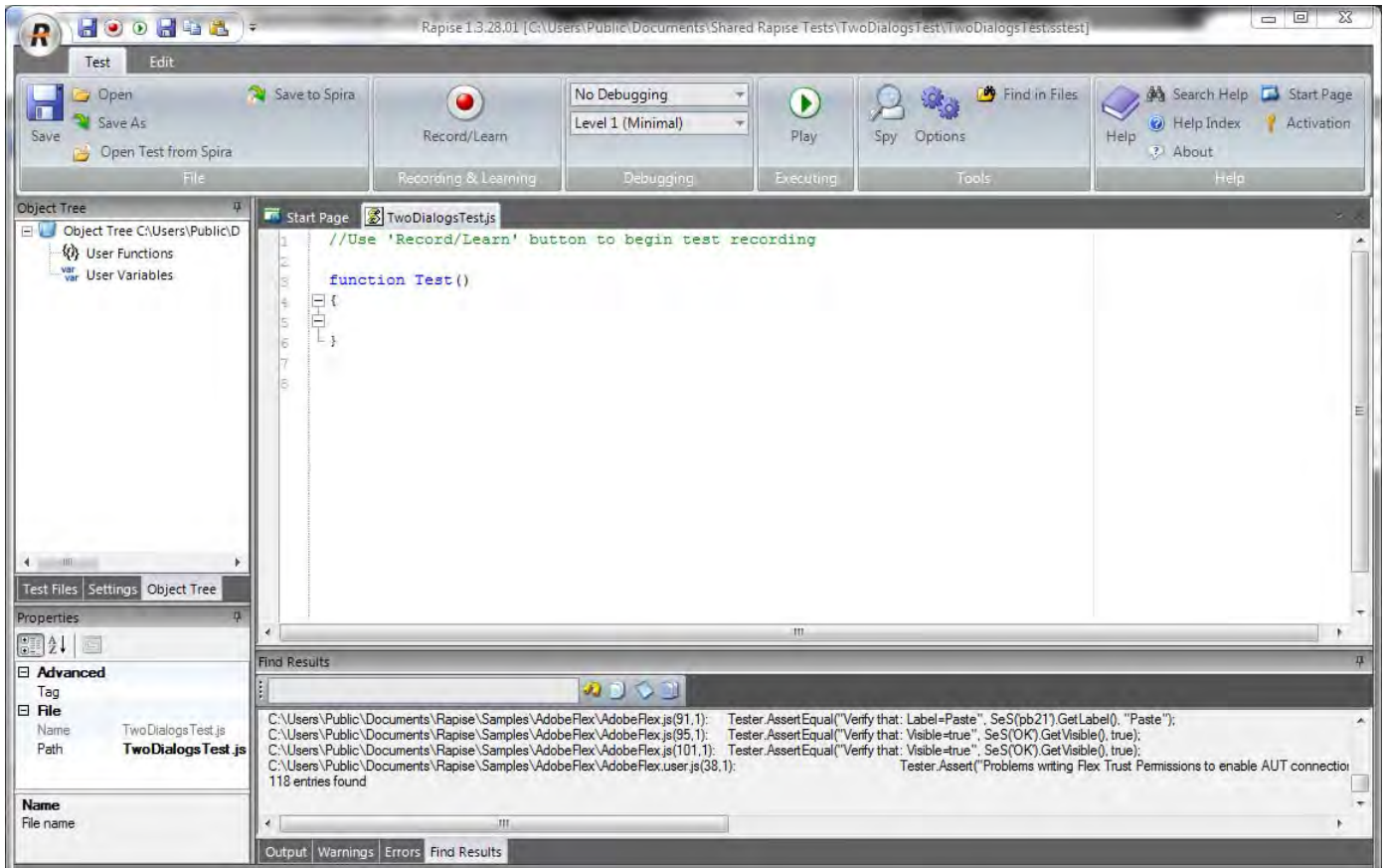


Step 3. Navigate to the desired path using the "..." button on the "Create New Test" dialog.

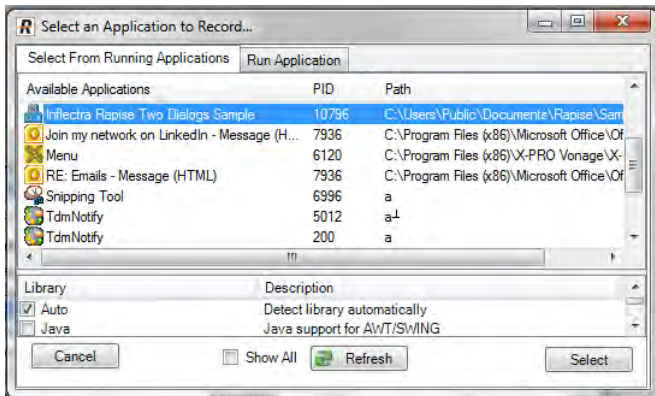
Leave the "Use Methodology" as "Basic" for now.  
Press the "Create" button.



You will now see the following:



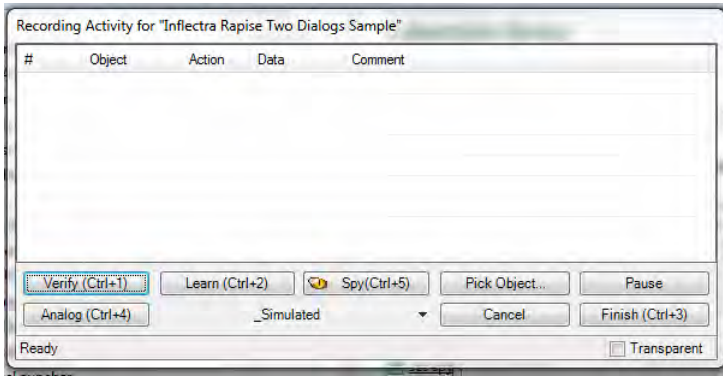
**Step 4. Recording the test sequence.** Press the "Record/Learn" button in either the ribbon or on the toolbar. It has an icon like this:  
You will see an application selection dialog like the following.



Select the "Inflectra Rapise Two Dialogs Sample" entry.  
Leave the library selection as "Auto."  
Press the "Select" button at the bottom right.

**Step 5. Record the activity in the application.**

Rapise will pause while it starts the necessary background processes and hooks into the running AUT.  
Once those tasks are complete, you will see the following "Recording Activity" for "Inflectra Rapise Two Dialogs Sample" dialog:



The AUT will be brought to the foreground and Rapise will be minimized.

You will achieve best results in recording if you observe the following guidelines:

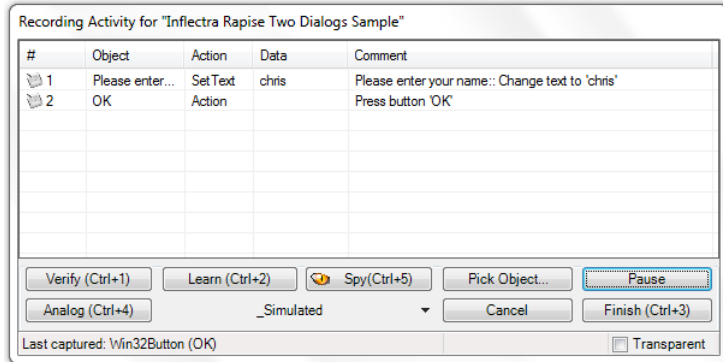
- (1) Work slowly while recording. Perform one action and wait for the results to be recorded in the Recording Activity dialog as a new grid line-item before going to the next item.
- (2) Use the mouse to select controls and operate them. Avoid using keyboard shortcuts and keyboard commands.

**Step 6. Click in the first edit box in the TwoDialogs application. Type a name in there.**

Watch the Recording activity dialog as you operate the AUT interface. As you press a button or fill a field, notice that the grid in the Recording activity has entries added to it.



As you take these actions, you will see the Recording Activity grid update accordingly:

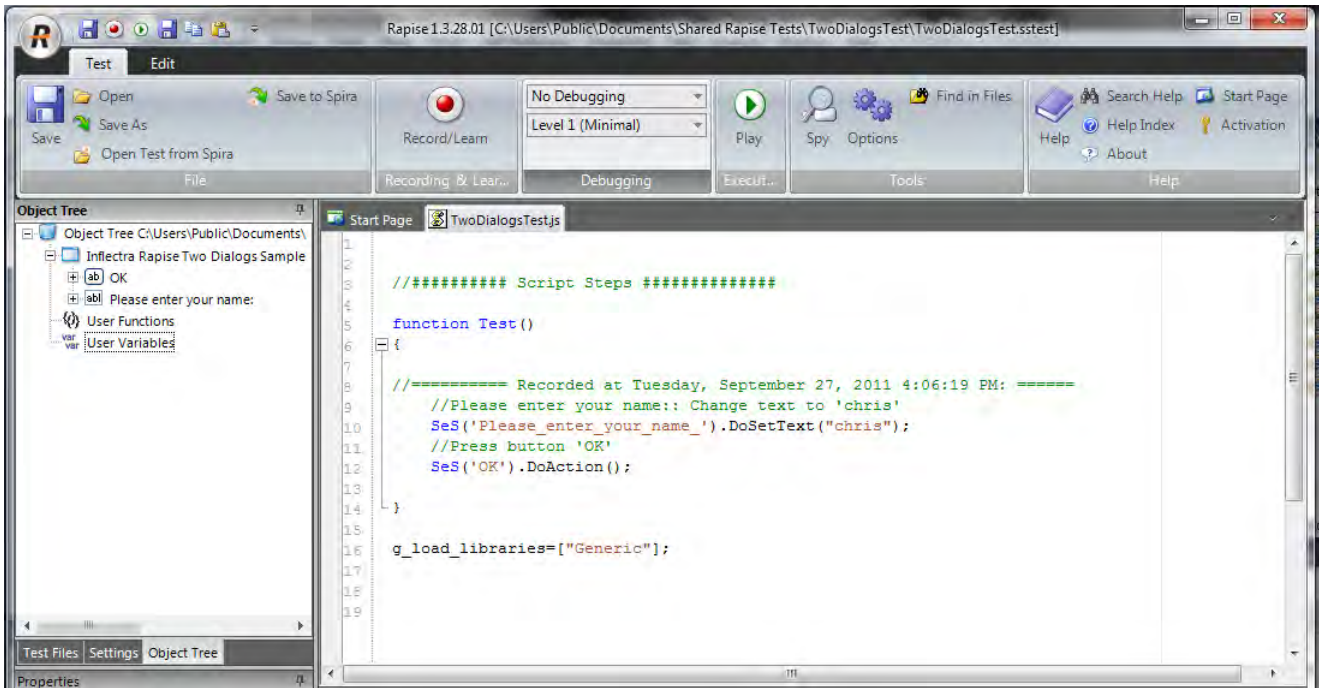


For a full explanation of the controls on this dialog, refer to the reference for [Recording Activity Dialog](#)

When you have finished recording the activity for the AUT, press the "Finish" button or CTRL+3.

Note: Do not terminate the `TwoDialogs` application.

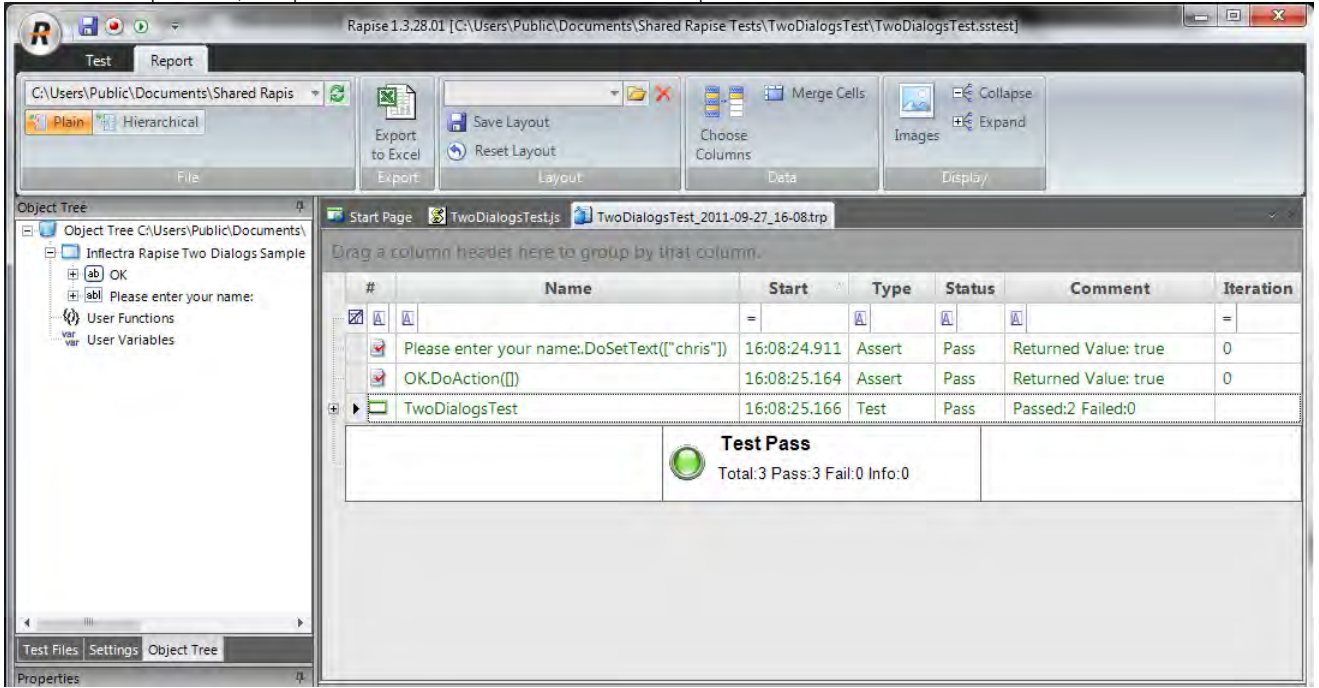
When you do this, the "Recording Activity" dialog will be closed and the AUT will lose focus. Rapise will change the view to display the newly recorded script. It will look something like the following:



Notice that the two steps of the script are automatically documented and that they correspond precisely and in the same order as the way they appeared in the Recording Activity dialog during recording.

Step 7: Run ("Play") the recorded test script. Press the "Play" button on the ribbon or the toolbar.

As the script runs, the Rapise window will be minimized to the taskbar and you will see the results of the script's activities on the TwoDialogs application window. At the end of the script execution, the Rapise window will be restored and the view will be of the report for the test:



**Step 8: A refinement on the launching of TwoDialogs.exe.**

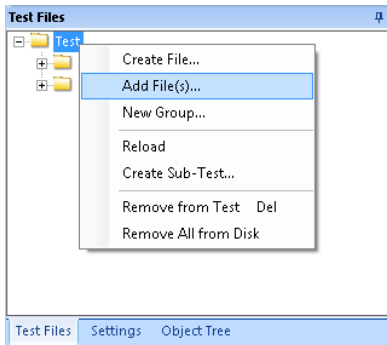
To date, we have operated on the assumption that the TwoDialogs sample program (application) is running. If this situation remained, the test script would require that the AUT be running before the script started. That would require that the person running the test remembered where it resided. To overcome this, Rapise provides a way to have the script run the program (AUT) before beginning the test.

Rapise has an underlying scripting language based on JavaScript (see [Scripting](#)). This help system covers available scripting objects in detail from a practical perspective. For the moment, we want to simply take the shortest path to starting the application before attempting to run the test.

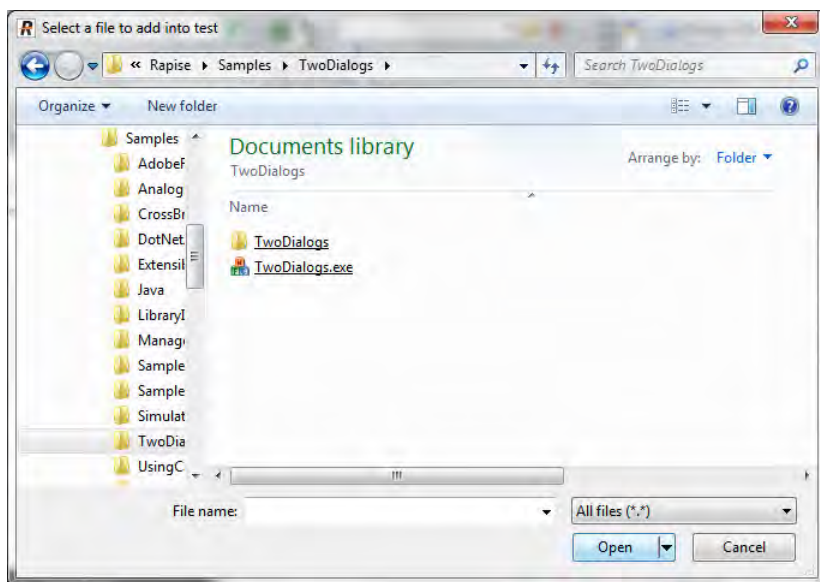
There are at least 3 ways of adding application launch code to your test.

**Way 1: Drag The File from the Test Files view**

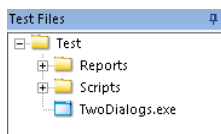
First, switch to [Test Files](#) view. Right-click on "Test" folder and choose "Add File(s)..." menu item:



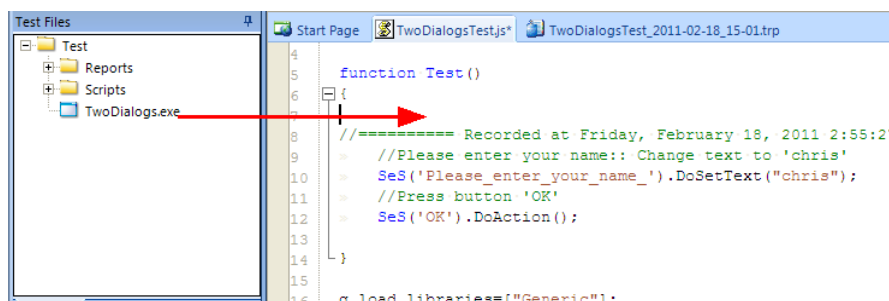
And select the location of the `TwoDialogs.exe` (normally, it is `C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe`),



Now you have the executable as a part of your test files set:



If you wish to launch `TwoDialogs.exe` once then just double-click on it in the tree. If you wish it to be launched every time the test starts then simply drag it from the tree into the source code:



The proper launch statement will be inserted:

```
function Test()
{
  > Global.DoLaunch('../..../Rapise/Samples/TwoDialogs/TwoDialogs.exe');
  > //===== Recorded at Tuesday, September 27, 2011 4:06:19 PM: =====
  > //Please enter your name:: Change text to 'chris'
  > SeS('Please_enter_your_name_').DoSetText("chris");
  > //Press button 'OK'
  > SeS('OK').DoAction();
}
```

## Way 2: Type the Code

The `Global` object contains methods that are available to all scripts.

Select the `TwoDialogs.js` file in the `Test Files` view of the Rapise main page.

Double-click the file name to open it in the main (editing) window of Rapise. You will see the generated script from the recording session from earlier steps in this sample.

Place the cursor in the main editing window and click on the first line after

```
function Test()  
{
```

Now type

Global.

As soon as you type the ".", Rapise will give you a drop down list of all the available methods available in the Global object:

- DoAnalogPlay(path, left, lc)
- DoAppActivate(title)
- DoInvokeTest(pathToTest)
- DoKillByName(processName)
- DoKillByPid(pid)
- DoLaunch(cmdLine, wrkD)
- DoMessageBox(prompt, buttons)
- DoOcrTesseract(img)
- DoOcrTesseract(img)
- DoSendKeys(keys)

Select the DoLaunch(cmdLine, wrkD) member and hit the Enter key.

Now your script contains the line:

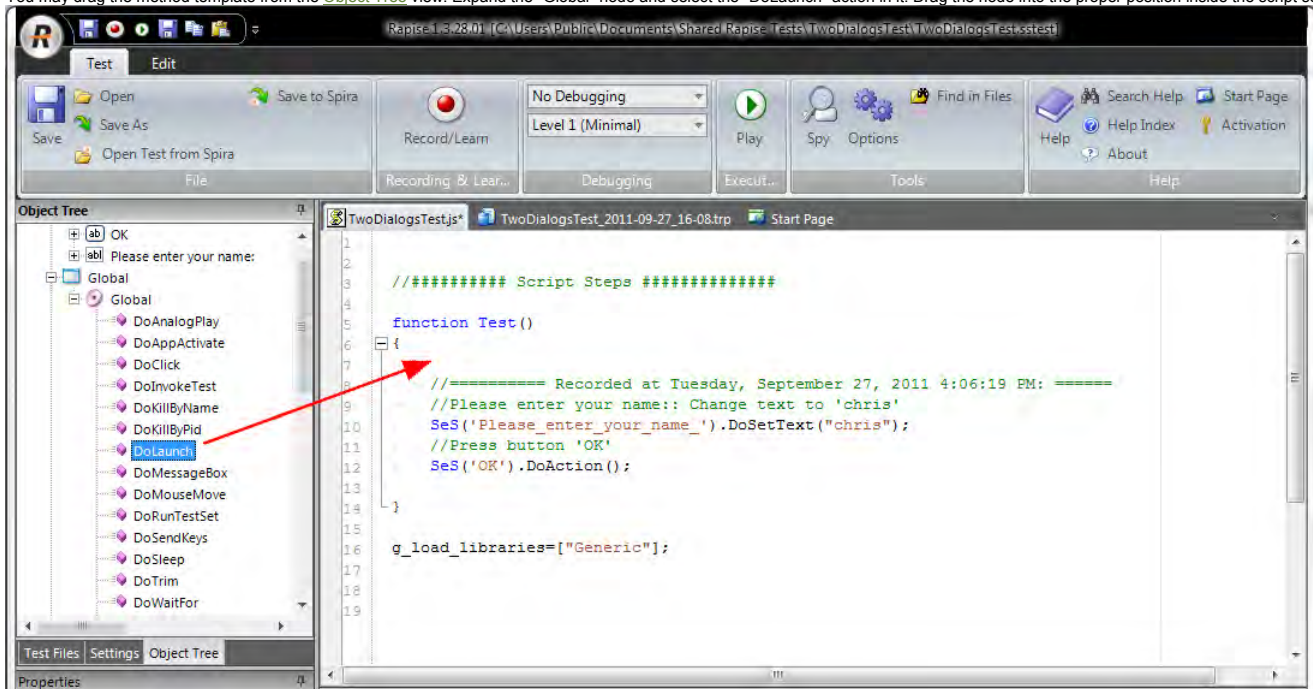
```
Global.DoLaunch("")
```

You need to correct the references to the command line:

```
Global.DoLaunch("C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe");
```

### Way 3: Drag the Action from the Objects Tree

You may drag the method template from the Object Tree view. Expand the "Global" node and select the "DoLaunch" action in it. Drag the node into the proper position inside the script source:



Template call is inserted:

```
4  
5 function Test()  
6 {  
7   Global.DoLaunch("");  
8   //===== Recorded at Friday, Feb  
9   //Please enter your name:: Chang  
10  //SeS('Please_enter_your_name_').D  
11  //Press button 'OK'
```

Now you need to correct the references to the command line:

```
Global.DoLaunch("C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe");
```

## Tutorial: Testing Adobe Flex Applications

[Top](#) [Previous](#) [Next](#)

### Contents

[Introduction](#)

[Prerequisites](#)

[Create a Simple Flex Application: Hello Flex](#)

[Enable HelloFlex Application for Testing](#)

[Link HelloFlex with Necessary Libraries](#)

[Add HelloFlex to FlashPlayerTrust](#)

[Record a Simple Test](#)

[Execute the Test](#)

[Using FlexLoader](#)

[See Also](#)

## Introduction

After going through this tutorial you'll get a basic idea of how to test browser-based Flex applications with Rapise.

## Prerequisites

This tutorial assumes that you have installed:

1. Rapise
2. Adobe Flex Builder 3 (<http://www.adobe.com/products/flash-builder-family.html>)  
OR  
Adobe Flash Builder 4 (<http://www.adobe.com/products/flash-builder-family.html>)

## Create a Simple Flex Application: HelloFlex

Let's start from creation of a very simple Flex application.

1. Create home directory for the application: **C:\HelloFlex**. You may create any other directory that is more suitable for you, just do not forget to change corresponding paths used in this tutorial.
2. Create main file of the application: **C:\HelloFlex\HelloFlex.mxml**. Place the following code in it:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  viewSourceURL="src/HelloFlex/index.html"
  horizontalAlign="center" verticalAlign="middle"
  width="640" height="480"
>
  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
    ]]>
  </mx:Script>
  <mx:Panel
    paddingTop="10" paddingBottom="10" paddingLeft="10" paddingRight="10"
    title="My Application"
  >
    <mx:Label text="Hello Flex!" fontWeight="bold" fontSize="24"/>
    <mx:Button id="button" label="Button" click="{Alert.show('Button Pressed')}"/>
  </mx:Panel>
</mx:Application>
```

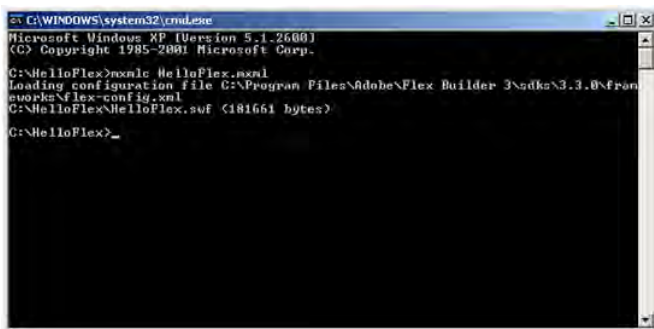
3. Create wrapper HTML for the application: **C:\HelloFlex\HelloFlex.html**. Place the following code in it:

```
<html lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>HelloFlex</title>
</head>
<body scroll="no">
  <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    id="HelloFlex" width="100%" height="100%"
    codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash.cab">
    <param name="movie" value="HelloFlex.swf" />
    <param name="quality" value="high" />
    <param name="bgcolor" value="#869ca7" />
    <param name="allowScriptAccess" value="sameDomain" />
    <embed src="HelloFlex.swf" quality="high" bgcolor="#869ca7"
      width="100%" height="100%" name="HelloFlex" align="middle"
      play="true"
      loop="false"
      quality="high"
      allowScriptAccess="sameDomain"
      type="application/x-shockwave-flash"
      pluginspage="http://www.adobe.com/go/getflashplayer">
    </embed>
  </object>
</noscript>
</body>
</html>
```

4. Compile the application (make sure that **mxmlic.exe** is available in command line window. If Flex Builder 3 is installed then it is available at: "**c:\Program Files\Adobe\Flex Builder 3\sdks\<SDK Version>\lib**

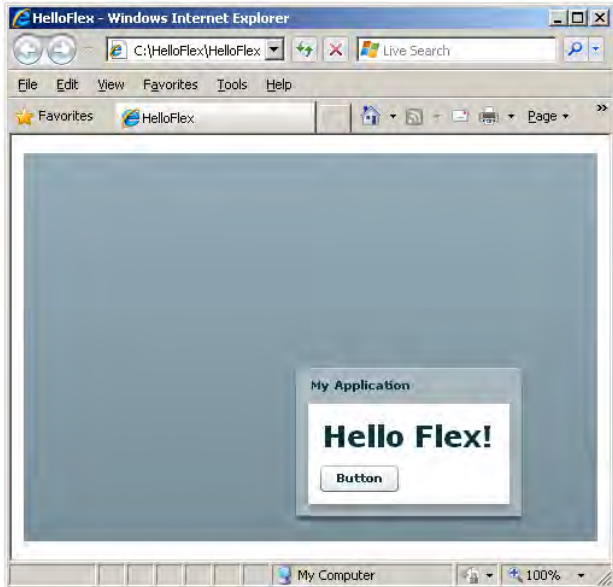
If Flash Builder 4 is installed then it is available at: "**c:\Program Files\Adobe\Flash Builder 4\sdks\<SDK Version>\bin\mxmlic.exe**")

- a) Open CMD window in **C:\HelloFlex** directory
- b) Run command: **mxmlic HelloFlex.mxml**



5. Test the application by opening **C:\HelloFlex\HelloFlex.html** in Internet Explorer.





### Enable HelloFlex Application for Testing

To make HelloFlex application testable by Rapise you need to link it with automation libraries.

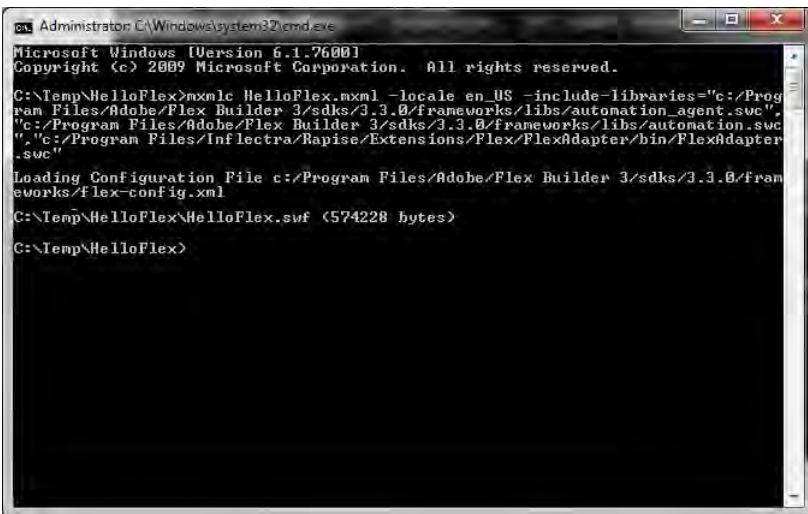
### Link HelloFlex with Necessary Libraries

For Flex Builder 3.x, recompile the HelloFlex application using the following command line that links **automation.swc** and **automation\_agent.swc** from Flex Builder 3 and **FlexAdapter.swc** from Rapise:

```
mxmhc HelloFlex.mxml -locale en_US -include-libraries="c:/Program Files/Adobe/Flex Builder 3/sdks/<Version>/frameworks/libs/automation_agent.swc","c:/Program Files/Adobe/Flex Builder 3/sdks/<Version>/frameworks/libs/automation.swc","c:/Program Files/Inflectra/Rapise/Extensions/Flex/FlexAdapter/bin/FlexAdapter.swc"
```

For Flash Builder 4.x, recompile the HelloFlex application using the following command line that links **automation.swc** and **automation\_agent.swc** from Flash Builder 4.x and **FlexAdapter.swc** from Rapise:

```
mxmhc HelloFlex.mxml -locale en_US -include-libraries="c:/Program Files/Adobe/Flash Builder 4/sdks/<Version>/frameworks/libs/automation_agent.swc","c:/Program Files/Adobe/Flash Builder 4/sdks/<Version>/frameworks/libs/automation.swc","c:/Program Files/Inflectra/Rapise/Extensions/Flex/FlexAdapter/bin/FlexAdapter.swc"
```



### Add HelloFlex to FlashPlayerTrust

Adobe Flash Player has restricted security settings for SWFs opened from file system. To enable testing of such SWFs their corresponding folders must be listed in FlashPlayerTrust directory.

Path to FlashPlayerTrust directory:

to enable testing for all users:

```
<system>\Macromed\Flash\FlashPlayerTrust
```

to enable testing just for current user:

```
<ApplicationData>\Macromedia\Flash Player\#Security\FlashPlayerTrust
```

(on Vista this path looks like:

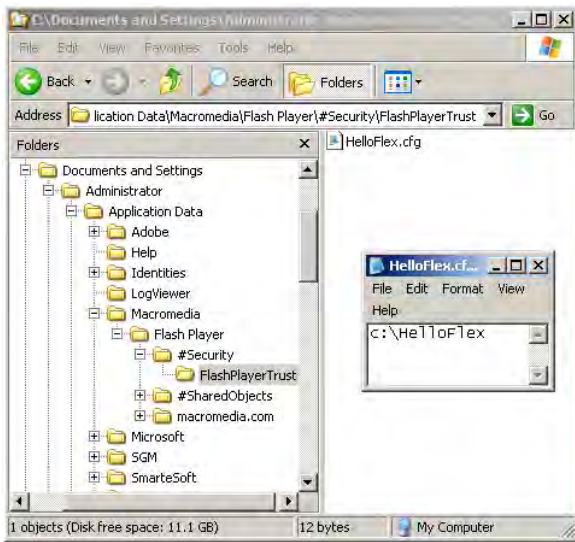
```
c:\Users\<User Name>\AppData\Roaming\Macromedia\Flash Player\#Security\FlashPlayerTrust
)
```

To register your SWF just create a file with the name "<name of your SWF>.cfg" and put it in this directory. In the file write a path to SWF folder.

**Note:** If you do not have *FlashPlayerTrust* directory in one of locations listed above then you will have to create missing directories yourself.

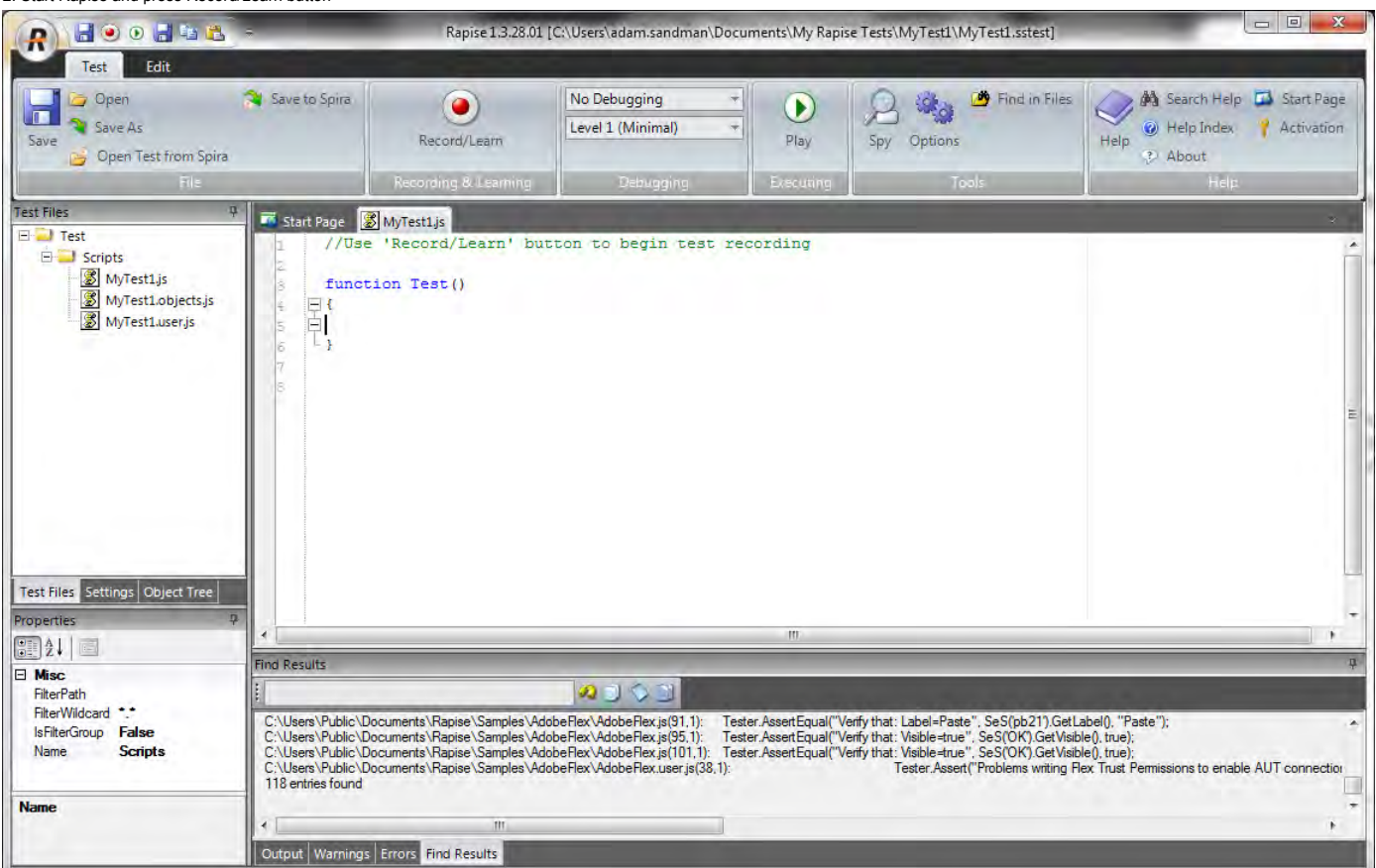
To register c:\HelloFlex\HelloFlex.swf

- a) create file <ApplicationData>\Macromedia\Flash Player\#Security\FlashPlayerTrust\HelloFlex.cfg
- b) add this to the file: c:\HelloFlex

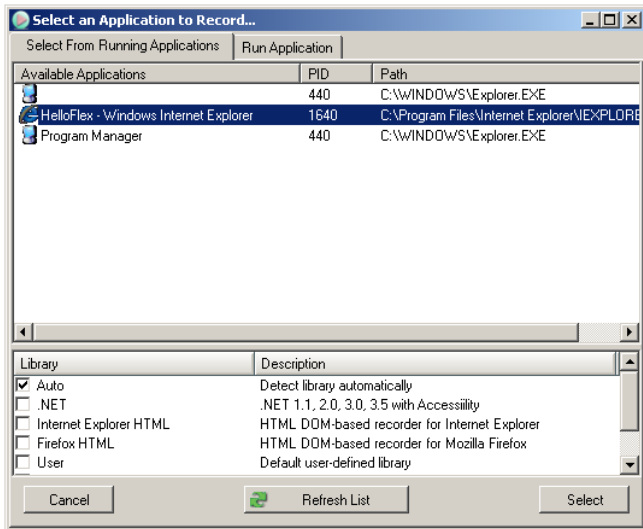


### Record a Simple Test

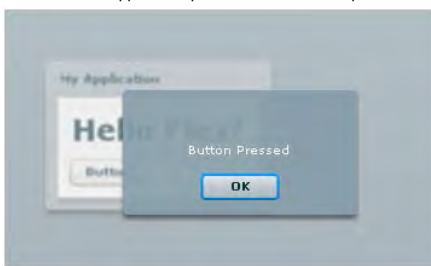
- 1. Open C:\HelloFlex\HelloFlex.html in Internet Explorer.
- 2. Start Rapise and press Record/Learn button



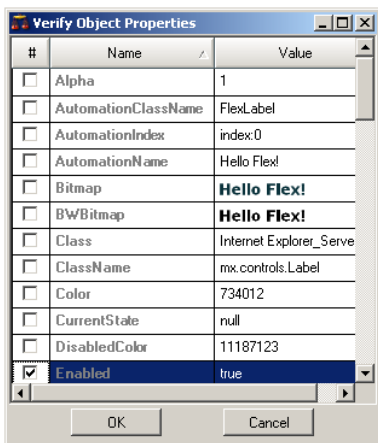
- 3. Choose HelloFlex application and press Select, recording will start.



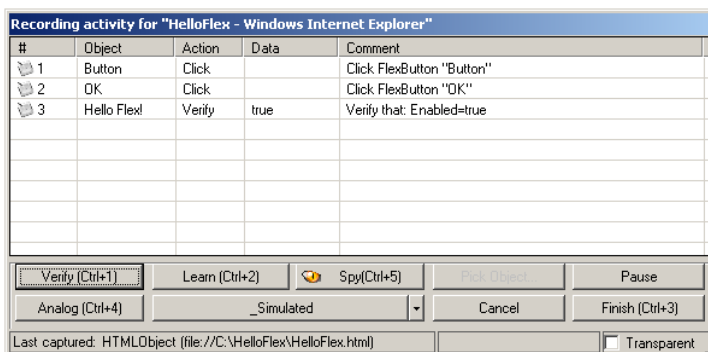
4. In HelloFlex application press Button and then press OK in the alert message.



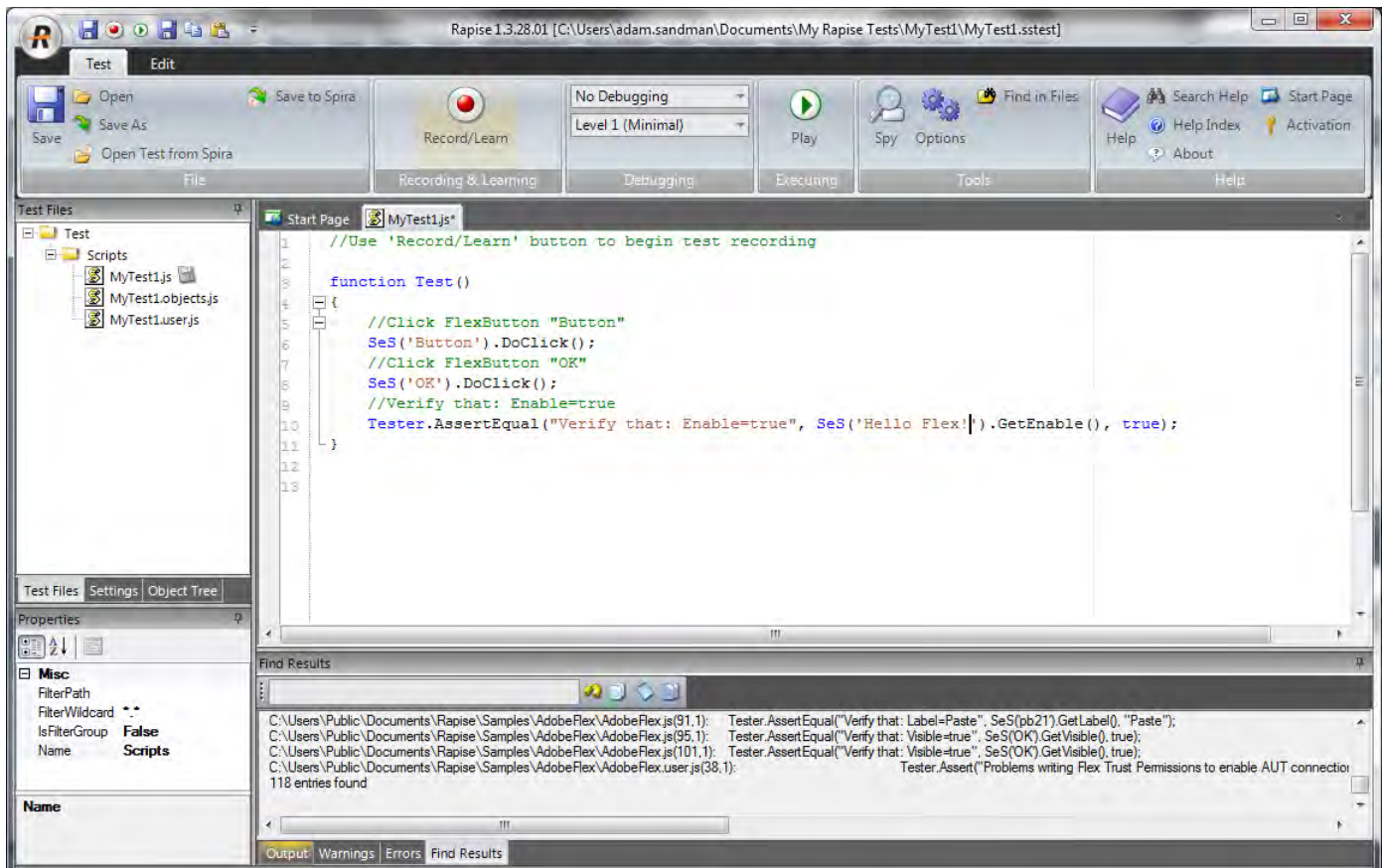
5. Then press Verify button on Recording activity dialog and click on "Hello Flex!" label. In Verify Object Properties dialog check Enabled property.



6. You have recorded three basic steps of your test.



7. Press Finish button on Recording activity dialog. You now have recorded the test.



### Execute the Test

Execute the test by pressing the Play button in Rapise.

Congratulations! You have successfully completed this tutorial and now know basics of testing Flex applications with Rapise.

### Using FlexLoader for Flex 3 Applications

If you do not want to compile your Flex 3 application with automation libraries you have an option to use FlexLoader.

FlexLoader is a Flex 3 application compiled with the required automation libraries and capable of loading any given SWF application. With FlexLoader you do not need to modify your application to make it (You will need to choose between FlexLoader 3 and FlexLoader 4 according to which Flex SDK version your application uses.)

To use FlexLoader 3 just copy **FlexLoader.html** and **FlexLoader.swf** from `c:\Program Files\Inflectra\Rapise\Extensions\Flex\FlexLoader\bin` to your web server near your application. Then type in browser URI `http://localhost/FlexLoader.html?automationswurl=Sample.swf`

You can find sample application for testing here: `c:\Program Files\Inflectra\Rapise\Extensions\Flex\FlexLoader\bin\Sample.swf`

### Using FlexLoader for Flex 4 Applications

If you do not want to compile your Flex 4 application with automation libraries you have an option to use FlexLoader4.

FlexLoader4 is a Flex 4 application compiled with the required automation libraries and capable of loading any given SWF application. With FlexLoader4 you do not need to modify your application to make it (You will need to choose between FlexLoader 3 and FlexLoader 4 according to which Flex SDK version your application uses.)

To use FlexLoader 4 just copy **FlexLoader4.html** and **FlexLoader.swf** from `c:\Program Files\Inflectra\Rapise\Extensions\Flex\FlexLoader4\bin` to your web server near your application. Then type in browser URI `http://localhost/FlexLoader4.html?automationswurl=Sample.swf`

You can find sample application for testing here: `C:\Users\Public\Documents\Rapise\Samples\AdobeFlex4\AUTFLexFP4\bin-debug\assets`

### See Also

- [Adobe Flex](#)

## Tutorial: Testing REST Web Services

[Top](#) [Previous](#) [Next](#)

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.libraryinformationsystem.org>, and its RESTful web service API can be found at: [www.libraryinformationsystem.org/Services/RestService.aspx](http://www.libraryinformationsystem.org/Services/RestService.aspx).

### 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.

### Overview

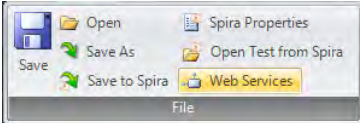
Creating a REST web service test in Rapise consists of the following steps:

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. Writing the test script in Javascript that uses the learned Rapise web service objects.

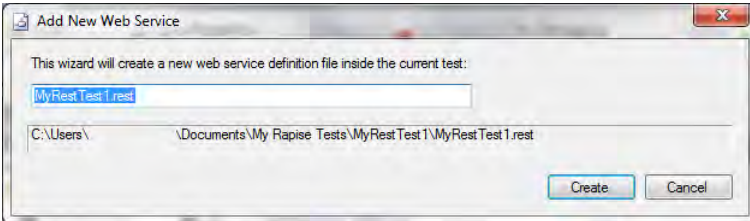
We shall discuss each of these steps in turn.

### 1. Using the REST Query Builder

Create a new test in Rapise called MyRestTest1.sstest. 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:



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](http://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.libraryinformationsystem.org/Services/RestService.svc/book?session\\_id={session\\_id}](http://www.libraryinformationsystem.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.libraryinformationsystem.org/Services/RestService.svc/book?session\\_id={session\\_id}](http://www.libraryinformationsystem.org/Services/RestService.svc/book?session_id={session_id})  
Method: POST  
Pass a populated book object:

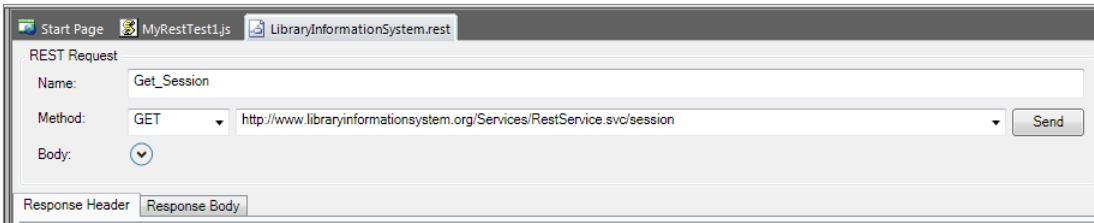
```
Body: {
    "Name": "Book Name",
    "AuthorId": 1,
    "GenreId": 1,
}
```

Returns: Single book object that has its BookId populated

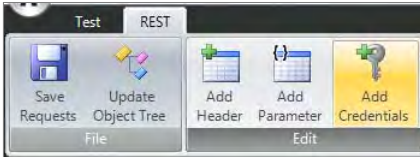
The first request will be to get the unique session ID that we will need to pass to the other requests. This is needed by our sample application to prevent testing by different users interfering with each other. To create this request, simply enter the following information on the REST Request form:

- **Name:** Get\_Session
- **Method:** GET
- **URL:** <http://www.libraryinformationsystem.org/Services/RestService.svc/session>

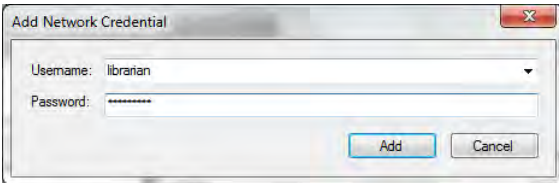
You should now have it populated as illustrated below:



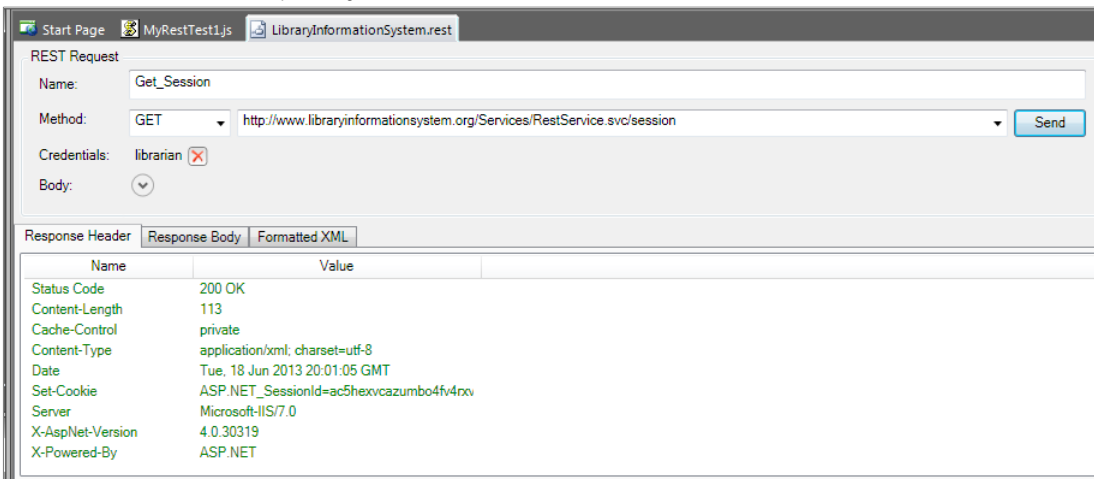
This web service request requires that we pass credentials by means of HTTP Basic authentication. So click on the "REST" tab in the Rapise ribbon and click on the "Add Credentials" button.



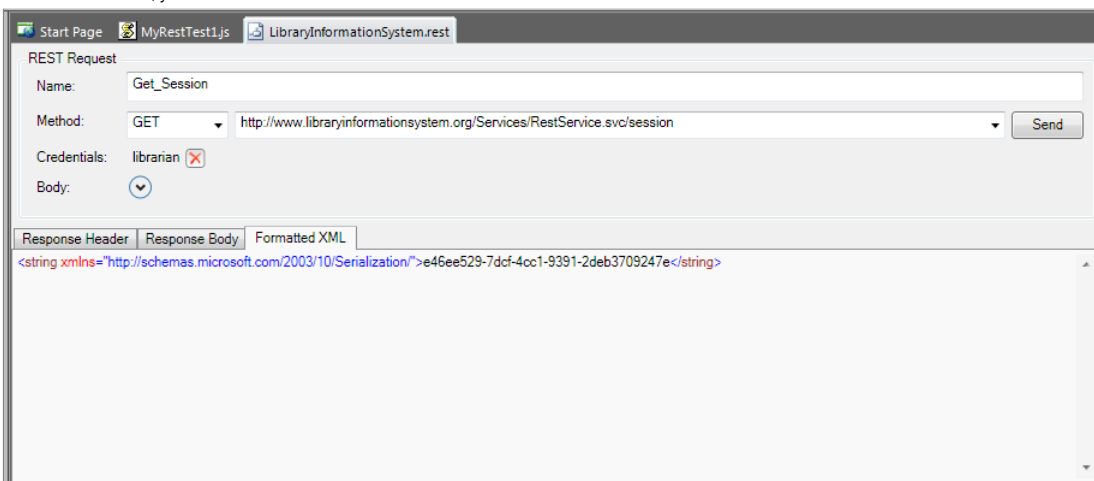
This will display the "Add Credentials" dialog box:



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:



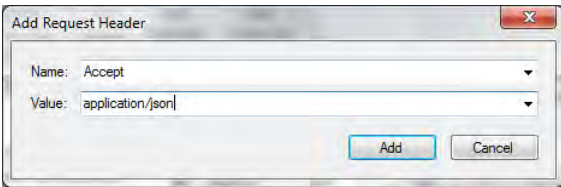
The Response Header tab will display the headers coming back from the web service. The Status Code **200 OK** means that the request succeeded and that data was returned. If you click on the "Formatted XML" tab, you will see the XML serialized data returned from the web service:



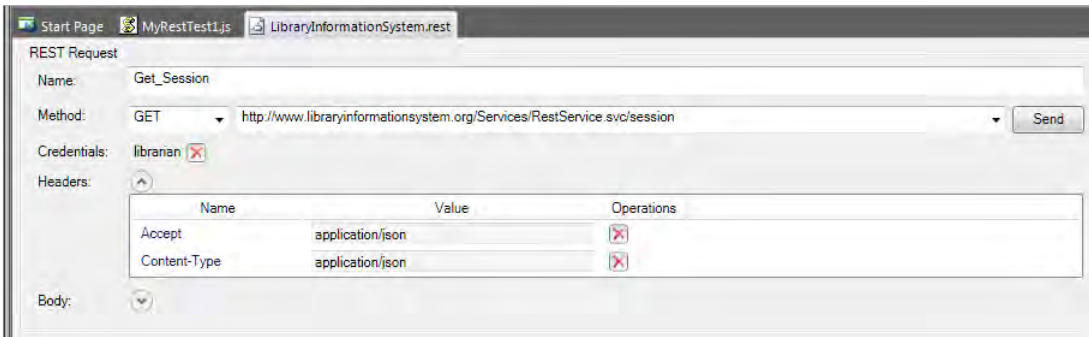
Since Rapise uses JavaScript as its scripting language, it is usually easier to work with JSON (JavaScript Object Notation) serialized data rather than XML. In the case of the sample Library Information System web service, you can change the format that it accepts and retrieves by sending two special HTTP headers:

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

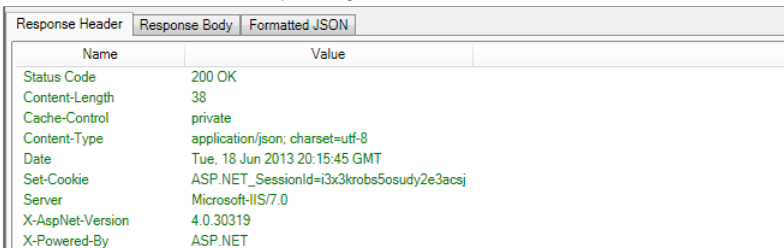
To add these headers to the request, simply click on the "Add Header" button in the REST ribbon tab. This will display the following dialog box:



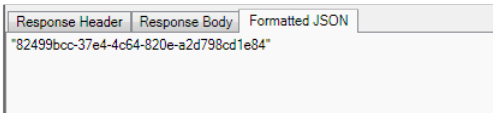
Choose the HTTP Header **"Accept"** from the list and enter **"application/json"** as the value. Repeat for the **"Content-Type"** header. You should now have the following populated request:



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

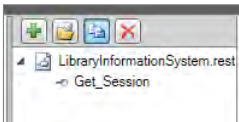


The Response Header tab will display the headers coming back from the web service. Note that the returned Content-Type is listed as "application/json" as requested. If you click on the "Formatted JSON" tab, you will see the JSON serialized data returned from the web service:

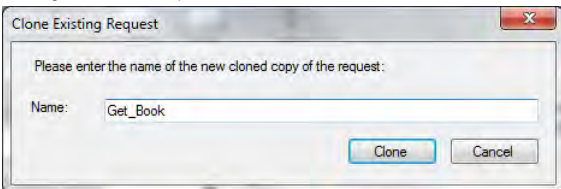


We have now completed the creation of our first test operation. Click on the "Save Requests" button in the Rapise REST Ribbon to make sure our changes have been saved.

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



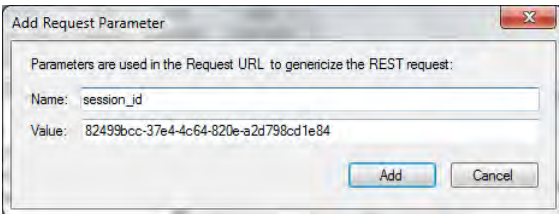
This will display the Clone Request dialog box. This lets us create a new REST request that contains the headers and authentication already defined on our existing request. This will save time over creating a new REST request from scratch:



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



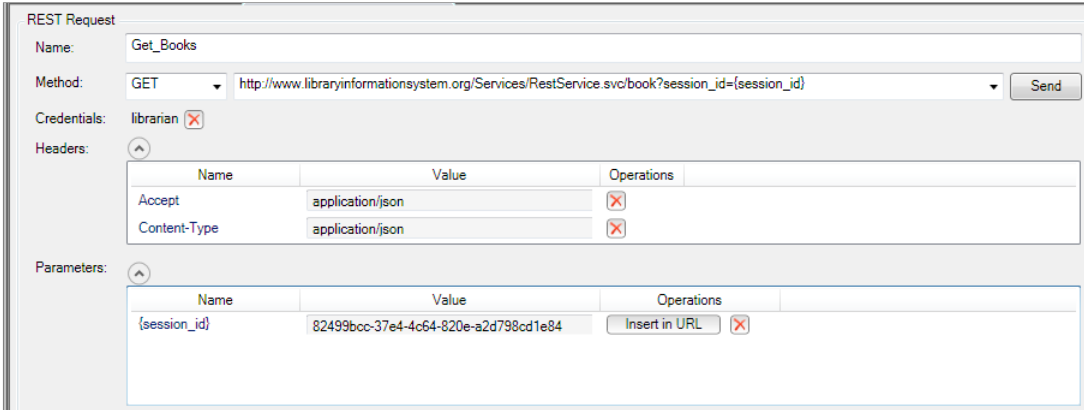
For this request we need to pass through the SessionID in the querystring. Rather than hardcoding it in the URL, we can make use of the parameterization feature of Rapise. Click on the **"Add Parameter"** button in the Rapise REST Ribbon. This will display the "Add Request Parameter" dialog box:



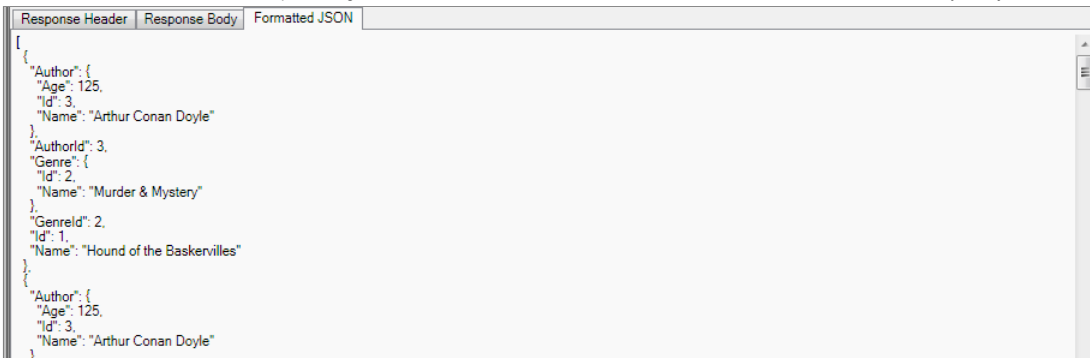
Click the "Add" button and the parameter will be added to the request. Now change the URL to:

**URL:** `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:

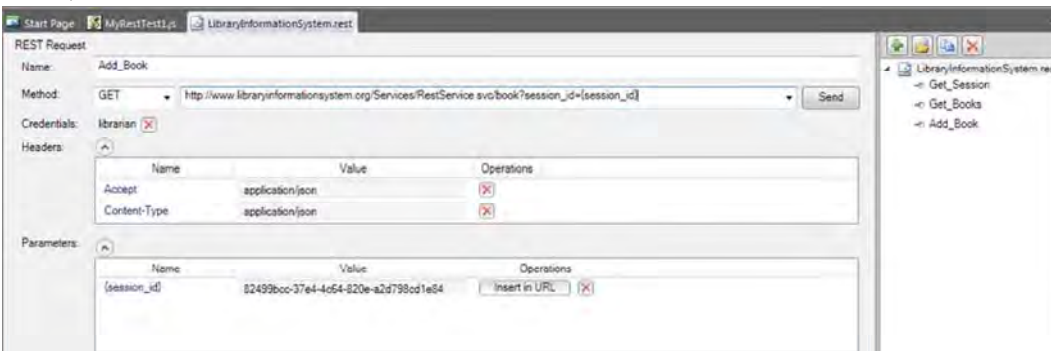


Now click the "Send" button and the request will get sent to the web service. This will return the list of books serialized as a JSON array of objects:



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.

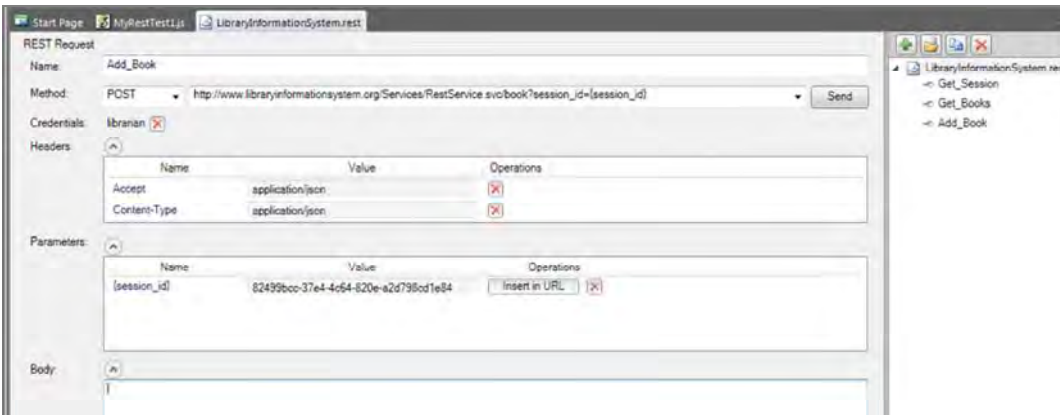
Now click on the "Clone request" icon in the REST request explorer in the right-hand side of the screen. Enter the name "Add\_Book" in the dialog box and click the "Clone" button. This will create a new REST request with this name:



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:





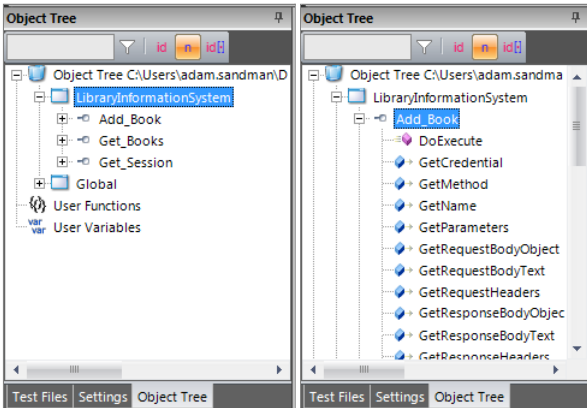
We have now completed the creation of our third test operation. Click on the "Save Requests" button in the Rapise REST Ribbon to make sure our changes have been saved.

## 2. 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 Tree" tab of the main Rapise explorer, click 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.

## 3. 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
2 //##### Script Steps #####
3
4 function Test()
5 {
6
7 }
8
9 g_load_libraries=["Web Service"];
10

```

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

```

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	
<p><b>Test Pass</b> Total:4 Pass:2 Fail:0 Info:2</p>						

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

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);
}

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	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":"c3d8dcd4-6125-4	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":"c3d8dcd4-6125-4	14:49:05.379	Assert	Pass	Returned Value: true	0
	Get_Books.DoExecute(["session_id":"c3d8dcd4-6125-4	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.

## Features

[Top](#) [Previous](#) [Next](#)

Rapise is a feature-rich test automation system, however all the features have been designed to make test automation as easy as possible.

Most of the features of Rapise fall into one of five categories:

- (1) Building test scripts with little or no manual scripting.
- (2) Reading and interpreting results and reports.
- (3) Additional features and capabilities for sophisticated testing.
- (4) Writing more involved or complicated tests using scripting.
- (5) Extending Rapise to learn new or extended libraries of capabilities.

Depending on the application set being tested, not all of these features are necessarily needed for every situation.

For each feature, this document describes:

- (1) The reason you might use a given feature.
- (2) A summary of the basic value of the feature.
- (3) An overview of how the feature works from the perspective of using it.
- (4) At least one useful sample that demonstrates how to use the feature.

## Recording and Learning

[Top](#) [Previous](#) [Next](#)

### Purpose

To understand what different objects might be found on a UI screen, and how to recognize them, record their characteristics and interact with them using Rapise.

### Value

A UI screen entity (object) may consist of many different parts and components. Actions on these objects, and usage of these controls, must be captured in different ways, depending on the properties of the object. Rapise provides four fundamental methods for capturing objects and corresponding user actions:

- (1) **Recording** - Rapise is able to track user interactions with AUT and automatically capture affected objects and corresponding user actions. See [Recording](#) for more information.
- (2) **Learning** - there are cases when it is not necessary or is not possible to track user interactions with AUT. In this case user can manually point to an object that should be captured by Rapise. See [Learning](#) for more information.
- (3) **Analog Recording (Absolute/Relative)** - this is for objects that are not standard in some important way, and so activity on them cannot be captured using recording or cannot be specified after learning. [Absolute Analog Recording](#) is used to track mouse usage (movement and clicks) and keyboard events. For absolute analog recording, the positions these events are recorded relative to the top-left corner of the system screen. (In contrast, in [Relative Analog Recording](#), the events are recorded relative to the upper-left corner of the selected objects.) The events are recorded in a file of type arf (Analog Recording File).
- (4) **Simulated Object Recording** - a Rapise user can use simulated objects when some objects are not natively supported by Rapise (e.g. their internal structure, properties and actions are unknown). In this case, what is recorded are mouse clicks and keyboard activity. Compare to Analog Recording when all mouse and keyboard actions are recorded, including mouse up/down, mouse move events. See [Simulated Objects](#) for more information.

### Usage

Before an operation (press, enter text, select, click, etc.) can be performed on an object automatically, Rapise must be able to identify the object. That identification must be able to locate the object definitively, and it must be able to duplicate the action or operation precisely. This carries several implications. Firstly, if the AUT is in a different position on the screen when it is started, Rapise must still be able to find the objects in the AUT window. Secondly, if the positioning of objects on the AUT window is proportional or relative to the screen size of shape, Rapise must still be able to locate the object.

A secondary set of considerations relates to the fact that the AUT UI layout maybe sensitive to the context of the state of the application. For example, consider the case of a word processor. Pressing the "bold" button doesn't predict what the result will be unless it is known whether the text highlighted is currently bold or not. A far more illustrative example is that of the Microsoft Paint utility. The Microsoft Paint utility is the subject of a Inflectra sample, [Simulated Object](#).

The most instructive way to identify the objects to Rapise is to practice with the tool and different types of objects. The best methodology to use is as follows:

- (1) First, try to use Record/Learn to learn the object and record actions in a single step.
- (2) If learning.recording fails to record actions in the grid, use [SeSSpy](#) to observe the object carefully and discover what libraries and classes are being used by the AUT.
- (3) Use Verify (Ctrl+1) from the Recording Activity dialog to get summary information about the object.
- (4) Use a more appropriate set of libraries when selecting the AUT prior to recording.
- (5) Use Analog Recording with absolute positioning to identify and locate the object.
- (6) Use Analog Recording with relative positioning to identify and locate the object.
- (7) Use Simulated Object Recording to track the actions required and at the positions required.
- (8) Look for [custom libraries](#) that support the technology being used by the AUT.
- (9) Build your own custom library to support the technology in use by the AUT.

## Recording

[Top](#) [Previous](#) [Next](#)

### Purpose

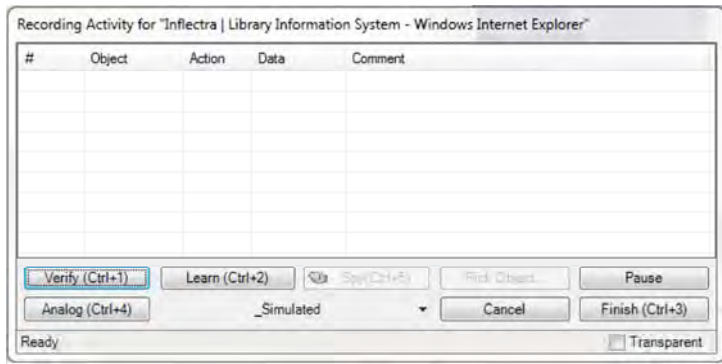
Recording is the name given to having Rapise track your interactions with an application.

### Value

The actions you take in using the [AUT](#) are observed by Rapise and are transformed into a script (javascript), which you can execute using the Play button. The script can be extended and modified to suit special purposes.

### Usage

The **Recording Activity (RA) Dialog** is opened when you start recording using the Record/Learn button. When the Recording Activity dialog appears, Rapise has connected to your AUT and is ready to monitor and record your interactions. You'll find instructions [here](#) or look at one of the examples - [TwoDialogs](#) or [Sample Record and Playback](#)



You'll notice that the RA dialog has a grid. As you interact with the AUT, your actions will be listed in the grid. If you record an incorrect action, you can right-click on the action and delete it.

To ensure you successfully record your interaction with the AUT, navigate slowly through the AUT. Wait a second or two between each action to make sure Rapise has time to interpret and record your action. Once your interaction is updated in the RA dialog grid, you are free to continue with the next action.

When you are done recording, press the Finish button on the RA dialog or type **Ctrl+3**. The RA dialog will disappear, and you will see an automatically generated script opened in Rapise.

### See also

- If you have already recorded a script and want to record additional interactions in the same test, be sure to read [Making Multiple Recordings](#).
- The RA dialog is described more thoroughly in [Recording Activity Dialog](#).
- To learn how to run the script, see [Playback](#). To learn how to modify the script, see [Scripting](#).
- For a detailed tutorial, see [Tutorial: Record and Playback](#) in the Getting Started section.
- For more information on the Spy (ObjectSpy) capability, see [Object Spy](#).

## Learning

[Top](#) [Previous](#) [Next](#)

### Purpose

Objects are the controls and items on the screen of the AUT. "Learning" an object refers to the process of Rapise collecting enough information about the on-screen item to be able to reference the item when the test script is run without ambiguity and regardless of its location on the UI.

### Value

When Rapise "learns" an object, it records the object's type, its name and how to find the object again (locator). It saves everything it learns to the script so that the object can be identified when the test is run. Rapise gives the object a simple name so that you can easily refer to it later if you decide to modify the script.

### Usage

Objects are learned in two ways: (1) during recording or (2) explicitly.

#### Recording

During a Recording session, Rapise learns about each object with which you interact. For details, see [Recording](#).

#### Explicitly

1. Open the **Recording Activity Dialog**. Instructions are [HERE](#).
2. Place your mouse over the object you wish to learn. It should become surrounded by a purple box.
3. Press **CTRL+2**.
4. You will see a new entry in the Recording Activity Dialog, signifying that the object was learned.

Everything Rapise learns about an object is saved in `saved_script_objects`. You can see this variable defined in the `<project-name> objects.js` file that will be listed in the Test Files tab of the Rapise. The following shows what Rapise saved about the "Please enter your name" text box in the [TwoDialogs](#) example:

```
Please_enter_your_name_{
  "locations": [
    {
      "locator_name": "Location",
      "location": {
        "location": "4.4",
        "window_name": "param:window_text",
        "window_class": "param:window_class"
      }
    }
  ],
}
```

```

{
  "locator_name": "LocationPath",
  "location": {
    "window_name": "param:window_text",
    "window_class": "param:window_class",
    "path": [
      {
        "object_name": "param:object_name",
        "object_class": "param:object_class",
        "object_role": "param:object_role"
      },
      {
        "object_name": "param:window_text",
        "object_class": "param:window_class",
        "object_role": "ROLE_SYSTEM_DIALOG"
      }
    ]
  }
},
{
  "locator_name": "LocationRect",
  "location": {
    "window_name": "param:window_text",
    "window_class": "param:window_class",
    "rect": {
      "object_name": "param:object_name",
      "object_class": "param:object_class",
      "object_role": "param:object_role",
      "x": 222,
      "y": 40,
      "w": 140,
      "h": 23
    }
  }
},
"window_text": "Inflectra Rapise Two Dialogs Sample",
"window_class": "#32770",
"object_text": "Chris",
"object_role": "ROLE_SYSTEM_WINDOW",
"object_class": "Edit",
"object_name": "Please enter your name:",
"version": 0,
"object_type": "Win32Text",
"object_flavor": "Text",
"object_library": "Generic"
},
...

```

## See Also

- [Recording](#)
- Learning invisible and [Simulated Objects](#) is slightly more complicated. You can find information on both in the [Recording Activity Dialog](#) section. Look for descriptions of the **Pick Object** button and the **\_Simulated** drop-down menu.
- [Learn Object](#)

## Analog Recording

[Top](#) [Previous](#) [Next](#)

### Concept

During **Analog Recording**, Rapise records mouse movements, keyboard inputs, and clicks.

There are two types of Analog Recording: **Absolute** and **Relative**.

- **Absolute**: Mouse coordinates are recorded relative to the top left corner of the screen.
- **Relative**: Mouse coordinates are recorded relative to the top left corner of the object beneath the mouse cursor.

### Usage

1. Open the **Recording Activity** dialog. Instructions are [HERE](#).
2. To record in **Absolute** mode, press **CTRL+4**. To record in **Relative** mode, press the **Analog** button.
3. Press **CTRL+Break** to end analog recording. You can alternate between normal and analog recording in the same Recording session.

## See Also

- [Recording Activity Dialog](#)

## Absolute Analog Recording

[Top](#) [Previous](#) [Next](#)

### Purpose

Absolute analog recording is used to track mouse usage (movement and clicks) and keyboard events. For absolute analog recording, the positions these events are recorded relative to the top-left corner of the system screen. (In contrast, in relative analog, the events are recorded relative to the upper-left corner of the selected objects.) The events are recorded in a file of type arf (Analog Recording File).

## Value

Not all applications can be recorded by locating and learning objects being used. A very good example of this is free-hand drawing in an application such as Microsoft Paint (Start Menu -> Accessories -> Paint). There are several reasons why this application cannot be recorded using object tracking, learning and recording. The most important is that when the mouse is moved free-hand, it is operating on the same object the whole time - the blank "canvas." Another reason is that the application changes behaviour and the positions of the canvas change depending on the size of the canvas and the positions of floating toolbars.

Absolute analog recording is provided by Rapise to make it possible to make it possible to design and implement tests for these types of applications.

## See Also

- [Do Absolute Analog Recording](#)
- [Relative Analog Recording](#)

## Relative Analog Recording

[Top](#) [Previous](#) [Next](#)

### Purpose

Relative analog recording is used to track mouse usage (movement and clicks) and keyboard events. For relative analog recording, events are recorded in relation to the top-left corner of the application's window. The events are recorded in a file of type art (Analog Recording File).

## Value

Not all applications can be recorded by locating and learning objects being used. A very good example of this is free-hand drawing in an application such as Microsoft Paint (Start Menu -> Accessories -> Paint). There are several reasons why this application cannot be recorded using object tracking, learning and recording. The most important is that when the mouse is moved free-hand, it is operating on the same object the whole time - the blank "canvas." Another reason is that the application changes behaviour and the positions of the canvas change depending on the size of the canvas and the positions of floating toolbars.

Relative analog recording is provided by Rapise to make it possible to make it possible to design and implement tests for these types of applications.

## See Also

- [Do Relative Analog Recording](#)
- [Absolute Analog Recording](#)

## Simulated Objects

[Top](#) [Previous](#) [Next](#)

### Purpose

During normal recording, Rapise [Learns about the Objects](#) you interact with. If, for some reason, Rapise cannot learn an object, you can create a **Simulated Object**. Rapise identifies a simulated object by its location in the Window or Dialog and can perform certain generic actions on it, such as Click and Fill In. This works in the reverse sense also. That is, if Rapise cannot identify an object, or, for example, you click outside any defined object in the AUT's UI, Rapise will create a simulated object to represent the action.

## Value

Not all objects on a screen are "standard" or can be recognized by the libraries loaded. Some are compound objects, consisting of two or more individual objects that work together to deliver a UI effect or behaviour. Simulated objects "fill in the blanks" to allow Rapise to cause an event outside the normal set of objects.

## See Also

- [Recording Activity Dialog](#)
- [Sample Tests: The SimulatedObject sample.](#)
- [Deal with a Simulated Object](#)

## Object Libraries

[Top](#) [Previous](#) [Next](#)

### Purpose

**Object libraries** define what objects and interactions Rapise understands during [Recording](#) and [Learning](#). Most Object Libraries are specific to an application or a set of applications.

### Usage

Rapise comes with several different object libraries:

1. Auto
2. .NET
3. Internet Explorer HTML
4. Firefox HTML
5. Chrome HTML
6. User
7. DOM GWT
8. DOM YUI
9. Generic
10. Advanced Accessibility.
11. WPF
12. Console
13. Java
14. Managed
15. DevExpress
16. Infragistics
17. Telerik
18. Adobe Flex AIR

You can [add your own](#) Recording library--one that understands the objects in your application.

- Selecting **Auto** as the application recording library will cause Rapise to select the one that it deems is most appropriate.
- **.NET**: Use this library with .NET applications. When used with .NET 2.0+ applications you should also include the **Managed** library as well. When used with WPF applications, you should also include the **WPF** library.
- **Internet Explorer HTML**, **Chrome HTML** and **Firefox HTML** are used with Internet Explorer, Google Chrome and Firefox respectively. They understand only the **DOM** (document object model) and therefore capture interactions with the web application, not the browser. They also have access to passwords. Tests recorded with either of the libraries can be run in any of the three browsers. See [Cross Browser Testing](#) for more details.
- **User** refers to [Custom Libraries](#).
- The **DOM GWT** library uses the Document Object Model to learn or record objects found in the Google Web Toolkit.

- The **DOM YUI** library uses the Document Object Model to learn or record objects found in the Yahoo! User Interface library.
- The **Generic** library uses Microsoft's **MSAA** event model to capture user actions. The Generic library should be used if there is no library more specific to the AUT available. The Generic library will record a large set of applications, but it has drawbacks; it may skip some actions and/or record unintended actions. Passwords are not visible to the Generic library, and must be manually entered into the test after recording.
- The **Advanced Accessibility** library is for recording with Internet Explorer. In general, you will want to use the Internet Explorer HTML library. However, there is some information available through Advanced Accessibility that is unavailable when looking solely at the DOM. For example: the absolute screen position of an object. Advanced Accessibility is not precise, as Internet Explorer HTML is, and may miss actions or record unintended actions.
- The **WPF** library is for use with Windows Presentation Framework (WPF) applications.
- The **Console** library is for use with Windows Console Applications that run in the command-line.
- The **Java** library is for use with Java GUI applications that are written using either AWT or SWING.
- The **Managed** library is for use with Microsoft .NET 2.0+ applications. It adds some additional .NET 2.0+ specific-controls to the list supported in the Generic and .NET libraries.
- The **DevExpress** library allows you to record and learn using the various controls provided in the DevExpress DXperience v1.0 component library. This allows you to save time by having the system recognize the various controls directly.
- The **Infragistics** library allows you to record and learn using the various controls provided in the Infragistics component library. This allows you to save time by having the system recognize the various controls directly.
- The **Telerik** library allows you to record and learn using the various controls provided in the Telerik RadControls for Winforms component library. This allows you to save time by having the system recognize the various controls directly.
- The **Adobe Flex AIR** library is for use with applications that are written using Adobe Flash, Flex or AIR.

#### See Also

- [Recording](#)
- To write an Object library specific to your application, see [Custom Libraries](#).
- [Cross Browser Testing](#)
- If you interact with an object that is not defined in your chosen recording library, it will be treated as a [Simulated Object](#).

## Custom Libraries

[Top](#) [Previous](#) [Next](#)

#### Purpose

If your application doesn't work with the predefined [Recording Libraries](#), you can create your own.

#### Usage

Your library can provide **Basic** or **Full** support for your application. Basic support allows you to manually [Learn](#) objects, [write test scripts](#), and [Playback](#) your scripts. Full support allows you to [Record](#) as well. Create your library in the **LibUser** directory. Unless you specified otherwise, you will find it at:

C:\Program Files\Infrectra\Rapise\Engine\Lib\LibUser.

#### Basic Support

**Add a Matcher Rule** to the library for every window type in your application. The SeSMatcherRule includes information to identify your application, and a set of behaviors.

```
var yourApplicationRule = new SeSMatcherRule(
{
  object_type: "yourAppObject",
  classname: "yourAppFrame", //You can use a regular expression here
  behavior: [yourAppBehavior]
})
```

**Override Actions:** Override actions in **yourAppBehavior** (above). The action definitions you provides will be used during [Playback](#). Overriding actions does not affect recording.

```
var HTMLFirefoxBehavior =
{
  actions: [
    {
      actionName: "Click",
      DoAction: function(){}
    },
    {
      actionName: "SetText",
      DoAction: function(**String*/txt){}
    }
  ]
}
```

#### Full Support

**Enable Recording:** You can enable recording in two ways. If your application notifies the [Accessibility Events](#) interface about application events, you can override events in the **behavior** section of **SeSMatcherRules**:

```
var newBehavior={
  actions: [/*section deleted for brevity*/],
  events:
  {
    OnSelect: function(**SeSObject*/ param, /**Boolean*/ badd)
    { /*...*/
    },
    OnSelectAdd: function(**SeSObject*/ param, /**Boolean*/ badd)
    { /*...*/
    }
  }
}

var newRule = new SeSMatcherRule({
  object_type: "someType",
  role: "someRole",
  behavior: [newBehavior],
})
```

Otherwise, you will have to implement **Custom Recording**.

**Custom Recording:** With custom recording, it is the library's responsibility to:

- detect user actions in the application, and
- call **RegisterAction()** (which writes the action to the script).

#### See Also

- To see what actions and events can be overridden, see **SeSBehavior.js** (in the Rapise [Engine](#)).
- Check the **Engine/Lib** directory for examples.
- You can alter the behavior of an action without creating an entire library. See the [Actions](#) section for more details.

## Actions

[Top](#) [Previous](#) [Next](#)

#### Purpose

**Actions** are anything the user can do to a GUI control, such as click, select, fill with text, etc. You can override the behavior of an action, without creating or altering a [Recording Library](#), using **SeSExtendAction()**. Overriding an action affects [Playback](#), but not [Recording](#).

#### Usage

**SeSExtendAction()** is used to override an action handler or add a new **DoAction** handler:

```
function SeSExtendAction(objectType, doActionName, replacementFunction)
```

where:

- **objectType** is the name or [regular expression](#) specifying the object type(s) for which this extension should apply.
- **doActionName** is the name or [regular expression](#) specifying the **DoAction** handler that should be overridden.
- **replacementFunction** is the function containing overriding behavior.

In most cases **SeSEExtendAction()** should be called from within [TestInit\(\)](#).

### Calling Base Actions

The function you are overriding is called the **BaseAction**. You can call it like this:

```
this.BaseAction(arguments);
```

You may override actions several times. For example:

```
function DoActionB()
{
    this.BaseAction();
}

function DoActionC()
{
    this.BaseAction();
}

SeSEExtendAction("Win32Button", "DoAction", DoActionB);
SeSEExtendAction("Win32Button", "DoAction", DoActionC);
```

When **DoAction** is called for the **Win32Button**, the following sequence is executed:

```
DoActionC->DoActionB->DoAction
```

### See Also

- To see what actions can be extended, look in **SeSBehavior.js** (in the [Rapise Engine](#)).

## Multiple Recordings

[Top](#) [Previous](#) [Next](#)

### Purpose

Every time you record, the script recorder updates your test script. Be cautious about what changes you make to the test script; some changes could be lost if the recorder is re-run (see **Usage**).

### Usage

The test script path can be found in the [Settings Dialog](#) under **Settings > ScriptPath**. Unless you specify otherwise, the test script is named *testname.js* (where *testname* is whatever you named your test).

Note that the Script Recorder only has knowledge of four functions and two data structures:

1. function Test()
2. function TestInit()
3. function TestFinish()
4. function TestPrepare()
5. array "g\_load\_libraries"
6. map "saved\_script\_objects"

You can make changes to the body of any of the above functions, and you can alter the initialization of **g\_load\_libraries** and **saved\_script\_objects**. All other changes are unsafe.

During Recording, the Script Recorder:

1. Appends newly recorded actions to the **Test()** function
2. Appends newly encountered objects to the **saved\_script\_objects** array
3. Updates **g\_load\_libraries** to reflect the library selections you made in the [Select an Application to Record... Dialog](#)
4. Ignores (and leaves intact) the definitions of **TestInit()**, **TestFinish()**, and **TestPrepare()**

For example, suppose that you have the following code inside your script file:

```
//External comment // UNSAFE: will be removed by recorder
/*Another comment*/ // UNSAFE
var external_var; // UNSAFE

function Test ()
{
    //comment --SAFE
    var external_var; //SAFE: defines a local variable for function "Test"
    global_var=value; //SAFE: updates (or defines) a global variable
    //SAFE everything inside this function will be kept intact after recording
}
```

The parts of code marked **UNSAFE** will be deleted by the script recorder.

### See Also

- [Settings Dialog](#)
- [Select an Application to Record... Dialog](#)
- [Recording](#)

## Object Spy

[Top](#) [Previous](#) [Next](#)

### Purpose

**Object Spy** allows you to inspect an object's properties and state.

### Value

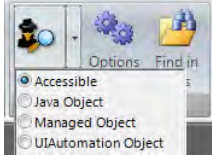
Many controls on **Ujs** are compound objects or there may be many instances of a similar object. To be sure to select precisely the correct object, or to select the correct object from a collection of similar objects, the object's properties can be used to further identify the correct instance.



## Usage

To spy on an Object:

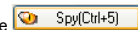
1. Choose the type of **Object Spy** that you want to use. This can be done by clicking the down-arrow next to the Spy icon in the Tools ribbon:



There are **four** types of Spy available:

1. **Accessible** - This is used to inspect applications that expose their properties using the Microsoft Active Accessibility (MSAA) technology. This is typically used by applications written in MFC, ATL, Qt, C++ and Visual Basic.
2. **Java Object** - This is used to inspect applications written using the Java AWT and Swing UI frameworks.
3. **Managed Object** - This is used to inspect applications written in .NET 1.1, .NET 2.0, .NET 4.0 using Microsoft Windows Forms.
4. **UIAutomation Object** - This is used to inspect applications that expose their properties using the Microsoft's newer UIAutomation technology. This is typically used by applications written in WPF, Silverlight and Java SWT.

For more details on each Spy type, refer to the [Spy Dialog](#) information.



2. Open the **SeS Spy Dialog**. This can be done directly using the Spy button in the main Rapise window's toolbar, or by pressing the **Spy(Ctrl+5)** button in the [Recording Activity](#) dialog during recording or learning.
3. Press the **Start Tracking** button (or type CTRL+G).
4. As you mouse over different objects, you will see the contents of the SeS Spy dialog change as it collects information about the object.
5. Mouse over the object you wish to spy on and press **CTRL+G**. The reduced-size tracking dialog will be expanded into the larger [SeS Spy Dialog](#) dialog, presenting all the available information for the object.

## See Also

- See the [SeS Spy Dialog](#) for more details.

## Playback

[Top](#) [Previous](#) [Next](#)

### Purpose

When you record a test, Rapise translates your actions into a script. When you **playback** the test, the script is executed.

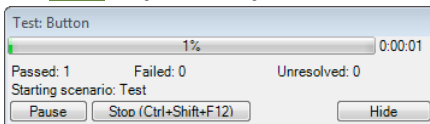
### Usage

You can either run your script from the [Command Line](#), or you can play it back while Rapise is open (described below):

1. You will first need to [open your test](#). There is no need to have the AUT (Application Under Test) open. Rapise will open the AUT before it begins execution of the test.
2. Now, press the play button at the top of the Rapise window.



3. During test execution, Rapise displays an execution monitor dialog box that lets the user see the progress of testing playback. The dialog is only shown during test execution and can be turned off in the [Options](#) dialog. The following is a screenshot of the test execution monitor.



The user can pause or stop the test execution by clicking either the **Pause** or **Stop** button.

4. When Rapise is done executing the test, results will be displayed in a table. The rows with green text are steps that passed; the rows with red text are steps that failed. The following is a screenshot of test results where every step passed:

Drag a column header here to group by that column.

Name	Start	Type	Comment	Status	Iteration
Submit Transaction.DoAction()	14:09:32.17	Assert	Returned Value: true	Pass	0
Verify that: WindowText=Transaction	14:09:32.48	Assert		Pass	0
OK.DoClick()	14:09:32.87	Assert	Returned Value: true	Pass	0
Transfer	14:09:32.87	Test	Passed:7 Failed:0	Pass	0
Balance.DoAction()	14:09:33.14	Assert	Returned Value: true	Pass	0
OK.DoClick()	14:09:33.40	Assert	Returned Value: true	Pass	0
Balance	14:09:33.40	Test	Passed:2 Failed:0	Pass	0
Application.DoMenu(["Account:Exit "])	14:09:35.06	Assert	Returned Value: true	Pass	0
Exit	14:09:35.06	Test	Passed:1 Failed:0	Pass	0
C:\Program Files	14:09:35.06	Test	Passed:7 Failed:0	Pass	

**Test Pass**  
Total: 33 Pass: 33 Fail: 0 Info: 0

## See Also

- For more information about the report, see [Automated Reporting](#).
- For information about recording a test, see [Recording](#).
- For instructions on using the [Command Line](#), look [HERE](#).

## Command Line

[Top](#) [Previous](#) [Next](#)

### Purpose

Rapise test scripts can be run from the **command line**.

## Usage

The form of the command is:

```
cscript SeSExecutor.js path_to_sstest_file [evals]
```

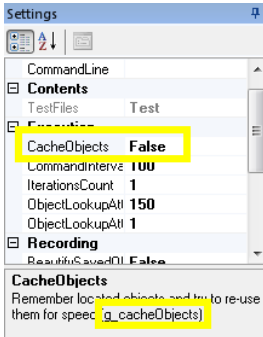
where

`path_to_sstest_file` is a path to sstest file, e.g. "C:\Program Files\Inflectra\Rapise\Samples\SmarteATM\SmarteATM.sstest"

`evals` (optional) is a statement like this:

```
-eval:varname1=value1,varname2=value2;...
```

`varname` is a global variable associated with an option in the [Settings Dialog](#). Global variables are prefixed with a `g_`. The global variables under the **Execution** and **Recording** headings can be found by clicking on the corresponding option in the Settings Dialog (see below):



Other variables include:

- `g_scriptPath`
- `g_reportPath`
- `g_objectsPath`
- `g_configPath`

## Exit Code

- 0 indicates a pass
- 1 indicates failure

## See Also

- [Settings Dialog](#)

## Object Locator

[Top](#) [Previous](#) [Next](#)

## Purpose

Object locators are created during [Recording/Learning](#) and used during [Playback](#) to identify [learned objects](#) and [simulated objects](#). There are four types of locators:

- **Location**: This locator uses the object's index relative to encapsulating objects for identification. The location is stored as a period separated list of indexes. For instance, 1.2.3 would be "the third object in the second object in the first object." The name, class, and role of the object are also stored.
- **LocationPath**: This locator remembers name, class, and role property information for the object and all of its encapsulating objects.
- **LocationRect**: This locator stores screen coordinates.
- **Ordinal**: This locator creates an array of object name/object class combinations. Each object is assigned an index in the array.

## Usage

The locator for each object is specified in `saved_script_objects` in `<scriptname>.objects.js` your test script. Locator information is highlighted in the simulated object example below:

```
Obj10:{"version":0,"object_type":"SeSSimulated","object_name":"regex:.* - Paint",  
"object_class":"MSPaintApp","object_role":"ROLE_SYSTEM_WINDOW",  
"object_text":"regex:.* - Paint",  
"object_rect":{"x":100,"y":100,"width":100,"height":100},  
"locations":[{"locator_name":"Location","location":{"location":""},  
"window_name":"regex:.* - Paint","window_class":"MSPaintApp"}]}
```

## Locator Parameters

If a piece of information in the locator matches a piece of object info (`object_name`, `object_class`, `object_role`, `object_text`) then it is stored in the locator as "param:<object\_info>". For example:

```
"object_name": "param:object_name",  
"object_class": "param:object_class",  
"object_role": "param:object_role",
```

## Over-riding Locator Parameters

You can over-ride the information used to locate your object at runtime. Normally, to refer to an object, you use the SeS function:

```
SeS('Obj9')
```

To override locator parameters, specify the new value in the function call. In the following example, we over-ride the `object_name` parameter for object 9:

```
SeS('Obj9', {object_name:"regex:.*"})
```

You may want to change a parameter value for every locator/object in the program. For instance, perhaps the url of the webpage has changed. Use the global variable `g_locatorparams` as in the following example:

```
function Test()  
{  
  // Here we use direct parameter overriding  
  SeS('Obj1', {url:"http://newaddr/"}).DoAction();  
  SeS('Obj2', {url:"http://newaddr/"}).DoAction();  
  
  // And this is equivalent to above  
  g_locatorparams["url"]="http://newaddr/";  
  SeS('Obj1').DoAction();  
  SeS('Obj2').DoAction();  
}
```

## See Also

- [Object Learning](#)
- [Playback](#)

## Automated Reporting

[Top](#) [Previous](#) [Next](#)

### Purpose

Each time you playback a test, Rapise automatically generates a report detailing the steps of the test, the data values used, and the outcome of each step.

### Usage

Execute your test using the instructions [here](#). When the test is complete, the [Report Tab](#) will appear in the Ribbon, and a report file (ending in .trp) will open in the [Content View](#). It will look like this:

Drag a column header here to group by that column.

#	Name	Start	Type	Comment	Status	Iteration
	Character read successfu	15:32:28.89	Assert	T	Pass	0
	Letter size is 44	15:32:28.89	Assert		Fail	0
	Character read successfu	15:32:28.90	Assert	e	Pass	0
	Letter size is 24	15:32:28.92	Assert	Text = 'e' Font = (Name='Calibri'; Size=2	Pass	0
	Character read successfu	15:32:28.93	Assert	s	Pass	0
	Letter size is 12	15:32:28.93	Assert	Text = 's' Font = (Name='Calibri'; Size=1	Pass	0
	Character read successfu	15:32:28.95	Assert	t	Pass	0
	Letter size is 72	15:32:28.96	Assert	Text = 't' Font = (Name='Calibri'; Size=7	Pass	0
	C:\Program Files	15:32:28.96	Test	Passed:7 Failed:1	Fail	

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

The first row (with a white background) is used for [Report Filtering](#). The rows below that each represent a step in the test. The rows with green text represent success; the rows with red text represent failure. You can reposition the columns by dragging and dropping the column names.

### The Columns

- **#:** For displaying icons.
- **Name:** The test name.
- **Start:** The time the test step began executing.
- **Type:** Can be one of the following values: Test; Assert; Message.
- **Comment:** Assertions and messages have associated comments. They are displayed here.
- **Status:** Whether the step passed, failed, or was merely informational.

### Drag a column header here...

Drag a column header here to group by that column.

Use to order by the values in the chosen column. The result of dragging the **Status** column over looks like this:

Status ▾

#	Name	Start	Type	Comment	Iteration
Status : Fail (2 items)					
Status : Pass (7 items)					

You can expand each item to see the corresponding report rows:

Status ▾

#	Name	Start	Type	Comment	Iteration
Status : Fail (2 items)					
	Letter size is 44	15:32:28.89	Assert		0
	C:\Program Files\	15:32:28.96	Test	Passed:7 Failed:1	
Status : Pass (7 items)					

Drag the **Status** icon back to undo the sort:

Status ▾

#	Name	Start	Status	Type	Comment	Iteration
Status : Fail (2 items)						
Status : Pass (7 items)						

## See Also

- [Report Filtering](#)
- The report output file is specified in the [Settings Dialog](#) (**Settings > ReportPath**).
- The [Report tab](#) of the Ribbon is used to alter the report layout.

## Writing to the Report

[Top](#) [Previous](#) [Next](#)

### Purpose

You can write to individual columns, create columns, and add data to the [report](#).

### Usage

#### Writing to and Creating a Column

Use `Tester.PushReportAttribute` or `Tester.SetReportAttribute` to set values in specific rows. `Tester.PopReportAttribute` reverses the effect of `Tester.PushReportAttribute`:

#### PushReportAttribute

```
Tester.PushReportAttribute(columnName, value);
```

```
...some test steps... //the rows corresponding to these steps will have
```

```

//value in their columnName column
Tester.PushReportAttribute(columnName, value2);
...some test steps... //the rows corresponding to these steps will have
//value2 in their columnName column
Tester.PopReportAttribute(columnName); //test steps proceeding this will be back to value
If columnName does not exist, it will be added to the report.

```

**SetReportAttribute**

```

Tester.SetReportAttribute(columnName, value);
If columnName does not exist, it will be added to the report. Column columnName will be populated with value for rows created after this function call (unless specified otherwise).

```

**Adding Data**

Data must be associated with an **Assert** row or a **Message** row.

```

Tester.Assert(description, expression, data, columnValuePairs)
Tester.Message(description, data, columnValuePairs)

```

- **description** is a string.
- **expression** is the Boolean expression that the assertion tests.
- **data** is an array of data objects. Each data element is written to its own row below the assert/message row with which it is associated. Data can be text, a link, or an image. The following is an array with text, link, and image data.

```

[
  new SeSReportText(text),
  new SeSReportLink(uriString, linkText),
  new SeSReportImage(ImageWrapperObject, imageDescription)
]

```

- **columnValuePairs** is an object with key/value pairs. Column names are the keys. If the specified column does not exist, it will be created. Ex: {requirement: "Req1.2.3", paragraph: "12.5"}

**See Also**

- [Automated Reporting](#)
- The [test samples](#) include a sample about reporting (Reporting.sstest)



**Report Filtering**

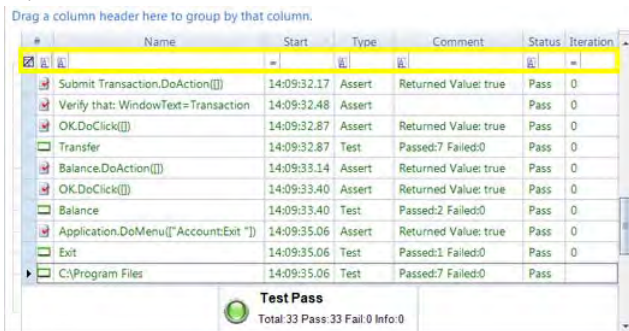
[Top](#) [Previous](#) [Next](#)

**Purpose**

**Report Filtering** lets you specify criteria to filter your view of the [test execution report](#). Rows that do not match your criteria are hidden.

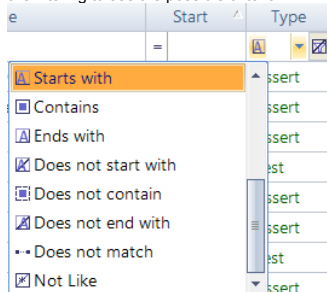
**Usage**

You can filter the report view while the file is open. Directly above the first row of the report, there is a row of filter cells. Each one has a **matching criteria** button , a text-box to specify a filter value, a drop-down menu with **predefined filter values**, and a **clear** button :



**Matching Criteria**

Matching criteria determine how to compare the filter string value you input with the values in the report. You can select from 16 matching criteria. Press the button marked **A** above the column you are filtering to see the possible criteria:



**Predefined Filter Values**

If we expand the filter cell's drop-down menu, we will see a list of predefined filtering options:

ent	Status	Iter
(Custom)		0
(Blanks)		0
(NonBlanks)		0
Fail		0
Pass		0

- **(Custom)**: This option has to do with the next section *Custom Filter Options*.
- **(Blanks)**: Matches all rows where the value for this column is blank.
- **(NonBlanks)**: Matches all rows where the value for this column is not blank.
- All other predefined values are copied from cells in the column you are filtering.

### Custom Filter Option

To create a filter with multiple matching criteria and filter values, select **(Custom)** from the filter cell's drop-down menu. The **Enter filter criteria for...** Dialog will open. Instructions for how to use it are [here](#).

ent	Status	Iter
(Custom)		0
(Blanks)		0
(NonBlanks)		0
Fail		0
Pass		0

### Undo Filtering

To undo filtering for a particular column, press the clear button for that column:

Status	Iter
= Pass	0
Pass	0

### See Also

- [Automated Reporting](#)
- [Enter filter criteria for... Dialog](#)

## Scripting

[Top](#) [Previous](#) [Next](#)

### Purpose

There are three reasons to script with Rapise:

1. To modify a [recorded](#) test to increase coverage, add [assert statements](#), or make the test [data-driven](#).
2. To extend recording functionality by defining your own objects, actions, and libraries.
3. To [customize the Rapise Engine](#).

### Usage

Rapise scripts are written in JavaScript (Microsoft JScript). You can run and debug your script using the full featured [Internal Debugger](#). Rapise includes a testing API, with methods for manipulating images, spreadsheets, common GUI widgets, and more.

### See Also

- Learn about MS JScript [HERE](#).

## Understanding the Script

[Top](#) [Previous](#) [Next](#)

### Purpose

When you [create a new test](#) in Rapise, four files are created:

- **<TestName>.sstest** ? the test meta-data
- **<TestName>.js** ? the test script file
- **<TestName>.objects.js** ? the file that contains recorded objects.
- **<TestName>.user.js** ? the file that contains user defined functions.

where **<TestName>** is the name of your Test.

You can have as many javascript files in your test directory as you like, but **<TestName>.js** is the test script (unless you specify otherwise in the [Settings Dialog](#)). When you record, your interactions are written to **<TestName>.js** and objects are written to **<TestName>.objects.js**; when you Playback the test, **<TestName>.js** is the script that will run. All Rapise test scripts must have the same basic structure.

### Usage

If you are going to modify the script, or create a test script from scratch, you will need to know the test script structure:

#### Basic Script

The Recording tool creates a Rapise Script with three sections:

1. **<TestName>.js: A Test()** function

```

//##### Script Steps #####
function Test()
{
    //script logic
}

```

2. **<TestName>.js: A list of required libraries: `g_load_libraries`**

```
g_load_libraries=["Generic"]; // This script will load the Generic library
```

3. **<TestName>.objects.js A list of learned objects in `saved_script_objects`.**

```
var saved_script_objects={
```

```
//list of objects used in this script ?  
};
```

All Scripts must have the above three sections.

### Full script

The following functions are also recognized by Rapise and may be present in the test script. Put these functions either in `<TestName>.js` or `<TestName>.user.js`.

- **TestInit()** : This function is called once before script playback. It should be used to initialize script-wide data (counters, open datasets, etc).
- **TestFinish()** : This function is called once after test execution. It should be used to release resources (data sets, spreadsheets). TestFinish() is a good place to post-process Reports. It may also be used as an integration point with external test management or bug tracking systems.
- **TestPrepare()** : For advanced users; TestPrepare() is called before recording and before playback. It may be used to properly initialize libraries.

### See Also

To specify a different test script, see the [Settings Dialog](#). The test script is specified by **Settings > ScriptPath**.

## Naming Conventions

[Top](#) [Previous](#) [Next](#)

### Purpose

The Rapise engine and API follow some simple naming conventions.

### Usage

You will find descriptions of the naming conventions below. Note: italicized text represents placeholders.

- **SeS<xxx>** ? public functions for user
- **Do<Action>** ? action implementations
- **\_<somevar>** and **.<somename>** ? private functions and objects
- **g\_<varname>** ? system global data.

## Defining Functions

[Top](#) [Previous](#) [Next](#)

### Purpose

The Rapise test script is in Javascript. You may define as many Javascript functions as you would like to call from your test script.

### Usage

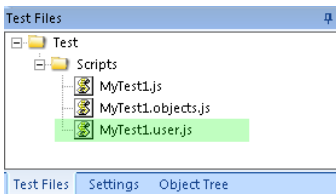
There are two ways to maintain additional functions: (1) Inside your test script and (2) in an external file.

### Inside your Test Script

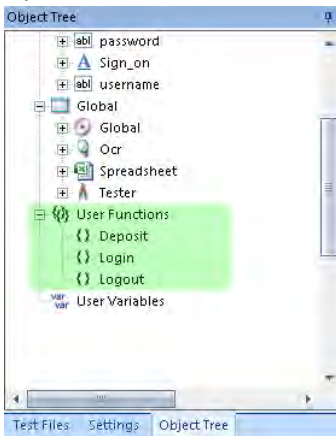
Define the function inside of one of the following functions: **Test()**, **TestInit()**, **TestFinish()**, or **TestPrepare()**. The Script Recorder will erase code placed outside of these functions.

### Inside \*.user.js File

It is recommended to put all user functions into `<testname>.user.js` file available in any test from its creation.



This file is automatically attached into every script. All variables and functions defined in it may be used in the test. User-defined functions are also available under the "User Functions" node in the Object Tree:



### In an External File

You can define your function in another file and include it.

For example:

```
function Test()  
{
```

```

// Withdraw is defined inside the "Test" function
function Withdraw(amount)
{
    Log("Start Withdraw of:"+amount);
    // Withdraw logic is here
}

Withdraw(12.34);

// Include "UtilityFunctions.js" to get at function Deposit()
eval(g_helper.Include(Global.GetFullPath("UtilityFunctions.js")));
// Deposit is defined in "UtilityFunctions.js"
Deposit(56.78);
}

```

### See Also

- To learn more about what the Script Recorder will change in your test script, see [Multiple Recordings](#).

## Global Variables

[Top](#) [Previous](#) [Next](#)

### Purpose

**Global variables** are variables that can be accessed anywhere in the script. There are restrictions (specific to Rapise) as to where they may be placed in the test script. These restrictions do not apply to any additional script files you write and then call from your test script.

### Usage

Define your global variables in **TestInit()**. Because Rapise uses javascript, you can initialize global variables inside of functions. See the sample TestInit() below.

```

function TestInit()
{
    number_of_visited_links = 0; //This variable becomes global
    var local_var = 5; //This variable is local for Testinit function
}

```

The keyword **var** gives variables local scope. A variable initialized without the keyword **var** will have global scope.

The **Script Recorder** knows about the following functions: **Test()**, **TestInit()**, **TestPrepare()**, and **TestFinish()**. Do not declare global variables outside of one of the preceding four functions. The Script Recorder alters the script each time it is run, and may erase your changes.

### See Also

- See [Making Multiple Recordings](#) for details on what effect the script recorder will have on your test script.
- For details on the structure of the test script, see [Understanding the Script](#).

## Including other Files

[Top](#) [Previous](#) [Next](#)

### Purpose

The **eval** keyword lets you use external functions and data structures in your test script; **eval** is a javascript reserved word.

### Usage

See the example below:

```

function Test()
{
    eval(g_helper.Include(Global.GetFullPath("myfunctions.js")));
}

```

### See Also

- [Understanding the Script](#)

## Regular Expressions

[Top](#) [Previous](#) [Next](#)

### Purpose

A **regular expression** is a sequence of characters that describes how to construct a set of strings. It is composed of character literals and special characters. Each character literal represents one single character (such as "a", "b", "C", "1"). The special characters can represent a character, many characters, or a choice about how to select characters.

#### Special Characters:

Char	Description	Examples
?	Combines with whatever character/sub-expression precedes it to represent 0 or 1 occurrences of that character/sub-expression.	<b>a?</b> describes the set: { "", "a" }
*	Combines with whatever character/sub-expression precedes it to represent 0 or more occurrences of that character/sub-expression.	<b>a*</b> describes the set: { "", "a", "aa", "aaa", "aaaa", "aaaaa", "aaaaa", ... }
+	Combines with whatever character/sub-expression precedes it to represent 1 or more occurrences of that character/sub-expression.	<b>a+</b> describes the set: { "a", "aa", "aaa", "aaaa", "aaaaa", "aaaaa", ... }
.	Any arbitrary character.	<b>.*</b> describes the set of all possible strings.
	Denotes a choice between two strings	<b>ab ba</b> describes the set: { "ab", "ba" }
()	Denotes a sub-expression.	<b>(abc)?d</b> describes the set: { "abcd", "d" }
[]	Denotes one character chosen from all the characters with the brackets. You can use a hyphen to denote a range.	<b>[abcde]</b> describes the set: { "a", "b", "c", "d", "e" } <b>[A-Z]</b> describes the set of all one-character, alphabetic, capitalized, strings. { "A", "B", "C", ... , "Z" }

<b>{n,m}</b>	Quantifier expression. Meaning: "Between n and m occurrences of whatever sub-expression or character precedes."	<b>(abc){1,2}</b> describes the set: {"abc", "abcabc"}
<b>^</b>	The beginning of a string.	<b>^a.*</b> matches all strings that begin with an a.
<b>\$</b>	The end of a string.	<b>.*a\$</b> matches all strings that end with an a.
<b>\</b>	Precedes a special character to take away any special meaning.	<b>[\\\$!-+*]</b> represents the set: {"\\", "\$", "!", "-", "+", "*"}

A string and regular expression **match** if the string is an element of the set described by the regular expression.

## Usage

In Rapise, you must prepend regular expressions with the string **"regex:"**. So the regular expression describing all strings would be: **regex: .\***

There are three uses for regular expressions in Rapise: (1) in [Object Locators](#), (2) in [action overriding code](#), (3) in [Custom Libraries](#).

## Assertions

[Top](#) [Previous](#) [Next](#)

### Purpose

An **assert statement** is a special Boolean condition that represents an assumption about program state at a particular point in test execution. When an assert is encountered, the condition is evaluated. A value of **False** indicates a program error. In some languages, execution will halt if an assertion evaluates to **False**. In Rapise, the result is logged to the report with failed status, and execution continues.

### Create a Checkpoint

To create a [checkpoint](#) using an assertion, you will have to manually alter the test script (another way is to use the [Verify Object Properties](#) dialog during [Recording](#)):

1. **Select a location** in your script and a subset of application state to check.
2. **Query for the application state**. For images, use the `ImageWrapper` class provided with Rapise. For object properties, `Get<.>` methods. For example:

```
var xx = SeS(?OkButton?).GetX(); // X position of the object
```

3. **Save the state**. If you are creating an image checkpoint, you will want to save the image to a file. If you are looking at text data, you could use a database, spreadsheet or text file. The `SeSSpreadSheet` class gives you access to excel spreadsheets.
4. **Compare**. Use the `ImageWrapper` class to compare images; use `Spreadsheet` to read and compare spreadsheet data.
5. **Write an Assert Statement**. Make an appropriate call to `Tester.Assert` method. Besides a Boolean condition, pass additional data to be placed in the [Report](#).

Read about [Tester.Assert syntax](#) in the Rapise Objects documentation part.

### See Also

- The [test samples](#) include a `UsingImageCheckpoint.sstest`
- [Verifying Object Properties](#)
- [Writing to the Report](#)

## Data Driven Testing

[Top](#) [Previous](#) [Next](#)

### Purpose

**Data Driven Testing** is an automated testing technique in which test case data is separated from test case logic. Each set of test case data consists of input values and a set of expected output values. The actual output values are compared to the expected output values to determine whether the test passed.

### Usage

The [Spreadsheet](#) object is useful for implementing data-driven tests. It allows you to connect to, query, and read an excel spreadsheet from your test script. To create a data-driven test, you will:

1. **Record a test**. The exact inputs you use for the recording will not matter as much as your interactions with the objects. The following excerpt was recorded using [www.google.com](http://www.google.com):

```
function Test()
{
  //Set Text Inflectra in q
  SeS('Obj1').DoSetText("Inflectra");
  //Click on btnG
  SeS('Obj2').DoClick();
}
```

The actions recorded were: (1) Type **Inflectra** into the search box. (2) Press the **Google Search** button.

2. **Parameterize the Test() function**. The `Test()` function has all of the procedural logic for the test. Replace input values with variables. Encapsulate the logic in a nested function with one parameter for each variable you created. As an example, we will parameterize the `Test()` function we created in step one:

```
function Test()
{
  function Logic(searchterm){ //our new function encapsulates the test logic
    //Set Text using searchterm
    SeS('Obj1').DoSetText(searchterm) //here we changed a hard-coded value into a variable
    //Click on btnG
    SeS('Obj2').DoClick()
  }
  Logic("Inflectra") //don't forget to call your new function
}
```

3. **Create the test case data**. In an excel spreadsheet, create a column for every variable in step two. Add columns for any expected output values you wish to verify. Each row is a test case.

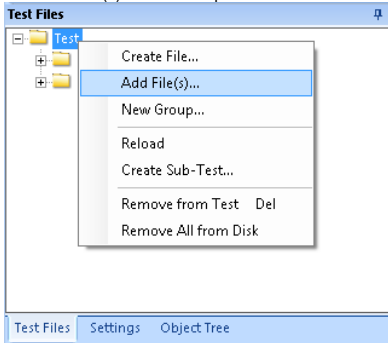
In our google example, we only have one input value (`searchterm`) and we're not comparing any expected output values, so we will only need one column in our spreadsheet. Save the spreadsheet in the test folder as `searchterms.xls`:



	A
1	SpiraTest
2	SpiraPlan
3	SpiraTeam
4	Rapise
5	Remotelaunch

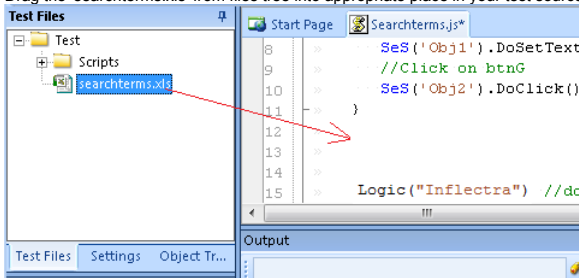
#### 4. Add spreadsheet to the test

Use "Add File(s)..." to add a spreadsheet to the test files:



#### 5. Attach Spreadsheet object to searchterms.xls

Drag the 'searchterms.xls' from files tree into appropriate place in your test source:



#### 6. Use Spreadsheet to access the test case data.

In our example, we use a [Spreadsheet](#) object and run the test logic once for every row.

```
function Test()
{
    function Logic(searchterm)
    {
        //Set Text searchterm in q
        SeS('Obj1').DoSetText (searchterm)
        //Click on btnG
        SeS('Obj2').DoClick()
    }

    Spreadsheet.DoAttach('searchterms.xls', 'Sheet1');

    // Go through all rows
    while(Spreadsheet.DoSequential())
    {
        // Read cell value from column 0
        var term = Spreadsheet.GetCell(0);
        // Pass it into Logic function
        Logic(term);
    }
}
```

## Customizable Engine

[Top](#) [Previous](#) [Next](#)

### Purpose

The source for most of the Rapise implementation is available for you to read and modify. You may find it useful to look at if you decide to create a [library](#) customized for your application.

### Usage

Unless you specified otherwise, Rapise will be installed at

`C:\Program Files\Inflectra\Rapise\`

The source code is in the **Engine** directory. You'll find the [recording/learning](#) libraries in **EngineLib**. The core logic is in four files: **SeSAction.js**; **SeSBehavior.js**; **SeSCommon.js**; **SeSConfig.js**.

If you plan to make changes to the Rapise Engine, we recommend you use a version control system capable of reconciling code conflicts, as we do not support user customizations. However, let us know if you feel that your customizations are generally useful; if we decide to integrate them into Rapise, we will support them.

### See Also

- [Custom Libraries](#)
- [Scripting](#)

## Javascript IDE

[Top](#) [Previous](#) [Next](#)

### Purpose

The **Javascript IDE** includes an [editor](#) and a [debugger](#).

## Usage

Simply [open a script](#) to view the editing features; create a [breakpoint](#) and [play](#) the script to view the [debugging features](#).

## See Also

- Learn about MS Jscript [HERE](#).

## Internal Debugger

[Top](#) [Previous](#) [Next](#)

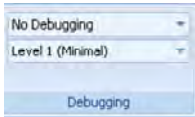
### Purpose

The Internal Debugger provides [Persistent Breakpoints](#), [Control Execution](#), a [Watch View](#), a [Variable/Call Stack View](#), and [Tooltips](#).

### Usage

To use the internal debugger, you must first install [Microsoft Script Debugger](#).

You can choose the Internal Debugger on the Rapise Ribbon (**Test** tab > **Debugging** menu).

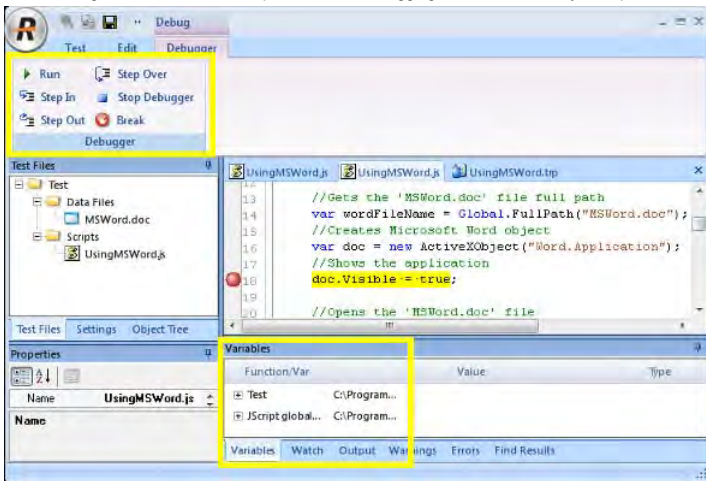


The top drop-down menu has four options. Choose the **Run with Internal Debugger** option.

When you [Playback](#) your test script with a breakpoint, the debugging related menus and views will appear:

- The [Debugging](#) tab of the Ribbon
- The [Watch View](#) and [Variable/Call Stack View](#)

The following screenshot shows the placement of Debugging related functionality in Rapise:



## See Also

- You can use the [External Debugger](#) to debug your scripts as well.

## Tooltips

[Top](#) [Previous](#) [Next](#)

### Purpose

**Tooltips** let you view a variable's value during debugging.

### Usage

1. Put a breakpoint in the script at or near where you wish to investigate
2. Mouse over variables as you advance through the script. A small box will popup, displaying the variables' values:

```
var y=x;  
x = 13
```

## See Also

- [Breakpoints](#)
- [Internal Debugger](#)

## Control Execution

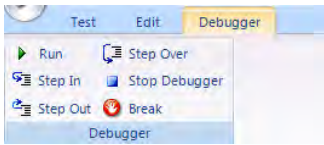
[Top](#) [Previous](#) [Next](#)

### Purpose

**Control Execution** allows you to manually direct the execution of the script.

### Usage

1. Set a [Breakpoint](#) where you want to take control of the execution
2. Use the buttons on the **Debugger** tab of the Ribbon to step through the script.



#### See Also

- [Ribbon: Debugger](#)

## Breakpoints

[Top](#) [Previous](#) [Next](#)

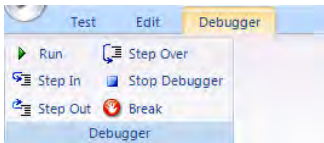
#### Purpose

**Breakpoints** stop execution of the test at a specific line in the script. They allow you to investigate program state, and trace execution flow.

#### Usage

To set a **Breakpoint**:

1. Open the script you would like to debug in the [Source Editor](#).
2. Place the cursor at the line where you want a breakpoint.
3. Press **F9** or the **Break** button on the Ribbon (**Debugger** tab).



#### See Also

- [Ribbon: Debugger](#)
- [Control Execution](#)

## External Debugger

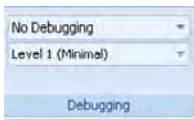
[Top](#) [Previous](#) [Next](#)

#### Purpose

When you enable the **External Debugger**, the **Microsoft Script Debugger** is used to debug your script. Rapise provides an [Internal Debugger](#) as well.

#### Usage

You can enable the External Debugger on the Rapise Ribbon (**Test** tab > **Debugging** menu).



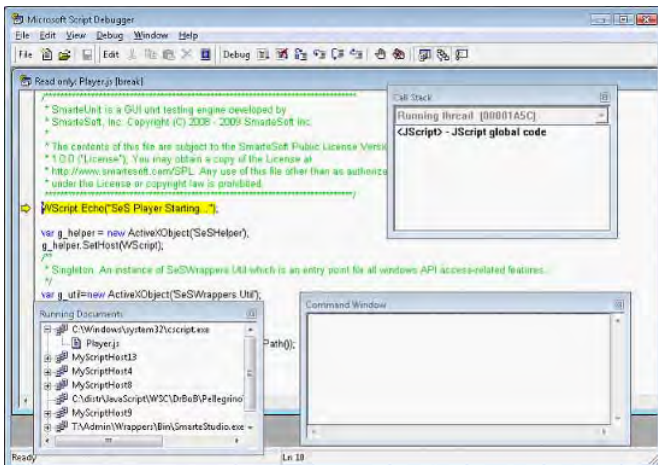
The top drop-down menu has four options:

- **No Debugging**
- **Run with Internal Debugger:** See [Internal Debugger](#) for more info.
- **Run with External Debugger:** Open the Microsoft Debugger to run the script.
- **Run External Debugger on Error:** Open the Microsoft Debugger only if an error occurs.

When you choose the **Run with External Debugger** option, Microsoft Script Debugger will open as soon as you begin [Playback](#) of your script. The debugger will pause on the line

```
WScript.Echo("SeS Player Starting...")
```

and display an error message. There is no actual error; you can begin debugging. Note, however, that Rapise is mostly written in javascript, and the Debugger will step through Rapise implementation as well as your test script.



#### See Also

- [Internal Debugger](#)
- For instructions on using the Microsoft Script Debugger, try this link: <http://msdn.microsoft.com/en-us/library/ms532989.aspx>

## Verbosity Levels

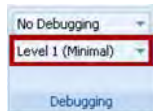
[Top](#) [Previous](#) [Next](#)

### Purpose

The **Verbosity Level** affects the amount of information written to the [Output View](#).

### Usage

The Verbosity Level is set on the Ribbon (**Test** tab > **Debugging** menu). See below:



### See Also

- [Ribbon: Test Tab](#)

## Syntax Highlighting

[Top](#) [Previous](#) [Next](#)

### Purpose

With **Syntax Highlighting**, words in a program are displayed so as to immediately indicate their function. Reserved words, variables, literals, and comments may be differentiated by color, boldness, underline etc. Syntax Highlighting makes programs easier to read and modify.

### Usage

Every javascript file opened in Rapise will display with **Syntax Highlighting**.

### See Also

- [Source Editor](#)

## Code Folding

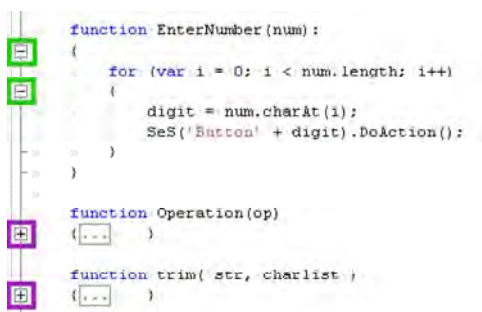
[Top](#) [Previous](#) [Next](#)

### Purpose

**Code Folding** allows you to hide or show blocks of code. These blocks have syntactic meaning, such as a function body, a class declaration, a loop, or a comment.

### Usage

Every javascript file opened in Rapise will display with **hide** and **show** buttons to the top left of their corresponding block. In the following screenshot, hide buttons are highlighted with green boxes; show buttons are highlighted with purple boxes:



### See Also

- [Source Editor](#)

## Syntax Checking

[Top](#) [Previous](#) [Next](#)

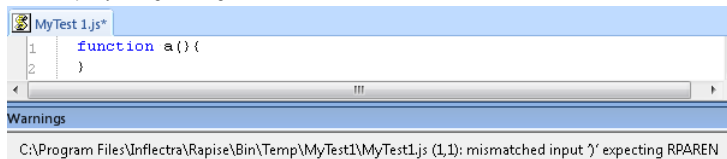
### Purpose

An editor performs **Syntax Checking** if it notifies the user of syntax errors in their program/script.

### Usage

Rapise performs **Syntax Checking** as you type into the [Source Editor](#). Messages regarding syntax errors can be found in the [Warning View](#).

For example, you begin writing a function:



We have a typo here. We used ?? instead of ??. Once the error is corrected, the warning view clears automatically:

```

MyTest1.js
1  function a(){
2  }

```

### See Also

- [Source Editor](#)

## Code Completion

[Top](#) [Previous](#) [Next](#)

### Purpose

Rapise provides **Code Completion** for class, method and field names.

### Usage

Begin typing a class, method, or field name. Press **CTRL+space** to open a list of possible completions.

```

SeS('Editbox').Do

```

### See Also

- [Source Editor](#)

## Unit Testing

[Top](#) [Previous](#) [Next](#)

### Purpose

**Unit Testing** involves testing individual units of a piece of software to make sure they act as intended. The units tested are usually functions or class methods.

### Usage

There are five ways that Rapise can help you Unit Test:

1. Rapise methods support testing objects and methods in [DLLs](#).
2. Rapise can test **ActiveX** objects and their methods through their [COM Interface](#).
3. If you choose to write your Unit tests in a third-party tool, Rapise has a [Command Line](#) interface where you can access its functionality.
4. Test results are written to a [TAP](#) file, which allows integration with Unit Testing frameworks.
5. Rapise tests can be invoked from [MUnit](#) and [NUnit](#) tests.

## DLL Testing

[Top](#) [Previous](#) [Next](#)

### Purpose

You can create objects and invoke methods from both managed and unmanaged dlls.

### Usage

Rapise provides API calls to work with managed DLLs. The Windows object **WScript** can be used with unmanaged DLLs.

#### Managed DLLs

- **Util.InvokeMember**: Invoke a class method in a managed DLL.
- **Util.CreateInstance**: Creates an instance of a class in a managed DLL.
- **Util.SetFieldValue**: Sets a field value in an object created with CreateClassInstance.

#### Unmanaged DLLs

- **WScript.CreateObject("DynamicWrapper")**: Create a DynamicWrapper object. The **Register** and **ShellExecute** methods of the DynamicWrapper object can be used to invoke DLL methods as in the following example:

```

var UserWrap = WScript.CreateObject("DynamicWrapper");
UserWrap.Register("shell32.dll", "ShellExecute", "I=hsssl", "f=s", "x=1");
UserWrap.Register("USER32.DLL", "MessageBoxA", "I=HsSu", "f=s", "R=1");
UserWrap.MessageBoxA( null, "" + elapsed, "Time Elapsed:", 0x30 );

```

#### Test Samples

There is a **Samples** folder in your Rapise directory. There are two [test samples](#) that illustrate working with DLLs:

- UsingDLLHandlerManaged
- UsingDLLHandlerUnManaged

### See Also

- For more information on the WScript object, see: [http://msdn.microsoft.com/en-us/library/at5ydy31\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/at5ydy31(VS.85).aspx)

## COM Testing Support

[Top](#) [Previous](#) [Next](#)

### Purpose

Microsoft's **Component Object Model** (COM) is a standard for communication between separately engineered software components ([source](#)). Any object with a COM interface can be created and used remotely.

### Usage

## Creating a COM Object

You can create a COM object using Windows' **ActiveXObject** class. Once the object is created, method invocation is the same as with any other object in your program. The methods available will depend on the object's COM interface. The following example shows how to create an instance of the Word application and open a file.

```
var doc = new ActiveXObject("Word.Application");
doc.Documents.Open(wordFileName);
```

## Test Samples

There are several [test samples](#) that show how to Unit Test application modules via COM interface:

- UsingMSWord
- UsingMSExcel
- UsingMSAccess

## See Also

- Learn more about COM [HERE](#).
- Learn more about ActiveXObject [HERE](#).

## Custom Strings

[Top](#) [Previous](#) [Next](#)

### Purpose

**Custom Strings** allow you to associate meta data with your test. Each custom string has a **name** and a **value**. The value can be retrieved using the name.

### Usage

#### Adding a Custom String

1. Open the **NameValue Collection Editor** dialog. Instructions are [HERE](#).
2. Press the **Add** button.
3. Fill in a **name** and **value** for the custom string.
4. Press **OK**. The dialog will close.

#### Retrieving a Custom String value

Use the **GetCustomString()** method to retrieve a custom string's value. See the example below:

```
var factory = new ActiveXObject("Rapise.Test.Test");
var test = factory.LoadFromFile( Global.GetFullPath("UsingCustomStrings.sstest"));
var BugID = test.GetCustomString("BugID");
var TestID = test.GetCustomString("TestID");
```

## See Also

- [NameValue Collection Editor Dialog](#)
- There is a [sample test](#) called **UsingCustomStrings**.

## MbUnit

[Top](#) [Previous](#) [Next](#)

### Purpose

SeSMBUnit.vsi is a visual studio installer packaged with Rapise. It facilitates calling Rapise tests from MbUnit tests.

### Usage

#### Installation

- You will need **Visual Studio**, **MbUnit 3**, and **Gallio** to use SeSMBUnit. MbUnit is bundled with Gallio, which is available at [www.gallio.org](http://www.gallio.org).
- To install SeSMBUnit, open the following directory:  
C:\Program Files\Inflectra\Rapise\Extensions\UnitTesting\MbUnit\SeSMBUnit
- Double-click SeSMBUnit.vsi. The **Visual Studio Content Installer** will appear. Select the components for the language you will use and then click **Next**.

#### Syntax

Use both the **MbUnit.Framework** and the **SeSMBUnit** namespaces:

```
using MbUnit.Framework;
using SeSMBUnit;
```

MbUnit uses the class attribute **[Test]** to identify test methods. The corresponding attribute for SeSMBUnit is **[SeSMBUnitTest(@"<path to .sstest>")]**. Note that the SeSMBUnitTest attribute has a parameter, the file-path to the test that will be invoked.

The following example uses a test method simply as a wrapper for calling an **.sstest**:

```
[SeSMBUnitTest(@"T:\Samples\Cross Browser\CrossBrowser.sstest")]
public void TestIEandFirefox()
{
    int exitCode = SeSMBUnitHelper.TestExecute();
    Assert.AreEqual(0, exitCode);
}
```

#### Templates

SeSMBUnit.vsi will install a template for Visual Studio called **SeSMBUnitTests**. The template includes the appropriate **using** statements and a blank test method. You can insert additional **SeSMBUnitTest** methods by right-clicking in the editor in Visual Studio, and selecting **Insert Snippet > SeSMBUnitTest**. The following code will be added:

```
[SeSMBUnitTest(/*Insert path to .sstest file which must be run.*/)
public void TestSeS()
{
    int exitCode = SeSMBUnitHelper.TestExecute();
    Assert.AreEqual(0, exitCode);
}
```

You'll need to specify the file-path.

#### Samples

There is a sample dll you can run in MbUnit. From the Rapise directory, you'll find it at: *Extensions\UnitTesting\MbUnit\SeSMBUnit\SeSSamples\MbUnit\bin\Debug\SeSSamples\MbUnit.dll*

## See Also

- MbUnit and related documentation can be found at [www.mbunit.com](http://www.mbunit.com)

## NUnit

[Top](#) [Previous](#) [Next](#)

### Purpose

SeSNUit.vsi is a visual studio installer packaged with Rapise. It facilitates calling Rapise tests from NUnit tests.

### Usage

#### Installation

- You will need **Visual Studio** and **NUnit** to use SeSNUit. NUnit is available at <http://www.nunit.org/index.php?p=download>.
- To install SeSNUit, open the following directory:  
C:\Program Files\Inflectra\Rapise\Extensions\UnitTesting\NUnit\SeSNUit
- Double-click SeSNUit.vsi. The **Visual Studio Content Installer** will appear. Select the components for the language you will use and then click **Next**.

#### Syntax

Use both the **NUnit.Framework** and the **SeSNUit** namespaces:

```
using NUnit.Framework;  
using SeSNUit;
```

NUnit uses the class attribute **[Test]** to identify test methods. The corresponding attribute for SeSNUit is **[SeSNUitTest(@"<path to .sstest>")]**. Note that the SeSNUitTest attribute has a parameter, the file-path to the test that will be invoked.

The following example uses a test method simply as a wrapper for calling an **.sstest**:

```
[SeSNUitTest(@"T:\Samples\Cross Browser\CrossBrowser.sstest")]  
public void TestIEandFirefox()  
{  
    int exitCode = SeSNUitHelper.TestExecute();  
    Assert.AreEqual(0, exitCode);  
}
```

#### Templates

SeSNUit.vsi will install a template for Visual Studio called **SeSNUitTests**. The template includes the appropriate **using** statements and a blank test method. You can insert additional **SeSNUitTest** methods by right-clicking in the editor in Visual Studio, and selecting **Insert Snippet > SeSNUitTest**. The following code will be added:

```
[SeSNUitTest(/*Insert path to .sstest file which must be run.*/)]  
public void TestSeS()  
{  
    int exitCode = SeSNUitHelper.TestExecute();  
    Assert.AreEqual(0, exitCode);  
}
```

You'll need to specify the file-path.

#### Samples

There is a sample dll you can run in NUnit. From the Rapise directory, you'll find it at: *Extensions\UnitTesting\NUnit\SeSNUit\SeSSamplesNUnit\bin\Debug\SeSSamplesNUnit.dll*

### See Also

- NUnit and related documentation can be found at [www.nunit.org](http://www.nunit.org)

## TAP Results

[Top](#) [Previous](#) [Next](#)

### Purpose

Rapise supports the **Test Anything Protocol (TAP)**. TAP specifies communication between unit tests and testing frameworks, such as [MbUnit](#) or [NUnit](#).

### Usage

The results of a Rapise test are saved to a TAP file in the same directory as the test. Tap files have a **.tap** extension.

### See Also

- More information about tap is available at the TAP wiki: [www.testanything.org](http://www.testanything.org)
- [MbUnit](#)
- [NUnit](#)

## Web Service Testing

[Top](#) [Previous](#) [Next](#)

### What is a Web Service?

A Web service is a unit of managed code that can be remotely invoked using HTTP, that is, it can be activated using HTTP requests. So, Web Services allows you to expose the functionality of your existing code over the network. Once it is exposed on the network, other application can use the functionality of your program.

Web Services allows different applications to talk to each other and share data and services among themselves. Other applications can also use the services of the web services. For example VB or .NET application can talk to java web services and vice versa. So, Web services is used to make the application platform and technology independent.

### What types of Web Service are There?

There are two broad classes of web service:

1. **SOAP** - These 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.
2. **REST** - 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. Typically REST web services expose their operations as a series of unique "resources" which correspond to a specific URL. Each of the standard HTTP methods (POST, GET, PUT and DELETE) then maps into the four basic CRUD (Create, Read, Update and Delete) operations on each resource. REST web services can use different data serialization methods (XML, JSON, RSS, etc.).

### Why do we Test Web Services?

The purpose of **Web Service Testing** is to verify that all of the Application Programming Interfaces (APIs) exposed by your application operate as expected. In some ways they are similar to [unit tests](#) in that they test specific pieces of code rather than user interface objects.

Unlike simple unit tests however, web services tests will normally need to be developed for each of the supported versions of the API so that when a new version of a product comes out, you can

regression test the latest version of the API and all previous versions. This ensures that legacy clients using the older version of the API don't need to make any changes.

Also, unlike unit tests, web services are being called across a network using the HTTP/HTTPS protocol rather than simply calling code that is resident on the same system as the test script. In that sense, they are similar to testing web sites.

Finally, in situations where you have an AJAX web application, as well as testing the front-end user interface using the appropriate UI library, you may need to test the web service that is providing the data to the user interface at the same time. In these situations you have a hybrid, web user interface and web service test.

## Testing Web Services with Rapise

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](#).

## Testing REST Web Services

[Top](#) [Previous](#) [Next](#)

### 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.

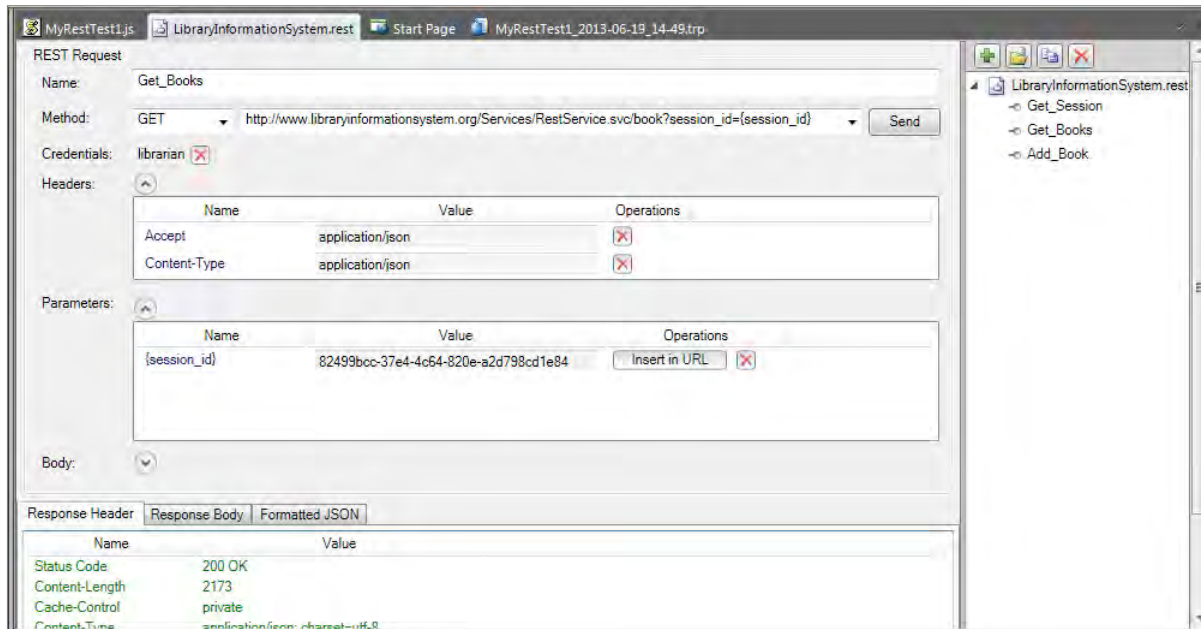
### How does Rapise test REST web services?

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

1. Using the [REST definition 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. Writing the [test script](#) in Javascript that uses the learned Rapise web service objects.

### Rapise REST Definition Builder

When you add a web service to your Rapise test project, you get a new REST definition file (.rest) that will store all of your prototyped requests against a specific REST web service. The various REST requests are then created in the REST definition builder:

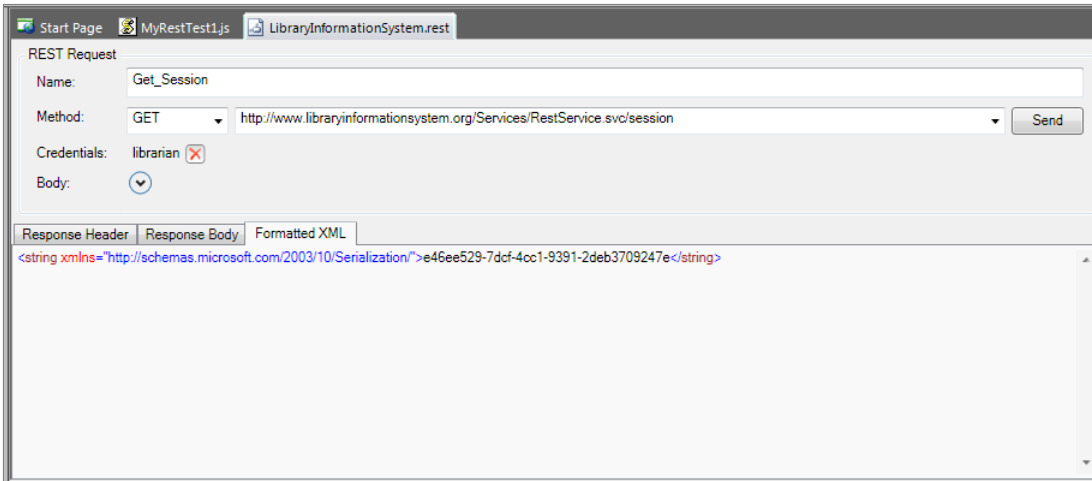


Each REST request can then include the following items:

- **Method** - the type of HTTP request being made (GET, POST, PUT, DELETE, etc.)
- **URL** - the URL of the web service request with any parameter tokens included (e.g. {session\_id} in our example above)
- **Credentials** - Any HTTP Basic Authentication Headers
- **Headers** - Any other HTTP headers (both standard and custom)
- **Parameters** - Any parameters that have been defined in the URL that will be called from the Rapise test script.
- **Body** - The body of the request (for POST and PUT requests). This can be in any text-serialized format such as XML or JSON.

When you execute the request, it will return back the HTTP response headers and if it recognizes the MIME content-type as either XML or JSON, it will format it to make it more readable by the tester:

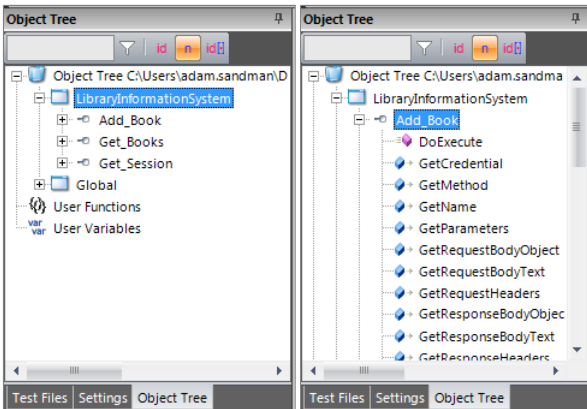




Once you have finished with your prototyping of the web service test operations, you can then save the request definitions and use the 'Update Object Tree' option to populate the main Rapise [Object Tree](#).

### Web Service Object Recognition

Each of the REST web service requests that has been prototyped in the REST definition editor is converted by Rapise into a scriptable object:



Each of the [REST service](#) objects in the tree has operations designed to let you call the method and access the returned body either in its raw text format, or if it's a web service that returns data in JSON format, it will be able to send/receive data as native JavaScript objects.

Rapise provides you with access to the following attributes of the HTTP request before/after the request has been executed:

- **Request:**
  - Method
  - Url
  - Headers (inc. authentication)
  - Body
- **Response:**
  - Headers
  - Body

### Rapise REST Test Scripts

Once all the REST operations have been defined and saved as Rapise learned objects, you can call the REST operations from within your Rapise test scripts:

```
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);

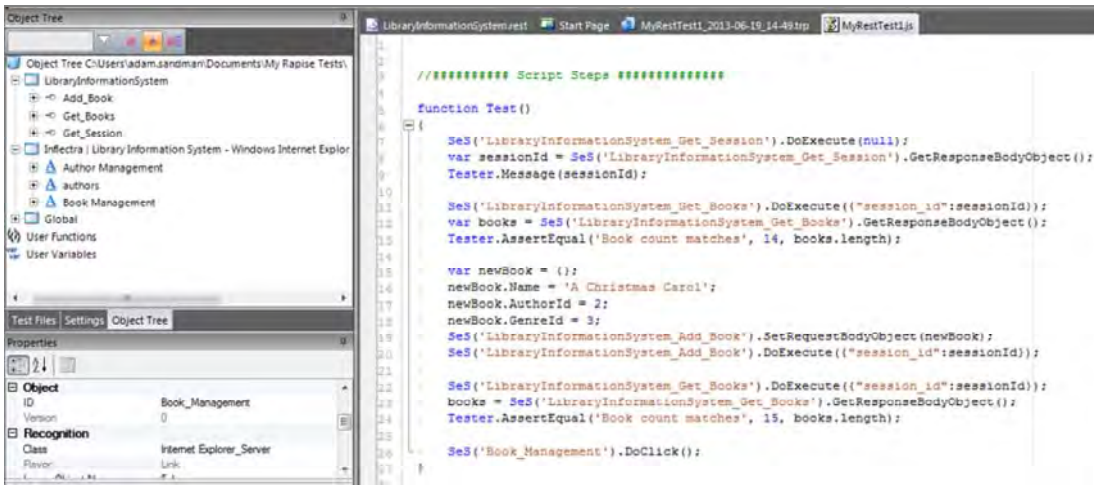
  var newBook = {};
  newBook.Name = 'A Christmas Carol';
  newBook.AuthorId = 2;
  newBook.GenreId = 3;
  SeS('LibraryInformationSystem_Add_Book').SetRequestBodyObject(newBook);
  SeS('LibraryInformationSystem_Add_Book').DoExecute({"session_id":sessionId});

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

As well as simply calling the **DoExecute()** method of each REST web service object to call the previously defined operation, you can use the various properties on the REST service object to send through specific parameter values, add/remove headers, change the authenticated user, change the request body as well as inspect all of the attributes in the request and response.

This allows you unparalleled control over the web service request, with the ability to debug and diagnose web service issues in addition to being able to quickly call the learned operations.

Since the REST objects are just like any other Rapise object, you can have hybrid test scripts that call web service methods and also test GUI objects. This is very useful when you want to test how the user interface changes in response to specific web service API interactions, or when you have a user interface that connects to the sever using a web service (for example with a JSON-based AJAX web user interface).



Once you have created your REST web service test, you can use the standard [Playback](#) functionality in Rapise to execute your test and display the report:

#	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":"c3d8dcd4-6125-4	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":"c3d8dcd4-6125-4	14:49:05.379	Assert	Pass	Returned Value: true	0
	Get_Books.DoExecute({"session_id":"c3d8dcd4-6125-4	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

## SpiraTest Integration

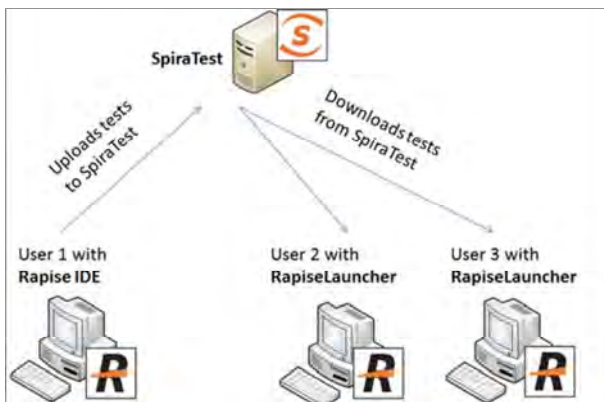
[Top](#) [Previous](#) [Next](#)

For more details on using SpiraTest with Rapise, please refer to the separate ["Using SpiraTest with Rapise"](#) guide.

### Overview

**SpiraTest** is a web-based quality assurance and **test management system** with integrated release scheduling and defect tracking. SpiraTest includes the ability to execute manual tests, record the results and log any associated defects. *Note: SpiraTeam is an integrated ALM Suite that includes SpiraTest as part of its functionality, so wherever you see references to SpiraTest in this section, it applies equally to Spira Team.*

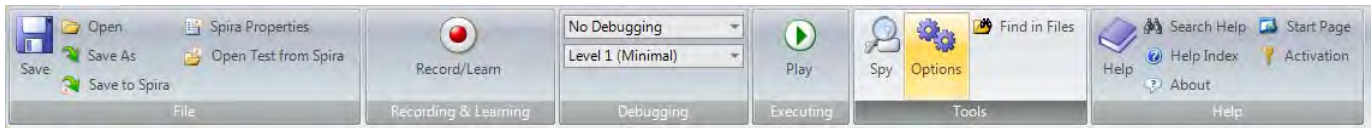
When you use SpiraTest with Rapise you get the ability to store your Rapise automated tests inside the central SpiraTest repository with full version control and test scheduling capabilities:



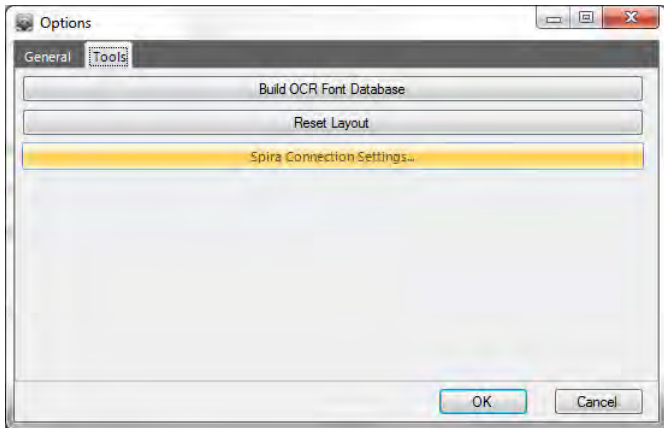
You can record and create your test cases using Rapise, upload them to SpiraTest and then schedule the tests to be executed on multiple remote computers to execute the tests immediately or according to a predefined schedule. The results are then reported back to SpiraTest where they are archived as part of the project. Also the test results can be used to update requirements' **test coverage** and other key metrics in real-time.

### Connecting to SpiraTest

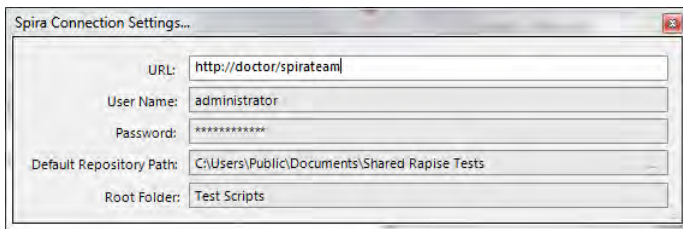
The first thing you need to do is to configure the connection to SpiraTest. To do this, click on the Options button in the Tools section of the Rapise Test ribbon:



This will bring up the Options dialog box. Click on the Tools tab to bring up the settings related to the various Tools:



Click on the "Spira Connection Settings" button to bring up the dialog box that lets you configure the connection to SpiraTest:



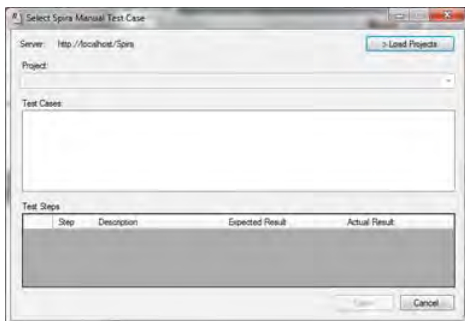
Enter the URL, login and password that you use to connect to SpiraTest and then click the "Test" button to verify that the connection information is correct.

- The **"Default Repository Path"** is a folder that used to store local copies of the non-absolute repositories.
- The **"Root Folder"** is used to specify the name of the top-level folder in the SpiraTest Document Repository that is used to store all the Rapise test scripts. By default this field is blank, meaning that all Rapise test cases will be saved as root-level folders in SpiraTest.

You need to be running SpiraTest / SpiraTeam v3.1 or later to use the integration with Rapise.

### Creating a Rapise test from a SpiraTest test case

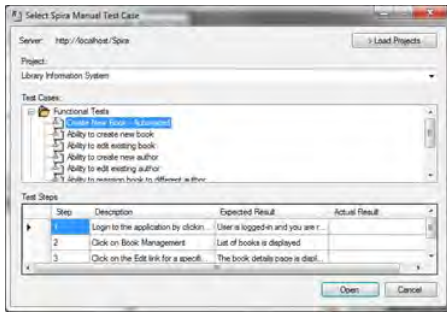
To create a new Rapise test based on the manual test steps already defined in a SpiraTest test case, click on the Rapise icon in the top left of the application and from the popup menu, choose the option "Create From Spira Manual Test". This will bring up the following dialog box:



1. Click on the [Load Projects] button and the dropdown list will display the list of available projects.
2. Select the project that has our new test case. The list of test case folders will be displayed.
3. Expand the folders until you can see the desired test case:



Now click on the test case we previously created and you will see its test steps displayed:



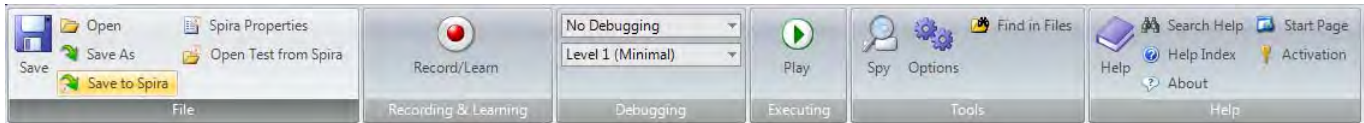
Once you are satisfied that this is the correct test case, click the [Open] button and Rapise will display the normal "Create New Test" dialog box that lets you choose a name and location for this new Rapise test.

### Saving a Test to SpiraTest

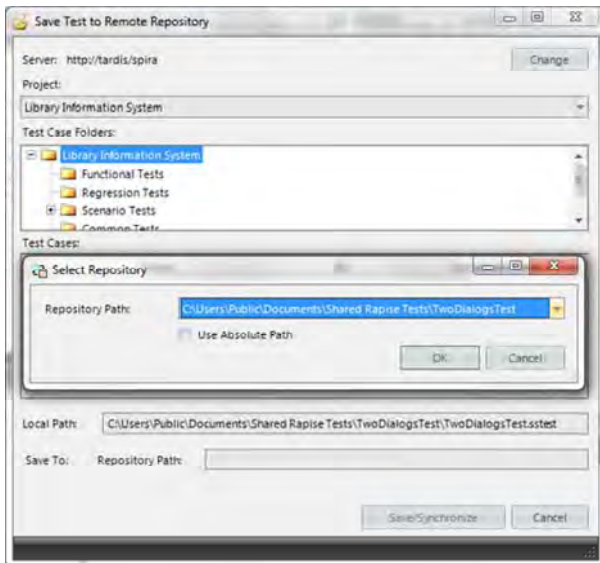
To save the a Rapise test into SpiraTest you need to make sure that the following has been setup first:

1. You have a project created in SpiraTest to store the Rapise tests in. The Rapise tests will be stored in a repository located inside the **Project Home > Documents** section of the project.
2. The user you will be connecting to SpiraTest with has the permissions to **create new document folders**. In SpiraTest v3.1 that means the user needs to be a **Project Owner** role, however in future versions it will not require such a high-priviledged role.
3. You have created the Test Case in SpiraTest that the Rapise test will be associated with. This is important because without being associated to a SpiraTest Test Case, you will not be able to schedule and execute the tests using SpiraTest and RapiseLauncher.
4. You have created an AutomationEngine in SpiraTest that has the token name "Rapise". This will be used to identify Rapise automation scripts inside SpiraTest.

Once you have setup SpiraTest accordingly, click on the **Save to Spira** icon in the File section of the Rapise Test ribbon:

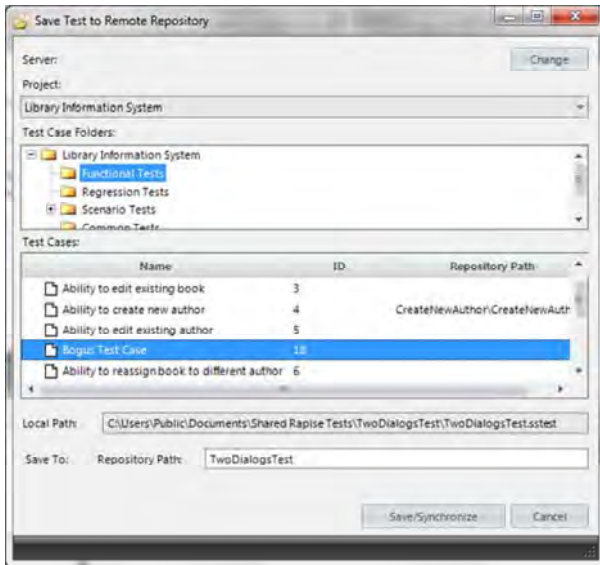


That will bring up the Save to SpiraTest dialog box:



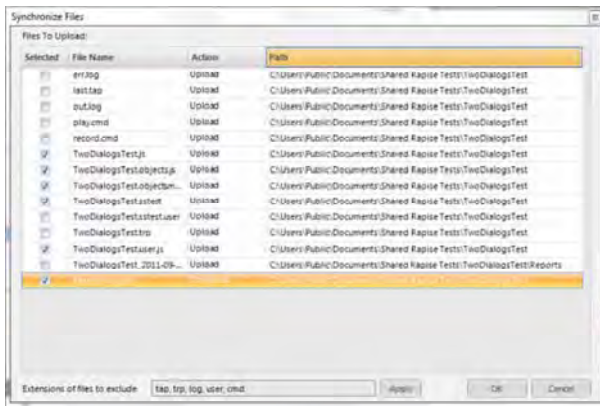
The first thing you will need to do is choose the SpiraTest project from the dropdown list. Once you have done that, the system will ask you to choose the local Repository path for the Rapise test. This is when the test will get saved-to locally before checking-in from SpiraTest. Also it's where the test gets checked-out to if you open the test in the future.

Once you have chosen the repository path, you need to expand the test case folders in SpiraTest and choose the existing Test Case that you want to attach the Rapise test to:



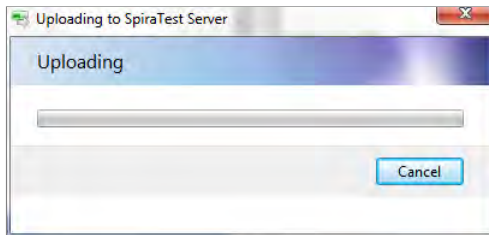
When you expand the folders to display the list of contained test cases, it will display the name of the associated Rapise test script associated with it (to the right). If you are adding a new Rapise test, choose a test case that doesn't have an associated Repository path. If you are updating an existing test, choose a test case that has the matching Repository path.

Once you have chosen the appropriate test case, confirm the path of the test repository folder that will be created in SpiraTest (normally the default is fine). Then click the [Save/Synchronize] button:



A dialog box will be displayed that lists all the files in the local working directory and shows which ones will be checked-in to SpiraTest. The system will filter out result and report files that shouldn't be uploaded. You can change which files are filtered out and also selectively include/exclude files. Once you are happy with the list of files being checked-in, click the [OK] button:

The system will display the message that it's saving the files to the server:



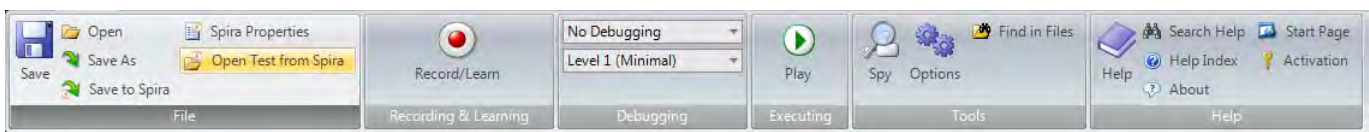
If an error occurs during the save, a message box will be displayed, otherwise the dialog box will simply close.

### Opening a Test from SpiraTest

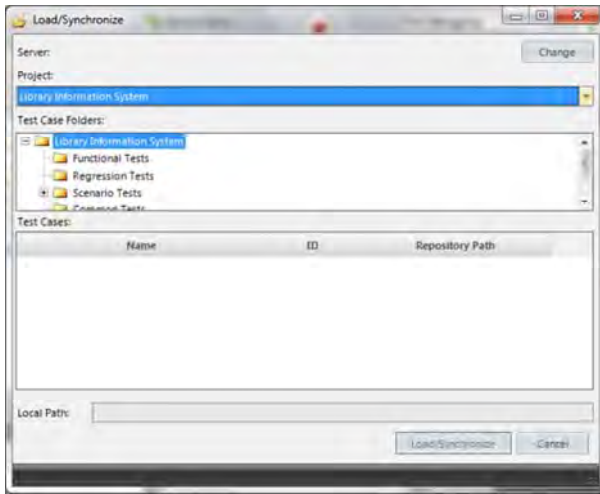
To open a Rapise test from SpiraTest you need to make sure that the following has been setup first:

1. You have already configured the connection to the SpiraTest service (see the instructions at the top of this page).
2. The user you will be connecting to SpiraTest with has the permission to view the project that the tests are being stored in.

Once you have setup SpiraTest accordingly, click on the **Open Test from Spira** icon in the File section of the Rapise Test ribbon:

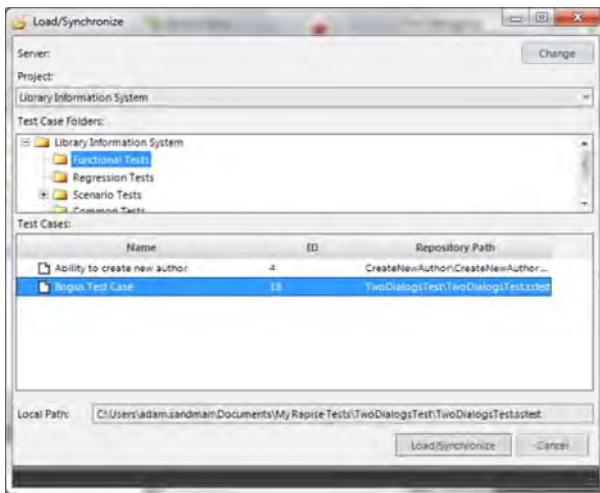


That will bring up the Open Test from SpiraTest dialog box:



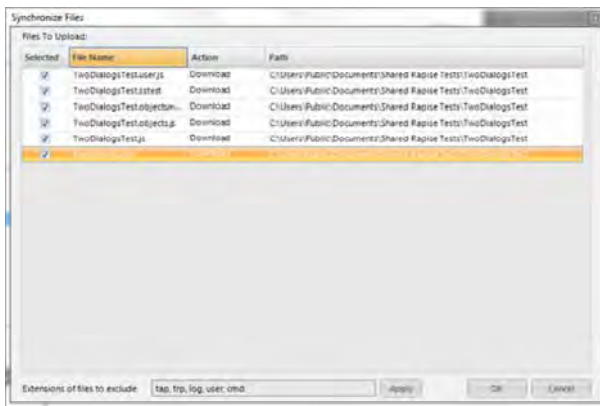
The first thing you will need to do is choose the SpiraTest project from the dropdown list. Once you have done that, the system will ask you to choose the local Repository path for the Rapise test. This is when the test will get saved-to locally after checking-out from SpiraTest.

Once you have chosen the repository path, you need to expand the test case folders in SpiraTest and choose the existing Test Case that you want to attach the Rapise test to:



When you expand the folders to display the list of contained test cases, it will display the name of the associated Rapise test script associated with it (to the right). Choose a test case that has the matching Rapise test case listed to the right of it (in the Repository Path column).

Once you have chosen the appropriate test case, confirm the path of the test repository folder that will be created in SpiraTest (normally the default is fine). Then click the [Load/Synchronize] button:

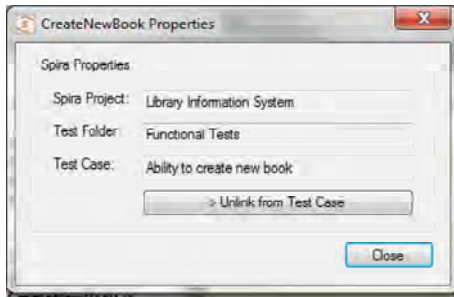


A dialog box will be displayed that lists all the files on the server which will be downloaded from SpiraTest. You can change which files are to be downloaded. Once you are happy with the list of files being checked-out, click the [OK] button:

The system will display the message that it's downloading the files from the server. If an error occurs during the download, a message box will be displayed, otherwise the dialog box will simply close.

### Viewing the SpiraTest Properties of a Test

To see which SpiraTest **project** and **test case** the current Rapise test is associated with, click on the **Spira Properties** icon in the File section of the Rapise Test ribbon. This will bring up the Spira Properties dialog box:



This will display the name of the current Rapise test together with the name of the SpiraTest project, test folder and test case that this test is associated with.

If you would to save the current Rapise test into a new SpiraTest project or if you want to save it against a new test case in the same project, you must first unlink the test. To do this click on the **Unlink from Test Case** button. This will tell Rapise to remove the stored SpiraTest information from the .sstest file so that it can be associated with a new project and/or test case in SpiraTest.

*Warning: This operation cannot be undone so please make sure you really want to unlink the current test.*

### Using RapiseLauncher

**RapiseLauncher** is a separate application that installs with Rapise. It allows you to remotely schedule the automated tests in SpiraTest and have RapiseLauncher automatically invoke the tests according to the schedule. Details on using SpiraTest with RapiseLauncher to remotely schedule and execute tests is described in the separate **"Using SpiraTest with Rapise"** guide. This guide can be found in the Rapise program files folder. Click on Start > Programs > Infectra > Rapise in Windows and you will see the shortcut for the guide.

## Checkpoints

[Top](#) [Previous](#) [Next](#)

### Purpose

A **Checkpoint** is defined by two things: (1) a location in the test execution path and (2) a subset of AUT state. Each time the checkpoint executes, the AUT state is compared to a predefined value. Discrepancies are noted, and may show a regression in program behavior.

### Usage

A checkpoint can be added in two ways:

- (1) during recording, with the [Verify Object Properties dialog](#), or
- (2) by manually adding an [Assertion](#) to the test script.

### See Also

- [Recording](#)

## Tests and Sub-Tests

[Top](#) [Previous](#) [Next](#)

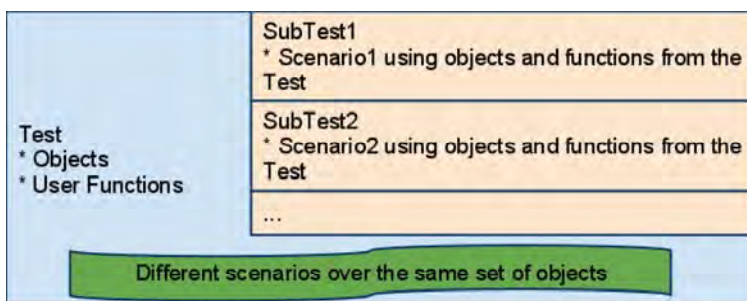
The concept of Sub-Test is an organic way to organize the whole work with Tests in organic way. By having sub-tests one may meet one of the following goals:

1. Create multiple test scenarios working with same set of Objects and Functions.
2. Organize different test scenarios into a single workspace.
3. Use Sub-test to make cross-browser tests

We will consider each of described goals separately. The test containing the sub-test(s) we will call **base** or **parent** test.

### Make Multiple Test Scenarios with the Same Set of Objects

In this case 'parent' test contains all learned objects and user-defined functions.



For example, the parent test may have objects "User Name", "Password", "Sign On". And function

```
function Login(username, password)
{
  ...
}
```

SubTest1 may be used to check login with valid Credentials, **SubTest1.js** looks like:

```
function Test()
{
  Login("validuser", "validpassword");
  // Now check that login is successfull
  Tester.Assert("Login leads to welcome message: ", Global.DoWaitFor('Welcome_User'));
}
```

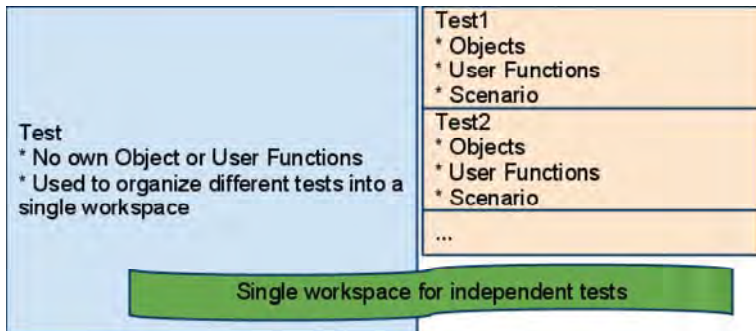
SubTest2 may be used to check login with invalid Credentials (i.e. it is a fail-test). **SubTest2.js** looks like:

```
function Test()
{
    Login("invaliduser", "invalidpassword");
    // Now check that login is successful
    Tester.Assert("Login leads to invalid user object: ", Global.WaitFor('Invalid_User'));
}
```

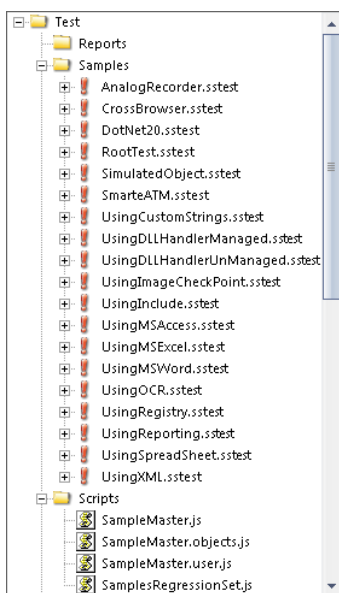
Function `Login` and objects `Welcome_User` and `Invalid_User` are defined in `Test`. The subtests are just implementing various scenarios for the same set of objects.

**Organize different tests into a single workspace.**

Each test has its own objects, functions and scenarios.



The usage of such an approach is well demonstrated by example. We created a test called 'SampleMaster' and put all Rapise samples into it by using **Add File** context menu in the the Test Tree dialog. Finally the Files tree looks like:



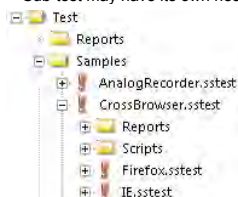
All tests in this tree are independent. We use the Sample Master to manage all the tests from a single environment.

**Using Sub-Tests for a Cross-browser testing**

See 'Cross Browser'.

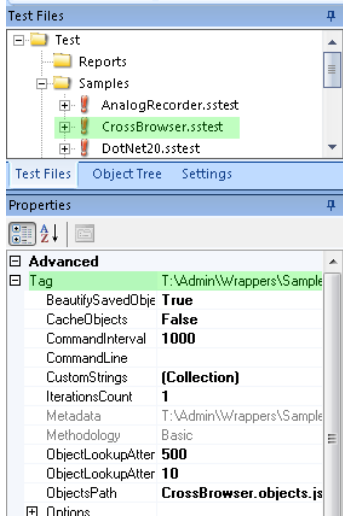
**Sub-Test Features**

- Sub-test may have its own nested sub-tests. For example, in the parent test contains reference to 'CrossBrowser' subtest having 'IE' and 'Firefox' subtests inside:

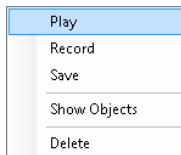


- Sub-test options are available from the 'Tag' property in the 'Properties window':





- The following options are available in the context menu fore each of the sub-tests:



- **Play:** Execute selected sub-test
- **Record:** Start recording into selected sub-test
- **Save:** Save options of a sub-test
- **Show Objects:** Show objects form a sub-test in the Object Tree
- **Delete:** Remove reference to a sub-test from its parent test

## Dialogs, Views, and Menus

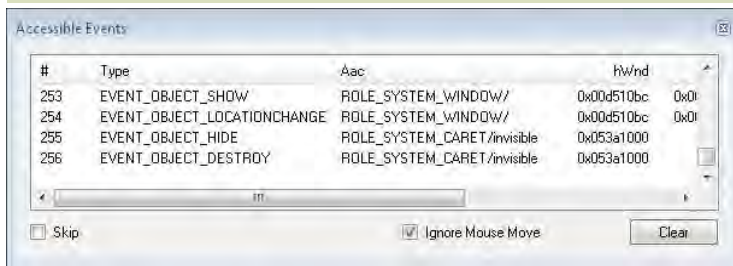
[Top](#) [Previous](#) [Next](#)

This section details the Rapise GUI. Each subsection describes the function of a particular Dialog, View, or Menu. The purpose and consequences of all buttons, options, lists, and check boxes are listed.

### Accessible Events Dialog

[Top](#) [Previous](#) [Next](#)

#### Screenshot



#### Purpose

To display **Microsoft Active Accessibility** event notifications.

#### How to Open

Press the **Monitor Events** button in the [SeS Spy Dialog](#).

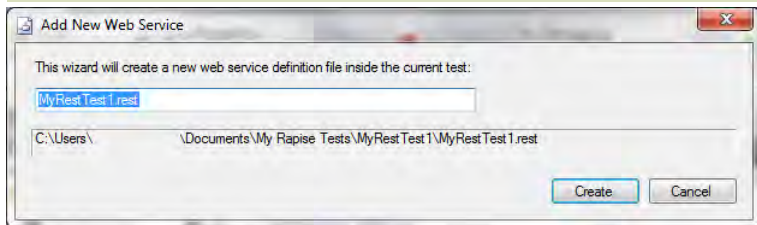
#### Widgets

- **Skip:** While the **Skip** option is selected, the Accessible Events Dialog stops printing event notifications. The number of notifications skipped is printed to the right of the word Skip:
  - Skip (195)
- **Ignore Mouse Move:** Do not print notifications of mouse movement.
- **Clear:** Clear the Accessible Events dialog.

#### See Also

- Microsoft Active Accessibility is described here <http://msdn.microsoft.com/en-us/magazine/cc301312.aspx>

### Screenshot



### Purpose

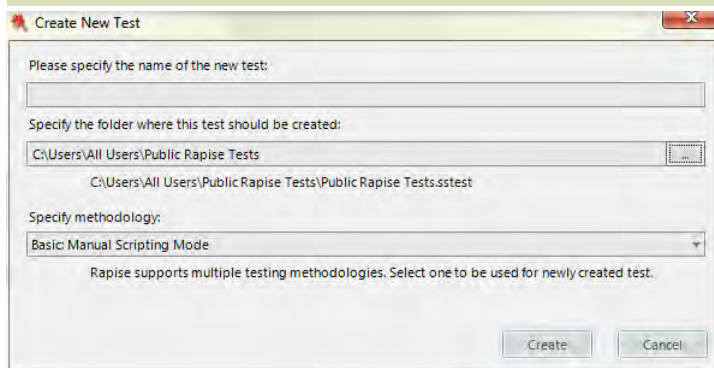
Adds a new REST web service to your Rapise test. It adds the web service as a .rest file that is added to the "Services" folder of the "Test Files" section:

### How to Open

Click on the "Web Services" icon in the Rapise [Test](#) ribbon tab.

## Create New Test Dialog

### Screenshot



### Purpose

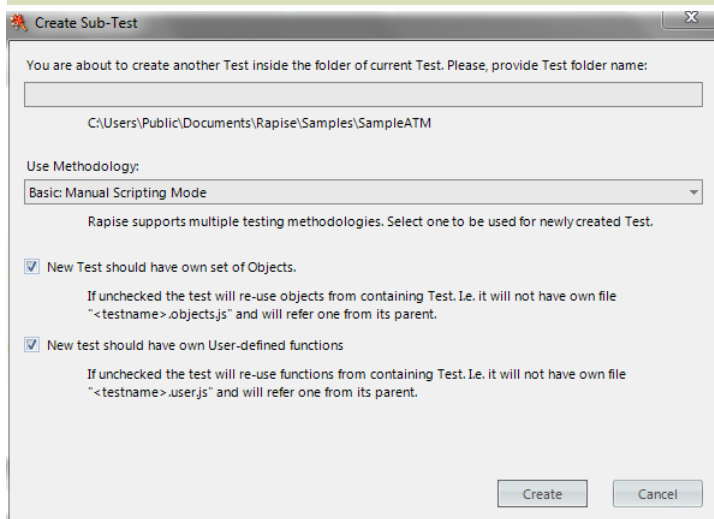
Create a new test.

### How to Open

Menu Button > **New Test**.

## Create Sub-Test Dialog

### Screenshot



### Purpose

Create a [sub-test](#).

- **New test should have own set of Objects:** Uncheck it if you want to create a scenario re-using objects from parent test.
- **New test should have own User-defined functions:** Uncheck it if you want to create a scenario re-using utility functions from its parent test.

The Sub-Test is always created inside the folder of its parent test. If parent test is saved to a new location then sub-test is also saved as a sub-folder of a new location.

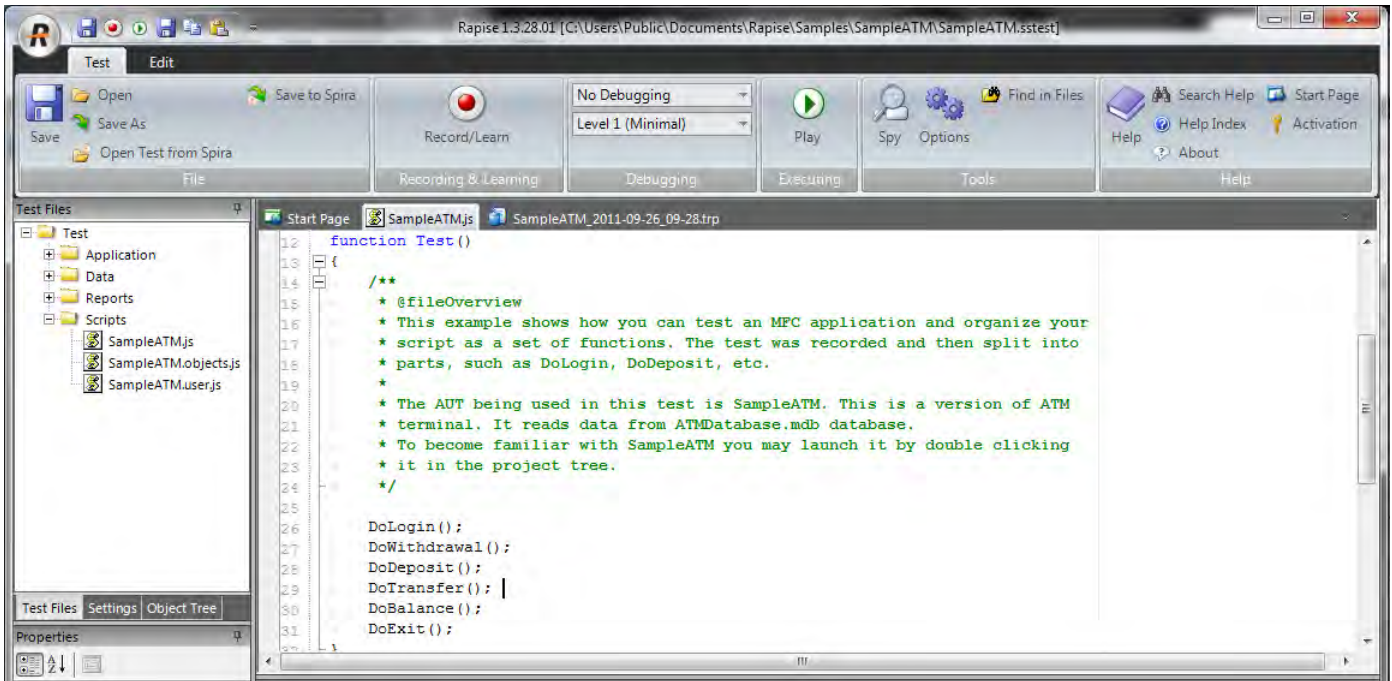
## How to Open

Choose **Create Sub-Test...** in the context menu of a folder in [Test Files](#) dialog.

## Content View

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

To view and edit files. This includes javascript (js), report (trp), and excel (xls) files.

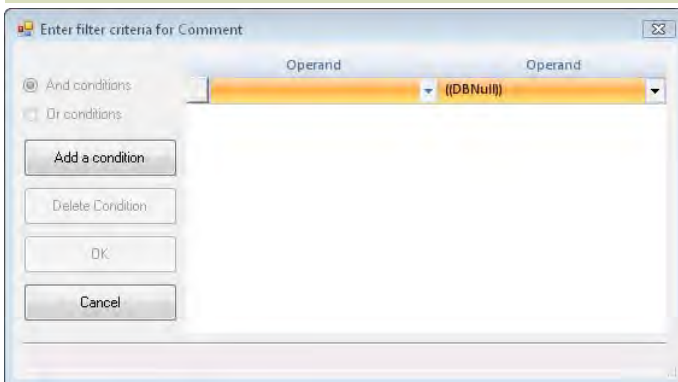
## How to Open

Open a file using the [Test Files Dialog](#). The file will open inside of the **Content View**.

## Enter filter criteria for... Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

Allow more than one [filter criteria](#) for the same column.

## How to Open

In the [Report Viewer](#), open the drop-down menu for one of the [filter cells](#); select the **Custom** option (see below):

ent	Status	It
	(Custom)	0
	(Blanks)	0
	(NonBlanks)	0
ont	Fail	0
	Pass	0

## Conditions

You may specify as many conditions as you like. Each condition has two properties, a **Matching Criteria** on the left and a **filter value** on the right. The **filter value** is a string, and the **matching criteria** specifies what constitutes a match. For more details, look [HERE](#).

## Radio List

And conditions  
 Or conditions

- **And conditions:** All conditions must be true to constitute a match.
- **Or conditions:** At least one condition must be true to constitute a match.

## Buttons

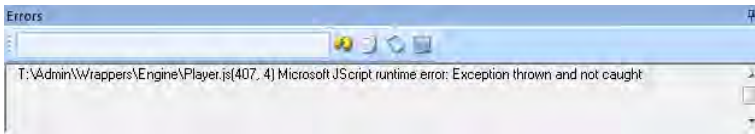
- **Add a condition:** Add an extra condition row.
- **Delete Condition:** Delete the selected condition.  
You can select a condition by pressing the button to the left of it:

- **OK:** Close the dialog and apply the filter.
- **Cancel:** Close the dialog. Do not apply the filter.

## Errors View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

The **Errors View** displays execution error details. Execution errors are those that cause [Recording](#) or [Playback](#) to stop.

### How to Open

The **Errors View** is part of the [Default Layout](#).

### Error Message

T:\Admin\Wrappers\Engine\Player.js(407, 4) Microsoft JScript runtime error: Exception thrown and not caught

Double click on an error message to go to the corresponding source line.

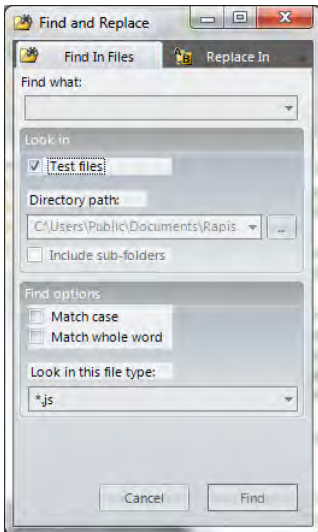
### Widgets

- The text box is a search box.
- The icons from left to right are **Find Next Entry**, **Copy Selected**, **Clear All Text**, and **Select All Text**.

## Find and Replace Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

To find and replace text in files displayed in the Rapise [Content View](#).

### How to Open

Select the **Find in Files** button on the Ribbon (**Test tab > Tools** menu).

### Find in Files Tab

- **Find what:** Place the string you would like to search for in the **Find what** text box.
- If the **Test files** checkbox is checked, the search will take place in the project directory and its sub-folders.
- **Directory path:** Use the Directory Path text-box to specify the directory in which to search. The Directory path text-box cannot be accessed (and is ignored) if the **Test files** checkbox is checked.
- Check the **Include sub-folders** option to search recursively from the directory specified in the **Directory Path** text-box. The Include sub-folders option cannot be accessed if the **Test files** checkbox is checked.
- **Match case option:** If unselected, case is ignored in the search.
- **Match whole word option:** If set to true, parts of words will not count as matches.
- **Look in this file type:** Search only files with the specified file type.

### Find and Replace Tab

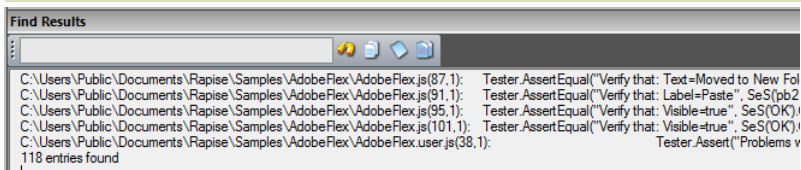
There is only one significant difference between the **Find in Files** Tab and **Find and Replace** Tab: the **Replace with** text-box.

- **Replace with text-box:** All occurrences of the string in the **Find what** text-box will be replaced with the string in the **Replace with** text-box when you press the **Replace** button.

## Find Results View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

Displays results for the [Find and Replace Dialog](#).

### How to Open

The **Find Results** view is part of the [Default Layout](#).

### Messages

C:\Users\Public\Documents\Rapis\Samples\AdobeFlex\AdobeFlex.js(101,1): Tester.AssertEqual("Verify that: Visible=tru

Double click on a message to go to the corresponding source line.

### Widgets

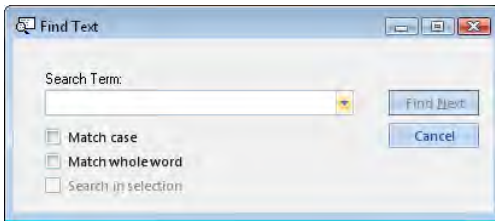


- The text box is a search box.
- The icons from left to right are **Find Next Entry**, **Copy Selected**, **Clear All Text**, and **Select All Text**.

## Find Text dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

Find occurrences of the **Search Term** text in the currently visible [Source Editor](#).

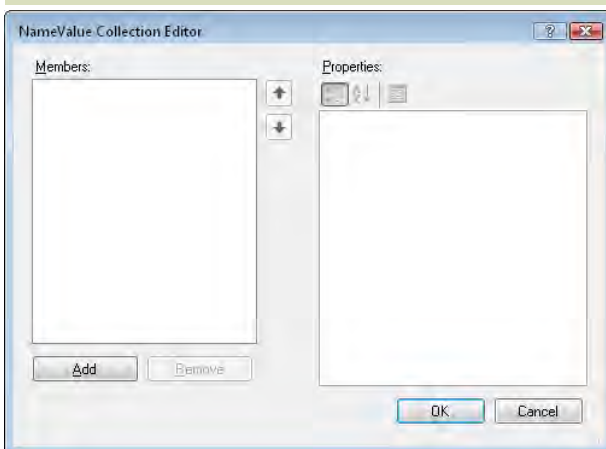
### How to Open

Ribbon > [Edit Tab](#) > **Search** menu > **Find** button 

## NameValue Collection Editor Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot

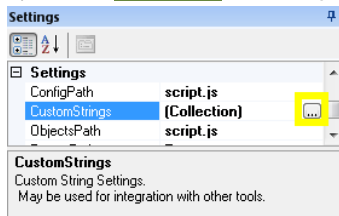


### Purpose

To specify [Custom Strings](#) and their values.

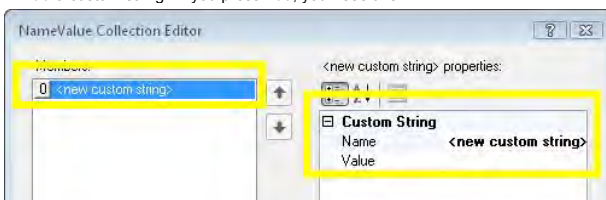
### How to Open

Open from the [Settings Dialog](#), **CustomStrings** option:



### Widgets

- **Add** a custom string. If you press **Add**, you'll see this:

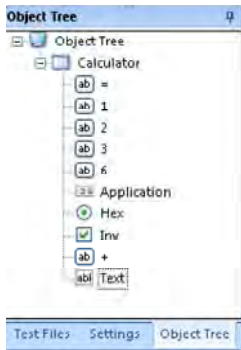


- **Remove**: removes selected custom string.
- **OK**: Save changes and close dialog.
- **Cancel**: Close dialog without saving changes.

## Object Tree Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

Display learned objects.

### How to Open

The **Object Tree** dialog is part of the [Default Layout](#).

### Context Menu (root node)

Right click the **Object Tree** node to see:



- **Refresh** checks for new objects to display.

### Context Menu (object)

Right click on an object in the **Object Tree** dialog to see:

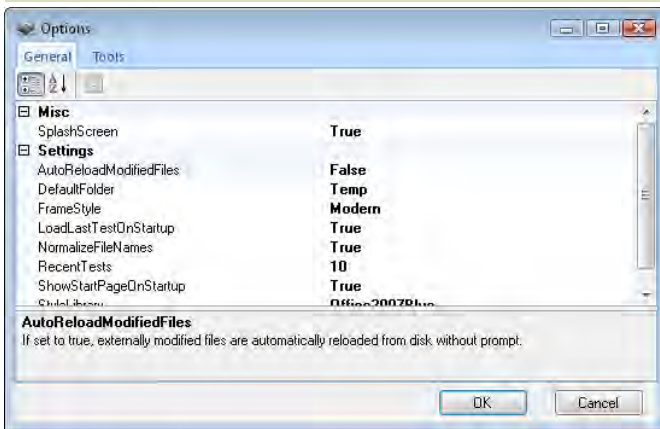


- **Flash** opens the application/url where the object is located. A frame will blink around the object to show you where it is on the page.

## Options Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot

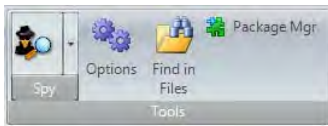


### Purpose

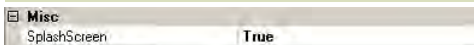
Use the **Options** dialog to change Rapise settings. Your changes will apply to all tests.

### How to Open

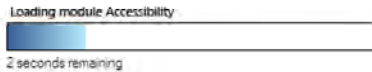
Press the **Options** button on the Ribbon (**Test** tab > **Tools** menu).



### Misc



- **SplashScreen**: A splash screen is the image that appears while a program initializes. The Rapise splash screen looks like this:



1.1.24

Set **SplashScreen** to **False** to prevent the splash screen from appearing.

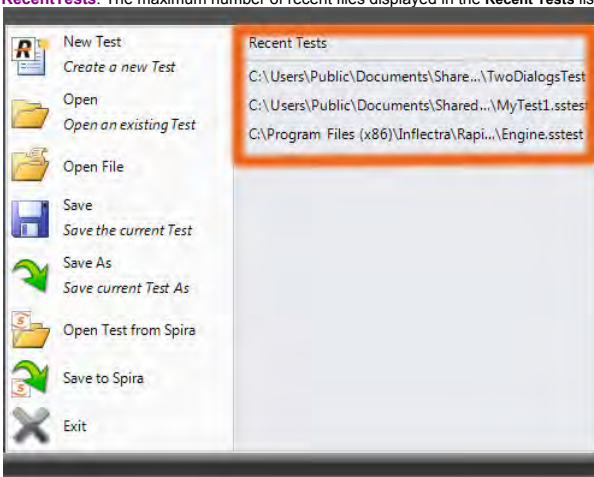
### Settings

Settings	
AutoReloadModifiedFiles	True
DefaultFolder	Temp
FrameStyle	Modern
LoadLastTestOnStartup	True
NormalizeFileNames	True
RecentTests	10
ShowStartPageOnStartup	True
StyleLibrary	

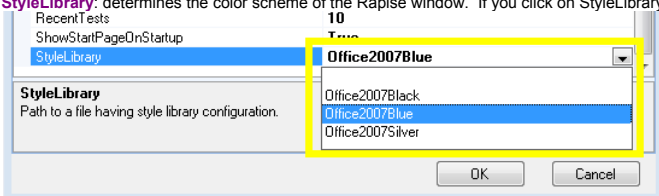
- **AutoReloadModifiedFiles**: If set to **True**, any files you modify outside of Rapise are automatically reloaded in Rapise.
- **DefaultFolder** specifies where new tests are kept before you explicitly save them. The location is relative to the Rapise executable.
- **Enable Execution Monitor** - specifies whether the execution monitor dialog box will be displayed during [playback](#).
- **FrameStyle**: Specifies which frame to draw around objects when you [Record](#), [Learn](#), and [Spy](#). The **Basic** frame is on the left and the **Modern** frame is on the right:



- **LoadLastTestOnStartup**: If set to **True**, Rapise will open the last test you worked on and saved. If set to **False**, Rapise will create a new test named MyTest<#> where <#> is an integer. A folder for MyTest<#> is created in the folder specified by the **DefaultFolder** option.
- **NormalizeFileName**: If set to **True**, files are referred to (in the \*.sstest file) using a path relative to the \*.sstest file. Otherwise, their absolute path is used.
- **RecentTests**: The maximum number of recent files displayed in the **Recent Tests** list. To see the Recent Tests list, open the Application Menu:

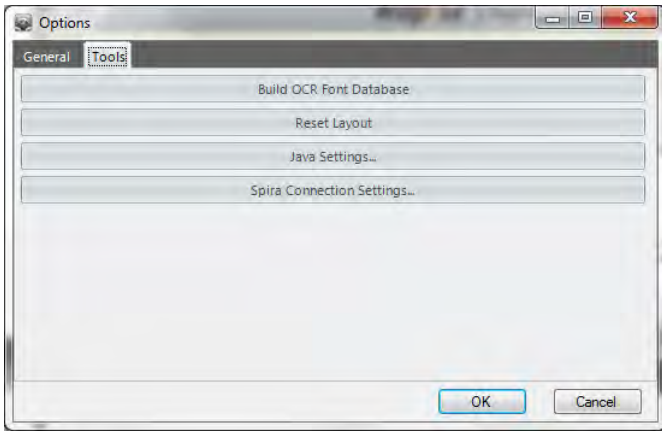


- **Remember Debugger Layout**: If **True**, Rapise will remember the window layout for debug mode separately. For example, this may be useful if you want to work full screen while authoring the Test and half-screen to debug. This way the AUT and the Rapise debugger fit on the screen.
- **ShowStartPageOnStartup**: If **True**, the [Start Page](#) will open automatically when Rapise is opened.
- **StyleLibrary**: determines the color scheme of the Rapise window. If you click on StyleLibrary, you'll notice that a drop down arrow appears to the right. Press the arrow to see all of the Style options:



### Tools Tab



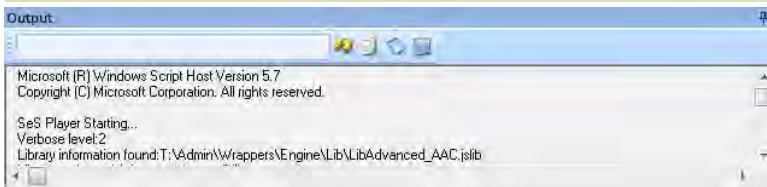


- **Build OCR Font Database:** Pressing the **Build OCR Font Database** button updates the list of screen fonts that Rapise recognizes when using an [OCR](#) object. Whenever you install new Fonts onto the computer you should click this button to have them added to the Rapise font database.
- **Reset Layout:** Pressing the **Reset Layout** button restores the [default layout](#). Rapise will restart.
- **Java Settings:** Pressing the **Java Settings** button displays the [Install Java Access Bridge](#) dialog box. Installing the Java Access Bridge lets Rapise connect to Java AWT/Swing applications so that they can be tested.
- **Spira Connection Settings:** Pressing the **Spira Connection Settings** button takes you to a dialog box that lets you change how Rapise is integrated with the [SpiraTest](#) test management system. It will let you change the URL, username and password used to connect.

## Output View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

The **Output View** displays Rapise output. The amount of output depends on the [Verbosity Level](#).

### How to Open

The **Output** view is part of the [Default Layout](#).

### Writing to the Output View

Use the global **Log()** function to write to the **Output View**.

### Widgets

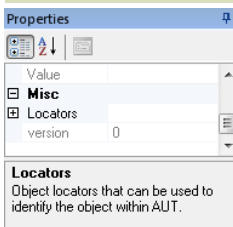


- The text box is a search box.
- The icons from left to right are **Find Next Entry**, **Copy Selected**, **Clear All Text**, and **Select All Text**.

## Properties Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

To display the properties of the object, file, or folder you last clicked on. Objects are in the [Object Tree Dialog](#) and files/folders are in the [Test Files Dialog](#).

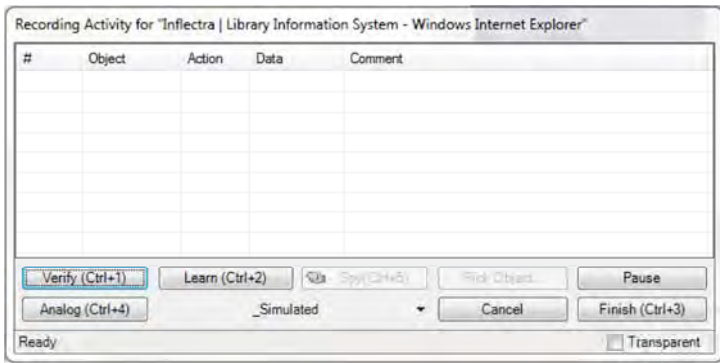
### How to Open

The **Properties** Dialog is part of the [Default Layout](#).

## Recording Activity Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

The **Recording Activity (RA) Dialog** is used for [Recording](#), Analog recording ([absolute](#) and [relative](#)), [Object Learning](#), and creating [Simulated Objects](#).

## How to Open

1. Open the **SAR Dialog**. Instructions are [HERE](#).
2. You must select two things: (1) which recording library to use during the recording session and (2) which process/program to record. Look [HERE](#) for more information on using the SAR Dialog.
3. Press either **Select** or **Run** on the SAR dialog to open the RA Dialog.

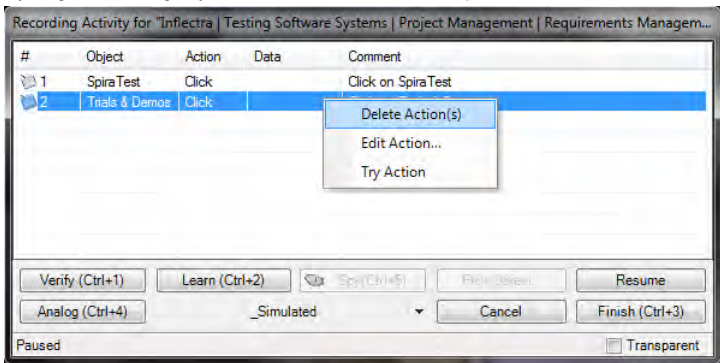
## The Grid

As you interact with the AUT (Application Under Test), your actions are recorded in the grid of the **RA dialog**. The following screenshot shows the RA dialog after two interactions with [www.google.com](http://www.google.com): (1) first, **Inflectra** was entered into the query text box and (2) the **Google Search** button was then pressed.

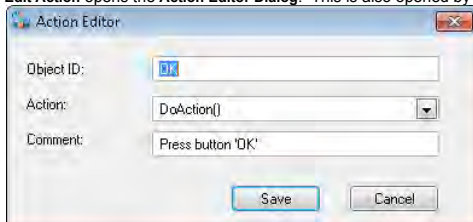
#	Object	Action	Data	Comment
1	q	SetText	Inflectra	Set Text Inflectra in q
2	btnG	Click		Click on btnG

## Context Menu

If you right click in the grid, you'll see a context menu with three options:

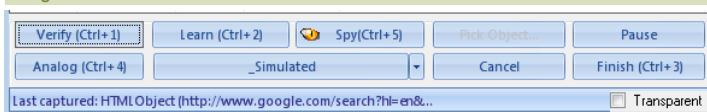


- **Delete Action** removes the selected row.
- **Edit Action** opens the **Action Editor Dialog**. This is also opened by double-clicking a grid entry.



- Press **Try Action** and Rapise will execute the action.

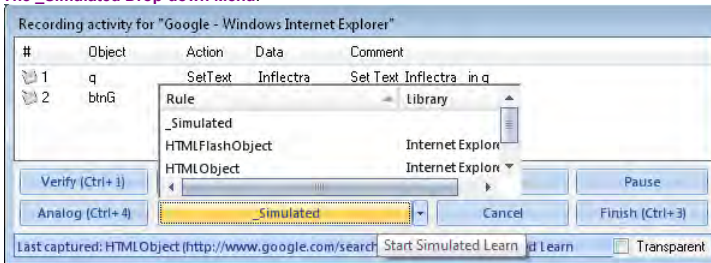
## Widgets



- **Verify**: Press to open the [Verify Object Properties](#) dialog.
- **The Learn Shortcut**: Use to [learn](#) an object. Place the mouse cursor over the object you wish to learn. It should become highlighted with a purple box. Press Ctrl+2 while the object is highlighted. You will see a line added to the RA dialog, signifying that the object was learned.
- **The Spy Button**: The Spy Button opens the **SeS Spy dialog**. The SeS Spy dialog allows you to view the state of the objects in your program. Viewing object state is called [Object Spying](#). The SeS Spy dialog is described [here](#).
- **Pick Object**: Use if the object you wish to learn is invisible (covered by another object). Pick Object is disabled for Internet Explorer and Firefox recording.
  1. The Pick Object button will open the [SeS Spy Dialog](#).

2. Spy on the obstructing object. (Press **Start Tracking**, mouse over the object, press CTRL+G)
3. Select the item you wish to learn from the **Tree** section.
4. Press the **Learn Selected** button.

- **The Pause Button:** The Pause Button on the RA dialog temporarily stops Recording. Any interacting you do with the AUT is ignored. When you press the Pause Button, the title of the button changes to **Resume**. Press the **Resume** button to continue recording.
- **The Analog Button:** The Analog button begins [Analog Recording](#). Analog Recording tracks mouse movements, keyboard inputs, and clicks. To end Analog Recording, press **CTRL+Break**.
- **The Simulated Drop-down Menu:**

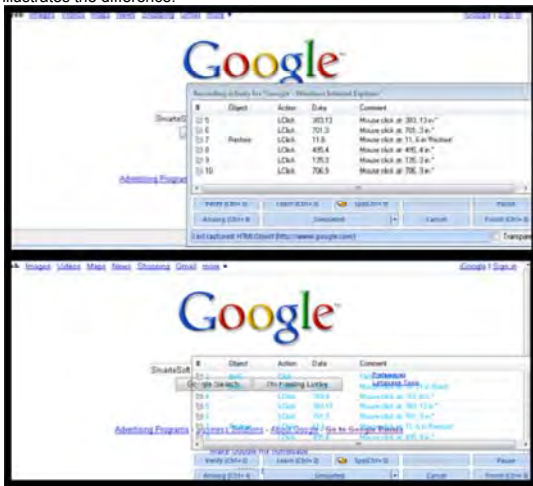


An object can be learned if it matches a rule specified in the [Recording/Learning libraries](#) available. The drop-down menu lists the possible rules for learning objects in the current application. If you cannot learn an object with one rule, try another in the list. Create a **Simulated Object** only if the other, more flexible alternatives have been exhausted.

Learning using a specific rule:

1. Double click on a rule in the drop down list. The button text should change to the text that you selected
2. Press the button
3. Select an object on the screen and make sure it is highlighted with a rectangle
3. Press **Ctrl+2** to learn the object

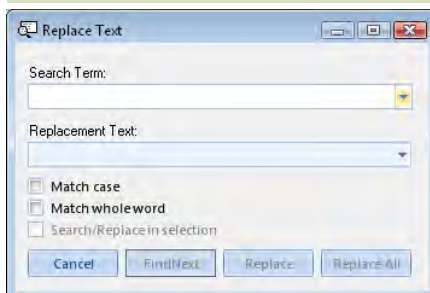
- **The Cancel Button:** The Cancel button stops Recording, closes the RA dialog, and discards any actions recorded or objects learned during the Recording session.
- **The Finish Button:** The Finish button ends the Recording session. The RA dialog is closed, and the information collected during Recording is used to create a script. The script is displayed.
- **Transparent Option:** While the RA dialog is open, it is always on top. The Transparent checkbox makes the RA Dialog transparent so that you can interact with objects behind it. The image below illustrates the difference:



## Replace Text Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

Replace occurrences of the **Search Term** text with the **Replacement Text** in the currently visible [Source Editor](#).

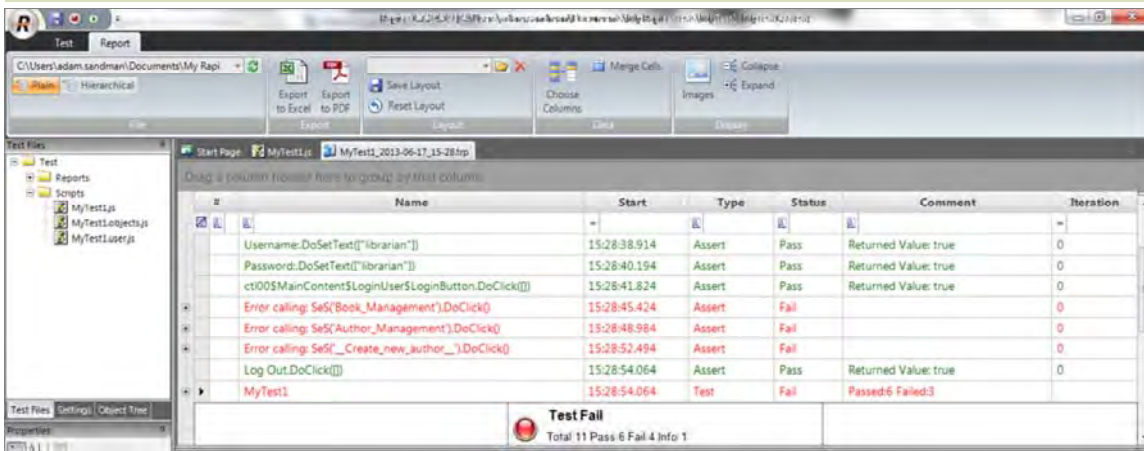
### How to Open

Ribbon > [Edit Tab](#) > **Search** menu > **Replace** button.

## Report Viewer

[Top](#) [Previous](#) [Next](#)

## Screenshot



## Purpose

The **Report Viewer** displays test result (.trp) files.

## How to Open

Use the [Test Files Dialog](#) to open a report (.trp) file. The report file will be opened in a **Report Viewer** in the [Content View](#). The [Report Tab](#) of the Ribbon will also open.

Or, you can [Playback](#) the test script. The report file will display in a **Report Viewer** after the test completes.

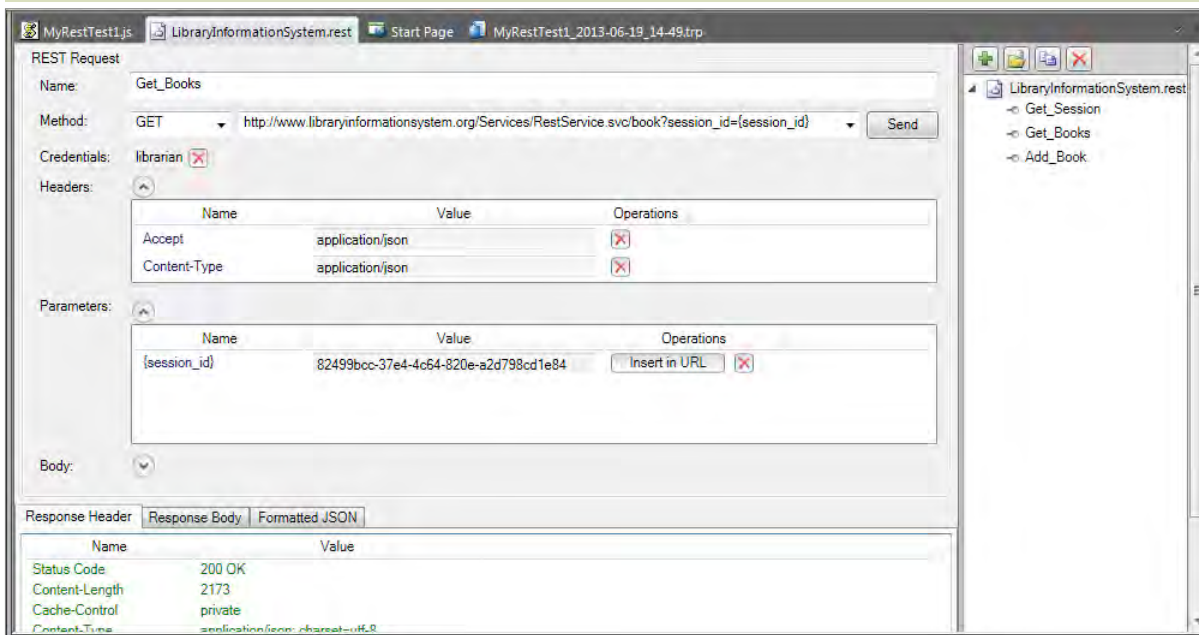
## See Also

- For more info on Reports, see [Automated Reporting](#).
- For information on manipulating reports, see [Ribbon: Report](#).

## REST Definition Editor

[Top](#) [Previous](#) [Next](#)

## Screenshot



## Purpose

The **REST Definition Editor** allows you to edit [REST web service](#) definition files (.rest).

## How to Open

Use the [Add Web Service Dialog](#) to create a new REST definition (.rest) file. The definition file will be opened in a **REST Editor** in the [Content View](#). The [REST Tab](#) of the Ribbon will also open.

Or, you can double-click on an existing .rest file in the [Test Files View](#) explorer window. The definition file will be opened in a **REST Editor** in the [Content View](#). The [REST Tab](#) of the Ribbon will also open.

## Request

REST Request

Name: Add\_Book

Method: POST http://www.libraryinformationssystem.org/Services/RestService.svc/book?session\_id={session\_id} Send

Credentials: librarian X

Headers:

Name	Value	Operations
Accept	application/json	<span>X</span>
Content-Type	application/json	<span>X</span>

Parameters:

Name	Value	Operations
{session_id}	82499bcc-37e4-4c64-820e-a2d798cd1e84	<span>Insert in URL</span> <span>X</span>

Body:

The request form has several sections that you need to populate:

- **Method** - the type of HTTP request being made (GET, POST, PUT, DELETE, etc.)
- **URL** - the URL of the web service request with any parameter tokens included (e.g. {session\_id} in our example above)
- **Credentials** - Any HTTP Basic Authentication Headers
- **Headers** - Any other HTTP headers (both standard and custom)
- **Parameters** - Any parameters that have been defined in the URL that will be called from the Rapise test script.
- **Body** - The body of the request (for POST and PUT requests). This can be in any text-serialized format such as XML or JSON.

## Response

Response Header | Response Body | Formatted XML

Name	Value
Status Code	200 OK
Content-Length	113
Cache-Control	private
Content-Type	application/xml; charset=utf-8
Date	Thu, 20 Jun 2013 18:00:27 GMT
Set-Cookie	ASP.NET_SessionId=3ggghumijkg54n4xb02h
Server	Microsoft-IIS/7.0
X-AspNet-Version	4.0.30319
X-Powered-By	ASP.NET

Response Header | Response Body | Formatted XML

```
<string xmlns='http://schemas.microsoft.com/2003/10/Serialization'>6c8a3e1e-ed0-42d0-bea4-7375604b9175</string>
```

Response Header | Response Body | Formatted JSON

```
"82499bcc-37e4-4c64-820e-a2d798cd1e84"
```

This displays the output from the last web service request. It has several tabs:

- **Response Header** - Displays a list of the HTTP response headers (name and value). If the request received a 200 OK code back, it's displayed in **green**, if it receives an error code back, it's displayed in **red**.
- **Response Body** - Displays the raw text of the HTTP response body received from the server.
- **Formatted XML** - If the received body content is identified as XML, this tab displays nicely formatted XML that is easier to read than the raw response body.
- **Formatted JSON** - If the received body content is identified as JSON, this tab displays nicely formatted, indented JSON that is easier to read than the raw response body.

## Operation Explorer

LibraryInformationSystem.rest

- > Get\_Session
- > Get\_Books
- > Add\_Book

This section lets you add, open, delete and clone REST requests in the definition file.

- **Add request** - Adds a new REST operation to the current .REST definition file
- **Open request** - Opens the currently selected REST operation in the current .REST definition file. This is the same as double-clicking on the item name.
- **Clone request** - Makes a copy of the currently selected REST operation and allows you to give the copy a new name.
- **Delete request** - Deletes the currently selected REST operation from the current REST definition file.

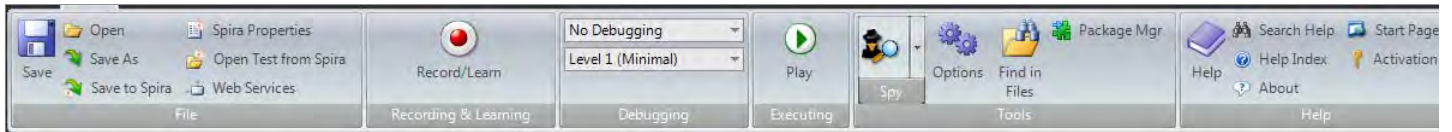
## See Also

- For more info on REST Web Services, see [REST Web Services](#).
- For a tutorial on creating a REST web service test, see the [Web Services REST Tutorial](#).

## Ribbon: Test

[Top](#) [Previous](#) [Next](#)

## Screenshot



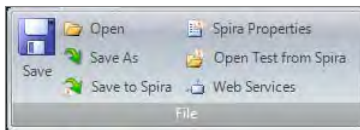
## Purpose

The **Test** tab provides tools to help with creating and executing tests.

## How to Open

The **Test** tab is always available.

## File



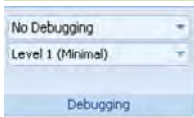
- **Save** the test.
- **Open** a test.
- **Save As** allows you to create a new, differently named copy of the test you are editing.
- **Save to Spira** allows you to save a Rapise test so that it is stored in a [SpiraTest](#) test management repository.
- **Spira Properties** allows you to see the name of the SpiraTest project and test case that the current Rapise test is linked to.
- **Open from Spira** allows you to open a Rapise test that is stored in a [SpiraTest](#) test management repository.
- **Web Services** allows you to add a new [web service](#) definition to your Rapise test. Clicking on this displays the [Add Web Service](#) dialog box.

## Recording and Learning



- Press the **Record/Learn** button to open the [Recording Activity Dialog](#).

## Debugging



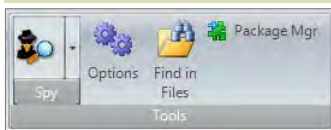
- The top drop-down list specifies if you would like to use an [External Debugger](#). If so, you can either connect on execution (the **Run with External Debugger** option) or only connect if an error occurs (the **Run External Debugger on Error** option).
- The lower drop-down list controls the [Verbosity Level](#).

## Executing



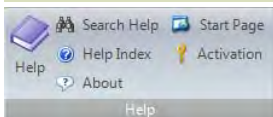
- Press **Play** to execute the test script (\*.js) file associated with the open test. You can change which test script to open in the [Settings Dialog](#). The test script is specified by **Settings > ScriptPath**.

## Tools



- The **Spy** button opens the [Spy Dialog](#).
- Press the **Options** button to open the [Options Dialog](#).
- The **Find in Files** button opens the [Find and Replace Dialog](#).
- The **Package Mgr** button opens the [Package Manager](#) add-in.

## Help

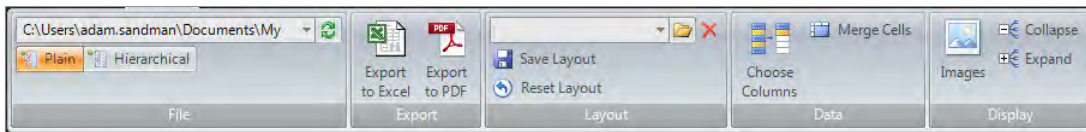


- The **Help** button opens the Rapise user's manual and makes the **Contents** tab visible.
- The **Search Help** button opens the Rapise user's manual and makes the **Search** tab visible.
- The **Help Index** button opens the Rapise user's manual and makes the **Index** tab visible.
- The **Start Page** button opens the Rapise [Start Page](#).

## Ribbon: Report

[Top](#) [Previous](#) [Next](#)

## Screenshot



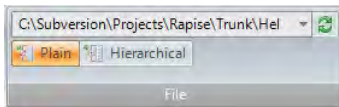
## Purpose

The **Report** tab is for use with report (trp) files.

## How to Open

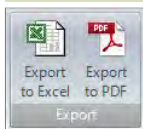
The **Report** tab is available anytime you have a report (trp) file visible in the [Content View](#).

## File



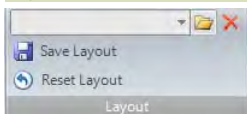
- The drop-down menu contains a history of previously opened reports.
- Press **Plain** to view test steps, assertions, and messages aligned in a table.
- Press **Hierarchical** to more clearly see what assertions, messages, and data are associated with which test steps.

## Export



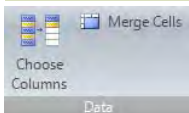
- Press **Export to Excel** to save the report as an excel file.
- Press **Export to PDF** to save the report as an Acrobat PDF file.

## Layout



- The drop-down menu lets you choose between previously saved layouts.
- You must press **Save Layout** to keep your layout changes after closing Rapise.
- Press **Reset Layout** to undo any changes you've made.

## Data



- Press **Choose Columns** to hide or reveal report columns.
- **Merge Cells**: Merge identical consecutive cells.

## Display



- **Images**: Toggle between hiding and revealing images.
- **Collapse**: Collapse the report to show only the top level. What is visible will depend on how the report is sorted.
- **Expand**: Expand all report rows.

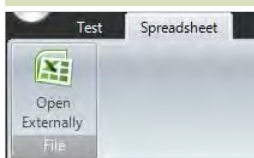
## See Also

- [Automated Reporting](#)

## Ribbon: Spreadsheet

[Top](#) [Previous](#) [Next](#)

## Screenshot



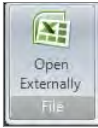
## Purpose

The **Spreadsheet** tab is for use with excel (xls) files.

## How to Open

The **Spreadsheet** tab is available anytime you have an excel (xls) file visible in the [Content View](#).

## File



- The **Reload** button reloads the excel file from disk. Use it if the excel spreadsheet was modified by an external application after you opened it in Rapise.

## Ribbon: Edit

[Top](#) [Previous](#) [Next](#)

## Screenshot



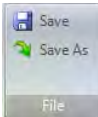
## Purpose

The **Edit** tab of the Ribbon provides tools for editing script files.

## How to Open

The **Edit** tab is available anytime you have a javascript file visible in the [Content View](#).

## File



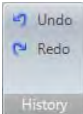
- The **Save** button (Shortcut: CTRL+S) saves the script file you are editing.
- The **Save As** button allows you to create a new, differently named copy of the script file you are editing.

## Clipboard



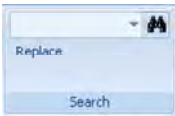
- The **Paste** button (Shortcut: CTRL+V) pastes from the clipboard.
- The **Cut** button (Shortcut: CTRL+X) erases whatever text you have highlighted, and copies it to the clipboard.
- The **Copy** button (Shortcut: CTRL+C) copies whatever text you have highlighted to the clipboard.


## History



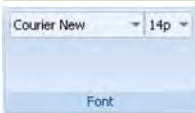
- The **Undo** button (CTRL+Z) reverses the last deletion or insertion made in the [Source Editor](#).
- The **Redo** button (CTRL+Y) reverses the last undo action.

## Search



- The above text box is a search box.
- Pressing the find button  opens the [Find Text dialog](#).
- The **Replace** button opens the [Replace Text Dialog](#).

## Font



- Use the above font and size drop-down menus to change the text appearance. The entire file will be affected.

## Debug



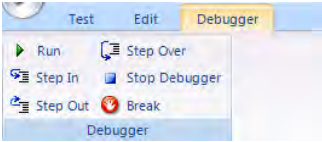


- Press the **Toggle Breakpoint** button (Shortcut: F9) to insert or remove a breakpoint at the current cursor position.

## Ribbon: Debugger

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

The **Debugger Tab** provides tools for use with the [Internal Debugger](#).

### How to Open

The **Debugger Tab** is available while the **Internal Debugger** is being used. To use the Internal Debugger, first enable it, then [Playback](#) your script. Instructions for enabling the Internal Debugger are [HERE](#).

### Debugger

- **Run** (F5): Continue executing the script.
- **Step In** (F11): Step into a function/procedure.
- **Step Out** (Shift+F11): Continue until the current procedure is exited.
- **Step Over** (F10): Go to the next line in the current procedure/function.
- **Stop Debugger** (Shift+F5): Stop executing the script and exit the debugger.
- **Break** (F9): Create a breakpoint in the script at the cursor.

## Ribbon: REST

[Top](#) [Previous](#) [Next](#)

### Screenshot



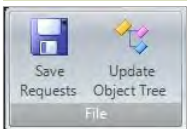
### Purpose

The **REST tab** is for use with editing [REST web service](#) definition files.

### How to Open

The **REST tab** is available anytime you have a REST definition file (.rest) file visible in the [Content View](#).

### File

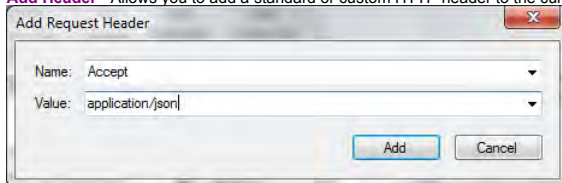


- **Save Requests** - Saves the current request definitions to the .rest file.
- **Update Object Tree** - Updates the main Rapise [Object Tree](#) with the current REST definitions. This turns each of your REST requests into Rapise learned objects that can be scripted against.

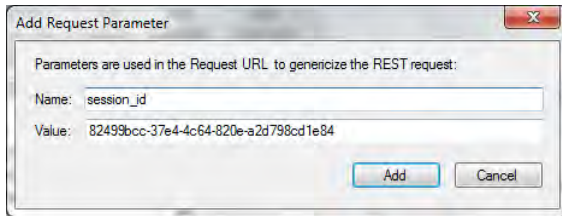
### Edit



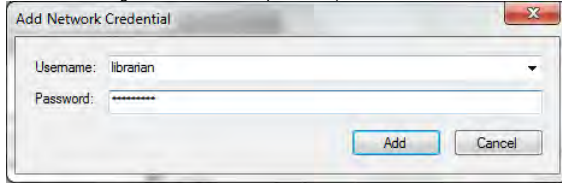
- **Add Header** - Allows you to add a standard or custom HTTP header to the current REST request:



- **Add Parameter** - Allows you to add a parameter name/value to the current REST request. This is useful when you want your test script to be able to pass through different values (e.g. get book #1 vs. book #2):



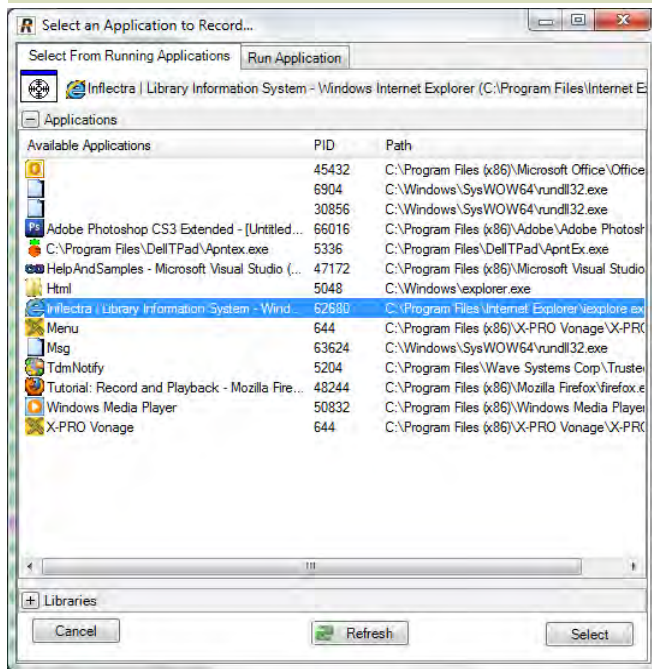
- **Add Credentials** - Allows you to add an HTTP basic authentication credential (username and password) to the request. Saves you having to add the header manually (which would require base64 encoding the username and password):



## Select an Application to Record... Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

The **Select an Application to Record...** (SAR) Dialog appears before [Recording](#) takes place. It queries the user for which program to record, as well as what [Recording Library](#) to use.

If you are recording the same application for the second time then SAR is not shown. The recording proceeds to last used application if it is still available on the screen.

### How To Open

To open the SAR Dialog, press the **Record/Learn** button on the Ribbon (**Test** tab > **Recording & Learning** menu):



### Libraries

Library	Description
<input type="checkbox"/> Auto	Detect library automatically
<input type="checkbox"/> .NET	.NET 1.1, 2.0, 3.0, 3.5 with Accessibility
<input checked="" type="checkbox"/> Internet Explorer HTML	HTML DDM-based recorder for Internet Explorer
<input type="checkbox"/> Firefox HTML	HTML DDM-based recorder for Mozilla Firefox
<input type="checkbox"/> Generic	Generic library contains basic definitions for most commo...

The **Library** table lists the available Recording Libraries. Select the one appropriate to the process/program you will record. If you select **Auto**, Rapise will attempt to choose the correct recording library for you. See the [Recording Library](#) section for more information.

### Available Applications

Available Applications	PID	Path
	7032	C:\Windows\explorer.exe
	3744	C:\Program Files\Google\GoogleToolbar
C:\Windows\system32\cmd.exe	4796	C:\Windows\system32\cmd.exe
Help & Manual	7024	C:\Program Files\EC Software\HelpAndM
Program Manager	7032	C:\Windows\explorer.exe
Sample ATM Login - Windows Internet Ex...	7000	C:\Program Files\Internet Explorer\explor...

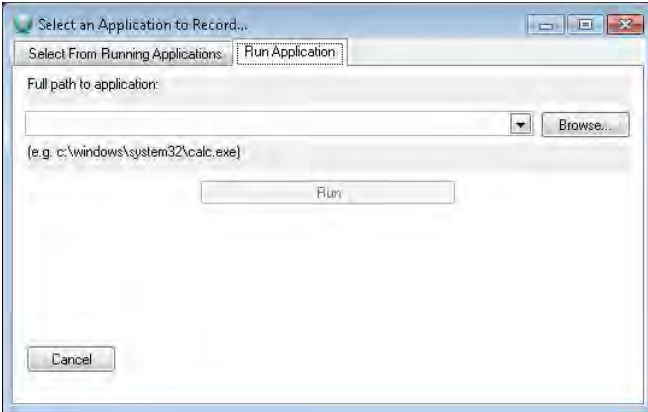
The **Available Applications** table lists all of the processes running at the time you open the **SAR dialog**. If the process you would like to record is already open, you can select it from the table. Pick the appropriate recording library (above) first before you pick an application to record; your application choice will become unselected if you do not do it last.

## Widgets



- The **Cancel button** closes the dialog.
- **Show All:** While unchecked only top level application windows reflected in the Windows Task Bar are shown in the 'Available Applications' list. Check this and press Refresh to see all top level windows available on the screen.
- **Refresh List:** Press to refresh the **Available Applications** table. After refreshing, you will see processes that began after the **SAR dialog** was opened.
- **Select button:** To record a process from the **Available Applications** table, select the process and then press the **Select** button.

## Run Application Tab

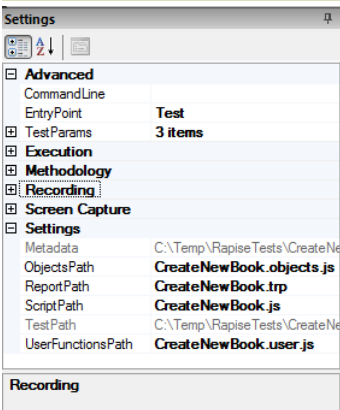


- **Path drop down list:** If the program you would like to record is not already open, you can specify its path here. If the program is already running, you can select it from the **Available Applications** table.
- **Browse button:** Browse for an application to open and record.
- **Run button:** To record a program that is not currently open, fill in the **Path** text-box and press the **Run** button.
- The **Cancel button** closes the dialog.

## Settings View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

Use the Settings Dialog to change test specific settings.

### How to Open

The Settings dialog is part of the [Default Layout](#).

### Advanced



- **CommandLine** is a freeform text box. Use it to specify values for global variables (beginning in **g\_**) to pass the [recorder](#) and [player](#). You can view which global variables are available in the source files

### Execution

Execution	
CacheObjects	False
CommandInterval	100
IterationsCount	1
ObjectLookupAttemptInterval	150
ObjectLookupAttempts	1

- **CacheObjects**: Remember object locations and try to reuse them for speed. This is helpful with dialog based applications.
- **CommandInterval**: Time interval (in milliseconds) between script commands during script execution.
- **IterationsCount**: Your test script will be executed this many times consecutively during [Playback](#).
- **ObjectLookupAttemptInterval**: This is the time Rapise will wait between attempts to locate an object.
- **ObjectLookupAttempts**: This is the number of times Rapise will attempt to locate an object.

## Recording

Recording	
BeautifullySavedObjects	False

- **BeautifullySavedObjects** affects how the [Script Recorder](#) writes object information to your test script. If **False**, the object definition will be written as a single line:

```
var saved_script_objects={
  Balance:{"version":0,"object_type":"SeSSimulated","object_name":"Transaction Completed Successfully\n\nAccount 00000005 Balance:1046.00","object_class":"Static","object_role":"ROLE
};
```

If **True**, the object definition will be written in a manner that takes more space, but is easier to read and change:

```
var saved_script_objects={
  Balance:{
    "version": 0,
    "object_type": "SeSSimulated",
    "object_name": "Transaction Completed Successfully\n\nAccount
00000005 Balance:1046.00",
    "object_class": "Static",
    "object_role": "ROLE SYSTEM STATICTEXT",
    "object_text": "Transaction Completed Successfully\n\nAccount
00000005 Balance:1046.00",
    "locations": [
      {
        "locator_name": "Location",
        "location": {
          "location": "4.4.4",
          "window_name": "SmarteATM",
          "window_class": "#32770"
        }
      },
      {
        //section omitted for brevity
      }
    ]
  }
};
```

Objects that were learned in previous recordings are affected by the value of **BeautifullySavedObjects**.

## Screen Capture

Screen Capture	
Capture Execution	False
Capture Recording	False
Include in Report	False
Widget Only	False

- **Capture Execution**: Set this to **True** if you want to save screen images for each recognized object during playback.
- **Capture Recording**: Set this to **True** if you want to save screen images for each action during recording.
- **Include in Report**: Set this to **True** to include the saved images in the execution report during playback.
- **Widget Only**: Set this to **True** to only save the widget area in the screenshot, as opposed to the whole window.

## Settings

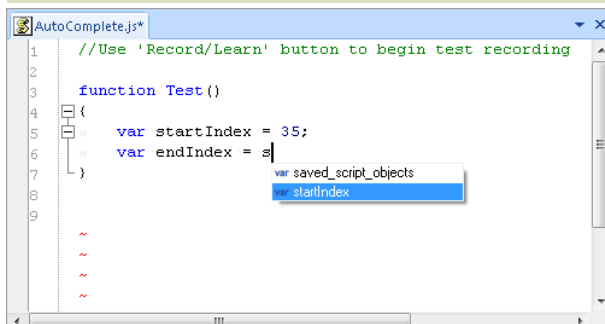
Settings	
Metadata	C:\Users\Public\Documents\Shared
ObjectsPath	TwoDialogsTest.objects.js
ReportPath	TwoDialogsTest.trp
ScriptPath	TwoDialogsTest.js
TestPath	C:\Users\Public\Documents\Shared
UserFunctionsPath	TwoDialogsTest.user.js

- **UserFunctionsPath**: Path (relative to the test directory) to the file with user-defined functions utilized in this test. Normally this file has name in form \*.user.js.
- **CustomStrings**: Click to open the [NameValue Collection Editor Dialog](#).
- **ObjectsPath**: Path (relative to the test directory) to file containing object tree information. This file contains `saved_script_objects` structure with all object locators gathered during recording and learning.
- **ReportPath**: Path (relative to the test directory) to the test's report file.
- **ScriptPath**: Path (relative to the test directory) to the test script.
- **TestPath**: Path to the test definition file (\*.sstest).

## Source Editor

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

To display and edit javascript files. The editor supports [Syntax Highlighting](#), [Syntax Checking](#), [Code Folding](#) and [Code Completion](#).

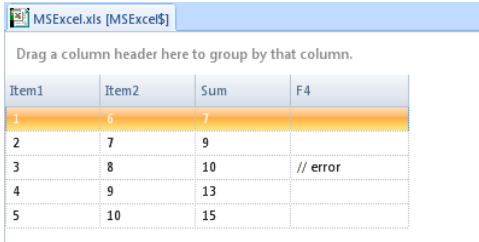
## How to Open

Use the [Test Files Dialog](#) to open a javascript file. The javascript file will be opened in a **Source Editor**, in the [Content View](#). The [Edit Tab](#) of the Ribbon will also open.

## Spreadsheet Viewer

[Top](#) [Previous](#) [Next](#)

### Screenshot



Item1	Item2	Sum	F4
1	6	7	
2	7	9	
3	8	10	// error
4	9	13	
5	10	15	

## Purpose

To display excel (xls) files.

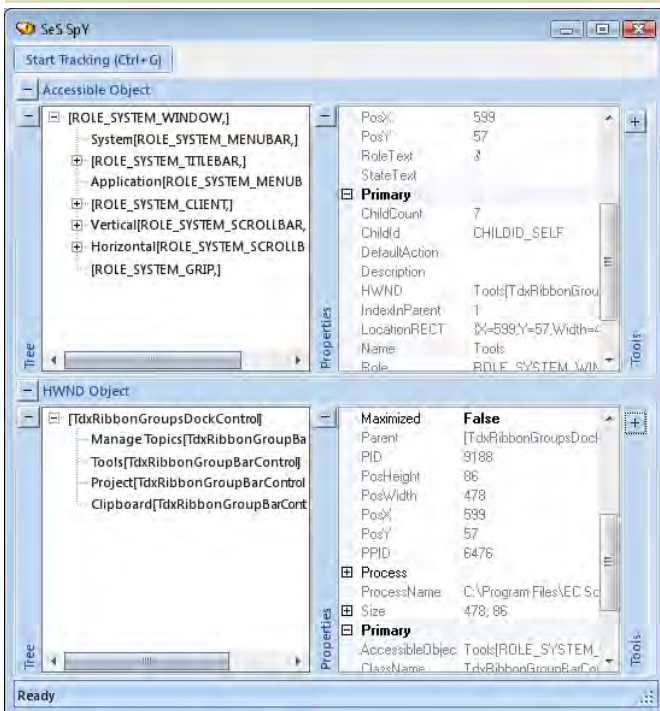
## How to Open

Use the [Test Files Dialog](#) to open an excel file. The excel file will be opened in a **Spreadsheet Viewer**, in the [Content View](#). The [Spreadsheet](#) tab of the Ribbon will also open.

## SeS Spy Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

The **SeS Spy** dialog is used to [Object Spy](#).

## How to Open

There are three ways to open the SeS Spy dialog:

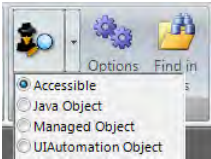
1. Press the **Spy** Button on the Ribbon (**Test** tab > **Tools** menu)



2. Press the **Spy** Button on the [Recording Activity Dialog](#)
3. Press the **Pick Object** button on the [Recording Activity Dialog](#). Note: If you use this method, the dialog has an extra **Learn Selected** button.

## Choosing the type of Spy

You can change the type of Spy that will be launched by clicking on the down arrow to the right of the Spy icon in the main application Ribbon:



There are four types of Spy available:

1. **Accessible** - This is used to inspect applications that expose their properties using the Microsoft Active Accessibility (MSAA) technology. This is typically used by applications written in MFC, ATL, Qt, C++ and Visual Basic.
2. **Java Object** - This is used to inspect applications written using the Java AWT and Swing UI frameworks.
3. **Managed Object** - This is used to inspect applications written in .NET 1.1, .NET 2.0, .NET 4.0 using Microsoft Windows Forms.
4. **UIAutomation Object** - This is used to inspect applications that expose their properties using the Microsoft's newer UIAutomation technology. This is typically used by applications written in WPF, Silverlight and Java SWT.



### Start Tracking

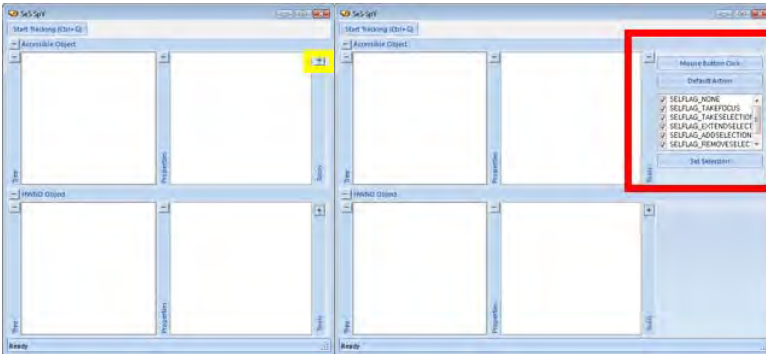
The **Start Tracking** button (or CTRL+G) causes Rapise to enter **Tracking Mode**. In **Tracking Mode**, Rapise investigates the object under your mouse. It identifies the object's type and learns the object's properties. As you move your mouse, the objects you point to are highlighted (a box is drawn around them).

### Stop Tracking

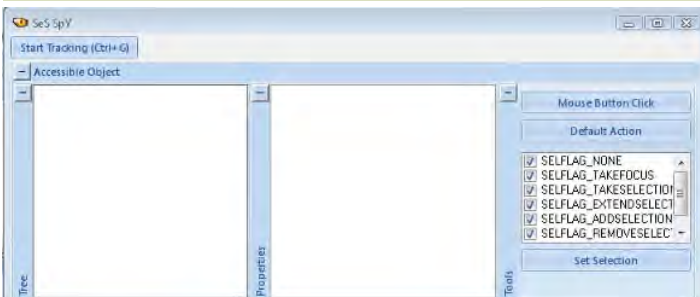
The **Stop Tracking** button is only visible in **Tracking Mode**. Press Stop Tracking (or CTRL+G) to exit Tracking Mode. The SeS Spy dialog will display information for the last object highlighted.

### Maximize/Minimize buttons

The maximize  and minimize  buttons control the appearance of the dialog. They either hide or make visible the sections to their right or below. See the example below. The button highlighted in yellow in the left image is pressed to get the image on the right. Changes are surrounded with a red box.



### Accessible Object



The **Accessible Object** section of the SeS Spy dialog shows properties of the object that are visible through the Microsoft Active Accessibility interface.

#### Tree

The spied upon object and its children are displayed here.

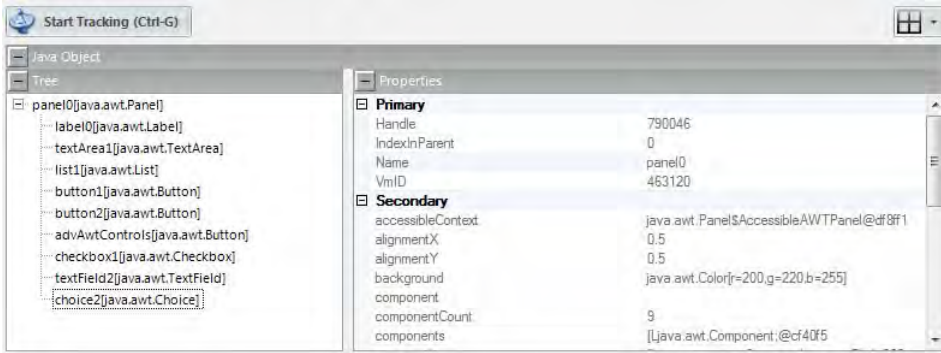
#### Properties

Object fields and field values are displayed here.

#### Tools

- **Mouse Button Click**: Emulate Left mouse click for the item selected in Spy tree.
- **Default Action**: Execute **DoDefaultAction** for given accessible object.
- **Set Selection**: Perform **accSelect** using the option flags set in the corresponding checklist (above).

### Java Object



The **Java Object** section of the SeS Spy dialog shows properties of the object that are visible through the [Java Access Bridge](#) interface.

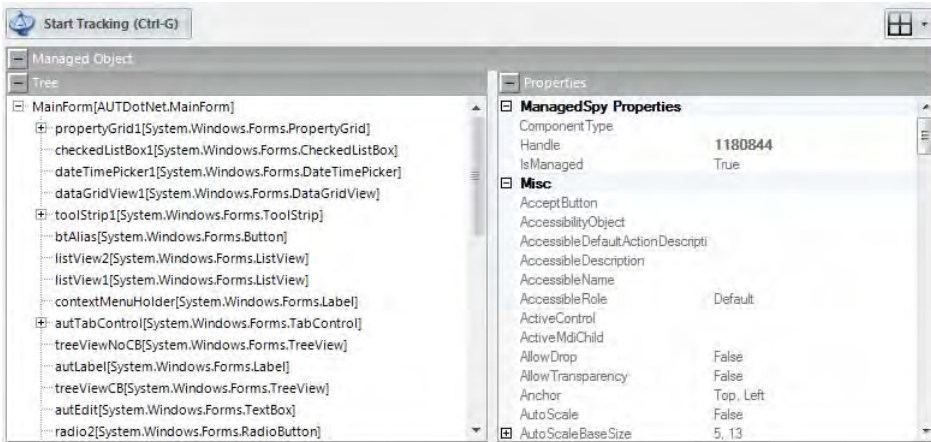
**Tree**

The spied upon object and its children are displayed here.

**Properties**

Object fields and field values are displayed here.

**Managed Object**



The **Managed Object** section of the SeS Spy dialog shows properties of the object that are visible through .NET Framework reflection interface.

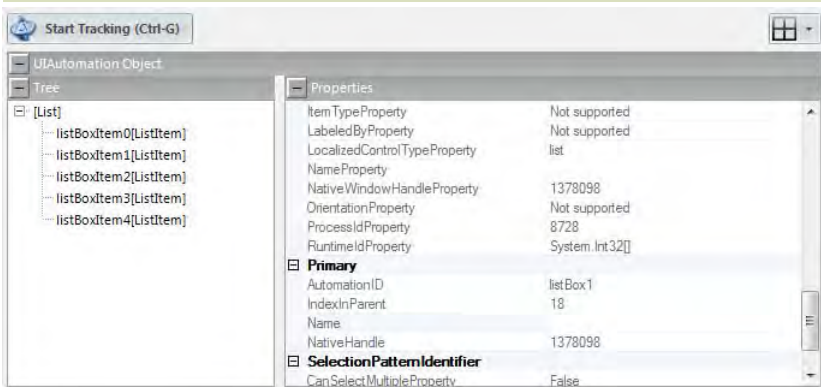
**Tree**

The spied upon object and its children are displayed here.

**Properties**

Object fields and field values are displayed here.

**UIAutomation Object**



The **UIAutomation Object** section of the SeS Spy dialog shows properties of the object that are visible through the UIAutomation interface.

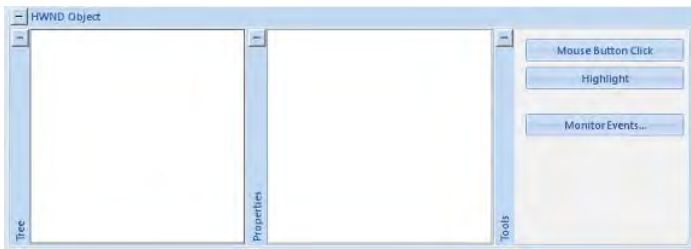
**Tree**

The spied upon object and its children are displayed here.

**Properties**

Object fields and field values are displayed here.

## HWND Object



The **HWND Object** section of the SeS Spy dialog shows properties of the object that are visible with its HWND handle.

### Tree

The spied upon object and its children are displayed here.

### Properties

Object fields and field values are displayed here.

### Tools

- **Mouse Button Click:** Emulate Left mouse click for the item selected in Spy tree.
- **Highlight:** Draw rectangle surrounding selected object (HWND or Accessible).
- **Monitor Events** opens the [Accessible Events Dialog](#).

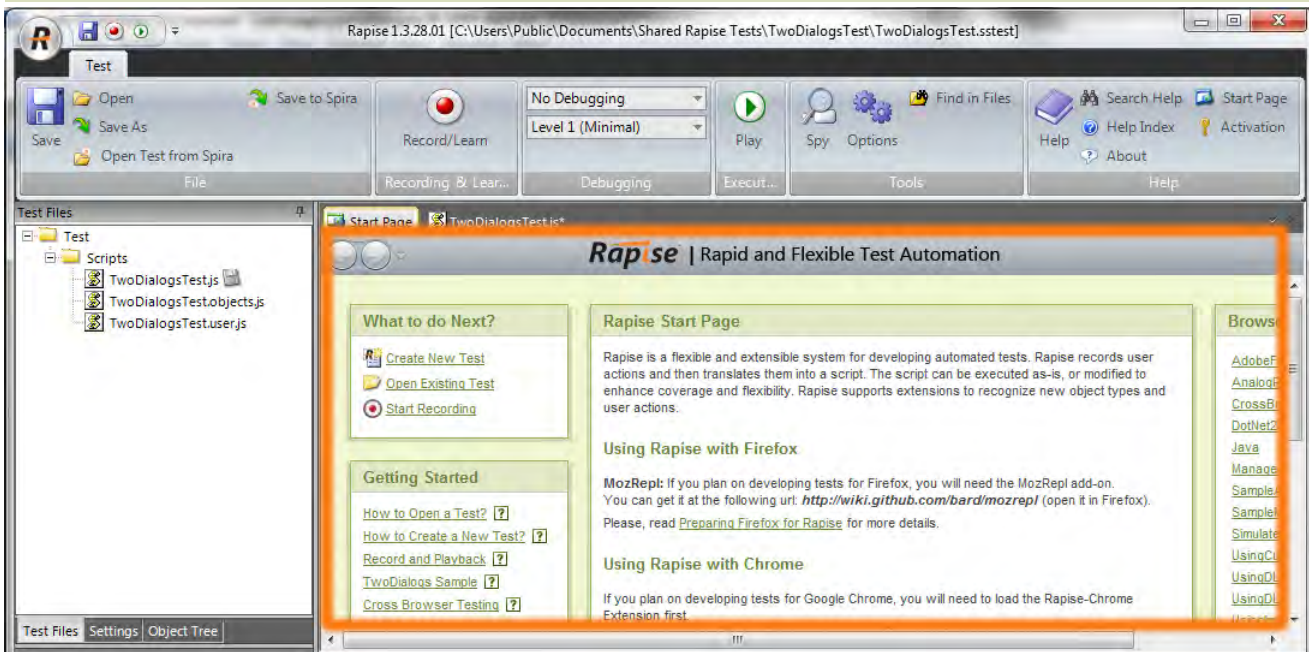
### See Also

- Microsoft Active Accessibility is described here <http://msdn.microsoft.com/en-us/magazine/cc301312.aspx>
- HWND is described [HERE](#).
- Microsoft UIAutomation is described here <http://support.microsoft.com/kb/971513/>

## Start Page

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

To display the latest news and information regarding Rapise.

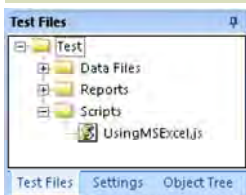
### How to Open

The **Start Page** opens automatically with Rapise. This behavior can be modified in the [Options](#) dialog using the **ShowStartPageOnStartup** setting.

## Test Files View

[Top](#) [Previous](#) [Next](#)

### Screenshot





## Purpose

The **Test Files** dialog allows you to navigate and alter the Test hierarchy, including the following:

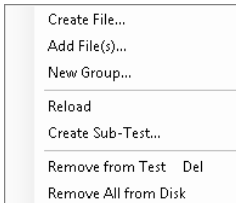
- the script
- Report files (\*.trp)
- Images captured during execution using [Checkpoints](#)
- Analog recording files (\*.arf)
- data files

## How to Open

The **Test Files** dialog is part of the [Default Layout](#).

## Context Menu (Folder)

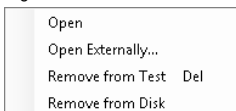
Right click on a folder to see:



- **Create File:** Create and add a new file to the test.
- **Add File:** Add an existing file to the test.
- **New Group:** Create a logical grouping of files in the test. This will **not** add a folder to the file system.
- **Reload:** Refresh group contents. Use it for [filter groups](#) ('IsFilterGroup' is set to 'True' in group properties), e.g. for Report group.
- **Create Sub-Test...:** Launch Create Sub-Test dialog.
- **Remove from Test:** Remove the selected grouping from the test. This does **not** delete included files from your hard disk.
- **Remove All from Disk:** Remove all files included into the selected grouping from your hard disk.

## Context Menu (File)

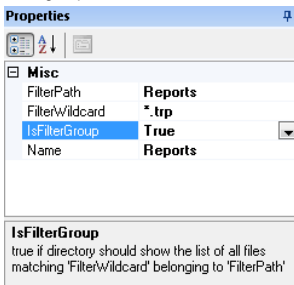
Right click on a file to see:



- **Open:** Open the file in Rapise.
- **Open Externally...:** Open the file using associated program. E.g. if a Notepad is registered in Windows to open TXT files, then TXT file will be opened by Notepad.
- **Remove from Test:** Remove the file from your test. This does **not** delete the file from your hard disk.
- **Remove from Disk:** Remove the file from your test and hard drive.

## Filter Groups

Filter groups read its contents from disk according to specified path and wildcard. You may setup a filter group by editing group properties:

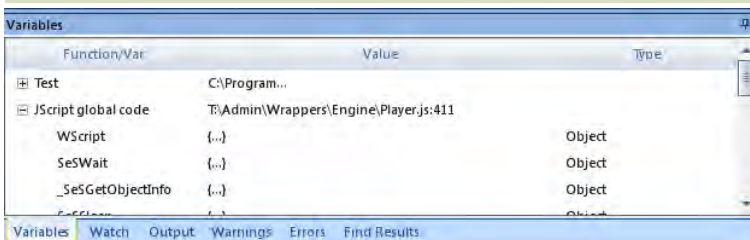


- **FilterPath:** Root path to find files via wildcard (valid only if 'IsFilterGroup' is 'True').
- **FilterWildcard:** Filter wildcard (valid only if 'IsFilterGroup' is 'True').
- **IsFilterGroup:** 'True' if directory should show the list of all files matching 'FilterWildcard' belonging to 'FilterPath'.
- **Name:** Group name.

## Variable/Call Stack View

[Top](#) [Previous](#) [Next](#)

### Screenshot



## Purpose

Lists the functions in the current call stack. Beneath each function, variables/objects local to that function are listed with their value and type.

## How to Open

Begin [debugging](#) a script. The **Variable/Call Stack View** will open automatically.

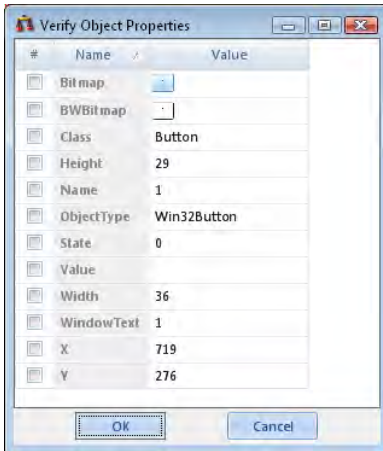
## Go to a function definition

Double click on a function to go to its definition.

## Verify Object Properties Dialog

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

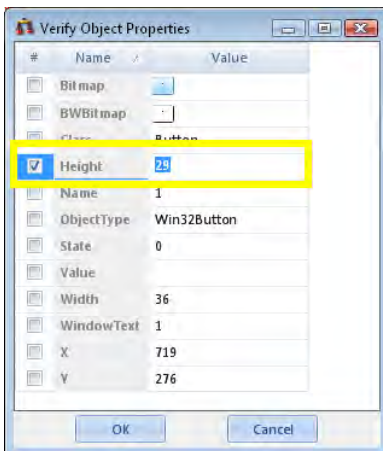
Use the **Verify Object Properties** dialog during [recording](#) to add [checkpoints](#).

### How to Open

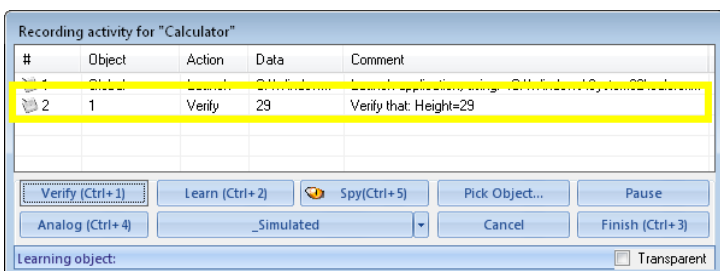
1. First, open the [Recording Activity Dialog](#).
2. Position the mouse over an object and press **Ctrl+1**, or
3. Press the **Verify** button and then click the target object with the mouse cursor.

### Create a Checkpoint

Your checkpoint will be associated with a particular object. That object's properties will be listed in the **Verify Object Properties** dialog. Check those properties that you wish to verify during [playback](#). Enter expected values for the selected properties in the **Value** column. **Note:** The **Bitmap** and **BWBitmap** properties are images of the object.



Press the **OK** button. The **Verify Object Properties** dialog will close, and the [Recording Activity](#) dialog will contain a new **Verify** action:



The generated script will have a corresponding [assert statement](#):

```
//Verify that: Height=29
Tester.AssertEqual("Verify that: Height=29", SeS('Obj1').GetHeight(), "29");
```

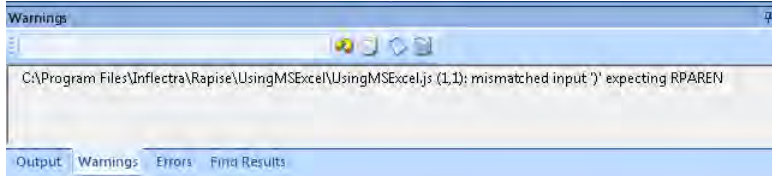
### See Also

- [Recording](#)
- [Assert Statements](#)

## Warning View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

To display syntax error messages as you edit javascript files.

### How to Open

The **Warning** view is part of the [Default Layout](#).

### Error Message

C:\Program Files\Inflectra\Rapise\UsingMSEXcel\UsingMSEXcel.js (1,1): mismatched input ')' expecting RPAREN  
 Double click on an error message to go to the corresponding source line.

### Widgets



- The text box is a search box.
- The icons from left to right are **Find Next Entry**, **Copy Selected**, **Clear All Text**, and **Select All Text**.

### See Also

- [Syntax Checking](#)

## Watch View

[Top](#) [Previous](#) [Next](#)

### Screenshot



### Purpose

To input expressions and view their values as the script executes.

### How to Open

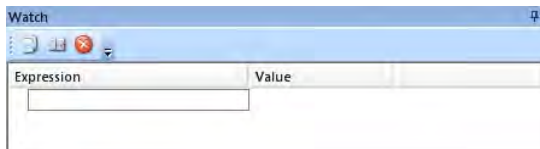
Begin [debugging](#) a script. The **Watch View** will open automatically.

### Inputting an Expression

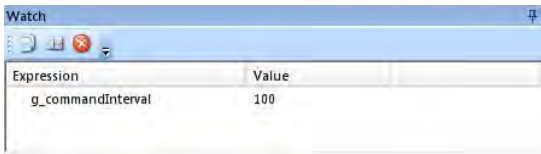
1. Click the first blank line:



2. Double click on the highlighted line, under the **Expression** column. A text box will appear.



3. Input the expression you wish to investigate. Press **Enter**.



## Widgets



From left to right: **Copy** (an entire row), **Copy Watch Value**, **Delete**.

## HowTos

[Top](#) [Previous](#) [Next](#)

This section focuses on specific tasks that a Rapise user might want to accomplish.

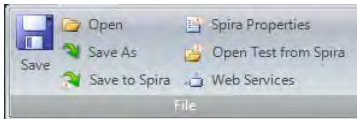
## Open a Test

[Top](#) [Previous](#) [Next](#)

You can open a test in two ways: (1) From the Ribbon, and (2) From the Application menu.

### Ribbon

Select the **Open** option from the **File** menu on the **Test** Tab of the Ribbon:



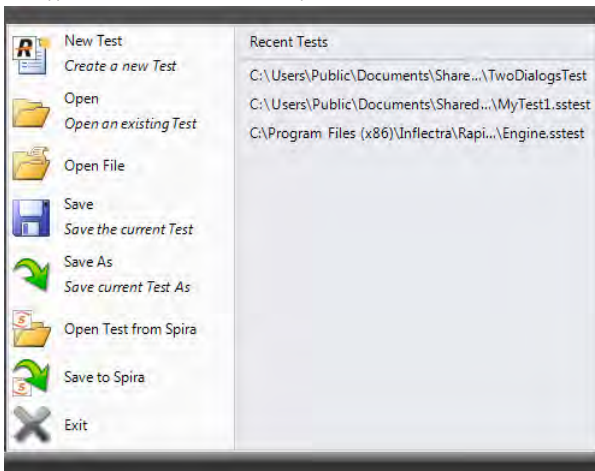
You can also open a test that is stored in **SpiraTest** (our web-based test management system) instead of the local filesystem. This is done by clicking on the **Open Test from Spira** option instead. More details on using Rapise with SpiraTest can be found in the [SpiraTest Integration](#) section.

### Application Menu

Open the Application Menu by clicking on the Menu Button at the top left of the Rapise window:



The Application menu has an **Open Test** option, and a list of **Recent Test** from which you may choose:



You can also open a test that is stored in **SpiraTest** (our web-based test management system) instead of the local filesystem. This is done by clicking on the **Open Test from Spira** option instead. More details on using Rapise with SpiraTest can be found in the [SpiraTest Integration](#) section.

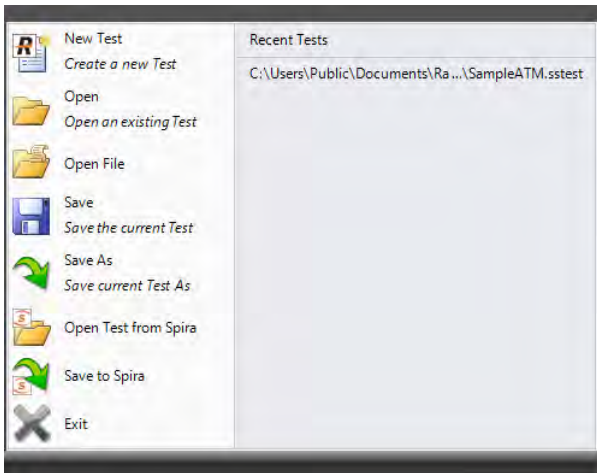
## Create a New Test

[Top](#) [Previous](#) [Next](#)

Select the Menu Button at the top left of the screen:



A menu will open up:



Select the **New Test** option. The [Create New Test](#) dialog will appear. Follow the instructions on this dialog.

## Restoring the Default Layout

[Top](#) [Previous](#) [Next](#)

There are two ways to restore the default layout: (1) On Startup, and (2) In the Options Menu.

### On Startup

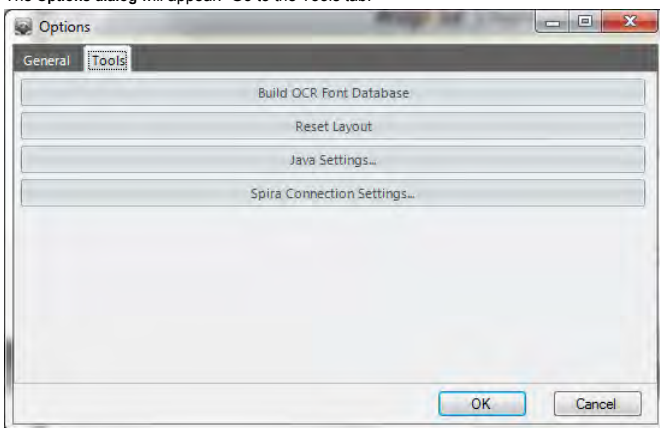
Press the **Shift** key while you open Rapise. Keep the Shift key down until Rapise is done initializing.

### Options Menu

In Rapise, select the **Options** button. It is on the Ribbon in the Tools section:



The **Options** dialog will appear. Go to the Tools tab:



Select the **Reset Layout** button. Rapise will restart.

## Change Test Entry Point

[Top](#) [Previous](#) [Next](#)

Rapise assumes that the entry point of a test - `Test()` function is defined in a file specified in [ScriptPath](#) property of the [Settings](#) dialog. If you want to place `Test()` function in another file then do not forget to update [ScriptPath](#) property of the test.

## Do Absolute Analog Recording

[Top](#) [Previous](#) [Next](#)

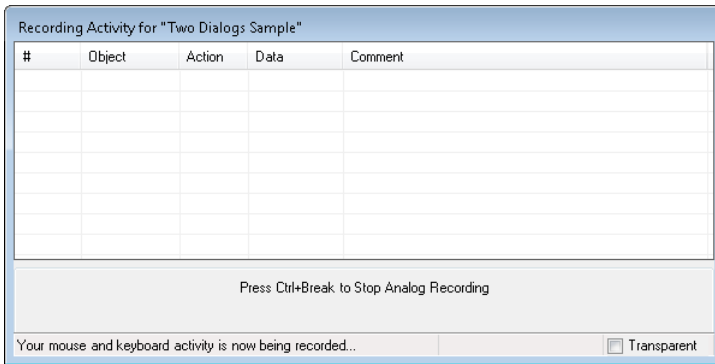
Let's once again use our trusty over-simplified TwoDialogs sample application to learn how to use absolute analog recording and use it to discover the value as well as the dangers associated with absolute analog recording.

Steps:

- (1) Run TwoDialogs sample AUT. By default this will be located in C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe
- (2) Start Rapise and create a new test and call it TwoDialogsAnalogAbsolute.
- (3) Press the Record/Learn button in the toolbar of Rapise.
- (4) When the "Select an Application to Record" dialog is displayed, choose the TwoDialogs.exe application and ignore the library list - we will not be using any library for analog recording. Press the Select button.
- (5) The Recording Activity dialog will be displayed with an empty grid.

**NOTE:** this recording session is going to be a little different from previous sessions. Previously we could interrupt our object-related recording/learning with other activities and because Rapise was recording activity related only to the target application, our recording or object learning would be unaffected. However, in analog recording, Rapise is monitoring the mouse and keyboard for the entire system - for all applications. This means that if you answer an email in the middle of analog recording, or log in to a secure system, all the steps including mouse movement, keystrokes, etc., will all be recorded. However, note also that screen contents are not recorded by Rapise.

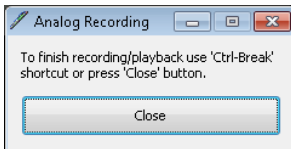
- (6) If the TwoDialogs UI has been occluded, bring it back to the front so you don't have to hunt for it when you start recording.
- (7) When you're ready to record the session, hit Ctrl+4 on the Recording Activity dialog.



NOTE: Pressing the Analog button on the Recording Activity dialog starts a relative analog recording session. Use the Ctrl+4 key sequence to start the absolute analog recording session. Rapise will begin recording all mouse and keyboard activity until you stop the recording.

Note also that the prompt in the notification/status area of the Recording Activity dialog is different from that for relative analog. It tells you that "Your mouse and keyboard activity is now being recorded."

A minimized window will be created that indicates that analog recording is in progress and allowing you to stop the recording.



(7) Go to the TwoDialogs AUT and click anywhere in the application's window to start the analog recording.

Click the mouse on the empty "Please enter your name" text box.

Type a name in the text box.

Hit the <tab> key or click the left mouse button to advance the input position to the second text box.

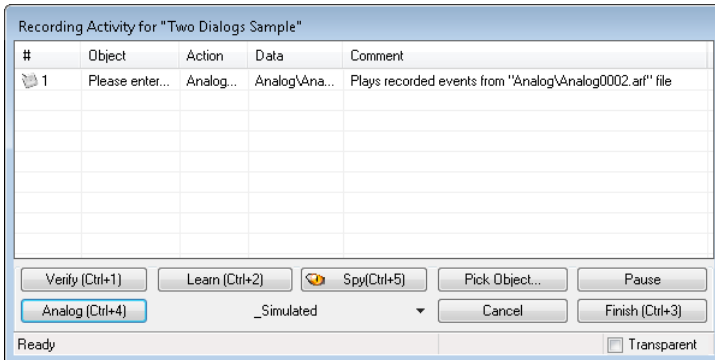
Type another name.

Move the mouse to the OK button and press the mouse left button.

(8) When you have recorded enough, switch to the Analog Recording dialog box and press the close button or hit the keys Ctrl+Break.

NOTE: If you use the "close" button on the Analog Recording dialog, the movement of the mouse to the Analog Recording dialog, and the mouse-click on the Close button will be recorded as part of the analog recording output. This might not be a desirable outcome at playback time because the Analog Recording dialog will not be present and the mouse click will be played in a potentially random place on the screen. For this reason, Ctrl+Break is probably a better option to terminate analog recording.

NOTE: The grid will have no entry added until you end the analog recording with the Close button in the Analog Recording dialog. When you do, it will add an entry to the grid.



(9) You can now record additional analog sessions, if you wish.

(10) You can record normal object activity before and/or after the analog recording. When you have finished all recording press the Finish button or hit Ctrl+3. Notice that the Analog entry is added to the grid.

(11) The Rapise screen will now be restored and will have placed focus in the editor pane of the Rapise with TwoDialogsAnalogAbsolute.js script displayed. You should see code something like the following:

```
//Plays recorded events from "Analog\Analog0003.arf" file
SeS('Simulated').DoAnalogPlay("Analog\Analog0003.arf");
```

(12) Press the Play button on the Rapise toolbar to playback the recording you made. Be sure not to interfere with the mouse or keyboard whilst the recording is playing back.

NOTE: You will see all mouse and keyboard activity reproduced as the analog recording plays. The recording will start from the point where you left-clicked the mouse to begin the recording (step 7 above) and will end with clicking the close button in the Analog Recording dialog or at the last action before you pressed Ctrl+Break.

(13) When the analog playback is complete, use the mouse to move the Two Dialogs AUT to a different location on the screen. Play the recording again, and watch the operation unfold. The most important thing to realize is that the absolute analog recording will playback the recording wherever the application is positioned on the screen wherever the AUT was positioned when you made the recording. Absolute analog recording records relative to the top-left corner of the system screen. Try this for yourself, but be sure to minimize all applications before starting.

## Do Relative Analog Recording

[Top](#) [Previous](#) [Next](#)

Let's once again use our trusty over-simplified TwoDialogs sample application to learn how to use relative analog recording.

Steps:

(1) Run TwoDialogs sample AUT. By default this will be located in <C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe>

(2) Start Rapise and create a new test and call it TwoDialogsAnalogRelative.

(3) Press the Record/Learn button in the toolbar of Rapise.

(4) When the "Select an Application to Record" dialog is displayed, choose the TwoDialogs.exe application. Since we will not be using a library for this recording, the library selection is irrelevant. Press the Select button.

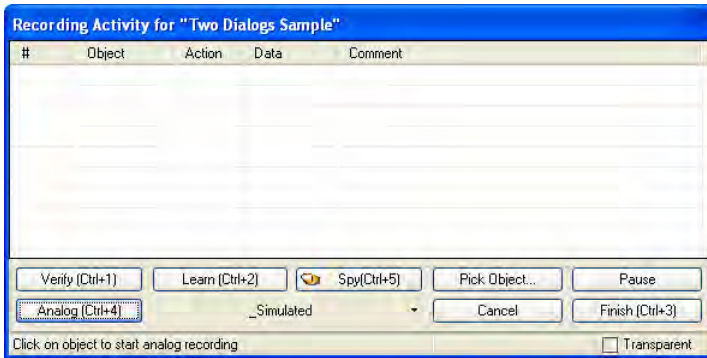
(5) The Recording Activity dialog will again be displayed with an empty grid.

NOTE: this recording session is going to be a little different from previous sessions. Previously we could interrupt our object-related recording/learning with other activities and because Rapise was recording activity only related to the target application, our recording or object learning would be unaffected. However, in analog recording, Rapise is monitoring the mouse and keyboard for the entire system - for all applications. This means that if you answer an email in the middle of analog recording, or log in to a secure system, all the steps including mouse movement, keystrokes, etc., will all be recorded. However, note also that screen contents are not recorded by Rapise.

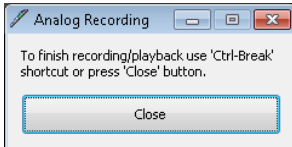
(6) If the TwoDialogs UI has been occluded, bring it back to the front so you don't have to hunt for it during recording.

(7) When you're ready to record the session, hit the Analog button on the Recording Activity dialog.

NOTE: The key sequence Ctrl+4 starts an absolute analog recording session. Press the Analog button to start the relative analog recording session. When you press the Analog button, two things will happen. Firstly, the status bar of the Recording Activity dialog will change to read, "Click on object to start analog recording."



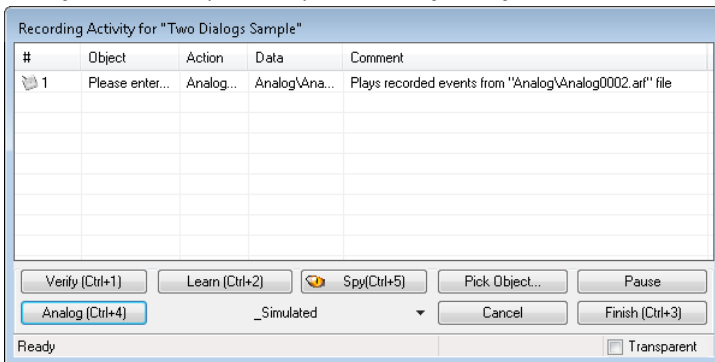
After the next mouse click, Rapise is recording all mouse and keyboard activity until you stop the recording. Secondly, a minimized window will be created that indicates that analog recording is in progress and allowing you to stop the recording.



- (7) Go to the TwoDialogs AUT and click anywhere in the application's window to start the analog recording.
  - Click the mouse on the empty "Please enter your name" text box.
  - Type a name in the text box.
  - Hit the <tab> key or click the left mouse button to advance the input position to the second text box.
  - Type another name.
  - Move the mouse to the OK button and press the mouse left button.

- (8) When you have recorded enough, switch to the Analog Recording dialog box and press the close button or press the key sequence Ctrl+Break. If you use the "close" button on the Analog Recording dialog, the movement of the mouse to the Analog Recording dialog, and the mouse-click on the Close button will be recorded as part of the analog recording output. This might not be a desirable outcome at playback time because the Analog Recording dialog will not be present and the mouse click will be played in a potentially random place on the screen. For this reason, Ctrl+Break is probably a better option to terminate analog recording.

NOTE: The grid will have no entry added until you end the analog recording with the Close button in the Analog Recording dialog. When you do, it will add an entry to the grid.



- (9) You can now record additional analog sessions if you wish.
- (10) You can record normal object activity before and/or after the analog recording. When you have finished all recording press the Finish button or hit Ctrl+3.
- (11) The Rapise screen will now be restored and will have placed focus in the editor pane of the Rapise with TwoDialogsAnalogAbsolute.js scrip displayed. You should see code something like the following:

```
//Plays recorded events from "Analog\Analog0003.arf" file
SeS('Simulated').DoAnalogPlay("Analog\Analog0003.arf");
```

- (12) Press the Play button on the Rapise toolbar to playback the recording you made. Be sure not to interfere with the mouse or keyboard whilst the recording is playing back.
 

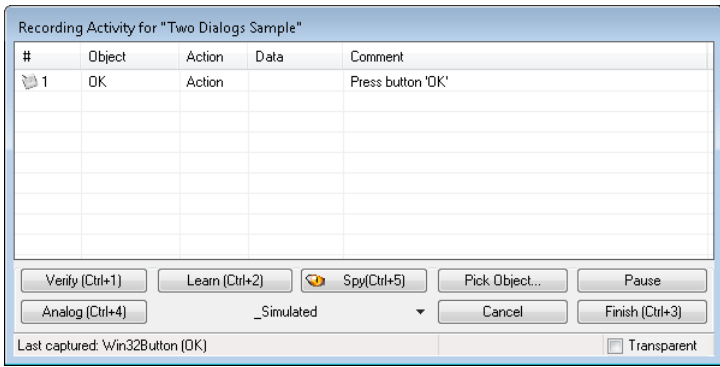
NOTE: You will see all mouse and keyboard activity reproduced as the analog recording plays. The recording will start from the point where you left-clicked the mouse to begin the recording (step 7 above) and will end with clicking the close button in the Analog Recording dialog. If you used Ctrl+Break to end the recording then the last recorded activity will be the one that keystroke.
- (13) When the analog playback is complete, use the mouse to move the Two Dialogs AUT to a different location on the screen. Play the recording again, and watch the operation unfold. The most important thing to realize is that the relative analog recording will playback the recording wherever the application is positioned on the screen. This is because you used relative analog recording. However, once the recording within the AUT is complete, all mouse motion and keyboard strokes are relative to the current position of the AUT. Suppose that during analog recording, you click the OK button in TwoDialogs.exe, then move the mouse to terminate the recording using the analog recording Close button. Now, prior to playback, you move the AUT to a different location on the screen and hit playback. All the activity within the AUT will be faithfully reproduced. However, the mouse motion outside the AUT will be relative to the position, so the following activities will not be accurately reproduced. Try this for yourself, but be sure to minimize all applications before starting so you don't cause mouse events where they will do harm to other applications on the screen.

## Learn an Object

[Top](#) [Previous](#) [Next](#)

To illustrate learning an object, we return to the [TwoDialogs](#) sample. First, let's learn the OK button using recording. We have done this before in the [TwoDialogs](#) sample. Steps:

- (1) Run TwoDialogs sample AUT. By default this will be located in C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe
- (2) Start Rapise and create a new test and call it TwoDialogsRecording.
- (3) Press the Record/Learn button in the toolbar of Rapise.
- (4) When the "Select an Application to Record" dialog is displayed, choose the TwoDialogs.exe application and in the library list, select only the top library on the list - "Auto." Press the Select button.
- (5) In the TwoDialogs AUT, use the mouse to press the OK button. Dismiss the alert message box complaining about the empty name.
- (6) Notice that two things will happen. Firstly, the OK button will be surrounded with a red marker, indicating that the OK button has been learned. Secondly, the action of clicking the OK button is recorded in the Recording Activity dialog. That recording has a single entry:



(7) Press the Finish button (or press Ctrl+3) to end the recording.

(8) Rapise will return to be the foreground application, and it will have selected the TwoDialogsRecording.js (or whatever name you gave the test when you created it).

(9) Notice that there is a single line or script that has been added to the script file:

```
SeS('OK').DoAction();
```

This line of script has two interesting parts.

The "SeS('OK') is the identity (not the locator or location) of the OK button. This is the object that was learned during recording.

The "DoAction()" is the instruction to the running script to take the action associated with a button. A normal button has only a single possible action - to be pressed.

The Record/Learn process has taken both steps for you, and joined them together.

Now, let's use (normal) object learning to learn the same OK button and to call a method for the object.

Steps:

(1) Run TwoDialogs sample AUT. By default this will be located in C:\Program Files\Inflectra\Rapise\Samples\TwoDialogs\TwoDialogs.exe

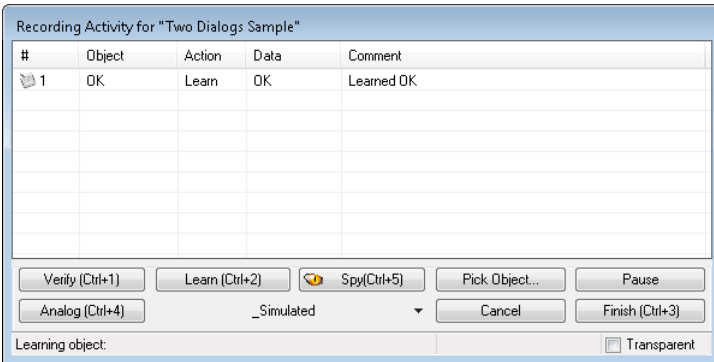
(2) Start Rapise and create a new test and call it TwoDialogsLearn.

(3) Press the Record/Learn button in the toolbar.

(4) When the "Select an Application to Record" dialog is displayed, choose the TwoDialogs.exe application. Leave the library selection in its default state - we will not be using it this time. Press Select. Wait for the Recording Activity dialog to appear in the lower-right corner of the screen.

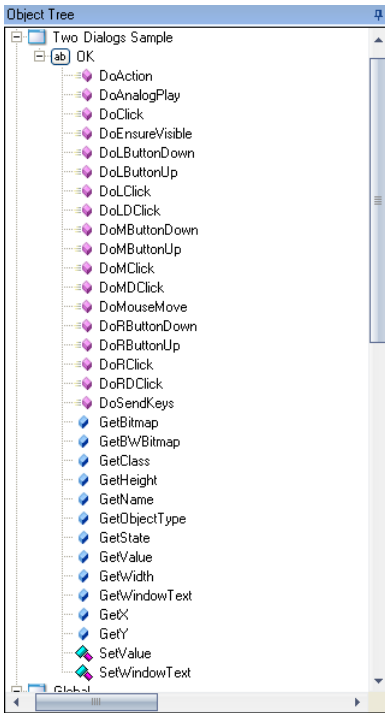
(5) Hover the mouse over the OK button of the TwoDialogs AUT but do not press the button.

(6) With the mouse positioned over the OK button, press Ctrl+2 ( the "Learn" command). You will see the OK button surrounded with a red highlight. You will also see that the Recording Activity dialog has been updated with a Learn event.

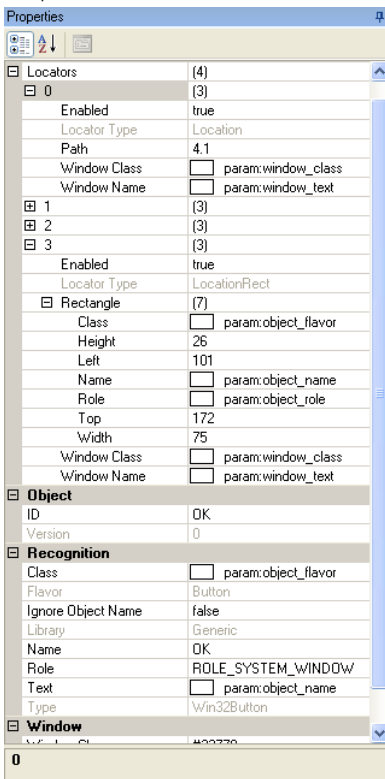


(7) Press the Finish button or Ctrl+3 to end the recording session. You will now see that Rapise has "learned" about the OK button, and the Object Tree in the upper left-hand pane of the Rapise has a new entry called "OK" (shown here expanded). The list of items contained under the OK button entry in the Object Tree is the set of methods and properties available for the OK object. Methods are listed with purple icons, read properties are listed with blue icons, and write properties are listed with blue and purple icons. Notice that the DoAction property is listed and recall in the previous section when we recorded pressing the button, the DoAction method was chosen for the button-press action.





(8) While we are looking at this OK object, let's make a few observations about it. These observations will be useful for your later dealings with Rapise and will make the script more informational and relevant as you delve into Rapise. First, look down at the Properties box that appears under the Object Tree in the bottom left corner of the Rapise screen. The screenshot below has some of the tree nodes expanded.



First, notice that the OK button has four (4) "locators" defined. When you have Rapise "learn" an object, it must collect data about that object so that it can relocate it even if the application has moved on the screen, and even if the application is in a different state of execution. In order to accomplish this, Rapise looks for all useful ways to uniquely identify the object. As bad, or perhaps worse, than not being able to find an object would be to find the wrong object on the AUT. Every time Rapise is required to locate this object, it will first try to use the first locator. If it fails to positively and uniquely match with that locator, it will try the second, and so on. Rapise will not give up and declare failure until it has failed to identify with all available locators.

Second, notice the ID entry in the Object section of the pane. This is the name of the object from Rapise's perspective. All Rapise names are available through the SeS() function call. Therefore, if we want to refer to the "OK" object, we will use SeS("OK") to refer to it. Once we have correctly identified the object, all valid methods and properties can be accessed by using that object as the basis.

Thirdly, notice in the main editor window of the Rapise, that no code has been added. When you identified the OK button, all Rapise did was add the new object to the Object Tree. It did not write any code in the javascript file.

(9) In the automated (recorded) section above, you saw that when you pressed the OK button on the dialog, Rapise recorded a function like this:

```
SeS("OK").DoAction();
```

This time, you will use the established name of the OK button object, but do something a little more interesting than its default action to demonstrate how to use Rapise.

(10) Move the cursor into the editor part of the Rapise and make sure you are editing the file called TwoDialogsLearn.js. At the moment, this file still looks something like this:

```
##### Script Steps #####
function Test()
```

```

{
}
g_load_libraries=["Generic"];

```

Between the open and close brace, add the following command:

```
SeS("OK").DoClick();
```

Hit the Play button and watch what happens.

The click will register as a command to the object and it will perform the action on the object.

While we have the context of this situation, let's complicate it just a little more to illustrate the intricacy as well as the flexibility of Rapise and SeS.

There is a method whose names looks interesting: DoLButtonDown().

If we were to invoke DoLButtonDown() on the "OK" object, we would expect this would be the same as DoClick().

However, go back to the AUT for a moment. Using the mouse, press the left mouse button over the OK button but don't take your finger off the left mouse button.

What happens is that the button takes its pressed state in appearance, but the button is not clicked.

The reason for this is that the DoClick() (or DoAction()) events cause the mouse button to be clicked as well as released.

Therefore, we would need to have a pair of events:

```
SeS("OK").DoLButtonDown();
SeS("OK").DoLButtonUp();
```

in order to make the "click" happen.

Try this in the test script you have created by adding those two lines of code in place of the DoClick() line.

It doesn't work!

Let's play a little with this problem.

When you press the Play button, leave the mouse alone. Just press the left mouse button on the Rapise Play button and take your hand away from the mouse.

The script does not press the OK button in the TwoDialogs AUT.

Now, press the Play button on the Rapise and **quickly** move the mouse to hover over the OK button in the TwoDialogs AUT.

Now it works!

What's going on here is that the DoLButtonDown() and DoLButtonUp() methods are pressing the mouse irrespective of where the mouse cursor is positioned.

The other functions, DoClick and DoAction are methods that are applied to the button and so they are applied to the button.

Before we can expect DoLButtonDown() and DoLButtonUp() methods to work, we have to first the mouse cursor to the button.

```
function Test()
{
  SeS("OK").DoMouseMove(25, 15);
  SeS("OK").DoLButtonDown();
  SeS("OK").DoLButtonUp();
}

```

will accomplish that.

Notice that Rapise will actually move the mouse to the coordinates (25, 15) within the OK button. Also notice that if you move the mouse while the test is playing, you will make the test fail.

As a last experiment in this arena, try moving the mouse outside the boundaries of the OK button object before calling the DoLButtonDown() function.

```
function Test()
{
  SeS("OK").DoMouseMove(250, 150);
  SeS("OK").DoLButtonDown();
  SeS("OK").DoLButtonUp();
}

```

Once again, the script will fail.

## Deal with a Simulated Object

[Top](#) [Previous](#) [Next](#)

Example: The toolbox of Microsoft's Paint utility (<c:\windows\system32\mspaint.exe>) is a compound object that contains custom buttons and is surrounded by a containing box. To understand this completely, start mspaint.exe from the Rapise.

Steps:

- (1) Open a new test under Rapise.
- (2) Press the Record/Learn button on the application bar.
- (3) When the "Select an Application to Record" dialog appears, select the Run Application tab. Enter mspaint in the "Full path to application" edit box. Press the Run button.

If you are unfamiliar with MS Paint, take a few minutes to play with it.

In particular, notice the toolbox that appears in the upper-left margin of the utility and the colour selection box that appears on the bottom-left of the application window.

- (4) Press Ctrl+5 to spy on the UI. Press Ctrl+G to spy on the Paint application. Notice several things about the behaviour of the MS Paint application under SesSpy.
  - (i) As you move the mouse inside the tools box, the entire surrounding box will show a red highlight but the individual tool buttons will not.
  - (ii) The same is true of the colour palette and the bottom-left of the screen.
  - (iii) As you move the mouse over the apparent buttons and controls, the information in the spy dialog is more sparse than for other applications. The tool buttons do not have default actions, and they are not identified as buttons. Rather they are identified only as "child" objects.

This combination makes it impossible for Rapise to identify and learn the objects as integral objects.

Furthermore, notice that as you change the size of the Paint window, the relative positions of the colour palette and the tool box change.

The only way in which Rapise can be 'taught' these controls (and others we will discover later) is by "simulating" them as though they were buttons that can accept commands such as the press event.

In fact, Rapise will recognize these non-objects without you having to take particular action. Let's discover this and what it means:

- (1) Open a new test under Rapise; call it MSPaint.
  - (2) Press the Record/Learn button on the application bar.
  - (3) When the "Select an Application to Record" dialog appears, clear all selection boxes in the library list box. You will have to scroll that section of the dialog box to make sure all selections are clear. We are choosing no loaded libraries so that Rapise will not be able to "cheat" and know about any objects on the screen.
  - (4) select the Run Application tab. Enter mspaint in the "Full path to application" edit box. Press the Run button.
- (Applications that reside in C:\windows\system32 can be started by their names because C:\windows\system32 must be in the system path.)
- (5) When the Recording Activity dialog is displayed, press Learn (Ctrl+2)
  - (6) Do a small amount of things in Paint. For example:
    - (i) Click on the light-grey colour in the palette.
    - (ii) Click on the tipping paint-can (Fill with colour).
    - (iii) Click on the empty canvas.
    - (iv) Click on the red colour in the palette.
    - (v) Click on the "A" tool (Text).
    - (vi) Click in the canvas and type a few characters, such as "Hello."
    - (vii) Click in a blank place under the tool button.
  - (7) Look at the Recording Activity dialog grid. It will be something like this:

#	Object	Action	Data	Comment
1	Colors	LClick	172,84	User clicks at: 172, 84 in 'Colors'
3	Fill with color	LClick	5,10	User clicks at: 5, 10 in 'Fill with color'
3	Simulated	LClick	422,111	User clicks at: 422, 111 in "
4	Colors	LClick	158,84	User clicks at: 158, 84 in 'Colors'
5	Tools	LClick	45,82	User clicks at: 45, 82 in 'Tools'
6	Simulated	LClick	336,89	User clicks at: 336, 89 in "
7	Tools	LClick	37,83	User clicks at: 37, 83 in 'Tools'
8	Text	LClick	373,169	User clicks at: 373, 169 in 'Text'
9	Simulated	LClick	267,165	User clicks at: 267, 165 in "
10	Untitled - Paint	SendK...	Hello	Type
11	Global	SendK...	{ENTER}	Type

Verify (Ctrl+1)   Learn (Ctrl+2)   Spy (Ctrl+5)   Pick Object...   Pause

Analog (Ctrl+4)   \_Simulated   Cancel   Finish (Ctrl+3)

Last captured: SeSSimulated (1)    Transparent

Notice that the two clicks in the canvas were recorded as "simulated" objects.

Notice also that the two pairs of clicks in the tools and colours sections were recorded as LClick (left click) in "Tools" and "Colors". However, there are no objects by these names. To find out where these pseudo objects came from, we need to look in the file MSPaint.objects.js (the name will be the name you gave the test project). The following excerpt from the MSPaint.object.js shows the start of the definition of the "Colors" object:

```

1  var saved_script_objects={
2  > "Colors":{
3  > "locations": [
4  > > {
5  > > > "locator_name": "Location",
6  > > > "location": {
7  > > > > "location": "4.4.4.1.4",
8  > > > > "window_name": "param:window_text",
9  > > > > "window_class": "param:window_class"
10 > > > }
11 > > },
12 > > {
13 > > > "locator_name": "LocationPath",
14 > > > "location": {
15 > > > > "window_name": "param:window_text",
16 > > > > "window_class": "param:window_class",
17 > > > > "path": [
18 > > > > > {
19 > > > > > > "object_name": "param:object_name",
20 > > > > > > "object_class": "param:object_class",
21 > > > > > > "object_role": "param:object_role"
22 > > > > > }},
23 > > > > > {
24 > > > > > > "object_name": "param:object_name",
25 > > > > > > "object_class": "param:object_class",
26 > > > > > > "object_role": "ROLE_SYSTEM_WINDOW"
27 > > > > > }},
28 > > > > > {
29 > > > > > > "object_name": "param:object_name",
30 > > > > > > "object_class": "AfxControlBar42u",
31 > > > > > > "object_role": "param:object_role"
32 > > > > > }},
33 > > > > > }],

```

(8) Press Ctrl+3 to end the recording.

## Technologies

[Top](#) [Previous](#) [Next](#)

This section focuses on specific technologies supported by Rapise.

## Adobe Flex

[Top](#) [Previous](#) [Next](#)

### Purpose

Rapise includes support for Adobe Flex applications executed

- inside Adobe Flash Player in Internet Explorer or Firefox
- and Adobe Integrated Runtime (AIR).

Flex versions 3 and 4 are supported.

### Usage

To test Flex applications, you must have **Flex Builder** installed. Link your application with **FlexAdapter.swc** (part of Rapise) and **automation\_agent.swc** and **automation.swc** (part of Flex Builder).

The compiler arguments for FlexBuilder 3 should look like:

```
-include-libraries "C:/Program Files/Adobe/Flex Builder 3/sdks/3.2.0/frameworks/libs/
automation_agent.swc" "C:/Program Files/Adobe/Flex Builder 3/sdks/3.2.0/frameworks/libs/
automation.swc" "C:/Program Files/Inflectra/Rapise/Extensions/Flex/FlexAdapter/bin/FlexAdapter.swc"
```

The compiler arguments for FlashBuilder 4 should look like:

```
-include-libraries "C:/Program Files/Adobe/Flash Builder 4/sdks/4.0.0/frameworks/libs/
automation_agent.swc" "C:/Program Files/Adobe/Flash Builder 4/sdks/4.0.0/frameworks/libs/
automation.swc" "C:/Program Files/Inflectra/Rapise/Extensions/Flex/FlexAdapter/bin/FlexAdapter.swc"
```

**Note:** You can avoid linking with third-party libraries if your application is browser-based and you will use [FlexLoader](#).

## Adobe Flash Player

Adobe Flash Player has restricted security settings for SWFs opened from the file system. To enable testing of such SWFs, their corresponding folders must be listed in the **FlashPlayerTrust** directory. You can find the FlashPlayerTrust directory here:

```
<system>\Macromed\Flash\FlashPlayerTrust
```

to enable testing just for the current user, use this FlashPlayerTrust directory:

```
<ApplicationData>\Macromed\Flash Player\Security\FlashPlayerTrust
```

To register your SWF just create a file with the name "*<name of your SWF>.cfg*" and put it in this directory. In the file, write a path to the SWF folder.

**Note:** If you do not have *FlashPlayerTrust* directory in one of locations listed above then you will have to create missing directories yourself.

## Adobe AIR

To record and playback tests for Adobe AIR application you need to run the application manually. E.g.:

```
"C:\Program Files\Adobe\Flex Builder 3\sdk\3.2.0\bin\adl.exe" C:\Program Files\Inflectra\Rapise\Samples\AdobeFlex3\AUTFlexAIR\bin-debug\AUTFlexAIR-app.xml
```

## Sample Applications and Test

Two sample Flex 3 applications are available with the Rapise installation. They can be found at:

```
<Rapise install dir>/Samples/AdobeFlex3/AUTFlexFP/bin-debug/AUTFlexFP.html
```

and

```
<Rapise install dir>/Samples/AdobeFlex3/AUTFlexAIR/bin-debug/AUTFlexAIR-app.xml
```

The binaries and source are both provided.

One sample Flex 4 applications is available with the Rapise installation. It can be found at:

```
<Rapise install dir>/Samples/AdobeFlex4/AUTFlexFP4/bin-debug/AUTFlexFP.html
```

The binaries and source are both provided.

Sample tests for the sample applications can be found in *<Rapise install dir>/Samples/AdobeFlex3* and *<Rapise install dir>/Samples/AdobeFlex4*. To select the target for testing edit the following line in *AdobeFlex.user.js* file:

```
/**
 * Select flex target for testing.
 */
var testTarget = "FlexIE"; // "FlexAIR", "FlexFirefox", "FlexIE"
```

**Note:** If you choose AIR target, please, do not forget to run the sample application before executing the test.

## See Also

- [Tutorial: Testing Adobe Flex Applications](#)

## Cross Browser Testing

[Top](#) [Previous](#) [Next](#)

You can run your recording in a different browser than the one in which it was recorded. You can also run the recording in multiple browsers in succession. Both options require modification of the script. The necessary modifications are described below.

### Selecting a new Playback Browser

First, open the script for your test using the [Test Files Dialog](#). Locate the line where **g\_load\_libraries** is initialized.

If you recorded your script in IE you will see:

```
g_load_libraries=["Internet Explorer HTML"];
```

If you recorded it in Firefox, you will see:

```
g_load_libraries=["Firefox HTML"];
```

Erase the current library name (**Firefox HTML** or **Internet Explorer HTML**) and replace it with the other option. [Playback](#) the script normally.

### Playback in Multiple Browsers

Executing a test in multiple browsers is slightly more complicated. We recommend to use [sub-tests](#) to properly organize the multi-browser testing.

1. Record [base](#) test. Put all the recorded actions into a User-defined function and place it into **<testname>user.js** file. For example, function `Login()` inside file **MyTest.user.js**.
2. [Create Sub-Test](#) for **IE** re-using objects and functions from the base test
3. Modify script file in sub-test as follows:

```
function Test()
{
  // Re-use 'Login()' scenario from parent test
  Login();
}

g_load_libraries=["Internet Explorer HTML"];
```

4. Create Sub-Test for **Firefox** re-using objects and functions from parent test
5. Modify script file in subtest as follows:

```
function Test()
{
  // Re-use 'Login()' scenario from parent test
  Login();
}

g_load_libraries=["Firefox HTML"];
```

As a result you have a test for 2 browsers: **IE** and **Firefox**. Each browser is defined by a library in a corresponding sub-test. Rapise contains [Cross Browser](#) sample using this approach.

## Qt Framework

[Top](#) [Previous](#) [Next](#)

### Purpose

Rapise includes support for testing applications written using the Qt Framework written using QWidget controls.

### Usage

Rapise fully supports the test automation of Qt based applications. To ensure that Rapise can access the UI elements and properties in the Qt application, MSA (Microsoft Active Accessibility) support for your Qt application must be enabled. This provides additional information on Qt UI elements to automation software like Rapise and can be accomplished by shipping and loading the "Accessible Plug-in" included in the Qt SDK (Software Development Kit) with the Qt application under test (see below).

#### Loading Accessible Plug-in for your Qt application:

1. Copy the "accessible" directory (and all its contents) from the **Qt SDK** (used to build the application under test) installation folder to the folder of the automated application (e.g. "**Program Files/Your-Application/plugins**"). If you do not have access to the Qt SDK which the Qt application is developed with, please contact the developer of the application and request the "accessible" directory from him.
2. Create a file called "qt.conf" (or append if the file already exists) in the root directory of the automated application (e.g. "**Program Files/Your-Application**") with following content (copy and paste the following two lines):  
[Paths]  
Plugins = plugins

## Java AWT/Swing Testing

[Top](#) [Previous](#) [Next](#)

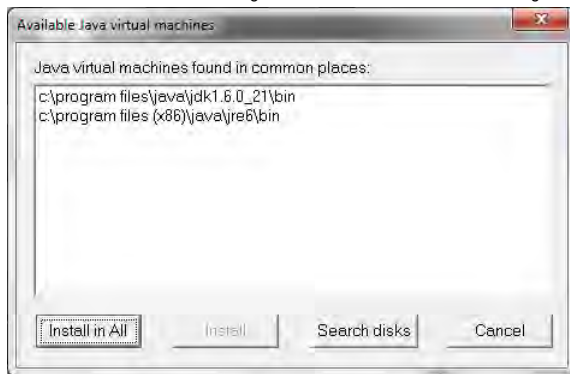
### Purpose

Rapise supports the testing of Java applications using either the Abstract Window Toolkit (AWT) or Swing graphic user interface toolkits. For maximum flexibility, Rapise can connect to your choice of JVM.

### Usage

In order to use a particular Java Virtual Machine (JVM) with Rapise you need to install Java Bridge into it. Installation process consists of several simple steps:

1. Click the Options icon in the Tools group of the main Rapise ribbon. That will bring up the [Options dialog](#).
2. Click on the Tools > Java Settings button. This will launch the Java Bridge installation dialog:



3. Choose target JVM in the list of available Java machines and press Install button
4. Verify that installation is successful

## Extensibility

[Top](#) [Previous](#) [Next](#)

The **Extensibility** section is for experienced Rapise users who want to extend capabilities of the tool.

## Tutorial: Custom Library

[Top](#) [Previous](#) [Next](#)

In this section, you will learn how to create a **Custom Library** and add support for a third-party GUI control to Rapise. We will be using a demo application called **CustomControlApp**. Our Custom Library will be simple. It will allow to Record and Learn objects of **CustomListboxControl** type and also Playback actions for this type of objects. This tutorial is complemented by a ready test **CustomControlTest** which you'll be able to examine and run.

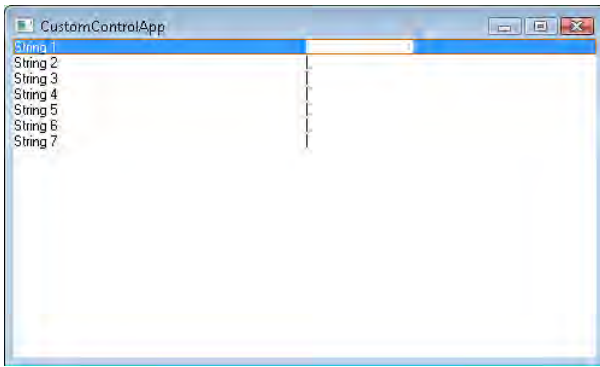
#### Tutorial Data

- CustomControlApp folder: C:\Program Files\Inflectra\Rapise\Samples\Extensibility\CustomLibrary\CustomControlApp. You may build this application yourself in Microsoft Visual Studio (C++) or use ready executable: <CustomControlApp folder>\Release\CustomControlApp.exe
- CustomControlTest folder: C:\Program Files\Inflectra\Rapise\Samples\Extensibility\CustomLibrary\CustomControlTest
- CustomLibrary file: C:\Program Files\Inflectra\Rapise\Samples\Extensibility\CustomLibrary\CustomLibrary.js

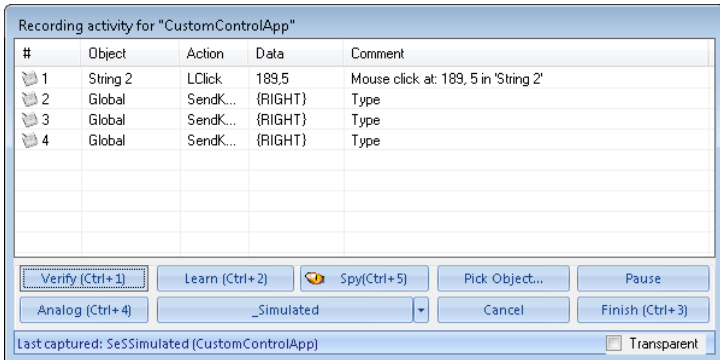
If you prefer active experimentation learning style you may first skip to subsection 9 and after playing with the ready test and library start reading from the beginning.

### 1. Application Under Test

CustomControlApp contains an object of type CustomListboxControl. The control is similar to a single-select listbox, but each line item has a corresponding **progress bar** indicator indicating a current value. Using the left/right cursor keys you can change the value of the currently focused item.



If you try to record a test for CustomControlApp using just Generic library you'll see that CustomListboxControl is treated as Simulated Object and all interactions with it are recorded as mouse clicks and key presses. For some tests such functionality is sufficient, but if you want to be able to recognize CustomListboxControl as a list, get its items, select an item by name, set value for a particular item you need to create a Custom Library.



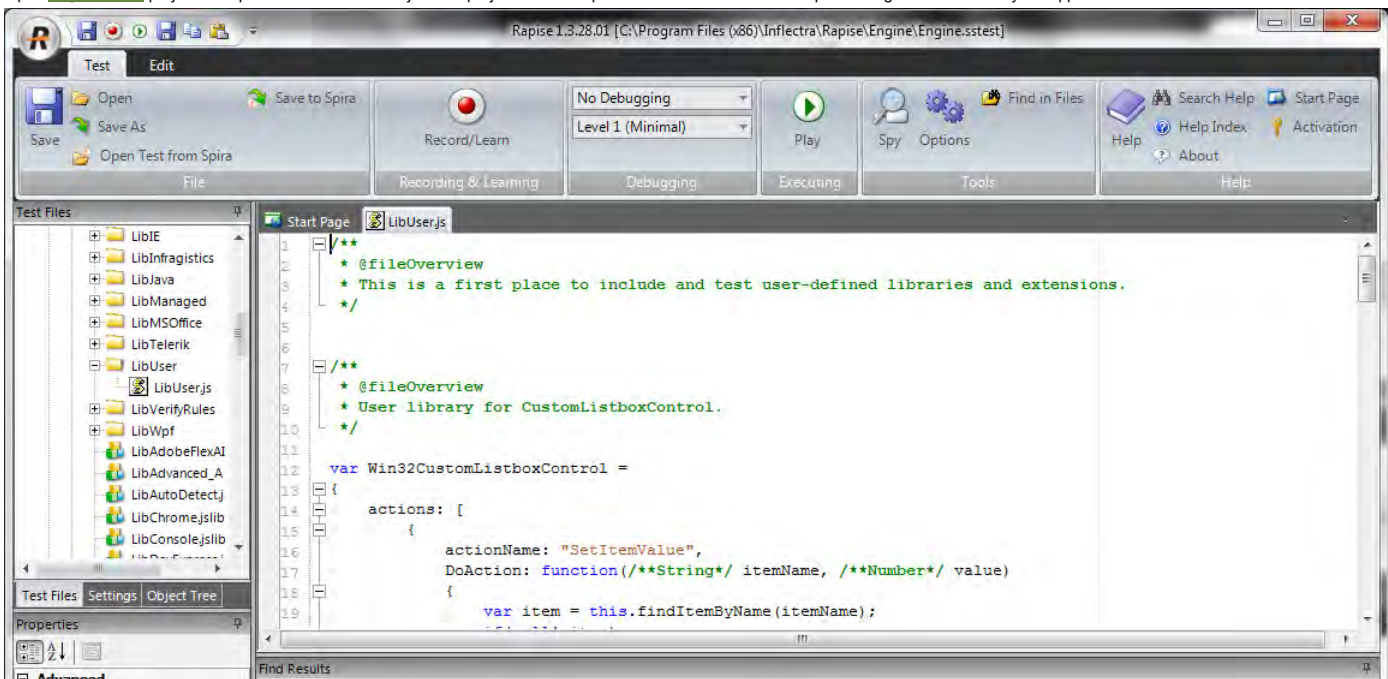
## 2. LibUser

A good place to start implementing a Custom Library is empty LibUser library included into Rapise. All Rapise libraries live in *C:\Program Files\Inflectra\Rapise\Engine\Lib* folder and LibUser is not an exception. LibUser library consists of two files:

1. *C:\Program Files\Inflectra\Rapise\Engine\Lib\LibUser.jslib* which is a library declaration file.
2. *C:\Program Files\Inflectra\Rapise\Engine\Lib\LibUser\LibUser.js* which is a library definition file.

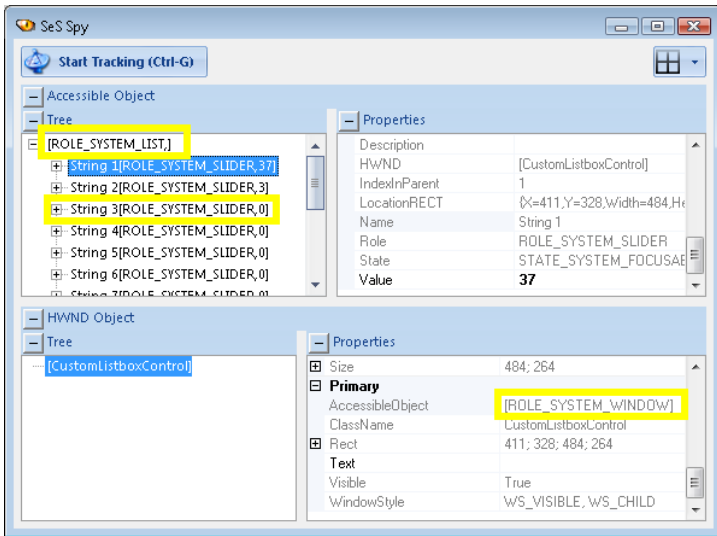
## 3. Open Engine.sstest

Open [Engine.sstest](#) project in Rapise. Then find LibUser.js in the project tree and open it. You are about to start implementing a Custom Library to support CustomListboxControl.



## 4. Analyze CustomListboxControl in Spy

Launch CustomControlApp and open [Spy](#). Spy on CustomListboxControl. It is easy to see that CustomListboxControl has the following accessibility tree: ROLE\_SYSTEM\_WINDOW top node contains ROLE\_SYSTEM\_LIST child that in its turn may contain zero to many ROLE\_SYSTEM\_SLIDER nodes.



## 5. Create Matcher Rule for CustomListBoxControl

With knowledge of CustomListBoxControl accessibility tree we can create a **matcher rule** that will make CustomListBoxControl recognizable by Rapise. Write the following code into LibUser.js:

```
new SeSMatcherRule(
{
  object_type: "CustomListBoxControl",
  object_flavor: "List",
  behavior: [Win32ItemSelectable, Win32CustomListBoxControl],
  role: "ROLE_SYSTEM_WINDOW",
  or_rules: [
    {
      role: "regex:ROLE_SYSTEM_LIST",
      save_to: "list",
      or_rules: [
        {
          role: "ROLE_SYSTEM_SLIDER",
          zero_to_many: true,
          save_to: "items"
        }
      ]
    }
  ]
});
```

Each matcher rule (instance of SeSMatcherRule) is a tree like structure that describes a particular GUI control type. Each node in this tree is a rule object that is defined by the following simplified grammar:

```
or_rules: (rule)+
and_rules: (rule)+

rule:
  role
  [save_to]
  [zero_to_many]
  [or_rules]
  [and_rules]
```

- object\_type**: the string that uniquely identifies this matcher rule and designates type of the control
- object\_flavor**: visual type of the control, it is used to show an appropriate icon in the [Object Tree](#) and to filter actions and properties in composite behavior patterns (like in Adobe Flex, see [FlexActions.js](#))
- behavior**: array of behavior patterns that define object actions, properties and events.
- role**: accessibility role of the corresponding node in the accessibility tree of the control. The role equals to a Role of the accessible element as displayed in the Spy.
- or\_rules**: array of rules (defining child nodes) joined with logical OR. Any OR rule can be satisfied to consider child nodes matched.
- and\_rules**: array of rules (defining child nodes) joined with logical AND. All AND rules must be satisfied to consider child nodes matched.
- save\_to**: SeSObject created for accessibility tree node corresponding to this rule is assigned to the field with "save\_to" name of the top level SeSObject. I.e. if rule has save\_to: "items" element then you can access learned element using SeS('ObjID').items. In many cases such named fields are used in behavior patterns.
- zero\_to\_many**: if this property is present in the rule and set to 'true' then it means that parent rule may contain from zero to many of child nodes that match this rule.

## 6. CustomListBoxControl Behavior

After defining the matcher rule we can proceed to **behavior patterns**. Behavior patterns operate with **SeSObject** contents, so they should not be aware about accessibility tree of the underlying GUI control and thus the same behavior pattern can be assigned to different matcher rules. There are a plenty of behavior patterns defined in SeSBehavior.js. After looking at those patterns it is possible to notice that Win32ItemSelectable pattern is the one that perfectly suites for capturing selection accessibility events and for selecting list items. This pattern contains OnSelect event that is called during recording when an item is selected in list and DoSelectItem action used to select desired item during playback.

But using just Win32ItemSelectable behavior pattern is not sufficient. It does not support recording of progress bar value change events and it does not support setting progress bar value during playback. That is why we need to define new behavior pattern: Win32CustomListBoxControl. Look at its code:

```
var Win32CustomListBoxControl =
{
  actions: [
    {
      actionName: "SetItemValue",
      DoAction: function(**String*/ itemName, /**Number*/ value)
      {
        var item = this.findItemByName(itemName);
        if(null!=item)
        {
          item.getTopObject().instance.HWND.SetForegroundWindow();
          item.instance.Value = value;
          return true;
        }
        return false;
      }
    },
    {
      actionName: "GetItemValue",
      DoAction: function(**String*/ itemName)
      {
        var item = this.findItemByName(itemName);
        if(null!=item)
        {
```

```

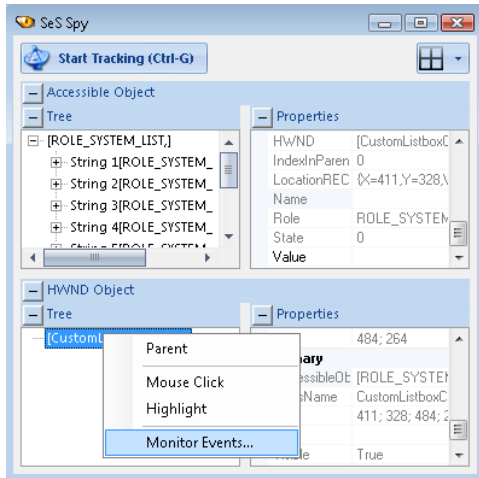
        return item.instance.Value
    }
    return null;
}
},
events:
{
    OnValueChanged: function (**SeSObject*/ param)
    {
        var itemName = param.name;
        if (!Log2("OnValueChanged:"+itemName);
        var item = this.findItemByName(itemName);
        if (null!=item)
        {
            var value = item.instance.Value;
            RegisterAction(this, param.name, "SetItemValue", parseInt(value), "Set item:"+param.name+" to "+value+" in '"+this.name+"'");
        }
        return;
    }
}
};

```

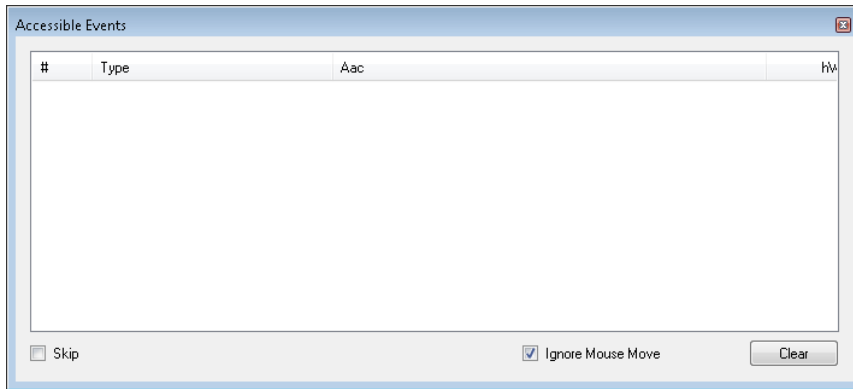
During recording process OnValueChanged function captures progress bar change events and calls RegisterAction function that adds SetItemValue action to the test.

## 7. CustomListboxControl Specific Accessibility Events

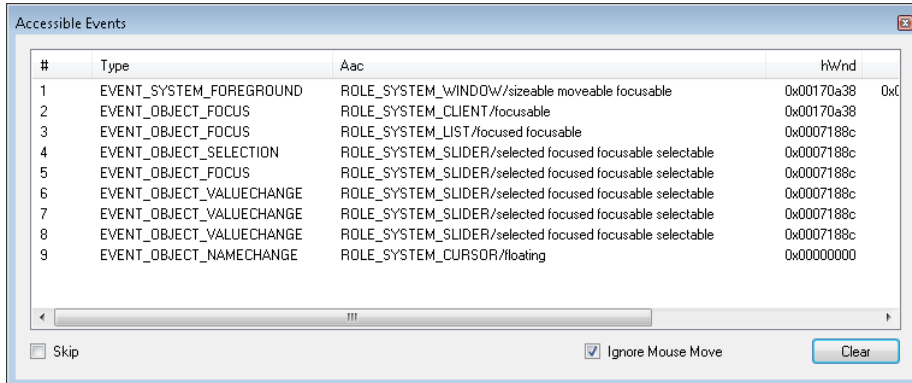
What accessibility events are fired when user changes progress bar value? Use Spy to find out. Launch CustomControlApp and open Spy window. Spy on CustomListboxControl. Choose Monitor Events...



You will see Accessible Events dialog:



Select an item in CustomControlApp and advance its progress bar using right key. Accessible Events dialog will show you captured events:





You can see that changing progress bar leads to generation of EVENT\_OBJECT\_VALUECHANGE events.

Not all accessibility events are processed and propagated by Rapise engine. EVENT\_OBJECT\_VALUECHANGE is one of such events. To consume this event and make an appropriate call to OnValueChanged of Win32CustomListBoxControl you need to add and register **custom accessibility event handler**:

```
function CustomRegisterAccessibleEvent(evt, etxt)
{
    if (etxt.indexOf("EVENT_OBJECT_VALUECHANGE")>=0)
    {
        var ao;
        try
        {
            ao = evt.AccessibleObject;
            if (!SeSisValidObject(ao)) return false;
        }
        catch(e)
        {
            Log("Error getting event object:"+e.Description+"/"+etxt);
            return false;
        }

        var ro = SeSCacheAccessibleObject(ao);
        if (13 && ro) Log3("CustomListBoxControl: " + ro.toString());
        if (ro != null && ("OnValueChanged" in ro))
        {
            ro.OnValueChanged();
        }

        return true;
    }
    return false;
}

g_customEventHandlers.push(CustomRegisterAccessibleEvent);
```

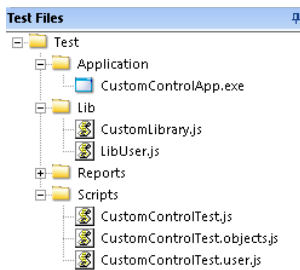
## 8. Record and Playback

Now you are ready to record and playback a test. Just remember that in Select an Application to Record dialog you need to uncheck Auto library and select User and Generic libraries.

Library	Description
<input checked="" type="checkbox"/> User	Default user-defined library
<input type="checkbox"/> Console	Console Application
<input checked="" type="checkbox"/> Generic	Generic library contains basic definitions for most comm...
<input type="checkbox"/> MSOffice	Microsoft Office with Accessibility

## 9. CustomControlTest

This tutorial is complemented by a ready test CustomControlTest which you can examine and run. Open CustomControlTest in Rapise and place contents of CustomLibrary file into LibUser.js file (C:\Program Files\Inflectra\Rapise\Engine\Lib\LibUser\LibUser.js). LibUser.js is added to CustomControlTest, so you can populate it with CustomLibrary code right in Rapise.



**Tip:** It is possible to launch CustomControlApp right from Rapise, just double click on CustomControlApp.exe in the project tree.

## 10. Wrap-up: Implementation Sequence

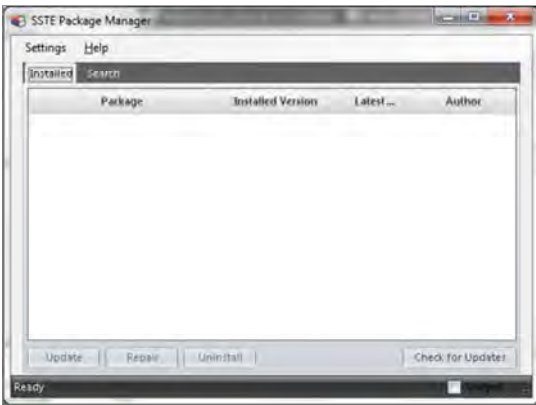
Full support for a custom object requires support for Record, Learn and Playback. Let's go over created library and specify the purpose of each component in it.

- **Matcher Rule**:- it is used to recognize the object inside an application, required for Record, Learn and Playback.
- **Events in Behavior Patterns**: handling events is required for Record.
- **Actions in Behavior Patterns**: actions are used to examine or change state of the control, required for Playback.
- **Custom Accessibility Event Handler**: required for Record if some important events are not processed by Rapise engine as needed.

## Package Manager

[Top](#) [Previous](#) [Next](#)

### Screenshot



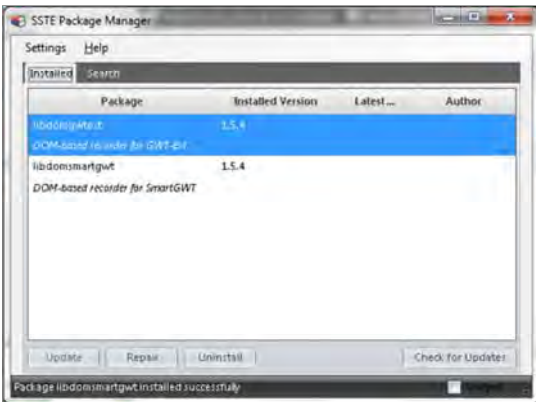
### Purpose

The **Package Manager** allows you to download **third-party libraries** from the Rapise QA-Forge. This website hosts various third-party libraries not provided by Inflectra.

### How to Open

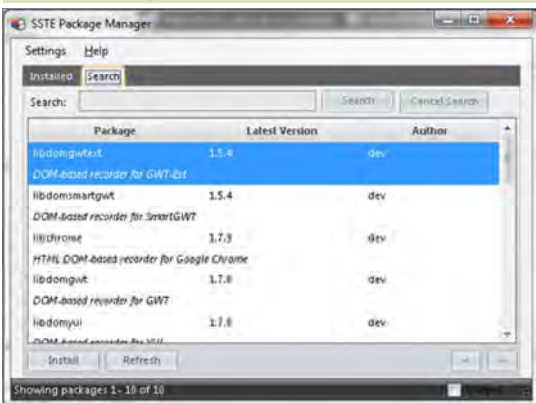
Click on the **Package Mgr** icon in the main Rapise Test Ribbon.

### Installed Packages



The **Installed** tab displays a list of the third-party libraries (from the QA-Forge) that are currently installed in Rapise. It displays the name of the package, together with the version number and author. You can click on the **Check for Updates** button to see if there are any newer versions of the currently installed packages.

### Search for Packages



The **Search** tab displays a list of the third-party libraries (from the QA-Forge) that are not currently installed in Rapise, but are available in the QA-Forge. It displays the name of the package, together with the version number and author. You can click on the name of the package followed by the **Install** button to download and install the package.

## 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.

Rapise<sup>®</sup> and Inflectra<sup>®</sup> are either trademarks or registered trademarks of Inflectra Corporation in the United States of America and other countries. 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](mailto:support@inflectra.com)