Setting Up Selenium for Web Client UI Testing

This chapter provides information on how to set up Selenium for visual testing the Servoy solutions in Web Client. Selenium IDE record and playback the user clicks on the UI.

In This Chapter

Setup Selenium

- Install the Selenium IDE plugin for Firefox from http://seleniumhq.org/download/
- Create a folder `wicketPathLocatorBuilder` in the local drive with the file `user-extension.js.wicketPathLocatorBuilder` inside. To add the wicketpath locator into the Selenium IDE Locator Builder options paste the following snippet into this file.

```
LocatorBuilders.add('wicketpath', function(e) {
  var path = '';
  var current = e;
  while (current != null) {
    if (current.parentNode != null) {
      path = this.relativeXPathFromParent(current) + path;
      if (1 == current.parentNode.nodeType && // ELEMENT_NODE
          current.parentNode.getAttribute("wicketpath")) {
        return this.preciseXPath("//" + this.xpathHtmlElement(current.parentNode.nodeName.toLowerCase()) +
            "[@wicketpath=" + this.attributeValue(current.parentNode.getAttribute('wicketpath')) + "]" +
            path, e);
      }
    } else {
      return null;
    }
    current = current.parentNode;
  }
  return null;
    });
```

- Add the following code inside the `user-extension.js.wicketPathLocatorBuilder` to include the command waitForWicketAjax as a Selenium IDE extension. The waitForWicketAjax command checks if there is a pending wicket Ajax request from the current Browser window and will wait until the wicket Ajax request is completed.

```
/*
 * Some usefull links:
 * http://www.packtpub.com/article/user-extensions-add-ons-selenium-testing-tools
 * http://selenium.polteq.com/en/
 */
Selenium.prototype.isWicketAjaxReady = function(locator, text) {
        //TODO extend with checking JQuery.ajax as well. See http://hedleyproctor.com/2012/07/effective-
selenium-testing/
        if (!selenium.browserbot.getCurrentWindow().wicketAjaxBusy) {
                var doc = selenium.browserbot.getCurrentWindow().document;
                var scriptTag = doc.createElement("script");
                scriptTag.type = "text/javascript"
                var script = 'wicketAjaxBusy = function() {for (var c in Wicket.channelManager.channels)
{if (Wicket.channelManager.channels[c].busy) { Wicket.Log.info("Channel " + c + " is busy");return true;}}
Wicket.Log.info("No channels are busy");return false;}';
                try {
                        scriptTag.appendChild(document.createTextNode(script));
                } catch (e) {
                        scriptTag.text = script;
                }
                doc.body.appendChild(scriptTag);
        }
        return !selenium.browserbot.getCurrentWindow().wicketAjaxBusy()
};
```

- Open the Selenium IDE, go to menu **Options** > **Options**. In the **Selenium Core extensions** input field, paste the path to the earlier created folder `wicketPathLocatorBuilder`.
- Go to **Options** > **Locator Builders** and move the wicketpath entry to the top of the list
- Restart FireFox.

Setup Servoy

- Open the Servoy Admin Pages at http://localhost:8080/servoy-admin
- setup web client property `servoy.webclient.debug.wicketpath` to true.
- Save and restart the Application Server.

Selenium Description

Main Window Description

- The left part of the main window contains the tree structure that represents the test-suite. The right side displays the details of a selected node. At the bottom of the main window is the terminal output area, which displays standard messages and communications in between the test-suite and the client application being tested.
- The basic structure of a test-suite and thus the child nodes of the **Test-suite** root node is fixed. An arbitrary number of **Test-set**, **Test-case** or **Test** nodes are followed by the **Procedures**, **Extras** and **Windows** and **components** nodes. The **Procedures** node holds **Packages** and **Procedures**.

Starting the Application

- If Selenium and Servoy Web Client are configured, then the solution can be run on Web Client with Firefox browser.
- When the application is running on browser, then the Selenium can be started using **CTRL+ALT+S** or going to **Firefox** > **Web Developer** > **Selenium IDE**.

Recording with Selenium IDE

Selenium IDE records the events generated by the user in a Test Case as Selenium commands (clicks/ typing/ keyboard events..).. Each command is targeting a specific element of the DOM (The HTML tree generated by the rendering of the Web Client) using the default Locator Builder; the default is the wicketpath locator if it has been correctly placed on top of the Locator Buillder list.

Just recording is not enough for a correct playback of the Test Case. To validate the test should be added UI verification checks. Is the element present after i have clicked the button ? is the result value equal to the expected value ?
When recording Selenium IDE assumes that all the elements are already present in the DOM but this is not always the case. Selenium IDE is asynchronous therefore is necessary to add an explicit command to put Selenium waiting until the result of the triggered action is received. For example a click on the button 'next record' result in showing the next record of the foundset. The result of the action is not immediate therefore the expected value cannot be immediately target with Selenium. Selenium should wait until the result of the click action is complete before targeting the values of the next record. Use the commands `waitFor` to wait a specific element or a specific value to be loaded on the page.

Recording Events

- Click record button
- Go to WC, do the actions needed, typing data / adding records / delete
- When done go to selenium and hit **Stop recording** button
- A sequence is created which can be saved using **CTRL+S**

Recording Gotcha's

- When using `sendKeys`, also add a `fireEvent` with value `blur` on the target for the `sendKeys`, otherwise the `sendKeys` is not applied
- When recording, Selenium inserts `selectWindow` commands. Most of these can be removed. They are only needed when the test needs to switch to another tab or to a different dialog (or back to the main form from a dialog)
- By default Selenium does not handle Ajax events very well when recording. It inserts simple `click` events, but those clicks usually result in an Ajax call to the server. Use the `waitForWicketAjax` command to wait the result from the server before executing the next command. Note that `waitForWicketAjax` fails if there are multiple Ajax call to the server or if the call results in closing the JSWindow (The command would not find the current window and generate an error).
- When `waitForWicketAjax` cannot be used wait for a specific element to be loaded instead.
- Use the xxxAndWait command when the action causes a new page to load. Selenium will wait until the page is loaded.
- Before selecting a value from a valuelist (combobox/typehead) use `waitForSelectOptions` to wait for the value to be loaded on the list.
- When testing resizable components based on JQuery UI Resizable execute a mouseOver command BEFORE the dragAndDrop command. This is needed to bring the resizable component in the right state for Drag 'n' Drop to work. See the comment in JQuery UI's own test code: https://github.com/jquery/jquery-ui/blob/master/tests/unit/resizable/resizable_test_helpers.js

Playback - Running the Script

- If a suite of case is already recorded then can be played from **Play entire test suite** button
- For testing case by case of case suite already recorded then can be played from **Play current test case** button
- TIP: for easier following of the test run, add a default delay: go to **Fast-delay** slide bar and adjust the speed of running cases.

Tip: for more detailed documentation and video tutorials user this link: http://jroller.com/selenium/

Plugins and add-on for Selenium IDE

Many plugins are available for Selenium IDE as Firefox add-on. Listed below some nice to have plugins.

SelBlock

Install the Firefox Extension https://addons.mozilla.org/en-US/firefox/addon/selenium-ide-sel-blocks/. Provides javascript-like conditionals, looping, callable functions, error catching, and JSON/XML driven parametrization to perfom Data Driven tests.

Implicit Wait

Install the Firefox Extension https://addons.mozilla.org/en-US/firefox/addon/selenium-ide-implicit-wait/?src=search. Allows Selenium IDE to automatically wait until the element is found before executing each command using a locator. To use it add the command `setImplicitWaitCondition` at the beginning of each Test Case.

File Logging

Install the Firefox Extension FileLogging: https://addons.mozilla.org/en-US/firefox/addon/file-logging-selenium-ide/?src=search Log the test results of the Selenium IDE into a file. Append text to existing log. does never delete the log file. Use Advanced text editor to view results. Is possible to add a timestamp to any log.

Setup Log extension: Open the Selenium IDE, go to menu **Options** -> **Options** -> **File Logging**.
Select the log file and log level. Default Info

Selenium Best Practice

Use the following best practices to make the automated test easily maintainable.

Select the proper locator

- The selection of a stable locator is the most important step to have a maintainable test script. Element's locator can change frequentely in the DOM resulting in an unmaintenaible test. When recording the Test Case the developer should choose the locator that most likely will not change when modifying the form.
- Is a good practice to store the locator of often used element into Selenium variables. If the locator is then affected by any change would be easier to set the new locator in the test script. In such a way the Test Case becomes also more readable.

ID locator

Target the element having the specific ID. Is the most fragile type of locator since in the Web Client the markup Id of elements change frequentely.

Wicketpath

Target the element using the servoy wicketpath attribute of the element.

Is a stable locator for forms generated with Servoy at Design time; the wicketpath will not change at any edit of the form. The wicketpath is not stable instead when the form or Servoy element is generated at Runtime using the Solution Model. In this case will change at any execution.

Example: //div
[@wicketpath='servoy__page_servoy__dataform_forms_0_webform_servoywebform_View_sv____BC22D853__A837__4D0B__8937__8ED2D086451E_sv____F48C87B8__E69A__46D1__93C2__33F59B96538E__wrapper']/input

XPath: position

Target the element using the exact position of the element in the DOM. This type of locator is unstable for any change happening in the structore of the page but can be use to target element generated with the Solution Model (Note that the Solution Model should always generate the same result otherwise the DOM of the page will look different and most likely the locator will break)

Example: //div[13]/div/div/div[3]/div

XPath: contains text/property

Target the element containing the specific text or the specific property. Resist to changes made to the form at Design time and even to the solution model. Fails if there are multiple elements containing the same text or the same property or if the text/property is changed.

Examples

//div[text()='logout']/../.. find the parent of the parent of the div element having the text equal to 'logout'

//img[contains(@src, "pv_btn_logout.png")] find the element containing the value "pv_btn_logout.png" in the src attribute

FormName-ElementName

Servoy has in it's road map the possibility to provide a Selenium locator based on formName/elementName values which would be more resistant to failure then the mentioned selectors.

Keep the test maintainable

- Store frequently used values in variables.
- Avoid duplicates in the test scripts. Modulate the Test Cases and Test Suites and reuse same Test Cases in multiple Test Suites. Create new extentions or Rollup for frequently used sequence of commands.
- Keep the Test Case and the Test Suites small. Each Test Suite should test a single Use Case. Separate the Test Suite in multiple Test Cases, each Test Case is a single Unit of test.
  Example: create new customer. Create a Test Suite and the following Test Cases: Login, Open form customers, create customer, logout.
- Refactor the test when refactoring the code or changing the form.
- Document the tests. Use proper naming conventions and save the Test Suite and Test cases in a organized folder structure.
- Use setUp and tearDown functions to make the test repeatable. Avoid using other Test Cases as setUp or tearDown, if the test case of the setUp fails then the setup will not be correct. Use Servoy functions as setUp or tearDown. Is possible to inject wicket callback to the Servoy functions into the client libraries. Once the wicket callback is correctly injected execute from Selenium with the command `runScript`