

1. LISA Web 2.0 Guide	2
1.1 PART 1 - LISA Web 2.0 - User Guide	2
1.1.1 1. Introduction to Web 2.0	2
1.1.1.1 1.1 System Requirements	3
1.1.1.2 1.2 Technologies and Platforms	3
1.1.1.3 1.3 Getting Started with LISA Browser	3
1.1.1.3.1 1.3.1 Browser Menu	5
1.1.1.3.2 1.3.2 Browser Toolbar	5
1.1.1.3.3 1.3.3 Browser Settings	7
1.1.1.3.4 1.3.4 Browser & Extension Updates	13
1.1.1.3.5 1.3.5 Browser Architecture	14
1.1.2 2. Recording Mode	16
1.1.2.1 2.1 Recording Example	16
1.1.2.2 2.2 Recording a Swing Test	19
1.1.2.3 2.3 Recording an Applet Test	20
1.1.2.4 2.4 Different Views of a Web Page	25
1.1.2.5 2.5 Post Recording	30
1.1.3 3. Playback Mode	36
1.1.4 4. Edit Mode	39
1.1.4.1 4.1 Event Types	40
1.1.4.2 4.2 Logical Events	43
1.1.4.3 4.3 Object Details	46
1.1.4.4 4.4 Filters	49
1.1.4.5 4.5 Assertions	52
1.1.4.6 4.6 Datasets	54
1.1.4.7 4.7 Editing Steps in Workstation	55
1.1.5 5. Debugging	56
1.1.6 6. Setting up ADF Extensions	59
1.1.7 7. Running Browser Standalone	60
1.1.8 8. Troubleshooting	61
1.1.9 9. Known Limitations	61
1.2 PART 2 - LISA Web 2.0 - How Tos	61
1.2.1 1. Introduction	62
1.2.2 2. Web sites and frameworks	62
1.2.3 3. How To - Generate random data (4.5.1.x)	62
1.2.4 4. How To - Capture Dynamic HTML for later test editing	63
1.2.5 5. How To - Deal with time-sensitive events	64
1.2.6 6. How To - Parametrize dynamic data entry in loops	65
1.2.7 7. How To - Deal with dynamic elements	68
1.2.8 8. How To - Extract complex data from a page	68
1.2.9 9. How To - Ajax auto-complete fields	70
1.2.10 10. How To - Write custom Web 2.0 steps	72
1.2.11 11. How To - Write cross-browser tests	73
1.2.12 12. How To - Use Pathfinder integration	75
1.2.13 13. How To - Write Java Swing and WebStart tests	77
1.2.14 14. How To - Write .NET WinForms tests	80
1.2.15 15. How To - Debug a test	80
1.2.16 16. How To - Use global filters and global assertions	83
1.2.17 17. How To - Interact with external resources	84
1.2.18 18. How To - Run Load Tests	88
1.2.19 19. How To - Run in a non-privileged account or on 64 bit platforms	90
1.2.20 20. How To - Record and replay against non us-english websites	90
1.2.21 21. How To - Run in Crash Dump mode	92
1.3 PART 3 - LISA Web 2.0 - Reference	92
1.3.1 1. Recorder Reference	92
1.3.2 2. Debugger Reference	93
1.3.3 3. Settings Reference	94
1.3.4 4. XPath syntax Reference	98
1.3.5 5. Scripting Objects Reference	99
1.3.6 6. Command line Reference	100
1.4 PART 4 - LISA Web 2.0 - Videos	100
1.5 PART 5 - LISA Web 2.0 - Repository	101
1.5.1 1. Instructions	101
1.5.2 2. Always update	101
1.5.3 3. Update with major revisions changes	101
1.5.4 4. First time update (i.e. only if you're missing these files)	101
1.6 PART 6 - LISA Web 2.0 - FAQ	102

LISA Web 2.0 Guide

LISA Web 2.0 Guide

This LISA Web 2.0 guide user documentation is divided into six parts.

PART 1 - LISA Web 2.0 - User Guide
PART 2 - LISA Web 2.0 - How Tos
PART 3 - LISA Web 2.0 - Reference
PART 4 - LISA Web 2.0 - Videos
PART 5 - LISA Web 2.0 - Repository
PART 6 - LISA Web 2.0 - FAQ

PART 1 - LISA Web 2.0 - User Guide

PART 1 - LISA Web 2.0 - User Guide

Recording a Web Site via DOM Events

LISA Web 2.0 testing, works by letting LISA **emulate a web browser**.

LISA Web 2.0, allows you to record events at the **DOM** (Document Object Model) **level** (such as mouse clicks, mouse movements, keys being typed, etc.) and play those events back as a browser during test execution.

DOM-level testing gives you fine-grained control over what a test can and cannot do, what type of object it can access and what kind of result it can return. In particular, all client-side logic is accessible to it. A Web 2.0 test can easily interact with frames, JavaScript, CSS, Ajax, Plugins, Applets and so forth. Many modern web sites make heavy use of these technologies, Ajax in particular, and those are usually called Web 2.0 sites, hence the term **Web 2.0 tests**. However, any web site can be tested in this fashion.

For more information on Web 2.0, you can also refer to its [How To doc](#) and [Reference Guide](#) in the [LISA Web 2.0 Guide](#).

The following topics are available in this section.

1. [Introduction to Web 2.0](#)
2. [Recording Mode](#)
3. [Playback Mode](#)
4. [Edit Mode](#)
5. [Debugging](#)
6. [Setting up ADF Extensions](#)
7. [Running Browser Standalone](#)
8. [Troubleshooting](#)
9. [Known Limitations](#)

1. Introduction to Web 2.0

1. Introduction to Web 2.0

One of the major strengths of LISA is its outstanding ability to create and run tests that make use of a mix of different technologies (web, j2ee, web services, swing, etc...) as is so often necessary in the enterprise software world.

Web 2.0 tests are no exceptions in this regard and can be mixed with any other type of step. Nothing special is required to achieve this.

Until version 3.5, LISA has fully supported **HTTP-level web testing** and it will continue to do so in future releases.

HTTP-level testing works by having LISA install a **proxy** between itself and the web server. It then captures the HTTP and HTTPS traffic flowing between the client and the web server during a recording session. It submits GET or POST requests during a playback session. For more details

you can consult the LISA web testing chapter in the User Guide.

Later, LISA supported the testing and recording of the Web 2.0 browser. By contrast, the Web 2.0 testing works by letting LISA emulate a web browser. It allows you to **record events at the DOM level** (such as mouse clicks, mouse movements, keys being typed, etc...) and play those events back as a browser during test execution.

There are advantages to each approach:

The HTTP-level testing might be more resistant to client-side changes during test execution since it is only aware of URLs. In addition, it is very lightweight so it is well suited to massive load-testing. DOM-level testing however is more resistant to server-side changes, it gives you much finer-grained control over what a test can and cannot do, what type of object it can access and what kind of result it can return. In particular, all client-side logic is accessible to it. A Web 2.0 test can easily interact with Frames, Javascript, CSS, Ajax, Plugins, Applets and so forth. Many modern web sites make heavy use of these technologies, Ajax in particular.

In addition to web testing, the LISA Web 2.0 can record, replay, and validate other so-called RIA (Rich Internet Applications) such as Java Applets (Swing and AWT), ActiveX controls and in particular Flash and Flex applications.

The following topics are available in this section.

- [1.1 System Requirements](#)
- [1.2 Technologies and Platforms](#)
- [1.3 Getting Started with LISA Browser](#)

1.1 System Requirements

1.1 System Requirements

In addition to the LISA installation on a Windows NT or better machine, the following is also required.

- An install of the .NET 2.0 SP1 runtime (or newer)
- A public jre (1.4 or greater) if you intend to test java applications or use HTTP recording

1.2 Technologies and Platforms

1.2 Technologies and Platforms

Web 2.0 tests might be better described as GUI tests because they support a lot more than basic web tests.

In the pure web realm, any server side-technology or platform is supported (because it's irrelevant from the client's perspective), whereas on the client side Web 2.0 can run Internet Explorer, Mozilla Firefox, or Safari (initial support) making it a truly cross-browser solution (those browsers constitute about 99% of the browser market at the time of this writing).

In addition to web testing, Web 2.0 can record, replay and validate other so-called RIA (Rich Internet Applications) such as Java Applets (Swing and AWT), ActiveX controls and in particular Flash and Flex applications.

Finally Web 2.0 supports non-web hosted technologies. It can record, replay and validate Java desktop applications (Swing, AWT), .NET WinForms or native Win32 applications.


Important Note: The support for all these technologies is deep and does not rely on so-called analog record and replay technology that many tools only have. Analog testing is technology-agnostic because it relies on screen coordinates, which makes it very brittle and not very powerful.

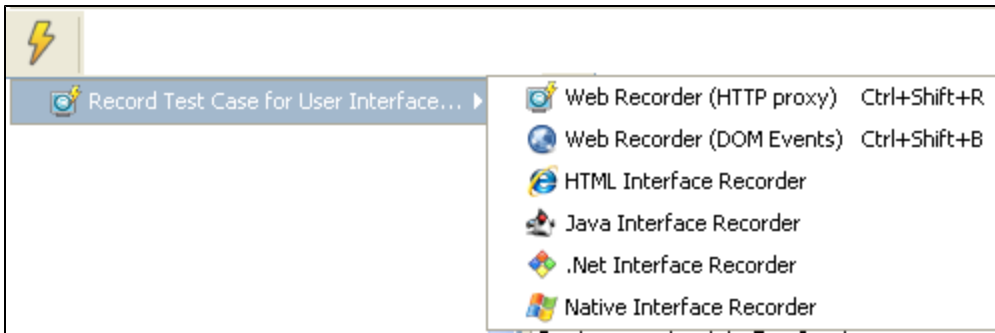
1.3 Getting Started with LISA Browser

1.3 Getting Started

To open the Web Browser,



- Click the  icon on the test case menu. This will open a menu further -



Click **Record Test Case for User Interface > Web Recorder (DOM Events)**...

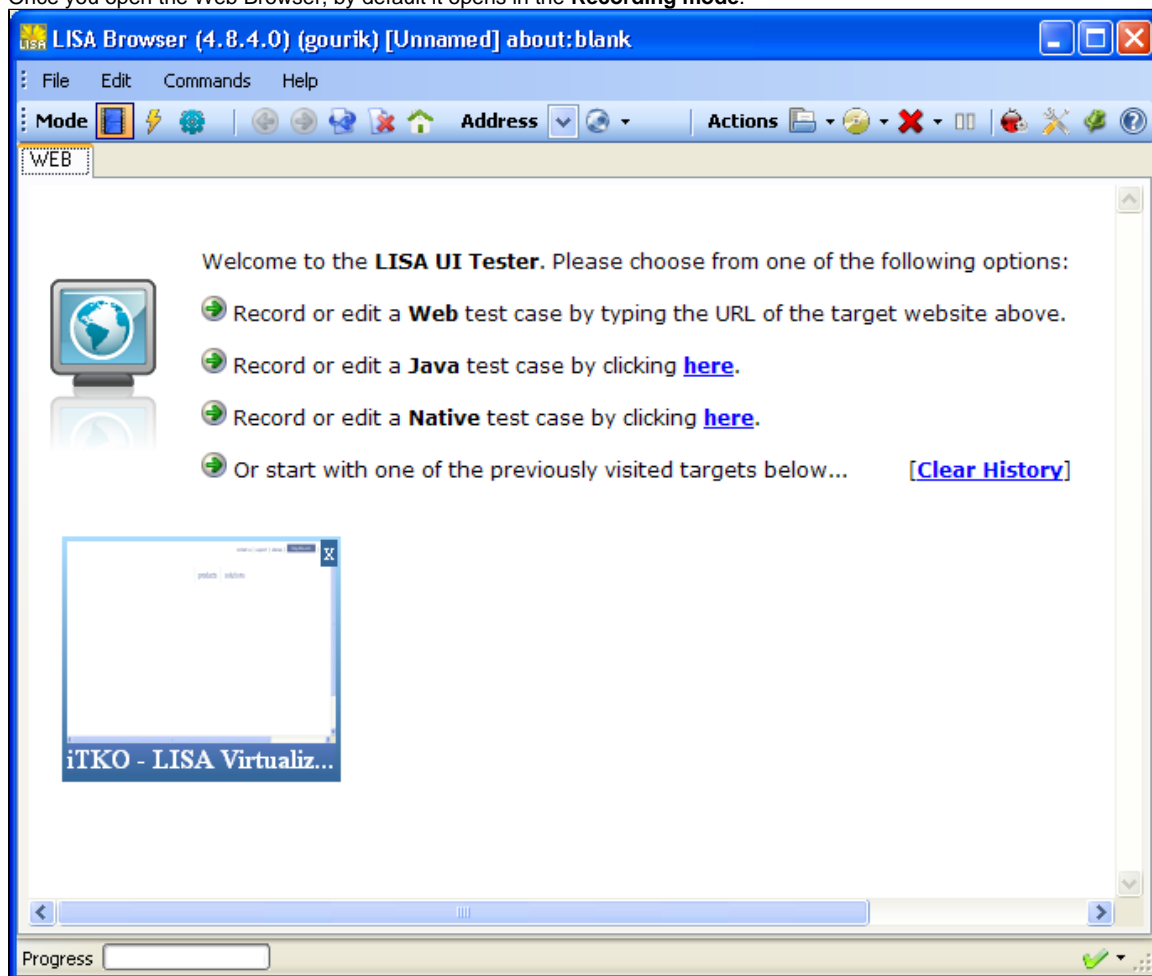
Or click **Actions > Record Test Case for User Interface > Web Recorder (DOM Events)** from the main menu.

This allows you to launch the browser that LISA uses to record and playback Web 2.0 tests.

The first time the browser is launched; there is a wait dialog that indicates it is synchronizing its initial state with LISA. Subsequent invocations will not do this since synchronization will happen on the fly.

Typically, the first interaction you will have with a web test is of recording a session.

Once you open the Web Browser, by default it opens in the **Recording mode**.



There are three main sections in the Web 2.0 Recorder:

- Recording Mode
- Editing Mode
- Playback Mode

Within the browser, there are menus which allow you to Record or edit a **Web or Java or a Native** test case.

You can also see the recently opened web pages if there are any recordings done previously.

More details regarding the same can be found in the subsequent sections.

- [1.3.1 Browser Menu](#)
- [1.3.2 Browser Toolbar](#)
- [1.3.3 Browser Settings](#)
- [1.3.4 Browser & Extension Updates](#)
- [1.3.5 Browser Architecture](#)

1.3.1 Browser Menu

1.3.1 Browser Menu

The LISA Browser opens in the Recording mode by default. The LISA Browser has a typical main menu and a toolbar, which has functions and icons depicting various activities or actions.

Browser Menu

The Browser menu is as shown below and explained.



File Menu

- File > Load** – Loads the current web address
- File > Save** – Saves the current recording
- File > Save As** – Saves the current recording under a different name
- File > Close** – Closes the current recording
- File > Exit** – Exits the LISA Browser

Edit Menu

- Edit > Pause Recording** – Pauses the recording
- Edit > Clear Steps** – Clears all steps
- Edit > Browser Settings** – Opens the Settings dialog box, where you can set the browser settings.
- Edit > Internet Options** – Opens the Internet Properties dialog box, where you can set the Internet options.

Commands Menu

- Commands > Back** – Loads the previous page
- Commands > Forward** – Loads the next page
- Commands > Reload** – Reloads the page
- Commands > Stop** – Stops the recording
- Commands > Toggle Debug Window** – Toggles the debug window
- Commands > Capture Session** – Captures the current recording session

Help Menu

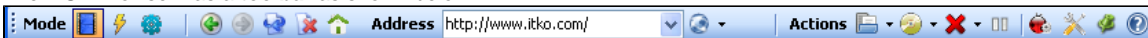
- Help > Documentation** – Opens the documentation page for LISA web 2.0
- Help > Browser Updates** – Opens the LISA Component update dialog box.
- Help > Extension Updates** – Opens the LISA Extension update dialog box.

1.3.2 Browser Toolbar

1.3.2 Browser Toolbar

The LISA Browser opens in the Recording mode by default.

The LISA Browser has a **toolbar** as shown below:



On the **left** of the toolbar, there is the Mode button, where in you can select the Mode of operation within the browser:

- Recording Mode - Where you can record the operation.
- Edit Mode - Where you can edit the transactions.

- Playback Mode - Where you can playback the recorded operation.

By default the Recording mode is selected.

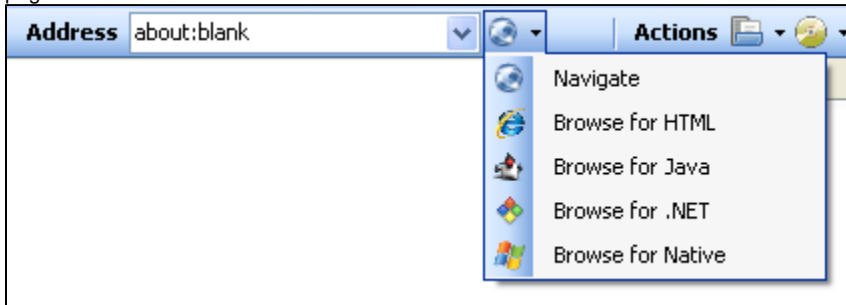
You can Record the web page and playback the recording by clicking on the Playback mode.

You can click on the Edit mode to view add/delete the Logical, Physical events and view the Object details which are described later.

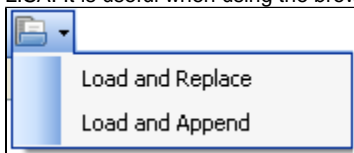
There are the usual **Web** page buttons to take you – Back, Forward, Reload, Abort and Home page.



You can enter the Web page address in the Address bar provided or select the "Go" button to navigate and open HTML/ Java/.Net or Native page.

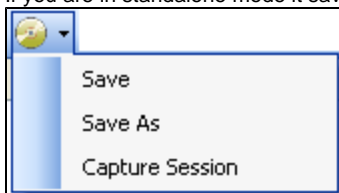


You can Load and Replace or Load and Append from the "**Actions**" button. The Load button is not normally used when using the browser within LISA. It is useful when using the browser is standalone. You can either Load and Replace or Load and Append a previous recording here.

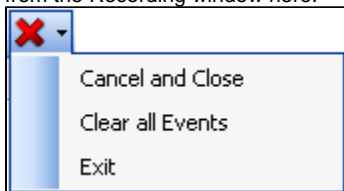


You can Save the recordings from the "**Save**" button. The Save button is what commits the recording to LISA's Test Case, and closes the browser.

If you are in standalone mode it saves the recording to a file. You can do a Save, Save as or Capture a Session here.






You can Close and Cancel the recording from the "**Cancel**" button. The Cancel button simply discards the current recording and closes the browser without modifying the Test Case. The window close button has the same effect. You can also Clear all previous Events here and Exit from the Recording window here.




The following icons are described below:



Icon	Description
	The Pause button allows you to navigate without recording. Recording resumes when it is pressed again. It is useful to skip some undesired events.
	The Debug button allows you to open or close the debug window.

	The Settings button opens the Settings dialog that allows you to configure global behaviors of the both the recorder and the debugger.
	The Pin window button makes the browser (in recorder or debugger mode) stay as a topmost window, which means that it will stay on top of any other window on the screen, even if it does not have the focus. This is useful when you want to observe the tests running while other windows try to grab the focus, especially when you test external applications (like Swing or .NET Winforms).
	The Help button displays the download area from where you can download this document.

The Debug  and the Settings  buttons are described in detail in the next section.


1.3.3 Browser Settings

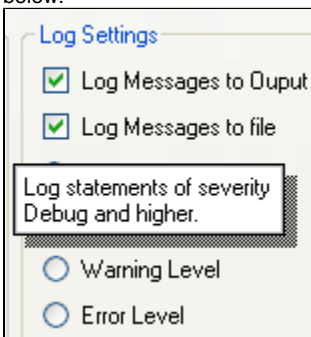
1.3.3 Browser Settings

The LISA Browser Settings allows you to control how the **Recording and Playback** will work.

From the main menu, click **Recording > Settings** or Click the **Settings icon** from the top right toolbar.

The Settings are divided in 4 sections: General, Recording, Playback and Environment.

While the settings window is open, you can quickly lookup the meaning of a setting by clicking the  icon at the top right corner of the window and then the desired setting, or position the mouse over the desired setting and type the F1 key. You will get a tooltip for that field as shown below:



Each playback setting can be overridden on a per test case basis by defining a property (usually but not necessarily in the configuration of the test case or suite), whose name is indicated in the reference and in the help pop-up described above.

For example, there is a setting called "Synchronize Ajax Calls" (which is used to force the browser to treat all ajax calls as synchronous calls). If you bring up the help pop-up you will see this setting can be overridden with the `SYNC_AJAX` property, which means if a test case defines the `SYNC_AJAX` property with an appropriate value (true or false in this case), the behavior for this test case as defined by this property will override the one defined in the settings window.

General Tab

The General Tab is used to set basic options.

Settings

General | Recording | Playback | Environment

Information

BUILD: 4.8.4.0 (LATEST AVAILABLE: 4.8.4.0)

MACHINE: GAURILAPTOP

OS: Microsoft Windows NT 5.1.2600 Service Pack 3

IE: 8.0.6001.18702 Firefox: 3.6

.NET: 2.0.50727.3615 JRE: 0.0

EXE: "C:\Lisa5.0GA\bin\browser\lisa_browser.exe" -m recorder -p 2641 -pid 2556

MEMORY: 86941 KB

General Behavior

☐ Disable browser popups

☐ Hide Error Dialogs

☐ Enable ActiveX / Flash / Flex (beta)

☐ Enable Applets (Needs public JRE)

☐ Capture HTTP traffic (Needs public JRE)

☐ Enable Pathfinder Integration (Needs public JRE)

☐ Enable SWT (preview)

Remote Applications Port: 0

Maximum Bandwidth: 0 KB/s

Output Directory: {{BROWSER_HOME}}\out

Scripts Directory: {{BROWSER_HOME}}\scripts

Maps Directory: {{BROWSER_HOME}}\maps

Log Settings

☒ Log Messages to Output Window

☒ Log Messages to file

☒ Debug Level

☐ Info Level

☐ Warning Level

☐ Error Level

Custom HTTP Headers (requires restart)

Name	Value

For help click on the ? icon in the top-right corner, then on a setting's label.

Save Cancel

In the **Information** box – you can see the machine information which runs LISA browser

In the **General Behavior** box – you can check/uncheck general behavior of the browser properties.

In the **Log Settings** box – you can check/uncheck log related information.

- **Disable browser popups:** acts like a popup blocker. Overrideable in a test with `DISABLE_POPUPS`.
- **Suppress Javascript Dialogs:** alert and confirm dialogs will auto-respond during playback `SUPPRESS_DIALOGS`.
- **Enable ActiveX / Flash / Flex:** Turns on support for ActiveX events. Only reason to turn it off might be faster startup time.
- **Enable Applets:** Turns on support for ActiveX events. Only reason to turn it off might be faster startup time or improved stability (the Java Plugin Interface has some known bugs in certain versions of it, notably 1.5.0_01 through 1.5.0_14, that could cause crashes).
- **Enable SWT:** Turns on support for testing non-Eclipse based SWT applications.
- **Capture HTTP traffic:** Turns on a proxy to capture all HTTP(S) traffic and stores all headers in the event requests. Only reason to turn it off might be faster startup time or already having a proxy.
- **Enable Pathfinder integration:** this causes the browser to receive and decrypt Pathfinder payloads for Pathfinder-enabled applications.
- **Remote Applications Port:** specifies the port to use the control remote applications (Swing, SWT or WinForms). The default, 0, picks the port dynamically.
- **Maximum Bandwidth:** throttles request and response speed to simulate a network with the specified throughput. 0 means no limit.

- **Log messages to Output window:** self-explanatory.
- **Log messages to Output file:** self-explanatory. The log files go in the same directory as the DOM browser.
- **Log Level:** the level of log statements required to be logged in the Output window or file if turned on above.

Recording Tab

The Recording Tab is used to set Recording options.

Settings

General **Recording** Playback Environment

Recording Strategy

All

Ignore ids and names whose value matches:

☐ Externalize recorded text

DOM

Ignore frame names whose value matches:

Ignore text whose value matches:

Use the following attributes (in this order):

1) <input type="text" value="id"/>	5) <input type="text" value="<none>"/>
2) <input type="text" value="name"/>	6) <input type="text" value="<none>"/>
3) <input type="text" value="<none>"/>	7) <input type="text" value="<none>"/>
4) <input type="text" value="<none>"/>	8) <input type="text" value="index"/>

Or use the following locator javascript function
☐

☒ Ignore invisible elements

Java

Record using:

☐ Component names ☒ Deep paths

☒ Component text ☐ Geometry

Recording Options

☒ Use context menus for filters and assertions

☐ Write Traffic to Disk

☒ Compress recording files

Capture Level

☐ Capture DOM Level 2

☐ Verbose HTML recording

☐ Ignore HTML responses

☐ Capture HTML changes

☐ Capture Applet snapshots

☐ Capture ActiveX snapshots

Capture Diff Size Bytes

Capture Max Time ms

Capture DOM Events

<input checked="" type="checkbox"/> navigate	<input checked="" type="checkbox"/> docload
<input checked="" type="checkbox"/> focus	<input checked="" type="checkbox"/> dblclick
<input checked="" type="checkbox"/> mousedown	<input checked="" type="checkbox"/> change
<input checked="" type="checkbox"/> mouseup	<input checked="" type="checkbox"/> contextmenu
<input checked="" type="checkbox"/> mouseover	<input checked="" type="checkbox"/> drag/drop
<input type="checkbox"/> mouseout	<input checked="" type="checkbox"/> mousemove
<input checked="" type="checkbox"/> click	<input checked="" type="checkbox"/> keypress
<input checked="" type="checkbox"/> open/close	<input type="checkbox"/> ajax callback

For help click on the ? icon in the top-right corner, then on a setting's label.

Save Cancel

- **Use context menus for filters and assertions:** Override the right-click menu in web pages to popup a custom menu that offers choices about filters, assertions or debugging. Turn it off if your site already uses custom context menus.
- **Capture applet snapshots:** stores in the test the applet screenshots used in applet test editing (browse mode). Turn it off if the tests get too large.
- **Compress recording files:** keep that turned on (used for debugging).
- **Verbose recording:** Records events for which we could not detect an event handler (possibly DOM Level 2). Turn it off for most sites, try to turn it on if you notice some necessary event does not get recorded (happens using ExtJS for instance).
- **Capture HTML changes:** Store the HTML at any click or change event to make it easier for later editing.
- **Capture Diff size:** changes over this size will store the whole response, changes under this size will store the diff.

- **Capture Max time:** the diff above won't be captured if it takes more than this amount of time.
- **Save Dialog Triggers:** for pages with a non text/plain mime type, this decides whether to pop up a "Save As" dialog instead of performing a navigation if the target url matches the regular expression (by default, it does this for Excel, Word, PDF, Text, PowerPoint, Executable and Zip files).
- **Recording strategy:** which HTML attributes to use and in which order when generating XPath expressions.
- **Exclude ids matching:** any element id matching this regular expression won't be used in the auto-generated XPaths.
- **Record component names/text:** use java component names or text (or not) in the XPath generated when recording Java based applications.
- **Capture DOM events:** Turn off any type of DOM event you're not interested in capturing.

Playback Tab

The Playback Tab is used to set Playback options.

The screenshot shows the 'Settings' dialog box with the 'Playback' tab selected. The dialog has four tabs: General, Recording, Playback, and Environment. The Playback tab contains several sections: Playback Options, Timeout Settings, and Browser Options. The Playback Options section includes checkboxes for 'Supress Javascript Dialogs', 'Show Browser Dialogs', 'Synchronize Ajax calls.', 'Use Hardware Input', 'Enable mouse movement', and 'Send Responses back to LISA'. It also has input fields for 'Max missing targets' (set to 2) and 'Min matching score' (set to 1). Below these is a text field for 'Failed assertions screenshot directory:' with a browse button. A 'Playback Speed' slider is set to the fastest position. The 'Wait multiplier' is set to 0x. The Timeout Settings section has input fields for 'DOM Load Timeout' (10000 ms), 'DOM Lookup Timeout' (1000 ms), 'Applet Load Timeout' (10000 ms), 'Applet Lookup Timeout' (5000 ms), 'ActiveX Load Timeout' (10000 ms), and 'ActiveX Lookup Timeout' (5000 ms). The Browser Options section has a list of browsers to use by default: Internet Explorer (checked), Firefox, and Safari. It also has a section for 'Installation Directories' with dropdown menus and browse buttons for Internet Explorer and Mozilla Firefox.

Settings

General Recording **Playback** Environment

Playback Options

☐ Supress Javascript Dialogs

☐ Show Browser Dialogs

☐ Synchronize Ajax calls.

☐ Use Hardware Input

☐ Enable mouse movement

☐ Send Responses back to LISA

Max missing targets: 2

Min matching score: 1

Failed assertions screenshot directory:

Playback Speed (left is slowest - right is fastest)

Wait multiplier: 0x

Timeout Settings

DOM Load Timeout: 10000 ms

DOM Lookup Timeout: 1000 ms

Applet Load Timeout: 10000 ms

Applet Lookup Timeout: 5000 ms

ActiveX Load Timeout: 10000 ms

ActiveX Lookup Timeout: 5000 ms

Browser Options

Use the following browsers by default:

☒ Internet Explorer

☐ Firefox

☐ Safari

Installation Directories:

C:\Program Files\Internet Explor... ..

C:\Program Files\Mozilla Firefox

For help click on the ? icon in the top-right corner, then on a setting's label.

Save Cancel

Of particular interest on this tab is the **Synchronize Ajax calls** checkbox. This if checked, the step will not return until the Ajax call completes.

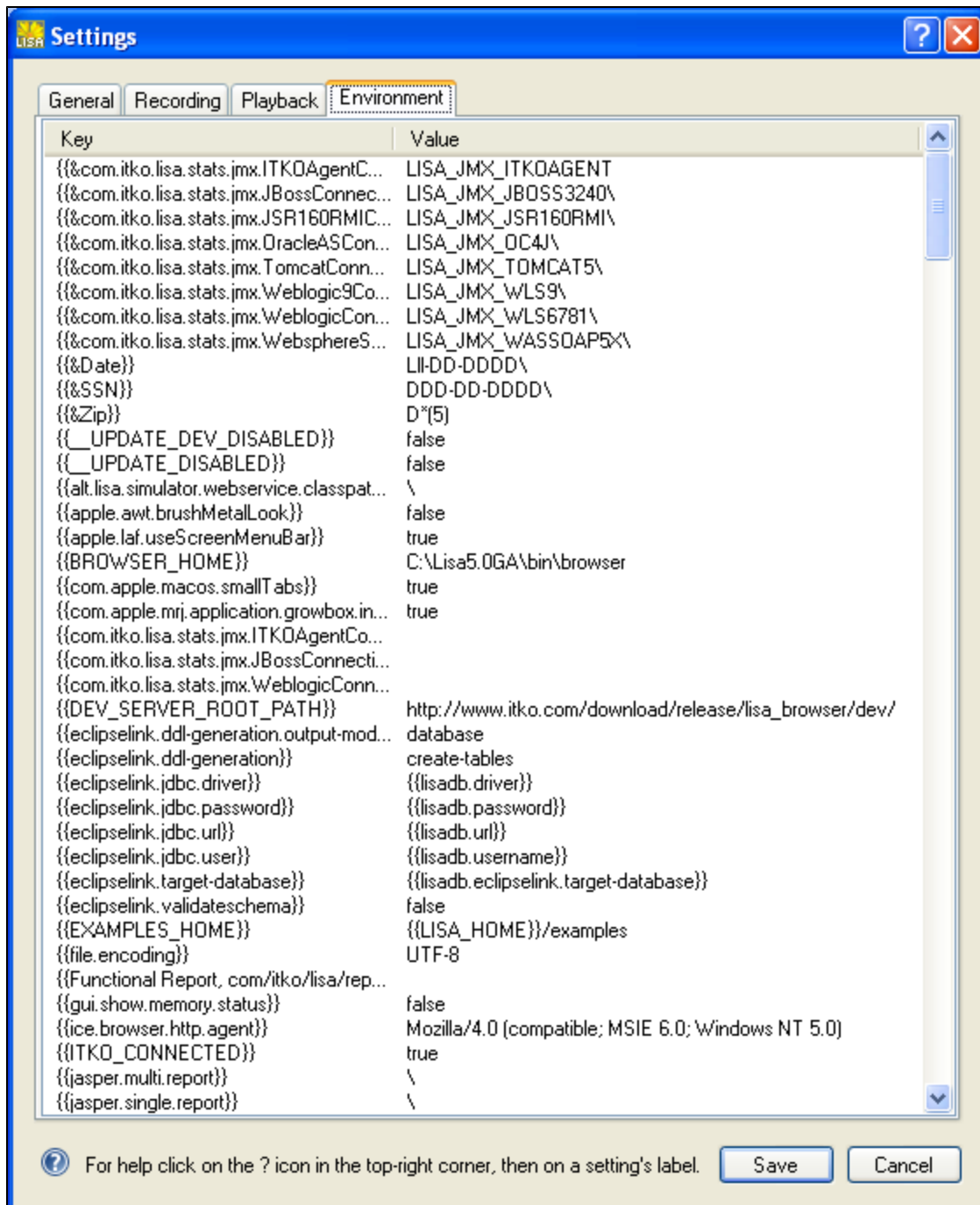
- **Fail step on missing target:** Generates a failure instead of the default warning when a target can not be found for a step (that includes

browser window, frame, element). Overrideable in a test with `FAIL_MISSING`.

- **Synchronize Ajax Calls:** forces all ajax calls to be executed synchronously. Overrideable in a test with `SYNC_AJAX`.
- **Use Hardware Input:** Replays tests by controlling keyboard and mouse. Overrideable in a test with `USE_HARDWARE`.
- **Send Responses back to LISA:** By default responses will not be sent back to LISA to improve performance and memory since it's usually not necessary.
- **Min Matching Score:** How many differences are allowed between a recorded XPath value and the best match found during playback. Overrideable in a test with `MIN_BACKTRACKING`.
- **Default Browser Mode:** What browsers to enable by default during playback (pipe-delimited combination of IE, FF and WK). Overrideable in a test with `DEFAULT_BROWSER`.
- **Playback Speed:** The default speed to use to replay test as a multiplier of the recorded speed. 0x is usually the best choice. Overrideable in a test with `PLAYBACK_SPEED` (integer between and 10).
- **XXX Load Timeout:** The maximum amount of time waited before a new page/applet/control load before proceeding. Overrideable in a test with `XXX_LOAD_TIMEOUT`.
- **XXX Lookup Timeout:** The maximum amount of time waited to find an element on a page/applet/control before proceeding. Overrideable in a test with `XXX_LOOKUP_TIMEOUT`.

Environment Tab

The Environment Tab is used to display the Environment settings.



This tab shows the list of global environment variables available to the browser, as read from **lisa.properties** and **local.properties**.

LISA Driver Settings:

In addition to the settings above that can be overridden from LISA in a test case or in the **local.properties**, the following are available:

- **lisa.browser.launch.timeout**: The amount of time allowed for a browser to launch (default is 10,000). Specify in **local.properties**.
- **lisa.browser.exec.timeout**: The amount of time allowed for a step to execute (default is 300,000). Specify in **local.properties**.
- **lisa.browser.max.instances**: The maximum number of browser instances per machine (default is 25). Specify in **local.properties**.
- **lisa.browser.client.user.single**: Whether to run staged browsers using the same user account as the currently logged-in users (default is true). Specify in **local.properties**.
- **lisa.browser.base.port**: The first port to use in the range of ports available to control web browsers (default is 0 for dynamic value). This normally does not need to be modified except in very secure environments that lock down some local ports. Specify in **local.properties**.
- **lisa.browser.client.user.<user name>=<encrypted password>**: when **lisa.browser.client.user.single** is set to false, browser instances will use the specified windows user accounts to run tests. If not enough user accounts are specified in this manner and the currently logged-in user has admin privileges, user accounts will be dynamically created to run the tests (and deleted at the end). To obtain an encrypted password, run the command line: `lisa_browser.exe -m encrypt -in <clear text>`.

- **lisa.browser.share.subprocess.state**: Whether to run a sub-process using the same browser instance as parent test (default is false). Specify in configuration of sub-process.
- **lisa.browser.swing.port**: The first port to use in the range of ports available to control Swing applications (default is 0 for dynamic value). Specify in local.properties.
- **lisa.browser.base.port**: The first port to use in the range of ports available to control web browsers (default is 0 for dynamic value). Specify in local.properties.

1.3.4 Browser & Extension Updates

1.3.4 Downloading Browser and Extension Updates

Because the web 2.0 environment evolves so fast, there is the ability to update it directly without waiting for a whole LISA release. Note that these intermediate releases are considered **unsupported**, but since most customer environments are not available to the outside world, we can not test a bug fix or feature enhancement against them so we work through this mechanism to allow a fast turnaround cycle. When the changes have been approved by a customer they are submitted to the official build environment for full testing.

One way to obtain the update is to run the update command from the command line:

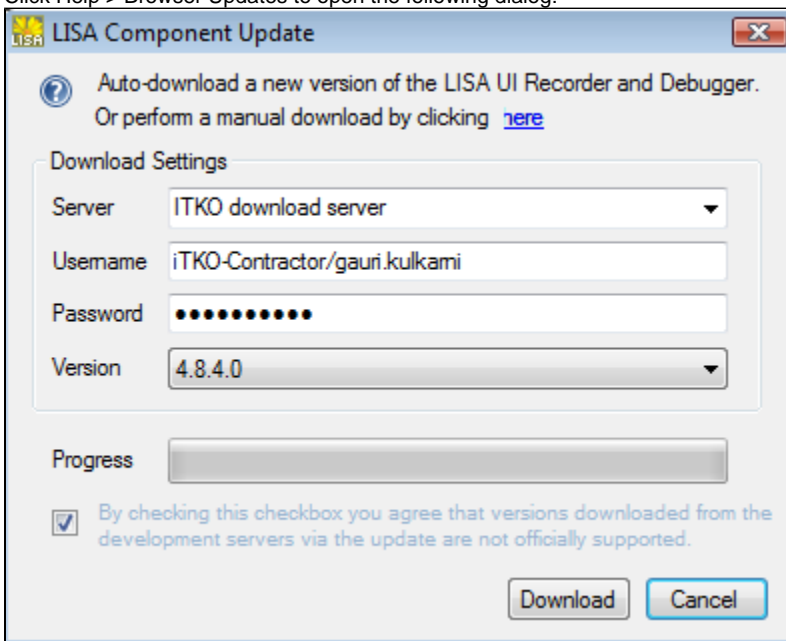
```
lisa_browser.exe -m update.
```

Or you can do it from the GUI by going to the Help menu and selecting the Update menu.

To download the updates of the web 2.0 browser,

Open the Web 2.0 browser within LISA.

- Click Help > Browser Updates to open the following dialog:

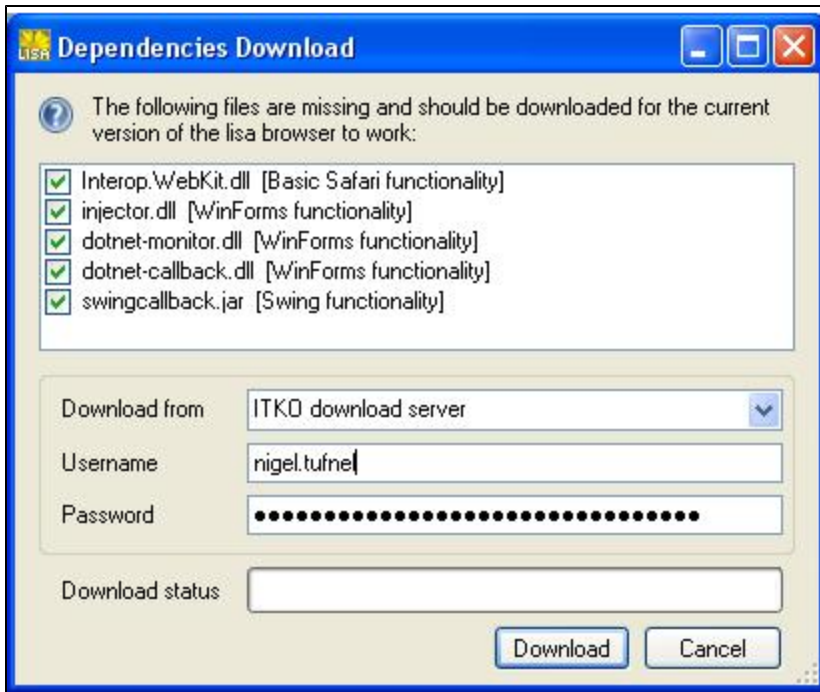


Click Download to start the download. You can monitor the download progress through the progress bar.

After the download is completed you will be notified that the update will be effective on restart.

Finally, when major new functionality is added, some dependencies will be added or modified that won't be available until you install from a full LISA installer.

This is why the browser checks for those on start up and prompts you for a download on start up if they are missing:



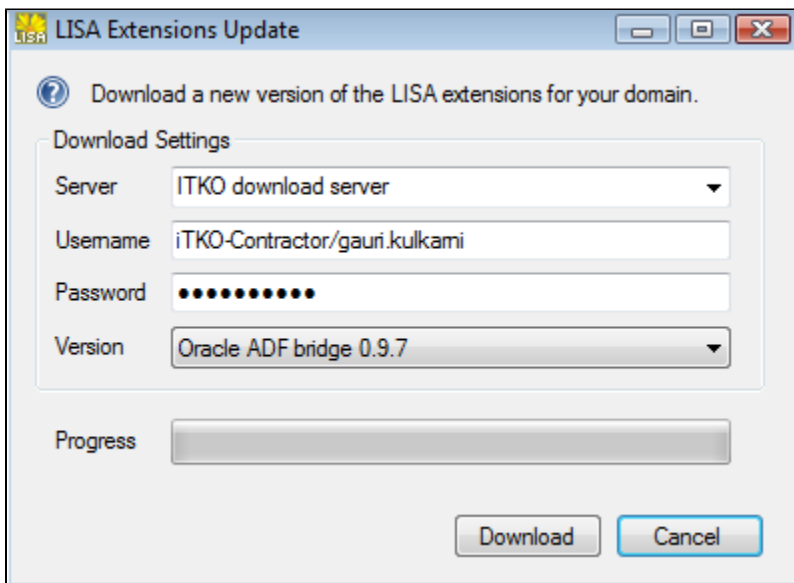
The browser won't start until those missing dependencies have been downloaded from one of the available locations: one of the release servers or one of the development servers. If none of these are accessible from the machine (no internet access for example), the listed files will have to be downloaded manually (see [LISA Web 2.0 Update Repository](#)).

Extension Updates

To download the updates of the Extensions,

Open the Web 2.0 browser within LISA.

- Click Help > Extensions Updates to open the following dialog:

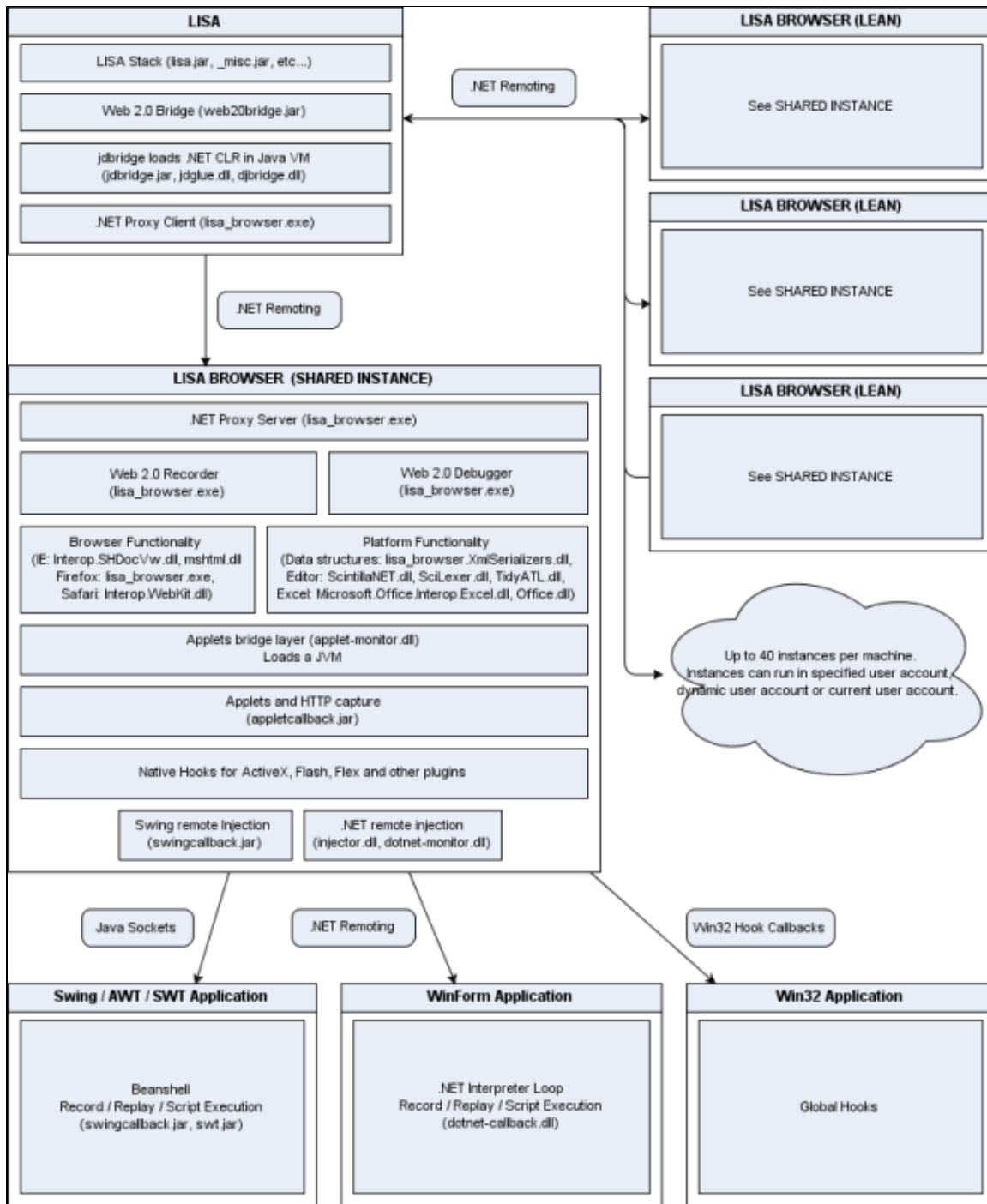


- Click **Download** to start the download.

1.3.5 Browser Architecture

1.3.5 Web 2.0 Browser Architecture

The Web 2.0 architecture in 4.6 and newer versions is as below:



Notes:

Q: Why is the browser hosted in .NET and native code rather than in TestManager (java)?

A: Hosting browsers in java correctly is difficult. There are a lot of products, open-source (jdlic, jrex, etc...) and commercial that claim to do this but our experience is that they are quite prone to crashing or freezing. Using a native environment (mostly) eliminates these problems.

In addition, even if this approach was more stable, we could not run java applets. The reason is that a JVM would host a browser, which would then try to launch another JVM in the same process to run the applets. Since only one JVM per process is currently allowed, this would cause a crash.

Q: Why is the communication mechanism between LISA and the browser .NET Remoting?

A: Given that we have a java process and a .NET process that need to communicate, we had a few options:

- In-process is not doable for the applets reason mentioned above, as well as the required ability to run browsers under different user

- accounts.
- A proprietary protocol over raw sockets was too much work, especially since the communication must be bidirectional.
- Web Services over HTTP is the approach used in 4.5 and earlier but it is slow, verbose and exhibits bugs in the Web Service stacks of those platforms that undermine reliability.
- The most elegant approaches are to either load a JVM in the .NET processes and use RMI, or load a .NET CLR in the JVM and use .NET remoting. The first approach suffers from the applets limitation outlined above, which left us with the second one.

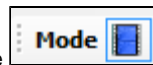
In 4.5 and earlier:

Will be completed upon request...

2. Recording Mode

2. Recording Mode

Within the Web 2.0 browser, you can record many type of applications like a simple web application, java application, swing application, .net application to name a few. In the recording mode, you can test any website and record all the events in the browser. These recorded events can later be replayed in the Playback mode.



When you open the LISA Browser, by default it starts in the Recording mode

The LISA Browser has a **toolbar** as shown below:



On the **left** of the toolbar you can use the **Mode** button to select the Mode of operation within the browser:

- **Recording Mode** - Where you can record the operation.
- **Edit Mode** - Where you can edit the transactions.
- **Playback Mode** - Where you can playback the recorded operation.

You can Record the web page in the Recording mode and playback the recording by clicking on the Playback mode.

You can view add/delete the Logical, Physical events and view the Object details in the Edit mode, which is described later.

To start Recording, select the appropriate option of recording from the Address bar drop down.

You can choose from -



Enter the website address in the Address bar.

You can navigate the website for testing and in the background it will record. Once done, click Save to save the recording.

This will save the recording and exit the web browser.

To playback or edit the events, open the web browser again. By default it will now open in the **Edit mode**, with the last saved recording.

For additional information see the topics below.

- [2.1 Recording Example](#)
- [2.2 Recording a Swing Test](#)
- [2.3 Recording an Applet Test](#)
- [2.4 Different Views of a Web Page](#)

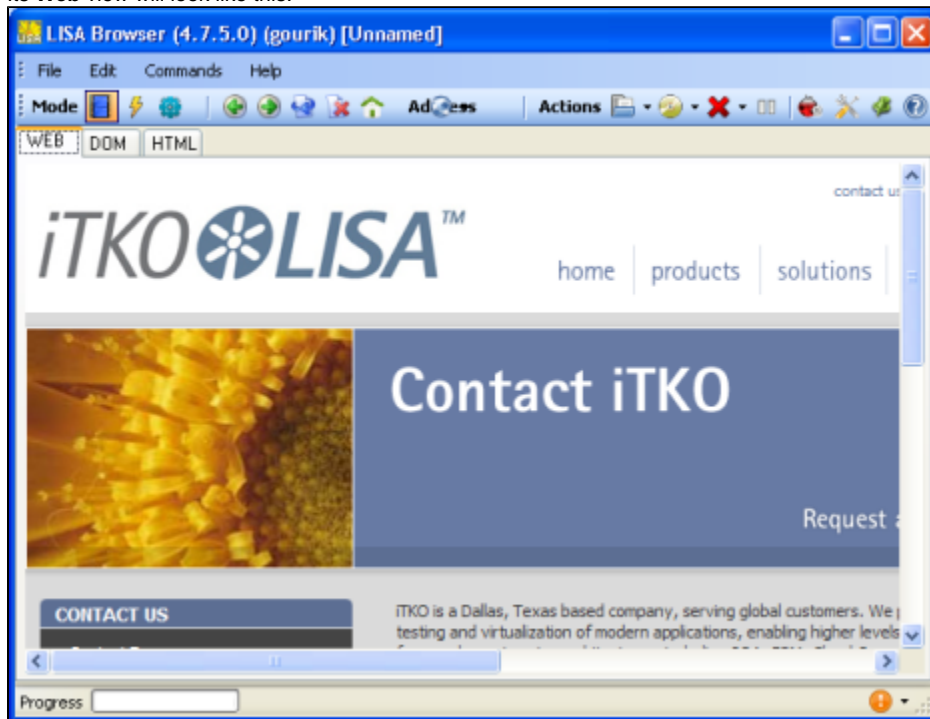
2.1 Recording Example

2.1 Recording Example

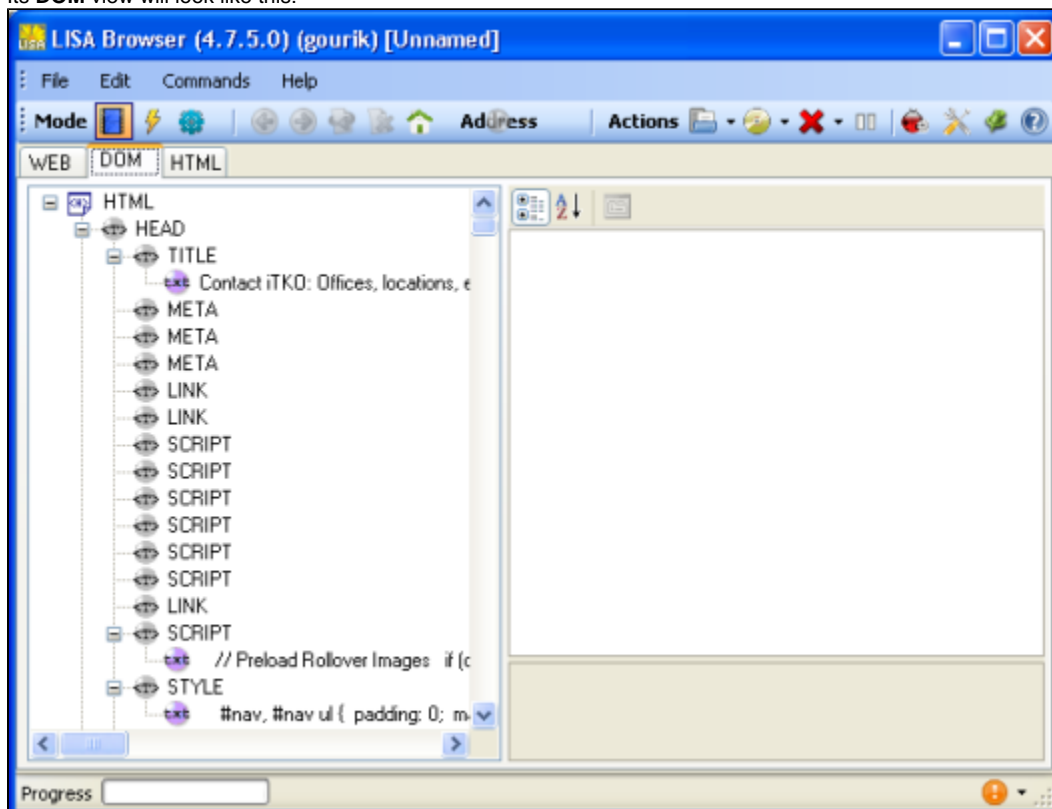
To start the recording,

- Type in the some web address, say: <http://www.itko.com/>
- Browse through the web pages, so that they get recorded.

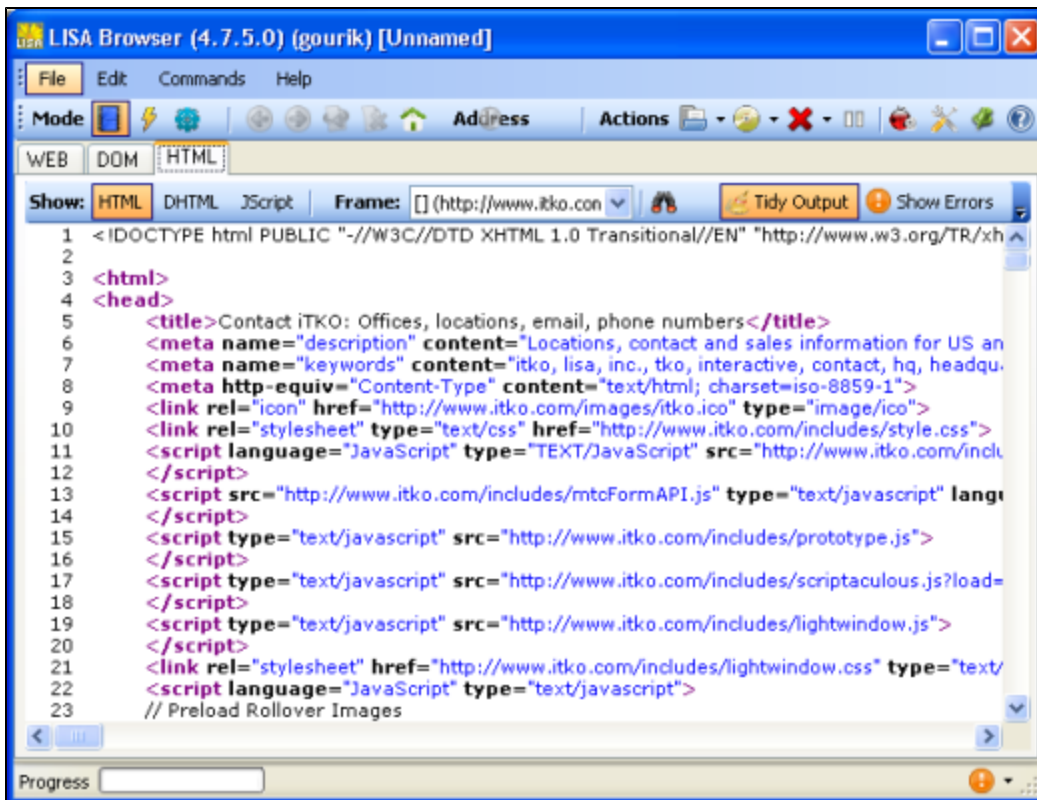
Its **Web** view will look like this:



Its **DOM** view will look like this:



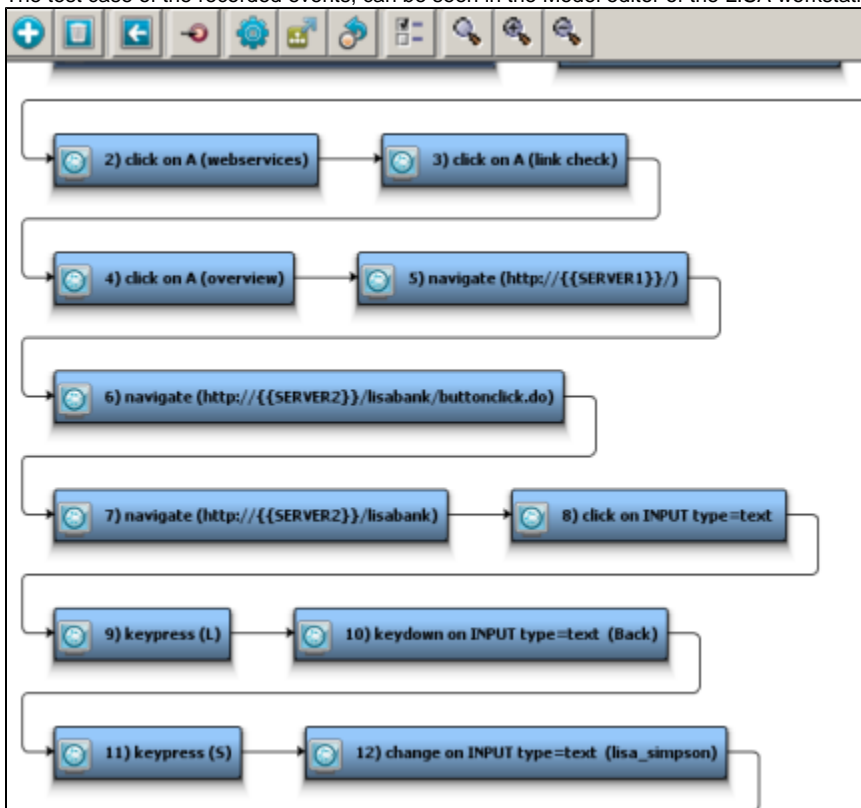
Its **HTML** view will look like this:



Once you are satisfied with the recording,

- Click the **Save** button to save and close the LISA browser.

The test case of the recorded events, can be seen in the Model editor of the LISA workstation as shown below:



The execution of this test can be triggered in the normal LISA ways, through:

- Staging a test or
- Using Interactive Test Run (ITR).

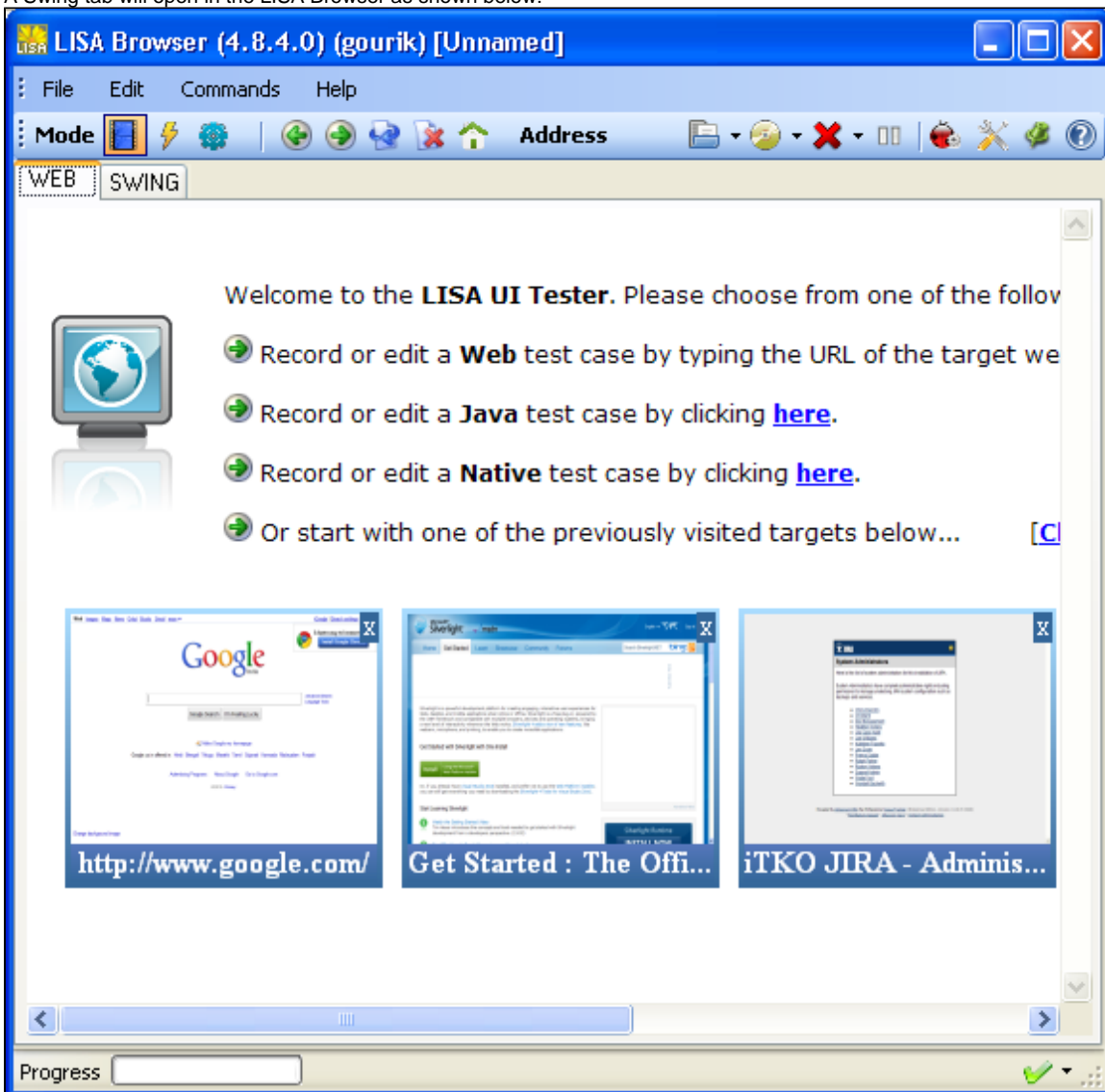
Reopen the LISA browser to view the recording in the Playback mode.

2.2 Recording a Swing Test

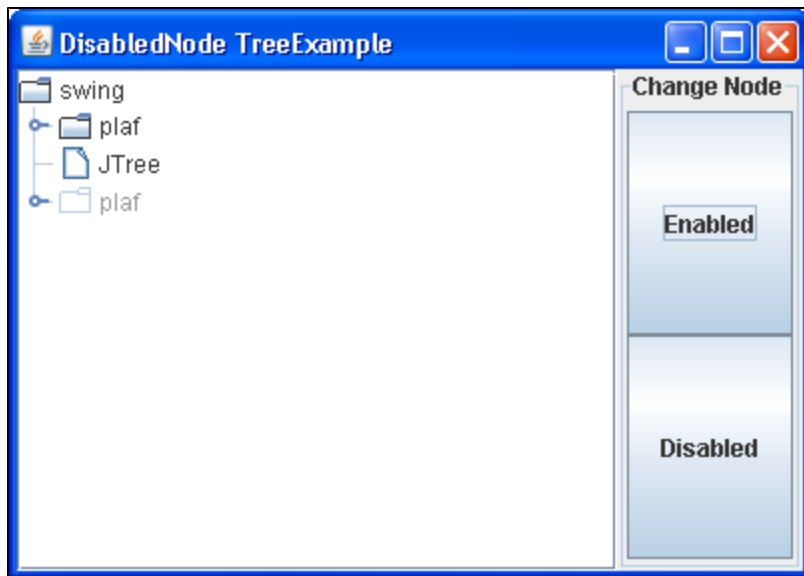
2.2 Recording a Swing Test case

To record a Swing test, you need to have a swing application.

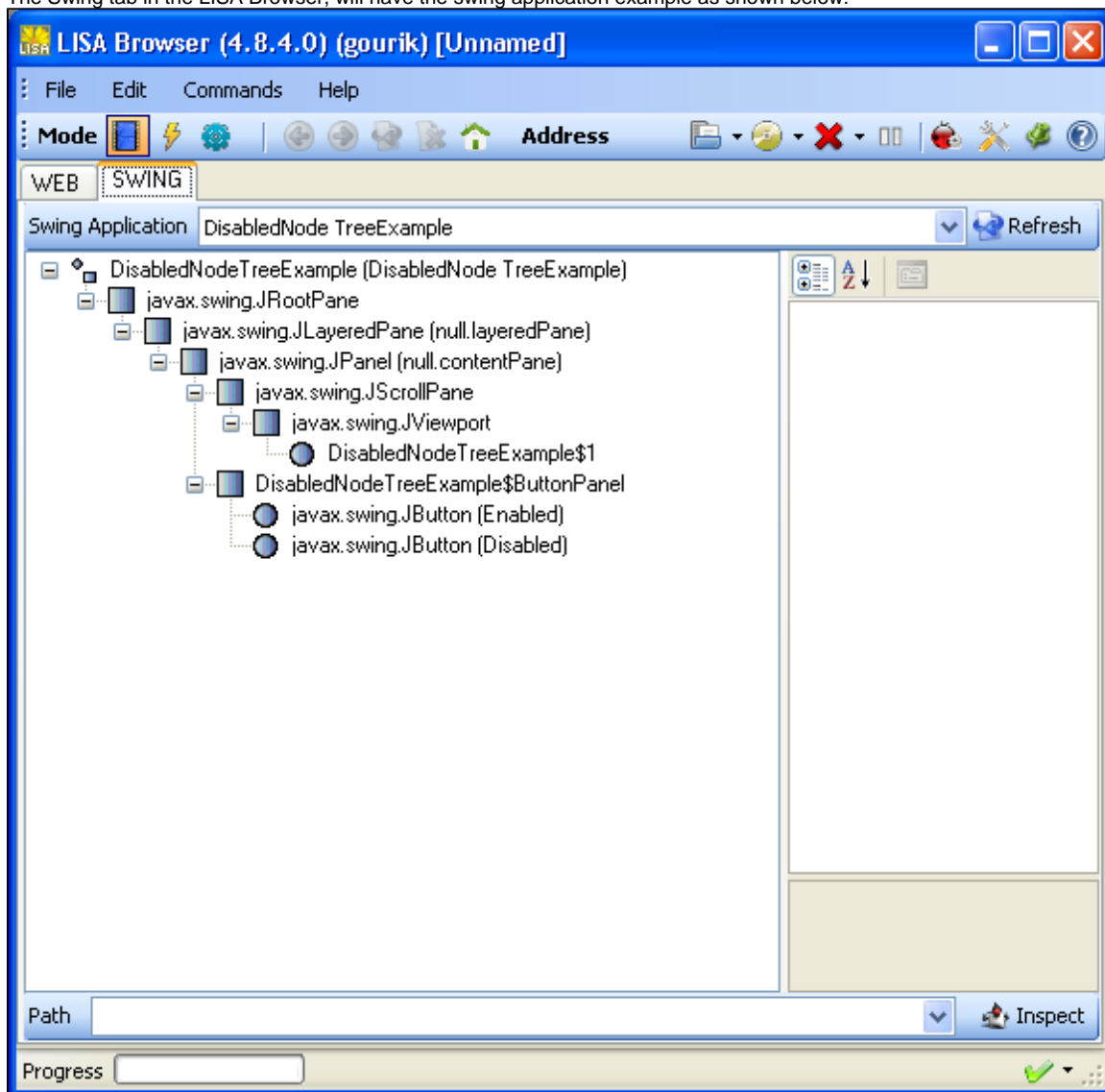
- Open the LISA Browser and click on "**Record or Edit a Java test case by clicking here**"
- Click on "here" and **browse** to the path of the swing application.
- Open the Swing application and browse for recording.
- A Swing tab will open in the LISA Browser as shown below:



A Swing example window will open as below:



The Swing tab in the LISA Browser, will have the swing application example as shown below:



2.3 Recording an Applet Test

2.3 Recording an Applet Test case

To record an Applet test, you need to have a web page which uses an Applet.

You also need to check the Applet related options in the Settings dialog box.

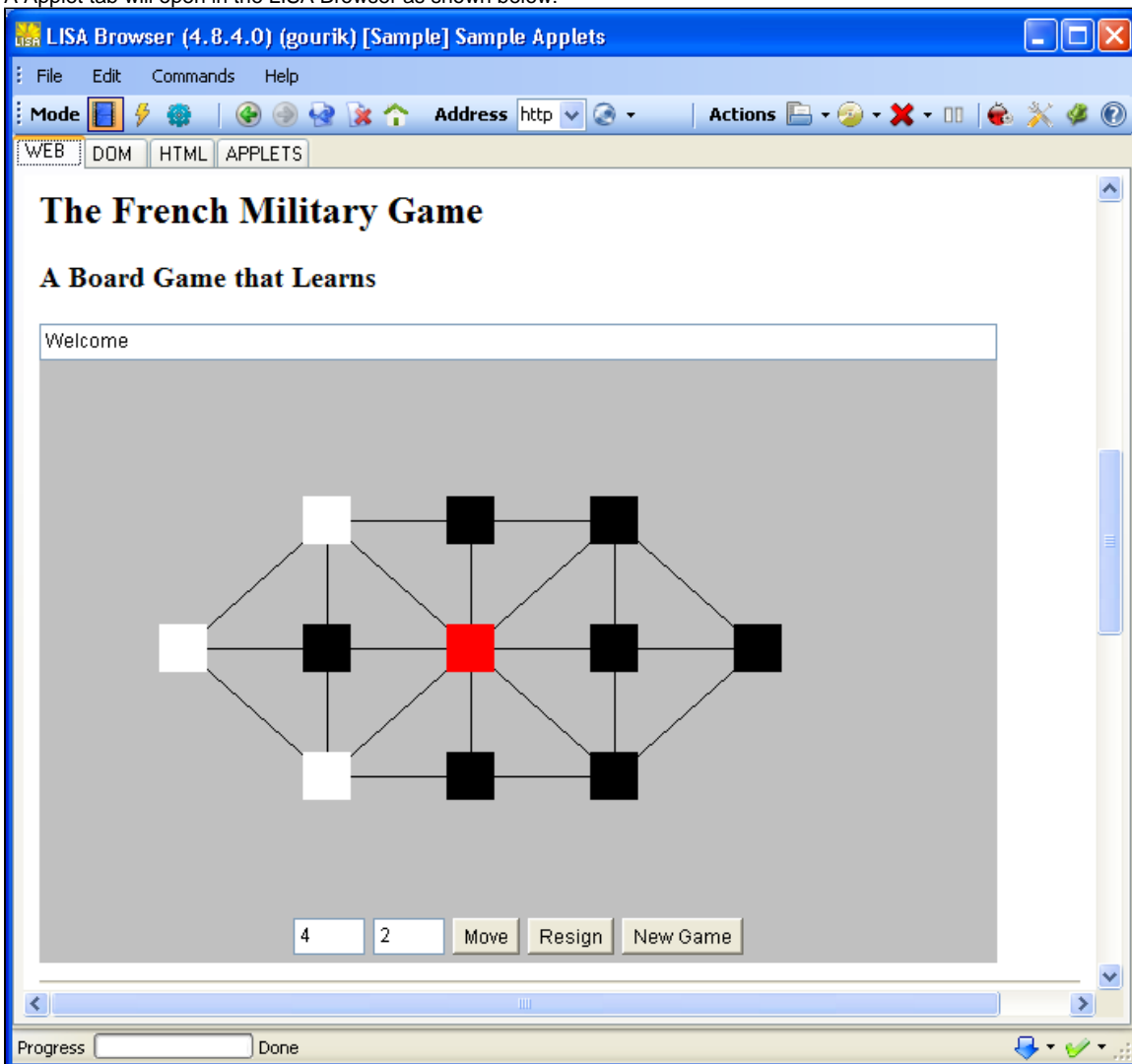
☒ Enable Applets

- In the General Settings tab

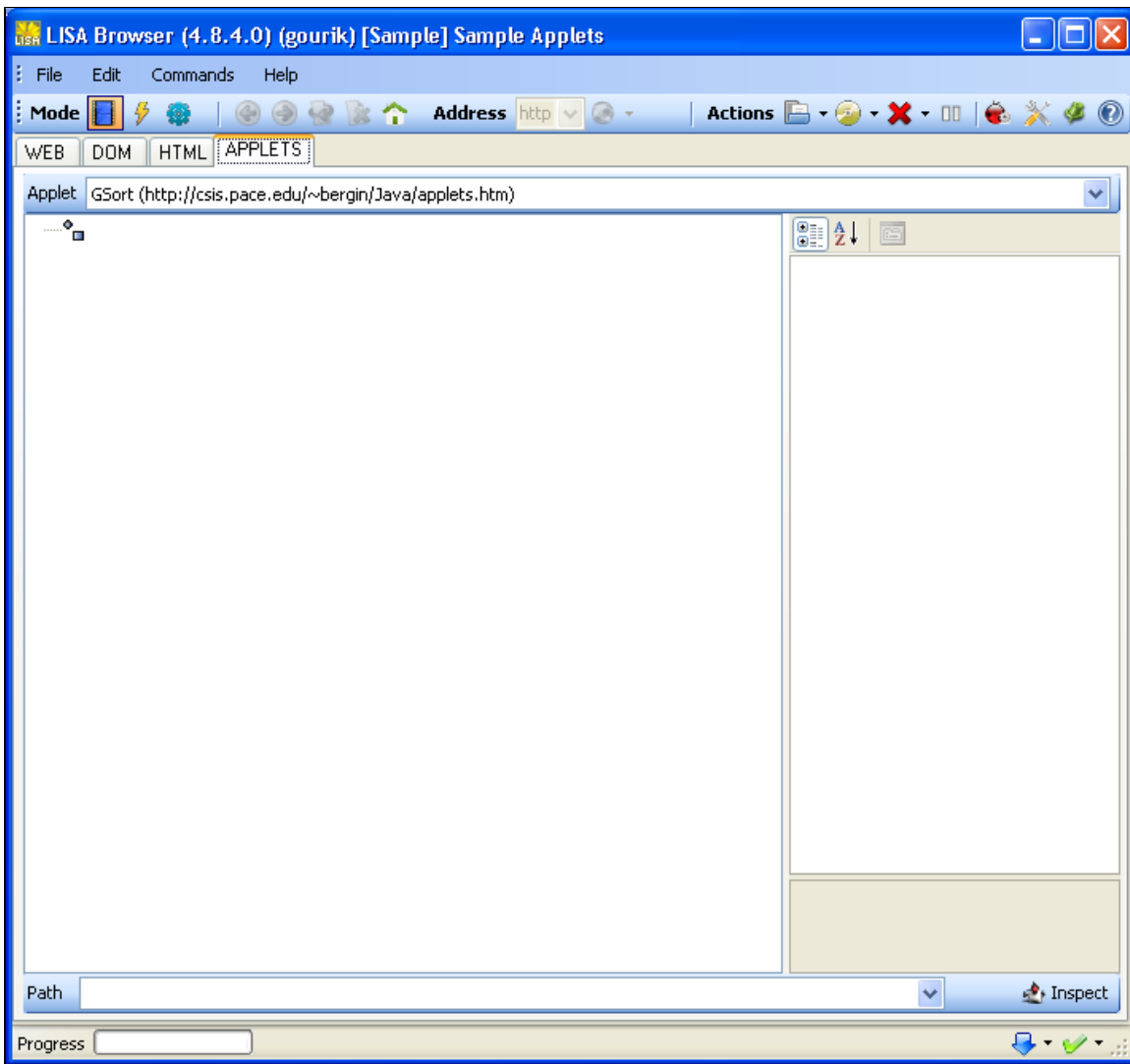
☒ Capture Applet snapshots

- In the Recordings Settings tab

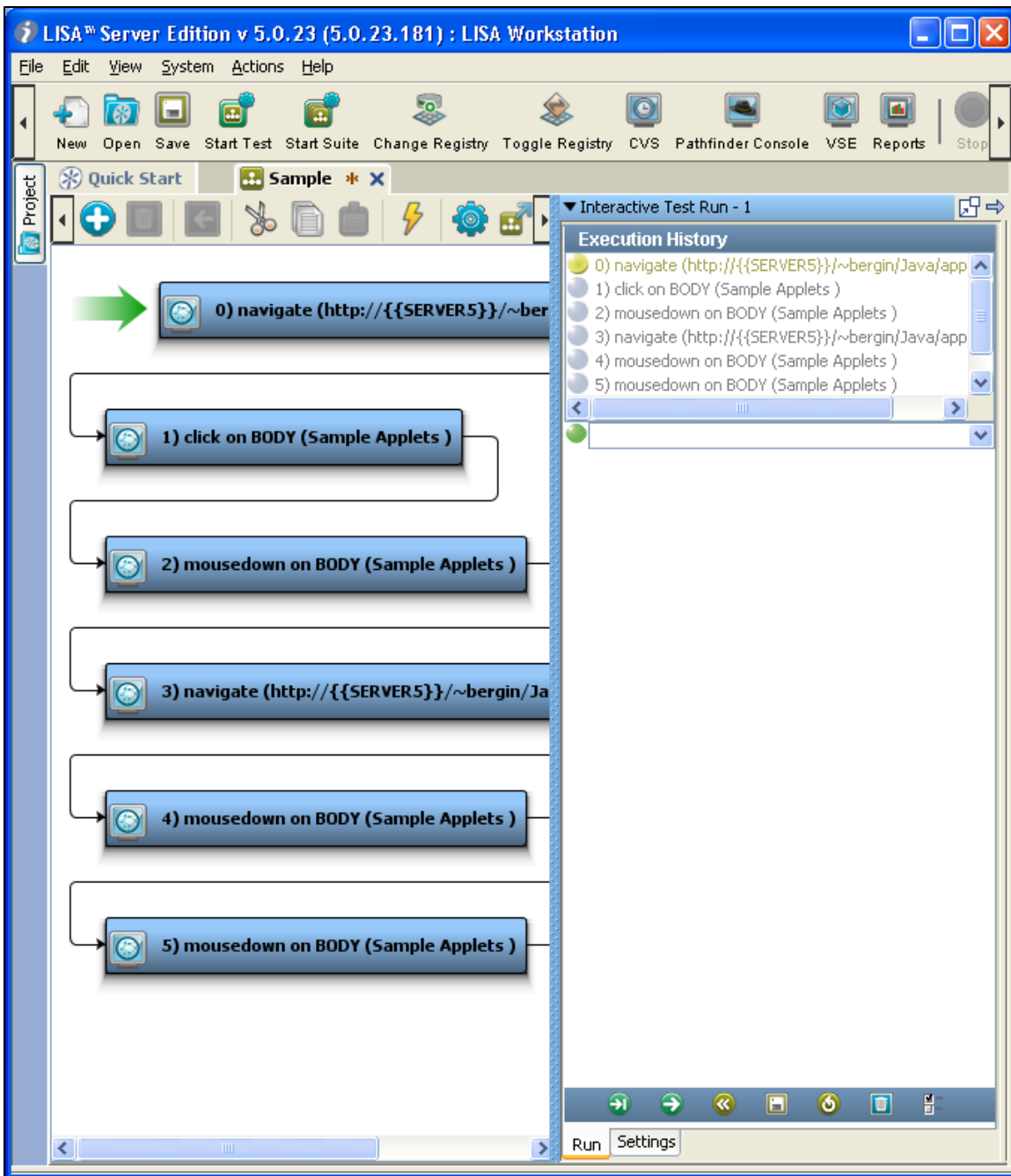
- Open the LISA Browser and click on "**Record or Edit a Java test case by clicking here**"
- Click on "here" and **browse** to the path of the web page which uses an Java Applet.
Example - <http://csis.pace.edu/~bergin/Java/applets.htm>
- Open the web page and start recording.
- A Applet tab will open in the LISA Browser as shown below:



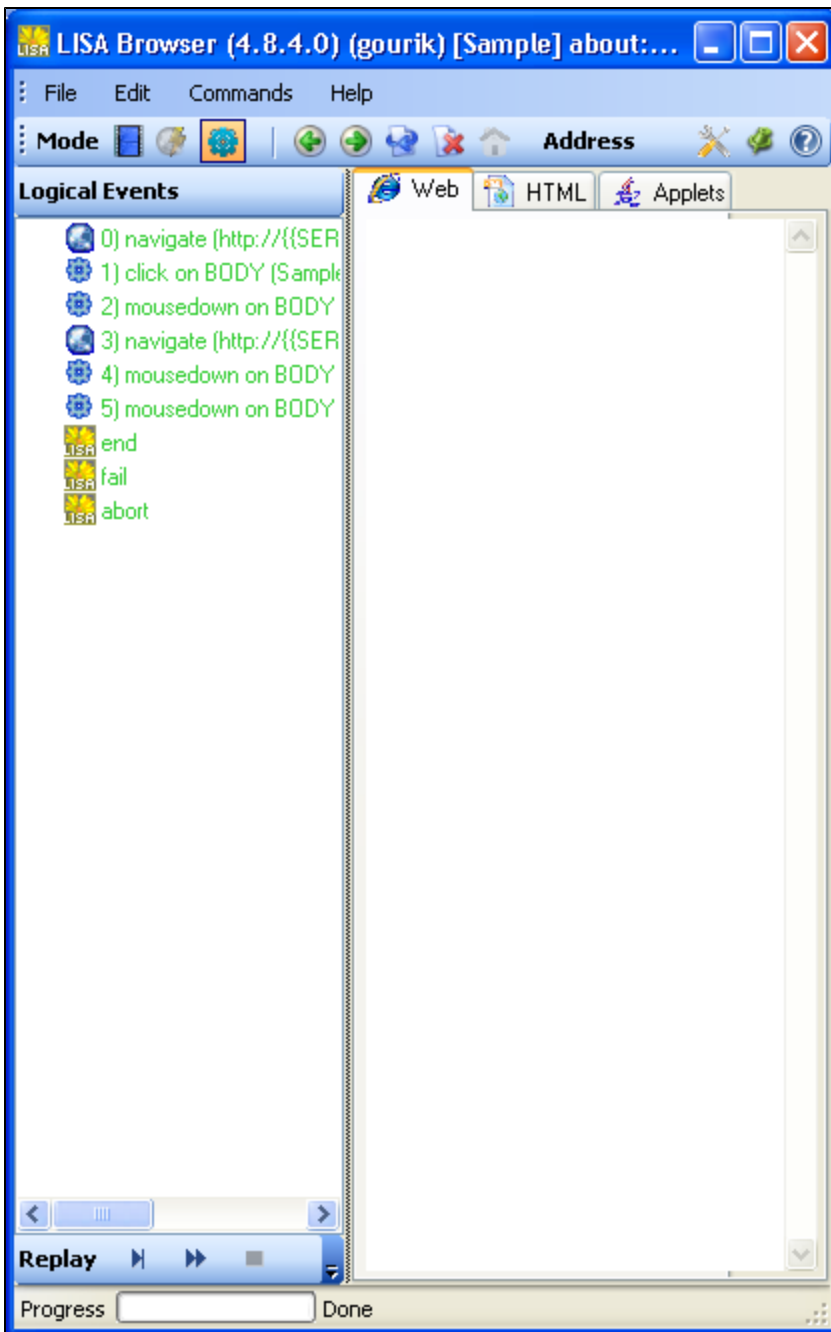
- Click on the Applet tab -



- Click Save to save the Recording. The browser closes upon saving.
- Open the LISA Workstation, where the recording appears as a test case and run it in ITR.



- When you play it in ITR, it will open the LISA Browser in the Playback Mode as shown below:



You can also see the Applets on the LISA Website.



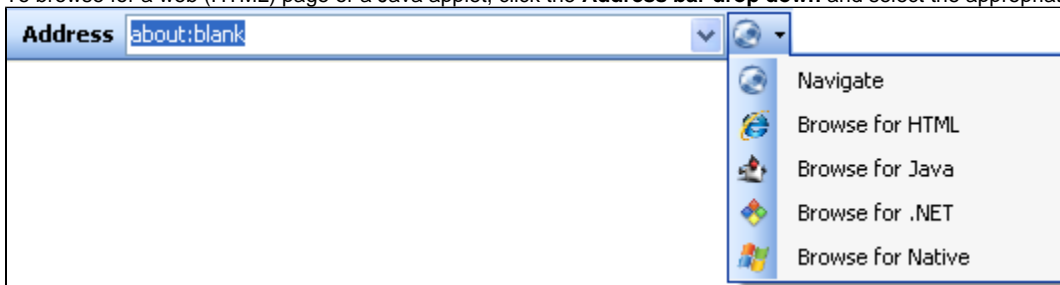
2.4 Different Views of a Web Page

2.4 Different Views of Recordings

When you record a test, it is sometimes desirable to understand how the HTML behind the page is organized. In particular, when you define Filters and Assertions, you sometimes need to know something about an html attribute on the page, or some text that might be hidden, etc.

To facilitate this, the LISA Browser is capable to show different views of pages within the browser window.

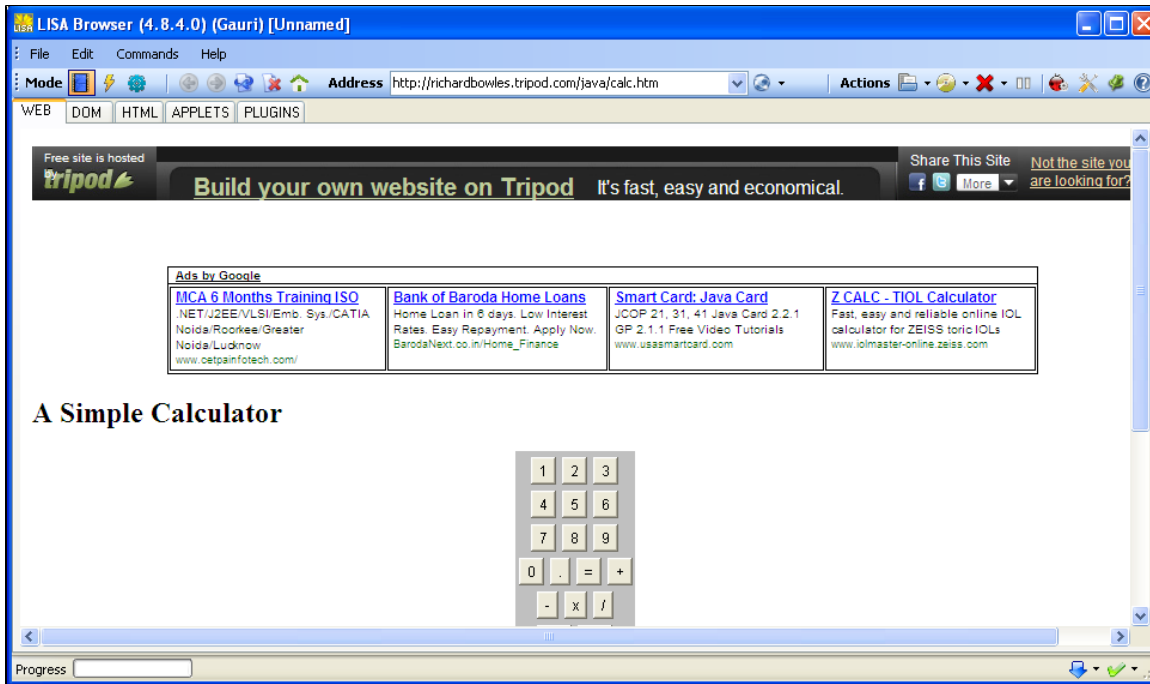
To browse for a web (HTML) page or a Java applet, click the **Address bar drop down** and select the appropriate link as shown below.



Web View

For the purpose of illustration, we have taken the example of a online calculator as shown below:

This web page is using an applet and a plugin, hence these two tabs are seen.

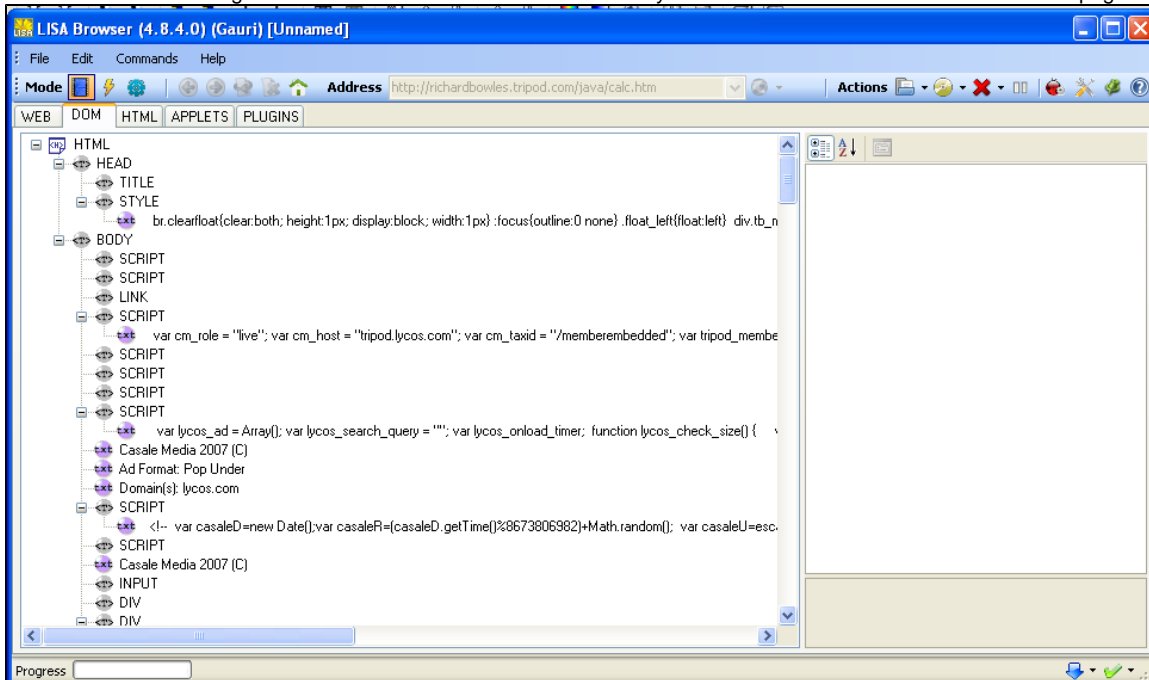


DOM View

The DOM view is a hierarchical view of the web page HTML document currently rendered in the browser. When you select an element in the tree, you can see all its attributes (name and value) displayed in the right hand-side grid.

Note that when you have a page made of frames, each frame node has its entire document available as child node so you can examine the whole hierarchy at once.

The **DOM view** is the regular view of a browser as seen below: Here you can see the DOM tree of the above web page.



HTML View

The **HTML view** is the textual source of the rendered page.

In addition, there are a few tabs at the top **Show** column, where you select the display.

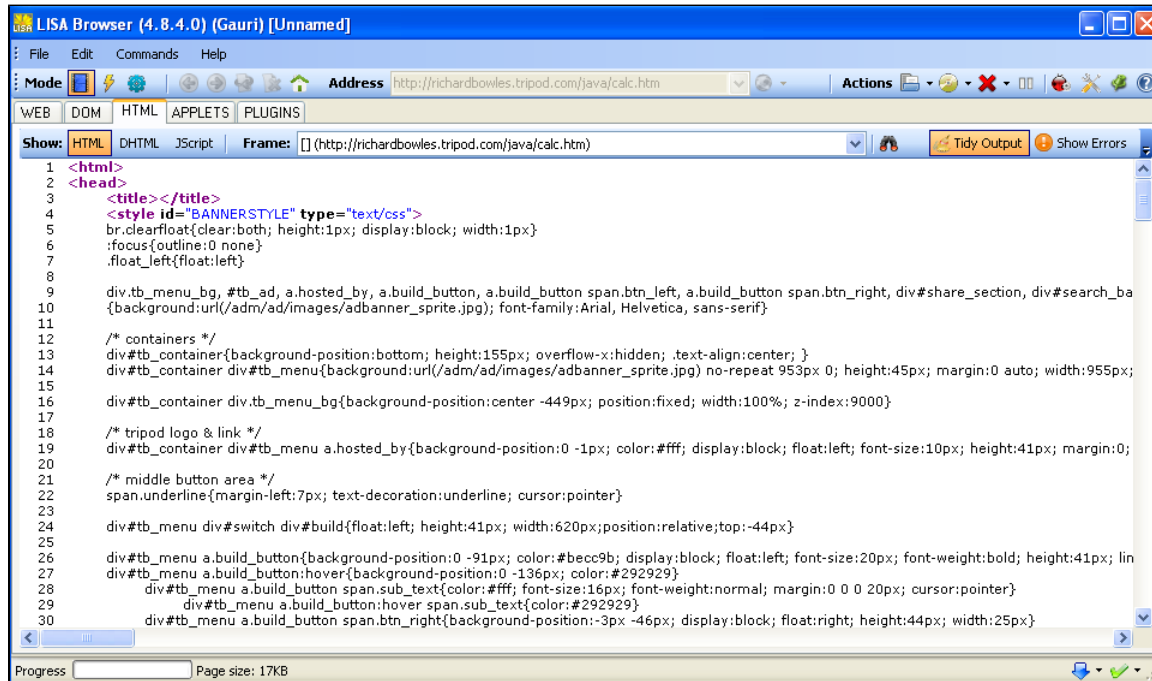
The Frame combo box lets you select which frame to display the source of in case of multiple frames, and the HTML / DHTML / JScript buttons

lets you select respectively whether you're seeing the HTML that comes from the server (static HTML), or the HTML at it is currently being seen by the client (Dynamic HTML), or the Javascript files and snippets used by the current page and frame.

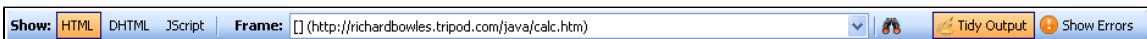
This is especially useful when pages make use of javascript or ajax that dynamically modify the html in the browser without reloading a whole page from the server.

Select **Browse for HTML** button from the Address bar drop down and enter the web page address. Here you can see the HTML view of the Java animation web page.

The page below shows the html view of the above web page.



In HTML view, there are a couple of controls at the top of the window as shown below:



- **Show HTML/DHTML/Jscript:** You can select the required script from HTML/DHTML/Jscript. The Static/Dynamic HTML group lets you select whether you're seeing the HTML that comes from the Server, or the HTML at it is currently being seen by the client. This is especially useful when pages make use of JavaScript or Ajax that dynamically modify the html in the browser without reloading a whole page from the Server.
- **Frame:** The Frame drop down box lets you select which frame to display the source of in case of multiple frames
- **Find:** The find button opens a Search dialog and **allows** you to search.
- **Tidy Output:** Shows or hides the **Tidy HTML output**
- **Show Errors:** Shows or hides the HTML errors.

Applet View

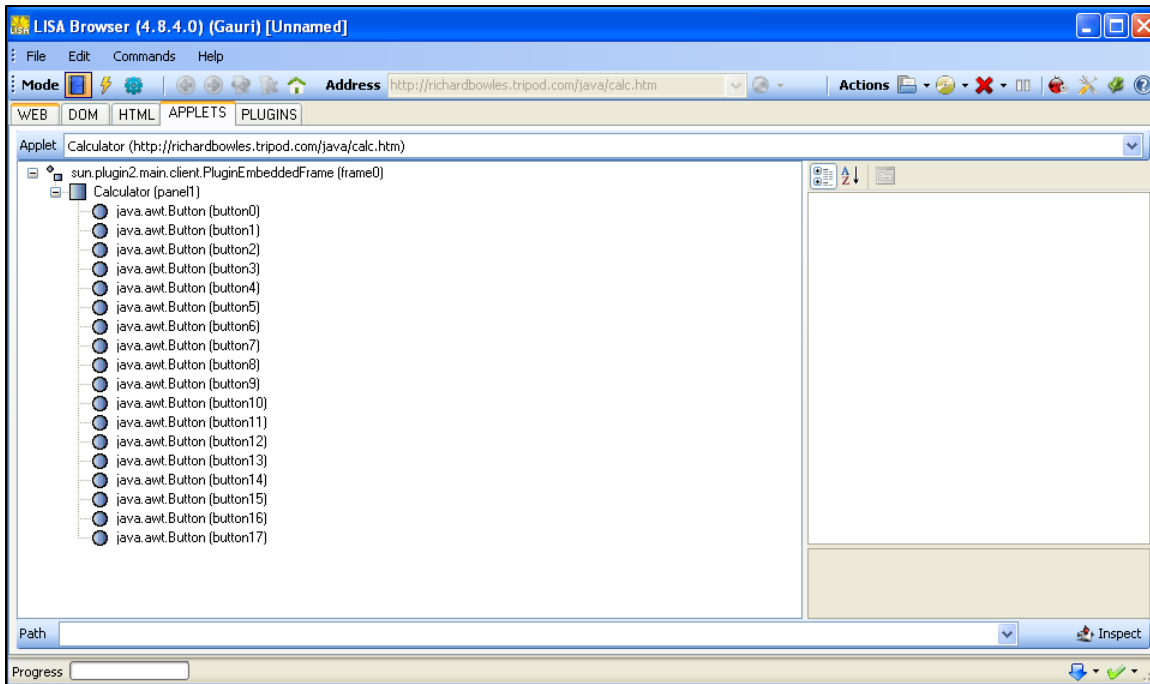
The **APPLET view** is a hierarchical view of (one of) the applet(s) currently displayed in the browser.

At the top of the panel, there is a combo box that allows you to select which applet to display the properties of in case there are several on the page.

In the tree on the left is the component hierarchy of all the UI elements that make up the applet. They are identified by class name and label or text. On the right side is the property grid that displays all the names and values of the fields of the java object that backs up the selected UI component in the tree. Those are organized by java class hierarchy (e.g. java.awt.Component properties, javax.swing.JComponent properties, etc...)

At the top of the panel, there is a drop down box that allows you to select **which applet** to display the properties of, in case there are several on the page.

Here you can see the Applet view of the Java animation web page.



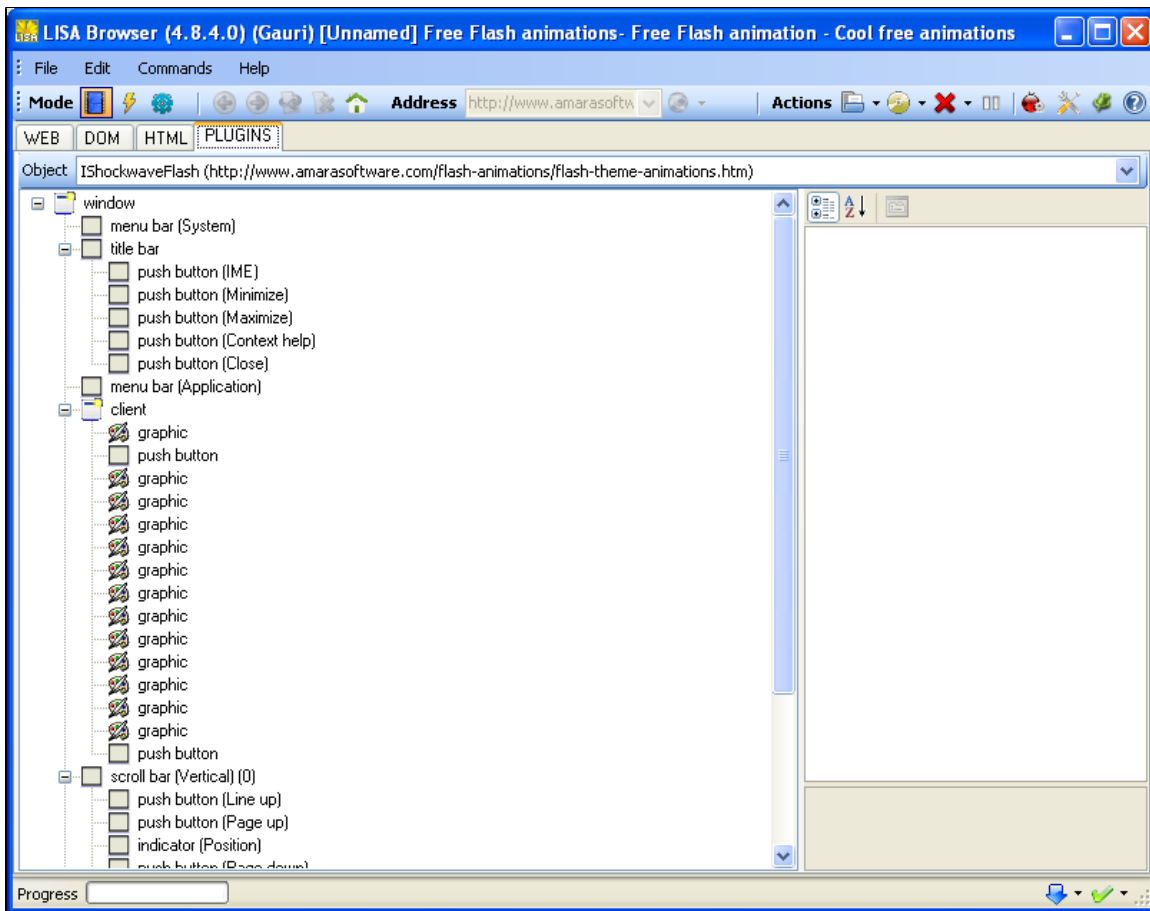
Plugins View

The **PLUGINS** view is a hierarchical view of (one of) the Active X control(s) currently displayed in the browser.

In particular those can be FLASH or FLEX controls. At the top of the panel, there is a combo box that allows you to select which object to display the properties of, in case there are several on the page. On the right side is a property grid that displays all the available information of a sub control identifiable in the object.

For the Plugins, we have taken an example of a Flash animation.

When the flash animation invokes, you can see the plugins tab as shown below:

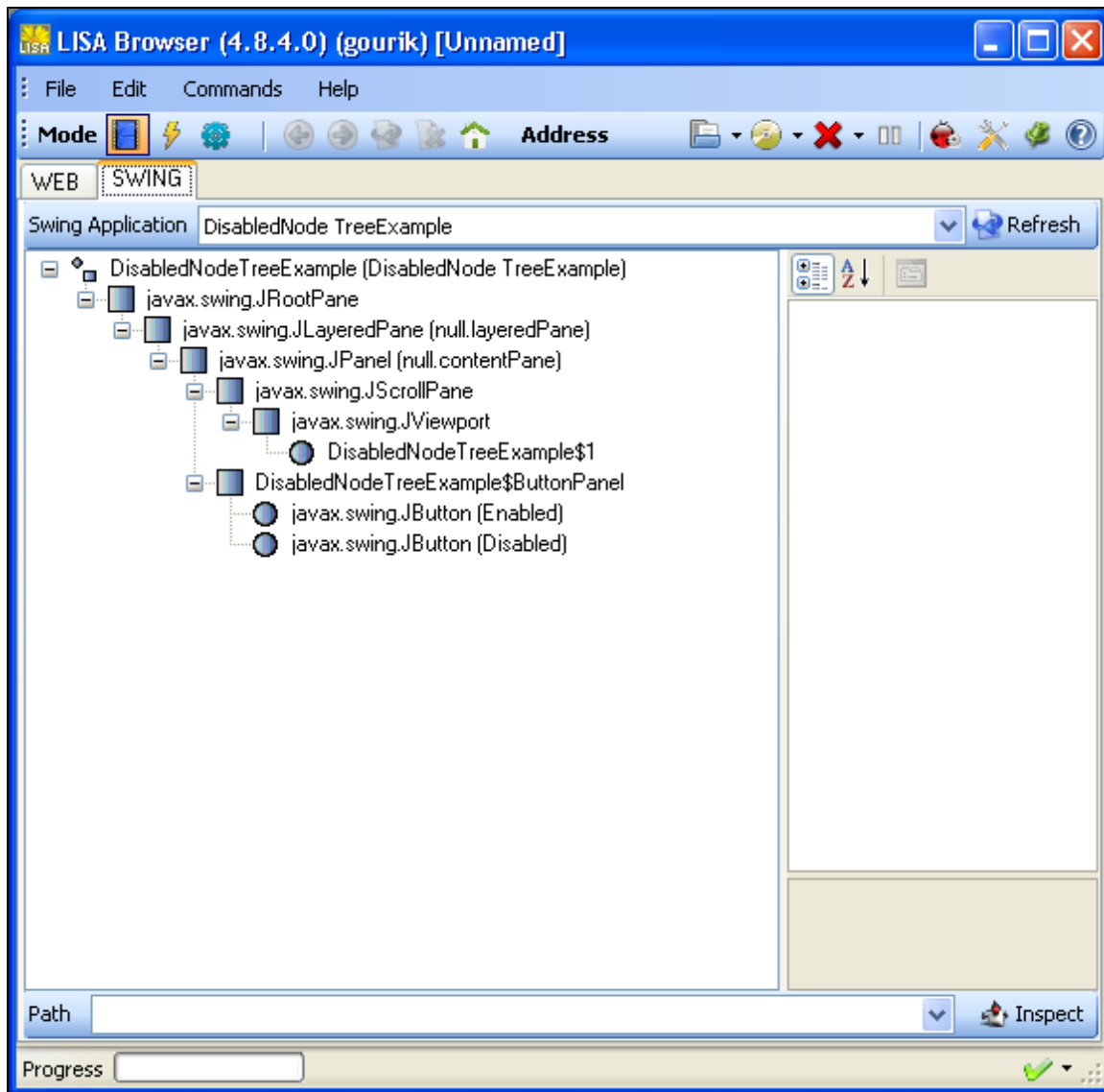


Swing View

The **SWING** view is identical to the **APPLET** view but represents the hierarchy of a recorded (or executed) Swing (or AWT) application.

In addition, selecting a node in the hierarchy will highlight the corresponding component in the Swing application.

The **Swing view** is a view of Swing operations. Here, we have running a swing application:



.NET View

The **.NET view** is a view of .Net operations.

The **.NET** view is identical to the **APPLET** and **SWING** views but represents the hierarchy of a recorded (or executed) .NET WinForms application. In addition, selecting a node in the hierarchy will highlight the corresponding control in the WinForms application.

In the tree on the left is the component hierarchy of all the UI elements that make up the applet. They are identified by class name and label or text. On the right side is the property grid that displays all the names and values of the fields of the java object that backs up the selected UI component in the tree.

2.5 Post Recording

2.5 Post Recording

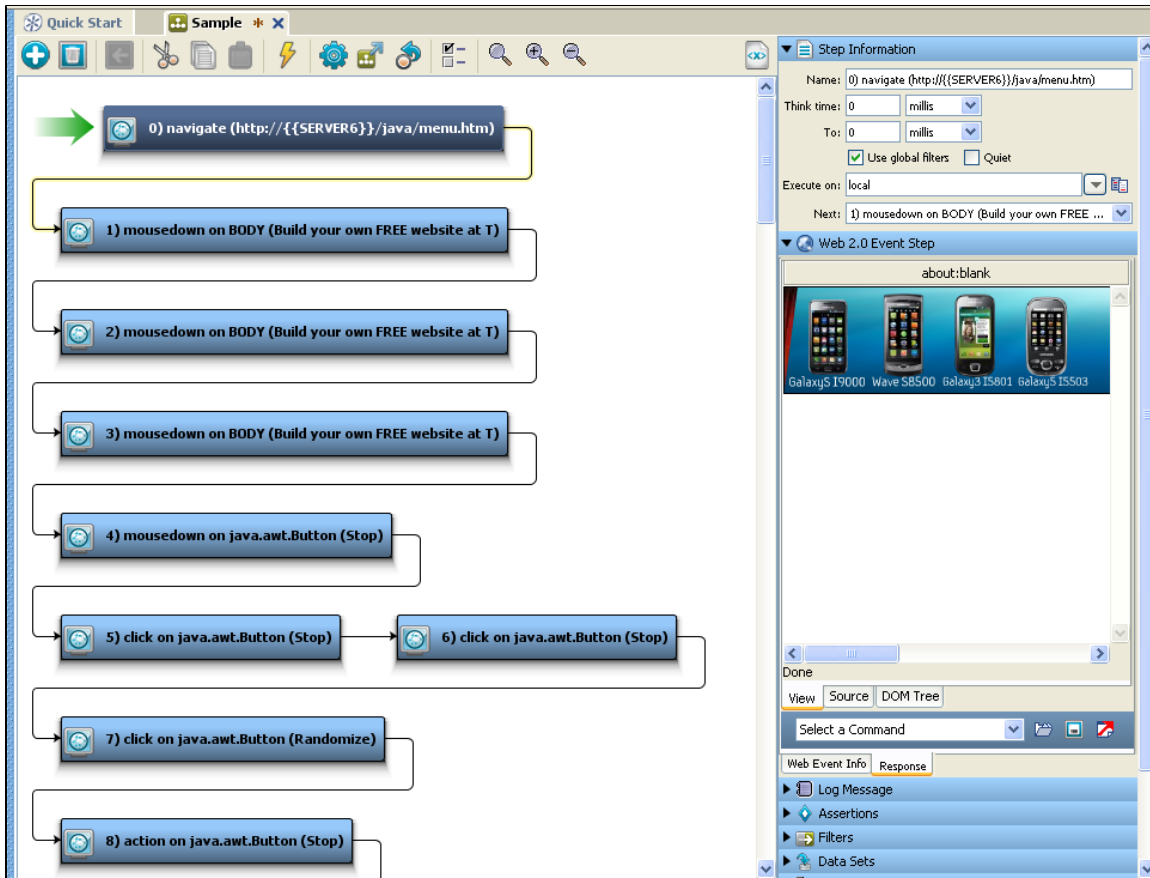
Once the recording is done and you save the recording, the browser closes the window.

You can now see the recording as a LISA Test case in LISA Workstation.

For example, we record a few pages of a website. Once we save this recording, it will close the browser window and you can see the test case formed in LISA workstation as shown below:

You can edit any of the test steps by double clicking on the test step in the LISA Workstation.

Select a particular step, to open its editor in the right panel:



There are two tabs at the bottom in the right panel.

Web Event Info tab - Click on this to see the Web Event Information. Click on the "Edit Events Via Recorder" button to open the LISA browser for editing as shown below:

▼ Step Information

Name:

0) navigate (http://{{SERVER6}}/java/menu.htm)

Think time:

0

millis

▼

To:

0

millis

▼

☒ Use global filters

☐ Quiet

Execute on:

local

▼

Next:

1) mousedown on BODY (Build your own FREE ...

▼

▼ Web 2.0 Event Step

vent

Launch Web Recorder

Edit Events Via Recorder

Event Type

navigate

▼

Window ID

0

▼

Frame ID

▼

URL

http://{{SERVER6}}/java/menu.htm

XPath

Value

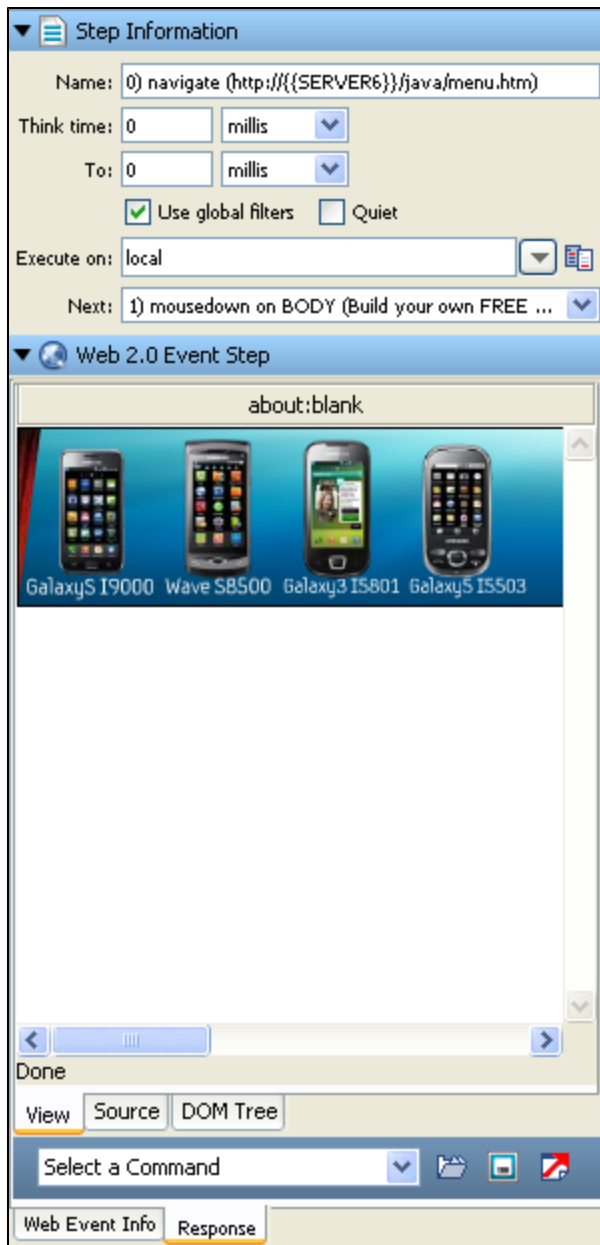
Think Time

0

Web Event Info

Response

Response tab - Click on the Response tab to see the response as shown below:



Similarly, you can click on the Source and DOM tree tabs to see the respective outputs.

▼

Step Information

Name:

0) navigate (http://{{{SERVER6}}}/java/menu.htm)

Think time:

0

millis

▼

To:

0

millis

▼

☒ Use global filters

☐ Quiet

Execute on:

local

▼

Next:

1) mousedown on BODY (Build your own FREE ...

▼

▼

Web 2.0 Event Step

<?xml version="1.0" ?><!DOCTYPE

<STYLE>A:link {

COLOR: #000000

}

A:visited {

COLOR: #000000

}

A:hover {

COLOR: #000000

}

A:active {

COLOR: #000000

}

#abgt .curve DIV {

BACKGROUND-COLOR: #666

}

</STYLE>

<SCRIPT>

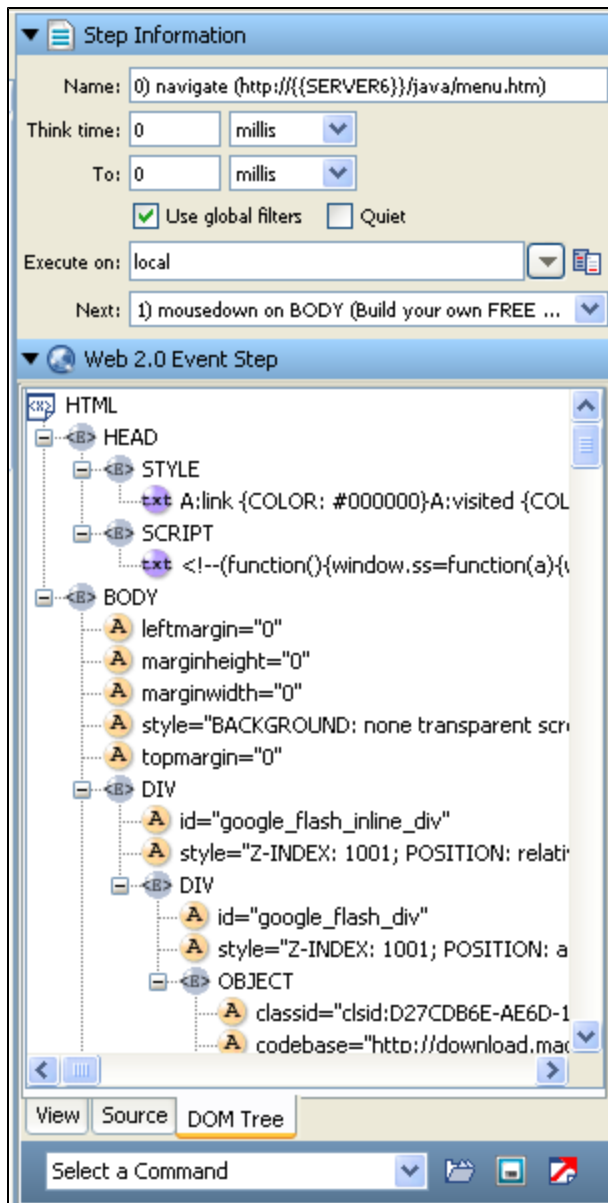
View

Source

DOM Tree


Select a Command

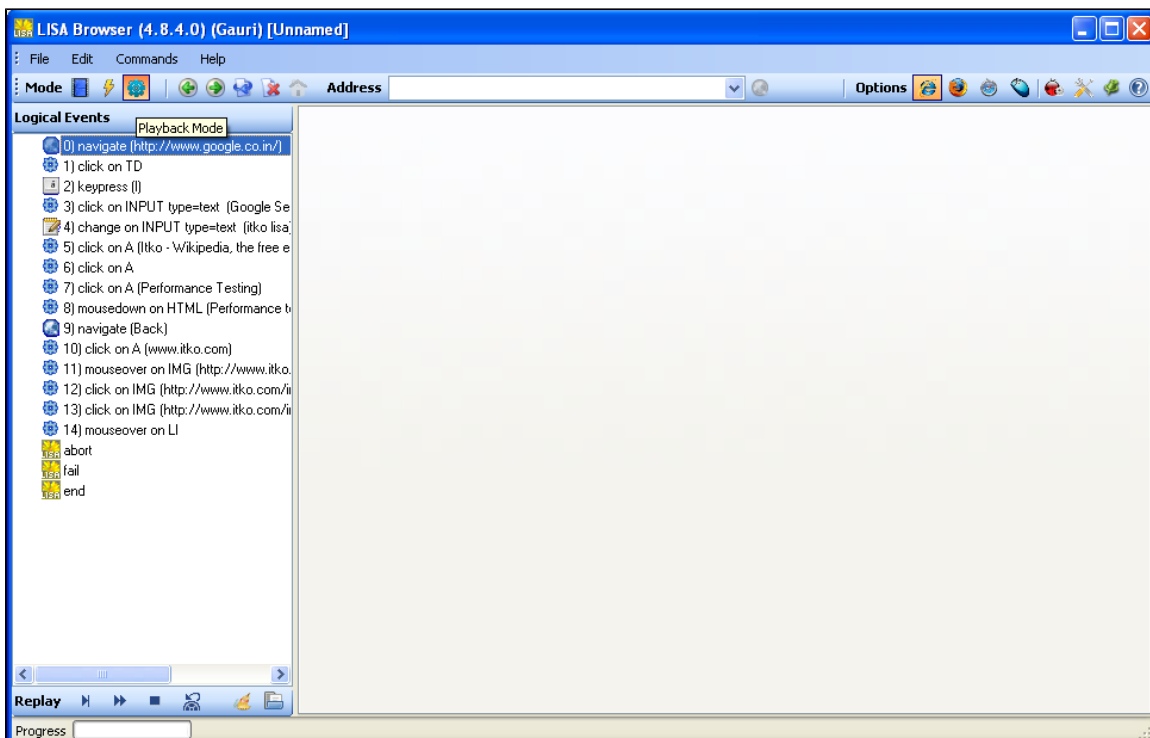
▼





Within the DOM tree view, you can select a Attribute and select a command to be applied to it as shown below:




The playback of the above recording is enabled by clicking on the playback mode button  on the toolbar as shown below:



The playback toolbar has a few different buttons than the recording mode, like , wherein you can select the **browser** for playback from the options given options..

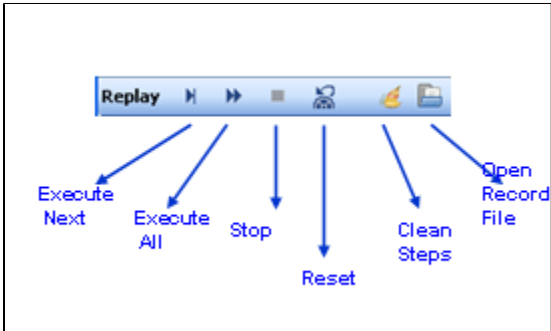
The Internet Explorer, Mozilla Firefox, Safari buttons,  control which browser is being used to execute the web test. You can select 1, 2 or 3 of them at a time. If more than one is selected, the selected browsers are used side-by-side. If none is selected, Internet Explorer is used as the default (since it is automatically installed on Windows).

You can use the **Move mouse** button  to turn on or off mouse movement during playback.

Playback Toolbar

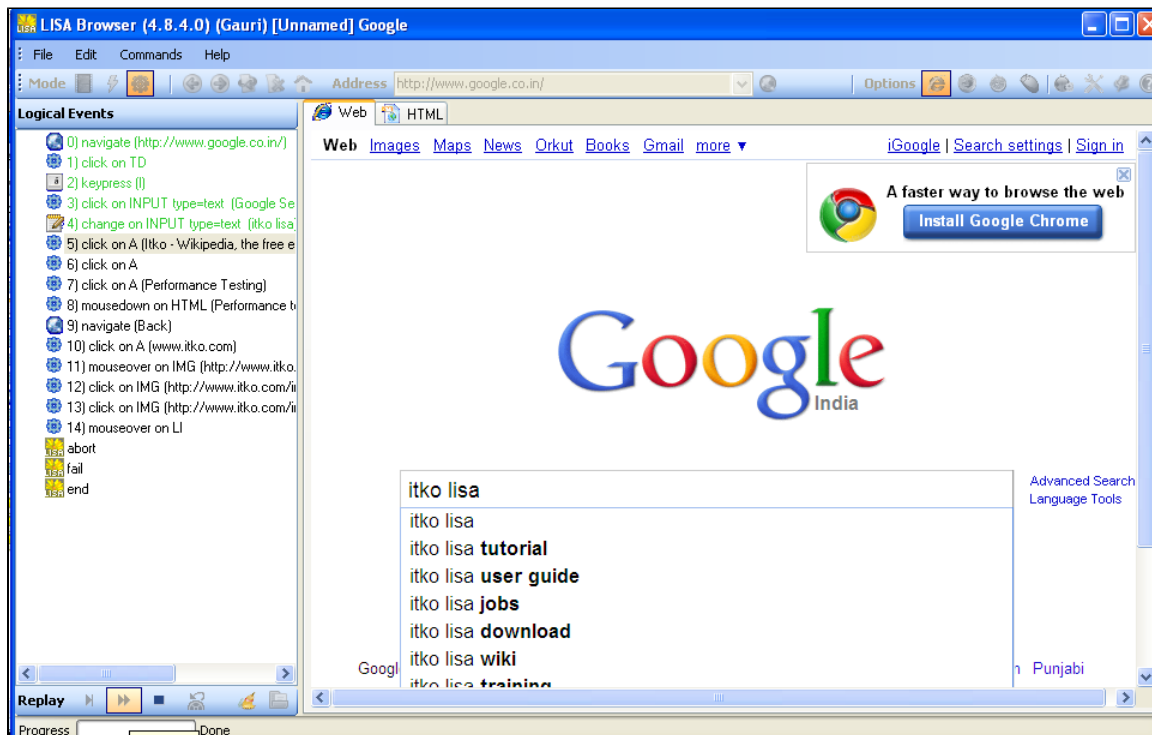
The playback mode has a toolbar  at the bottom of the Logical events pane, to control the movement of the playback activities.

Click on Execute Next or Execute all steps to run the playback as shown below:



- **Execute Next** executes the selected event in the list.
- **Execute All** executes all the events starting at the selected one.
- **Stop** will stop the playback.
- **Reset** will reset all the variables and executes all the events in the list starting with the first one.
- **Clean Steps** will disable all events that generated a warning in the last run.
- **Open Record File** will open a previously recorded file.

The Logical Events are listed on the left side and the entire recording will be displayed on the right side.

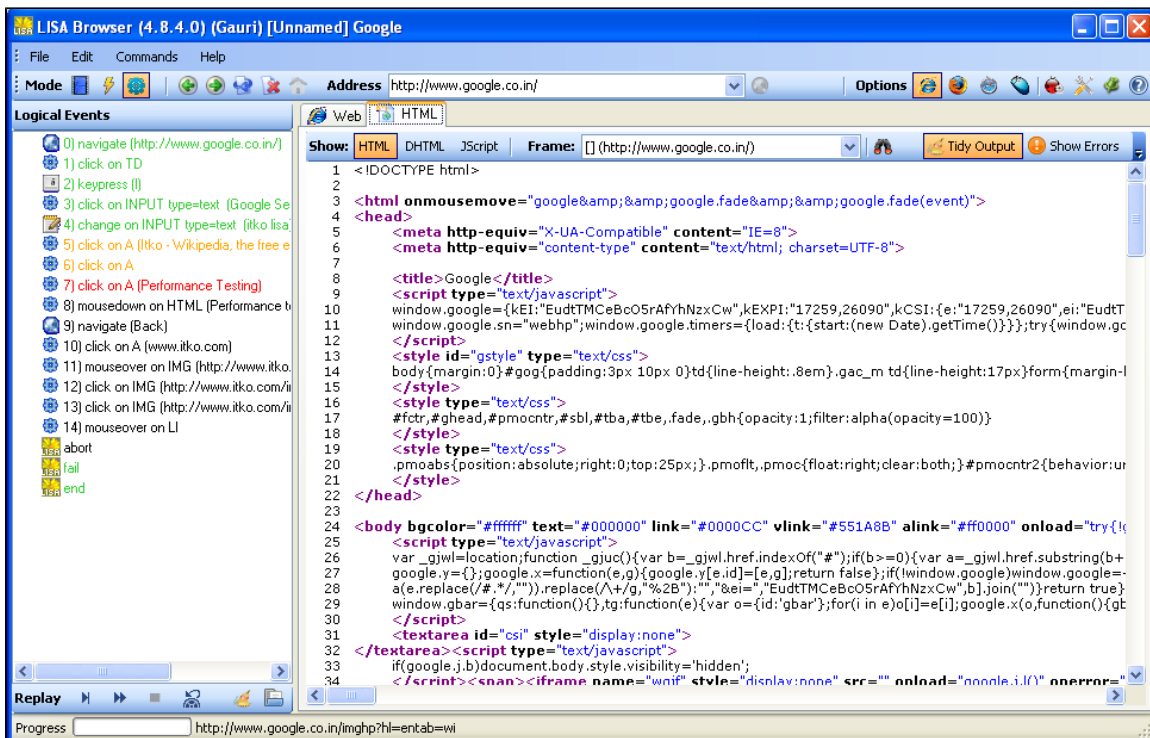


Once the playback starts, the steps that have run, are shown in green color in the Logical Events panel.

Tip: You can manually alter the flow by simply selecting an event in the list and clicking **Next** or **Play**. Alternatively, you can also press <CTRL><E> to execute the next node, which is useful if mouse movement is turned on.

If something is not going as you expect during a long test, you can also press CTRL-C to stop execution.

Click on the **HTML** tab within the Playback mode,



This will open a new menu bar as shown below:



- **HTML** - Click to view the html source
- **DHTML** - Click to view the DHTML source
- **JScript** - Click to view the Java Script source
- **Frame** - Will display the frame name
- **Find** - Will open a search window
- **Tidy Output** - Will show the tidy output
- **Show Errors** - Will open the error window

4. Edit Mode

4. Edit Mode

To open the Edit mode,

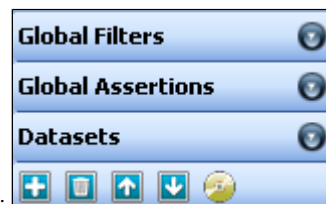


Click the Edit mode option in the toolbar to open the browser in the Edit mode as shown below:

In the Edit mode, you can view the Logical and Physical Events, Object Details and Response Panel.

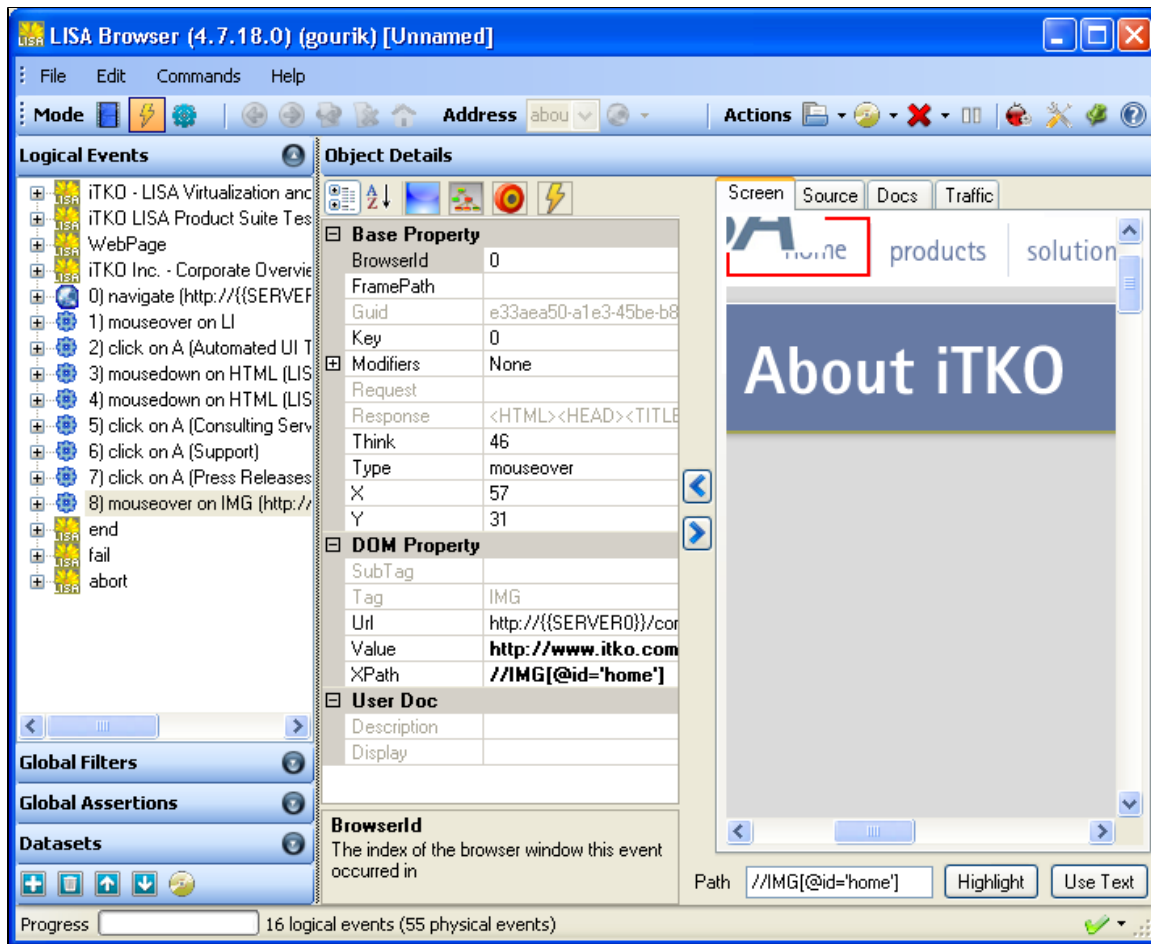


By default, the Logical events tab is open in the Edit mode.



In addition to this, there are the following tabs in the Edit mode which can be expanded:

The main edit mode window is as seen below:



In the left panel, a list of Logical Events is seen and on the right panel, you can see the Object details.

Please see the available subtopics for more information.

- 4.1 Event Types
- 4.2 Logical Events
- 4.3 Object Details
- 4.4 Filters
- 4.5 Assertions
- 4.6 Datasets
- 4.7 Editing Steps in Workstation

4.1 Event Types

4.1 Event Types

HTML pages can generate many different types of events and we're going to review them here to make it clearer when we talk about them in the rest of the document.

There are several **sources for Events**:

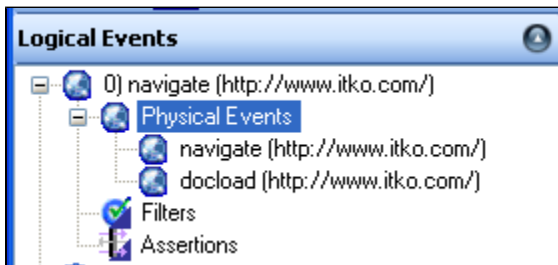
- Html page (DOM events),
- Browser environment (Native events),
- Plugins (Applet events and others),
- and finally actual events that are imported by LISA from steps that make use of other technologies or are used as markers (such as the fail or end events).

There are two types of events:

- Physical Events
- Logical Events

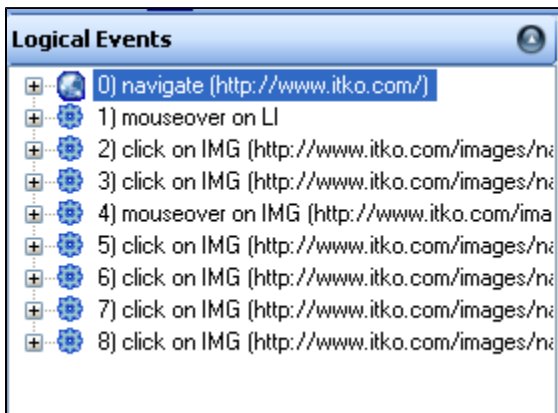
Physical Events - When we record or playback a Web 2.0 test, all the events occurring within or all actions taking place, are the physical events.

In the Web 2.0 browser, it is seen here -



Logical Events - However, from the user's point of view, only a small subset of these events is interesting and those are the logical events.

In the Web 2.0 browser, it is seen here -






For example, clicking on an html link could result in a mouse down, focus, mouse up and click events (Physical events), but the user sees this as only a click event (Logical Events).










LISA will actually group these physical events together into an event bucket and mark the click as the bucket's logical event.

DOM Events








| Icon | Event | Description |
|------|--------------|--|
| | Navigate | the user navigates to a page by using the address bar or the navigation buttons |
| | Doc Load | a html page or frame is loaded as a result of a user action |
| | KeyPress | the user presses a key |
| | Change | a DOM element's value is changed (inputs, selects or text areas are subject to this for example) |
| | Focus | a DOM element receives the focus in a page or frame |
| | Click | a DOM element is clicked |
| | Double Click | a DOM element is double-clicked |
| | Mouse Down | the user presses a mouse button |



| | | |
|---|------------|--|
|  | Mouse Up | the user releases a mouse button |
|  | Mouse Over | the mouse hovers over a DOM element area |
|  | Mouse Out | the mouse leaves a DOM element area |

Applet Events

| Icon | Event | Description |
|---|---------------------|---|
|  | Applet Load | a new applet is loaded by a Doc Load |
|  | Asynchronous Change | the applet hierarchy or visibility changed as the result of a user action |
|  | Focus | an AWT or Swing component receives the focus |
|  | Click | an AWT or Swing component is clicked |
|  | Double Click | an AWT or Swing component is double-clicked |
|  | Change | an AWT or Swing component's value is changed (text fields or com boxes are subject to this for ex.) |
|  | Mouse Down | the user presses a mouse button |
|  | Action | AWT/Swing's notion of an action event |
|  | KeyPress | the user presses a key |

Swing Events





| Icon | Event | Description |
|---|---------------------|---|
|  | Applet Load | a new applet is loaded by a Doc Load |
|  | Asynchronous Change | the applet hierarchy or visibility changed as the result of a user action |
|  | Focus | an AWT or Swing component receives the focus |
|  | Click | an AWT or Swing component is clicked |
|  | Double Click | an AWT or Swing component is double-clicked |
|  | Change | an AWT or Swing component's value is changed (text fields or com boxes are subject to this for ex.) |
|  | Mouse Down | the user presses a mouse button |

| | | |
|---|----------|---------------------------------------|
|  | Action | AWT/Swing's notion of an action event |
|  | KeyPress | the user presses a key |

Native Events

| Icon | Event | Description |
|------|-------------|---|
| | Open | a new browser window is opened |
| | Close | a browser window is closed |
| | Alert | a JavaScript alert dialog is clicked by the user |
| | Confirm | a JavaScript confirm dialog is clicked by the user |
| | File Dialog | a native File Open or File Save dialog is clicked by the user |

External Events

| Icon | Event | Description |
|---|----------|---|
|  | Continue | a no-op event |
|  | End | marks a test end |
|  | Fail | marks a test failure |
|  | | Any other non-Web 2.0 event imported from a LISA step |

.NET Events:

Each event has many properties attached to it that describe all the information necessary to replay it. We will go into those properties in detail in the next section, as we discuss how to modify these events after a recording.

4.2 Logical Events

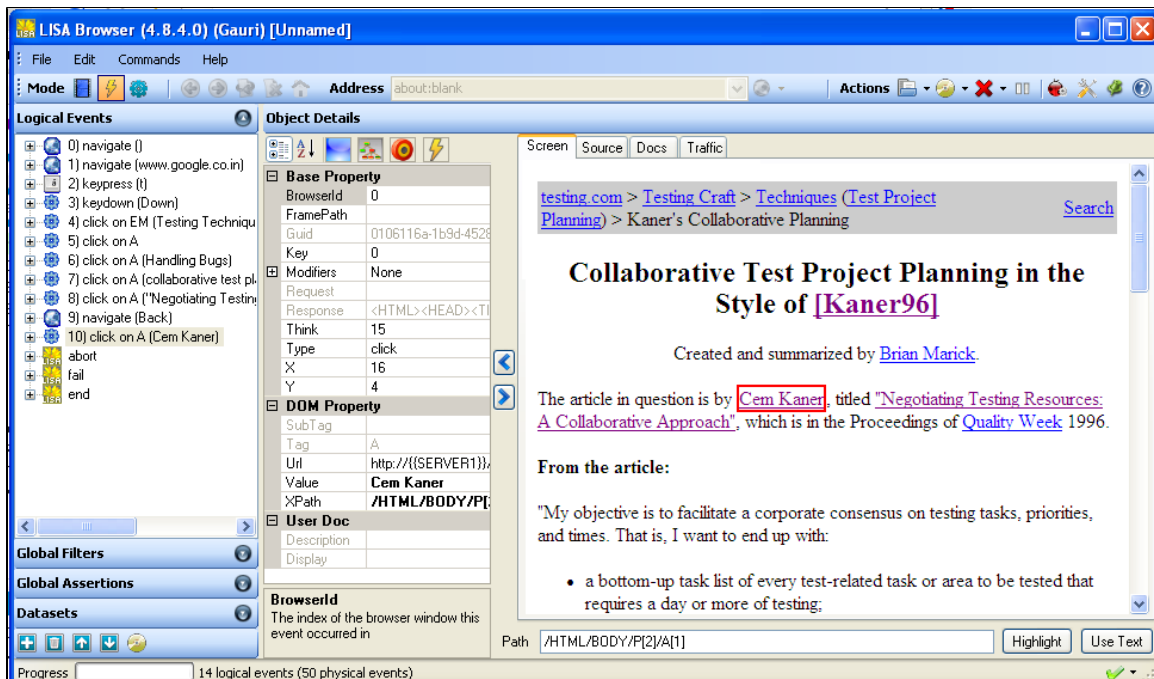
4.2 Logical Events

Once a recording is complete, a LISA test can be generated instantly by **saving** the recording as-is in the Browser window. You can then replay the recording in the Playback mode by reopening the browser window.

However, there are several reasons why it might be desirable **to inspect or modify** events before committing them to a Test Case, or to edit them on an existing Test Case.

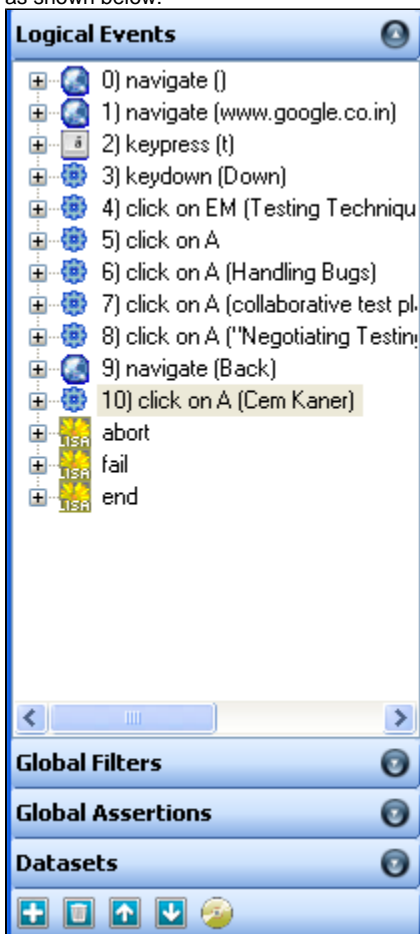
This is the purpose of the Logical Events tab in the browser.

In the Editing mode, you are able to view the logical and physical events of the test case. The Logical Events window in an Editing mode is as seen below:



Logical Events Panel

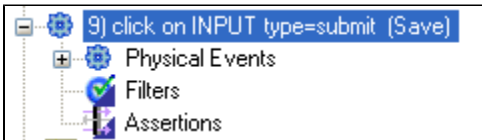
This is on the left of the Browser window. In the **Logical Events panel**, you can see the list of events recorded so far. The logical events panel is as shown below:



To view the Physical events (mouse clicks etc), expand the event by clicking on the "+" sign.

Each of these events is expandable so you can inspect all the physical events in its bucket, all the Filters and all the Assertions attached to that

event as shown below:








You will also see tabs for Global Filters, Assertions and Data Sets at the bottom.



Add/Modify/Delete Events

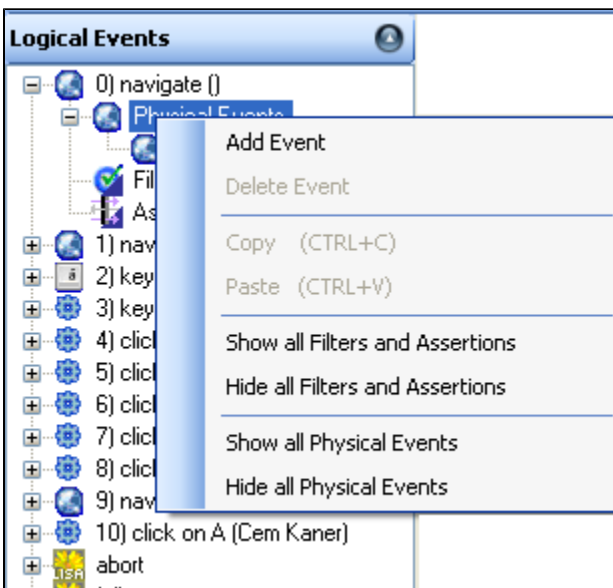
At the bottom, there is a toolbar which allow you to **Add/Delete/Modify an Event/Filter/Assertion** depending on what you select.



| Button | Action | Description |
|---|--------------------------------|---|
|  | Add an event | If a top-level event is selected, a new logical event will be added, otherwise a physical event will be added |
|  | Remove an event | External events can not be removed from the browser, you have to do it in LISA itself |
|  | Save an event | All the changes made to an event in the detail pane (on the right side) are committed |
|  | Move an event up in the list | Use with caution on physical events as results are sometimes hard to predict |
|  | Move an event down in the list | Use with caution on physical events as results are sometimes hard to predict |

In addition to the icons at the bottom of the screen, you can also manipulate events, Filters and Assertions in the left panel:

- Right-click the Step in the Logical Events column to open the screen below which helps to Copy/Paste, Add and Delete an Event



Similarly, you can right click on any filter or assertion to add or delete the same. You can also Physically drag and drop a step from one location to

another.

Configurations

A Configuration is a LISA element that provides a set of properties containing information about the environment or 'system under test'. Configurations can be de-coupled from the Test Case to obtain maximum portability. For detailed information on Configurations, see Using Configurations.

4.3 Object Details

4.3 Object Details

In the Edit mode, you can also see the Object Details Panel. All the details regarding the selected object are listed in this panel.

This panel changes according to the item selected in the left panel.

If you select the Logical events tab, its object details will be seen.

If you select the Global Filters/Assertions/Dataset tabs, their respective editors will be seen.

- Double click on an **object** in the Logical Event panel to open the **Object Details** as shown below:

Object Details

Screen Source Docs Traffic

testing.com > Testing Craft > Techniques (Test Project Planning) > Kaner's Collaborative Planning [Search](#)

Collaborative Test Project Planning in the Style of [Kaner96]

Created and summarized by [Brian Marick](#).

The article in question is by [Cem Kaner](#), titled "[Negotiating Testing Resources: A Collaborative Approach](#)", which is in the Proceedings of [Quality Week](#) 1996.

From the article:

"My objective is to facilitate a corporate consensus on testing tasks, priorities, and times. That is, I want to end up with:

- a bottom-up task list of every test-related task or area to be tested that requires a day or more of testing:

Path: /HTML/BODY/P[2]/A[1] [Highlight](#) [Use Text](#)

This panel is again divided into two parts: Left and Right

In the left part, all the properties specific to an object are seen.

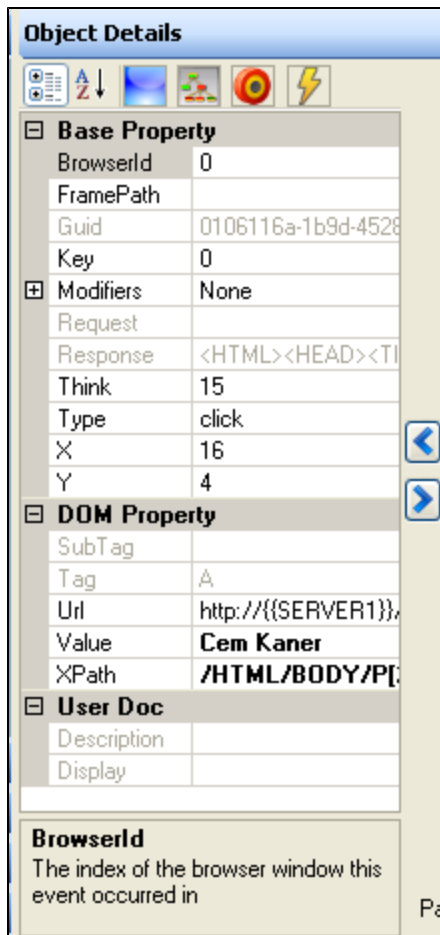
It is divided in 2 sections:

1. General properties (common to all types of objects),
2. Object-Specific properties (i.e. DOM/Applet or Native Properties etc.)

The right part shows the response of the selected object and can be seen only if it is enabled from the  icon.

Let's go over each of these in detail.







Object Details panel



At the top of the Object Details panel, there are buttons which can rearrange the data as required:



Each icon has can be clicked or un-clicked to show or hide data.

- Click to get a categorized view of the data 
- Click to sort the data alphabetically 
- Click to show/hide response pane 
- Click to show/hide hierarchy 
- Click to show/hide Invisible elements 
- Click to enable/disable scripts 

The **Base property tab** is made up of the following:

- **Browser ID:** The index of the browser window this event occurred in
- **Frame Path:** The path to the frame this event occurred in
- **Guid:** The unique event identifier
- **Key:** The keyboard key associated with this event
- **Modifiers:** The mouse or keyboard identifiers in use when this event occurred
- **Request:** The request data associated with this event
- **Response:** The source of the container at the time of this event
- **Think:** The think time of this event
- **Type:** The type of this event
- **X:** The X component of this event relative to its event

- **Y:** The Y component of this event relative to its event

The **DOM property tab** is made up of the following:

- **Sub tag:** The html type attribute of the target element
- **Tag:** The html tag of the target element
- **URL:** The URL of the browser in which the event happened. For Ajax events the URL shown is not the URL shown in the browser, but the URL available to the Server to handle the Ajax calls.
- **Value:** the Value of the DOM element after the event fired, if it makes sense in this context.
- **XPath:** the XPath expression that uniquely identifies the DOM element that was the target of the event.

The **Native properties** are made up of the following:

- **Button:** the label of the button the user used to dismiss the native dialog (such as Open, Save, OK, Cancel, etc.)
- **Selection:** an optional selection value that some dialogs offer (such as a filename)
- **Username:** currently not used
- **Password:** currently not used

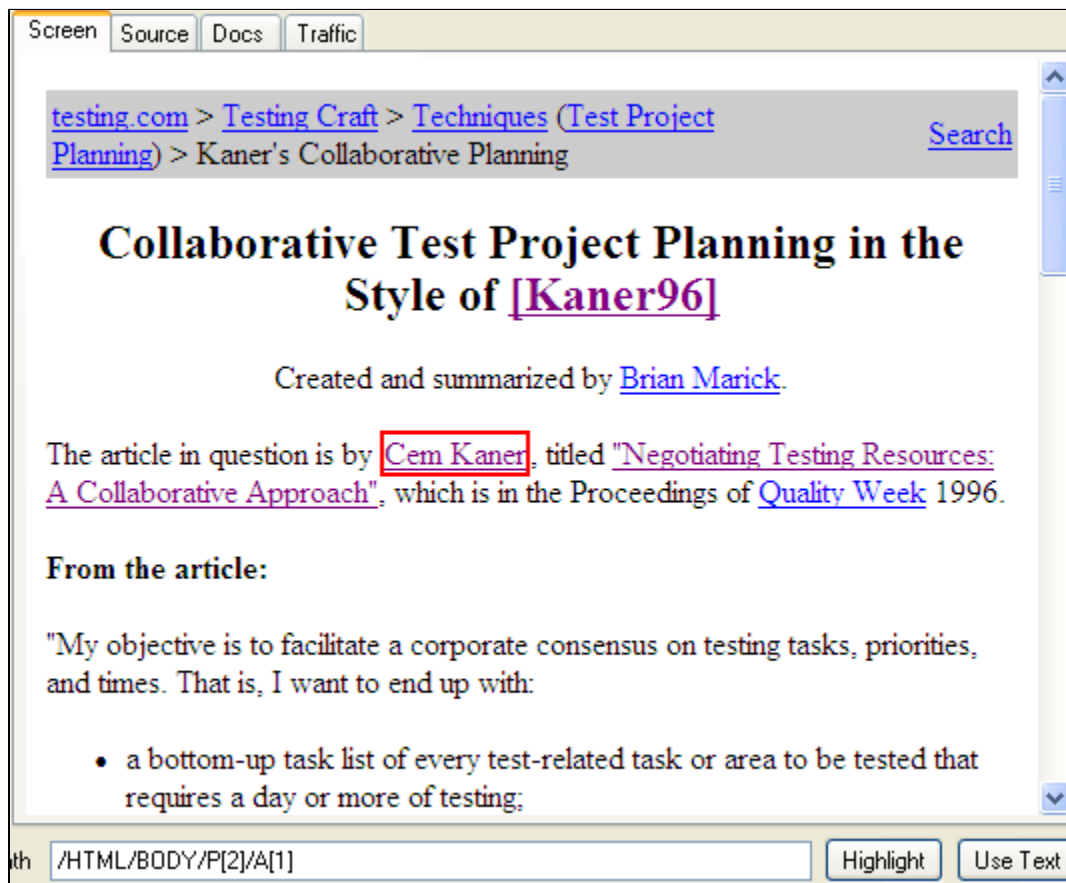
The **Applet properties** are made up of the following:

- **Applet:** the class name of the applet that received the event
- **Path:** the xpath-like expression that uniquely identifies the AWT or Swing component that was the target of the event
- **Class:** the class name of the target component
- **Text:** the text or label of the component after the event was fired

Response Panel

Finally, the Response panel contains the state of the page after the event was fired.

In the case of a DOM event, that translates to an HTML source, in the case of an Applet event, it is an Applet hierarchy. As of now, responses are empty for Native Events.



It has four tabs -

Screen - This shows the actual html page

Source - This shows the html source of the page

Docs - Will show the docs if any

Traffic - This shows the Request and Response headers for the html page.

For an **Ajax request** the Response shows the resulting document that gets loaded.

When you decide to edit an event after a recording is complete, some of the fields are hard to fill correctly, notably the **Path or XPath** fields.

To assist you, there are some Browse Buttons available next to those fields. When you click them, it will bring up a mode dialog window that contains a browser and helps you select those paths.

Below are pictures of these browsers in the case of a DOM event and of an Applet event.

These DOM and Applet browsers have 3 main sections:

- An editable combo box at the top that contains an XPath expression along with a Select and Cancel buttons
- A tree view on the left along with a property grid underneath it
- An actual browser that renders an HTML page or a snapshot of the applet.

You can browse the page or applet by clicking either in the browser or the tree and they will automatically synchronize with each other and the XPath combo box. Typically you would click an element in the browser to generate its xpath, unless it's a hidden element, in which case the tree view is appropriate. When you click an element, either in the tree or the browser, it gets highlighted so you can be certain it is what you think it is. When you are satisfied with the element you chose, you can click the Select button and the Xpath will be transferred to the field that offers the Browse. If you change you mind, Cancel will discard the browser without any changes.

Note that this browser disables all JavaScript or dynamic behavior and can be accessed offline. However it still goes to the Server for css and images if the cache is empty on your disk.

4.4 Filters

4.4 Applying Filters

Authoring or executing a test, steps or events are only half of the equation in LISA.

You need to be able to parameterize inputs and outputs to make the test generic and be able to assert on the results to decide what constitutes success and what constitutes failure. Filters address the first of these points.

In the context of Web 2.0 tests, you can think of Filters as functions that execute after an event and store the result in a variable that can then later be accessed by other events, Filters or Assertions.

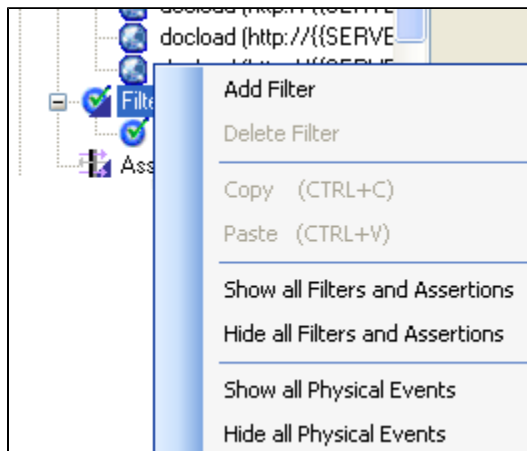
You can inspect, add, remove, or edit Filters in Edit mode of the browser.

These Filters are listed in the Logical Events section. You can see them once you expand the Object tree.

Event Filter

To add an Event Filter

- Select any logical event and click to expand the tree.
- Right click on the Filter node and select **Add Filter** option as shown below:



The filter editor will open in the right panel as shown below:

Object Details

Definition
A filter is a function that executes before or after an event is triggered and stores its result (Filter Value) in a variable (Filter Key).

Filter Key:

☐ Wait up to ms for value

Properties
A text filter retrieves the inner text of a DOM element using a regular expression (first capturing group).

☐ DOM Element

☒ Text

☐ DOM Attribute

☐ Script

☐ Expression

☐ Cache

☐ Capture

☐ Import

☐ Hardware

☐ Code

☐ File

☐ Dataset

☐ Browser ☐ Internet Explorer ☐ Firefox ☐ Safari

☐ Sleep ms

☐ Secure Prompt

Quick Test
Click Evaluate to see what value this filter would return if it were evaluated during the recording.

The Filter object details pane has all the information required to set up a new filter or edit an old one.

After the definition of the Filter, you can see the following:

Filter Key: The key is the name of the variable that is going to receive the Filter result after it executes.

- Or Click on the *Browse* button to open a *Property Browser* as shown.

Property Browser

| Key | Value |
|---------------------------|--------|
| {{instance.name}} | gourik |
| {{instance.number}} | 0 |
| {{event.name}} | |
| {{event.index}} | -1 |
| {{event.type}} | |
| {{event.frame.path}} | |
| {{event.path}} | |
| {{event.response}} | |
| {{event.url}} | |
| {{event.status.code}} | 0 |
| {{event.script.error}} | |
| {{event.bytes.in}} | 0 |
| {{event.bytes.out}} | 0 |
| {{event.bytes.total.in}} | 0 |
| {{event.bytes.total.out}} | 0 |

- Click Select to add.

Filter Properties: the following types are available. Clicking on any of the Filter properties will give its description.

DOM Element: A DOM element Filter extracts an HTML element object from its event's DOM

Text: A text Filter extracts a piece of text from its event's response

DOM Attribute: A DOM attribute Filter extracts an attribute value from its event's response

JavaScript: A JavaScript Filter executes arbitrary JavaScript code in the context of its event's response

Expression: An expression Filter allows you to combine other Filters

Cache: A cache Filter clears the specified browser cache

Capture: A capture Filter saves the current response to the specified location

Import: The import Filter makes the specified file (js or java) available as a library to the running application

Hardware: A hardware Filter executes the selected mouse or keyboard action on the specified desktop

Code: A code Filter executes arbitrary .Net code that has access to all the LISA browser variables

File: A file Filter allows you to execute a variety of operations on file, local or remote

Data Set: A Data Set Filter automatically extracts data out of the specified element into a Data Set according to row and column path rules

Browser: A browser more Filter toggles the state of the playback window to use the selected browsers

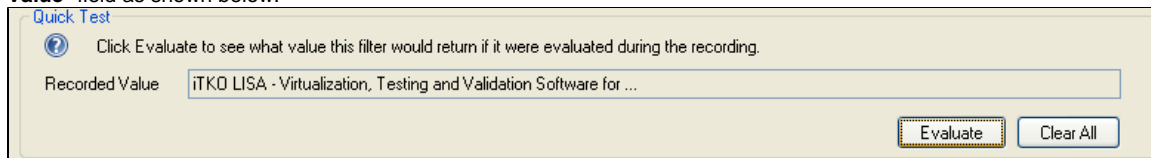
Sleep: A sleep Filter sleeps for a specified amount of time

Secure Prompt: A secure prompt Filter lets you fill in some variable at runtime if you do not want them persisted

Quick Test:

Click to evaluate to see what value this filter will return.

Evaluate: Click on Evaluate to evaluate the filter. The Filter runs with the response it got during recording and displays its result in the "Recorded Value" field as shown below:



Global Filter

To add a global filter,

- Click on the Global Filter button  at the bottom of the Logical Events pane.
- Click the Add button  to add a global filter.

This will open the same filter editor as for event filter.

- Enter the appropriate criterion so that the filter is applied to all the events.

Example

Let's say the page has (and should have) the text "Welcome John" inside some div after a certain DOM event. To extract the value "John" and put it in a variable for later use, you should pick a Text Filter, and a regular expression like "Hello (w+)". The first capturing group in the regular expression will be used to find John in the page. If you want to be even more specific, or you think there could be more than one match in the page, you could browse for the div that should contain this text and use it as the DOM element.

If you wanted to put the document's title into a variable, you could use a JavaScript Filter with the code: "document.title". If you needed to put the number of rows in a certain table, some JavaScript code like "document.getElementById('mytable').rows.length" would be appropriate.

Let's now say you had 2 tables whose number of rows should always add up to the same value. You could add 2 Filters like the previous one (say "f1" and "f2"), and add another Composite Filter "f3" whose value is "f1 + f2" and use it. Once you add some Filters, the resulting variables are available to use almost anywhere else, and the combo boxes for editable fields will then contain those variables.

Finally, there are some intrinsic Filters that are not displayed in the list because they are not editable.

They populate a set of "well-known" variables after each step:

```
event.type: The last event type
event.path: The last event path
event.response: The last event response
event.raw.response: The last event raw response
{event.url: The last event URL.
```

Note: The distinction between response and raw response is that responses may be computed from raw responses and other factors such as previous event responses. In particular most events generate only a raw response that is a diff between various DOM states. The response just

rebuilds the current DOM based on a previous DOM by applying successive diffs.

4.5 Assertions

4.5 Applying Assertions

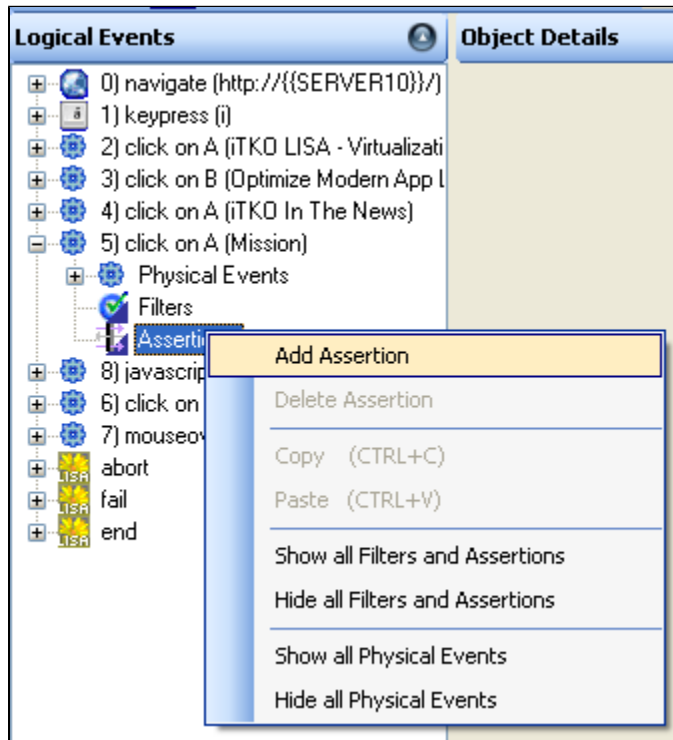
Assertions are also functions that execute after an event is fired, and return a Boolean value. If the return value is true, the test proceeds normally, otherwise what happens depends on the Assertion itself. It is a typical if-then-else scenario.

Practically speaking, you can inspect, add, remove or edit Assertions in the Events tab of the browser.

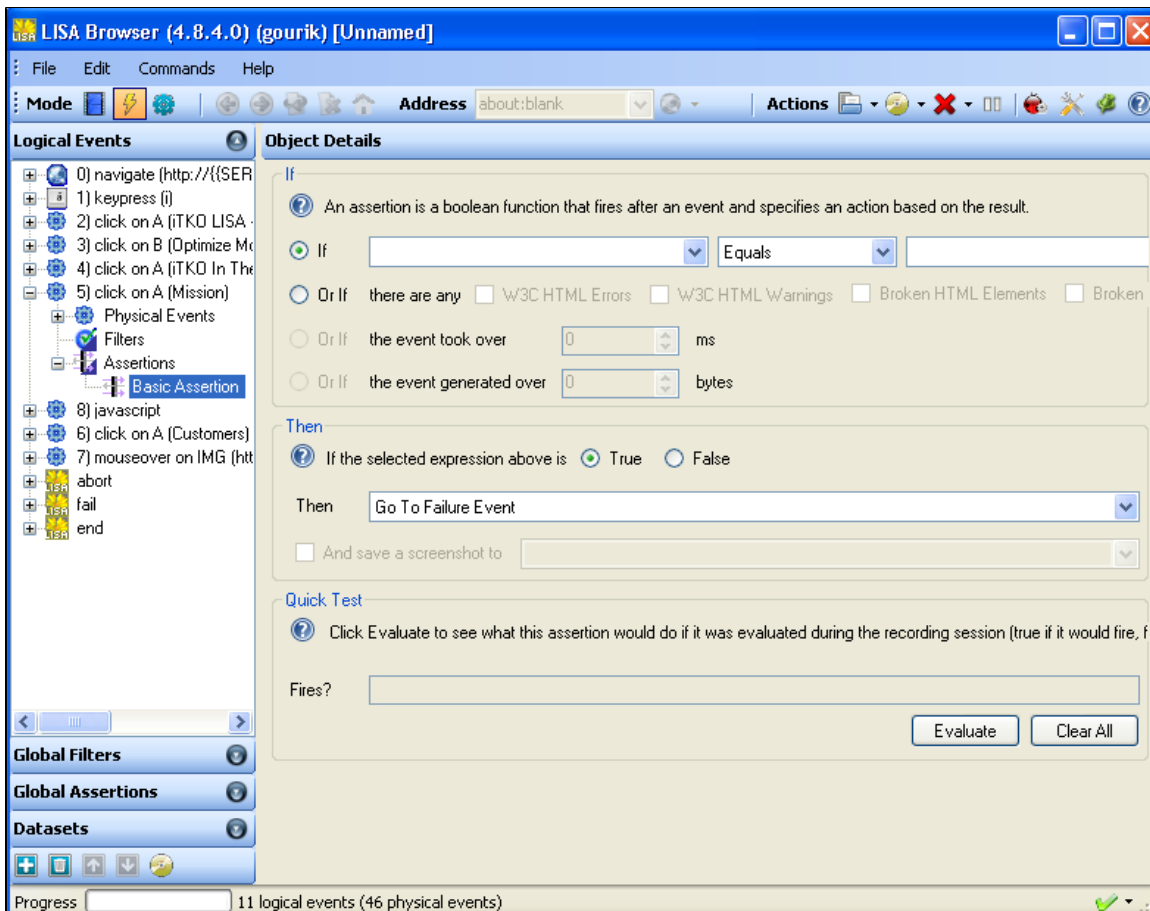
Event Assertion

To add an Event Assertion,

- Click on any logical event to expand the tree.
- Right click the Assertion node and select **Add Assertion** as shown below:



The Assertion Editor will open in the right "Object Details" pane as shown below:



On the Assertion object details pane, after a quick description of the Assertion type that is selected, you can see the following controls.

You need to select two parameters, which are to be compared.

IF - Select the event which you want to compare in the first drop down and select the event, with which you want to compare, in the second drop down.

Select the *type of operation* to be applied in the middle drop down.

Type of operations are -

Equals: those compare the 2 sides of the expression for equality

Not Equals: those compare the 2 sides of the expression for non-equality

Matches: those attempt to match (as regular expression) the left or the right side of the expression against the other side

Not Matches: those attempt not to match (as regular expression) the left or the right side of the expression against the other side

Less Than: those compare the 2 sides of the expression as numeric values for order. Returns false on non-numeric values

More Than: those compare the 2 sides of the expression as numeric values for order. Returns false on non-numeric values

THEN -- Select the step to execute if the expression is TRUE/FALSE

The main point in an Assertion is its expression.

It is this, that determines the success or failure of a test so it is important to be careful in constructing this expression.

There are several types of built-in expressions.

Quick Test



Click to evaluate to see what this expression will do if it was evaluated during the recording session (TRUE if it would fire, FALSE otherwise).

Typically, you will use a variable created by a Filter on the left side and then equal it or match it to a constant value or another variable on the right side. In more advanced cases you can use JavaScript expressions to assert on arbitrary conditions.

Evaluate: Click to evaluate.

Global Assertion

To add a global Assertion,

- Click on the Global Assertion  button at the bottom of the Logical events panel.
- Click the Add button  on the toolbar to add an Assertion.

This will open an editor similar to the Event Assertion.

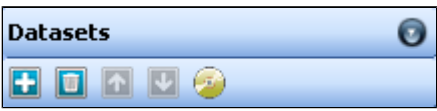

- Enter the required details to run the assertion on every event.

4.6 Datasets

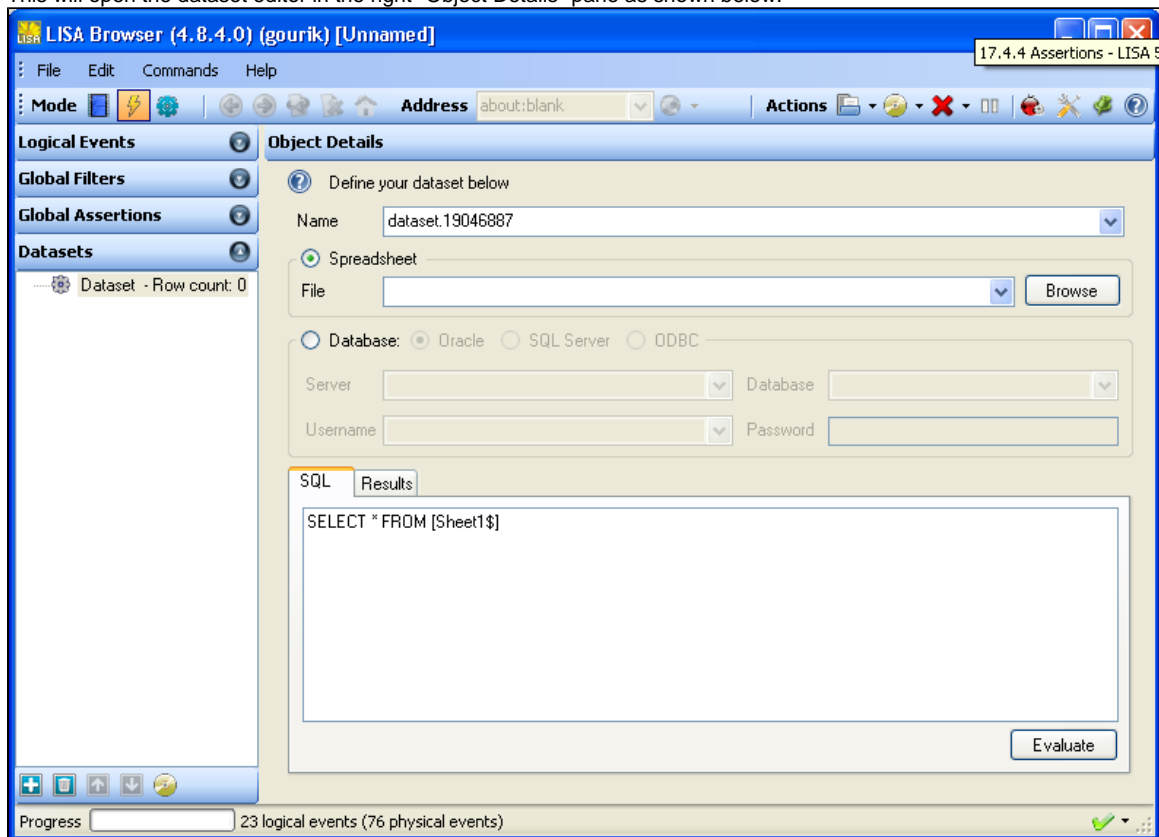
4.6 Applying Datasets

Datasets can only be applied globally, i.e to the entire test cycle.

To add a global dataset,

- Click on the Datasets button  located at the bottom of the Logical events pane.
- Click the Add button  to add a dataset.

This will open the dataset editor in the right "Object Details" pane as shown below:



Define the dataset in the dataset editor.

Name - Enter the appropriate name or accept the default provided by LIS

Data Source - Select the source of the dataset - either from a spreadsheet or from a database.

Spreadsheet - Select this to attach an excel sheet as data input.

- Enter the name of the excel file or browse and select the file.

Database - Select this to select the data source from a database. It will further open the type of databases to choose data from:

- Select one from the Oracle database, SQL database or ODBC database.
- Enter all the details related to the database.
- Enter the SQL query
- Click Evaluate to evaluate the results.

4.7 Editing Steps in Workstation

4.7 Viewing and Editing the Test steps

One of the major strengths of LISA is its outstanding ability to create and run tests that make use of a mix of different technologies (web, j2ee, web services, swing, etc.) as is so often necessary in the enterprise software world.

Web 2.0 tests are no exceptions in this regard and can be mixed with any other type of step. Nothing special is required to achieve this.

To Add steps to an existing test case,

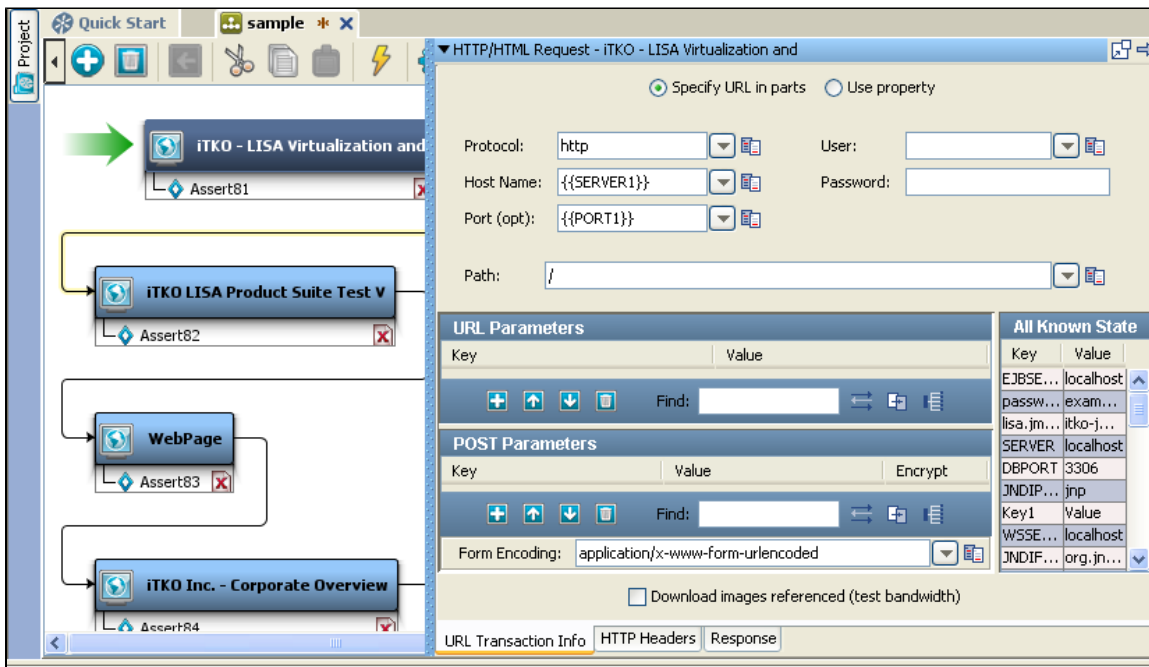
- Open an Existing test case and then start the web 2.0 recording.
- Once done, Save the recording. This will close the web browser.
- This will add all the recorded steps to the already existing test case in the model editor.

To Edit/Add/Delete Web 2.0 steps,

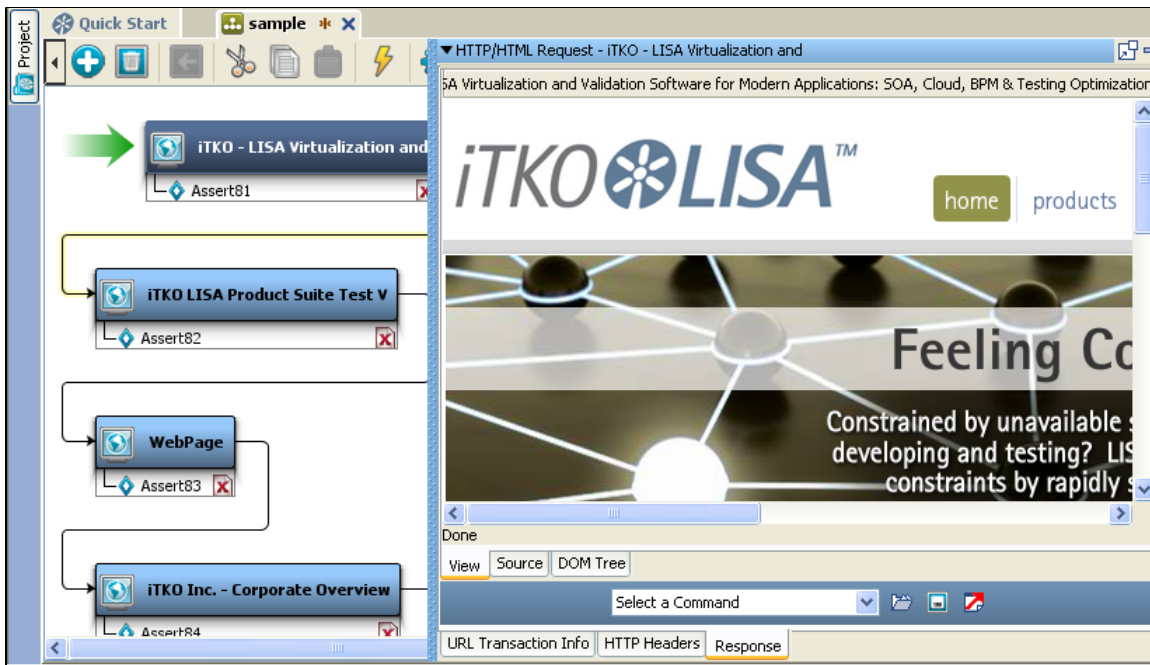
This can be done by requesting a **detailed view** of existing steps.

- Double click the step in the Model editor for which you need a detailed view.
This will launch the editor of the respective test case.

For example, we have chosen to view the detailed view of the HTML type of test step and hence it has opened a **HTTP/HTML Request editor** as seen below:



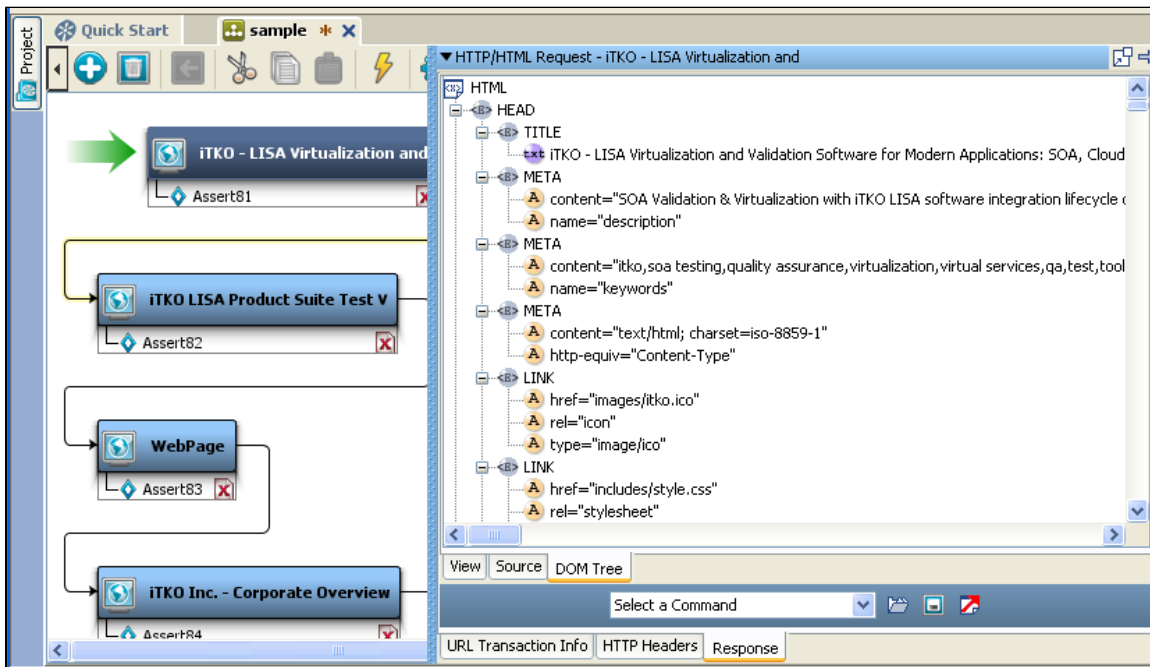
You can also view the response of the Web 2.0 steps exactly the same way you were doing it with regular web steps, by selecting the **Response** Tab in the step's detail pane.



For HTML pages, these responses come in 3 flavors:

- as Headers
- as XML source
- as a DOM tree

Image below shows the DOM tree response:



5. Debugging

5. Debugging

You may need debugging, If you want to debug a code or see the problem area.

Instead of clicking Next repeatedly for every step, you can set one or more breakpoints in the events list and click Play or Replay. Once you have reached the events of interest, you can step through one by one.

If something is not going as you expect during a long test, you can also press **CTRL-C** to stop execution.

Debugging within Browser

In the top toolbar, there is a **Show/Hide Debug Window**  icon.

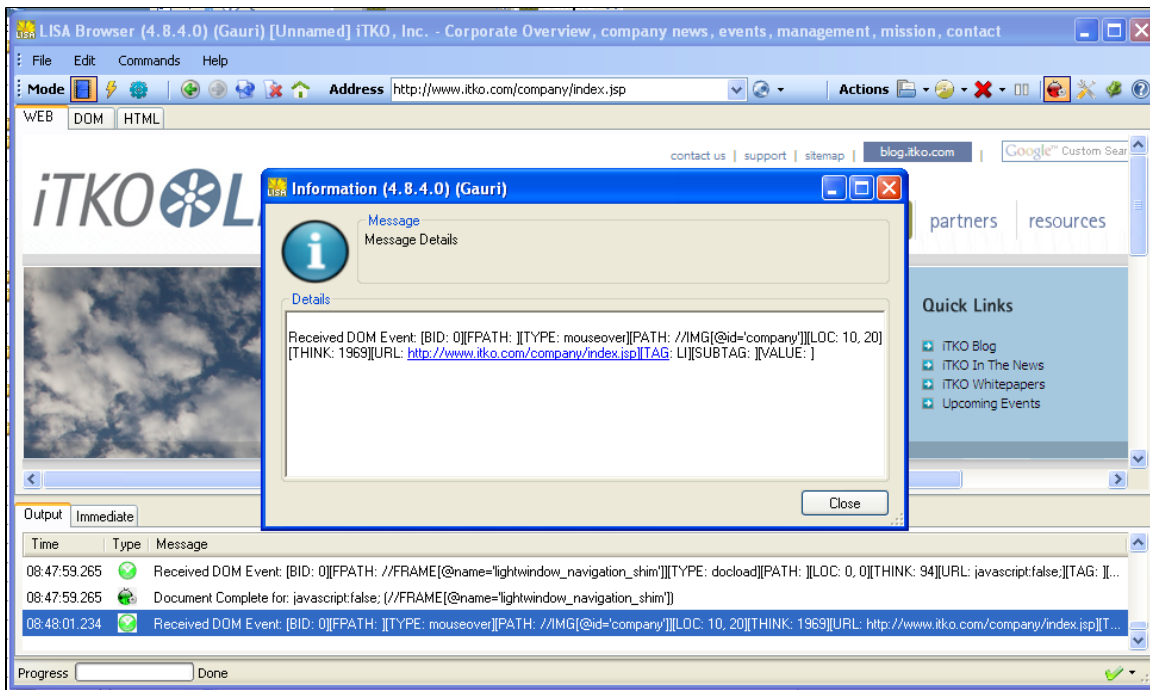
Click to open a debug window at the bottom with a set of tabs.

We have opened the debug window in the Recording mode as below:



The **Output tab** logs information about execution and potential errors or warnings.

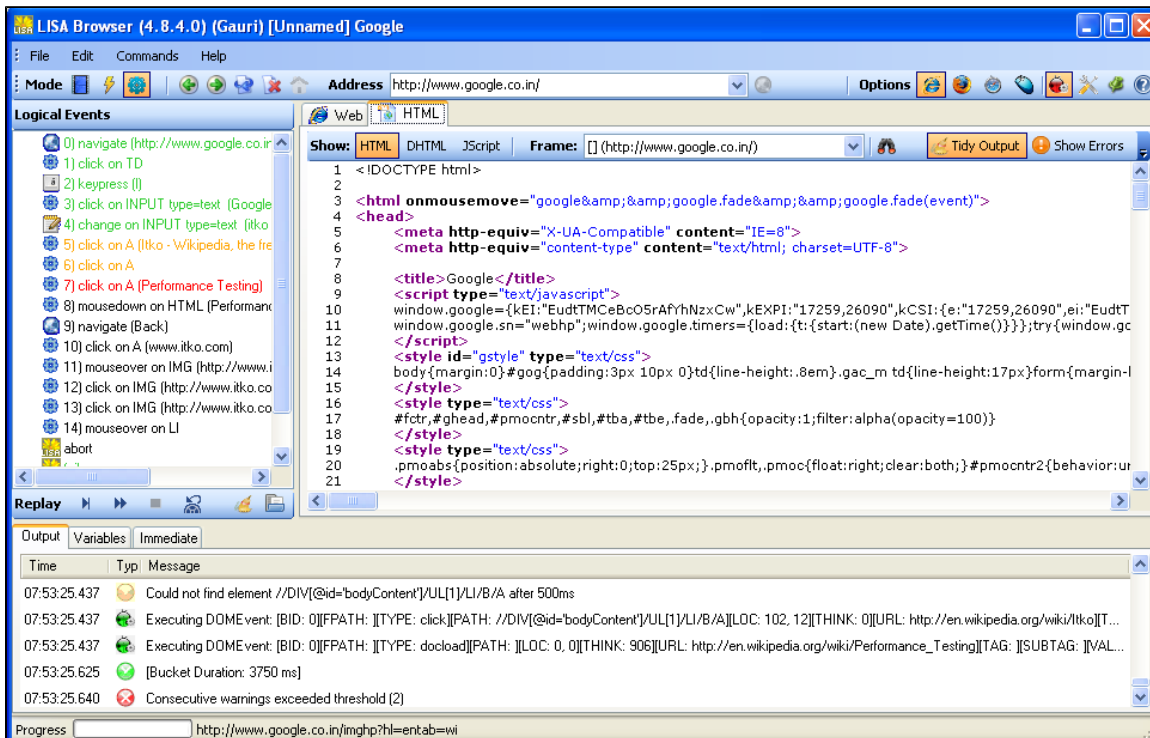
When you double click an **error/message** in the debug window, information about the same is seen.



The **Immediate** tab lets you execute arbitrary JavaScript functions (including using variables).

It is used at design time to debug and evaluate expressions, execute statements, print variable values etc.

We have opened the debug window in the **playback mode** as below:



These tabs are same as in the Recording/Edit mode - with an addition of one Variable tab.

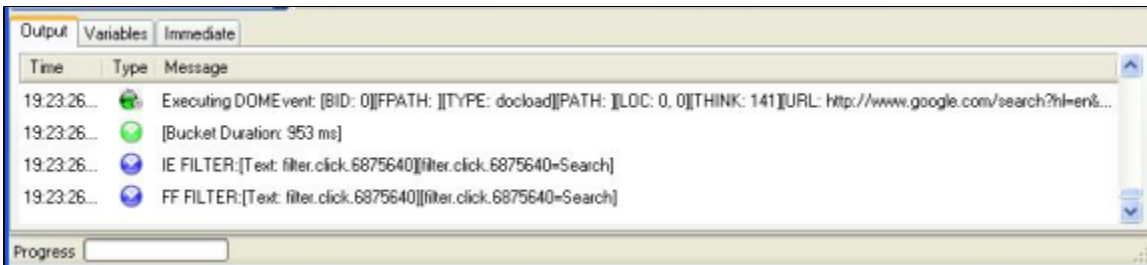
The **Variables** tab shows you all the variable names and values at the current step (the ones highlighted in red are the ones that were potentially modified by the last event)

As a further debugging mechanism, there is an HTML tab at the top that allows you to see the dynamic HTML source of the current page.

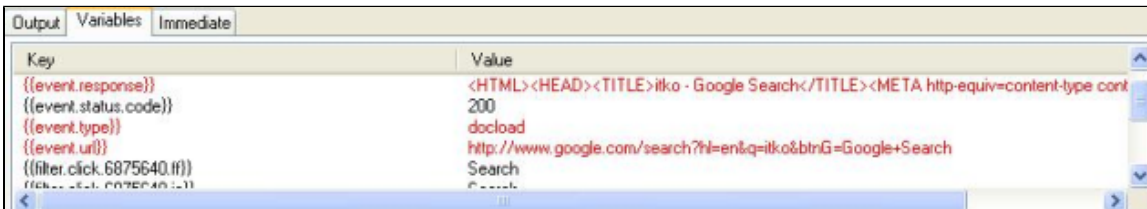
The Split toggle also applies in this source view for further comparison.

Multiple Browsers

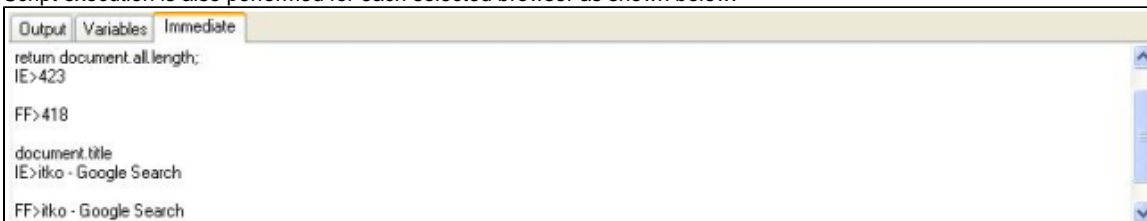
When multiple browsers are selected you get the logging information for each of them as shown below:



Filters also get executed for each browser and the browser abbreviation is appended to the filter key so they can be referenced individually, as shown below:



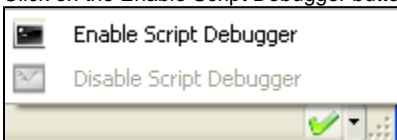
Script execution is also performed for each selected browser as shown below:



Opening a System Debugger

To open a system debugger,

Click on the Enable Script Debugger button at the bottom of the window:



This will open a script debugger if present in the system.

6. Setting up ADF Extensions

6. Setting ADF Extensions

To set up the ADF extensions in the LISA Browser, the browser needs to be updated with the extensions.

To update the extensions,

Open the LISA Browser.

Click on the **Lisa browser -> Help menu -> Extensions Update.**

To update the settings,

Open the LISA Browser.

Click the **LISA Browser > Edit menu > Browser settings.**

This will open the browser settings window.

Click on the **Recording** Tab as shown below:

LISA Settings

General **Recording** Playback Environment

Recording Strategy

All

Ignore ids and names whose value matches:

☐ Externalize recorded text

DOM

Ignore frame names whose value matches:

Ignore text whose value matches:

Use the following attributes (in this order):

| | |
|--|--|
| 1) <input type="text" value="id"/> | 5) <input type="text" value="<none>"/> |
| 2) <input type="text" value="name"/> | 6) <input type="text" value="<none>"/> |
| 3) <input type="text" value="<none>"/> | 7) <input type="text" value="<none>"/> |
| 4) <input type="text" value="<none>"/> | 8) <input type="text" value="index"/> |

Or use the following locator javascript function
☒

☒ Ignore invisible elements

Java

Record using:

| | |
|--|--|
| <input type="checkbox"/> Component names | <input checked="" type="checkbox"/> Deep paths |
| <input checked="" type="checkbox"/> Component text | <input type="checkbox"/> Geometry |

Recording Options

☒ Use context menus for filters and assertions
☐ Write Traffic to Disk
☒ Compress recording files

Capture Level

☐ Capture DOM Level 2
☐ Verbose HTML recording
☐ Ignore HTML responses
☐ Capture HTML changes
☐ Capture Applet snapshots
☐ Capture ActiveX snapshots

Capture Diff Size Bytes
 Capture Max Time ms

Capture DOM Events

| | |
|--|---|
| <input checked="" type="checkbox"/> navigate | <input checked="" type="checkbox"/> docload |
| <input checked="" type="checkbox"/> focus | <input checked="" type="checkbox"/> dblclick |
| <input checked="" type="checkbox"/> mousedown | <input checked="" type="checkbox"/> change |
| <input checked="" type="checkbox"/> mouseup | <input checked="" type="checkbox"/> contextmenu |
| <input checked="" type="checkbox"/> mouseover | <input checked="" type="checkbox"/> drag/drop |
| <input type="checkbox"/> mouseout | <input checked="" type="checkbox"/> mousemove |
| <input checked="" type="checkbox"/> click | <input checked="" type="checkbox"/> keypress |
| <input checked="" type="checkbox"/> open/close | <input type="checkbox"/> ajax callback |

For help click on the ? icon in the top-right corner, then on a setting's label.

Save Cancel

In the "Or use the following locator javascript function" box, click on the check box to enable it.

Enter "\$adf" in the drop down list.

Click **Save** to save these settings.

7. Running Browser Standalone

7. Running Browser Standalone

While taking advantage of the full power of LISA (multi-technology, Data Sets, Companions, etc.) requires running embedded in LISA, doing some quick testing or debugging can be achieved using the LISA browser in standalone mode.

To do this, run the **lisa_browser.exe** executable in the **%LISA_HOME%/bin/browser** directory.

```
lisa_browser.exe -s true [-m <recorder|playback|service>][-f <recording file>]
```

-s or **-standalone** should be true to run outside of LISA.

-m or **-mode** can be recorder, playback or service (service allows remote control through a web service interface)

-f or **-file** optionally specifies a recording file saved previously from the standalone recorder.

8. Troubleshooting

8. Troubleshooting

Here are a few guidelines that will help you and help us assist you in the process of submitting a ticket, from asking a simple question up to reporting a severe bug.

In some cases those tips may dramatically reduce the time to resolution.

- Does the scenario you're testing work in a browser (especially in IE since it is used for recording)? If not there is no chance it will work in the LISA browser.
- Have you read the documentation and made sure your questions are not answered therein? (in particular see [the troubleshooting section](#))?
- Have you familiarized yourself with the different settings in the browser and checked they couldn't address the problem? E.g. Applets not being recorded because applet support was disabled in the settings, or a site that uses mouseover events isn't replaying correctly because mouseover events are unchecked, or timeouts are too low for your applications, etc. (See [Web 2.0 Settings](#))
- Are you sure the test is not replaying correctly because of a lack of parametrization? Specifically, have you looked at the debug output window (or logs) to see what went wrong during replay (e.g. "could not find element "//DIV[@id='gen542348239]" because the id is dynamic)? For example see the How To for [Dynamic Elements](#).
- Are you providing all the information that we're likely to ask for? If the problem triggers an error dialog, you will have an option of "Generating an Error Report", which will contain all this required information (and you can send that to us). Otherwise you can look it up in the Settings dialog (Lisa Browser build number, OS version, IE version, .NET version, JRE version if applicable)
- Can you reproduce the problem consistently? Does it happen only when driven from TestManager or even when using the browser standalone?
- Is there a public URL where you can reproduce the problem? If yes let us know about it. If not, can you package the pages that are giving you trouble and send them to us (usually this can be accomplished by navigating to a browser and going to the File menu and selecting Save As).

If all of this yields nothing useful, we will probably need to contact support@itko.com and have a video conference call (like webex).

NOTE - Please get your system ready to show the wrong behavior to get the correct solutions.

9. Known Limitations

9. Known Issues and Future Work

In no particular order:

- Create an object repository to increase resistance to application change.
- Add the ability to generate test scripts.
- Improve usability, especially around the filters and assertions screens.
- Improve the screenshot algorithm used to capture applets and remote applications images at certain times.
- Ability to use the "Capture HTTP Traffic" mode to replay the tests at the HTTP level if desired (a la web 1.0).
- Full Safari support.
- Various bug fixes

PART 2 - LISA Web 2.0 - How Tos

PART 2 - LISA Web 2.0 - How Tos

1. Introduction
2. Web sites and frameworks
3. How-To: Generate random data (4.5.1.x)
4. How-To: Capture Dynamic HTML for later test editing
5. How-To: Deal with time-sensitive events
6. How-To: Parametrize dynamic data entry in loops
7. How-To: Deal with dynamic elements
8. How-To: Extract complex data from a page
9. How-To: Ajax auto-complete fields
10. How-To: Write custom Web 2.0 steps
11. How-To: Write cross-browser tests
12. How-To: Use Pathfinder integration
13. How-To: Write Java Swing and WebStart tests
14. How-To: Write .NET WinForms tests
15. How-To: Debug a test
16. How-To: Use global filters and global assertions
17. How-To: Interact with external resources
18. How-To: Run Load Tests
19. How-To: Run in a non-privileged account or on 64 bit platforms
20. How-To: Record and replay against non us-english websites
21. How-To: Run in Crash Dump mode

1. Introduction

1. Introduction

This document is a simple list of how-to's that cover typical scenarios and how to deal with them. You should first read [the user guide](#) to familiarize yourself with the basic concepts.

2. Web sites and frameworks

2. Web sites and frameworks

Most web 2.0 sites built today use one or more of many available ajax frameworks, which is where the complexity lies. For a good source of what those frameworks are today, you can consult ajaxpatterns.org. The following frameworks or websites using a framework have been tested during development:

- [Ext](#) (see the demos at [Ext JS](#) and [Ext GWT](#))
- [ICEfaces](#) (see the demos as [Components showcase](#))
- [jQuery](#) (see [Kayak](#))
- [Appcelerator](#) (see the demo at [Appcelerator Web Unit Tests](#))
- [Tibco GI](#) (see the demos at [Xignite](#))
- [Oracle ADF Faces](#)
- [Backbase](#) (see the demos at [Backbase Demos](#))
- [ASP.net Ajax](#) (see the Telerik demos at [Telerik](#))
- [Bindows](#) (see the demo on the home page)
- i2 websites (MDM, SCP, etc...) - no public URL available
- and many others...

If there are any websites or frameworks that don't work in the DOM browser, please let me know and I will add support for it. There is nothing that can't be supported (in theory), the DOM browser even works with DOM level 2.

3. How To - Generate random data (4.5.1.x)

3. How To - Generate random data (4.5.1.x)

There are many places in LISA where string generation patterns may be specified. By using a pattern, LISA will create a random string based on the specified pattern during the run of a test model. A string pattern is made up of a mix of pattern and literal characters that will form the final string. The following are the recognized pattern characters:

- D – replace with a random digit (0-9)
- L – replace with a random capital letter

l – replace with a random lower case letter
A – replace with a random digit or letter of either case
P – replace with a random punctuation character (.,-V)
.- replace with a random printable character

So, for example, the pattern "LDDD" will create a string with a random uppercase letter followed by three random digits. To use a pattern in the LISA browser, the following syntax may be used: =pattern (e.g. =LDDD).

Filter Key: random [Browse]

☐ Wait up to 0 ms for value Equals

Properties
An expression filter allows you to combine other filters or literals.

☐ DOM Element [Browse]
☐ Text [()]
☐ DOM Attribute [Browse]
☐ Script [Browse]
☒ Expression {{=AAAADDDD}} [Browse]
☐ Cache All
☐ Capture [Browse]
☐ Sleep 1000 ms
☐ Browser ☐ Internet Explorer ☐ Firefox ☐ Safari

Quick Test
Click Evaluate to see what value this filter would return if it were evaluated during the recording.

Recorded Value: STtj0744 [Evaluate] [Clear All]

A square bracket expression may be used to randomly select from a specific list. For example, the pattern "[1,2,3]" will create a single character string that is either "1", "2" or "3".

Any character that is not "D", "L", "I", "A", "P", ".", "[", "{", "}" or "*" is taken as a literal. For example, the pattern "BobDDD" will generate 6-character strings, all of which start with "Bob" and end with 3 random digits. If you need one of the "reserved" pattern characters as a literal, prefix it with a back-slash. For example, the pattern "DD\D" will generate strings with two random digits followed by the literal character "D".

With any of the constructs above, there are two types of modifiers that may be specified. A brace expression, "{}", may be used to specify an exclusion list. For example, the pattern "D{0,2,4,6,8}" may be used to generate a random digit that will only ever be an odd number.

An asterisk expression may be used as shorthand for repeating pattern characters. The asterisk must be followed by one or two numbers surrounded by parentheses. If two numbers are specified, they must be separated by a comma, dash or space.

When the repeater (asterisk) expression specifies only one number, the result of the pattern will be a string of the specified number of characters. For example, the pattern "A*(20)" will generate a 20-character string composed of random alpha-numeric characters.

When the repeater expression specifies two numbers, the result will be a string that is at least the first number in length but no longer than the second. For example, the pattern "L*(3-5)" will generate strings of random capital letters at least 3 characters, but no more than 5 characters long.

4. How To - Capture Dynamic HTML for later test editing

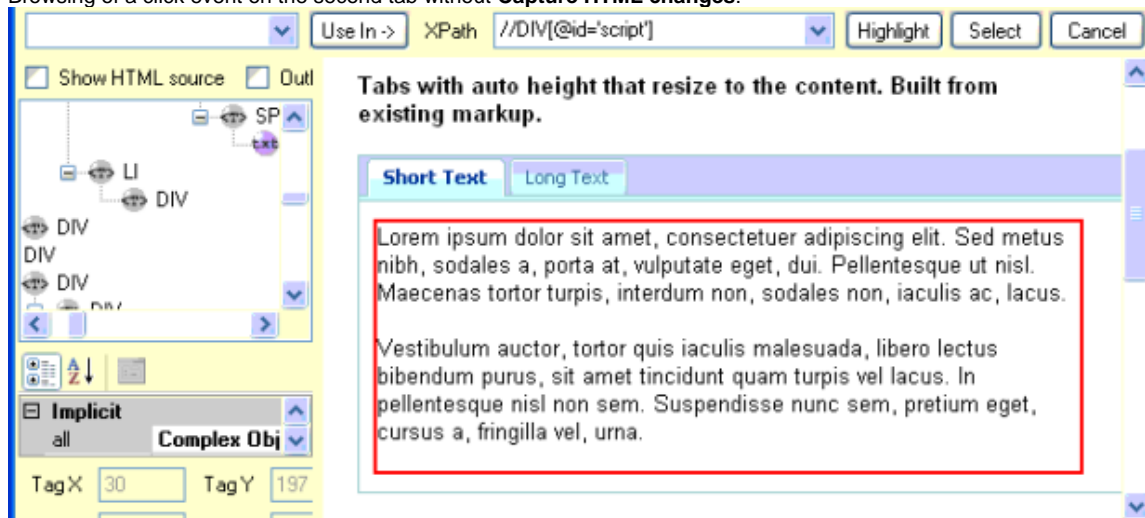
4. How To - Capture Dynamic HTML for later test editing

By default, when you record a test, the response (html document) is captured only when a new document is loaded in the browser, either through a direct navigation or as a result of an ajax load (if ajax callback is checked in the recording settings) so all the events between 2 page loads will have the same response (as you can see by looking at the events response tab).

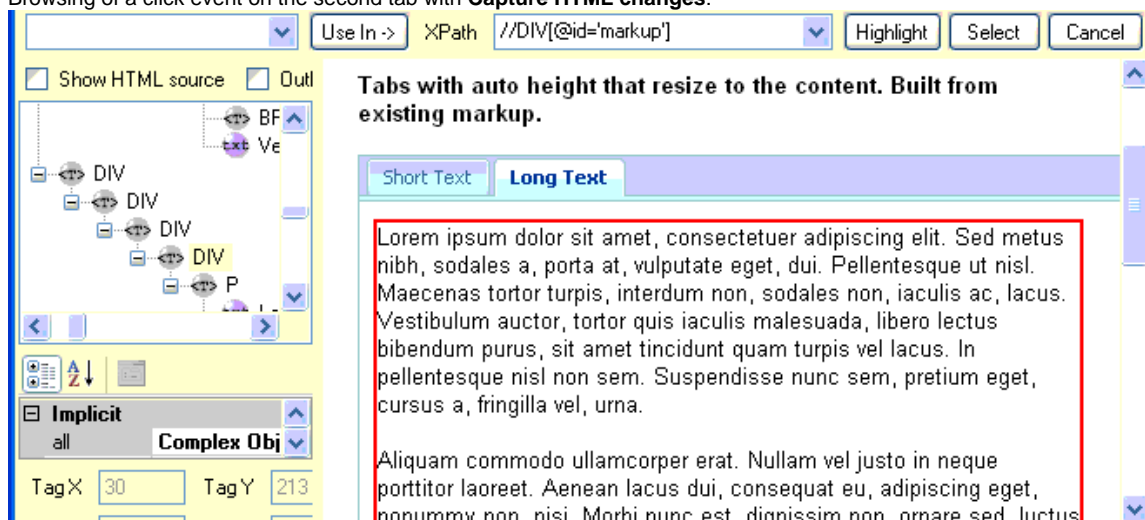
It is normally sufficient and keeps the test size relatively small but for some websites that are very javascript heavy, where the page changes a lot purely on the client as a result of javascript executions, it can make it difficult to edit the test or add filters and assertions through browsing.

For example if you have a page that has tabs (that are preloaded and just change visibility when you click them) and you click on the second tab and want to add a filter for some text on the second tab, if you browse on the events tab you will only ever see the first tab:

Browsing of a click event on the second tab without **Capture HTML changes**:



Browsing of a click event on the second tab with **Capture HTML changes**:



If you add **Capture HTML changes**, what happens is the response is captured on more types of events such as change or clicks, which will make the page as it looked at the time of the event available to you when you browse.

☒ Capture HTML changes

Capture Diff Size Bytes

Capture Max Time ms

Of course if that was the end of the story this would dramatically increase the test size, so to avoid this, there is another setting which is **Capture Diff Size**. What this does is that it compares the size of the change between the page as it is now and the page as it was when it loaded, and if the size of the change is above the number you specify in the settings, it will capture the whole page as it is now, but when the size is below that number it will only keep track of the diff, which is usually quite small and the response for the event is automatically rebuilt for the user from the page load response and the diff.

The reason it doesn't always do the diffs is because above a certain size they become very expensive to compute and would take too long (which is where the second setting, **Capture Max Time** comes in).

The defaults are set to reasonable values and shouldn't need to be changed in general. So if test size is not a concern, it's nice to have this checked to make it really easy to edit the steps/filters later on.

5. How To - Deal with time-sensitive events


5. How To - Deal with time-sensitive events

A lot of what makes web 2.0 sites difficult to test is the asynchronous nature of their interactions with the user. In a pure request/response environment, there are no race conditions or multiple threads of execution to worry about (at least from the client's perspective), but web 2.0 environment introduce those difficulties everywhere: frames, set Timeout and particularly asynchronous ajax calls in html pages as well as any java applet code or flash/flex code make heavy use of multiple threads of execution.

The main testability problem it creates is unpredictability. The application may be in a different state as any given event executes during two runs. If you need to capture a value on the page that is updated asynchronously, when do you do the capture? Right after the event, or a fixed amount of time after it? For example, when do you capture the price of a dynamically configured item as in the picture below?

SELECT MY PROCESSOR

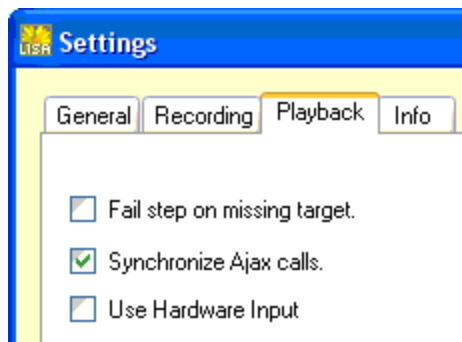
-  [Help Me Choose](#)
- ☐ Intel® Celeron M® M550 (2.0GHz, 1M L2 Cache, 533MHz FSB)
[subtract \$240]
 - ☐ Intel® Core™ 2 Duo T5670 (1.8GHz/800MHz FSB/2MB cache)
[subtract \$225]
 - ☐ Intel® Core™ 2 Duo T5870 (2.0GHz/800MHz FSB/2MB cache)
[subtract \$150]
 - ☒ Intel® Core™ 2 Duo T8100 (2.1GHz/800MHz FSB/3MB cache)
[Included in Price]
 - ☐ Intel® Core™ 2 Duo T9300 (2.5GHz/800MHz FSB/6MB cache)
[add \$250]

 [Go to Next Component](#)



The DOM browser gives you 2 powerful tools to deal with those situations:


- The **Synchronize Ajax calls** setting



This will force all ajax calls made by the application to be executed synchronously so no event, filter, or assertion will execute until the event that triggered the completion returns (note: some websites, that do ajax event pushes through regular polling, are not working well with this setting. If you don't know how the application is coded, just give it a try and see if it works or if it seems to freeze the application from time to time). This is especially useful when there is no visual cue as to when the code completed, as in the example pictured above.

- The **wait for** option of any filter.

Definition

 A filter is a function that executes before or after an event is triggered and stores its result (Filter Value) in a variable (Filter Key).

Filter Key: [Browse](#)

☒ Wait up to ms for value

This will make the test execution wait until either the wait for condition is met, or the timeout specified expires. You can see an example of how to use this in the [Autocomplete](#) section. Another typical use of this is for pages that have splash screens. For instance, if the splash screen is a div containing the text "Loading..." you would pick a text filter with DOM element the splash div (which you can identify by adding a quick filter while it's shown onscreen for example) add a "wait for" condition with an operator of "Does Not Match" and a value of "Loading...".

6. How To - Parametrize dynamic data entry in loops

6. How To - Parametrize dynamic data entry in loops

To understand this example, you should first be familiar with XPath expressions, as described in [Web 2.0 XPath](#).

Let's look at a couple of typical scenarios to understand how to deal with dynamic data in a loop.

First, let's say you have a list of cities in a table with a checkbox next to them, and you want to check the checkboxes.

| | |
|---|----------------------------------|
| <div><input type="checkbox"/> Austin</div> <div><input type="checkbox"/> Dallas</div> <div><input type="checkbox"/> Metropolis</div> <div><input type="checkbox"/> Gotham</div> | The code for a table row here is |
|---|----------------------------------|

```
<tr>
  <td><input type=checkbox></td>
  <td>Metropolis</td>
</tr>
```

If we want to check them all it is very simple, we record the clicking of just one of them which will result in a change event with an XPath value like `/HTML/BODY/TABLE/TBODY/TR[1]/TD[1]/INPUT`.

So first we define a looping variable. The easiest way to do this is create a filter of type Expression on an event before the change event, let's call it `i` and set its value to 0.

Then we parametrize the XPath of the change event to be something like: `/HTML/BODY/TABLE/TBODY/TR[i]/TD[1]/INPUT`. To increment the value of `i`, we can create another expression filter on the change event with key `i` and value `i + 1`. Each time this filter executes this will increment the value of `i` by 1.

Finally we need to loop onto the change event until `i` is greater than the number of rows (in this example 4, but we could dynamically compute the number of rows first), so we add an assertion on the change event whose expression reads `If i Less Than 5 Then "Go To change event"`. That's it.

Alternatively, if you are coding inclined, you can simply create a javascript step whose DOM Element is the `/HTML/BODY/TABLE` containing the cities and with the following script: `"for (i=0; i<4; i++) _arg.rows[i].cells(0).childNodes(0).checked=true; return 0;"`.

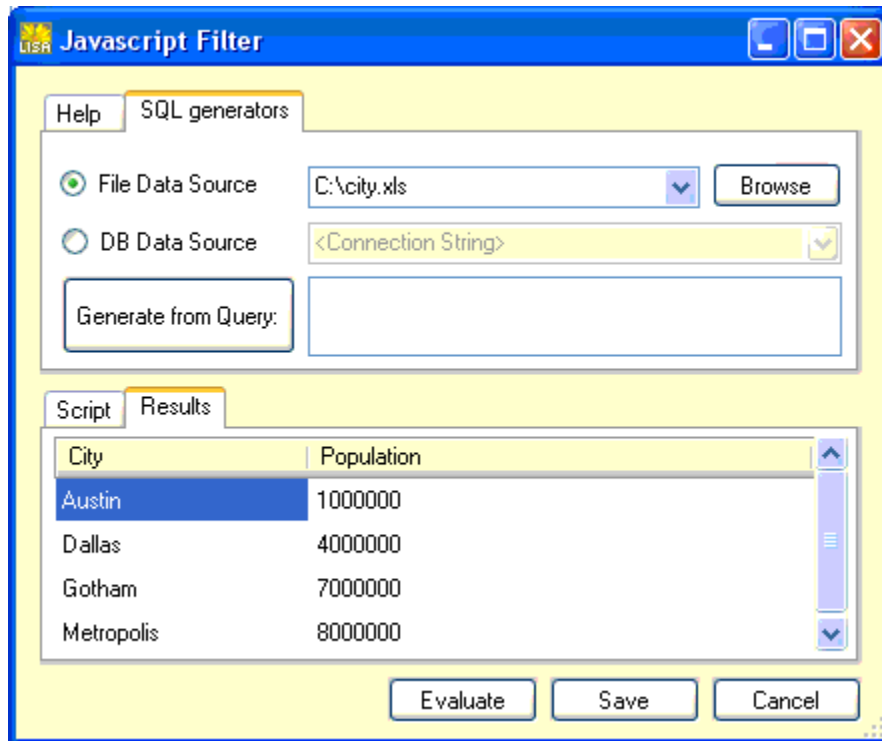
Ok, now for something a little more complicated, let's say we have a list of cities coming from a Dataset along with population number and we want to input those values on the page:

| <div><table><thead><tr><th>City</th><th>Population</th></tr></thead><tbody><tr><td><input type="checkbox"/> Austin</td><td><input type="text"/></td></tr><tr><td><input type="checkbox"/> Dallas</td><td><input type="text"/></td></tr><tr><td><input type="checkbox"/> Metropolis</td><td><input type="text"/></td></tr><tr><td><input type="checkbox"/> Gotham</td><td><input type="text"/></td></tr></tbody></table></div> | City | Population | <input type="checkbox"/> Austin | <input type="text"/> | <input type="checkbox"/> Dallas | <input type="text"/> | <input type="checkbox"/> Metropolis | <input type="text"/> | <input type="checkbox"/> Gotham | <input type="text"/> | The code for a table row here is |
|---|----------------------|------------|---------------------------------|----------------------|---------------------------------|----------------------|-------------------------------------|----------------------|---------------------------------|----------------------|----------------------------------|
| City | Population | | | | | | | | | | |
| <input type="checkbox"/> Austin | <input type="text"/> | | | | | | | | | | |
| <input type="checkbox"/> Dallas | <input type="text"/> | | | | | | | | | | |
| <input type="checkbox"/> Metropolis | <input type="text"/> | | | | | | | | | | |
| <input type="checkbox"/> Gotham | <input type="text"/> | | | | | | | | | | |

```
<tr>
  <td><input type=checkbox></td>
  <td>Metropolis</td>
  <td><input type=text></td>
</tr>
```

Here the technique is different because instead of iterating based on the elements on the page, we iterate on a dataset and look up the

corresponding elements on the page. Let's first create a web 2.0 dataset using a script filter:

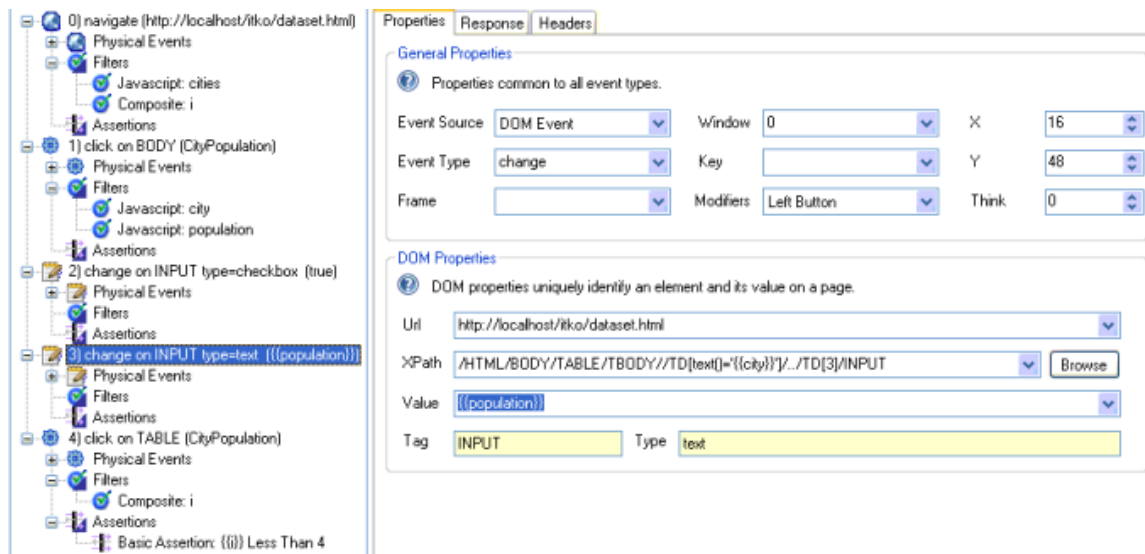


The process is very similar to the one we just followed: first we record checking one checkbox and entering a value in the corresponding text field. It can be any box, the goal is just to generate 2 changes events, one for a checkbox, one for a text field.

Then we create the `i` looping variable before any of the two change events and we parametrize the XPath of those change events. To do this we can no longer rely directly on the index but instead on the values coming from the dataset. To make things clearer, we can define a couple of script filters called `city` and `population` with the following scripts:
`return cities.rows(i).cells(0)` and `return cities.rows(i).cells(1)`.

Now we can use those variables in the XPath on the 2 inputs by first identifying the city name cell: `/HTML/BODY/TABLE/TBODY//TD[text()='city']` and then navigating the DOM from there, which gives us: `/HTML/BODY/TABLE/TBODY//TD[text()='city']/../TD[1]/INPUT` and `/HTML/BODY/TABLE/TBODY//TD[text()='city']/../TD[3]/INPUT`. At the same time, we parametrize the Value of the second change event to use the value coming from the dataset: `population`.

Finally we add the filter that increments `i` and the assertion that loops onto the first change events while there are more rows in the dataset. The screenshot below shows what the test case looks like after adding those filters and assertions.



All of this can also easily be accomplished in a code way as follows:

```

for (i = 0; i < cities.length; i++)
{
city = cities.rows[i].cells(0);
population = cities.rows[i].cells(1);

lisa.select(document,"/HTML/BODY/TABLE/TBODY//TD[text()='\" + city + "\']/../TD[1]/INPUT").checked=true;
lisa.select(document,"/HTML/BODY/TABLE/TBODY//TD[text()='\" + city + "\']/../TD[3]/INPUT").value=population;
}

return 0;

```

7. How To - Deal with dynamic elements

7. How To - Deal with dynamic elements

To understand this example, you should first be familiar with XPath expressions, as described in [Web 2.0 XPath](#).

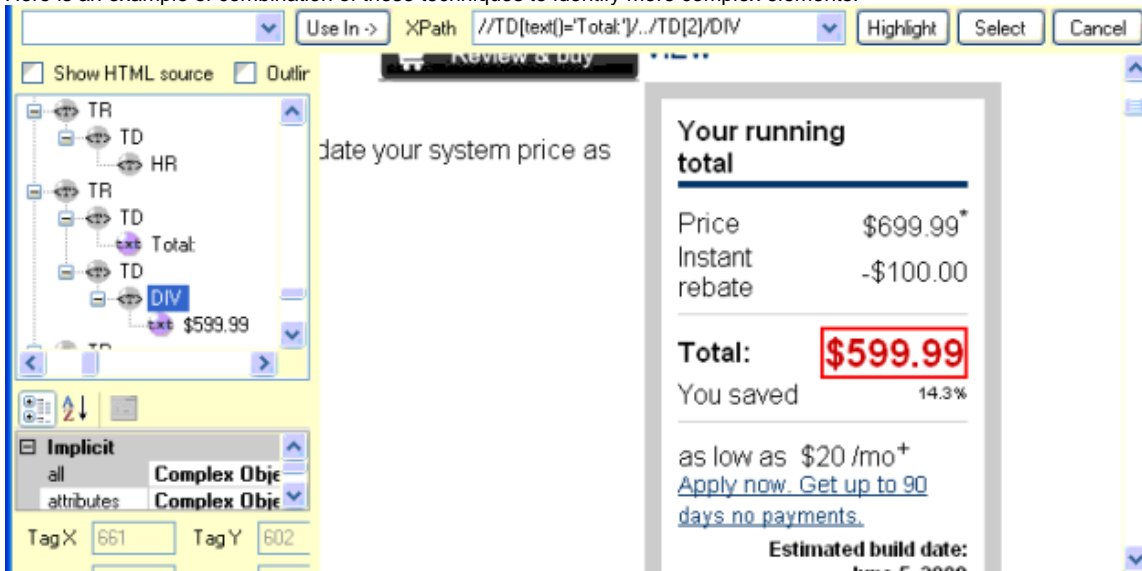
One of the main problems facing recording/replay tools is that of identifying elements on a page or in a control using various properties of the element, such as html attributes, text or value, position in the DOM, position in the page, etc...or any combination of those. Most of these tools boast various algorithms to optimize identification without getting false positives, and the DOM browser is no exception.

The primary means of identifying an html element on the page is through [XPath](#). Those XPath expressions are automatically generated but can also be modified or parametrized to improve reliability in some cases.

In the vast majority of cases, the **id** attribute of html elements is a reliable way to identify them so it will be used whenever possible. However, those ids are sometimes dynamically generated and will change from run to run, sometimes for good reason (as in websites that have dynamic layout), often as a result of poor coding. To avoid having to manually parametrize all these xpaths to make them more reliable, there is a recording setting, **Exclude ids matching**, which will automatically prevent ids whose value matches the supplied regular expression to be used during recording. The rationale is that dynamic ids frequently follow a certain pattern, such as having multiple digits (and the default pattern is indeed 3 digits or more).

Generally, other typical ways to identify elements such as text or value are easily expressible as xpath as well, using the syntax `//TAGNAME[@attributeName='value']` e.g. `//A[@href='www.google.com']`, or `//DIV[text()='Some Value Here']`

Here is an example of combination of these techniques to identify more complex elements:



In the picture above, if the DIV element containing the price has a dynamic id (i.e. it changes from run to run), we don't want to use it. So we Browse and pick the closest element we know doesn't change, in this case the TD whose text is "Total:". Then we navigate up one level using the `..` operator, and then back down a couple of levels using the DOM tree as a guide. This gives us the following xpath: `//TD[text()='Total:']/../TD[2]/DIV`. The Highlight button confirms our guess.

8. How To - Extract complex data from a page

8. How To - Extract complex data from a page

To understand this example, you should first be familiar with XPath expressions, as described in [Web 2.0 XPath](#).

Most interesting assertions rely on data that has been extracted from pages through the use of filters. The vast majority of cases will require no configuration and will automatically pick up the data you want:

Element filters will select the desired element (to verify it exists for example) and fill the filter value with its xpath. The element will be already chosen for you if you use a filter add during recording (Add Quick Filter or Add and Edit Filter). If you do it post recording, the browse button will easily let you do it visually.

Text and Attribute filters work the same way, there is no work required other than deciding on a regular expression for Text filters (which 95% of the time will be the default (.*) to capture the whole inner text), and as for attributes, the list of available attributes will be prepopulated in the drop-down after an element browse is completed.

Let's take a simple example: we add a user to a list of users and it shows up in a table with ID, Name and Email. We want to make sure that the user has in fact been added and that the email field value is what we think it is:

Wednesday, May 28, 2008

Welcome iTKO !

Add User

Modify User

Delete User

List Users

| UserID | Name | Email |
|-----------------------------------|---|----------------------|
| userC0A80AC8000.. | userC0A80AC80000011A5622E.. | null |
| multitier-43784.. | null null | null |
| multitier-64966.. | null null | null |
| User | null null | null |
| multitier-10045.. | null null | null |
| Value | null null | null |
| userC0A80078000.. | null null | null |
| userC0A80078000.. | null null | null |

We can first add a DOM filter, and browse for the row we want in the table. We'll get something like:
//FORM[@name='edit_users']/TABLE/TBODY/TR[493]/TD[1]/A.

Use In -> XPath //FORM[@name='edit_users']/TABLE/TBODY/TR[493]/TD[1]/A Highlight Select Cancel

| | | |
|-----------------------------------|---|---------------------------|
| hello | null null | null |
| test | null null | null |
| test2 | null null | null |
| idid | jd jd | id@id.com |
| user62326161616.. | user62326161616633312D666.. | |

Of course, this is not reliable because the row number (493 in this instance) could change, so we need to parametrize this expression to use the user name instead. We keep as much of the computed expression as possible, in this case

//FORM[@name='edit_users']/TABLE/TBODY

and then we look for the A element whose text is the user name, something like
A[text()='jdjd'].

This gives us the following expression: //FORM[@name='edit_users']/TABLE/TBODY//A[text()='jdjd']. Note the double // before the A to specify we search children of the TBODY down to any level.

All we have to do now is add an assertion that verifies the value of this filter is not blank, which will indicate the presence of this row in the table. In the same way, to capture the email of this user, we use a text filter where we start with the same expression:

//FORM[@name='edit_users']/TABLE/TBODY/TR[493]/TD[1]/A

then we go up two levels to the table row:

//FORM[@name='edit_users']/TABLE/TBODY/TR[493]/TD[1]/..

then we pick the anchor in the third cell of this row:

//FORM[@name='edit_users']/TABLE/TBODY/TR[493]/TD[1]/../TD[3]/A

At any time we can use the Highlight button to verify we're selecting the right element.

Use In -> XPath //FORM[@name='edit_users']/TABLE/TBODY//A[text()='jdjd']/../TD[3]/A Highlight Select Cancel

| | | |
|-----------------------------------|---|---------------------------|
| test | null null | null |
| test2 | null null | null |
| idid | jd jd | id@id.com |
| user62326161616.. | user62326161616633312D666.. | |
| user7F000101000.. | user7F0001010000011A2D5E4.. | |

Now we just have to add an assertion to make sure the value of this filter is indeed the email we think. Of course, all of these values, such as user name, email etc...can be parametrized in the usual way username, email, etc...

For the cases when you need finer-grained control over the retrieved data, you can use the Script Filters. They give you full control over the DOM (or the applet hierarchy when testing applets). As an illustration, let's reuse the previous example, where we want to extract data from the table.

The first thing you would do in a filter is select the DOM element to be the table (by browsing as usual). This gives you access to the table object in script as the `_arg` variable (see the `_arg` scripting object). Then you can easily retrieve all kinds of information:

| | |
|---------------------------------------|---|
| Number of rows: | <code>return _arg.rows.length</code> |
| Text of the 3rd cell of the 12th row: | <code>return _arg.rows(11).cells(2).innerText</code> |
| Text of the last row: | <code>return _arg.rows(_arg.rows.length - 1).innerText</code> |
| Email of the user jdjd: | <code>for (i=0;i<_arg.rows.length;i++) {if (_arg.rows[i].cells(0).innerText=='jdjd') return _arg.rows[i].cells(2).innerText;}; return null;</code> |
| Etc... | |

Properties
 A script filter executes a javascript function (that must return a value) on the HTML document of the event response.

☐ DOM Element

☐ Text

☐ DOM Attribute

☒ Script

☐ Expression

☐ Cache

Quick Test
 Click Evaluate to see what value this filter would return if it were evaluated during the recording.

Recorded Value

Finally, some dynamic data is automatically extracted for you, most notably request string parameters and cookies:

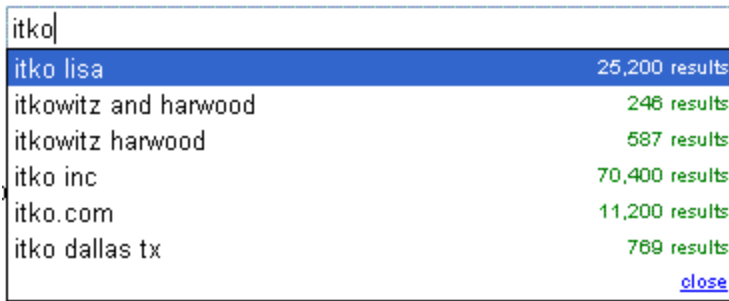
| Key | Value |
|-----------------------------------|--|
| {{event.cookie.NID}} | 15 |
| {{event.cookie.PREF}} | ID |
| {{event.frame.path}} | |
| {{event.param.aq}} | f |
| {{event.param.hl}} | en |
| {{event.param.oq}} | itko |
| {{event.param.q}} | itko |
| {{event.path}} | //DIV[@id='ads']/OL/LI |
| {{event.response.render.time.ie}} | 47 |
| {{event.response.render.time}} | 47 |
| {{event.response.time}} | 15 |
| {{event.response}} | <HTML><HEAD><TITLE>itko - Google Search</TITLE><META http-equiv=content-type content="text |
| {{event.script.error}} | |
| {{event.status.code}} | 200 |
| {{event.type}} | mouseover |
| {{event.url}} | http://www.google.com/search?hl=en&q=itko&aq=f&oq= |

As shown in the picture above, string parameter names are prefixed with `event.param` and cookie names are prefixed with `event.cookie`.

9. How To - Ajax auto-complete fields

9. How To - Ajax auto-complete fields

One of the most pervasive ajax controls is the so-called auto-complete (or type-ahead) control, which prompts the user with a list of choices even as they are typing letters in a field.



To record and replay this type of interaction, there are a couple of things to be aware of. First, the keypress event should be enabled for recording (as it is by default) in the settings. In most cases it can be disabled and it leads to leaner test cases but in this case it is needed.

Then, when you replay the test, the drop-down is populated asynchronously, so if you take no precautions, you may execute the next event or a filter, or an assertion before the drop-down is populated. There are 3 ways to deal with that, as explained in the [Time Sensitive](#) section:

- Using the **Synchronize Ajax calls** setting.
- The next way that is very simple is to add a filter that waits for a certain amount of time by specifying a never met condition, thus giving the call enough time to complete.

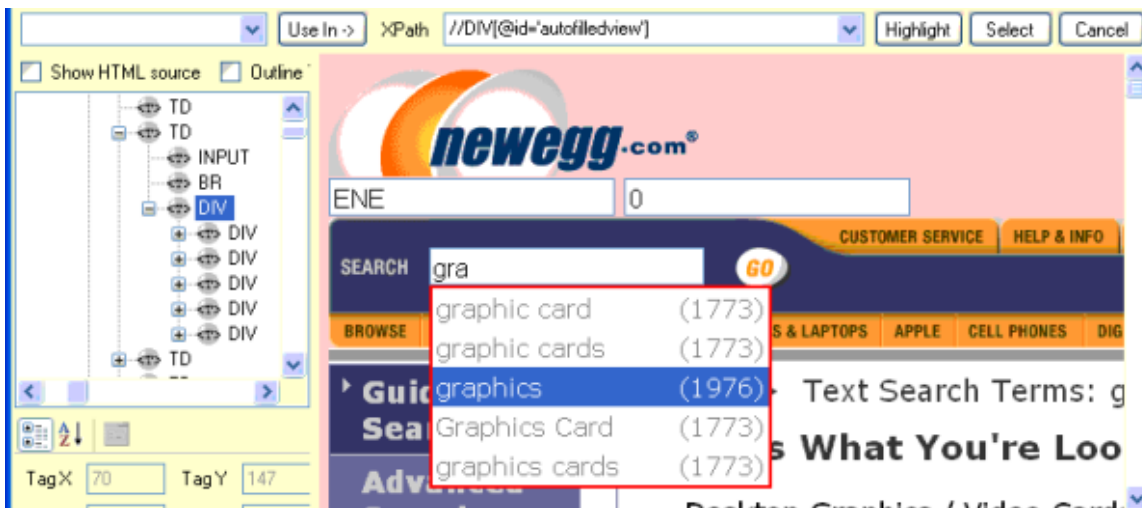
Definition
 A filter is a function that executes before or after an event is triggered and stores its result (Filter Value) in a variable (Filter Key).

Filter Key:

☒ Wait up to ms for value Equals

- The last way, which will always work but requires a bit more work is add a filter that waits for a specific condition to be fulfilled. In this case, the condition would be that the drop-down is visible and shows a certain number of rows for instance.

Typically, you would add a script filter on the event that triggered the drop-down, then browse for the DOM element and pick the drop-down popup, as illustrated in the picture below (from the search on www.newegg.com).



Then you would specify the script snippet to return something meaningful for this site. For instance, in this case, you can see from the picture above that the drop down is a div tag and each row is made up of a child div tag (easy to see from the DOM tree). So a meaningful filter script might return the number of rows: `return _arg.getElementsByTagName("DIV").length` (see the [_arg](#) scripting object).

Finally, since you expect the filter value to be 5 in this instance, you would specify in the Wait value box of the filter, and the filter screen would look like this:

Definition

☐ A filter is a function that executes before or after an event is triggered and stores its result (Filter Value) in a variable (Filter Key).

Filter Key:

☒ Wait up to ms for value

Properties

☐ A script filter executes a javascript function (that must return a value) on the HTML document of the event response.

☐ DOM Element

☐ Text

☐ DOM Attribute

☒ Script

☐ Expression

☐ Cache

Quick Test

☐ Click Evaluate to see what value this filter would return if it were evaluated during the recording.

Recorded Value:

As you can see from the picture, the Quick Test can be used to tell you if the script you're using is correct and returns the desired result. Also, note that to see the drop down visually in the Browse window, as is pictured above, you will need to enable the **Capture HTML changes** setting in the recording settings as explained in the [Capture section](#).

You are now ready to add more filters for data extraction and assertions for validations. This is more thoroughly covered in the [Data extraction](#) section but in this instance, a very simple filter might be a Text filter with a DOM element set to be the drop down element, which will retrieve all the text contained in the popup:

Properties

☐ A text filter retrieves the inner text of a DOM element using a regular expression (first capturing group).

☐ DOM Element

☒ Text

☐ DOM Attribute

☐ Script

☐ Expression

☐ Cache

Quick Test

☐ Click Evaluate to see what value this filter would return if it were evaluated during the recording.

Recorded Value:

Then all that is left to do is add an assertion that checks some given text is contained in the value of this filter (using a Basic Match assertion).

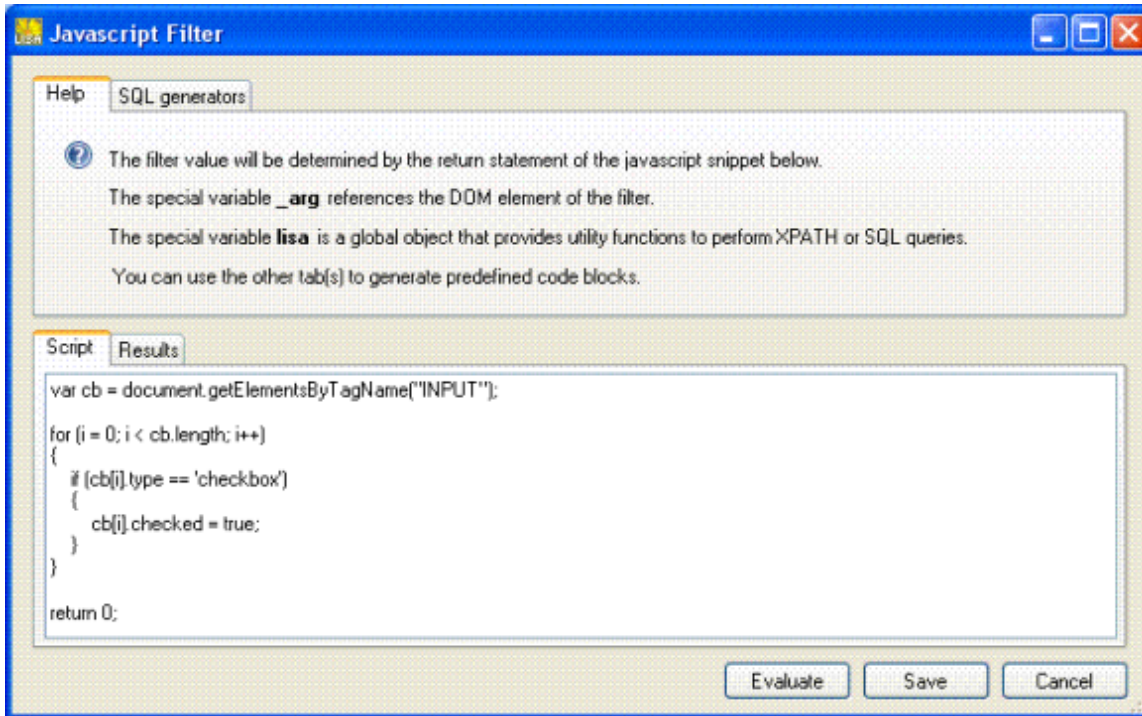
10. How To - Write custom Web 2.0 steps

10. How To - Write custom Web 2.0 steps

Many record/replay tools let you (or even require you to, if there is no recorder) write scripts to control user actions. There should generally be no need to do this as the recorder is good at capturing all user interactions, but it may still be useful in certain cases.

For instance, if you have a page with many checkboxes that you want to check, the GUI way to do this would be to check one, and then to parametrize it so that it can be embedded in a loop that will visit all the checkboxes with the change event (see the [parametrization](#) section). If you

have 2 levels of loops or other constraints it will be quite tedious, but it could be quite easy to do in a script.



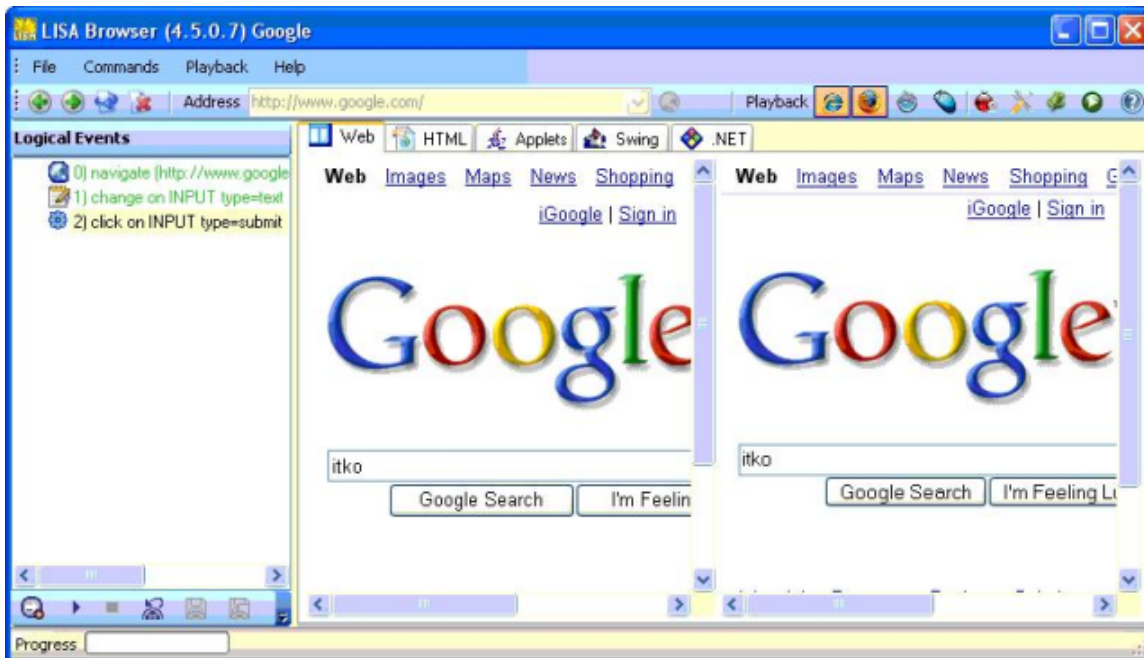
When you create a new step in the Events tab, a new event of type script gets created. It does nothing by itself but contains a script filter that can execute arbitrary javascript (or java for applets). The image above shows the script dialog that gets opened when you click the Browse button of a script filter. In this instance, it iterates over all checkboxes on the page and checks them.

11. How To - Write cross-browser tests

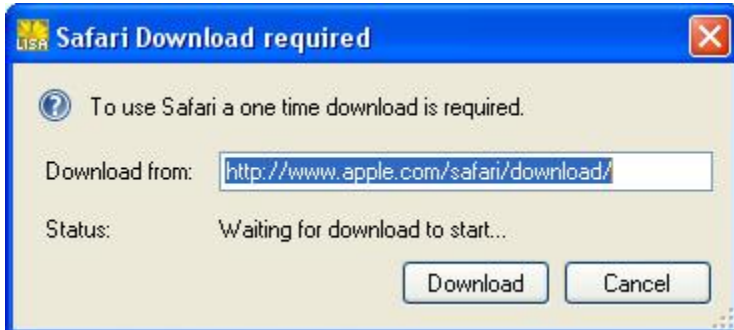
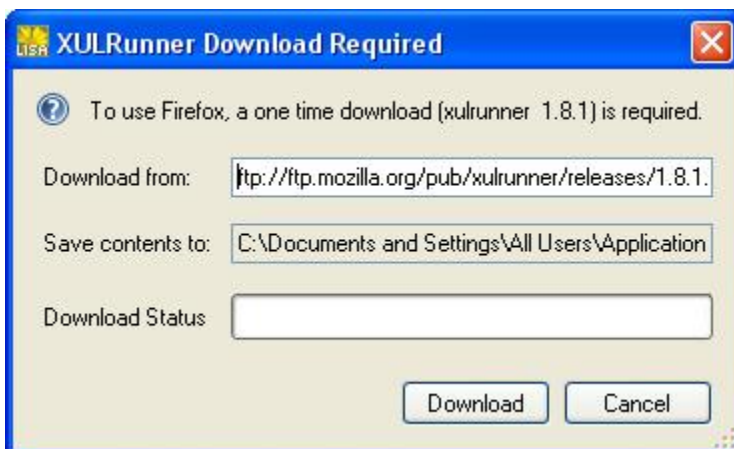
11. How To - Write cross-browser tests

The first thing to know about writing tests that run in multiple browsers is that there is nothing special to know. You would author, record and edit the test in the exact same way you normally would a test designed to run in a single browser, and during playback you simply select which browser(s) you want to run the test.

In the screenshot below, note how the Internet Explorer and Firefox buttons are pressed in the toolbar. This will cause both browsers to be used for this test. You can select Internet Explorer and/or Firefox and/or Safari to run a test.



While Internet Explorer is always installed on Windows, other browsers require an initial download. Every time you attempt to use Firefox or Safari, if those browsers have not been installed you will get the following prompts:



There is no difference in how events are executed in multi-browser mode, they are just executed for each browser. Assertions are just executed once since they don't depend on the browser, the only thing to pay attention to is filters because they highly depend on the browser.

First off, we can not execute filters for each browser without any changes because if we did, the filter value coming from one browser would overwrite the filter value coming from an other one since they have the same key.

This is why when multiple browsers are used, the keys used in filters are still in use but there are automatically new filter keys being generated, one for each browser, to allow us to distinguish between the filter values coming from different browsers. The convention is to append 2 letters to the filter key: **key.ie** (for Internet Explorer) or **key.ff** (for Firefox) or **key.wk** (for Safari).

The screenshot below shows what happens when we define a javascript filter called allcount with code: "return document.all.length;"

| Key | Value |
|-----------------------|---|
| {{allcount.ff}} | 93 |
| {{allcount.ie}} | 99 |
| {{allcount}} | 93 |
| {{event.path}} | /INPUT[@name='q'] |
| {{event.response}} | <HTML><HEAD><TITLE>Google</TITLE><META http-equiv=content-type content="text/html |
| {{event.status.code}} | 200 |
| {{event.type}} | focusin |

Since the different browser evaluate this javascript expression differently you see 3 values for the allcount variable: one for IE, one for Firefox, and one default one (IE in this instance) so as to not break assertions written for single browser mode. Now we can write assertions that compare allcount.ie and allcount.ff for example. This is particularly useful to compare rendering page properties, by writing filters that return object positions in the DOM and comparing their values in different browsers.

Finally we need to be able to automatically control which browser(s) is (are) selected in general, or for a given test, or even for a given step. There are 3 ways to do that, one for each level of granularity. To control globally which browser(s) is (are) used by default, the "Default Browser Mode" section in the settings should be used.

Default Browser Mode
☐ Internet Explorer
☒ Firefox
☐ Safari

At the test level it can be overridden using the usual mechanism of defining the following property: `DEFAULT_BROWSER`. It can take the value NONE, IE, FF, WK or any pipe-delimited combination of those (like IE|FF).

Finally, at the event level, which browser is being used is controlled by the Browser filter:

Properties

A browser mode filter toggles the state of the playback window to use the selected browser(s).

☒ Text
☐ DOM Attribute
☐ Script
☐ Expression
☐ Cache
☒ Browser

All

☐ Internet Explorer
☒ Firefox
☒ Safari

When a browser filter executes, it selects the browsers specified in the filter to run from that point on. Typically you would use this filter just prior to looping in a test case, so you could run a step of steps in a given browser and then again the same set of steps in an other browser to compare the results.

12. How To - Use Pathfinder integration

12. How To - Use Pathfinder integration

Pathfinder is a server-side LISA component that can be used to make information about what happens on the server available to the client. You can enable it in the settings dialog by checking the "Enable Pathfinder Integration" checkbox. This will automatically turn on the "Capture HTTP traffic" option as well.

The main effect of turning on this option is that several new variables will be automatically made available as can be seen on this screenshot:

| Key | Value |
|-----------------------------|--|
| {{event.bytes.in}} | 266669 |
| {{event.bytes.out}} | 10137 |
| {{event.cookie.JSESSIONID}} | 0C7CEE708EB39418CCDF986FA5024F0A |
| {{event.frame.path}} | |
| {{event.param.cmd}} | list |
| {{event.path}} | |
| {{event.pathfinder}} | <?xml version="1.0" ?><Lisaint ver="1.0"><Attributes><MapEntry><key>lisa.lisaint.SessionIdKey</key><value>0C7CEE708EB39418CCDF986FA5024F0A</value></MapEntry></Attributes><Transaction><TransInfo><failTest>false</failTest><failTestMsg></failTestMsg><endTest>false</endTest><endTestMsg></endTestMsg><ComplInfo><name>http://examples.itko.com/itko-examples/user-manage.jsp</name><status>S</status><statusMsg></statusMsg><Request></Request><Response></Response><Type>http-in</Type><startTime>1224210041708</startTime><endTime>1224210041720</endTime><jvmlInfo>hostname="examples2.itko.com" processId="15909" threadName="ajp-0.0.0.0-8009-28" heapSizeAtStartMb="142.53" heapSizeAtEndMb="143.1" sql prepared="SELECT * FROM users" sql="SELECT * FROM users" elapsedMs="3" connection="jdbc:derby://208.101.52.251:1527/reports/lisa-reports."</jvmlInfo><ResultSet><Row><USERS.LOGIN>user38643762316263332D313033342D3438</USERS.LOGIN><USERS.PwD>9eY16PegQM+IRKQDu+bbKleMfs</USERS.PwD><USERS.PwD>qUqP5cyxm6YcTAhz05Hph5gvu9M=</USERS.PwD><USERS.FNAME>itko</USERS.FNAME><USERS.LNAME>test</USERS.LNAME></Row></ResultSet></sql><sql prepared="commit" sql="commit" elapsedMs="0" connection="jdbc:derby://208.101.52.251:1527/reports/lisa-reports.db;create=true (autoReconnect=true)</sqlSummary><UniqueSql id="1" sql="SELECT * FROM users" numInvocations="1" averageExecTimeMills="3" totalExecTimeMills="3" batchCount="0" isPrepared="true" totalBatchCount=0</totalBatchCount><totalWaitTimeMills>3</totalWaitTimeMills></SqlSummary><content></content></ComplInfo></TransInfo></Transaction></Lisaint> |

In addition to the variables listed in the User Guide's [Filters section](#), a few more can be seen:

event.response.server.time measures the time it took the server to generate the payload received by the client for the last transmission.
event.response.network.time measures the time it took the payload to go from the server and reach the client.
auto variable name are custom variables defined by Pathfinder to represent some variable value on the server and made available for inspection one the client.

Note if those variables are exposed by the LEK they will be available automatically even without Pathfinder integration.

event.pathfinder is the most important variable. It is an xml representation of the Pathfinder payload and contains all pathfinder information, that can then be extracted using XPath expressions, with the [lisa.pathfind](#) API.

| Output | Variables | Immediate |
|--|-----------|-----------|
| Execute javascript or built-in commands (type .help for info) | | |
| <pre> {{event.pathfinder}} IE><?xml version="1.0" ?> <Lisaint ver="1.0"> <Attributes> <MapEntry> <key>lisa.lisaint.SessionIdKey</key> <value>0C7CEE708EB39418CCDF986FA5024F0A</value> </MapEntry> </Attributes> <Transaction> <TransInfo> <failTest>false</failTest> <failTestMsg></failTestMsg> <endTest>false</endTest> <endTestMsg></endTestMsg> <ComplInfo> <name>http://examples.itko.com/itko-examples/user-manage.jsp</name> <status>S</status> <statusMsg></statusMsg> <Request></Request> <Response></Response> <Type>http-in</Type> <startTime>1224210041708</startTime> <endTime>1224210041720</endTime> <jvmlInfo>hostname="examples2.itko.com" processId="15909" threadName="ajp-0.0.0.0-8009-28" heapSizeAtStartMb="142.53" heapSizeAtEndMb="143.1" <sql prepared="SELECT * FROM users" sql="SELECT * FROM users" elapsedMs="3" connection="jdbc:derby://208.101.52.251:1527/reports/lisa-reports." </jvmlInfo> <ResultSet> <Row><USERS.LOGIN>user38643762316263332D313033342D3438</USERS.LOGIN><USERS.PwD>9eY16PegQM+IRKQDu+bbKleMfs</USERS.PwD> <USERS.PwD>qUqP5cyxm6YcTAhz05Hph5gvu9M=</USERS.PwD><USERS.FNAME>itko</USERS.FNAME><USERS.LNAME>test</USERS.LNAME> </Row> </ResultSet> </sql> <sql prepared="commit" sql="commit" elapsedMs="0" connection="jdbc:derby://208.101.52.251:1527/reports/lisa-reports.db;create=true (autoReconnect= </sqlSummary> <UniqueSql id="1" sql="SELECT * FROM users" numInvocations="1" averageExecTimeMills="3" totalExecTimeMills="3" batchCount="0" isPrepared="true" totalBatchCount=0</totalBatchCount> <totalWaitTimeMills>3</totalWaitTimeMills> </SqlSummary> <content></content> </ComplInfo> </TransInfo> </Transaction> </Lisaint> </pre> | | |

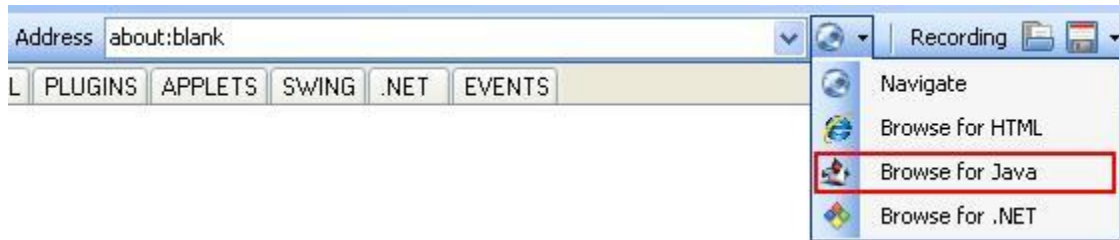
For instance, to get the query string out of the first SQL statement in the payload pictured above, you can use return

`lisa.pathfind('///sql/attribute::sql')`, which will return `SELECT * FROM users`.

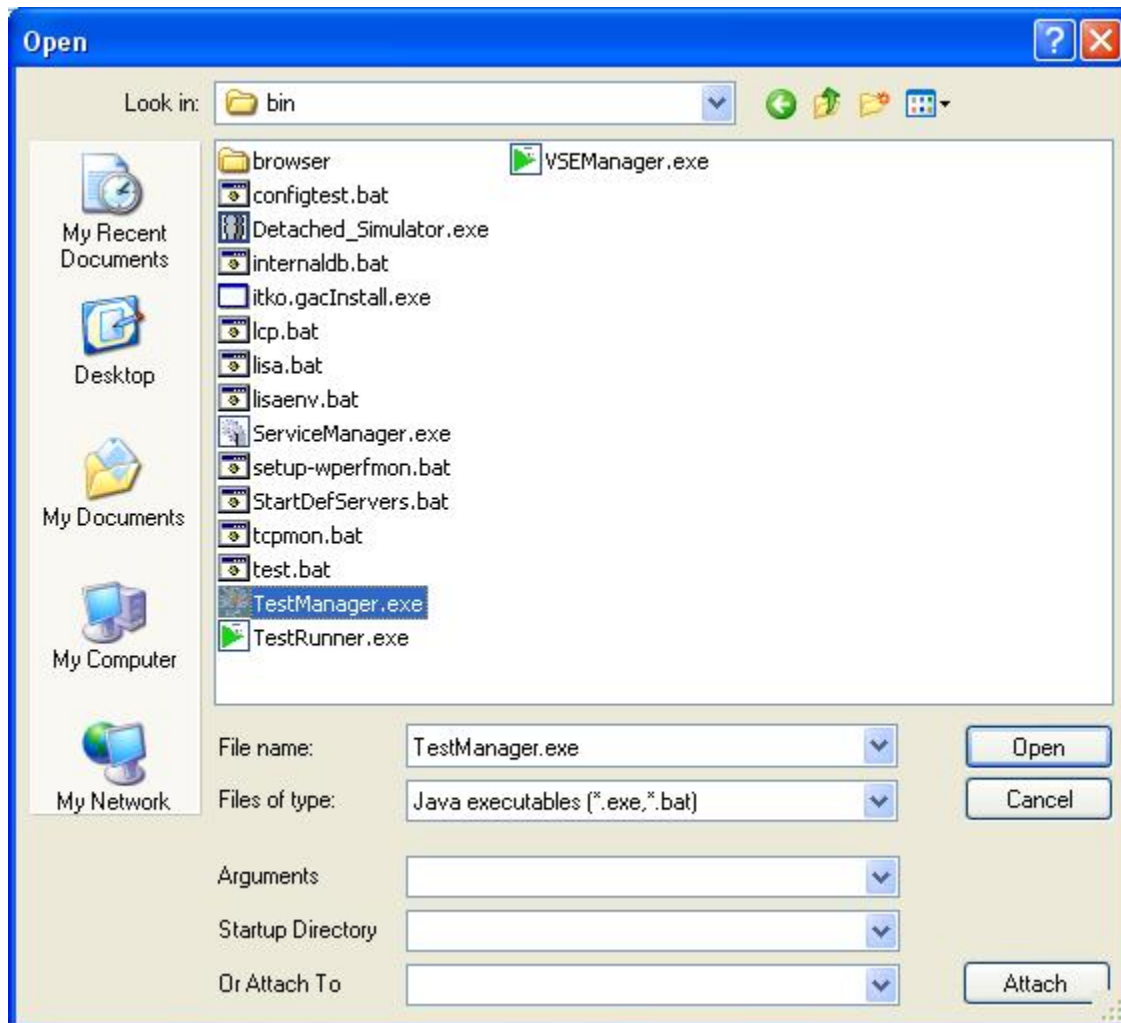
13. How To - Write Java Swing and WebStart tests

13. How To - Write Java Swing and WebStart tests

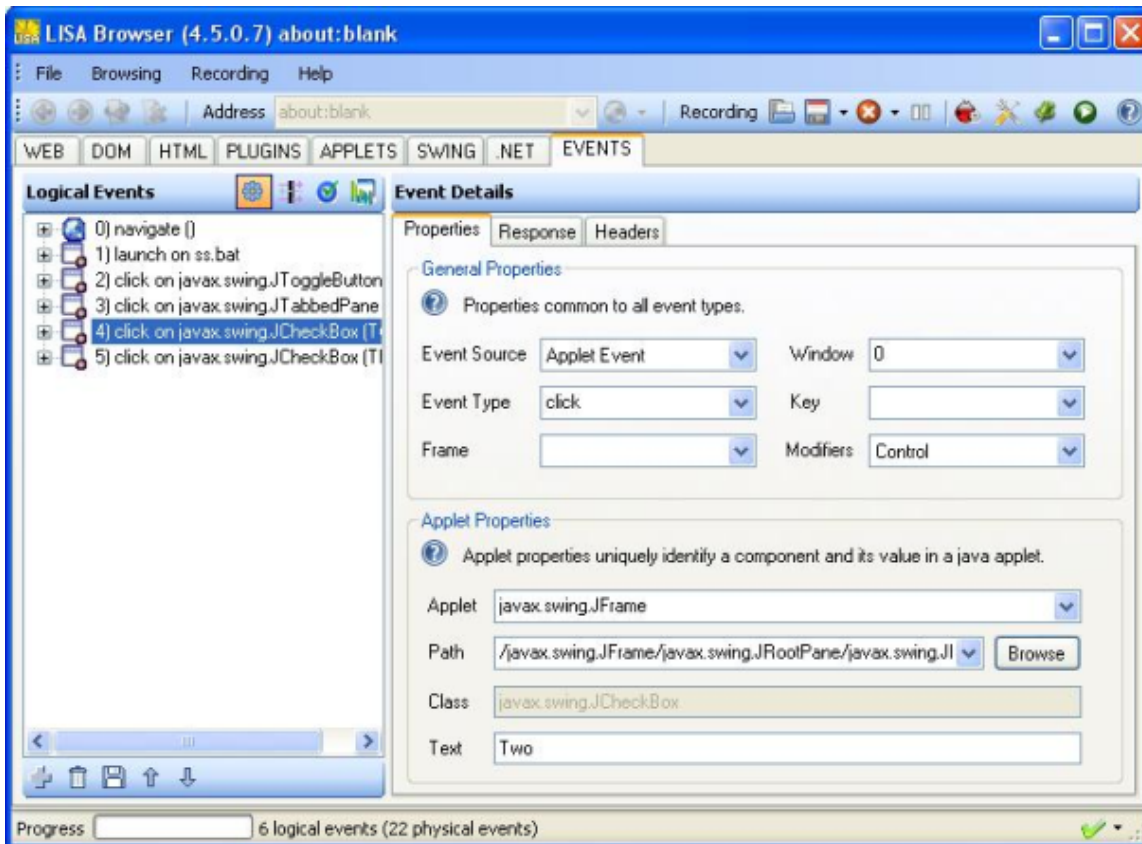
Authoring Swing tests is practically no different from writing applet tests. From a GUI perspective, the only difference is how you launch the application.



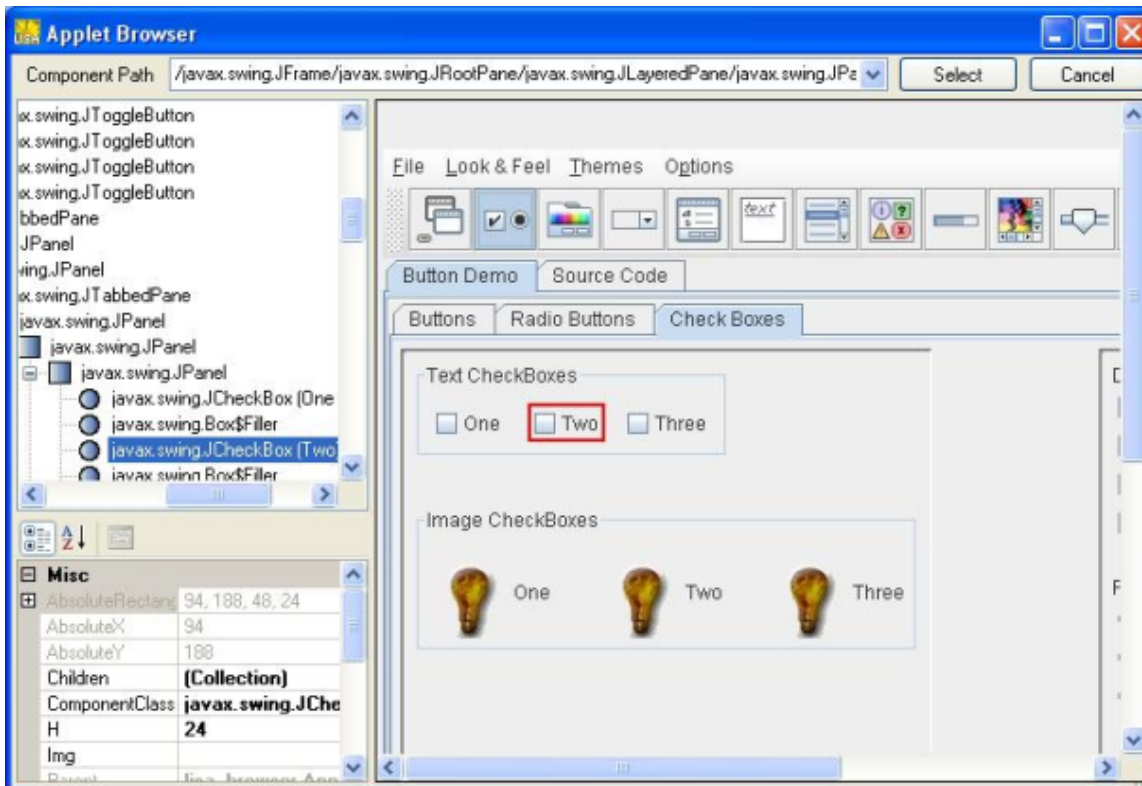
Instead of navigating to a URL, you browse the filesystem for an executable or batch file used to launch the swing application. In most cases the executable is going to be `java.exe` or `javaw.exe` and you can specify the command line (including classpath, VM arguments, etc...) in the Open Dialog window, as well as the start directory (optionally). In other cases the application is launched from a pre-packaged executable (as LISA TestManager is for instance) or from a batch script. This is supported too and makes no difference.



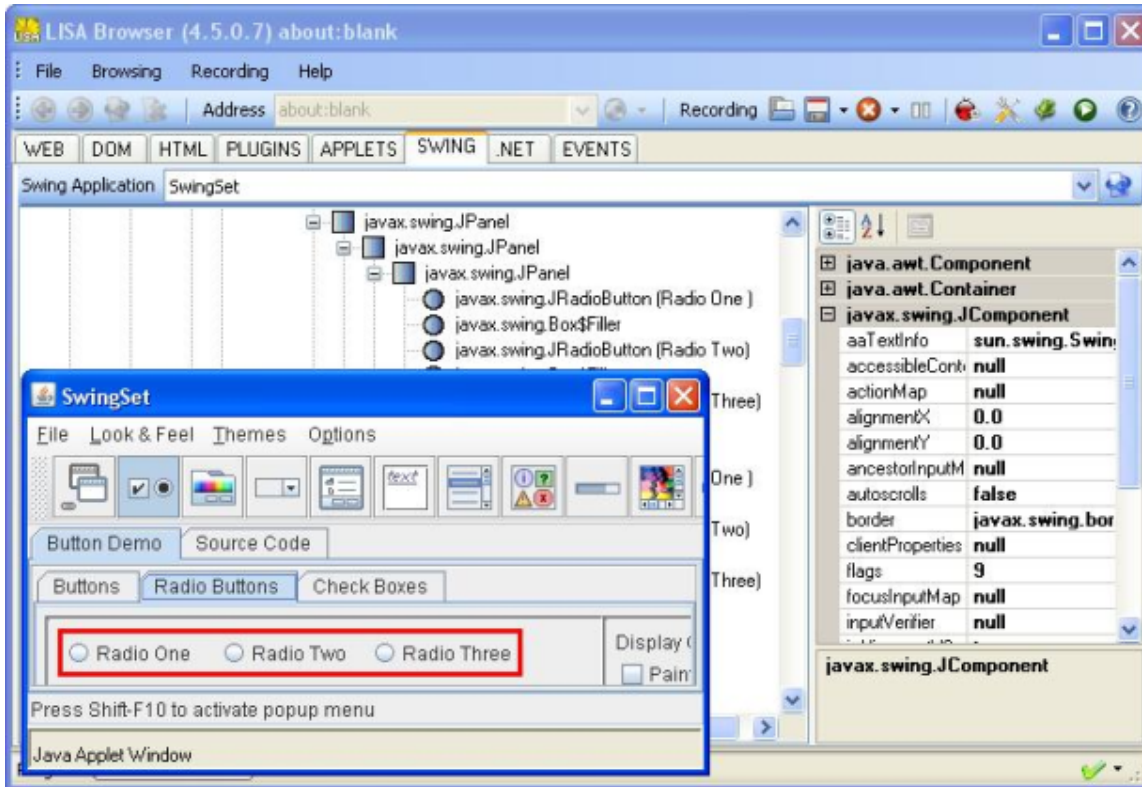
Below is a screenshot of a few events recorded against a Swing application, you will notice it looks almost identical to Applet events, except for the Applet field, which is now a JFrame.



You can similarly browse for offline editing and bring up the Applet/Swing browser to change and/or parametrize event targets (note the red highlight rectangle below is part of the application, not a screenshot artifact):

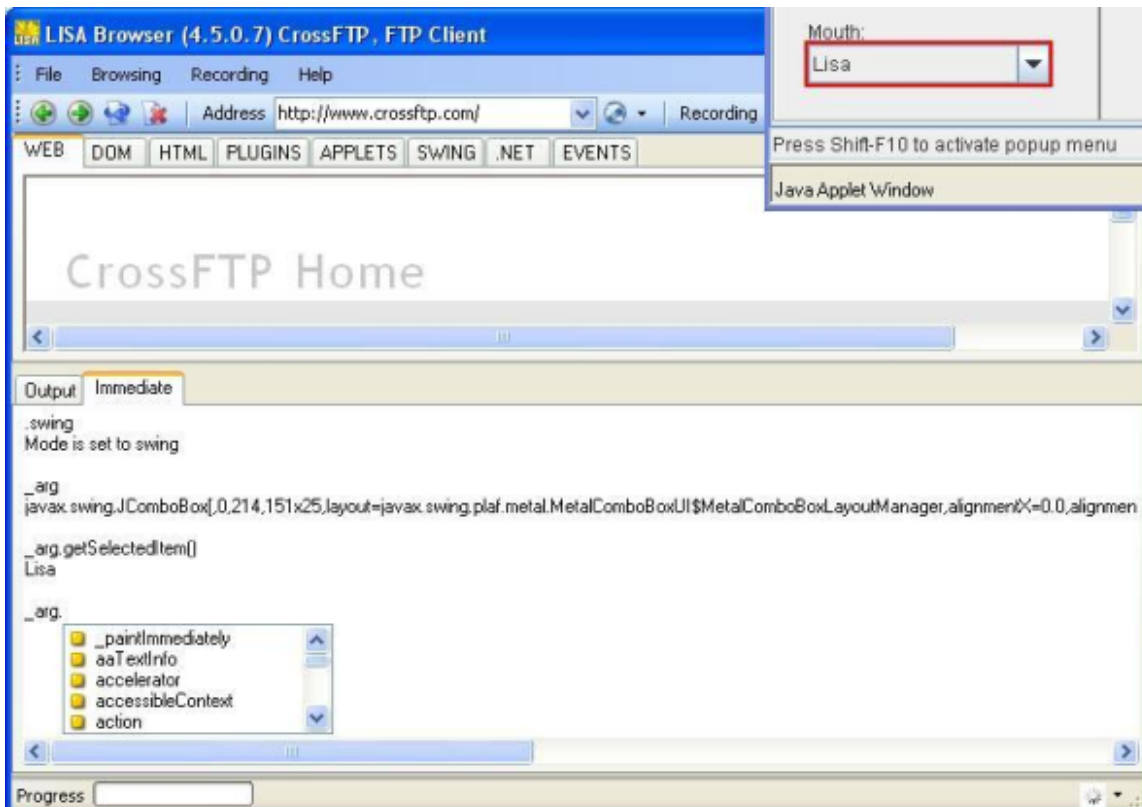


Finally, Swing applications also have their own live instance browser under the SWING tab (in both recorder and playback modes). When selecting a component in the swing hierarchy, this component will get highlighted in the live application so you can easily verify which elements you are targeting (note the red highlight rectangle below is part of the application, not a screenshot artifact):



Note that the exact same things apply to Java WebStart applications except that they can be launched directly from a jnlp link in a browser web page. The test will then automatically mix web steps and swing steps.

Support for arbitrary beanshell expressions is supported just as in applets and the last event target can also be referred to with the special variable `_arg`. You can test your java filters in the debug window by setting the mode to ".swing" and then evaluating the expression:



An example of how this would be useful is when the text filter is not easily able to get some value onscreen, in particular if you have a JTable, you can not get its cell values unless you double click them because the cells are not swing components (until they are double-clicked). With a script

filter, you could simply select the table as the component and then write a script like: `return _arg.getModel().getValueAt(2,3);`

Finally, an interesting point to note is that while this step's primary purpose is to do Swing or AWT testing, there is nothing that prevents you from running against headless or console java applications. You can run "in-container" tests by specifying any scripts you want in java script nodes.

14. How To - Write .NET WinForms tests

14. How To - Write .NET WinForms tests

Coming...

15. How To - Debug a test

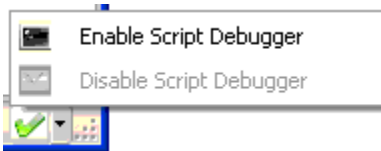
15. How To - Debug a test

Unexpected behavior can occur during recording or replay, and for both cases the best approach is to debug the test directly in the DOM browser, since it acts as the test execution environment, and is just driven by LISA during staging or ITR. The only exception to this rule is if the test must use multiple iterations of a dataset or variables defined in non-web 2.0 steps, because LISA takes care of sending those values to the DOM browser environment.

The main tool at your disposal to understand what is happening in a test is the debugging window, which you can toggle both in recording and playback mode by clicking the red debug icon in the toolbar:

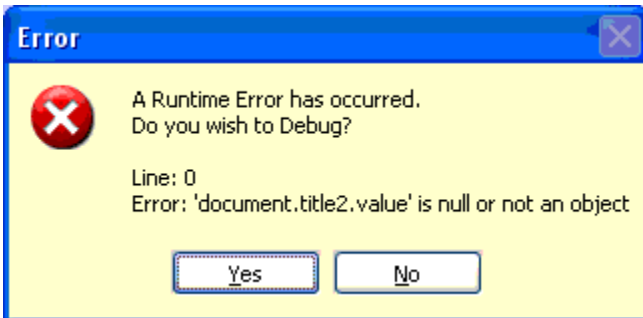


You can also toggle it by clicking the little status icon in the bottom-right corner of the screen:

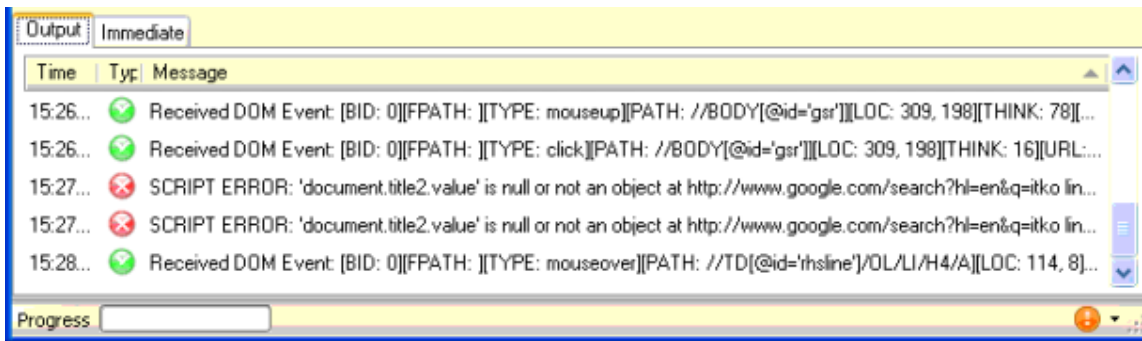


This icon has 3 states: a green checkmark to indicate no errors, a spinning wheel to indicate a document is loading or an orange icon to indicate there have been errors, most of the time script errors, but they could also be non-critical DOM browser errors. Script errors are sometimes difficult to debug given just the error message, hence the little popup menu as seen above to enable or disable a Javascript Debugger.

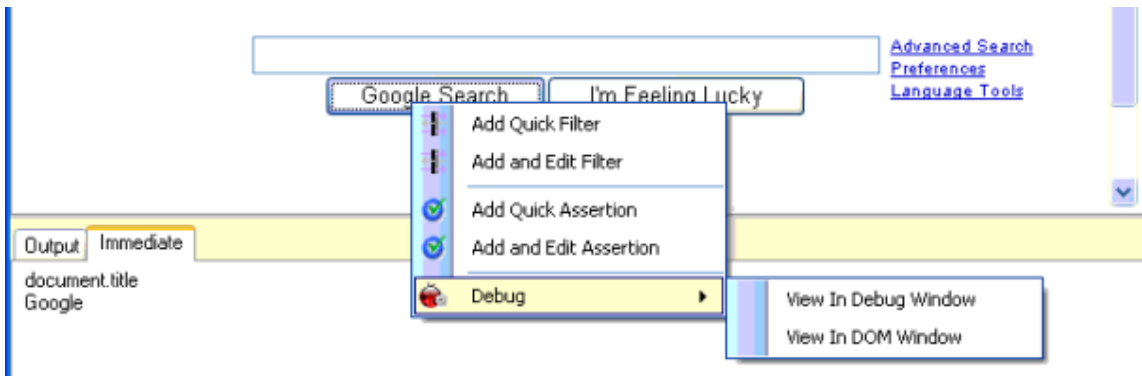
By default all debuggers are disabled to not interfere with the recording or playback, but if you enable it and there is a script error, you will see this error message popup:



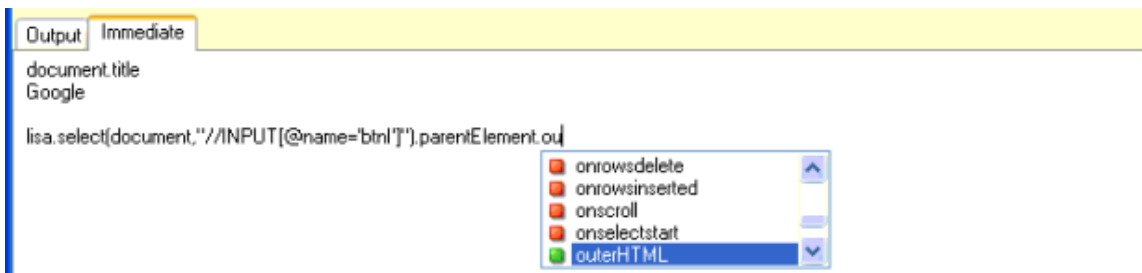
Clicking "Yes" will launch the debugger and show you all the details about the script error. Other than that, just opening the debug window will suffice in most cases:



In recording mode, 2 tabs are available in the debug window, the Output tab as pictured above and the Immediate tab. The Output tab just logs all messages (informational, warnings, debug as controlled by the log settings) and the Immediate lets you do more active debugging by typing in commands. It behaves like an interactive prompt on a web page and supports built-in commands, javascript and java (including the use of variables, as denoted by the usual syntax `variable`). It also supports point-and-click interaction with the currently viewed document as illustrated below:



By right clicking on an element, you get a menu that contains a **Debug** entry, to let you view the element in the DOM view or in the Immediate window. If you choose Immediate, you will be able to evaluate any DOM or javascript expression on the element, helped by intellisense:



This can be helpful in writing script filters but also to inspect javascript event handlers for instance. All built-in commands can be accessed by typing a dot (.) as the first character of the line. You can display what they do using the `.help` command.

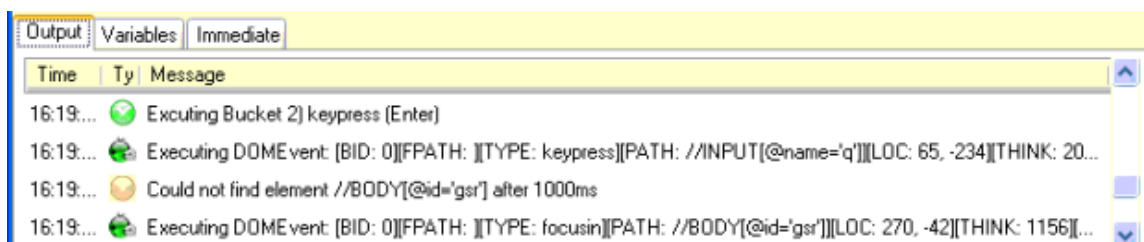


For instance, there are a `.js` and a `.java` entries that allow you to swap between javascript and java syntax for commands. The last element to receive an event can be referenced as the special variable `_arg`, just like in script filters.



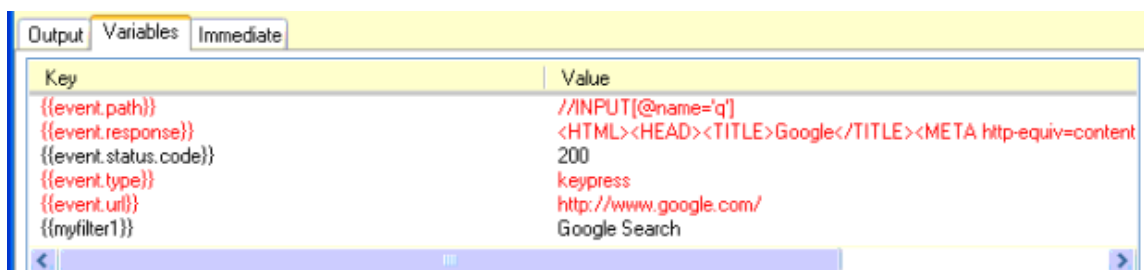
Note that all these javascript and java (beanshell to be exact) expressions can be used in filters so this is a useful way to design your filters during recording.

Most of the debugging will probably take place during playback since it will put to the test both the test design and the application under test. Playback offers the same debugging window with the Output and Immediate tab, and also a Variables tab. Most errors or warning you will see logged in the Output tab will be due to the failure of identifying an element on the page or in a control (as seen in this screenshot).

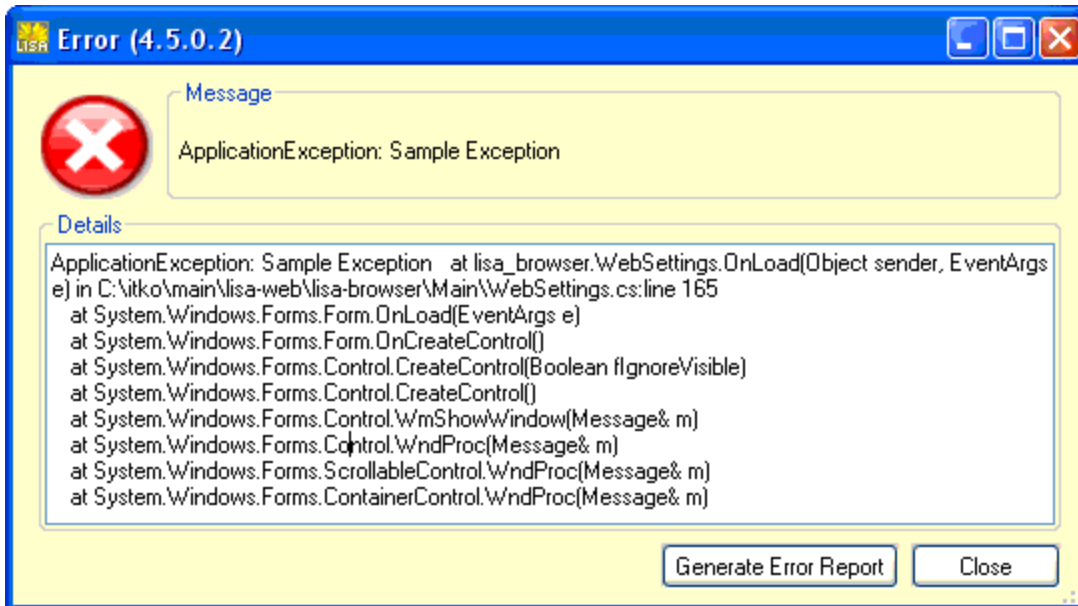


There could be several reasons for this: a genuine bug in the system under test or a failure to adequately parametrize a dynamic element (see the [Dynamic Elements](#) section). You will also see the filters and assertions being executed and their values, so you can quickly see why an assertion fired for example.

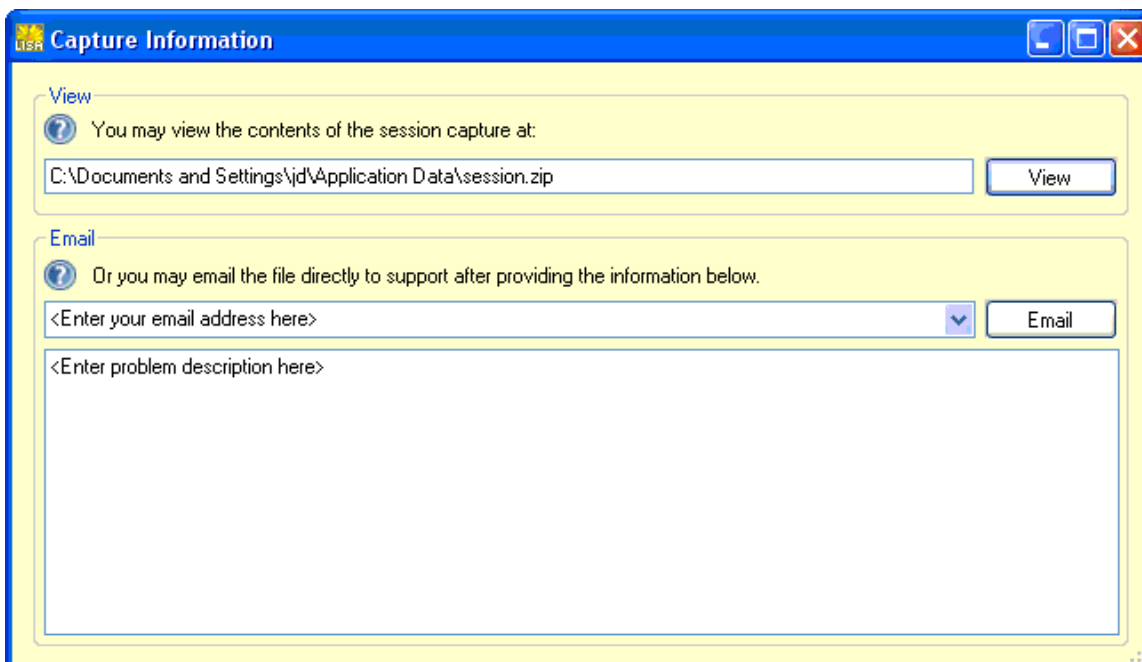
The Immediate tab behaves the same way as in recording mode. The Variables tab will list all the variable values for every step (highlighting in red the ones that were just modified). This is particularly useful when you set breakpoints or step through a test as in a debugger, you can observe the value of all the variables:



Finally, severe errors won't be caught and logged in the Output tab of the debug window, but instead will generate a dialog like:



In most cases (especially during recording) you can close the dialog and proceed but not in all cases. And since it should not happen, you can report it by clicking the **Generate Error Report** button. This will create a zip archive containing information about the environment, the settings, the current recording and playback, all the logs including the exception and a screenshot, and then present the user with the following dialog:



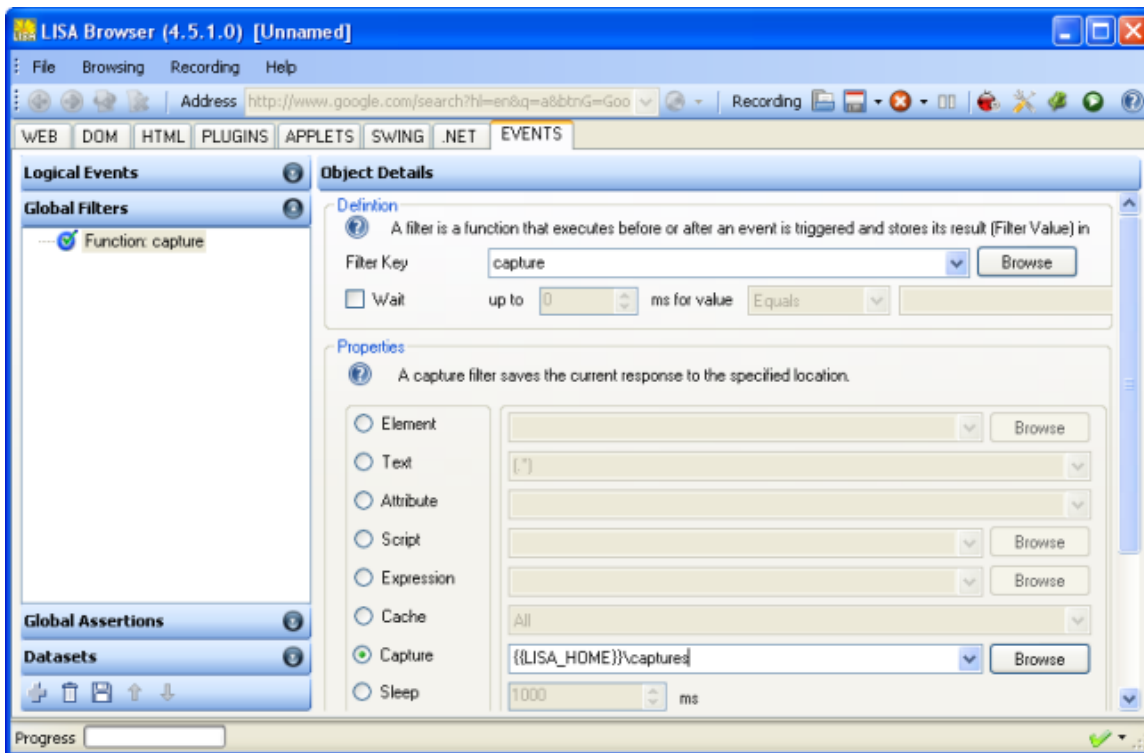
You can open the zip archive by clicking **View** and then email it to support by clicking **Email** after filling the required information for troubleshooting. If there is an SMTP server running on the machine (if IIS is installed for example), the email will be sent automatically, otherwise the default email client will be opened. If there is none, you will have to email the file manually.

16. How To - Use global filters and global assertions

16. How To - Use global filters and global assertions

The web 2.0 browser provides the ability to execute global filters and global assertions, that is filters and assertions that execute on every step. If you have repetitive actions you need to take, rather than defining those manually on every step, you can do define them once, globally and they will execute after every step. Global filters execute before local filters and global assertions execute before local assertions.

Adding a global filter is very similar to adding a normal filter or assertion. You go to the **EVENTS** tabs and select the "Global Filters" or "Global Assertions" collapsible panels.



This picture, for instance, shows how to capture screenshots on every step and save those in a specified directory for later investigation. Another typically useful usage of a global assertion is depicted below

If

☒ If

☐ Or If there are any ☐ W3C HTML Errors ☐ W3C HTML Warnings ☐ Broken HTML References

☐ Or If the event took over ms

☐ Or If the event generated over bytes

Then

☒ If the selected expression above is ☒ True ☐ False

Then

☐ And save a screenshot to

This verifies after each step that the server did not return an internal error code.

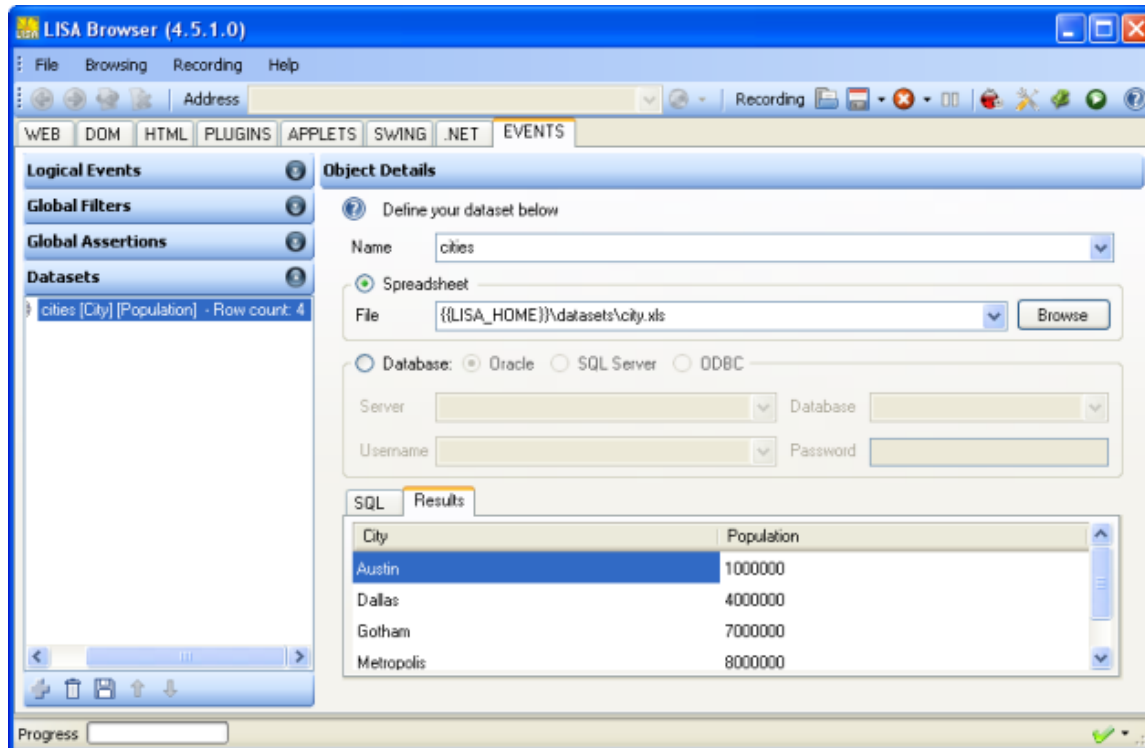
17. How To - Interact with external resources

17. How To - Interact with external resources

LISA has many steps to interact with external resources, be they flat files, excel files, databases, etc...and all of these resources will be automatically available to web 2.0 steps when run through LISA. However, if you run standalone, or through LISA but in the debugger, the external LISA steps won't execute and the external data won't be available. That's why there are ways to access external data from the DOM browser too.

First, there is the concept of web 2.0 dataset. Those are defined directly in the browser and don't depend on TestManager. To define them, you

go to the EVENTS tab and select the Datasets collapsible panel, then Add a dataset. The details panel lets you specify its name, its data source and makes it easy to test:



Once a dataset is defined, using it in steps or filters is very easy using the following syntax:
dataset_name[row_index][column_name_or_index].

General Properties

Properties common to all event types.

Event Source: DOM Event Window: 0 X: -111

Event Type: change Key: Y: -27

Frame: Modifiers: Think: 0

DOM Properties

DOM properties uniquely identify an element and its value on a page.

Url: http://www.google.com/

XPath: //INPUT[@name='q'] Browse

Value: {{cities[i++][Population]}}

Tag: INPUT Type: text

This allows you to retrieve data at any specified row and column without having to advance the dataset automatically. You simply need to keep track of an index to get to the desired row. However, if you want to automatically advance or go back, the syntax also supports the operators ++ and – as pictured above.

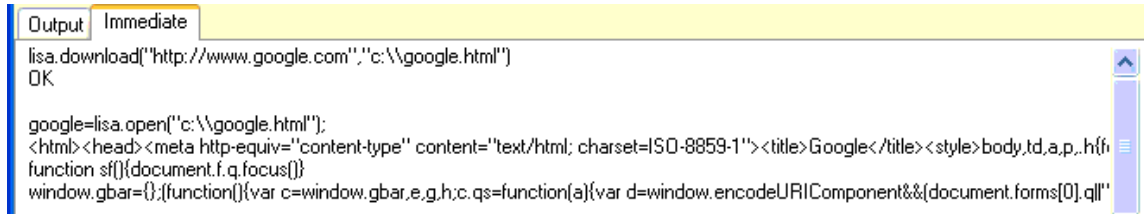
Something like cities[i++][Population] will get the value at the i-th row in the Population column and increase i by one for next time it is used. As a convenience, 1-letter variables used as integers like i will automatically be defined and set to 0 if they haven't been defined before.

Finally, the syntax cities.length or cities.count can be used to determine the number of rows in the cities datasets. This is useful to write exit condition assertions when iterating over a dataset (e.g. If i More Than cities.length Then Go To event xyz).

In addition you can use of the global **lisa** scripting object. Of interest here are the following methods: download, open, fileQuery, dbQuery.

- **download**: allows you to download a resource from a given url to the specified path, so you can interact with it, using for example:
- **open**: will read the contents of the specified file and return it as a string.
- **fileQuery**: will execute a SQL query against an Excel file and return a javascript **dataset**.
- **dbQuery**: will execute a SQL query against a Database file and return a javascript **dataset**.

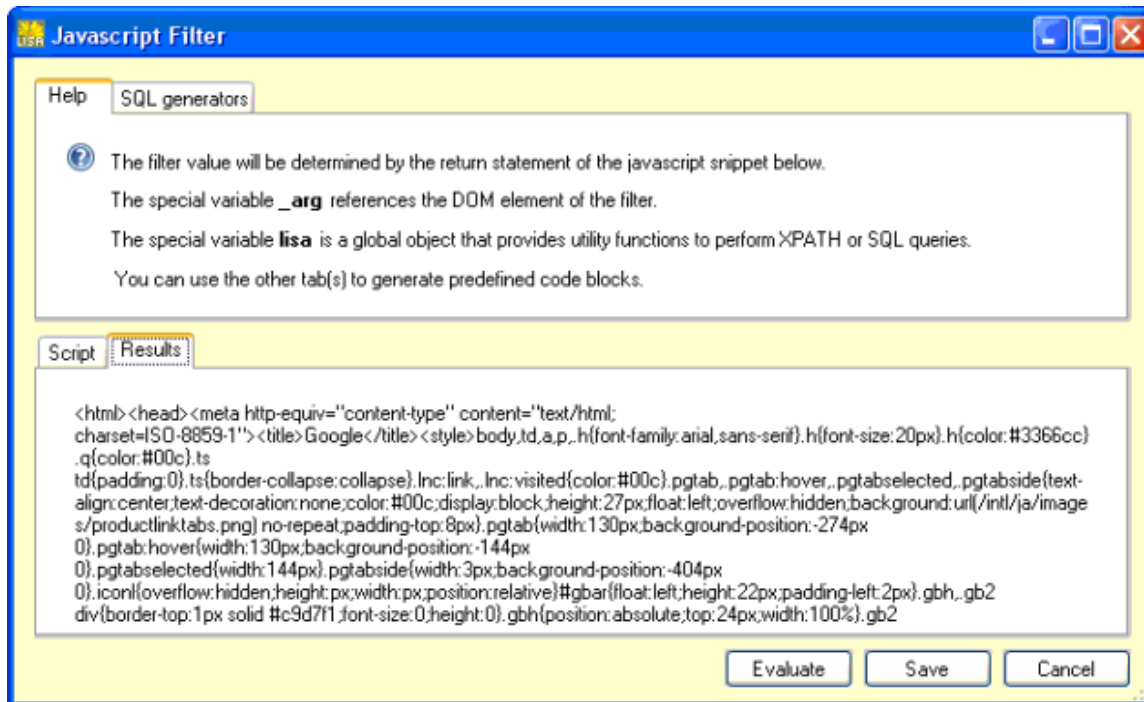
All of these functions can be used from the Immediate debug window or script steps or filters. For example, here is how you would use download and open from the Immediate window:



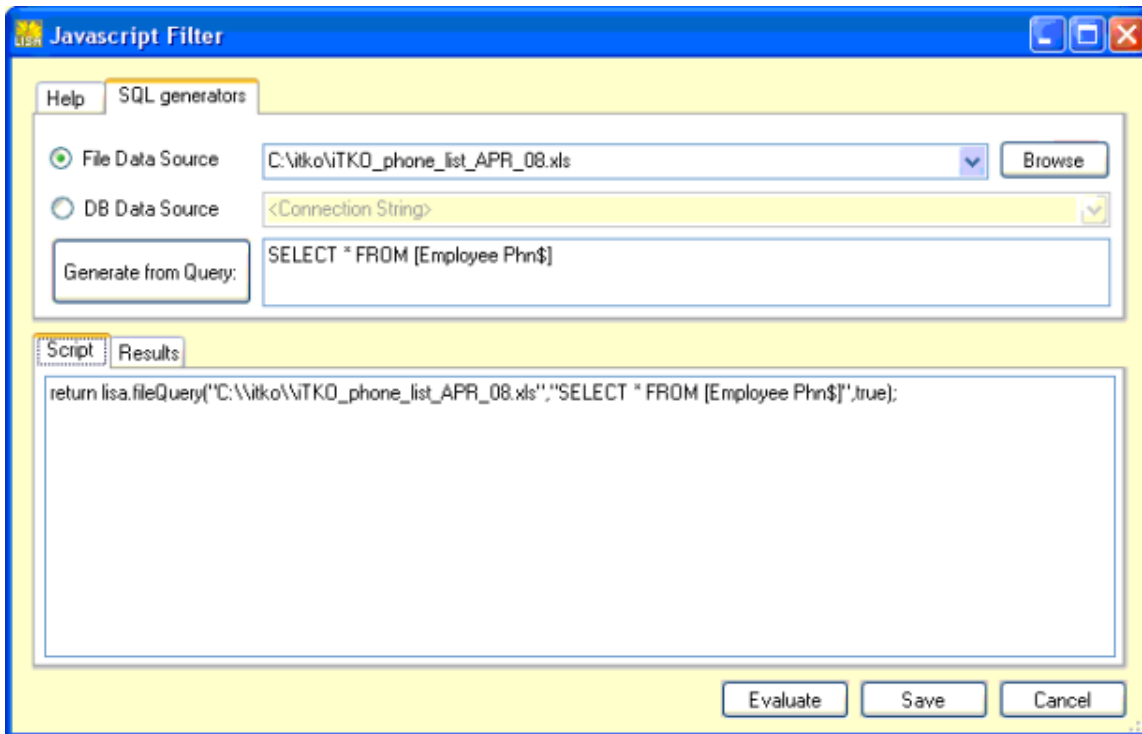
```
lisa.download("http://www.google.com","c:\\google.html")
OK

google=lisa.open("c:\\google.html");
<html><head><meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"><title>Google</title><style>body,td,a,p,.h{font-family:arial,sans-serif}.h{font-size:20px}.h{color:#3366cc}.q{color:#00c}.ts{border-collapse:collapse}.lnc:link,.lnc:visited{color:#00c}.pgtab,.pgtab:hover,.pgtabselected,.pgtabside{text-align:center;text-decoration:none;color:#00c;display:block;height:27px;float:left;overflow:hidden;background:url(/intl/ja/image_s/productlinktabs.png) no-repeat;padding-top:8px}.pgtab{width:130px;background-position:-274px 0}.pgtab:hover{width:130px;background-position:-144px 0}.pgtabselected{width:144px}.pgtabside{width:3px;background-position:-404px 0}.icon{overflow:hidden;height:px;width:px;position:relative}#gbar{float:left;height:22px;padding-left:2px}.gbh,.gb2{div{border-top:1px solid #c9d7f1;font-size:0;height:0}.gbh{position:absolute;top:24px;width:100%}.gb2
```

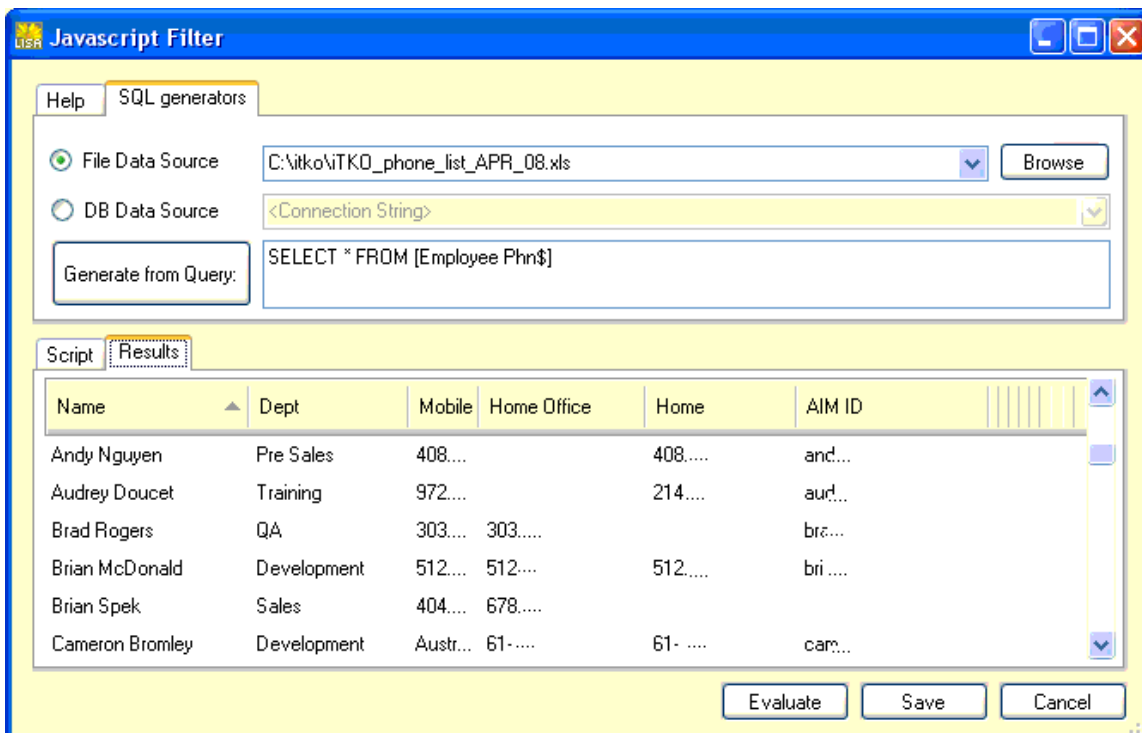
Similarly, if you created a script filter called google, clicked on **Browse** to bring up the script details popup, and clicked the **Evaluate** button after typing the script return lisa.open("c:google.html"), you would see the evaluation results:



From that point on, this html string is available as `google`. The other 2 functions, fileQuery and dbQuery can be used similarly, but there is a tab in the script details popup called **SQL Generators** that helps you generate the calls. Filling the properties at the top and clicking **Generate From Query** yields a script snippet:



which in turns evaluates to the following dataset:



Using the [dataset](#) API, you can create other filters and assertions to use the dataset. Let's assume the above dataset is saved in the `employees` filter. For example, all the following expressions are valid to use in a filter script: `return employees.count()`, `return employees.rows(0).cells(0)`, `return employees.rows(0).cells("Name")`, etc...

This makes it extremely easy to parametrize data and loop through datasets. This type of dataset is akin to a Local Lisa dataset since its data is not shared across test instances, but you have explicit access to any row or cell instead of implicitly moving to the next record on a given step.

18. How To - Run Load Tests

How To - 18. Run Load Tests

Starting in LISA 4.6, running web 2.0 load tests is supported in the same way as other steps, by specifying the desired number of virtual users in a staging document. Each virtual user runs a separate browser instance, which consumes a fair number of resources, so there is a hard limit of 40 virtual web 2.0 users per machine.

If your test is purely web-based, by running in dual IE/Firefox mode, you can double that number without using much more resources. Similar considerations apply for running other GUI technologies (Applets, Swing, .NET, Native, etc...). Multiple instances are supported up to point that is mostly determined by available hardware.

Several settings control how multiple instances run (sandboxed or shared, under which user account, etc...). Those can be consulted in the [Web 2.0 architecture](#) section.

If you are using LISA 4.5 or earlier, read on...

Currently web 2.0 steps do not support multiple virtual users running them in parallel in the same JVM (typically it will cause an error dialog to pop up stating there is a socket or listener error if you try to do so). You can still run multiple instances of the DOM browser in parallel, even on the same machine, provided you use one per simulator (i.e. one per JVM).

The other option is to run in headless mode. After a web 2.0 test is recorded and saved back to TestManager, the boolean variable HEADLESS will be added to the configuration (with a value of false by default). If you change that to true, web 2.0 steps will be replayed without the DOM browser, using the pure java javascript engine [Mozilla Rhino](#) and the [htmlunit](#) java library.

The advantages of this approach are:

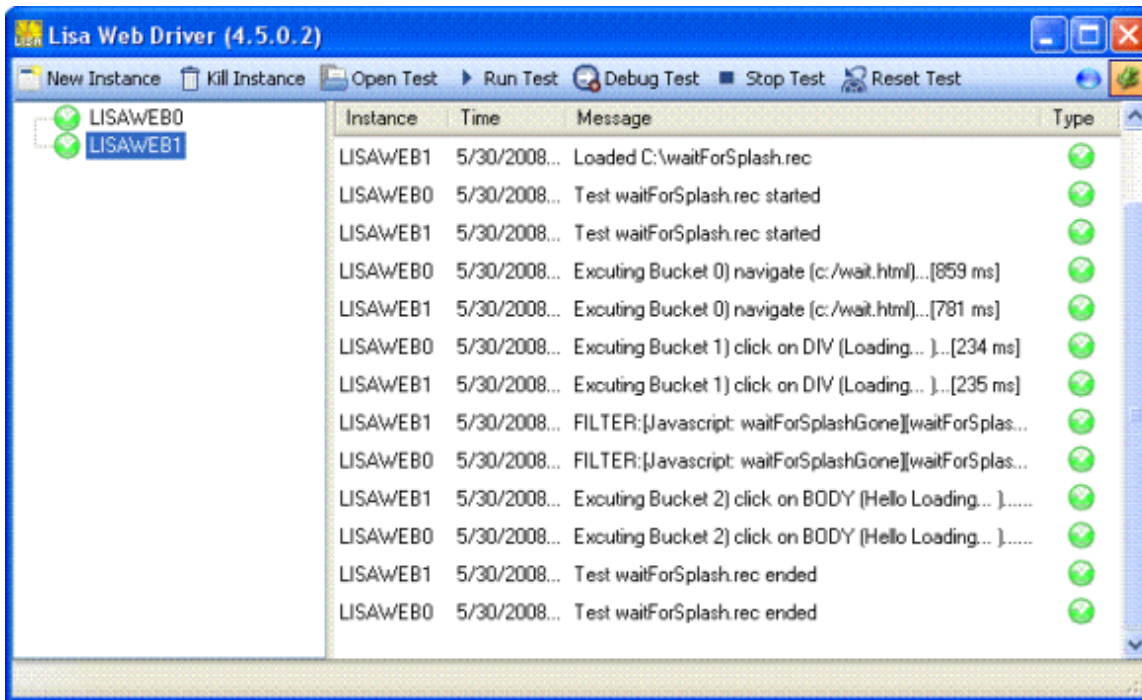
- you can run true load tests (multiple vusers per JVM).
- it is platform-independent
- it can simulate multiple browsers (by setting the BROWSER configuration variable).

The disadvantages are:

- some sites do not work well with it (the more complex the javascript they use, the more likely it is they may have problems).
- it is relatively heavyweight so don't expect to run hundreds of vusers on the same machine (dozens on a standard desktop is realistic though).
- it doesn't support frame (currently), authentication (currently), applets (currently) or Active X controls (never).

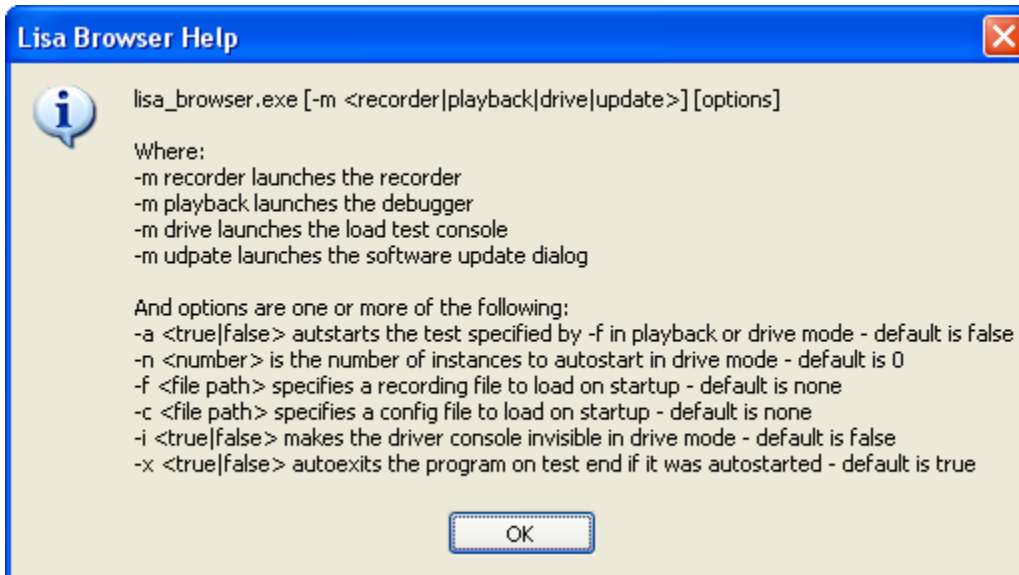
Finally, load testing in GUI mode is going to be available as well, but is only doable in standalone mode at this time.

Running `lisa_browser.exe -m drive` will launch the multi-instance driver:



Then you can add as many instances as you want of the DOM browser (in our environment it was able to handle up to about 40 instances), then load a test and run it. All instances will run it in parallel and report the results as they go. If you run in dual mode (IE + Firefox) you can double your number of clients with little overhead (so up to about 80 per machine).

More specifically, you can control through command-line parameters how to start those instances:



There is also a way to specify an xml file that contains a list of instances along with test files (and optionally config files) they need to run. Users can also be specified if you need a test to be run in the context of a given user (typically useful for websites with windows authentication). The syntax of this xml file is:

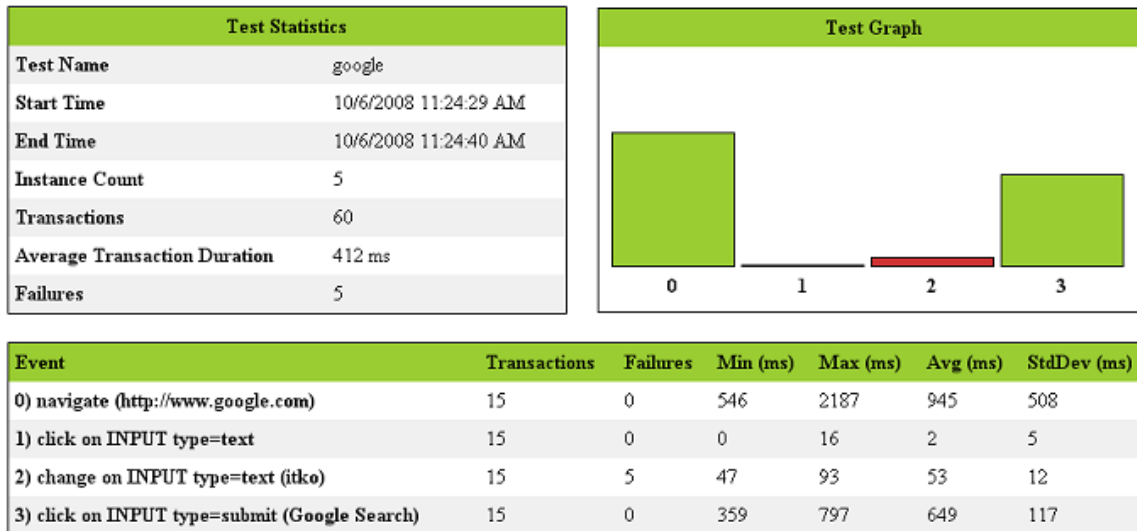
```
<staging>
<instance username="u1" password="p1" file="test1.rec"/>
<instance username="u2" password="p2" file="test2.rec" config="config2.config"/>
<test file="test3.rec" config="config3.config" instances="2"/>
</staging>
```

and it can be invoked as `lisa_browser.exe -m drive -stg stagingfile.xml`

All instances run in a separate windows user account (dynamically created and deleted as needed), so cookies and cache don't overlap between instances. The advantage of this approach is that it supports everything the DOM browser supports in single mode. The top-right icons allow you to pin the driver window on top and to hide all the DOM browsers as they run the test, making it look headless.

Finally, when the running tests are over, xml reports will be generated in the out directory. A default xsl stylesheet is provided to render the report as html but it could be customized to any other rendering as all the data is present in the xml.

LISA Web 2.0 report for [google]



Note: load tests in drive mode currently require you to be logged in as administrator on the computer.

19. How To - Run in a non-privileged account or on 64 bit platforms

19. How To - Run in a non-privileged account or on 64 bit platforms

Q: Can the web 2.0 step be used on non-administrator accounts?

A: Yes, it can be run even in Guest accounts but will require a bit of environment configuration:

1) Port specification

By default, the LISA browser uses dynamic ports to communicate with LISA. If some ports are locked down, you can assign them instead using the `lisa.browser.source.port` and `lisa.browser.target.port` properties (in the `lisa.properties` or `local.properties`).

2) Add permission for port listening

Some accounts may not have enough privileges to listen on a specific port (typically causing an Access Denied error). To add this permission, use the `httpcfg.exe` tool (available at <http://www.microsoft.com/downloads/details.aspx?familyid=49ae8576-9bb9-4126-9761-ba8011fabf38&displaylang=en>) and run the following command: `httpcfg.exe set urlacl /u http://+:8098/ /a D:(A;;GX;;;WD)` where you replace 8098 with the port you specified in step 1) as `lisa.browser.target.port`.

Note: a message box containing these instructions will appear the first time the error is encountered during a LISA run.

3) Register Tidy COM component

The LISA browser uses the `w3c Tidy` COM component to identify warnings and errors in the HTML code, as well as reformat it. This component is dynamically registered but if the account does not have the sufficient privileges to register a COM component, you should log in as administrator and run the command: `regsvr32 \bin\browser\TidyATL.dll`

Note: a message box containing these instructions will appear the first time the error is encountered during a LISA run. If no action is taken the web 2.0 step will still be functional but missing the above functionality.

Q: Can the web 2.0 step run on 64-bit platforms

A: Yes, it can. The only thing to be aware of is that the component used to display the HTML source during test authoring will be disabled, but it should not affect running the test in any way.

20. How To - Record and replay against non us-english websites

20. How To - Record and replay against non us-english websites

Using Web 2.0, there is nothing special you need to do to record and replay against websites that use any language or locale. Of course, to see the right glyphs on the screen you will need to download the language packs corresponding to the appropriate codepage, but if you can see them correctly in a browser, it means they're already installed.

The following screenshots show the DOM browser and TestManager after it recorded and replayed a test against a japanese website.

Authoring and evaluating a filter during recording:

Properties
A text filter retrieves the inner text of a DOM element using a regular expression (first capturing group).

☐ DOM Element

//TABLE[@id='content-table']/TBODY/TR[1]/TD[2]/TABLE/TBODY/TR/TD/H1

Browse

☒ Text

[.]

☐ DOM Attribute

☐ Script

Browse

☐ Expression

Browse

☐ Cache

All

Quick Test
Click Evaluate to see what value this filter would return if it were evaluated during the recording.

Recorded Value

レノボ・ジャパンについて

Evaluate

Clear All

Replaying the test in the DOM browser while showing debug events:

Logical Events
0) navigate (http://w
1) click on A (レノボ・
2) click on H1 (レノボ
3) click on H1 (レノボ
continue (quiet)
continue

Web HTML


Output Variables Immediate
Time T: Message
13:28:... Executing DOMEvent: [BID: 0][FPATH:][TYPE: click][PATH: //TABLE[@id='content-table']/TBODY/TR[1]/TD[2]...
13:28:... [Bucket Duration: 1062 ms]
13:28:... FILTER:[Text: filter.click.8214062][filter.click.8214062=レノボ・ジャパンについて]

Replaying the test in the ITR after saving it to TestManager:

Execution History

- 0) navigate (http://{{SERVER0}}/jp/ja/index.html)
- 1) click on A (レノボ・ジャパンについて)
- 2) click on H1 (レノボ・ジャパンについて)
- 3) click on H1 (レノボ・ジャパンについて)
- end

Response Properties Test Events

Initial property values may be changed prior to executing the step. They are read and edited as needed.

Key	Value
AJAX	true
BROWSER	Firefox
HEADLESS	false
LASTRESPONSE	<HTML lang=ja xml:lang="ja" xmlns=
LISA_HOST	dude
LISA_LAST_STEP	3) click on H1 (レノボ・ジャパンにつ
LISA_TC_PATH	C:\Temp\rand
LISA_USER	jd
SERVER0	www.lenovo.com
SERVER1	www-06.ibm.com
event.path	//TABLE[@id='content-table']/TBODY
event.response	<HTML lang=ja xml:lang="ja" xmlns=
event.status.code	200
event.type	click
event.url	http://www-06.ibm.com/jp/pc/lenovo
filter.click.8214062	レノボ・ジャパンについて

21. How To - Run in Crash Dump mode

21. How To - Run in Crash Dump mode

The LISA browser runs mostly in managed code but due to external native libraries bugs and portions running native code (e.g. ActiveX, Applets, jdglue, etc.), it is possible to sometimes observe crashes in certain environment and under special circumstances (those will usually manifest themselves as AccessViolation exceptions).

To facilitate resolution of those errors, the browser supports running in Crash Dump mode. When these errors occur in Crash Dump mode, the browser will generate a large dump file that can be later analyzed to identify the root cause of the issue. To turn on Crash Dump mode, follow these instructions:

- 1) Download Windows Debugging Tools at the following location: [x86](#) or [x64](#).
- 2) Define the `_LISA_CRASH_DUMP` environment variable to be the Windows Debugging Tools install directory (e.g. `_LISA_CRASH_DUMP=C:\Program Files\Debugging Tools for Windows (x86)`).
- 3) Restart LISA

When a crash occurs, it will generate a large (> 100MB) .dmp file in the directory `<LISA Install Dir>\bin\browser\out\Crash_Mode_Date_xx-xx-xxxx_Time_xx-xx-xxXX`. This is the file that support will need to resolve the issue.

Note: the first time the browser runs, it may fail because it will auto-download a lot of large symbol files from microsoft servers. It should behave normally in subsequent runs (except maybe for a minor slowdown). When the problem is resolved you can unset `_LISA_CRASH_DUMP` to return to normal operating mode.

PART 3 - LISA Web 2.0 - Reference

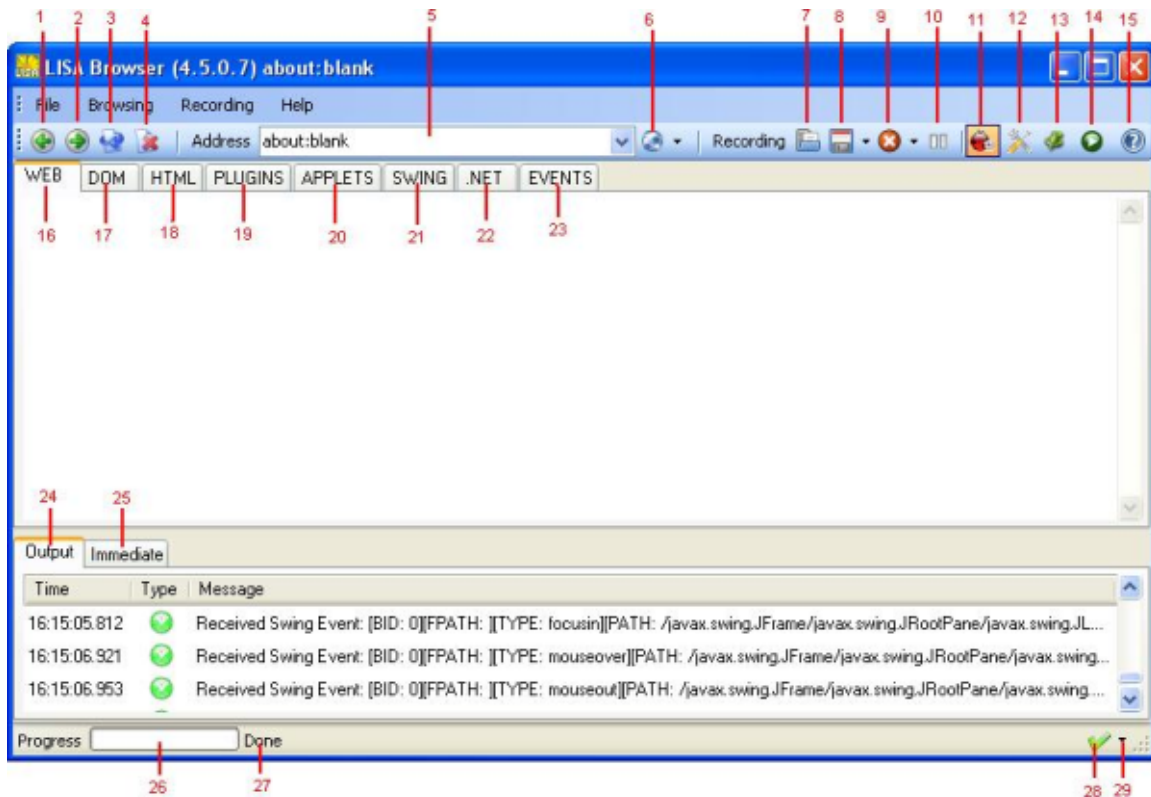
PART 3 - LISA Web 2.0 Reference

The following topics are available.

1. Recorder Reference
2. Debugger Reference
3. Settings Reference
4. XPath syntax Reference
5. Scripting Objects Reference
6. Command line Reference

1. Recorder Reference

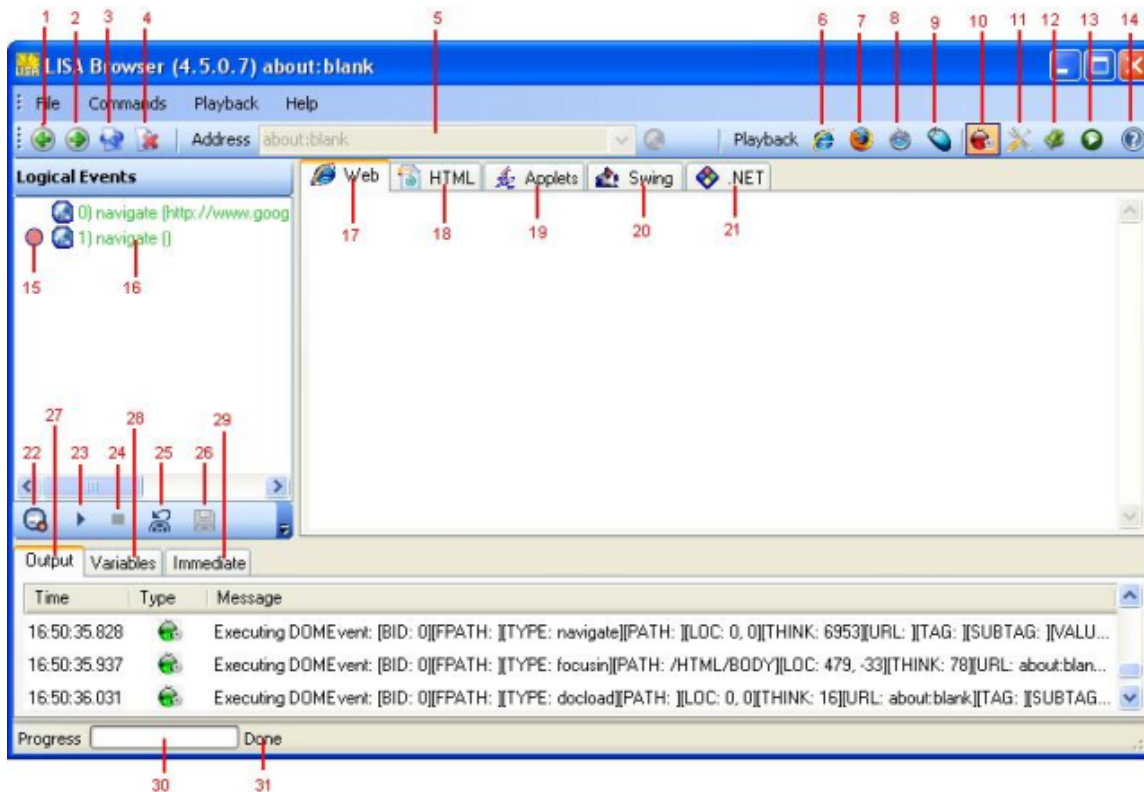
1. Recorder Reference



1. Browser back button
2. Browser forward button
3. Browser reload button
4. Browser cancel button
5. Browser address bar
6. Browser navigate button + drop-down to navigate file system for html, exes and bat files
7. Open existing recording
8. Save current recording (+ drop-down to recording file or to LISA or save capture file)
9. Close browser button (+ drop-down to clear all events)
10. Pause/Resume recording button
11. Debug button to toggle debug view (24, 25)
12. Settings button to bring up Settings Dialog
13. Pin button to force browser on top
14. Toggle button between recording and playback mode
15. Help button to open the help files
16. Web tab to record web site tests
17. DOM tab to inspect live instance DOM tree
18. HTML tab to inspect live instance HTML sources (static, dynamic and script)
19. PLUGINS tab to inspect live instance plugin objects in
20. APPLETS tab to inspect live applets hierarchies
21. SWING tab to inspect live Swing applications hierarchies
22. .NET tab to inspect live .NET WinForms applications hierarchies
23. EVENTS tab to edit and parametrize the currently recorded test
24. Debug Output window to see all log and error messages
25. Debug Command window to evaluate commands
26. Browser download progress bar
27. Browser status text
28. Browser status icon (loading, no errors, errors)
29. Browser javascript debugging menu

2. Debugger Reference

2. Debugger Reference




1. Browser back button
2. Browser forward button
3. Browser reload button
4. Browser cancel button
5. Browser address bar (disabled in playback mode)
6. Turn on/off Internet Explorer as a replay environment
7. Turn on/off Firefox as a replay environment
8. Turn on/off Safari as a replay environment
9. Enable/Disable mouse movements during test execution
10. Debug button to toggle debug view (24, 25)
11. Settings button to bring up Settings Dialog
12. Pin button to force browser on top
13. Toggle button between recording and playback mode
14. Help button to open the help files
15. Set/unset Breakpoint
16. Web 2.0 event (LISA step)
17. Browser tab to host web steps execution
18. HTML tab to inspect live instance HTML sources (static, dynamic and script)
19. APPLETs tab to inspect live applets hierarchies
20. SWING tab to inspect live Swing applications hierarchies
21. .NET tab to inspect live .NET WinForms applications hierarchies
22. Execute highlighted step
23. Execute All steps starting at the highlighted one until breakpoint or end is reached
24. Stop current step execution
25. Reset test: blanks out browsers, clears variables, outputs, etc...
26. Save button (disabled now)
27. Debug Output window to see all log and error messages
28. Debug Variables window to see all current variables and their values
29. Debug Command window to evaluate commands
30. Browser download progress bar
31. Browser status text

3. Settings Reference

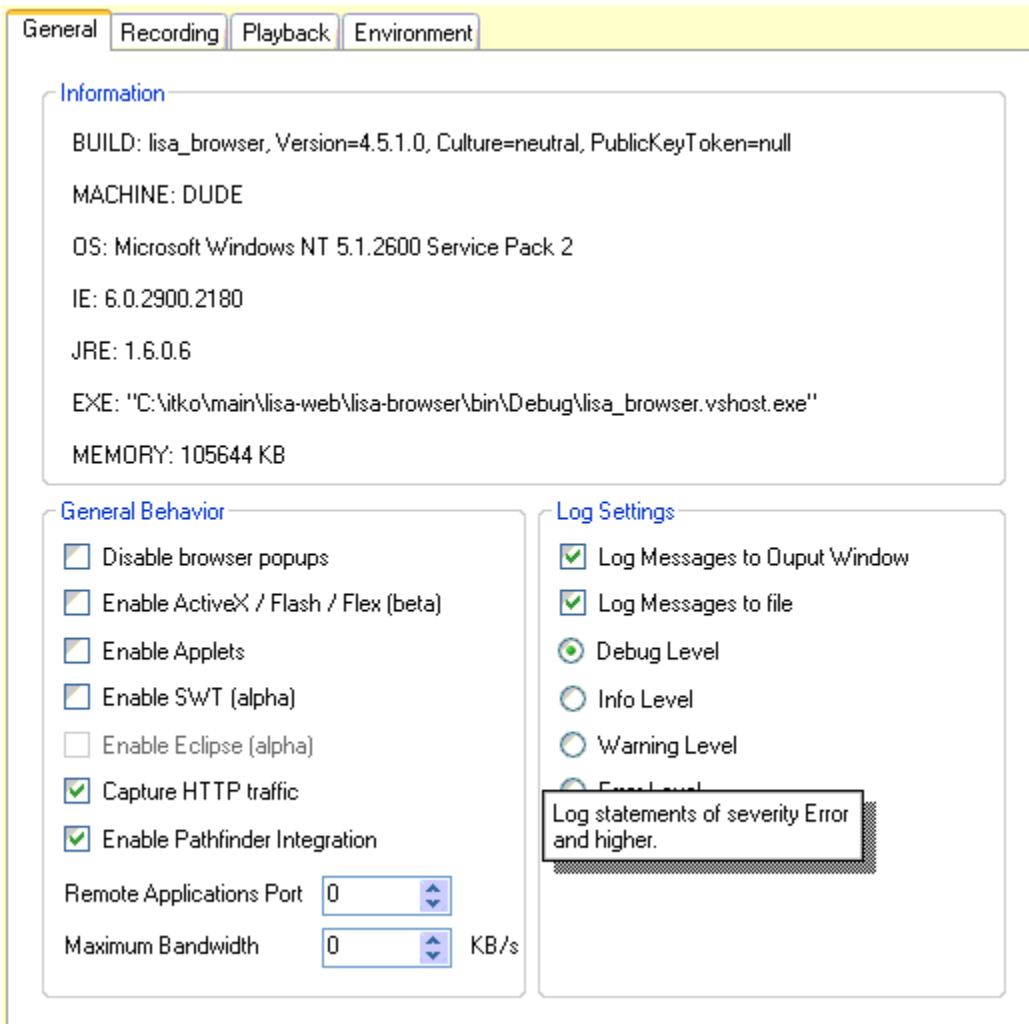
3. Settings Reference

Most of the settings have been mentioned in the documentation or the how-to's but this is the full reference for them.

They are divided in 4 sections: General, Recording, Playback and Environment.

For an online help, click on the  icon in the top right corner of the settings screen and then on one of the fields, to get a tooltip which gives more information about it as shown below:

General Settings:



General | Recording | Playback | Environment

Information

BUILD: lisa_browser, Version=4.5.1.0, Culture=neutral, PublicKeyToken=null

MACHINE: DUDE

OS: Microsoft Windows NT 5.1.2600 Service Pack 2

IE: 6.0.2900.2180

JRE: 1.6.0.6

EXE: "C:\itko\main\lisa-web\lisa-browser\bin\Debug\lisa_browser.vshost.exe"

MEMORY: 105644 KB

General Behavior

☐ Disable browser popups

☐ Enable ActiveX / Flash / Flex (beta)

☐ Enable Applets

☐ Enable SWT (alpha)

☐ Enable Eclipse (alpha)

☒ Capture HTTP traffic

☒ Enable Pathfinder Integration

Remote Applications Port

Maximum Bandwidth KB/s

Log Settings

☒ Log Messages to Output Window

☒ Log Messages to file

☒ Debug Level

☐ Info Level

☐ Warning Level

☐ Error Level

☐ Fatal Level

Log statements of severity Error and higher.

- **Disable browser popups:** acts like a popup blocker. Overrideable in a test with `DISABLE_POPUPS`.
- **Suppress Javascript Dialogs:** alert and confirm dialogs will auto-respond during playback `SUPPRESS_DIALOGS`.
- **Enable ActiveX / Flash / Flex:** Turns on support for ActiveX events. Only reason to turn it off might be faster startup time.
- **Enable Applets:** Turns on support for ActiveX events. Only reason to turn it off might be faster startup time or improved stability (the Java Plugin Interface has some known bugs in certain versions of it, notably 1.5.0_01 through 1.5.0_14, that could cause crashes).
- **Enable SWT:** Turns on support for testing non-Eclipse based SWT applications.
- **Capture HTTP traffic:** Turns on a proxy to capture all HTTP(S) traffic and stores all headers in the event requests. Only reason to turn it off might be faster startup time or already having a proxy.
- **Enable Pathfinder integration:** this causes the browser to receive and decrypt Pathfinder payloads for Pathfinder-enabled applications.
- **Remote Applications Port:** specifies the port to use the control remote applications (Swing, SWT or WinForms). The default, 0, picks the port dynamically.
- **Maximum Bandwidth:** throttles request and response speed to simulate a network with the specified throughput. 0 means no limit.
- **Log messages to Output window:** self-explanatory.
- **Log messages to Output file:** self-explanatory. The log files go in the same directory as the DOM browser.
- **Log Level:** the level of log statements required to be logged in the Output window or file if turned on above.

Recording Settings:

General **Recording** Playback Environment

Recording Options

☒ Use context menus for filters and assertions ☐ Capture HTML changes

☐ Capture applet snapshots Capture Diff Size Bytes

☐ Compress recording files Capture Max Time ms

☒ Verbose recording Save Dialog Triggers

Recording Strategy

Use the following attributes (in this order):

1) <input type="text" value="id"/>	5) <input type="text" value="<none>"/>
2) <input type="text" value="name"/>	6) <input type="text" value="<none>"/>
3) <input type="text" value="<none>"/>	7) <input type="text" value="<none>"/>
4) <input type="text" value="<none>"/>	8) <input type="text" value="index"/>

Except those whose value matches:

For Java Swing and Applets, record:

☒ Component names ☒ Component text

Capture DOM Events

<input type="checkbox"/> navigate	<input checked="" type="checkbox"/> docload
<input checked="" type="checkbox"/> focus	<input checked="" type="checkbox"/> dblclick
<input checked="" type="checkbox"/> mousedown	<input checked="" type="checkbox"/> change
<input checked="" type="checkbox"/> mouseup	<input checked="" type="checkbox"/> contextmenu
<input checked="" type="checkbox"/> mouseover	<input checked="" type="checkbox"/> drag/drop
<input type="checkbox"/> mouseout	<input type="checkbox"/> mousemove
<input checked="" type="checkbox"/> click	<input checked="" type="checkbox"/> keypress
<input checked="" type="checkbox"/> open/close	<input type="checkbox"/> ajax callback

- **Use context menus for filters and assertions:** Override the right-click menu in web pages to popup a custom menu that offers choices about filters, assertions or debugging. Turn it off if your site already uses custom context menus.
- **Capture applet snapshots:** stores in the test the applet screenshots used in applet test editing (browse mode). Turn it off if the tests get too large.
- **Compress recording files:** keep that turned on (used for debugging).
- **Verbose recording:** Records events for which we could not detect an event handler (possibly DOM Level 2). Turn it off for most sites, try to turn it on if you notice some necessary event does not get recorded (happens using ExtJS for instance).
- **Capture HTML changes:** Store the HTML at any click or change event to make it easier for later editing.
- **Capture Diff size:** changes over this size will store the whole response, changes under this size will store the diff.
- **Capture Max time:** the diff above won't be captured if it takes more than this amount of time.
- **Save Dialog Triggers:** for pages with a non text/plain mime type, this decides whether to pop up a "Save As" dialog instead of performing a navigation if the target url matches the regular expression (by default, it does this for Excel, Word, PDF, Text, PowerPoint, Executable and Zip files).
- **Recording strategy:** which HTML attributes to use and in which order when generating XPath expressions.
- **Exclude ids matching:** any element id matching this regular expression won't be used in the auto-generated XPaths.
- **Record component names/text:** use java component names or text (or not) in the XPath generated when recording Java based applications.
- **Capture DOM events:** Turn off any type of DOM event you're not interested in capturing.

Playback Settings:

General Recording **Playback** Environment

Playback Options

☐ Suppress Javascript Dialogs

☐ Fail step on missing target.

☐ Synchronize Ajax calls.

☐ Use Hardware Input

☐ Send Commands Asynchronously

☐ Send Responses back to LISA

Min matching score

Default Browser Mode

☐ Internet Explorer

☐ Firefox

☐ Safari

Playback Speed

Wait multiplier: 0x

Timeout Settings

DOM Load Timeout ms

DOM Lookup Timeout ms

Applet Load Timeout ms

Applet Lookup Timeout ms

ActiveX Load Timeout ms

ActiveX Lookup Timeout ms

- **Fail step on missing target:** Generates a failure instead of the default warning when a target can not be found for a step (that includes browser window, frame, element). Overrideable in a test with `FAIL_MISSING`.
- **Synchronize Ajax Calls:** forces all ajax calls to be executed synchronously. Overrideable in a test with `SYNC AJAX`.
- **Use Hardware Input:** Replays tests by controlling keyboard and mouse. Overrideable in a test with `USE_HARDWARE`.
- **Send Responses back to LISA:** By default responses will not be sent back to LISA to improve performance and memory since it's usually not necessary.
- **Min Matching Score:** How many differences are allowed between a recorded XPath value and the best match found during playback. Overrideable in a test with `MIN_BACTRACKING`.
- **Default Browser Mode:** What browsers to enable by default during playback (pipe-delimited combination of IE, FF and WK). Overrideable in a test with `DEFAULT_BROWSER`.
- **Playback Speed:** The default speed to use to replay test as a multiplier of the recorded speed. 0x is usually the best choice. Overrideable in a test with `PLAYBACK_SPEED` (integer between and 10).
- **XXX Load Timeout:** The maximum amount of time waited before a new page/applet/control load before proceeding. Overrideable in a test with `XXX_LOAD_TIMEOUT`.
- **XXX Lookup Timeout:** The maximum amount of time waited to find an element on a page/applet/control before proceeding. Overrideable in a test with `XXX_LOOKUP_TIMEOUT`.

Environment:

General	Recording	Playback	Environment
Key		Value	
{{&com.itko.lisa.stats.jmx.ITKAgentConnection}}		LISA_JMX_ITKAGENT	
{{&com.itko.lisa.stats.jmx.JBossConnection}}		LISA_JMX_JBOSS3240\	
{{&com.itko.lisa.stats.jmx.JSR160RMIConnection}}		LISA_JMX_JSR160RMI\	
{{&com.itko.lisa.stats.jmx.OracleASConnector}}		LISA_JMX_OC4J\	
{{&com.itko.lisa.stats.jmx.Weblogic9Connector}}		LISA_JMX_WLS9\	
{{&com.itko.lisa.stats.jmx.WeblogicConnector}}		LISA_JMX_WLS6781\	
{{&com.itko.lisa.stats.jmx.WebsphereSOAPConnecti...		LISA_JMX_WASSOAP5X\	
{{&Date}}		LI-DD-DDDD\	
{{&SSN}}		DDD-DD-DDDD\	
{{&Zip}}		D*(5)	
{{alt.lisa.simulator.webservice.classpath}}		\	
{{apple.awt.brushMetalLook}}		false	
{{apple.laf.useScreenMenuBar}}		true	
{{BROWSER_HOME}}		C:\itko\main\lisa-web\lisa-browser\bin\Debug	
{{com.apple.macos.smallTabs}}		true	
{{com.apple.mrj.application.growbox.intrudes}}		true	
{{com.itko.lisa.stats.jmx.ITKAgentConnection,com.i...			
{{com.itko.lisa.stats.jmx.JBossConnection,com.itko.li...			
{{com.itko.lisa.stats.jmx.WeblogicConnector,com.itk...			
{{EXAMPLES_HOME}}		{{LISA_HOME}}/examples	
{{file.encoding}}		UTF-8	
{{Functional Report, com/itko/lisa/report/layout/Fun...			
{{gui.show.memory.status}}		false	
{{ice.browser.cache.size}}		1048576	
{{ice.browser.http.agent}}		Mozilla/4.0 (compatible; MSIE 6.0; Windows N	
{{ice.browser.verbose}}		false	
{{jasper.multi.report}}		\	
{{jasper.single.report}}		\	
{{javax.xml.parsers.DocumentBuilderFactory}}		org.apache.xerces.jaxp.DocumentBuilderFacto	
{{javax.xml.parsers.SAXParserFactory}}		org.apache.xerces.jaxp.SAXParserFactoryImpl	
{{javax.xml.transform.TransformerFactory}}		org.apache.xalan.processor.TransformerFactor	
{{laf.default.url}}		https://license.itko.com	

This tab shows the list of global environment variables available to the browser, as read from **lisa.properties** and **local.properties**.

LISA Driver Settings:

In addition to the settings above that can be overridden from LISA in a test case or in the local.properties, the following are available:

- **lisa.browser.launch.timeout:** The amount of time allowed for a browser to launch (default is 10,000). Specify in local.properties.
- **lisa.browser.exec.timeout:** The amount of time allowed for a step to execute (default is 300,000). Specify in local.properties.
- **lisa.browser.max.instances:** The maximum number of browser instances per machine (default is 25). Specify in local.properties.
- **lisa.browser.client.user.single:** Whether to run staged browsers using the same user account as the currently logged-in users (default is true). Specify in local.properties.
- **lisa.browser.base.port:** The first port to use in the range of ports available to control web browsers (default is 0 for dynamic value). This normally does not need to be modified except in very secure environments that lock down some local ports. Specify in local.properties.
- **lisa.browser.client.user.<user name>=<encrypted password>:** when lisa.browser.client.user.single is set to false, browser instances will use the specified windows user accounts to run tests. If not enough user accounts are specified in this manner and the currently logged-in user has admin privileges, user accounts will be dynamically created to run the tests (and deleted at the end). To obtain an encrypted password, run the command line: `lisa_browser.exe -m encrypt -in <clear text>`.
- **lisa.browser.share.subprocess.state:** Whether to run a sub-process using the same browser instance as parent test (default is false). Specify in configuration of sub-process.
- **lisa.browser.swing.port:** The first port to use in the range of ports available to control Swing applications (default is 0 for dynamic value). Specify in local.properties.
- **lisa.browser.base.port:** The first port to use in the range of ports available to control web browsers (default is 0 for dynamic value). Specify in local.properties.

4. XPath syntax Reference

4. XPath syntax Reference

The XPath syntax supported to identify html elements in web 2.0 tests is a subset from the standard XPath specification, as described at [XPath](#)

2.0 specification. Roughly speaking, the supported vs. not supported functionality is as follows:

- `<Tag1>!/<Tag2>` to look for an element child is supported (e.g. `TR[1]/TD[2]`).
- `<Tag1>!!<Tag2>` to look for an element descendant is supported (e.g. `TABLE//INPUT`).
- `..` to select the parent of an element is supported (e.g. `TABLE//TD[@id='abc']/../TD[1]`).
- `>>` and `<<` to select the next and previous sibling of an element respectively are supported - the standard axis names following-sibling or preceding-sibling can be used instead with `>>` and `<<` just being shorthand for it - (e.g. `TABLE//TD[@id='abc']/../>>/TD[1]`).
- `<Tag>[@AttributeName='AttributeValue']` to look for an element with the specified tag meeting the `AttributeName='AttributeValue'` condition is supported (e.g. `INPUT[@id='abc']`).
- `<Tag>[text()='innerText']` to look for an element whose inner text equals the specified one is supported (e.g. `TD[text()='Hello']`).
- `<Tag>[matches()='regex']` to look for an element whose inner text matches the supplied regular expression is supported as a proprietary extension (e.g. `TD[matches()='\\s*Hello\\s*']`).
- Other XPath axes are not supported.
- Only XPath expressions returning a unique node are supported. Expressions returning potentially multiple nodes will simply return the first one matching the expression.
- XPath functions (in the fn: namespace) and operators (arithmetic or boolean) are not supported.

5. Scripting Objects Reference

5. Scripting Objects Reference

Both in its debug window, and in its script filters, the DOM browser supports standard javascript and java (beanshell) syntax, but additionally built-in commands and some useful objects are defined with the following APIs:

_arg: A DOM element, or Java component, or ActiveX accessible element which is specified either as the element in a filter, or is the last recipient of an event in the Immediate debug window while recording or debugging.

lisa: A global javascript object available on every HTML page:

- public object dbQuery(string connectionString, string query, [Optional] bool cache)
-
- Given a database connection string and a SQL query (and optionally a boolean to indicate whether to cache the results), this will return a DataSet object (see reference below).
- public void disableBlur(bool disable)
-
- When passed true, this will disable all the onBlur event handlers on any subsequent page. This can make it easier to debug or add filters on popup menus that would otherwise disappear when losing focus.
- public string download(string url, string path)
-
- Will download the resource at the given url to the specified path. Useful to download files without having to go through the Save As dialogs.
- public object fileQuery(string path, [Optional] string query, [Optional] bool cache)
-
- Generates a DataSetWrapper (see reference below) from an Excel file and optionally a SQL query and a flag to cache the results. If no query is given, the whole first worksheet is returned in the dataset.
- public void fire(HTMLDocument document, string xpath, string evtName)
-
- Manually fires the supplied event on the element identified by the specified xpath.
- public string open(string path)
-
- Returns the contents of the file at the specified path into a string.
- public string pathfind(string xpath)
-
- Returns the node value of the `event.pathfinder` xml selected by the specified xpath expression.
- public void print(object o)
-
- Prints a textual representation of the argument to the debug Output window.
- public object select(HTMLDocument document, string xpath)
-
- Returns the html element by specified xpath.
- public string showHandlers(HTMLDocument document, string xpath)
-
- Returns a list of all the javascript functions used as event handlers on the specified elements and all its parents.
- public string upload(string url, string path)
-
- Similar to download, but uploads a resource from the specified path to the specified url.
- public void evaluate(string code)
-
- Executes arbitrary C# .NET code. Can be used as last resort if desired functionality is not available.
- public bool compareFiles(string file1, string file2)
-
- Returns whether the contents of file1 and file2 are exactly identical.

- public double diff(string file1, string file2)
-
- Returns a value that indicates how different the files are (based on their length, average value and standard deviation).
- public void jsimport(HTMLDocument document, string jsfile)
-
- Automatically imports a js file into the specified page, making its variables and functions available to subsequent filters on the page. Note that any .js files in the browser directory will be automatically offered as an import in the filter drop-down of DOM event filters.
- public void javainport(string javafile)
-
- Automatically imports a java source file into the currently running java process, making its functions available to subsequent filters in the test. Note that any .java files in the browser directory will be automatically offered as an import in the filter drop-down of java event filters.

DataSetWrapper: A javascript class that encapsulate the result of SQL queries:

- public string asText()
-
- Returns a textual representation of the dataset where each cell is contained in brackets [].
- public int columnCount()
-
- The number of columns in the dataset.
- public string columns(int index)
-
- The name of the column as the specified index.
- public int count() or public int length()
-
- The number of rows in the dataset.
- public object rows(int index)
-
- Returns the DataRowWrapper (see reference below) object at the specified index.
- public string toString()
-
- A textual representation of the dataset.

DataRowWrapper: A javascript class that encapsulate a single row of a DataSetWrapper:

- public object cells(object indexOrName)
-
- Returns the value in the cell at the specified index or in the column with the specified name.
- public string toString()
-
- A textual representation of the row.

6. Command line Reference

6. Command line Reference

lisa_browser.exe supports the following options:

lisa_browser.exe [-m] [options]

Where:

-m recorder launches the recorder
 -m playback launches the debugger
 -m drive launches the load test console
 -m udpate launches the software update dialog

And options are one or more of the following:

-a autstarts the test specified by -f in playback or drive mode - default is false
 -n is the number of instances to autostart in drive mode - default is 0
 -f specifies a recording file to load on startup - default is none
 -c specifies a config file to load on startup - default is none
 -i makes the driver console invisible in drive mode - default is false
 -x autoexits the program on test end if it was autostarted - default is true
 -stg specifies a path to an xml file that lists instances along with test and config files to use

See [Load testing](#) for the syntax required by the -stg option.

PART 4 - LISA Web 2.0 - Videos

PART 4 - LISA Web 2.0 - Videos

http://www.itko.com/download/release/lisa_browser/docs/web20videos.html

The other information available at this location is not up-to-date. Please refer to the videos only.

NOTE: This site requires a valid login. If you have any questions, please contact your iTKO Sales Representative. Thank you.

PART 5 - LISA Web 2.0 - Repository

PART 5 - LISA Web 2.0 - Repository

The following topics are available...

1. Instructions
2. Always update
3. Update with major revisions changes
4. First time update (i.e. only if you're missing these files)

1. Instructions

1. Instructions

If you can not use the browser update mechanism for policy or connectivity reasons, you can download the updated files manually from this page. There are no installations steps.

Simply download the following files to the directory: <LISA install dir>\bin\browser. The only exceptions are: web20bridge.jar and jdbridge.jar that should go into the <LISA install dir>\lib directory
djbridge.dll and jdglue.dll that should go into the <LISA install dir>\bin directory

You usually have to download only a few files as indicated below. If you are unsure which ones to download, you can always download them all. Using manual download, you are responsible for backing up the files you overwrite in case you need to revert later.

2. Always update

2. Always update

- [lisa_browser.exe](#)
- [appletcallback.jar](#)
- [swingcallback.jar](#)
- [web20bridge.jar](#)

3. Update with major revisions changes

3. Update with major revisions changes

- [applet-monitor.dll](#)
- [dotnet-callback.dll](#)
- [dotnet-monitor.dll](#)
- [injector.dll](#)
- [lisa_browser.XmlSerializers.dll](#)
- [global-hook.dll](#)
- [jdbridge.jar](#)
- [djbridge.dll](#)
- [jdglue.dll](#)

4. First time update (i.e. only if you're missing these files)

4. First time update (i.e. only if you're missing these files)

- [Interop.SHDocVw.dll](#)
- [Interop.TidyATL.dll](#)
- [Interop.WebKit.dll](#)
- [Microsoft.mshtml.dll](#)
- [SciLexer.dll](#)
- [ScintillaNET.dll](#)
- [TidyATL.dll](#)
- [Microsoft.Office.Interop.Excel.dll](#)
- [Office.dll](#)
- [swt.jar](#)

PART 6 - LISA Web 2.0 - FAQ

Q: Can I test Flash and Flex applications?

A: The answer is "it depends".

First let's clarify Flash vs. Flex: Flex is an extra layer on top of Flash that helps programmers render certain layouts and controls more easily within a Flash VM. These days, non-Flex Flash applications are used almost only for video and/or animations so they are unusual in enterprise applications, so I'll focus on Flex in the remainder of the discussion.

Generally speaking there are 2 ways to test or automate Flex applications:

- using the so-called MSAA (accessibility interface) which provides ActiveX control clients with some level of visibility inside them. Support for the MSAA is pretty variable even within Flex applications, depending how they're coded and compiled. This means, in the best case that you can have visibility into every label, button, text field, drop down, etc...or in the worst case that the whole component (or large parts of it) are simply seen as graphics blobs. In the latter case all automation has to revert to coordinate based events and testing becomes very brittle, data can't be extracted - other than using OCR (optical character recognition) which some vendors do, with little success. To get an idea of how much detail is available in a given Flex app, you can download and use the `accexplorer32.exe` Microsoft tool.

This is the approach employed by the LISA browser: a Flash control (and thus a Flex one too) is embedded in a web browser as an ActiveX control or a plugin, so the the LISA browser has visibility into it inasmuch as it can see ActiveX controls and their subcontrols. This generally (but not always) is sufficient for small or simple controls embedded in HTML pages but rarely so for full-fledged apps that use the whole page or complex controls. Unfortunately there is no way to tell in advance and you'll have to try it to find out (or use `accexplorer32.exe` to have an idea).

The advantage of this technique is that it requires no code changes or recompilation of the Flex app to work, so it's been favored by many vendors in the past (including QTP) but its limitations are starting to push some vendors toward the second approach:

- using a Flex agent that allows clients to communicate directly with the Flash VM and its controls, variables and functions. This is the approach taken by vendors like GorillaLogic's FlexMonkey (or to a lesser extent Flex Selenium, which uses `ExternalInterface` objects to access the VM through javascript). When this approach is possible, it is much more robust and much preferable to the previous one, but as of Flex 3 it still required instrumentation and/or recompilation of the Flex application. This may no longer be needed now or in the future but I am not sure about the details.

We will probably invest into this alternative at some point but there is no ETA on this yet.