

25 Tips for Creating Effective Load Test Scripts using Oracle Load Testing for E-Business Suite and Fusion Applications.

ORACLE WHITE PAPER | SEPTEMBER 2014

Table of Contents

| | |
|--|----|
| Product Overview | 1 |
| Introduction | 1 |
| Tip 1: Be aware, the culprit of Failed to Solve variable playback error may be on the application side | 2 |
| Tip 2: Keep in mind that the variable name in the playback error may be unrelated to the real cause of the failure | 3 |
| Tip 3: Learn how OpenScript substitutes dynamic strings in HTTP requests | 4 |
| Tip 4: Remember three key correlation variable types; Regular Expression, XPath and JavaScript | 5 |
| XPath Variable | 5 |
| Regular Expression Variable | 5 |
| JavaScript Variable | 6 |
| Tip 5: Notice that script editing can be done better in Tree view rather than Java Code view in most cases | 6 |
| Tip 6: Take advantage of the intuitive GUI wizard to add custom correlations | 7 |
| Tip 7: Understand why OpenScript saves two sets of navigation information; Recorded and Playback | 8 |
| Tip 8: Perform Compute Similarity over playback results to unveil hidden failures | 9 |
| Tip 9: Make creative use of wildcard and regular expressions extracting dynamic values from source contents | 10 |
| Analyze the problem: What caused the "Failed to solve" playback error? | 10 |
| Resolve the problem: Add Wildcard to XPath variable | 14 |
| Alternative Solution: Replace with Regular Expression variable | 14 |
| Tip 10: Beware of client side cookies dynamically set by JavaScript | 16 |
| Compare Request Headers | 16 |
| Compare Cookie Strings | 17 |
| Tip 11: Be prepared for emergency script recreation | 19 |
| Tip 12: Learn why script scalability and reusability is always a trade off | 20 |
| Tip 13: Create load test script with proper step group names | 21 |
| Tip 14: Request static resources concurrently just like browsers do | 22 |
| Tip 15: Use Web recording mode for EBS and HTTP mode for ADF application | 23 |
| Tip 16: Capture web transactions from Chrome browser | 25 |
| Tip 17: Construct REST requests with OpenScript's built-in HTTP API | 26 |
| TIP 18: Take advantage of helpful utilities constructing REST requests | 28 |

| | |
|---|----|
| Tip 19: Record mobile applications directly from mobile devices | 30 |
| Record Mobile Transactions with OpenScript proxy | 30 |
| Record Mobile Transactions with WiFi tethering | 31 |
| Set up WiFi hotspot from Ethernet | 32 |
| Connect to the hotspot from the mobile device | 34 |
| Start OpenScript recording | 35 |
| Set proxy configuration on the mobile device | 36 |
| Tip 20: Record a test flow in two scripts to pinpoint dynamic session parameters | 37 |
| Tip 21: Record a flow twice into a single script to isolate parameter changes by iterations | 38 |
| Tip 22: Create a series of identical EBS test users for use within Oracle Load Testing | 39 |
| Tip 23: Import PFX file to test applications with client certificate | 41 |
| Save the PFX file from the client | 41 |
| Record with OpenScript | 41 |
| Playback with OpenScript | 41 |
| Playback with multiple client certificates | 42 |
| Tip 24: Add databank only after a playback using original data is successful | 43 |
| Tip 25: Tune script for better performance before adding to the load test scenario | 43 |
| Conclusion | 45 |

Product Overview

Oracle Application Testing Suite, or ATS, is an integrated, full lifecycle solution which ensures application quality and performance with complete end-to-end testing and test management capabilities. Oracle Application Testing Suite helps deliver high quality applications with three separately licensed products:

- » **Oracle Functional Testing** for automated functional and regression testing of Web applications, Web Services, Oracle packaged applications and databases.
- » **Oracle Load Testing** for automated load and performance testing of Web applications, Web Services, Oracle packaged applications and databases.
- » **Oracle Test Manager** for process management throughout the testing lifecycle, including test planning, requirements management, test management, test execution and defect tracking.

Introduction

This white paper focuses on robust and successful creation of load test scripts, using OpenScript to test **Oracle E-Business Suite** and **Oracle Fusion applications**. The Oracle Load Testing product consists of two main components, **OpenScript** to create test scripts, and **Oracle Load Testing** to run load test sessions. Load test scenarios include one or more scripts which replicate real user transactions, and are used to simulate application's various load profiles, such as peak production workload.

This document comprises 25 topics to share with Oracle Load Testing users who have requirements to create test scripts that run transactions against critical business applications. Such applications may have servers returning complex and large page contents, or clients sending hundreds of post data parameters in a single HTTP¹ request. Each request may contain dynamic session values that change over sessions, iterations or with different data. These applications may engage dynamic cookies set by JavaScript, or involve client side certificate to secure user transactions. Testing may need to be conducted under severe time sensitive circumstances to meet go-live deadlines, while frequent changes are still being made to the application's code. Test requirements may include executing REST² API³, or running web transactions from mobile tablets.

This white paper was written by a group of experienced ATS product managers who have years of vast exposures deploying load testing projects. Each of the discussions is derived from the real life customer use cases and summarized as a "Tip" to share our expertise with the readers. The target audience of this paper is for OpenScript users who are starting up with load test scripting. Therefore this paper aims to explain the load testing concepts in the easiest language possible.

Although the paper's primary target test applications are E-Business Suite and Fusion applications, the subjects discussed are commonly effective among any web applications and not tied to Oracle packaged applications.

1 HTTP: Hypertext Transfer Protocol

2 REST: Representational State Transfer

3 API: Application Programming Interface

Tip 1: Be aware, the culprit of Failed to Solve variable playback error may be on the application side

The “**Failed to solve variable**” error is typically the first scripting problem a user encounters when creating a load test script in OpenScript. Failed to solve errors can occur for two main reasons:

- A. During script playback, the application returned a legitimate, but slightly different page than that of recorded. Imagine when a different user logged in to a user profile page that displays user’s personal information. The search string in the correlation variable was not generic enough to successfully parse server response. For example, search string may include a string that is specific to the user used during the recording.
- B. During script playback, the application returned a page that is totally different than that of recorded, thus the matching string does not exist in the playback content. The reason why the unexpected page was returned could be a server side error, or an invalid request sent by the script.

Case A is typically seen when page content is dynamic, often when a databank that parameterizes the user entry is added to the script. For example, a script was recorded against a shopping website using the user account “Tom”. The web site returns a page based on Tom’s purchase history. But if a different user account, “Jennifer”, is used to sign in at the script playback, the website returns a page based on Jennifer’s preferences, which is different from Tom’s. From the application’s perspective, this is a legitimate difference, however from the script perspective, this is nothing but a different page returned from the application.

Load testing scripts in OpenScript employs a concept of **correlation**, which is a method to identify a parameter that changes in different sessions, and to substitute that parameter with a valid value dynamically during playback. The correlation expects to find a matching string from the page content, using search strings defined in the **correlation variables**. The search strings vary from the types of variables, and can be **Regular expressions**, **XPath expressions**, or **JavaScript index**. The variables can find the value as long as their string extraction logic is valid against the page content. However, when a server returns a different page content, it can break the variable’s string extraction logic. Even with a slight difference it can end up with a Failed to solve variable error. The solution to this problem is to strengthen the search logic so that it tolerates the content differences of the different user accounts.

On the other hand, in many times the root cause of the “Failed to solve variable” playback error is due to Case B, which is when the server returned a significantly different page content from that of the recording. This is often an error page, such as “Internal Server Error 500”. The error page can be returned for two reasons. 1) The application is not responding because of its own issues, or 2) the script needs better correlation. Note that even when the root cause resides in the application, “Failed to solve variable” error can still be reported because from a correlation variable point of view, it is simply unable to find a matching string from the page content returned.

The troubleshooting steps for Case B are described below.

TABLE 1. TROUBLESHOOTING STEPS WHEN ERROR PAGE IS RETURNED IN PLAYBACK

| Step1: Isolate the application side issues | Step2: Examine the script, find any missing or incorrect correlations and apply the fix. |
|---|--|
| Verify the application state to segregate server’s internal problems, network issues, or scheduled downtimes. Contact application administrators, or access the application manually using the browser to ensure the application is up and running. | Find and fix any missing or invalid correlations. A request may include a dynamic session value, which changes every time a user signs in to the application. If a script is created without correlating the dynamic session value, an obsolete value will be used every time the script is replayed. This can result in the application returning an error page with a message, “Your session has expired”. |

To summarize, the right approach to solving “Failed to solve variable” error is to first verify whether the page content returned at the playback is legitimate or not in order to isolate server side issues. Next, examine the script correlations to ensure that valid HTTP requests are being sent. This is the first tip to develop an effective troubleshooting methodology for load test scripting.

TABLE 2. ROOT CAUSE FOR FAILED TO SOLVE VARIABLE ERRORS AND APPROACH TO THE SOLUTIONS

| Failure Pattern | Recorded Content | Playback Content | Why the variable fails to locate the text from the page content | How to resolve the problem |
|-----------------|--|--|---|---|
| Case A | <p>Welcome Tom!</p> <p>Target String</p> | <p>Welcome Jennifer!</p> <p>Target String</p> | OpenScript fails to solve a variable because the target string is at an unexpected location within the playback content. This is a script problem. | Redefine the variable definition in the script so that it can locate the target string regardless of its location within the page content. |
| Case B | <p>Welcome Tom!</p> <p>Target String</p> | <p>Error Page!</p>  | OpenScript fails to solve a variable because the target string does not exist in the playback content. This could be either a server side problem, or a script problem. | First, isolate the server side issues. Once that is cleared, verify if the script has any missing or incorrect correlations which resulted in sending an invalid request. |

The first tip briefly touched upon the surface of various topics that will be discussed extensively in the remaining tips of this paper. Terms and concepts such as correlations, variables, as well as OpenScript’s built-in debugging utilities will be discussed in depth and reiterated throughout the paper.

Tip 2: Keep in mind that the variable name in the playback error may be unrelated to the real cause of the failure

When a script playback fails, the failure result reports an error in its session report. Typically the error is “Failed to solve variable” and includes a variable name that caused the playback failure. However, in some cases the message can mislead the user from going towards a correct troubleshooting path.

When the server returns a significantly different page content from that of the recording, the variable name in the error message is not directly related to the root cause of the problem. For example, a script may report a playback failure with the message below.

Failed to solve variable FormsSessionID using regex JServSessionIdforms=(.)*(?:\r)

The message indicates the variable “FormsSessionID” was not created during the playback thus the script playback failed. Users may be misled to think they need to fix the variable’s regular expression pattern so that it can capture the value properly from the page content, or be tempted to remove this variable to see if the problem goes away.

However, please recall the discussion in the previous section. This could be an example of Case B where the server returned a wrong page. The variable was not created, because the matching string does not exist in the page content. Therefore the real cause of the playback failure is the error page returned by the application, and the variable “FormsSessionID” that shows up in the error message has no relation to the root cause of the failure. If this variable is removed, then another variable defined in the page will fail at the playback. Be aware of this possibility and do not be misled by the error message description. The real problem may reside in other factors and not the particular variable mentioned in the error message.

Tip 3: Learn how OpenScript substitutes dynamic strings in HTTP requests

The statements below are examples of the HTTP commands used in OpenScript for a Load Testing script.

TABLE 3. EXAMPLES OF HTTP API COMMANDS GENERATED IN THE SCRIPT

| HTTP Navigations | |
|------------------|---|
| 1 | <pre>http.get(81, "https://myserver.oracle.com/", null, null, true, "UTF8", "UTF8"); { http.solveXPath("web.framesrc.ActionList_1", "//FRAME[@name='ActionList']/@src", "applist.html", 0, EncodeOptions.None); http.solveXPath("web.framesrc.Title_1", "//FRAME[@name='Title']/@src", "apptitle.html", 0, EncodeOptions.None); http.solveRefererHeader("referer.httpsmyserveroraclecom", "/"); }</pre> |
| 2 | <pre>http.get(85, "https:// myserver.oracle.com /{{web.framesrc.ActionList_1,applist.html}}", null, null, true, "UTF8", "UTF8")</pre> |
| 3 | <pre>http.get(89, "https://myserver.oracle.com/", /{{web.framesrc.Title_1,apptitle.html}}", null, http.headers(http.header("Referer", "https:{{ referer.httpsmyserveroraclecom,}}/ myserver.oracle.com /", Header.HeaderAction.Modify)), true, "UTF8", "UTF8");</pre> |

At the script playback, OpenScript executes the first navigation statement in three steps. First, it sends a GET request for a web page called “myserver.oraclecom”. Second, OpenScript receives the response content back from the application. Third, it parses the HTML content and extracts the dynamic values, then stores them in variables so that it can be used when string substitutions are required later in the script.

One of the three variables defined in Navigation 1 is “web.framesrc.ActionList_1”. This is an XPath variable that defines the string extraction in the XPath expression. During the script playback, OpenScript runs a search against the source content and looks for a “Frame” object, which has an attribute “Name” with a value “ActionList”. Once the object is identified, the variable finally extracts a value of the “SRC” attribute of that object, and stores the value into a variable called “web.framesrc.ActionList_1”. The value will be stored in the variable during the script playback, and used to substitute the recorded value in the subsequent navigations.

The statements in Navigation 2 and 3 show string substitutions of the variables defined in Navigation 1. The variables “web.framesrc.ActionList_1” and “web.framesrc.Title_1” partially substitute the navigation URL string, and the variable “referer.httpsmyserveroraclecom” partially substitutes the referrer URL string defined in Navigation 3. The substituted strings are displayed in the curly braces.

But why does the URL string need to be substituted? Because web applications may expect clients to use different session parameters each time requests are sent to the server, and without substitution the recorded value that is no longer valid will be used. The application in the sample statement above requests the client to use different URL string for every session, and that is why the substitution is required.⁴

During an HTTP session, web applications pass dynamic session strings to the browser by means of hidden fields in the HTML, cookies, or headers and use those values to identify the client to keep the session alive. Without substitutions, the script playback will use obsolete values to send a request that could be taken as an invalid request by the server. Therefore the dynamic values need to be extracted from the application and substituted in the HTTP requests so that OpenScript always uses the valid runtime value at the playback. This is the third tip, which explains the reason for **correlations** in load test scripts.

⁴ OpenScript auto-correlates dynamic strings according to the correlation rules configured in the library. In this example, the variables substitute partial URL strings but many cases URL is static and correlation is not required. Query Strings or POST data parameters are where usually values are dynamic and require correlations.

Tip 4: Remember three key correlation variable types; Regular Expression, XPath and JavaScript

OpenScript provides a number of different correlation variables in its built-in HTTP API. There are three main types of variables that the load script users need to know; Regular Expression, XPath and JavaScript correlation variables. All three types of variables are used to achieve a common goal; to identify and extract the dynamic value from the response content, store the value into the variable, and use that value to substitute the request parameters in the subsequent navigations. The only clear difference is how they extract strings, as each variable type uses a different method to extract dynamic values from the source content.

XPath Variable

XPath correlation variable uses XPath expression to extract values from the source. XPath variables are generated by OpenScript's auto-correlation feature at the time of the recording. The statements start with "**http.solveXPath**".

TABLE 4. EXAMPLE OF SOLVE XPATH VARIABLE STATEMENT AND TARGET SOURCE

| | |
|--------------------|---|
| Variable Statement | <code>http.solveXPath("web.framesrc.ActionList_1", "//*[@name='ActionList']/@src", "applist.html", 0, EncodeOptions.None);</code> |
| Response Content | <code><frame src="applist.html" name="ActionList" scrolling="AUTO"></code> |

In the statement above, SolveXPath variable "**web.framesrc.ActionList_1**" uses XPath expression "**//*[@name='ActionList']/@src**" to extract a value from the last response content. The variable queries a "frame" object whose attribute "name" is "ActionList". Upon finding the frame object, the variable extracts the value of "src", which is "applist.html". Users who are familiar with Structured Query Language (SQL) can think this way - If the XPath logic was translatable to a SQL statement, it would look like: **select value from FRAME.src where name=ActionList**.

Regular Expression Variable

Regular Expression variable uses regular expression patterns to extract dynamic values. Just like the XPath variables, Regular Expression variables are automatically generated during the script recording. In addition, they can be created manually by users in the **Substitute Variable** wizard to add additional correlations to the script.

TABLE 5. EXAMPLE OF SOLVE (REGULAR EXPRESSION) VARIABLE STATEMENT AND TARGET SOURCE

| | |
|--------------------|---|
| Variable Statement | <code>http.solve("viewstate_1", "<input type='hidden' name='javax.faces.ViewState' value='(.+?)'>", "", true, Source.Html, 0, EncodeOptions.None);</code> |
| Response Content | <code>< input type="hidden" name="javax.faces.ViewState" value="!p33y2cxbq"></code> |

Regular expression variables use **http.solve** API to construct a statement. In the example above, it uses the multipurpose regular expression pattern, "**(.+?)**", to extract the value "**!p33y2cxbq**" from the last response content. The backslashes in the statement are added in order to comply with the Java syntax.

Another example for Regular Expression variable below uses a more granular regular expression, which narrows down the match. This is required when a search finds multiple occurrences in the source content and further screening is required. In the example, there are two occurrences of the string "**_afrLoop=**" in the source, and using the standard pattern "**(.+?)**" may extract the wrong value. By specifying the regular expression pattern "**(\d{10,})**", which extracts a numeric value that is 10 or more digits from the source, the variable retrieves the correct value from the source content. Note that a backslash and a pound (or a number) sign are added as the Java syntax requires it.

TABLE 6. EXAMPLE OF SOLVE (REGULAR EXPRESSION) VARIABLE STATEMENT AND TARGET SOURCE 2

| | |
|--------------------|--|
| Variable Statement | http.solve("_afrLoop_4", "_afrLoop=(\\d#{10,#})", "", true, Source.Html, 0, EncodeOptions.None); |
| Response Content | query = query.replace(/_afrLoop=[^&]*/,"_afrLoop=57836461323353"); |

JavaScript Variable

JavaScript variable is another type of correlation variable that is generated by the auto-correlation process. JavaScript variable statements start with “**http.javaScriptPath**”, and search the value using 0-based index from either JavaScript or VBscript blocks within the source content. In the example below, the variable looks for a string that is the 32nd script block (which is defined as <script> in the source), and 4th literal (which is wrapped by single quotes) within that block.

TABLE 7. EXAMPLE OF JAVASCRIPT VARIABLE STATEMENT AND TARGET SOURCE

| | |
|--------------------|--|
| Variable Statement | http.javaScriptPath("web.jscrip.httprws3260483usora_2", 1, 31, 3, 0) |
| Response Content | </script></td></tr><tr id="region132" align="left"><td id="region139"></td><td id="region133"><button id="SubmitButton" title="Login" class="x7z" onclick="submitForm('DefaultFormName',1,{'_FORM_SUBMIT_BUTTON':'SubmitButtont-prNJS2'});return false" type="submit"> |

Unlike the XPath or Regular expression variables, which can tolerate content changes to some extent, JavaScript variables are vulnerable to content changes due to the search logic. Because the index is used to identify the value of the source content, in case the application returns more or less script blocks in the page content compared to that of recorded, the value won't be found and the playback fails. Therefore, when an application under test (AUT) is a web application that returns dynamic content, JavaScript variables that were auto-generated during recording may need to be manually removed or replaced by Regular Expression variables in order to have the script successfully playback.

Tip 5: Notice that script editing can be done better in Tree view rather than Java Code view in most cases

OpenScript provides users a choice of viewing script content in two different interfaces, **Tree View** and **Java Code view**. Tree view shows navigation nodes in graphical user interface, and Java Code view shows the command in pure Java syntax. Users can select which interface they use to view, develop and edit scripts depending on their technical preference.

Two views can be easily flipped back and forth by clicking the tabs. When Java Code tab is clicked, Java Code view shows the corresponding lines of code that were selected in the Tree view. The below figures show the screenshots of the two views.

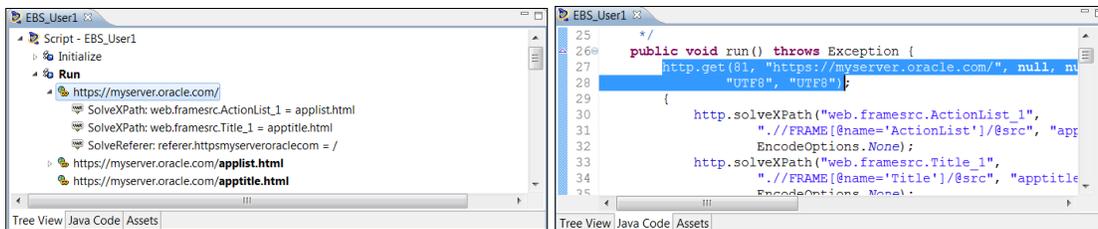


Figure 1. Users can edit scripts either in the Tree view (left) or the Java Code view (right).

How much Java skill a user needs to have in order to create a robust load test script in OpenScript is one of the frequently asked questions. The answer is that the load testing users may need entry level programming skills as the scripting language underneath is pure Java; however, in many cases the load script editing can be done much easier and more efficient in the Tree view. OpenScript provides a powerful graphical user interface to analyze, debug, and correct the various load scripting problems. Because load script editing can be done in Tree view more accurately and intuitively than in Java Code view in most cases, users may not need to see the Java code view at all during the load test script editing.

Advanced Java knowledge may be required rather when developing functional test scripts, which may be intended to be reusable over versions of application releases and often requires heavy code based customizations. The required skills for the load test users to create a robust load script are rather the knowledge of HTTP basics, correlation principles, regular expressions, and generic problem solving skills as the script troubleshooting often requires a trial and error approach that requires a lot of patience.

Tip 6: Take advantage of the intuitive GUI wizard to add custom correlations

Manual correlations can sound frightening until one has seen OpenScript's built-in **Substitute Variable wizard** feature. This feature is designed to fit the requirement of all levels of technical users, in order to add correlation variables to existing scripts intuitively using the graphical user interface (GUI).

How to use the wizard, as well as the other correlation techniques are explained comprehensively in ATS training materials, which is available in Oracle Learning Library.⁵ Please also note that the training videos and hands-on labs show examples or instruct users to work with the sample application that comes with WebLogic. Because of this, users may think the trainings are not relevant to test their applications, which are more complex than the sample application used in the training materials. However, the techniques discussed in the trainings are relevant to the core mechanism of OpenScript correlation concepts, and can be applied to any real life business applications. It is important to understand the basic concepts discussed in the trainings, in order to resolve the correlation problems found in more complex applications.

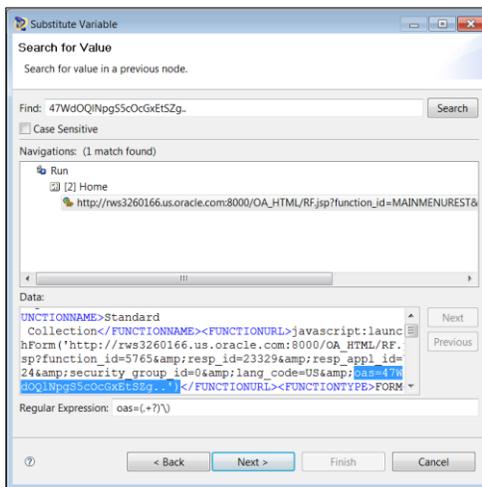


Figure 2. Substitute Variable wizard walks users through the correlation process step by step in the intuitive graphical user interface.

⁵ Oracle Learning Library (OLL), Oracle Application Testing Suite 12.x Video Series:, https://apex.oracle.com/pls/apex/f?p=44785:24:0::NO::P24_CONTENT_ID,P24_PREV_PAGE:6587,1

Tip 7: Understand why OpenScript saves two sets of navigation information; Recorded and Playback

In addition to viewing and editing scripts, OpenScript's graphical user interface allows users to see where in the source the variables are trying to extract the values from.

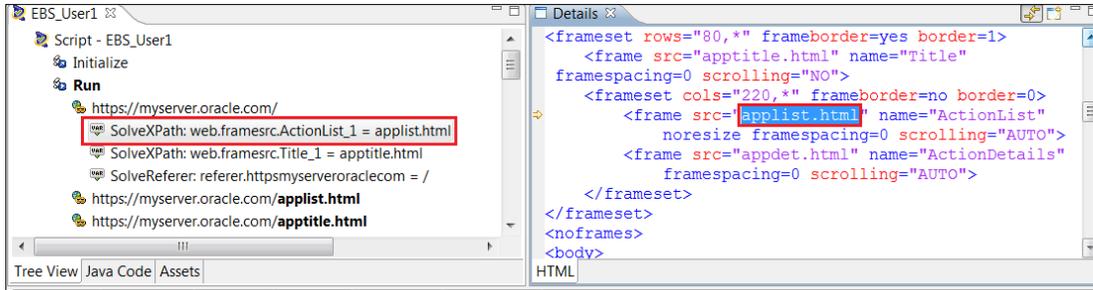


Figure 3. When a node in the Tree view is selected, recorded information for the selected navigation is displayed in the Details view.

Figure 3 shows the script commands in OpenScript's Script view interface. A variable "**web.framesrc.ActionList_1**" node is selected in the Tree view (left hand side), and the value "**applist.html**" is highlighted in the **Details view** (right hand side), which displays the original source content.

The Details view is in fact a multifunctional pane. It lets users view not only the recorded data, but also the information from playback, depending on where in the user interface the navigation node is selected. When a node in the Tree view is selected, the Details view shows the recorded information of the selected navigation. In addition, when a playback node is selected in the **Result view** (bottom side), the Details view shows the page information that was returned by the application at the playback time. As for the seventh tip, keep in mind that OpenScript saves two sets of data, recorded and playback. This allows users to compare the page content between the recorded and playback by selecting the **Comparison** tab in the Details view.

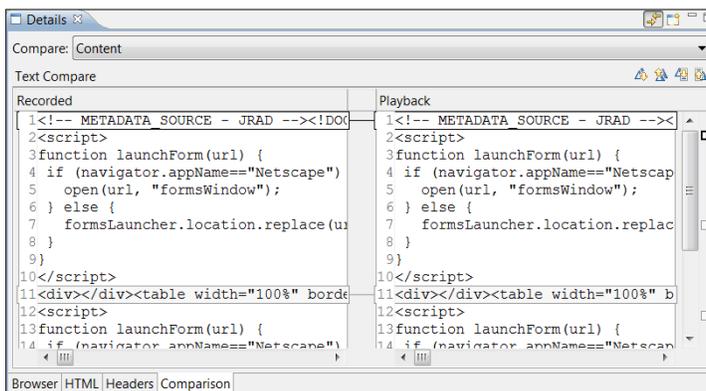


Figure 4. Comparison tab allows users to compare the navigation information, such as request and response headers, page content, and cookie strings between the recorded and playback.

Tip 8: Perform Compute Similarity over playback results to unveil hidden failures

“The application under test returned an incorrect page during the script playback, but why OpenScript still reports a passed result?” That is the question asked by users from time to time.

OpenScript load module defines playback status based on the results of the HTTP request command executions. The playback status for a command will be “Passed” as long as the application returns a page with a valid HTTP code (e.g. HTTP 200), and provides content that allow the script to create the variables defined for that page. In other words, the playback status can be “Passed” even when the application fails to execute the transaction as long as it meets the above success criteria. That is why it is recommended to run **Compute Similarity** feature over the “Passed” playback results to double check the page content validity.

Compute Similarity is an OpenScript feature that allows users to see differences between recorded and playback content and displays rates in percentage (%) for each navigation in the Result view. To apply this feature to a playback result, select the root node of the playback result in the Results view, then click **“Compute Similarity”** button in the tool bar. The button can be found in the upper right side of the Results view. Expand the result tree to have navigations displayed in the Result view, as the similarity rate shows only next to the navigations.

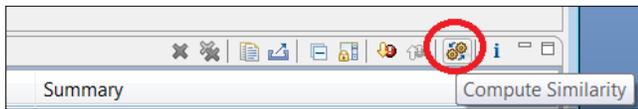
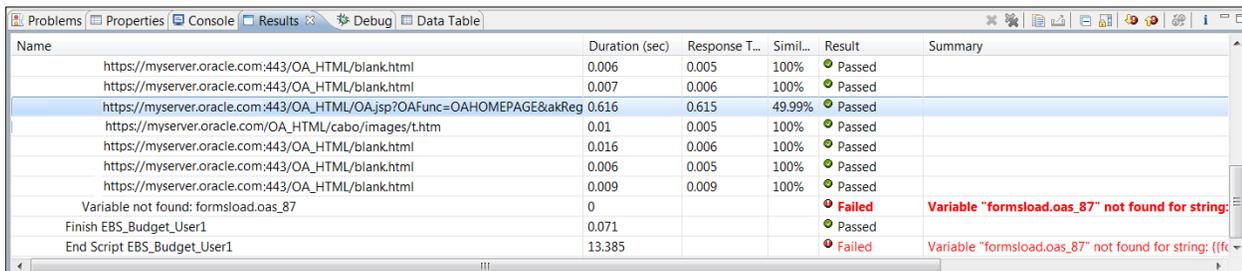


Figure 5. Compute Similarity button can be found on the upper right side of the tool bar of the Results view.

Even when a script playback is completed with a passed state, it is always suggested to run the Compute Similarity feature over the playback result to uncover the hidden failures. When the similarity rate is less than 90%, that is generally the recommended point to double check the content whether the playback is truly successful or not.

Compute Similarity isn't a tool that can only be used against the “Passed” playback results. In fact, it is the first step to troubleshoot failed playback results, such as “Failed to solve variable” failures, as it narrows down to which navigation the script was actually failing. In the example below, Compute Similarity gave a low content similarity rate, 49.99%, to one of the “Passed” navigation, where the cause of the playback failure may be found.

A screenshot of the OpenScript Results view showing a table of playback results. The table has columns for Name, Duration (sec), Response T..., Simil..., Result, and Summary. One row is highlighted in blue, showing a similarity rate of 49.99%. Below the table, there are several failed entries with error messages.

| Name | Duration (sec) | Response T... | Simil... | Result | Summary |
|--|----------------|---------------|----------|--------|--|
| https://myserver.oracle.com:443/OA_HTML/blank.html | 0.006 | 0.005 | 100% | Passed | |
| https://myserver.oracle.com:443/OA_HTML/blank.html | 0.007 | 0.006 | 100% | Passed | |
| https://myserver.oracle.com:443/OA_HTML/OA.jsp?OAFunc=OAHOMEPAGE&akReg | 0.616 | 0.615 | 49.99% | Passed | |
| https://myserver.oracle.com/OA_HTML/cabo/images/t.htm | 0.01 | 0.005 | 100% | Passed | |
| https://myserver.oracle.com:443/OA_HTML/blank.html | 0.016 | 0.006 | 100% | Passed | |
| https://myserver.oracle.com:443/OA_HTML/blank.html | 0.006 | 0.005 | 100% | Passed | |
| https://myserver.oracle.com:443/OA_HTML/blank.html | 0.009 | 0.009 | 100% | Passed | |
| Variable not found: formsload.oas_87 | 0 | | | Failed | Variable "formsload.oas_87" not found for string: |
| Finish EBS_Budget_User1 | 0.071 | | | Passed | |
| End Script EBS_Budget_User1 | 13.385 | | | Failed | Variable "formsload.oas_87" not found for string: {{fk |

Figure 6. Although the navigation has a “Passed” status, the Content Similarity rate is 49.99%, which raises a warning to the user and suggests content verification.

To recap, Compute Similarity is not only a utility to detect hidden failures from the “Passed” playback results, but also the first step to perform troubleshooting on load testing playback failures. Either passed or failed, it is always recommended to run this feature over the playback results to ensure script competency. As a key troubleshooting component, this feature will be repeatedly discussed in this paper in the later tips.

Tip 9: Make creative use of wildcard and regular expressions extracting dynamic values from source contents

Earlier in this paper we discussed the reasons for why “Failed to solve” problems occur in OpenScript playback. We also talked about the basics of the correlation concept, and built-in debugging features OpenScript has. This tip goes one step beyond and walks you through from concept to solution in a real world use case scenario.

Analyze the problem: What caused the “Failed to solve” playback error?

In this scenario, assume a user recorded a load testing script against a workflow in Fusion Applications. However, the script playback failed with a message “Failed to solve variable”.

Playback Failure Message

```
Failed to solve variable adf_comp_atkfr10r1j_id_111r10pt1tt10ci4 using xpath
//AdfRichTreeTable[@fullId=atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1']//AdfRichCommandImageLink[@behaviors="new
AdfShowPopupBehavior(&apos;:::rankpopup&apos;;,null,&apos;:ci4&apos;;&apos;:click&apos;:)" and @partialSubmit='true' and
@accessKey='\uffff' and @icon=/customer/B.png']/@fullId
```

The message states that a variable could not extract the value of the attribute “fullId” from the “ADFRichCommandImageLink” object with the given XPath syntax.

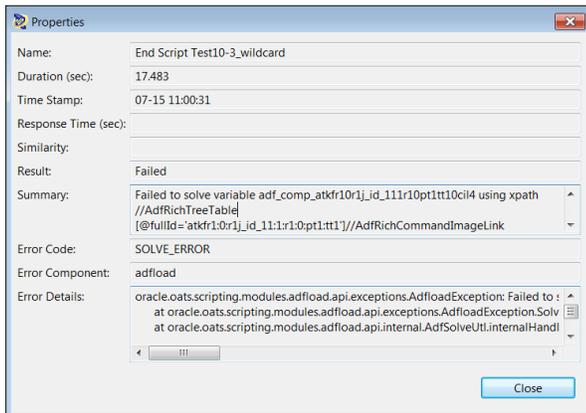


Figure 7. Playback failure reported a Failed to solve variable error message in its properties dialog.

The first thing to do always when troubleshooting a failed load testing script playback is to run **Compute Similarity** feature over the playback result. This will provide users the big picture on whether the playback failed because wrong page content was returned by the server, or a correct page was returned but a variable set in that page could not extract the target value from the page content. As a high level guideline, when the Content Similarity rate of a failed navigation is more than 90%⁶, there is a good chance that it is a valid page and variable definition simply needs more reinforcement. On the other hand, if the similarity rate is lower than 90%, it could be that an invalid page is returned by the application and further verification is recommended. Note when a percentage is significantly smaller, the variable that appears in the error message may be unrelated to the root cause of the failure.⁷

⁶ Similarity rate shows the similarity of the page in percentage. So the higher the %, the more similar the pages are between record and playback. But when small page content or binary content is compared, a small percentage may be shown even for a valid page. For example, when comparing a page with a single line content "Welcome Tom", if during playback the page has "Welcome Jennifer", then the content similarity will be 50%.

⁷ This point is discussed in Tip 2 in this paper.

In this scenario, the script playback failed with the **“Failed to solve variable”** error, however the similarity rate of the failed navigation is 99%,

Troubleshooting steps:

- » Select the root node of the playback result. Click on **“Compute Similarity”** icon from the toolbar to display the content similarity rate between the recorded and playback content.⁸
- » Expand the tree nodes in the Results view. Scroll down to find the navigation node where the failure is reported.
- » Verify the content similarity rate of the navigation in question. In this scenario, the rate is 99%. Therefore, it can be assumed that the correct page was returned, but a variable set in that page failed to extract a target value from the content.
- » Right mouse click on the error message in the summary column, and select **“Find Failure in Tree”**. The corresponding node in the Tree view is highlighted.

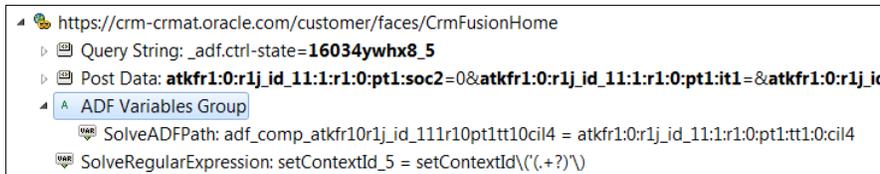


Figure 8. A variable node in the Tree view is highlighted.

- » Double click on the variable node to open the properties dialog.
- » In this example, the variable type is ADF XPath, and the variable name is **“adf_comp_atkfr10r1j_id_111r10pt1tt10cil4”**.

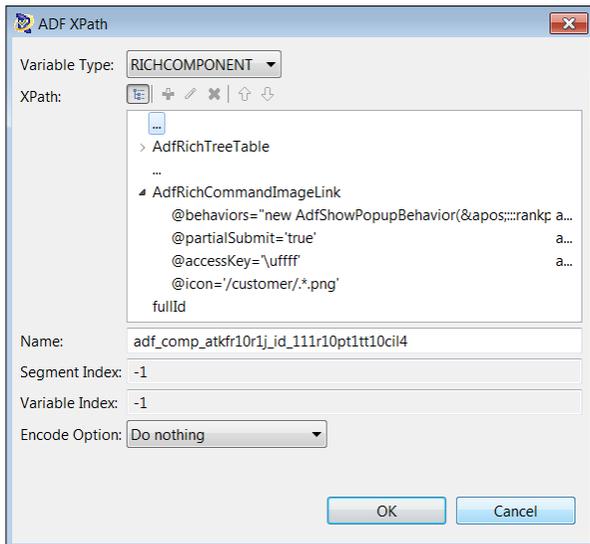


Figure 9. Variable's properties dialog has information on how the variable extracts a target string from the source content.

⁸ Compute Similarity feature is explained in Tip 8 in this paper.

- » In the XPath list box in the dialog, confirm there is an XPath definition of the string extraction.

XPath definition of the variable “adf_comp_atkfr10r1j_id_111r10pt1tt10cil4”

```
//AdfRichTreeTable[@fullId='atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1']
//AdfRichCommandImageLink[
@behaviors="new AdfShowPopupBehavior(&apos;:::rankpopup&apos;,null,&apos;:cil4&apos;,&apos;:click&apos;)" and
@partialSubmit='true' and
@accessKey='\uffff' and
@icon='/customer/B.png']
/@fullId
```

- » Close the dialog, and select the variable node in the Tree view once again.
- » The value “atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1:0:cil4” is highlighted in the Details view. This is the value, which the variable attempts to extract from the page content. Copy that text to the clipboard.

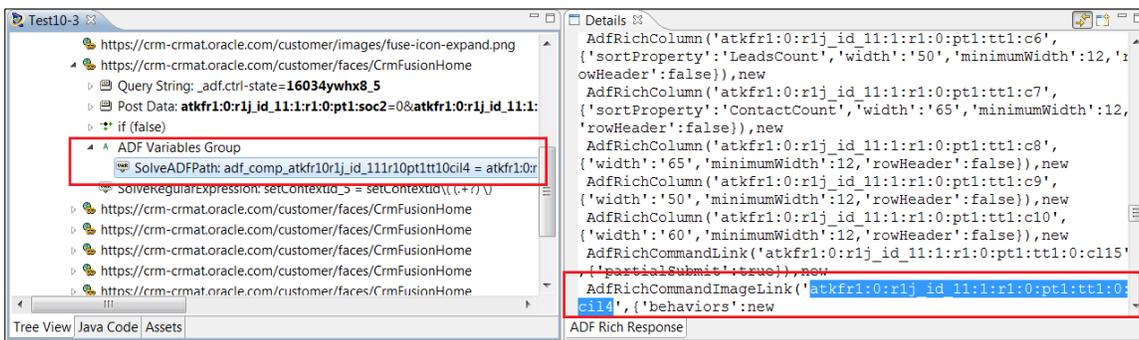


Figure 10. By selecting a variable node in the Tree view, the target value, which the variable extracts from the source content, is highlighted in the Details view.

- » Go back to the Results view, select the failed navigation node.
- » In the Details view, click **Comparison tab**.
- » Select “**Content (formatted)**” from the pull-down menu. Recorded and Playback contents are displayed.
- » Click anywhere in the Recorded pane (left side of the pane), and press **Ctrl-F** to open Find/Replace
- » Enter the target value to the Find textbox field, (in this example, “atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1:0:cil4”) and click **Find**.
- » The text is highlighted in the recorded content⁹
- » Note the corresponding source is also displayed in the Playback pane (right side of the screen)
- » See Figure 11 for a screenshot of the Comparison Tab.

⁹ The Find button may need to be pressed more than once to locate the right object, in case the search text is found in more than one location.

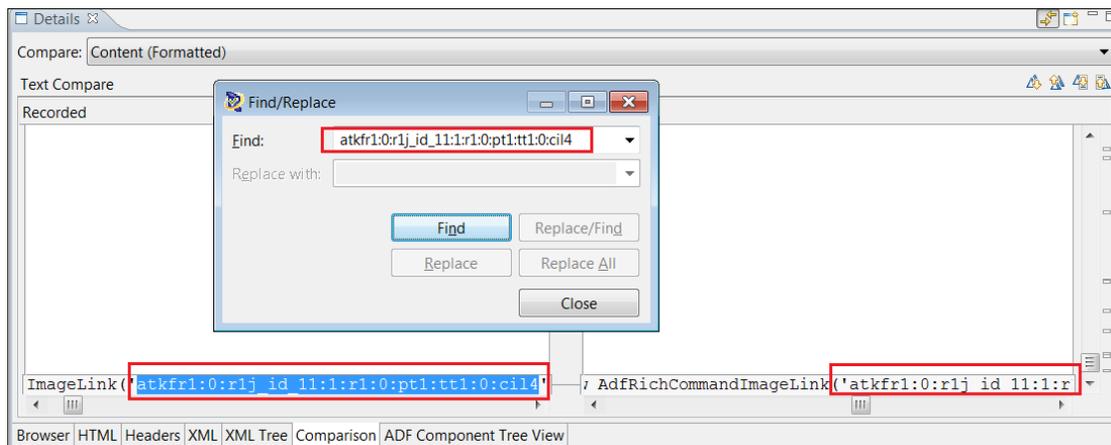


Figure 11. Locate the target text in the Comparison tab to identify the difference between the recorded and playback contents.

- » For an easier comparison, copy the object definition from the Recorded source and save to a notepad. Repeat the same for the Playback.^{10,11,12}
- » Compare the source contents of the target object between Recorded and Playback.
- » A couple of differences are seen in the object attribute values. In the example shown below in table 8, the recorded value “ci4” was changed to “ci5”, and “B.png” was changed to “C.png” during playback.

TABLE 8. OBJECT DEFINITION COMPARISON BETWEEN RECORDED AND PLAYBACK

| Target object definition in the recorded source | Target object definition in the playback source |
|---|---|
| <pre>new AdfRichCommandImageLink('atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1:0:ci4', {'behaviors':new AdfShowPopupBehavior(':::rankpopup',null,'ci4','click'), 'partialSubmit':true,'accessKey':'\uffff','icon':'/customer/B.png'}),</pre> | <pre>new AdfRichCommandImageLink('atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1:0:ci5', {'behaviors':new AdfShowPopupBehavior(':::rankpopup',null,'ci5','click'), 'partialSubmit':true,'accessKey':'\uffff','icon':'/customer/C.png'}),</pre> |

- » Go back to the Script view and double click the variable node to open the properties dialog.
- » Validate the variable's XPath definition once again.

XPath definition of the variable “adf_comp_atkfr10r1j_id_111r10pt1tt10ci4”

```
//AdfRichTreeTable[@fullId='atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1']
//AdfRichCommandImageLink[
@behaviors='new AdfShowPopupBehavior(&apos;:::rankpopup&apos;,null,&apos;ci4&apos;,&apos;click&apos;)' and
@partialSubmit='true' and
@accessKey='\uffff' and
@icon='/customer/B.png']
/@fullId
```

¹⁰ Note: copy only the relevant string which defines the object source, not the entire content.

¹¹ Alternatively, object definition can be copied from the HTML tab in the Details view.

¹² See Table 8 for sample object definitions

- » Confirm the XPath includes the “**ci14**” and also “**B.png**” in its object identification syntax, and each conditional is connected with “and” operator. This is a problem because with this logic, the given XPath will only identify the object when all of the conditions are satisfied. The XPath includes dynamic attribute values which do not exist in the playback content. As a result, due to the value change, OpenScript will not be able to identify the object during playback. This is likely the problem which caused the playback failure.

Resolve the problem: Add Wildcard to XPath variable

Having understood the root cause of the problem, the next step is to fix the problem. There are two ways to resolve the issue. One method is to apply a wildcard to the variables’ XPath definitions, so that OpenScript can identify the target objects even when they come back with different attribute values during playback.

- » Double click the variable node in the Tree view to open the variable’s properties dialog.
- » Double click the attributes that include the dynamic values. In this example, they are “**behaviors**” and “**icon**”.
- » Replace the dynamic values, which are included in the XPath, with a wildcard character. In this example, the asterisk (*) wildcard character was used to modify the values of “**ci14**” to “**ci1***” and “**B.png**” to “***.png**”.

```
@behaviors="new AdfShowPopupBehavior(&apos;:::rankpopup&apos;;null,&apos;ci1*&apos;;&apos;click&apos;)"
@icon="/customer/*.png"
```

- » With this change, the variable will identify the object from the playback source content, and extract the target string.

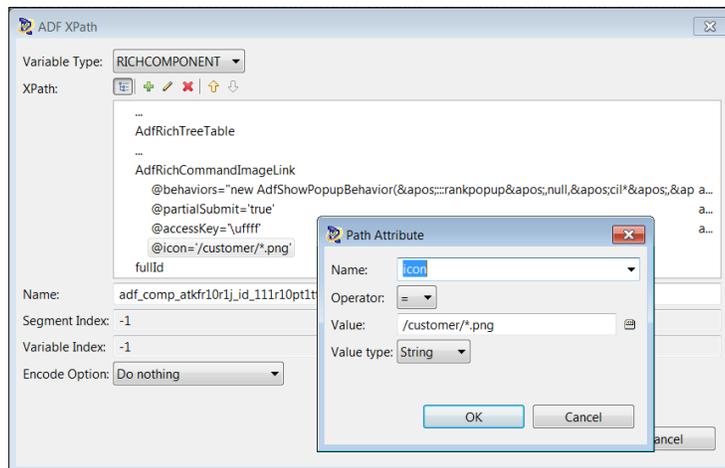


Figure 12. Replace dynamic attribute values with a wildcard. In the above example, the number “4” was replaced with “*” (asterisk).

Alternative Solution: Replace with Regular Expression variable

Replacing the dynamic text with a wildcard character is an effective method for relatively simple change patterns, such as replacing one digit numeric value or a single character. However, in case of dynamic texts with more complex change patterns, SolveXPath may not be the best variable to extract the string. An alternative way of fixing this problem is to create a new regular expression variable, using the Substitute Variable wizard, to extract the target string using a regular expression instead of XPath.

- » Right mouse click to “Skip” to the existing ADF XPath variable¹³
- » Select the HTTP navigation node where the ADF XPath variable is defined
- » Select **HTML tab** in the Details view
- » Press Ctl-F to search for the text which needs to be extracted. In this example, the text to search for is “atkfr1:0:r1j_id_11:1:r1:0:pt1:tt1:0:cil4”.
- » Select the highlighted text, right mouse click and select “Create variable”.
- » Follow the wizard to create a new regular expression variable and use the same name as the XPath variable.¹⁴
- » Once a new variable is created, go to the Tree view, and double click to open the properties of the variable.¹⁵
- » Modify the regular expression pattern as necessary. In this example, dynamic values are replaced with **\d** (replaces a single digit number) and **\w** (replaces a single letter).

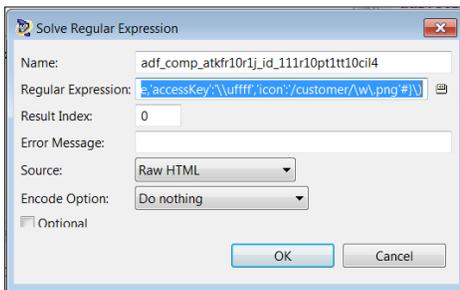


Figure 13. Properties dialog of a Regular Expression variable. The pattern that extracts the target string is specified in the 2nd field.

TABLE 9. ORIGINAL AND MODIFIED REGULAR EXPRESSION PATTERNS

| Original Pattern | Modified Pattern |
|---|--|
| AdfRichCommandImageLink\('(.*?)',\{'behaviors':new AdfShowPopupBehavior\(':::rankpopup',null,'cil4','click'),'partialSubmit':true,'accessKey':\\u{ffff},'icon':/customer/B\\. | AdfRichCommandImageLink\('(.*?)',\#{'behaviors':new AdfShowPopupBehavior\(':::rankpopup',null,'cil\d','click'),'partialSubmit':true,'accessKey':\\u{ffff},'icon':/customer/\w.png' |

OpenScript’s built-in **Substitute variable** wizard applies a regular expression pattern, “(.*?)”, which captures any characters with any length to extract a string from the source. This is sufficient in many cases, but when additional filtering is required, more advanced regular expression patterns can be creatively used to extract target strings from the source content.

For example, “d” matches a single digit, and “\d+” matches one or more digits. If the value to extract is always numeric but the length is unpredictable, then “(\d+)” can be used to capture the value more accurately from the source. If the pattern string includes another dynamic value within, that value can be replaced by the patterns too.

TABLE 10. CUSTOMIZED REGULAR EXPRESSION PATTERNS

| Source content | Regular Expression Patterns | Explanation |
|--|--|---|
| "question:0:choice" + value="100"> | "question:0:choice" \+ value="(.*?)"> | Captures any characters for “value” |
| "question:0128:choice" + value="11281988"> | "question:\d+:choice" \+ value="(\d+) "> | Captures one or more digits for “value” |

¹³ Select the variable node in the tree view and select Skip. Alternatively the variable can be deleted, but deleting variable also removes the string substitutions in the subsequent navigations.

¹⁴ Please see ATS trainings available in the Oracle Learning Library for instructions to use the Substitute Variable Wizard.

¹⁵ The same name as the original solveXPath variable was applied to the regular expression variable in order to keep the existing string substitutions effective.

Tip 10: Beware of client side cookies dynamically set by JavaScript

The previous section discussed the analysis and solution for a script playback problem, “Failed to solve variable”, when a valid content was returned by the server however an enhancement in the variable definition was required. Another real world use case scenario we will focus on in this section is when the server returns an invalid page content.

| Name | Duration (sec) | Time Stamp | Respon... | Similarity | Result | Summary |
|--|----------------|----------------|-----------|------------|--------|------------------------------------|
| [2] Welcome | 3.221 | 02-26 08:19:25 | | | Passed | |
| https://fuseapps.pgahq.com:10614/homePage/ | 2.128 | 02-26 08:19:25 | 2.123 | 99.9% | Passed | |
| https://fuseapps.pgahq.com:10614/homePage/ | 0.068 | 02-26 08:19:27 | 0.068 | 100% | Passed | |
| https://fuseapps.pgahq.com:10614/homePage/ | 0.451 | 02-26 08:19:27 | 0.45 | 2.61% | Passed | |
| Finish FusionPRC | 0.029 | 02-26 08:19:28 | | | Passed | |
| End Script FusionPRC | 16.881 | 02-26 08:19:11 | | | Failed | Failed in finding component wit... |

Figure 14. Failed navigation has 2.61% of content similarity rate. It is likely that the wrong page content was returned by the server.

Remember, the first thing to do when script playback has a failure is to run Compute Similarity over the failed playback result. Then scroll down to the end of playback result in the Results view. If there is a navigation with an obviously low similarity compared to others, it is likely that is the failed navigation. Note the navigation may still indicate “Passed”, but this only means there was no failed variable and no error code returned from the server for the request that OpenScript made. In other words, if there is no variable defined for that page, and if the server returned 200 OK for the response code, then the navigation will “Pass” even when an error page is returned.

Compare Request Headers

Comparison tab gives users information on differences in headers, contents and cookies between the recorded and playback. The standard troubleshooting process is to start the examination by comparing the request headers to detect any missing correlations. In many cases the problem and solutions reside in the request headers.

- » Right mouse click on the failed navigation, and select “**Find in Tree**”. The corresponding node in the Tree view (left upper pane) is highlighted.
- » Expand the node and verify the Post data parameters. Inspect if there are parameters that are obviously missing correlations.
- » Next, go back to the Results view, select the failed navigation, and select Comparison Tab from the Details view.
- » Select **Request Headers** from the pull-down menu. The Recorded and Playback headers are displayed.
- » Carefully examine the headers and find any missing or incorrect correlations.

| Recorded | Playback |
|--------------------------------|----------------------------|
| POST /homePage/faces/AtkHomePa | POST /homePage/faces/AtkHo |
| Accept-Encoding: gzip, deflat | Accept-Encoding: gzip |
| Accept-Language: en-US,en;q=0 | Accept-Language: en-US,en; |
| Accept: text/html,application | Accept: text/html, image/g |
| Adf-Rich-Message: true | Adf-Rich-Message: true |
| Cache-Control: no-cache | Cache-Control: no-cache |
| Connection: Keep-Alive | Connection: Keep-Alive |
| Content-Length: 1687 | Content-Length: 1742 |
| Content-Type: application/x-w | Content-Type: application/ |
| Cookie: JSESSIONID=EVGDSZ5NNE | Cookie: JSESSIONID=jhINTTT |
| Host: fuseapps.pgahq.com:1061 | Host: fuseapps.pgahq.com:1 |
| Pragma: no-cache | Referer: https://fuseapps. |
| Referer: https://fuseapps.pgal | User-Agent: Mozilla/5.0 (W |

Figure 15. Comparison Tab (right hand pane) provides differences of request headers between recorded and playback.

- » From here it is a trial and error process. Repeat “modifying & verifying” header parameters until the script runs. In addition to the request headers, Cookie strings sent in the Playback should be checked also.

Compare Cookie Strings

There may be cases where inspecting request headers do not detect any suspicious factor that is causing the playback failure for a particular navigation. When this happens, another place to take a look at is the Cookies.

- » Select **Cookies** from the pull-down menu. The Recorded and Playback cookies are displayed.
- » If there are extra or less numbers of cookies seen in the Playback pane, they might be added or removed dynamically from the client during the recording. OpenScript automatically handles server side cookies and also has correlation rules that take care of the client side cookies as well¹⁶. However if for any reason the correlation rule missed to pick up the client side cookies, they have to be manually added or removed in the script.

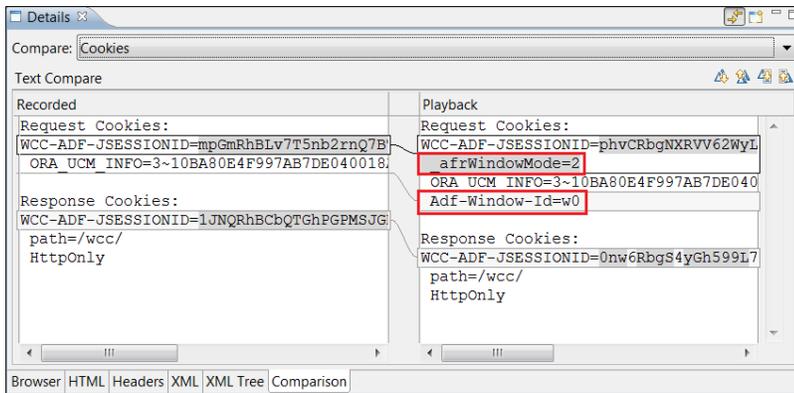


Figure 16. Playback has extra cookie parameters at the failed navigation. These cookie parameters do not exist in the Recorded pane, which explains the string was removed by client side JavaScript at the recording. They have to be removed from the script in order for playback to succeed.

In the above example, two cookie parameter-value pairs exist in the playback while it does not in the recorded. They are likely removed dynamically by the client during the recording; however OpenScript did not catch that change.

The solution is to add **http.removeCookie** commands before the navigation. If cookies have to be added back or new cookies need to be added in the subsequent navigations, use **http.addCookie** commands. Note the cookie values may need to be correlated. In the sample code shown below, cookie parameter “**_afrRedirect**” has a numeric value, which is likely to change dynamically over different sessions. This value should be correlated.

//Removing Cookies

```
http.removeCookie("_afrWindowMode");
```

```
http.removeCookie("Adf-Window-Id");
```

//Failed Navigation

```
http.post(23,"http://mytestserver.us.oracle.com:16225/wcc/faces/wcclogin?_adf.ctrl-state=3y0vhnxb0_3&Adf-Window-Id=w0",...
```

//Adding Cookies

```
http.addCookie("_afrRedirect=443465360090000; domain=mytestserver.us.oracle.com; path=/,");
```

```
http.addCookie("_afrLoop=; domain=mytestserver.us.oracle.com; path=/,");
```

¹⁶ A correlation rule named “Client Set Cookie” has been added to load correlation libraries since ATS version 12.3. To see the rule, go to OpenScript menu View > OpenScript Preferences > Correlation > HTTP > Expand the tree node “Web Default” In the list box and find “Client Set Cookie” This rule also exists in EBS, ADF and other load test modules.

Cookie strings can be correlated either from the Tree view, or Java code view. Below are the steps to correlate cookies using **Substitute Variable** wizard, which is a built-in feature of OpenScript.

- » In the Tree view, double click the Cookie node that includes the dynamic value. This opens a dialog.
- » In the “**Cookie String**” text box, highlight ONLY the value that requires a custom correlation.

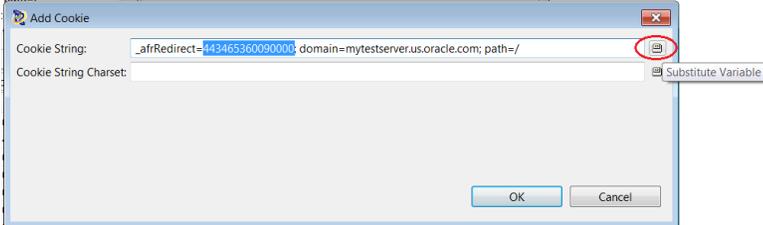


Figure 17. Select the value that is required to be correlated, and then click the Substitute Variable button. NOTE: If the Substitute Variable button was pressed without highlighting the correct portion of the cookie string, the Wizard will try to correlate the entire string shown in the text box.

- » Click “**Substitute Variable**” image button which is located next to the “Cookie String” text box.
- » Click “**Create New Variable**”, and follow the correlation wizard to create a new variable.

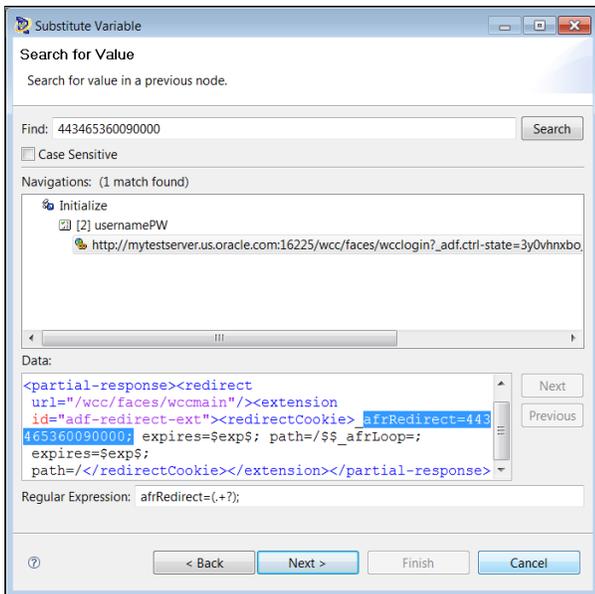


Figure 18. Substitute Variable wizard guides users step by step to add manual correlations.

Now the cookie string is correlated. Example code with a correlation is shown below. This is how the commands show up in the Java Code view. Note the value has curly braces, which means the value is substituted with a variable, “**myvar_afRedirect**”. The recorded value, 443465360090000, shows up as information purpose only and will not be used in the playback.

```
http.addCookie("_afRedirect={{myVar_afRedirect,443465360090000}}; domain=slcac572.us.oracle.com; path=", "");  
http.addCookie("_afrLoop=; domain=mytestserver.us.oracle.com; path=", "");
```

The screenshot below shows how the cookies appear in the Tree view.

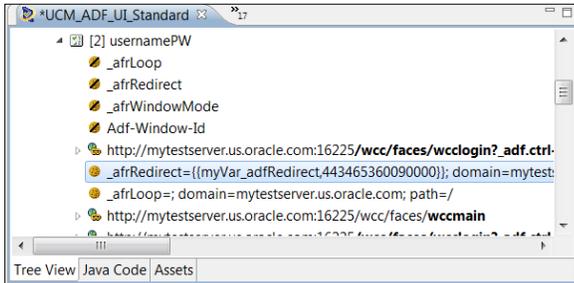


Figure 19. Cookie nodes in the Tree view. A custom correlation was added to the _afrRedirect cookie string.

Tip 11: Be prepared for emergency script recreation

Fruitfully created load test scripts are readable, robust, and re-creatable. The best approach for creating load test scripts is to keep the script creation process as simple as possible, so that the scripts can be quickly recreated when required. The common mistake is to duplicate the manual test flow which has a large number of test steps, into a single script and add excessive lines of custom code. This is rather a best practice for creating functional test scripts, which are designed to be reused over different releases of applications by applying code based customizations. Instead, load test scripts should be maintained as lean as possible, and testers should be prepared to quickly recreate a script in case of changes made to the application under test.

Ideally, testers should wait for the application to completely stabilize before starting the script creation process in order to minimize any further changes made to the application. This is the recommended approach so testers can move on to heavy customization without worrying about re-doing the scripting work. Unfortunately, the reality in many cases is that testers need to start scripting in ruthless circumstances before development is complete. To meet deadlines at the end of the product release cycle, new application drops can come down every morning by a nightly build process. New builds likely include changes in the HTML, headers, cookies and other factors that modify the application's HTTP behavior, and require re-creation of the load test scripts, while the testing effort has to be continued.

In such condition, in order to recreate the script as quickly as possible, the best practice is to keep the recording steps short, name the step groups meaningfully, add minimally required custom code into the script, export reusable logic into function libraries, and document all the steps for any customization that has to be added in a clear and organized manner.

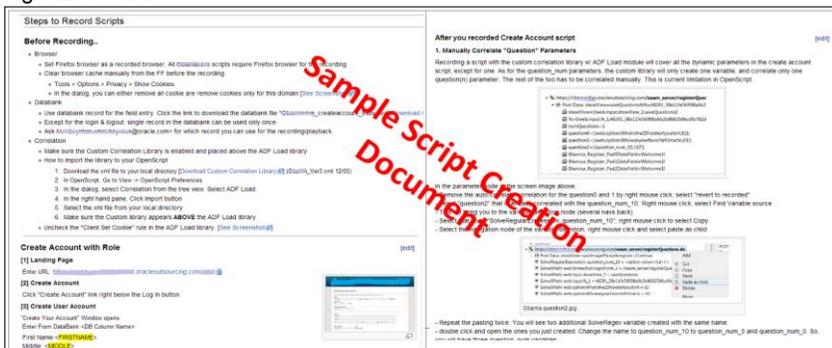


Figure 20. Document the script creation with clear steps and prepare for an emergency script recreation.

Tip 12: Learn why script scalability and reusability is always a trade off

Why do load test scripts require recreation upon application change, while that may not be required for the functional test scripts? The answer is because in order to enable high scalability, load scripts had to give up reusability to some extent in return.

By design, Oracle Load Testing removed the browser rendering process from its script. Because it does not need to carry a browser during script execution, it allows users to maximize the number of concurrent virtual users while minimizing the amount of test resources required for the load test. This approach, on the other hand, makes it harder to edit the script and may reduce the flexibility to run a script against different application configurations.

Figure 21 shows two scripts, functional and load, which recorded the same user transaction. The functional test script only recorded user's specific actions made on the browser, such as "Click Button". Therefore, after slight application change, the click command can still be executed as long as the button exists on the page. Even in the case where the target button object had a change in its attribute, a functional test script can easily be modified by updating its XPath in the click command. The load test script on the other hand, captures an entire snapshot of the HTTP traffic as a result of the user action. An application change may not bring too much difference at the GUI level, but there may be additional post data parameters in the request headers, or new cookies or dynamic session values introduced at the HTTP level. Unless users know exactly how these protocol level changes are made on the application side, these changes are usually not correctable in the existing script and re-recording is the only solution for a valid playback.

The load testing products in the market offer either a "thick" client approach or a "thin" client approach. A "thick" client approach enhances script reusability at the expense of reduced scalability, while a "thin" client approach increases scalability at the expense of reduced reusability. Scalability and reusability are often trade-offs in the load test scripting world.

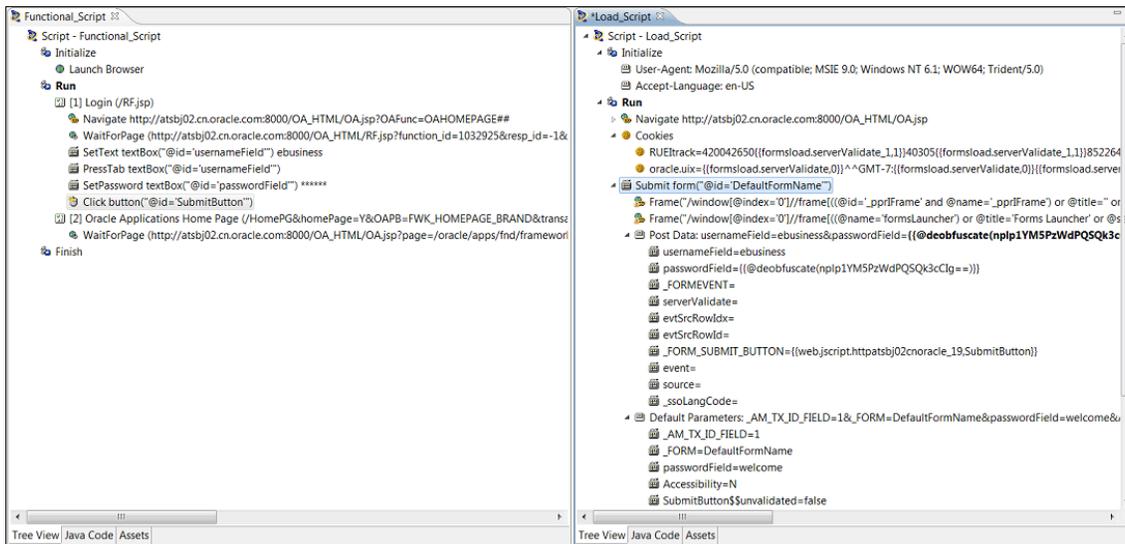


Figure 21. Functional Testing Script (left) and Load Testing script (right) that captured the same user transaction. The load script has a lot more commands recorded compared to the functional test script. Generally load test script has more constraints compared to the functional test script and can only run in the recorded environment, or on servers with identical configurations.

Tip 13: Create load test script with proper step group names

When recording a load test script, the same step group name may show up repeatedly even though different actions were recorded.

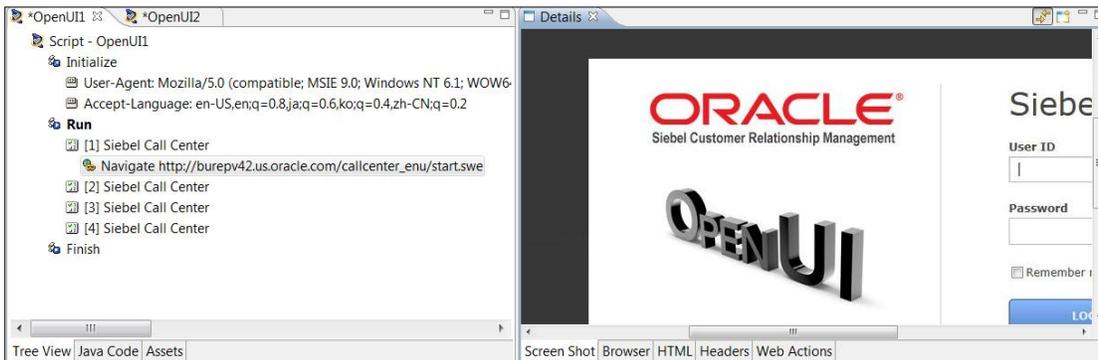


Figure 22. Step groups with the same name makes viewers hard to understand the flow. In this example, all step groups have the same name, Siebel Call Center, regardless of the transaction performed in each step.

With the default recording preferences, OpenScript automatically generates step group names from the page titles. However, this is not always a desirable way of constructing scripts as multiple pages can have the same page title and some actions may not result in a new page at all. A script recorded with multiple steps with the same name is not only difficult to comprehend and debug, but does not provide meaningful data to the session report when the script is used in an Oracle Load Testing scenario.

To create a script with meaningful step groups, it is recommended to disable automatic step group creation before recording the script. Then during recording, click “**Add step group**” button from the floating toolbar each time before a specific user transaction begins, and create a Step Group with a meaningful name. This way the user knows where exactly the transaction failed in case of playback failure, and can see response times for each step group easily in the session report.

- » Go to Preferences > Step Groups > HTTP.
- » Check “**Do not create steps**” and “**Do not name steps**”.
- » Repeat for other modules, like ADF or EBS, if the script is using them.
- » Click Record button.
- » Create and name a new step group from the floating toolbar before performing any action.
- » New navigations will be recorded under the new step group.
- » Repeat previous two steps when performing any new action on the application.



Figure 23. Click Add Step Group image button from the toolbar to manually create Step Groups with meaningful names.

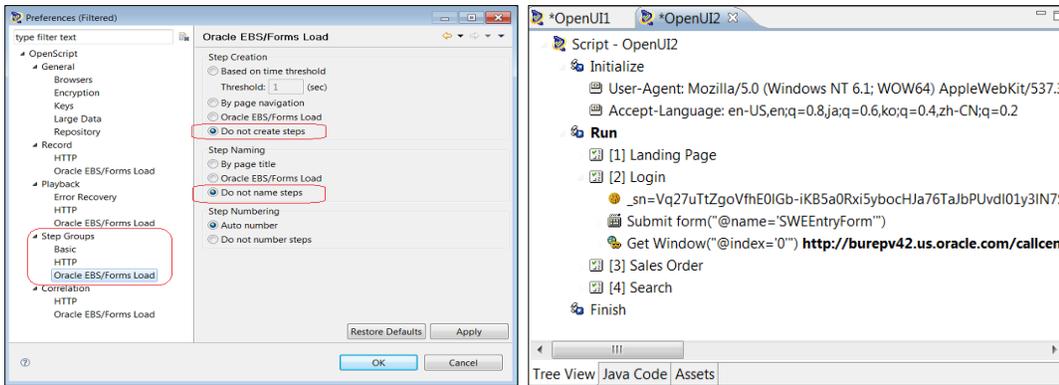


Figure 24. Turn off auto step creation and manually create step groups with meaningful names.

Tip 14: Request static resources concurrently just like browsers do

With the default settings, Oracle Load Testing scripts automatically filter the static resources, such as image files, CSS¹⁷, or js files, and suppress them from appearing in the script. These requests are stored within the script out of sight and executed concurrently at the playback just like browsers do, when **Download manager** is enabled in the playback preference. However, there are static resource requests that could not be filtered due to the current Download manager limitation. For example, resources requested from the CSS files are not filtered. These unfiltered static resource requests appear in the script as standard http navigation commands. This may result in an unexpectedly large transaction time as the real browsers run these requests in parallel, while OpenScript executes them in a sequential manner during playback.

OpenScript provides a built-in API to playback unfiltered static resource requests in parallel. By wrapping the static resource requests within a concurrent block, OpenScript playback will execute the resource requests in parallel with multiple connections open.¹⁸

```
beginConcurrent("MyTransaction1") ;

    //Place static resource requests

endConcurrent("MyTransaction1");
```

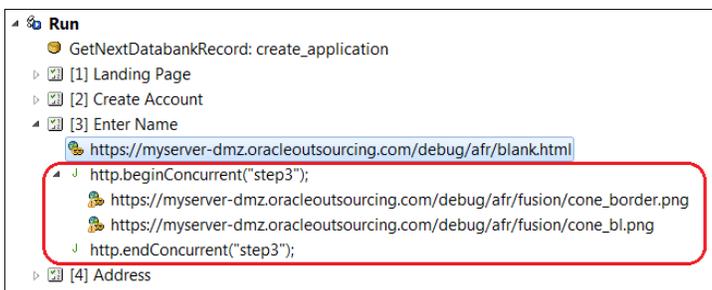


Figure 25. Wrap static resource requests with begin and end concurrent block to request them in parallel.

¹⁷ CSS: Cascading Style Sheets

¹⁸ This topic is discussed extensively in the Oracle white paper, OLT Deep Dive. Raja Vengala, 2014, p.2, section Generating Realistic Load using OpenScript Load Test Script

Tip 15: Use Web recording mode for EBS and HTTP mode for ADF application

OpenScript load testing scripts provide two different recording modes, Web and HTTP. Web recording mode, often called simply as **Web mode**, was introduced in OpenScript version 9.2 and is the default recording mode for Web/HTTP and E-Business Suite Load Testing modules. Web mode was designed especially for users who are not yet familiar with HTTP scripting, and provides usability benefits such as enhanced script readability and additional script automation, compared to the traditional HTTP recording mode.

The Web mode script code appears less verbose and more intuitive than scripts recorded in **HTTP mode**. HTTP mode instructs OpenScript to store all data (e.g. URL, request headers and parameters) required to construct an HTTP request within the script code, while Web mode stores only metadata that points to the related HTTP request information. The metadata represents user's browser actions such as "Submit Form" or "Click Link", which were performed during the recording.

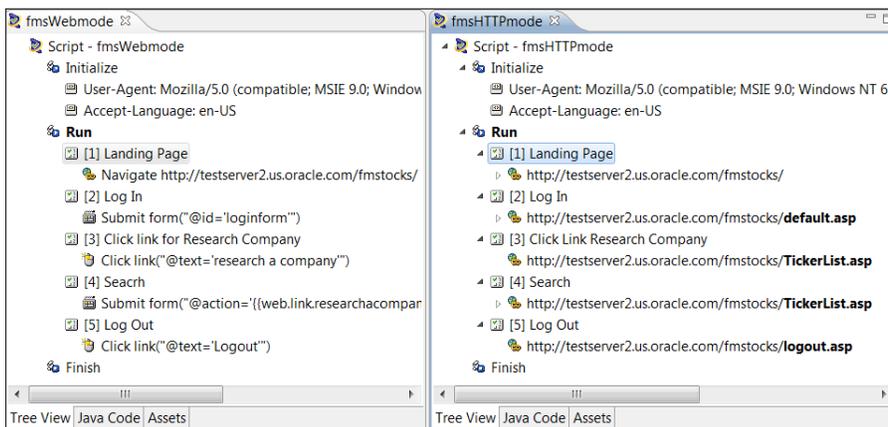


Figure 26. Two scripts recorded with the same test flow in Web mode (left) and HTTP mode (right). While the HTTP mode script has a set of URLs, Web mode script has user actions, such as "Submit Form" or "Click Link".

On playback, Web mode provides script automation that goes one step ahead of the HTTP mode. Unlike the HTTP mode script which requires all correlations pre-configured before the command execution, Web mode dynamically creates a request during command execution by parsing the page content from the previous response. It then identifies the form object which the metadata points against, and extracts URL and parameter-value pairs from the target Form object to construct the POST body. The parameter-value pairs found during this process will be a part of the Post data request without requiring user's interaction to correlate the values.

Only parameters not found in the form (or the values that were modified by the user when submitting the form) require traditional correlation in order to use runtime values dynamically during the script playback. Therefore, Web mode can reduce user's manual correlation effort drastically as parameters found in the form will not even show up in the script under the **Post Data** node, as they are taken care of by the script automatically in the background. This provides benefits, especially when scripting against Oracle E-Business suite applications, which can have hundreds of Post Data parameters in a single POST request.

In the script, parameter-value pairs that are found in the form will be placed under **Default Parameters** node, but they are displayed for information purpose only. Although the parameter-value pairs in the script appear as if they are hard-coded, there is no need to modify or correlate them as they will be handled by the script during command execution.

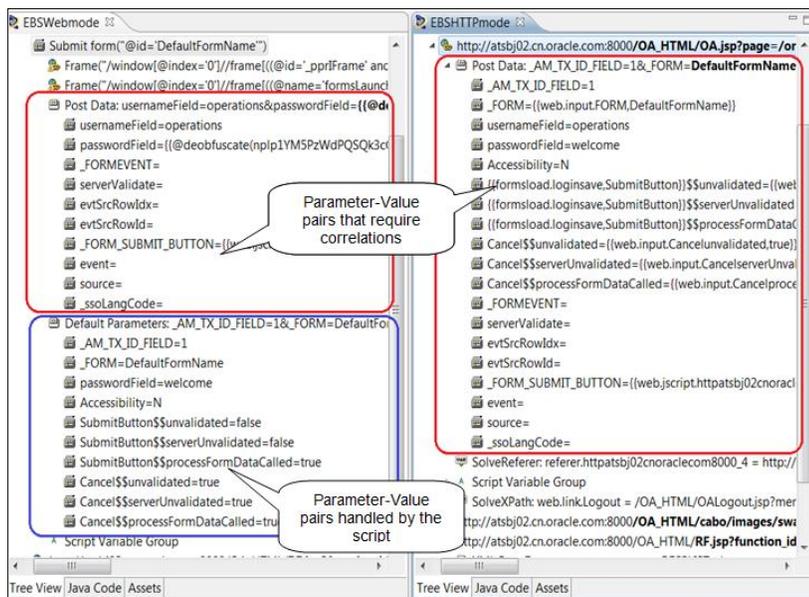


Figure 27. Same user transaction recorded by Web mode (left) and HTTP mode (right). The HTTP mode script has twice as many parameter-value pairs under Post Data tree node than in the Web mode script. Parameter-value pairs show recorded values in the Default Parameter node in the Web mode script, however runtime values will be used in the playback.

Web mode provides reduced correlations and increased script readability, on the other hand it consumes more system resource during playback than in HTTP mode, as it requires additional internal processing when sending the same HTTP request. While Web mode is recommended for applications with many Post data parameters, such as Oracle E-Business Suite, it is not recommended for content-rich web applications such as Oracle Fusion or ADF based applications. This is because parsing heavy web contents can reduce the agent performance during a load test.

TABLE 11: WEB MODE VS. HTTP MODE COMPARISON SUMMARY

| | Web Mode | HTTP Mode |
|-----------------------------|---|--|
| Request Processing | On playback, every action results in parsing of the previous response, building a DOM tree and searching for the target object using attributes (metadata) captured during recording. | On playback, a request will be sent immediately as all data required to create the HTTP request was captured as part of the recording. |
| Resource consumption | Requires more system resources compared to the equivalent HTTP mode script because additional background tasks (mentioned above) are involved in sending an HTTP request. | Requires less system resources compared to the equivalent Web mode script. |
| Script Readability | Script is readable even for users who are not familiar with load test scripting. | Script is readable only to users who are familiar and experienced with load test scripting. |
| Correlations | Script requires little or no correlation. | Script requires moderate to heavy correlation. |
| Good for | Applications having many or variable number of parameter-value pairs in its Post data requests, such as Oracle E-Business Suite Applications. | Any web applications, including ones with numerous tables or frames in the page contents, such as Oracle Fusion or ADF based applications. |

Tip 16: Capture web transactions from Chrome browser

OpenScript for load testing officially supports Internet Explorer and Firefox as recording browsers. When the record button is clicked, OpenScript enables a proxy and launches a browser. Once the user starts to perform web transactions in the browser, OpenScript uses the proxy recorder to capture HTTP requests and responses between the browser and the application. In fact, OpenScript recording is not tied to a specific browser that is launched by the tool. As it uses the proxy for recording, it captures the entire HTTP traffic to and from the system where OpenScript is installed. In this sense, OpenScript's recording mechanism is similar to that of a network debugging tool, such as Wireshark¹⁹ or Fiddler²⁰. What the proxy allows users to do is capture web transactions from any browser, even from the non-browser applications as long as the underlying protocol is HTTP, or HTTPS.

Steps to record HTTP/S traffic from a Google Chrome browser.

- » Create a Load Testing script.
- » Go to Preferences > Record > HTTP > General Tab > Set Recording Mode to HTTP.
- » Hit the Record button to start the recording.
- » Internet Explorer browser opens. Leave that browser open. Do not Close IE browser as that stops the recording.
- » Launch Google Chrome browser.
- » Enter application URL to record web transactions from and perform navigations within the application.
- » Confirm the navigations are recorded into the script.

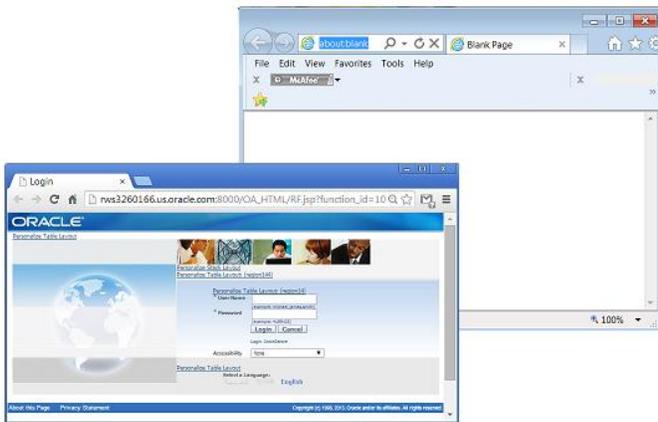


Figure 28. OpenScript can capture HTTP transactions from any browser, such as Google Chrome. To keep the proxy capturing enabled, do not close the IE browser opened by the recorder.

Before the recording starts, OpenScript may prompt a dialog to clear cache. Note this setting is effective only for the Internet Explorer browser. When using Chrome or Firefox as the recording browser, the browser cache has to be manually removed before starting the recording in order to create a clean script. Not clearing the cache can result in recording out-of-date cookies from the past browser sessions, which can cause the script playback to fail. If a cookie node is recorded at the top of the script, this indicates a cached cookie was recorded from a previous browser transaction.

¹⁹ Wireshark: <https://www.wireshark.org>

²⁰ Fiddler: <http://www.telerik.com/fiddler>

Tip 17: Construct REST requests with OpenScript's built-in HTTP API

Web Applications under test may use Representational State Transfer (REST) API to communicate between machines to create, read, update or delete data over the HTTP protocol. Each of these actions can be achieved by HTTP methods, POST, GET, PUT and DELETE respectively. OpenScript provides API support to manually construct GET and POST methods in Java code to test REST requests. Any other HTTP methods can be tested by using generic NAVIGATE API.

TABLE 12: HTTP GET AND POST API TO TEST REST REQUESTS

| GET | |
|--|---|
| <pre>http.get(null, "http://mytestserver.us.oracle.com/ws/", http.querystring(http.param("cmd", "login")), http.headers(http.header("<header-name>", "<header-value>", Header.HeaderAction.Add)), true, "UTF8", "UTF8");</pre> | <pre>// id. (May be null) // urlpath // query String (May be null) // header array // (Add,remove,modify headers) // encode, reqCharset, respCharset</pre> |
| POST | |
| <pre>http.post(null, , "http://mytestserver.us.oracle.com/ws/", http.querystring(http.param("cmd", "login"), http.param("lang", "ENG")), http.postdata(http.param("timezoneOffset", "-330"), http.param("userid", "user"), http.param("pwd", "password")), http.headers(http.header("<header-name>", "<header-value>", Header.HeaderAction.Add)), true, "UTF8", "UTF8");</pre> | <pre>// id. (May be null) // urlpath // query String. (May be null) // postdata // header array // (Add,remove,modify headers) // encode, reqCharset, respCharset</pre> |

To send a request using any other HTTP methods, such as PUT or DELETE, “**http.navigate**” should be used. This API also comes in handy to create POST requests that use JSON literals in the POST body. The following example shows how to construct a PUT request using http.navigate API.

TABLE 13: HTTP NAVIGATE API TO TEST PUT OR DELETE REQUESTS

| PUT, DELETE or any other methods | |
|--|--|
| <pre>Headers headers = new Headers(); Header header1 = new Header("Origin", "http://10.140.191.172:7001", HeaderAction.Add); Header header2 = new Header("X-Requested-With", "XMLHttpRequest", HeaderAction.Add); Header header3 = new Header("Content-Type", "application/json; charset=UTF-8", HeaderAction.Modify); headers.add(header1); headers.add(header2); headers.add(header3); String putBody = "[{"internalName": "MaxWBSTreeLevels", "value": "49"}]"; http.navigate(null, "http://mytestserver:7001/primavera/rest/settings/application", putBody.getBytes("UTF-8"), headers, true, "UTF-8", HTTPMethod.PUT);</pre> | <pre>// create variable to add header information // create variable to add postdata parameter information // id (May be null) // urlpath // postdata. (May be null) // additionalheaders // followRedirect, reqCharset // http method</pre> |

The sample code in table 13 above can be used as a template to create a request in OpenScript script, for PUT and DELETE methods or POST method that includes JSON literals in the request body. For example, to test a RESTful POST request with the following navigation requirements, create a new load testing script in OpenScript then open Java Code view to copy the sample code from Table 13 into the Run section.

TABLE 14: SAMPLE REQUIREMENTS CONSTRUCTING A RESTFUL POST REQUEST

| PARAMETERS | VALUES |
|-------------------|---|
| URL | https://mytestserver.oracle.com/myservices |
| HTTP Method | POST |
| Request Headers | Content-Type = application/json |
| JSON Request Body | { "userid": "testuser", "projectId": "12345", "dataFilters": { }, "quickFilters": { "myFilterFlag": "false" } } |

Once the sample code is copied into the script, insert and replace each of the parameter values to the corresponding location in the code in the required format. Upon successful code editing, the HTTP statement in OpenScript will look like the POST request shown in Table 15 below.

TABLE 15: SAMPLE RESTFUL POST REQUEST IN JAVA CODE VIEW

HTTP POST REQUEST STATEMENT IN OPENSRIPT

```
// create variable "header" and add header information

Headers headers = new Headers();
Header headerContentType = new Header("Content-Type", "application/json; charset=UTF-8",
HeaderAction.Modify);headers.add(headerContentType);

// create a variable postBody and place the post data parameters, which are JSON literals in this example

String postBody =
"\{"userid": "testuser","\projectId": "12345","\dataFilters": {},\quickFilters": {\myFilterFlag": "false"}}";

// insert URL, add header information and body content, and specify the method (POST in this example)

http.navigate (null,
    "https://mytestserver.oracle.com/myservices ",
    postBody.getBytes("UTF-8"),
    headers,
    true,
    "UTF-8",
    HTTPMethod.POST);
```

Constructing RESTful requests in OpenScript require users to manually write Java code into the script. It is a straightforward process once the API formatting and arguments are understood, but can still be tricky for non-Java users to write code. As a shortcut, it is recommended to use the sample code in this section as a template using a 2 step process. 1) Open Java code view in OpenScript to copy the code and paste into the script's Run section, 2) modify the argument values. The next tip will discuss helpful code-assist utilities that OpenScript provides when editing script in the Java Code view.

TIP 18: Take advantage of helpful utilities constructing REST requests

OpenScript is built on top of the Eclipse IDE. By default, OpenScript shows its user interface in the **Tester Perspective**, which hides the code debugging features that Eclipse originally provides²¹. However, many of these features are still accessible from the Tester Perspective's Java Code view. This section talks about some of the useful code assist utilities that are made available by Eclipse.

1. OpenScript API requires adding import statements to the script. For example, copying the sample code from Table 13 into the new script may cause a syntax error, as Headers API is not imported into the new script yet. Right click on Java Code view > Source > **Organize imports** to automatically add required import statements. The following statement will be automatically added into the script.
 - » Import oracle.oats.scripting.modules.http.api.Headers;
 - » import oracle.oats.scripting.modules.http.api.Header;

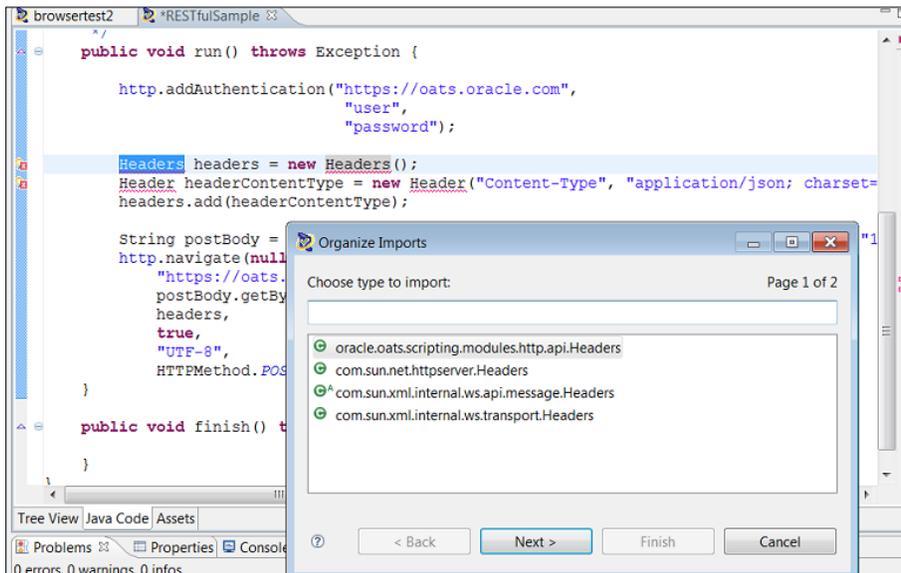


Figure 29. Organize Imports feature automatically adds required Java Imports into the script.

2. Another utility is **Key Assist**. Select Help > **Key Assist** to show the list of soft keys that may be useful. For example, to format the selected lines of code, press `Ctrl + Shift + F` keys.



Figure 30. Key Assist feature displays a list of shortcut keys available in the Java code view.

²¹ To enable original Eclipse user interface, select: View > Developer Perspective from the OpenScript menu.

- When constructing POST data manually, double quotes present in the Post data must be properly escaped. However, manually escaping JSON literals is cumbersome, and can be prone to errors. Use OpenScript (Eclipse) feature that automatically escapes the selected text. View > OpenScript preferences > Click on the "Clear" icon that is next to the text box. Java > Editor > Typing > Check **"Escape text when pasting into a string literal"**.

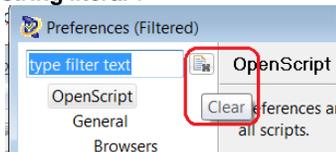


Figure 31. Click Clear icon that is next to the "Type filter text" text box to show the original Tree view of Eclipse

Once the check box is checked, auto-escape is in effect. Copy and paste the JSON literals into the script using OpenScript's Java Code view. The text will be automatically escaped in Java syntax, see the example below.

» **Original text:**

```
String postBody =
("{\"userid": "testuser", "projectId": "12345", "dataFilters": {}, "quickFilters": {"myFilterFlag": "false"}}")
```

» **Formatted text:**

```
String postBody =
("{\"userid\": \"testuser\", \"projectId\": \"12345\", \"dataFilters\": {}, \"quickFilters\": {\"myFilterFlag\": \"false\"}}");
```

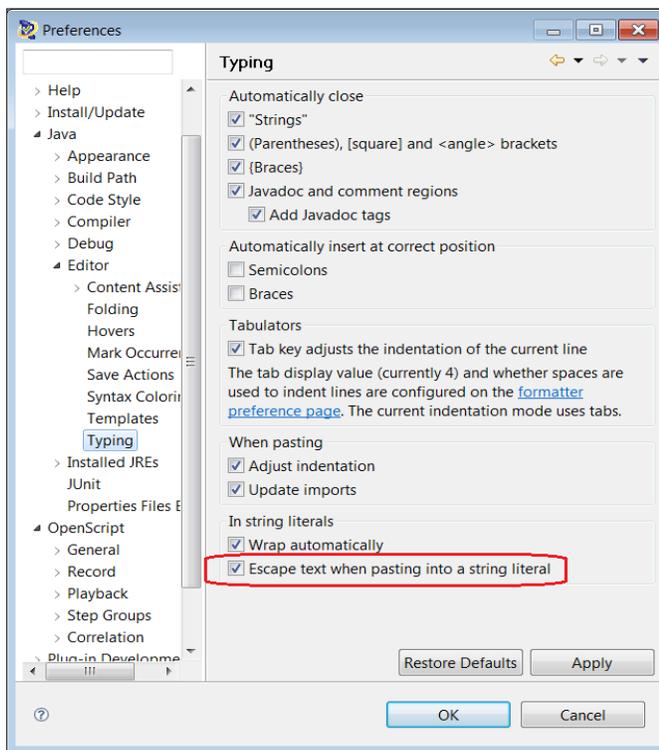


Figure 32. Enable the setting "Escape text when pasting into a string literal" to automatically escape JSON literals.

Tip 19: Record mobile applications directly from mobile devices

As mobile applications rapidly gain popularity, requirement to load test such applications is also gaining increasing attention. A quick way to load test mobile web applications is to record a script on a personal computer using Internet Explorer or Firefox browser, and modify the user-agent header string at playback to emulate the mobile client. Unlike a functional test which verifies the functionality of the application's user-interface, load test verifies the performance of the application's back-end servers. As these infrastructures reside on the data centers and not on the mobile devices, recording the script in a personal computer is a valid method to effectively load test mobile applications. However, this technique cannot be used for recording mobile native applications as they do not run within browsers. To test mobile native applications, HTTP traffic has to be captured directly from smart phones or tablet devices.

Following are the two most common scenarios encountered in the real world while creating load test scripts for mobile native applications with underlying HTTP protocol, or mobile web applications which runs within a browser.

Record Mobile Transactions with OpenScript proxy

The first model is to use OpenScript proxy. The expected use case is when OpenScript machine and mobile device are on the same network without any special setup required to gain network/internet access. This is usually a home network without VPN.

- » Launch OpenScript and Go to View > OpenScript Preferences > OpenScript > Record > HTTP > General Tab.
- » Uncheck **"Only record requests originating from the local machine"** > Click Ok.
- » Create a new load test script and click record. This will start the recording.
- » Make a note of the IP address of the machine where OpenScript is running. This value needs to be entered into the mobile device at a later step. In the below example, the IP address is 192.168.1.67.

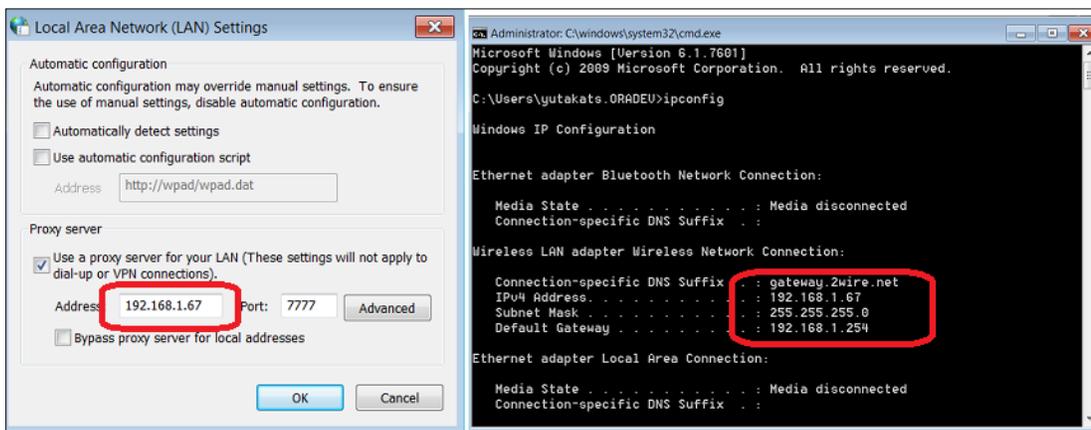


Figure 33. Verify the IP address by typing ipconfig command into the command prompt (right). IP address can also be identified by launching Internet Option dialog during the OpenScript recording (left) ²².

²² OpenScript disables browser's original proxy and applies its own proxy & port when the recording is enabled. Browser proxy can still be used by enabling the Chain Proxy setting in OpenScript preference. View > OpenScript Preferences > Record > HTTP > Proxy Settings > enable Chain Proxy settings.

- » In the mobile device, such as iPad, configure proxy for the network connection. (Proxy IP - <IP address>, Proxy port - 7777).

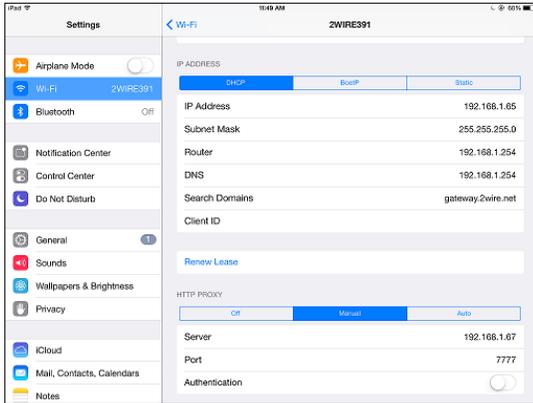


Figure 34. On the mobile device, set IP address and port number in the WiFi configuration settings. In this example, an IP address 192.168.1.67 is set in the Proxy server field, and 7777 is set for the Port.

- » Open a browser, such as Safari, in the mobile device.
- » Navigate to a website of the application to record against.
- » HTTP commands are recorded into the script. Traffic from mobile native applications can also be recorded as long as the underlying protocol is HTTP/S.

Record Mobile Transactions with WiFi tethering

Simply using OpenScript proxy can prevent the HTTP traffic from being recorded between devices if they reside on a different network or require enabling VPN. The following cases may prevent traffic from being captured between the devices.

-
- OpenScript machine is connected to office network using LAN and mobile device is connected to office network using WiFi + VPN*
 - OpenScript machine is connected to office network using WiFi and mobile device is connected to office network using WiFi + VPN*
 - OpenScript machine is connected to office network using WiFi + VPN and mobile device is connected to office network using WiFi + VPN*
-

WiFi tethering can be used to record traffic in such cases. This model allows mobile devices to access corporate networks without connecting to VPN by setting a wireless hotspot from the Ethernet connection on the OpenScript machine and using that connection to access the application from the mobile device.

A common use case is when a user is sitting at her/his office. The OpenScript machine is connected to the office network with an Ethernet cable. The user attempts to record HTTP traffic from a mobile application that runs on the corporate network from a mobile device without connecting over VPN. As typical business applications run within the corporate network, recording such mobile applications require usage of WiFi tethering.

Recording a script using WiFi tethering can be done in 4 steps. First, create virtual Wireless hotspot in an OpenScript machine using Microsoft Virtual WiFi Miniport adapter. Second, connect to the access point from the Mobile device. Third, start OpenScript recording. Forth, specify the proxy IP and port number into the WiFi setting on the mobile device so that OpenScript can capture the user transactions performed on the mobile device.

As a prerequisite, Microsoft Virtual WiFi Miniport Adapter, a Windows component, must be installed on the OpenScript machine. To check whether the Virtual WiFi Miniport Adapter is installed, go to Control Panel > System > Click Device Manager > and Expand Network adapters node.

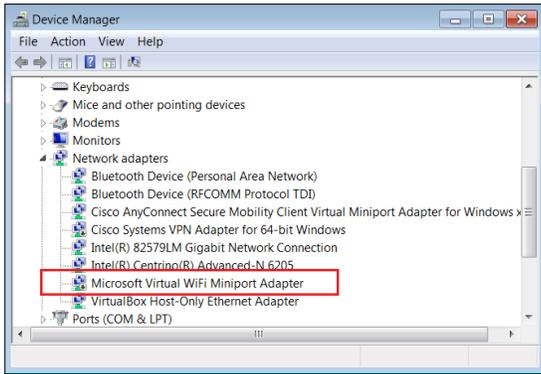


Figure 35. Microsoft Virtual WiFi Miniport Adapter must exist in the Device Manager.

Set up WiFi hotspot from Ethernet

- » Disconnect laptop from the network (both Ethernet and WiFi).
- » From the OpenScript machine go to Control Panel > Network and Sharing Center.
- » Click **Manage wireless networks** (left side menu).

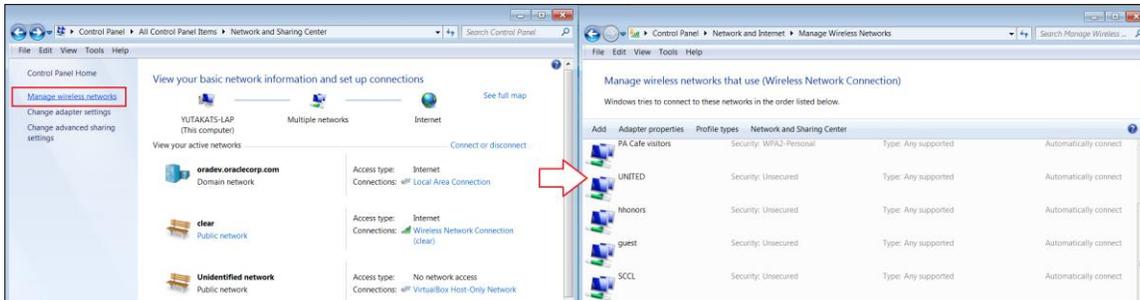


Figure 36. Click Manage wireless networks link to open the list of existing access points.

- » It is recommended to remove the active access points to prevent the machine from connecting to those networks using WiFi automatically.
- » Connect the laptop to the network using Ethernet cable and enable WiFi connection.
- » Go back to the Network and Sharing Center > Click **Change adapter settings** (left side menu)

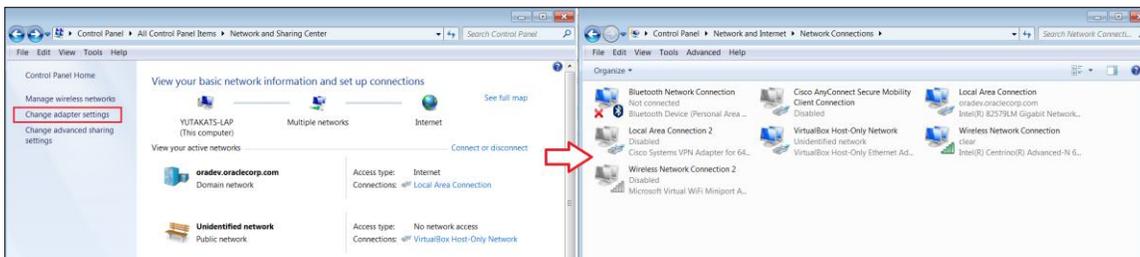


Figure 37. Click Change adapter settings to open Network Connections window.

- » Find "Wireless Network Connection 2" with the adapter name, "Microsoft Virtual WiFi Miniport Adapter"²³
- » If "Wireless Network Connection 2" is disabled, enable it (Right click > Enable)

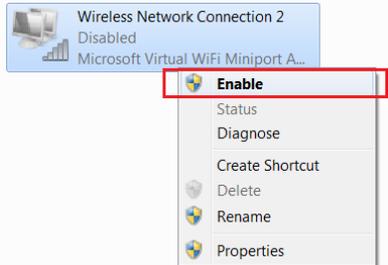


Figure 38. Right click on Wireless Network Connection 2 and select Enable.

Go to "Local Area Connection" > Right Click > Properties. This will open the Properties dialog.

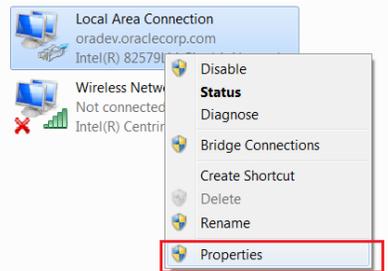


Figure 39. Right click on Local Area Connection to launch Properties dialog.

- » Click on **Sharing** tab > Check "Allow other network users to connect through this computer's internet connection".
- » Under "Home networking connection", select "Wireless Network Connection 2"²⁴

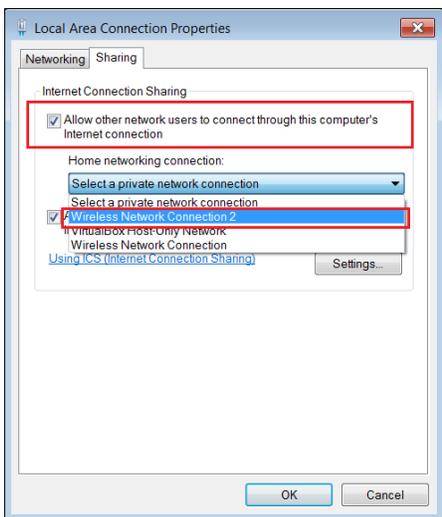


Figure 40. Share the Ethernet connection with the specified connection.

²³ "Wireless Network Connection 2" can be 3 or 4 depending on whether other wireless connections are set up on the system.

²⁴ "Wireless Network Connection" is the primary wireless adapter and "Wireless Network Connection 2" is the "virtual" WiFi adapter (named "Microsoft Virtual WiFi Miniport Adapter"). If the adapter is not listed in the pull down menu, make sure the adapter exists and is enabled. If not installed, search the Microsoft website for instructions to install the driver.

» Open the Command Prompt and run the following command ^{25,26}

```
netsh wlan set hostednetwork mode=allow ssid="Wireless Network Connection 2" key=password  
netsh wlan start hostednetwork
```



Figure 41. Execute the command to start the hostednetwork.

Connect to the hotspot from the mobile device

- » Connect the mobile device to "Wireless Network Connection 2" access point. Enter the password of the connection. If you followed the instruction above, the password is "password".
- » At this point the mobile device can use the hotspot set up by OpenScript machine to access the network.



Figure 42. From the mobile device, connect to the wireless hotspot set up on the OpenScript machine.

²⁵ If the hosted network cannot be started, verify the WiFi connection is enabled on the OpenScript machine.

²⁶ Click on Start > type "cmd" into Search program and files field. Right click on cmd.exe and select "Run as administrator".

Start OpenScript recording

- » Launch OpenScript > Create a new load test script.
- » Go to View Menu > OpenScript Preferences > OpenScript > Record > HTTP > General Tab.
- » Uncheck "Only record requests originating from the local machine" > Click Ok.
- » For "Network Interface", select the one which has "Microsoft Virtual WiFi...".
- » For the Record Mode, select HTTP²⁷.

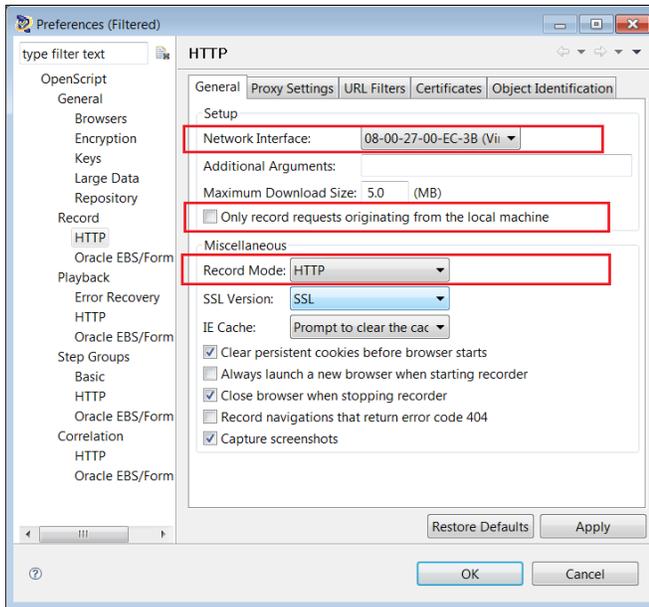


Figure 43. Allow OpenScript to capture traffic from the remote devices from the preference dialog.

- » Click Start button on the toolbar to start the recording.
- » Go to command prompt and type "ipconfig". Note down the IP address of "Wireless Network Connection 2".

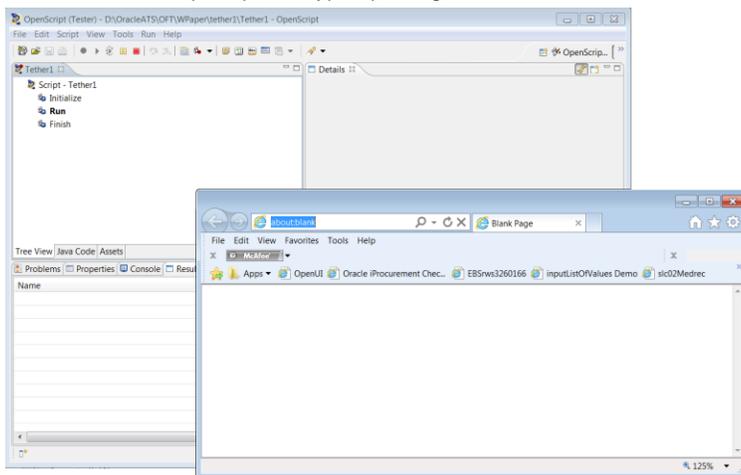


Figure 44. Hit the record button. Do not close the IE browser opened by OpenScript.

²⁷ Record mode can be either Web or HTTP, but the traffic will be captured as HTTP script. Selecting Web mode may delay capturing navigations.

Set proxy configuration on the mobile device

- » On the mobile device, enter the proxy IP and port into the WiFi connection settings. Proxy IP is the IP address of Wireless Network Connection 2, and Proxy port is 7777²⁸.

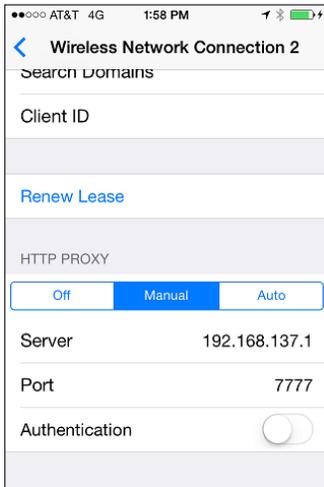


Figure 45. In the WiFi configuration of the mobile device, specify the IP and port of the OpenScript proxy.

- » Open browser on the device, and navigate to the application to record. The traffic is recorded into the script.

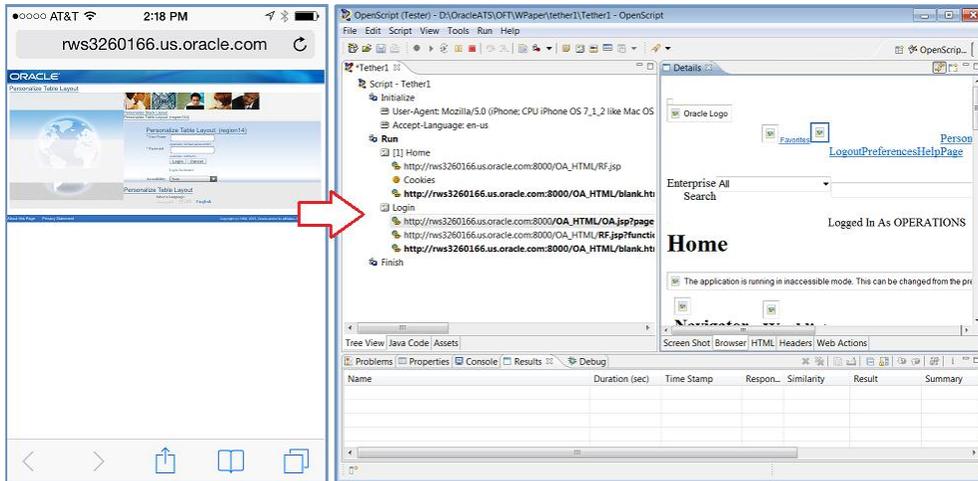


Figure 46. Navigations performed on the mobile device (left) are recorded into the script on the OpenScript machine.

- » When done with recording, run the following command in command prompt to stop WiFi hotspot²⁹.

```
netsh wlan stop hostednetwork
```

²⁸ If unable to record the traffic, verify the proxy IP on the OpenScript machine. Start the recording > Windows Start > type "internet option" in Find programs and files > Click connection tab > LAN settings. Verify the IP address in the Proxy server section and use that IP address on the mobile device.

²⁹ When done with the recording, it is recommended to set back the settings in Local Connection properties and disable the Wireless connection 2 connection.

Tip 20: Record a test flow in two scripts to pinpoint dynamic session parameters

OpenScript provides powerful and intuitive debugging assistance features to help users identify where in the script correlations may be missing. However, because OpenScript compares request headers between recorded and playback based on the traffic captured into the script, it has a limitation in identifying which of them are truly dynamic over different sessions, iterations, or with data parameterization.

For example, a post data request recorded into a script may have two parameter-value pairs, Param1 and Param2. During the recording, OpenScript applies auto correlation to the navigation parameters based on the correlation rules defined in the library. As a result, in this example, a correlation was added to Param1, but Param2 was left uncorrelated, as OpenScript assumed the parameter is static.

- » Param1="{{myVar_param1, 12345}}"
- » Param2 ="default"

Playback the navigation above and go to the Comparison Tab, select request header comparison and find Param1. The difference between the recorded and playback values are highlighted as shown in Table 16 below. This is the expected result as Param1 was correlated, and used dynamic value "09876" for the playback.

TABLE 16: REQUEST HEADER COMPARISON

| Recorded | Playback |
|--|--|
| Postdata: Param1=12345 &Param2=default | Postdata: Param1=09876 &Param2=default |

On the other hand, the value of the second parameter, Param2 has the same value "default" for both recorded and playback. This result is also expected as the parameter is hard-coded into the script, meaning the playback will send the same parameter value every time the script is executed. But then how can we make sure that Param2 is not dynamic? Be aware that the current comparison feature that OpenScript has cannot determine whether the parameters are actually dynamic or not, if not already correlated in the script.

What Table 16 is comparing for Param2 is **the request sent by the browser for one session** and **the request sent during playback using the recorded parameter**. Both recording and playback request sends the value "default" for the Param2 parameter.

What really needed to be compared is **the request sent by the browser for one session** and **the request sent by the browser for another session**. This is the only way to determine whether the parameter has a different value when requested in a different session.

To verify whether a parameter is dynamic or not, a user can correlate the value in the script and playback to validate whether the value actually changes over sessions or not. However, there may be hundreds of parameters in the script, and correlating every single parameter is not only tedious and time consuming, but also unnecessary. Excessive correlation can cause agent performance to degrade.

An alternative and quicker option is to create a new script and record the same test flow, and arrange the two scripts side by side. Expand the navigations and compare the list of post data parameters between the scripts. This way, users can visually identify which parameters are dynamic in two different sessions. See Figure 47 in the next page for the screenshot.

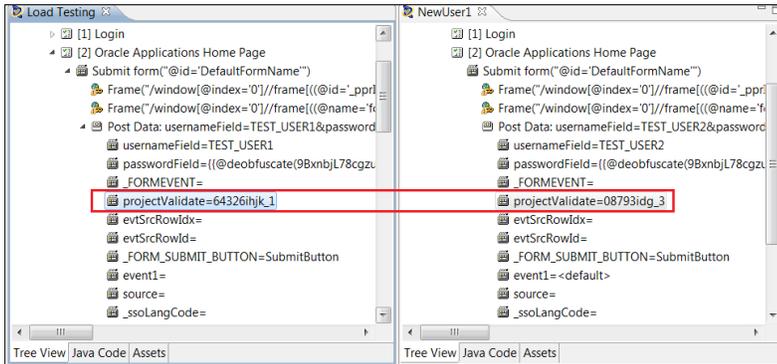


Figure 47. Record two scripts and compare the post data parameters side by side to find out which parameters are dynamic.

Tip 21: Record a flow twice into a single script to isolate parameter changes by iterations

The troubleshooting method from the previous tip can also be applied to the following two similar but different use cases. The first use case is when a playback is successful if a recorded user is used for the application login, but fails when a different user account is used. This is the case when a parameter is static among sessions as long as the same data is used, but it becomes dynamic when a different data is used. This is typically seen when a databank is used to parameterize the login user in the script. To isolate dynamic parameters, which changes when different user login is used, follow the steps below.

- » Record a script with login user A.
- » Create another script and record with login user B.
- » Arrange two scripts in the script view side by side.
- » Compare the request parameters to identify which ones change with a different user account. Refer to Figure 47 from the previous tip.

Another use case is while playback passes with a single iteration, it fails on the second iteration. This is the case when a parameter is static in the first iteration, but it becomes dynamic when a transaction is repeated within the same session. For example, in an ADF based application, a post data parameter name is “**pt1:USma:0:Mat1..**” in the first iteration. However, when the transaction is iterated without logging out from the session, the parameter name changes to “**pt1:USma:0:Mat2..**” and increments the value after “Mat” as the transaction iterates. This is rather a rare behavior in typical web applications, but can be seen in complex business applications such as Fusion applications. Table 17 below shows an example where the parameter names are dynamic, but not the values. Parameter values more commonly require correlations than parameter names, but either case these parameters, name or value, require correlation as long as they are dynamic.

TABLE 17: DYNAMIC POST DATA PARAMETER NAMES IN FUSION PRC APPLICATIONS

| POST Data Parameter | |
|---------------------------|--|
| 1 st iteration | Postdata: pt1:USma:0:Mat1:1:pt1:Purch1:0:AP1:r1:0:q1:value20=0 pt1:USma:0:Mat1:1:pt1:Purch1:0:AP1:r1:0:q1:j_id108:value00=CV_SuppA00 |
| 2 nd iteration | Postdata: pt1:USma:0:Mat2:1:pt1:Purch1:0:AP1:r1:0:q1:value20=0 pt1:USma:0:Mat2:1:pt1:Purch1:0:AP1:r1:0:q1:j_id386:value00=CV_SuppA00 |
| 3 rd iteration | Postdata: pt1:USma:0:Mat3:1:pt1:Purch1:0:AP1:r1:0:q1:value20=0 pt1:USma:0:Mat3:1:pt1:Purch1:0:AP1:r1:0:q1:j_id664:value00=CV_SuppA00 |

The dynamic behavior of these parameters are not detectible using OpenScript's built-in comparison tab, as the Diff shows the results of script playback, not how the actual application behaves. Parameters without correlation will use the recorded values and no difference will be seen in the Comparison tab. To reveal the dynamic parameters that change over iterations, the following steps can be followed to record a transaction twice in a single script.

- » Create a new script to use for test purpose only.
- » Start the recording. Repeat the transaction twice within a single session.
- » Expand the target step group nodes in the script view (in Figure 48, the nodes having red arrows).
- » Compare the request parameters between the 1st and 2nd to identify which ones change over iterations.

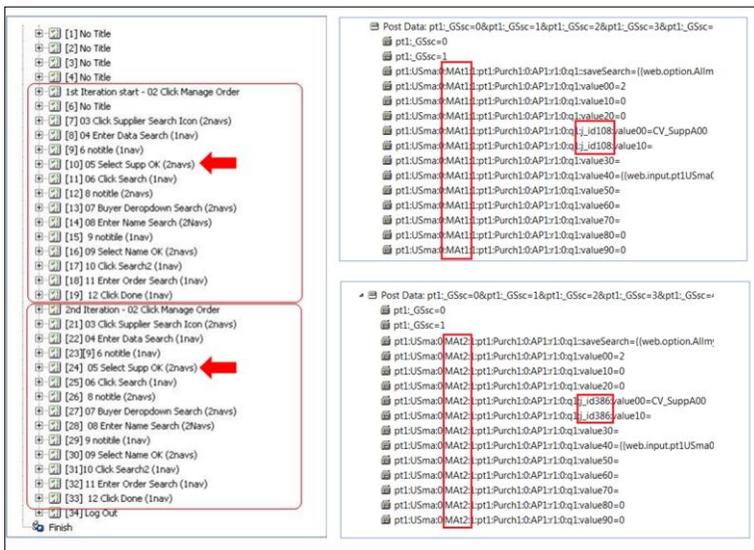


Figure 48. Image shows a script that recorded two iterations within a single flow to verify how session IDs change in the iterations (left image). The right image shows how the Post data parameters changed between the 1st and 2nd iterations when the navigation nodes are expanded.

Tip 22: Create a series of identical EBS test users for use within Oracle Load Testing

When testing Time card creation flow in E-Business Suite applications, there may be a case where script playback is successful at the Time card creation page only when the recorded user is used. Script fails when a different user account is used during playback. This can happen because the two user accounts have different user configurations, which generate different number of rows in the Time card creation page.

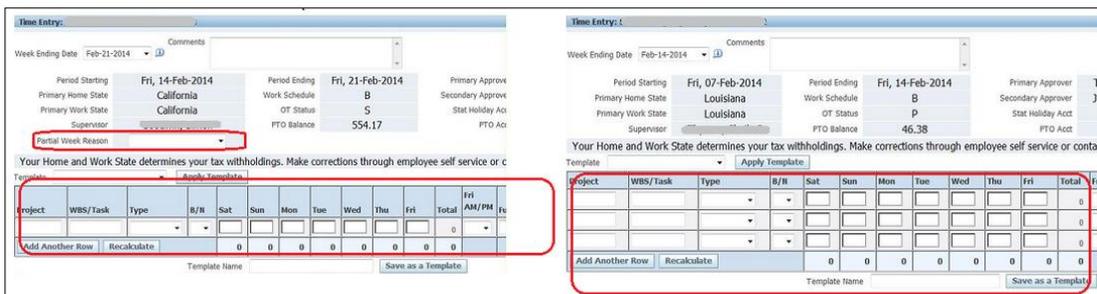


Figure 49. User A (left) has a single row in the table, also has a custom field, while User B (right) has 3 rows and no custom field.

This becomes a problem when requesting the next page, as each cell in the table requires a parameter-value pair in the post data. Because a very different set of post data parameter-value pairs are sent to the server, script playback will likely fail. This is certainly a use case where the troubleshooting method discussed in Tip 20 can be applied.

Create two different scripts and record with User A and User B respectively. Expand the Post data which requests the time card creation, and compare the difference in the post data navigation between the two scripts.

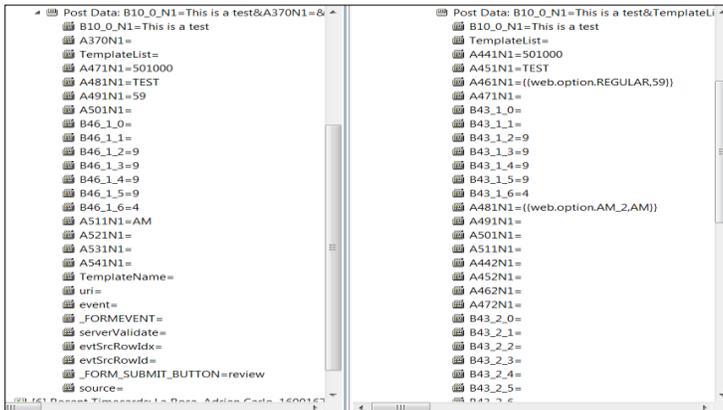


Figure 50. The image shows two scripts, which recorded User A and User B to logging into to the application respectively. The script with User B (right) has a lot more parameters than the script with User A (left).

The root cause of the problem is the different account configurations of the login users used in the scripts. Two users may have different Oracle Time and Labor (OTL) preferences, if the user's account configurations, such as the organizations, HR managers, company associations, payroll requirements, or countries of origin are configured differently between the two accounts. Users having different OTL preferences will have different page layout in the Time Card page that requires different requests. Therefore, randomly picking a number of consecutive EBS users already available in the application will likely cause the same problem, because only a few out of the selected users will have identical, or even similar OTL preferences. Other EBS users will not be able to complete the flow if they run the same script.

Unfortunately this is beyond the level of difference which can be fixed by applying correlations. The only way to run a load test using the current set of users is to group the users with different types of account configurations, and create a script for each group to run. Each script can have a specific group of users with similar account configurations. For example, script1 has databank with users X, Y and Z, which have similar OTL configurations, and script 2 has a databank with users H, J and K, which also have a similar set of OTL configurations. Script 1 and 2 can run within the same load test scenario in an Oracle Load Testing session to make up the total amount of virtual users.

While the example is specific to Timecard, these practices hold true regardless of product module being tested. The best practice to avoid this problem is to work with the application administrator to create a series of dedicated test users specific to each product area to be tested, carefully confirm all test users have identical HR managers, cost centers, operating units, countries of origin, associated profile values and responsibilities.³⁰ These dedicated EBS load test user names and their passwords can then be stored in a single databank for use by the load test scripts. It may require time and effort to generate a list of valid test users, but once a list of users have been successfully created, testers can use them to run valid multiple concurrent virtual users in load test sessions.

³⁰ EBS Application Administrators may find the following MOS article of use for more easily creating multiple identical users. Refer to MOS article ID: How To Create A New User From Existing User With Same Responsibilities (Doc ID 1474294.1)

Tip 23: Import PFX file to test applications with client certificate

The most common method of securing website traffic is using https, where the server sends a digital certificate to the client at the beginning of the https session. This is called one way authentication. Alternatively, the client can also authenticate itself to the server using its own digital certificate. This is called mutual or two-way authentication. In this communication mode, both server and client present their digital certificates to the other party at the beginning of the https session. This section describes the steps to follow to record and playback two-way https authentication sessions using Oracle Load Testing.

Save the PFX file from the client

- » Save the PFX file from the client to the file system (e.g. c:\keys\client1.pfx).

OpenScript requires password protected PFX (or P12) file that is PKCS#12 formatted, in order to record client side authenticated applications. If a user can access the application using a client certificate, the PFX certificate for that user can be exported from IE browser. However, it is recommended to request the application administrator for the PFX file to ensure the PFX file is valid and password protected.

Record with OpenScript

- » View > Preferences > Record > HTTP > **Certificates** tab.
- » Click **Store certificate**.
- » Browse to the PFX file that is saved from the above step. Enter password and Click Ok.
- » Record the transaction from the browser which has client1's certificate imported.

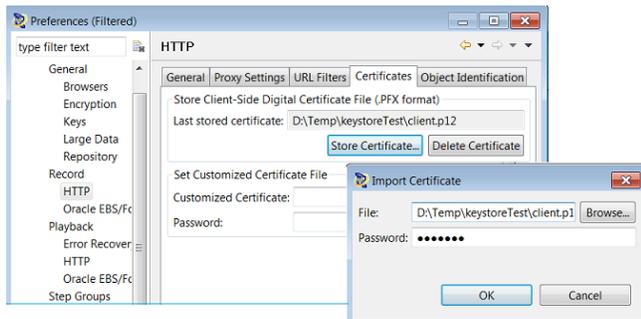


Figure 51. Import the PFX file into the Certificate tab before doing the recording.

Playback with OpenScript

- » Playback requires the client's keystore to be saved in JKS format. Convert the PFX file to JKS format using the commands shown in Table 18. Upon executing the command, client1's certificate and key are saved in JKS format to c:\keys\client1.jks and protected using a password.

TABLE 18: COMMAND TO CONVERT PFX FILE TO JKS FILE

| Command | Arguments |
|--|---|
| C:\OracleATS\openScript\jre\bin\java -jar C:\OracleATS\openScript\PFXConverter.jar c:\keys\client1.pfx <pfx-password> c:\keys\client1.jks <jks-password> | source file (PFX) & password destination file (JKS) & password |

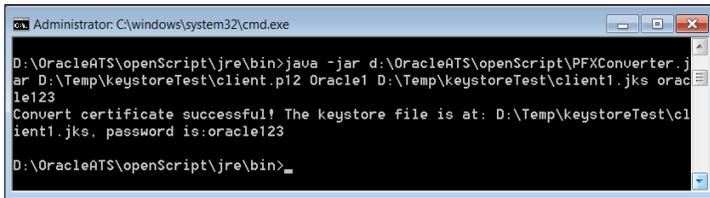


Figure 52. The Image shows a command prompt executing the conversion command. Upon successful conversion, a message “Convert certificate successful” will be displayed.

- » In the recorded script, go to Tree view > Initialize > Right click > Add > Other...
- » Expand HTTP > Select **Load Key Store** > Click Ok.
- » Select the JKS keystore file created above and enter the password > Click Ok.

```
public void initialize() throws Exception {
    http.setUserAgent("Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident; .NET4.0E; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 2.0.50727)");
    http.setAcceptLanguage("en-US");
    http.loadKeystore("D:\\Temp\\keystoreTest\\client1.jks",
        deobfuscate("6GaD7eW3kGVe5TKHmuI/+w=="));
}
```

Figure 53. How the command looks like in Java code.

Playback with multiple client certificates

- » Convert the PFX file for each user into JKS format, as shown in Table 19. If there are 3 users, 3 PFX certificates are required to create 3 JKS files to run multiple users in Oracle Load Testing.
- » Store the file path for the JKS files to a databank (csv) file (e.g. keystore.csv).

TABLE 19: SAMPLE DATABANK FILE CONTAINING A LIST OF JKS FILE PATH

| JKSname | Password | Filepath |
|----------|-----------|------------------------------|
| client1, | welcome1, | d:\temp\keystore\client1.jks |
| client2, | welcome1, | d:\temp\keystore\client2.jks |
| client3, | welcome1, | d:\temp\keystore\client3.jks |

- » Add databank created above to the script in the Assets tab.
- » Parameterize the key store name, so that it will use different key store names for each virtual user. Below example parameterizes file names only. Note: The editing has to be done in the Java code view.

TABLE 20: SAMPLE CODE DATABANKING KEYSTORE NAME

```
String keystore = "C:\\keys\\" + eval("${db.keystore.JKSname}") + ".jks";
http.loadKeystore(keystore, deobfuscate("4zusrct2Oc8OaAiVmHHImw=="));
```

- » When the script is executed with multiple virtual users in Oracle Load Testing, each virtual user presents the certificate from one of the key stores. Note: JKS files have to be located where OLT agents can access using the file path specified in the script.

Tip 24: Add databank only after a playback using original data is successful

Once a script is recorded, the next step a user may want to do is add a databank to the script to parameterize the user login, or use different purchase order numbers to drive the script. There may be situations where the original data can be used only once, and therefore parameterization is unavoidable. However, except for those rare cases, adding a databank should be done only after ensuring the script playback with the original data is successful.

For example, the user did not verify the playback with the recorded data, and proceeds to add a databank to parameterize the user login. He ran the script and then playback failed. In such cases it is hard to isolate the true reason of the problem, whether it was caused by the base script correlation or because a different data was used. Without knowing the status of the original script's stability, introducing additional factors into the script just makes the troubleshooting difficult.

As best practice for creating a load testing script, it is recommended to introduce a single factor at a time, verifying the playback at each step. Gradually expose the script to complexity from a single run to multiple concurrent runs. The script creation and verification can be done step by step in the following order.

TABLE 21: STEPS AND CHECKPOINTS FOR CREATING ROBUST LOAD TEST SCRIPTS

| Scenario | Checkpoints |
|--|---|
| 1 Run a single iteration in OpenScript with recorded data | Playback is successful using the recorded data. Isolates issues from running a script in different sessions. |
| 2 Run multiple iterations in OpenScript with recorded data | Playback is successful with repeated transactions. Isolates issues with repeated transactions. |
| 3 Run multiple iterations in OpenScript with databank | Playback is successful using parameterized data. Isolates issues with using different data to drive the script. |
| 4 Run 1 VU (Virtual User) in OLT with a single iteration with databank | An OLT session with one VU (using a script that runs fine in OpenScript) is successful. Isolates configuration and environmental issues specific to OLT. |
| 5 Run 2 concurrent VUs in OLT with multiple iterations with databank | An OLT session with two or more VU is successful. Isolates data conflict issues. E.g. Two VUs update a single PO# at the same time and creates a database lock. |
| 6 Run 10 or more concurrent VUs in OLT with multiple iterations with databank | An OLT session with a relatively small amount of VUs is successful. Further ensure there is no data or flow conflict problems. |
| 7 Run 100 or more concurrent VUs in OLT with multiple iterations with databank | Run VUs up to the expected production load. Isolates OLT environment issues such as agent or controller capacity problems. |

Upon the completion of each step, run a checkpoint to ensure successful script playback. This way, the user can narrow down to where the problem might reside, in the case of a script playback failure.

Tip 25: Tune script for better performance before adding to the load test scenario

Now the script is ready to be added into a scenario in Oracle Load Testing and run multiple concurrent users to simulate various production workloads. However, a script runs without failure does not necessarily mean it runs with the best performance. OpenScript is a user friendly product with an out-of-the-box feature to automatically correlate dynamic session variables. Due to this, OpenScript tends to add more content validation than is necessary. While more content validations ensure the test flow consistency, it can hurt the agent performance during a load test session when multiple concurrent users are running.

In fact, not all variables created by the auto-correlation process are dynamic. Those unnecessary variables can be safely removed from the script, and by doing so, it contributes to a better agent performance. OpenScript has a built-in feature to remove variables that are not used for string substitution.

- » Select Tools > **Remove Unchanging Variables** from the OpenScript menu.
- » “Remove Unchanged Variable” dialog opens and shows a list of variables with recorded and playback value³¹.
- » For variables which have the same value in both recorded and playback, their checkboxes are automatically selected.
- » Click OK button to remove the checked variables, as well as their string substitutions permanently from the script.

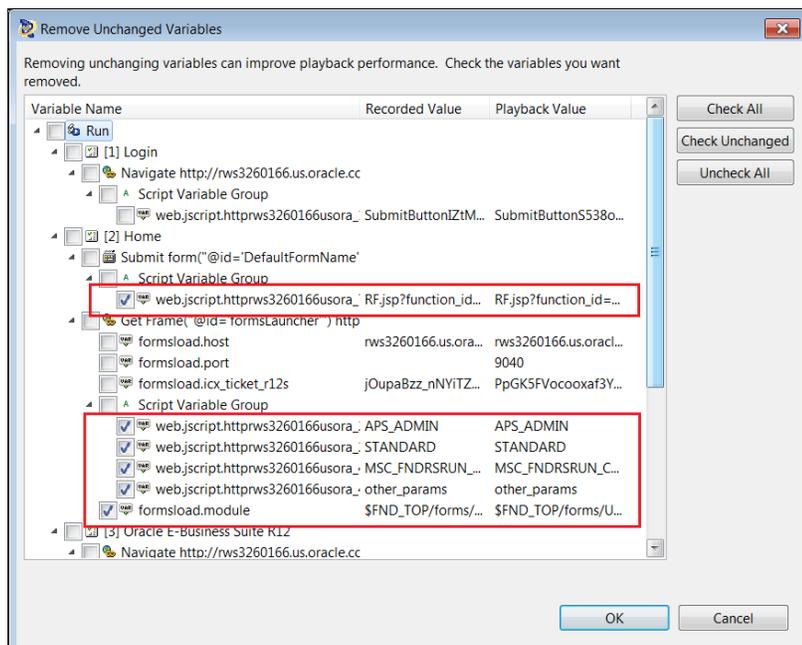


Figure 54. Remove Unchanged Variables dialog shows a list of variables with values from recorded and playback. It also shows which of these variables have the same values between recorded and playback.

In addition to the variable removal, debugging codes and verification checkpoints such as text matching tests should be removed from the script too. As best practice, save the original script and keep the debugging codes so that one can always go back to this script if the script needs refinement. Select File > **Save As** from OpenScript menu, name it as, for example, “Master debugging”. Next, duplicate the script with a different name such as “Production load” (or “Go-Baby-Go”) and remove the debugging and verification codes. The new script will be used only for load generation, and therefore should not have any debugging codes or verification checkpoints which could affect the agent performance.

This is the end of the 25 Tips for creating effective load test scripts for Oracle Load Testing. Once the scripts are optimized, you are ready to add your script to a scenario in Oracle Load Testing and start your load test session. Good luck!

³¹ Playback values are extracted from the latest playback run. If there is no playback result, Playback value will be blank.



Conclusion

How IT evolves can be compared to playing non-stop disco music. Inventions, improvements and innovations will continue to change the IT trend. Business applications have a mission to adapt to new IT trends, as staying where they are is not a choice. As new technologies emerge, users expect a more intuitive interface and a faster response to the applications, with or without knowing the underlying complexities of the new technology.

Successfully load testing such evolving applications is a challenging task. Testers are required to validate the performance of the applications, which are more complex than ever, within the constraints of a shorter release cycle. And in application load testing, traditionally the scripting process is known to be the hardest. Creating successful load test scripts can take up to 90% of the time and effort for an entire load testing project.

OpenScript, a scripting platform of Oracle Load Testing, provides open and extensible automated load test scripting solutions that can flexibly apply to various web technologies. This white paper consolidated the advanced scripting techniques gathered by ATS experts combining many years of load testing experience using OpenScript.



Author: Yutaka Takatsu

Contributing Authors: Raja Vengala, Karilyn Loui, Jerry Ji, Mike F. Smith

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200



CONNECT WITH US

-  blogs.oracle.com/oracle
-  facebook.com/oracle
-  twitter.com/oracle
-  oracle.com

Hardware and Software, Engineered to Work Together

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0914