1. LISA Web 2.0 Guide . . . 2
1.1 Web 2.0 User Guide . . . 2
1.1.1 Web 2.0 Introduction . . . 2
1.1.1.1 Web 2.0 System Requirements . . . 3
1.1.1.2 Web 2.0 Technologies and Operating Environments . . . 3
1.1.1.3 Web 2.0 Getting Started with LISA Browser . . . 3
1.1.1.3.1 Web 2.0 Browser Menu . . . 4
1.1.1.3.2 Web 2.0 Browser Toolbar . . . 5
1.1.1.3.3 Web 2.0 Browser Settings . . . 7
1.1.1.3.4 Web 2.0 Browser and Extension Updates . . . 15
1.1.1.3.5 Web 2.0 Browser Architecture . . . 17
1.1.2 Web 2.0 Recording Mode . . . 19
1.1.2.1 Recording Example . . . 21
1.1.2.2 Recording a Swing Test . . . 23
1.1.2.3 Recording an Applet Test . . . 24
1.1.2.4 Different Views of a Web Page . . . 26
1.1.2.5 After Recording . . . 30
1.1.3 Web 2.0 Playback Mode . . . 35
1.1.4 Web 2.0 Edit Mode . . . 38
1.1.4.1 Web 2.0 Event Types . . . 39
1.1.4.2 Web 2.0 Logical Events . . . 43
1.1.4.3 Web 2.0 Object Details . . . 45
1.1.4.4 Web 2.0_Filters . . . 50
1.1.4.5 Web 2.0_Assertions . . . 52
1.1.4.6 Web 2.0 Datasets . . . 54
1.1.4.7 Web 2.0 Editing Steps . . . 55
1.1.5 Web 2.0 Debugging . . . 56
1.1.6 Web 2.0 Setting up ADF Extensions . . . 59
1.1.7 Web 2.0 Running Browser Standalone . . . 60
1.1.8 Web 2.0 Troubleshooting . . . 61
1.1.9 Web 2.0 XPath Syntax . . . 61
1.1.10 Web 2.0 Scripting Objects . . . 61
1.1.11 Web 2.0 Command Line . . . 62
1.2 Web 2.0 How Tos . . . 62
1.2.1 How To Learn About Websites and Frameworks . . . 63
1.2.2 How To Generate Random Data . . . 63
1.2.3 How To Capture Dynamic HTML for later test editing . . . 64
1.2.4 How To Deal with Time-sensitive Events . . . 66
1.2.5 How To Parameterize dynamic data entry in loops . . . 67
1.2.6 How To Deal with Dynamic Elements . . . 69
1.2.7 How To Extract Complex Data from a Page . . . 70
1.2.8 How To Ajax Auto-complete Fields . . . 73
1.2.9 How To Write Custom Web 2.0 Steps . . . 75
1.2.10 How To Write Cross-browser Tests . . . 75
1.2.11 How To Use Pathfinder Integration . . . 77
1.2.12 How To Write Java Swing and WebStart Tests . . . 79
1.2.13 How To Debug a Test . . . 82
1.2.14 How To Use Global Filters and Global Assertions . . . 86
1.2.15 How To Interact with External Resources . . . 86
1.2.16 How To Run Load Tests . . . 90
1.2.17 How To Run in a Non-privileged Account or on 64 bit Operating Systems . . . 90
1.2.18 How To Record and Replay against Non US-English Websites . . . 91
1.2.19 How To Run in Crash Dump Mode . . . 92
1.2.20 How To Set a Web 2.0 Browser Timeout . . . 92
1.3 Web 2.0 Videos . . . 92
1.3.1 Web 2.0 Tutorials Part 1 . . . 93
1.3.2 Web 2.0 Tutorials Part 2 . . . 93
1.3.3 Web 2.0 Tutorials Part 3 . . . 93
1.3.4 Web 2.0 Tutorials Part 4 . . . 93
1.3.5 Web 2.0 Tutorials Part 5 . . . 93
1.3.6 Web 2.0 Tutorials Part 6 . . . 93
1.3.7 Web 2.0 Topics - Webstart . . . 93
1.3.8 Web 2.0 Topics - Win32 (Native Apps) . . . 93
1.3.9 Web 2.0 Topics - Filters . . . 93
1.3.10 Web 2.0 Topics - Datasets . . . 93
1.4 Web 2.0 Repository Instructions . . . 93

# LISA Web 2.0 Guide

The LISA Web 2.0 documentation consists of the following chapters.

**Web 2.0 User Guide**
**Web 2.0 How Tos**
**Web 2.0 Videos**

Web 2.0 Repository Instructions

## Web 2.0 User Guide

LISA Web 2.0 allows LISA to emulate a web browser, recording events at the DOM (Document Object Model) level and playing back those events as a browser during test execution. These events can include mouse clicks, mouse movements, keys being typed, and others.

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 DOM-level testing. A Web 2.0 test can easily interact with frames, JavaScript, CSS, Ajax, Plugins, applets and so forth. Many websites make heavy use of these technologies, Ajax in particular, and these are usually referred to as Web 2.0 sites, hence the term *Web 2.0 tests*.

For more information about Web 2.0, see Web 2.0 How Tos.

The following topics are available in this section.

Web 2.0 Introduction

Web 2.0 Recording Mode

Web 2.0 Playback Mode

Web 2.0 Edit Mode

Web 2.0 Debugging

Web 2.0 Setting up ADF Extensions

Web 2.0 Running Browser Standalone

Web 2.0 Troubleshooting

Web 2.0 XPath Syntax

Web 2.0 Scripting Objects

Web 2.0 Command Line

## Web 2.0 Introduction

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, and more) as is so often necessary in the enterprise software world.

Web 2.0 tests are no exception and can be mixed with any other type of step. You can do this easily.

Since version 3.5, HTTP-level web testing has been fully supported.

HTTP-level testing works by having LISA install a proxy between itself and the web server. It then captures the HTTP and HTTP/S 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, consult the Recording a Website chapter in the *User Guide*.

By contrast, Web 2.0 testing works by letting LISA emulate a web browser. It lets you record events at the DOM level (such as mouse clicks, mouse movements, keys being typed, and so forth) 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 because it is only aware of URLs. However, it is very lightweight so it is not well suited to massive load-testing. DOM-level testing 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 websites make heavy use of these technologies, Ajax in particular.

In addition to web testing, 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 Flash and Flex applications.

The following topics are available in this section.

Web 2.0 System Requirements

Web 2.0 Technologies and Operating Environments

Web 2.0 Getting Started with LISA Browser

## Web 2.0 System Requirements

In addition to the LISA installation on a Windows NT or later computer, 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

⚠ Running Web 2.0 as a Windows service is not supported.

## Web 2.0 Technologies and Operating Environments

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

In the pure web realm, any server side-technology or operating environment is supported (because it is irrelevant from the client's perspective), whereas on the client side Web 2.0 can run Internet Explorer or Mozilla Firefox, 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 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.

⚠ 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.

As of LISA 6.0, Web 2.0 no longer supports Firefox 2 playback (Gecko 1.8.x). We do support up to and including Firefox 3.6 (Gecko 1.9.x).

## Web 2.0 Getting Started with LISA Browser

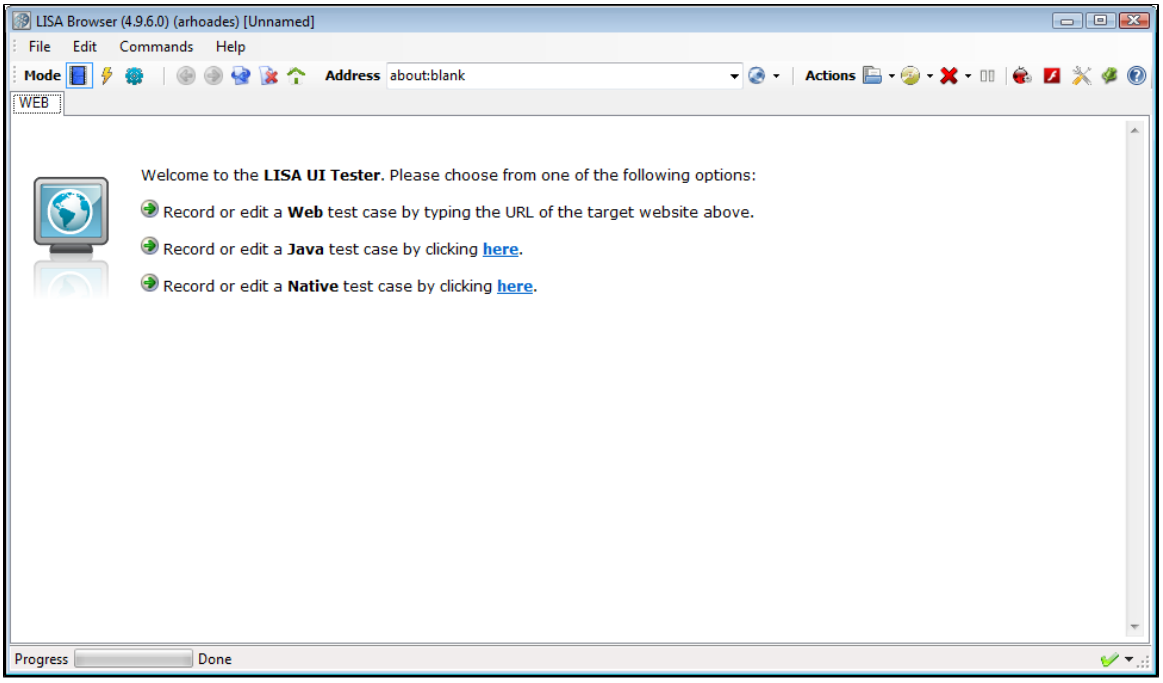
To open the Web 2.0 browser, click the Record  icon on the test case menu. This opens a recording menu.


Select **Record Test Case for User Interface > Web Recorder (DOM Events)** or, from the main menu, select **Actions** > **Record Test Case for User Interface > Web Recorder (DOM Events)**.

This launches the browser that is used to record and play back Web 2.0 tests.

The first time the browser is launched, there is a wait dialog that indicates it is synchronizing its initial state. Subsequent invocations will not do this because synchronization will happen as required.

Typically, the first thing you will do is record a session. When you open the Web 2.0 browser, it opens in 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 that let you record or edit a Web or Java or a Native test case.

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

> More details can be found in the subsequent sections.
>
> Web 2.0 Browser Menu
>
> Web 2.0 Browser Toolbar
>
> Web 2.0 Browser Settings
>
> Web 2.0 Browser and Extension Updates
>
> Web 2.0 Browser Architecture

## Web 2.0 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 contains four sub-menus.

#### 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, where you can set the browser settings.
- **Edit > Internet Options**: Opens the Internet Properties dialog, where you 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.
- **Help > Extension Updates**: Opens the LISA Extension update dialog.

## Web 2.0 Browser Toolbar

The LISA Browser opens in the Recording mode by default.


The Mode button is on the left side of the LISA browser toolbar. It lets you select the mode of operation within the browser.

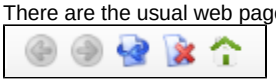
- **Recording Mode**: Where you can record the operation.
- **Edit Mode**: Where you can edit the transactions.
- **Playback Mode**: Where you can play back the recorded operation.

By default the Recording mode is selected.

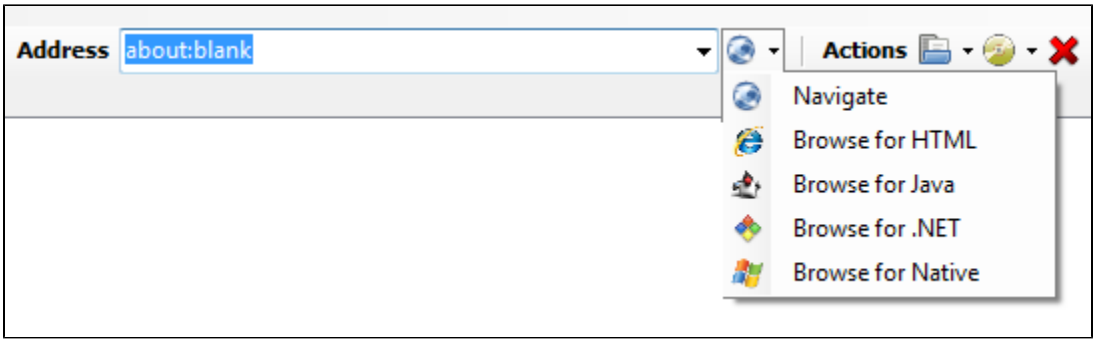
You can record the web page and play back the recording by clicking the Playback mode.

Click the Edit mode to view add/delete the Logical, Physical events and view the Object details that are described later.

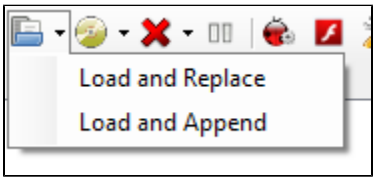
There are the usual web page buttons to take you Back, Forward, Reload, Abort and Home page.



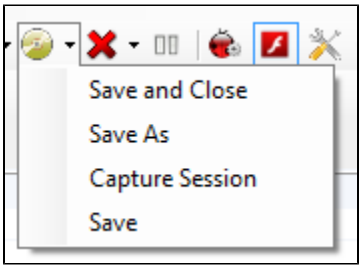
You can enter the web page address in the Address bar provided or select the Go  button to navigate and open a HTML/ Java/.NET or Native page.



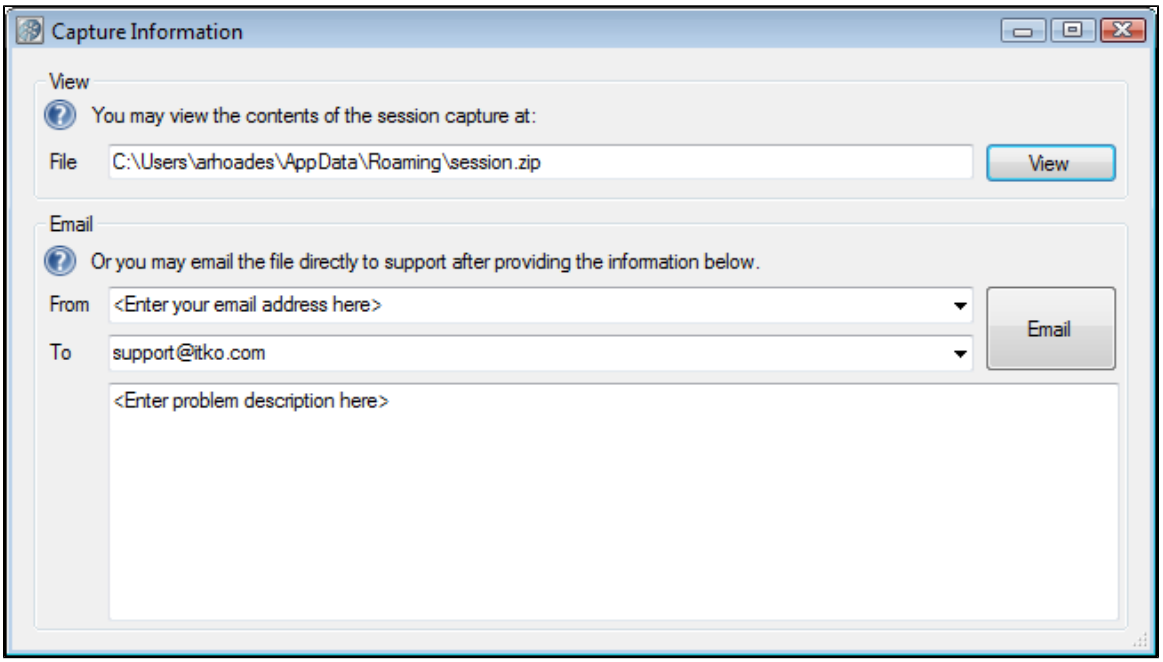

You can **Load and Replace** or **Load and Append** with the Load  button. The Load button is not typically used when using the browser within LISA. It is useful when using the browser as standalone.



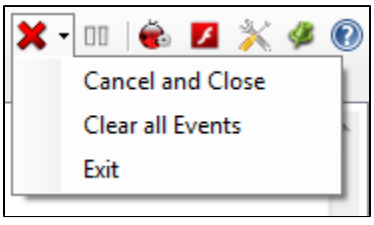
You can save the recordings using the Save button. The Save button is what commits the recording to a test case and closes the browser. If you are in standalone mode, it saves the recording to a file. This button also has a drop-down that lets you pick a non-default behavior: you can choose to explicitly save to a test case and close the browser, save to a standalone recording file, save to a session capture file, or save and continue (the browser remains open).




If you choose to **Capture Session**, you see the following screen. On this screen, you can choose to view the session data captured, or to send it directly to ITKO Support for analysis.




You can **Close and Cancel** the recording using the Cancel button. The Cancel button 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.




Additional toolbar icons are:

| Icon | Description |
| --- | --- |

| | |
|---|---|
|  | The Pause button lets you navigate without recording. Recording resumes when it is pressed again. It is useful to skip some undesired events. |
|  | The Debug button lets you open or close the debug window. |
|  | The Settings button opens the Settings dialog that lets you configure global behaviors of the both the recorder and the debugger. |
|  | The Flex button enables Flex recording. |
|  | 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. |

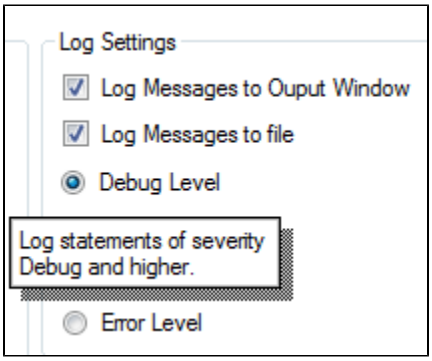
The Debug  and Settings  buttons are described in detail in the next section.

## Web 2.0 Browser Settings

LISA Browser Settings let you control how recording and playback will work.

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

While the Settings window is open, you can quickly find the meaning of a setting by clicking the Help  icon at the top right corner of the window and then the chosen setting, or position the mouse over the chosen setting and click the F1 key. You will get a tooltip for that field.




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 tooltip.

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 tooltip, you will see that 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.

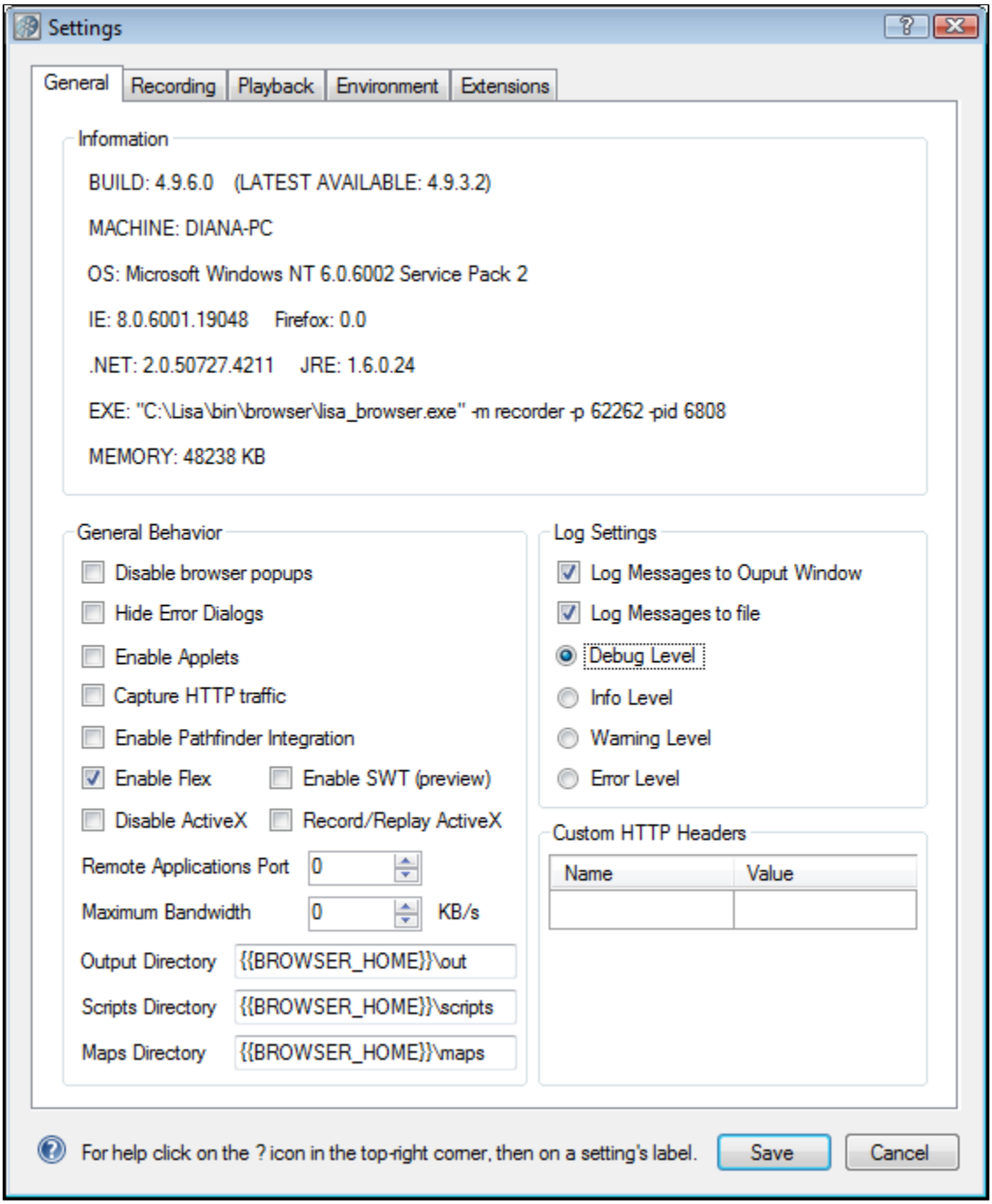
LISA browser settings are saved to both the settings file (on the local computer) and to the project's settings directory (as **lisa_browser.ini**), if LISA browser is executed in the context of a test. The settings are loaded in the order of local file followed by project file. If the LISA browser is executed in standalone mode, then only the local settings file is read/saved to.

Settings are divided in five sections: General, Recording, Playback, Environment, and Extensions.

General Tab
Recording Tab
Playback Tab
Environment Tab
Extensions Tab

## General Tab

The General Tab is used to set basic options.




#### Information

In the **Information** box you see information about the system that runs the LISA browser.

#### General Behavior

In the **General Behavior** box you select general behavior of the browser properties.

- **Disable browser popups**: Acts like a pop-up blocker. Overrideable in a test with `DISABLE_POPUPS`.
- **Hide Error Dialogs**: Log severe errors without producing a pop-up dialog.
- **Enable Applets**: Turn on support for ActiveX events. The 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).
- **Capture HTTP traffic**: Turn on a proxy to capture all HTTP(S) traffic and stores all headers in the event requests. The only reason to turn it off might be faster startup time or already having a proxy.
- **Enable Pathfinder integration**: Cause the browser to receive and decrypt Pathfinder payloads for Pathfinder-enabled applications.
- **Enable Flex**: Enable support for recording Flex controls.
- **Enable SWT (preview)**: Turn on support for testing non-Eclipse based SWT applications.
- **Disable ActiveX**: Prevent ActiveX/Flash/Flex controls to render in the browser (useful in load testing).
- **Record/Replay ActiveX**: Capture and replay events occurring within ActiveX/Flash/Flex controls.
- **Remote Applications Port**: Specify the port to use to control remote applications (Swing, SWT or WinForms). The default, 0, picks the port dynamically.
- **Maximum Bandwidth**: Throttle request and response speed to simulate a network with the specified throughput. 0 means no limit.
- **Output Directory**: The directory where all browser logs will be stored.
- **Scripts Directory**: The directory from which all *.jx, *.java, *.cs files will be picked up automatically for use in tests.
- **Maps Directory**: The directory from which all *.map files will be picked up automatically to use external resources.

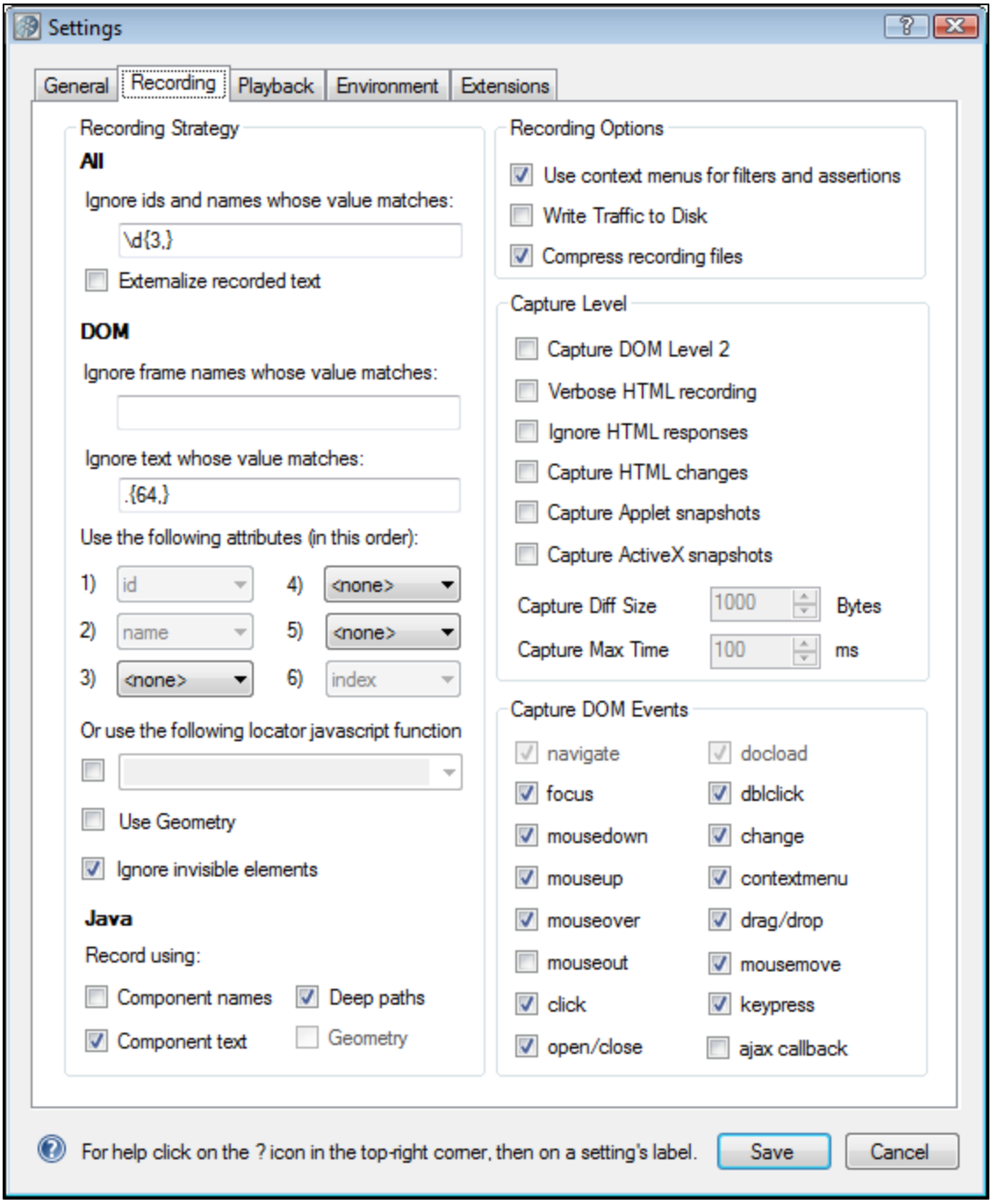
#### Log Settings

In the **Log Settings** box you can specify log-related information.

- **Log messages to Output window**: Self-explanatory.
- **Log messages to file**: Self-explanatory. The log files go in the same directory as the DOM browser.
- **Debug / Info / Warning / Error Level**: The level of log statements required to be logged in the Output window or file.

### Recording Tab

The Recording tab is used to set Recording options. It has four sections: Recording Strategy, Recording Options, Capture Level, and Capture DOM Events.




**Recording Strategy**

- **Ignore ids and names whose value matches**: Ignore all ids and names whose value matches the specified regular expression.
- **Externalize recorded text**: Automatically translate hardcoded text values into variables.
- **Ignore frame names whose value matches**: Automatically ignore frame names and use indexes instead.
- **Ignore text whose value matches**: Automatically ignore text that matches the specified regular expression during recording.
- **Use the following attributes (in this order)**: Use the specified DOM attributes in the order supplied during recording.
- **Or use the following locator javascript function**: Use the specified custom JavaScript function for recording. This function takes a DOM event as an argument, and must return a pair of function names: one a target locator, the other an API call on the target.
- **Use Geometry**: Attempt to use geometric aspects of the page for recording if XPaths have more than two elements.
- **Ignore invisible elements**: Ignore invisible elements while recording.
- **Record using: Component names**: Component names as set in Java code will be used in recorded paths. Do not use unless developers explicitly set those names to help with testing.
- **Record using: Deep paths**: Component paths will be recorded as //component[f=getText()='***'] if possible. This strategy is more robust but less specific.
- **Record using: Component text**: Use labels or text of non-editable components in recorded paths (if they are unique).
- **Record using: Geometry**: Future option, where the location of the page elements is specified using geometric relationship rather than the DOM Element XPath. This option is disabled in the current version of the LISA browser.

#### Recording Options

- **Use context menus for filters and assertions**: Override the right-click menu in web pages to display a custom menu that offers choices about filters, assertions or debugging. Turn it off if your site already uses custom context menus.
- **Write traffic to disk**: Write HTTP(S) traffic to a log file if **Capture HTTP(S) Traffic** is enabled.
- **Compress recording files**: Used for debugging; keep turned on.

#### Capture Level

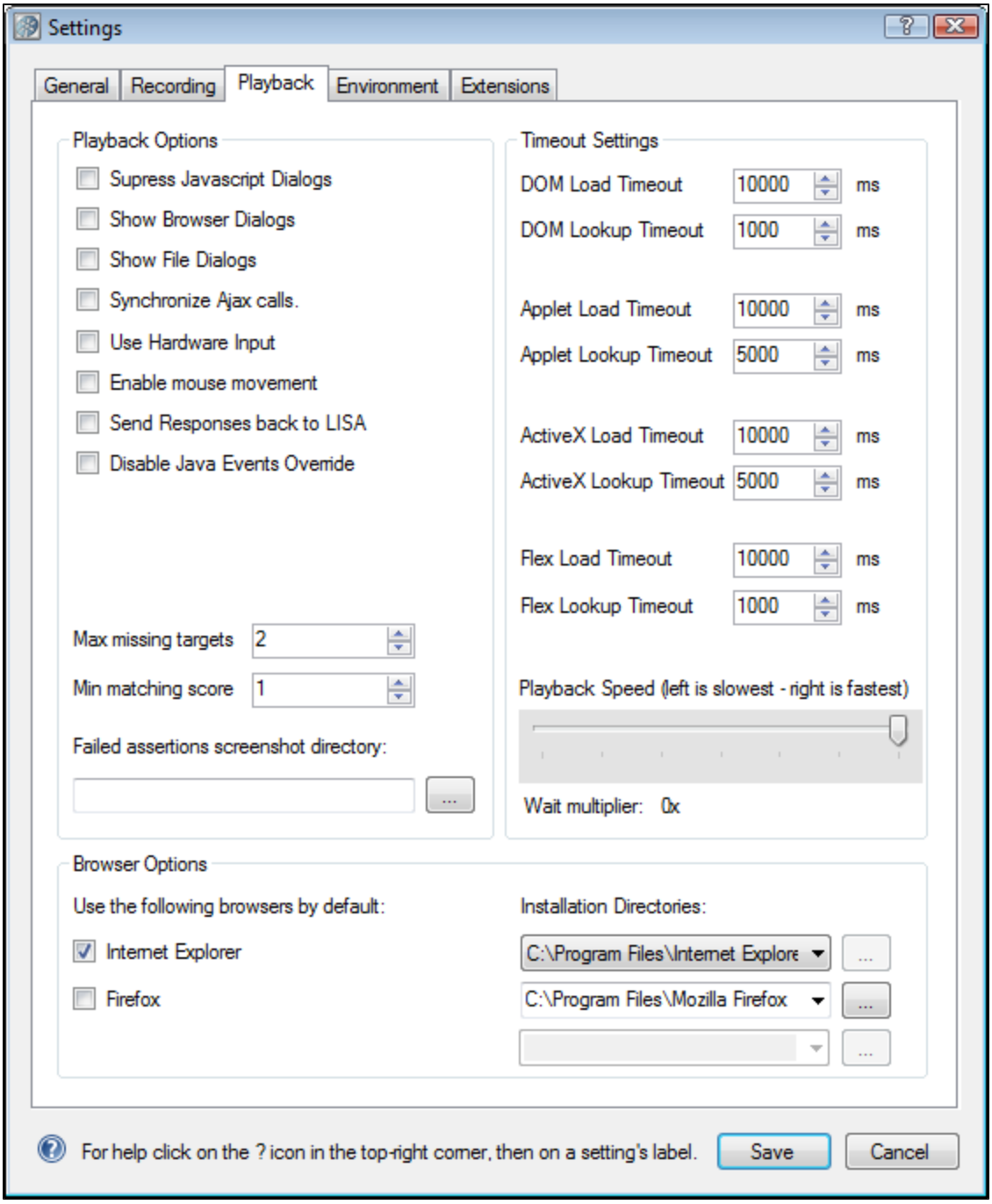
- **Capture DOM Level 2**: Intercept Level 2 DOM event handlers attachment/detachment. This does not interfere with traditional Level 0 DOM event capture/replay.
- **Verbose HTML recording**: Attempt to capture events for which the browser could not detect the handler (ex DOM Level 2 handlers). Only use this option when an event does not get recorded.
- **Ignore HTML responses**: When enabled, the response HTML is not stored within the test events. Hence no HTML is displayed on the events editor.
- **Capture HTML changes**: Store the HTML at any click or change event to make it easier for later editing.
- **Capture applet snapshots**: Store in the test the applet screenshots used in applet test editing (browse mode). Turn it off if the tests get too large.
- **Capture ActiveX snapshots**: Store in the test events the captured screen snapshots of the ActiveX/native dialogs. Turn this option off if the tests are taking too much disk space.
- **Capture Diff size**: Changes over this size will store the whole response; changes under this size will store the diff.
- **Capture Max time**: Set a time limit on the capture time of the diff. The diff in **Capture Diff size**will not be captured if it takes more than this amount of time.

#### Capture DOM Events

- **Capture DOM events**: Turn off any type of DOM event you do not want to capture.

### Playback Tab

The Playback Tab is used to set Playback settings options.



The Playback Settings tab is made up of three sections: Playback Options, Timeout Settings, and Browser Options.

**Playback Options**

- **Suppress Javascript Dialogs**: Attempt to suppress browser error and warning messages. Override with {{SUPPRESS_DIALOGS}}.
- **Show Browser Dialogs**: Attempt to replay browser dialogs from the GUI if checked, programmatically otherwise. Override with {{SHOW_BROWSER_DIALOGS}}.
- **Show File Dialogs**: Show the file upload dialog during playback instead of filling in the textbox value. Override with {{WEB_SHOW_FILE_DIALOGS}}.
- **Synchronize Ajax Calls**: Force all Ajax calls to be executed synchronously. Override in a test with {{SYNC_AJAX}}. If this box is checked, the step will not return until the Ajax call completes.
- **Use Hardware Input**: Replay tests by controlling keyboard and mouse. Overrideable in a test with {{USE_HARDWARE}}.
- **Enable mouse movement**: Turn on replay of user mouse gestures (mouse clicks and context menu actuations and mouse movements), if they were recorded. The recording options are configured on the recording tab. This option is only supported on the Internet Explorer browser.
- **Send Responses back to LISA**: By default, responses will not be sent back to LISA to improve performance and memory because it is usually not necessary. Override with {{SEND_RESPONSES}}.
- **Disable Java Events Override**: Use for Java GUIs that freeze on playback (load testing will be disabled). Override with {{

NO_AWT_OVERRIDE}}.

- **Max missing targets**: Generate a failure if more than this number of consecutive events fail with a warning.
- **Min Matching Score**: Specify how many differences are allowed between a recorded XPath value and the best match found during playback. Override in a test with {{MIN_BACTRACKING}}.
- **Failed assertions screenshot directory**: Specify the location for the screenshots captured at the moment assertions are triggered.

#### Timeout Settings

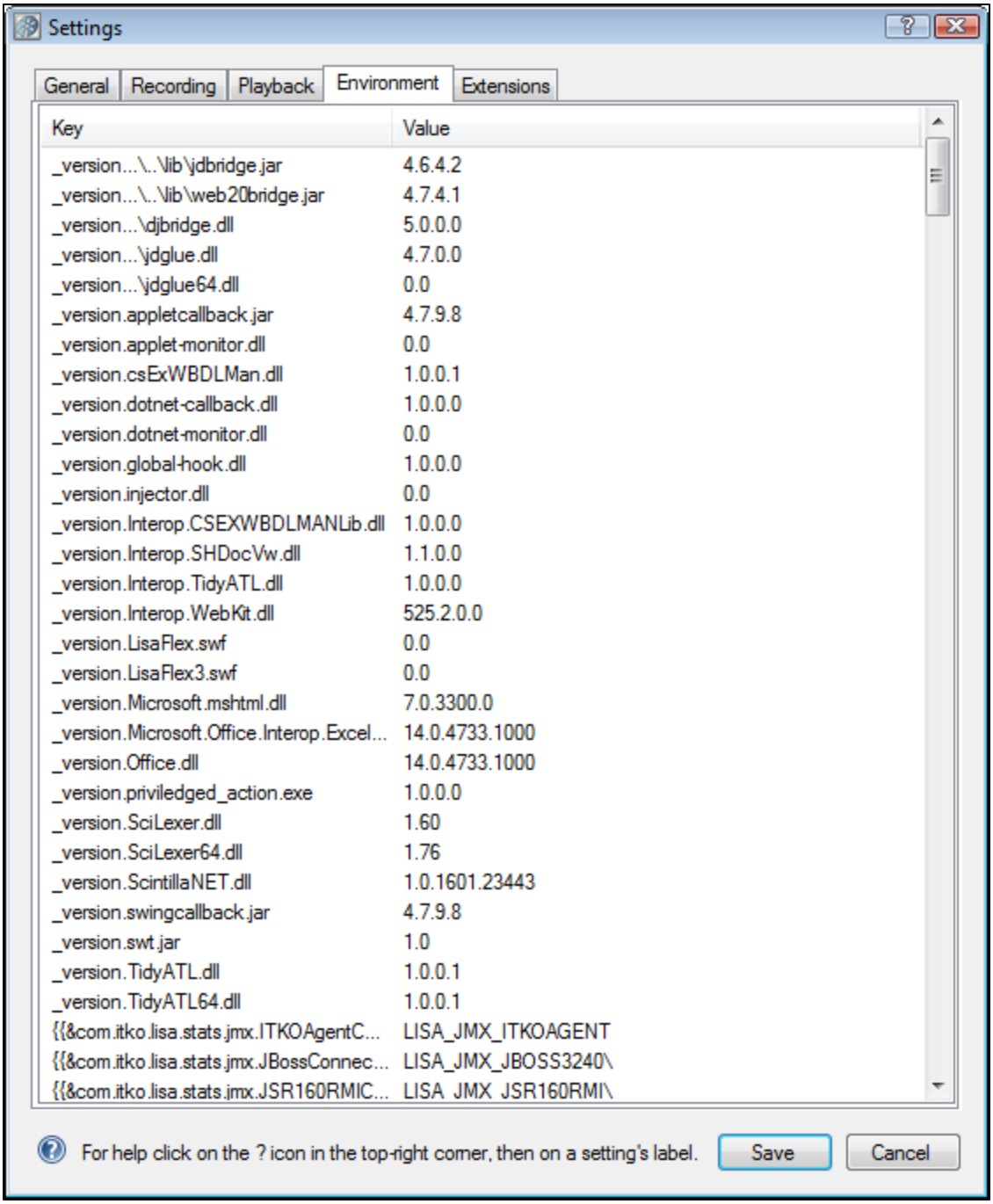
- **DOM Load Timeout**: Timeout for an HTML page load. Override with {{DOM_LOAD_TIMEOUT}}.
- **DOM Lookup Timeout**: Timeout for an HTML element lookup. Override with {{DOM_LOOKUP_TIMEOUT}}.
- **Applet Load Timeout**: Timeout for an applet load. Override with {{APPLET_LOAD_TIMEOUT}}.
- **Applet Lookup Timeout**: Timeout for Java component lookup. Override with {{APPLET_LOOKUP_TIMEOUT}}.
- **ActiveX Load Timeout**: Timeout for an ActiveX load. Override with {{ACTIVEX_LOAD_TIMEOUT}}.
- **ActiveX Lookup Timeout**: Timeout for an ActiveX component lookup. Override with {{ACTIVEX_LOOKUP_TIMEOUT}}.
- **Flex Load Timeout**: Timeout for Flex component lookup. Override with {{FLEX_LOAD_TIMEOUT}}.
- **Flex Lookup Timeout**: Timeout for a Flex component lookup. Override with {{FLEX_LOOKUP_TIMEOUT}}.
- **Playback Speed**: Default speed to use to replay test as a multiplier of the recorded speed. 0x is usually the best choice. Override in a test with {{PLAYBACK_SPEED}} (integer between and 10).
- **Wait multiplier**: The multiplier of the event replay speed. 1x is the "normal" (as captured) speed.

#### Browser Options

- **Use the following browsers by default**: Internet Explorer, Firefox, or both.
- **Installation directories**: The installation directories for the specified browsers.

### 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 environment settings that can be overridden from LISA in a test case or in the local.properties, the following are available:

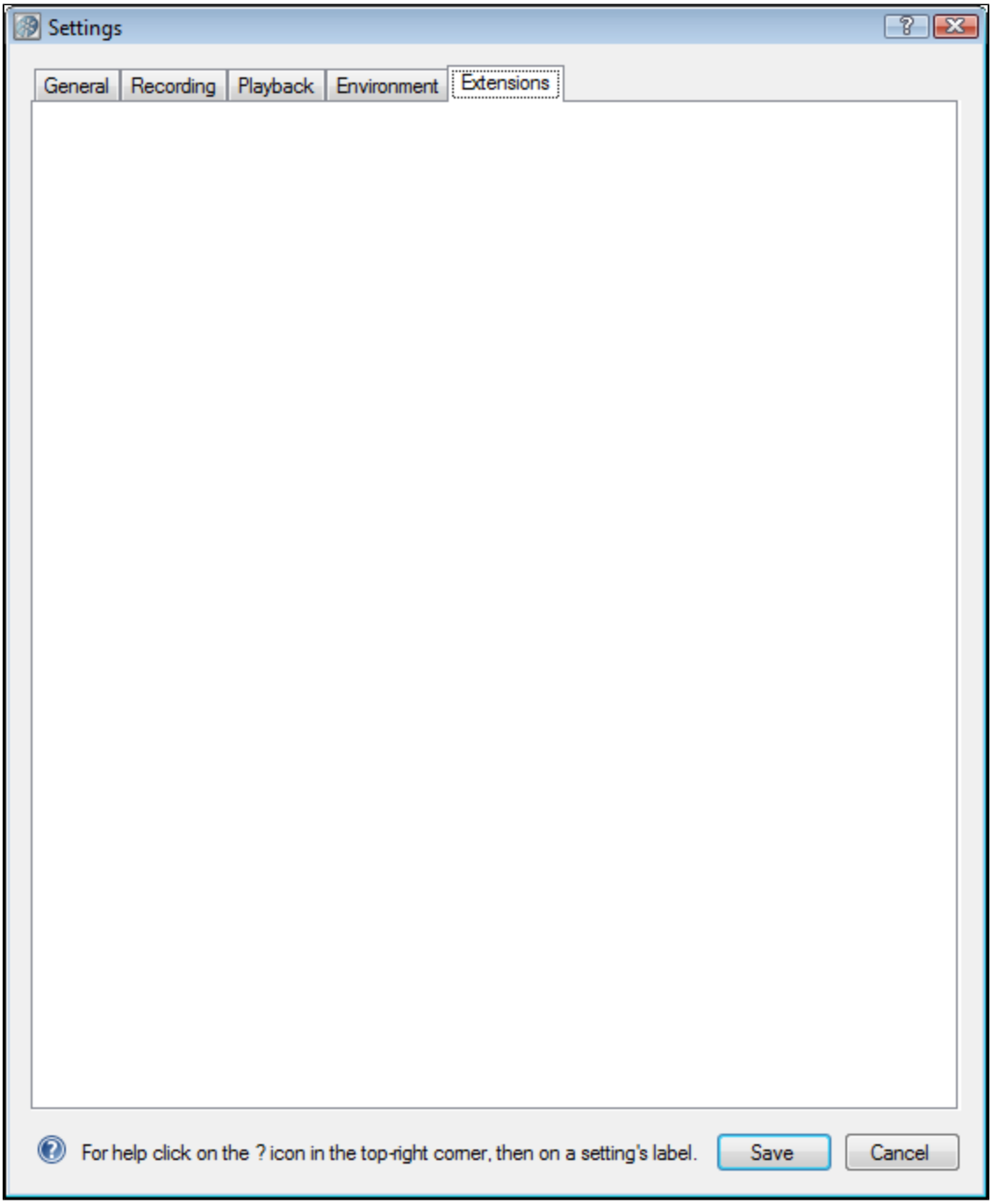
- **lisa.browser.launch.timeout**: Amount of time allowed for a browser to launch (default is 10,000). Specify in local.properties.
- **lisa.browser.exec.timeout**: Amount of time allowed for a step to execute (default is 300,000). Specify in local.properties.
- **lisa.browser.max.instances**: Maximum number of browser instances per computer (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 usually 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 get an encrypted password, run the command line: **lisa_browser.exe -m encrypt -in <clear text>**.
- **lisa.browser.share.subprocess.state**: Whether to run a subprocess using the same browser instance as parent test (default is false).

Specify in configuration of subprocess.

- **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.

### Extensions Tab

The Extensions Tab is not currently being used.




## Web 2.0 Browser and Extension Updates

Because the Web 2.0 environment evolves so fast, you can update it without waiting for a whole LISA release. These intermediate releases are considered **unsupported**, but because most customer environments are not available to the outside world, we cannot test a bug fix or feature enhancement against them so we work through this mechanism to provide a fast turnaround cycle. When the changes have been approved by a customer, they are submitted to the official build environment for full testing.

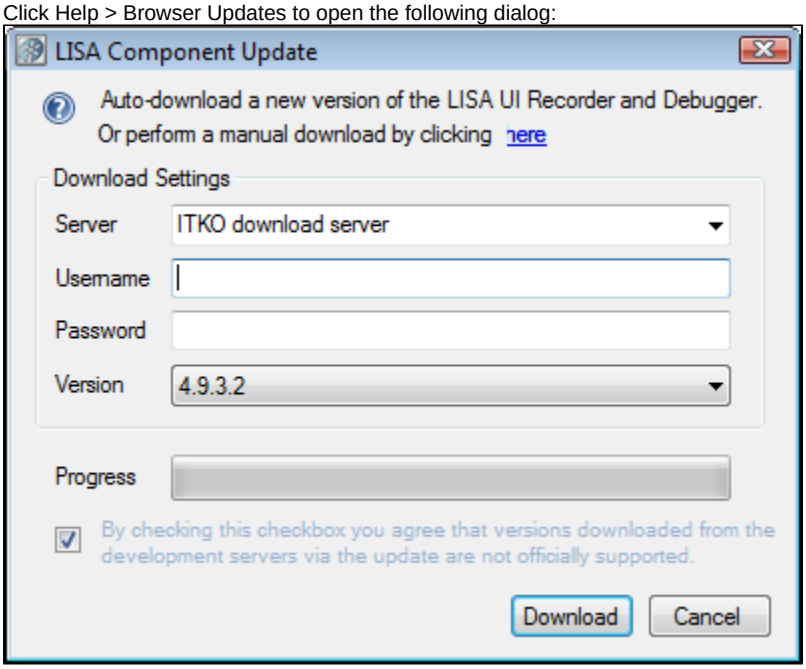
### *Browser Updates*

One way to get the update is to run the *update* command from the command line:

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

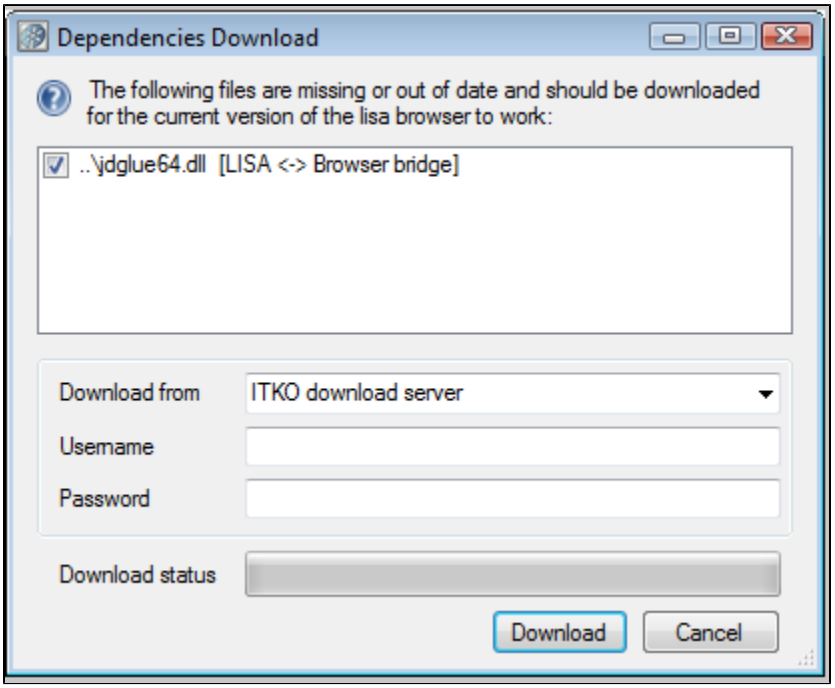
To download the updates of the Web 2.0 browser, open the Web 2.0 browser within LISA.

1. Click Help > Browser Updates to open the following dialog:




2. Enter your LISA license information and click Download to start the download. You can monitor the download progress though the progress bar.
3. After the download is completed you will be notified that the update will be effective on restart.

When major new functionality is added, some dependencies will be added or modified that will not be available until you install from a full LISA installer. This is why the browser checks for those on startup and prompts you for a download on startup if they are missing.

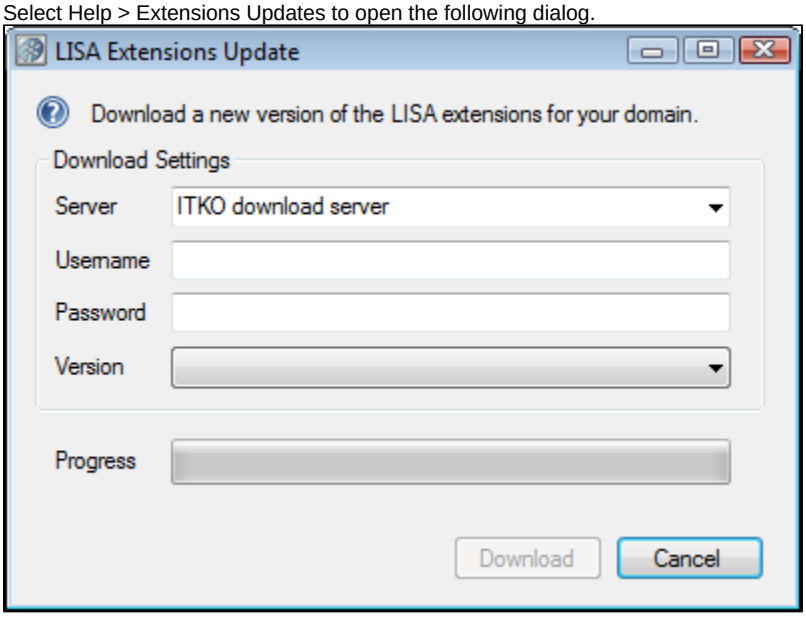



The browser will not start until those missing dependencies have been downloaded from one of the available locations: one of the release servers or one of the development servers. If none of these are accessible from the computer (no internet access, for example), you will need to download the listed files 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.

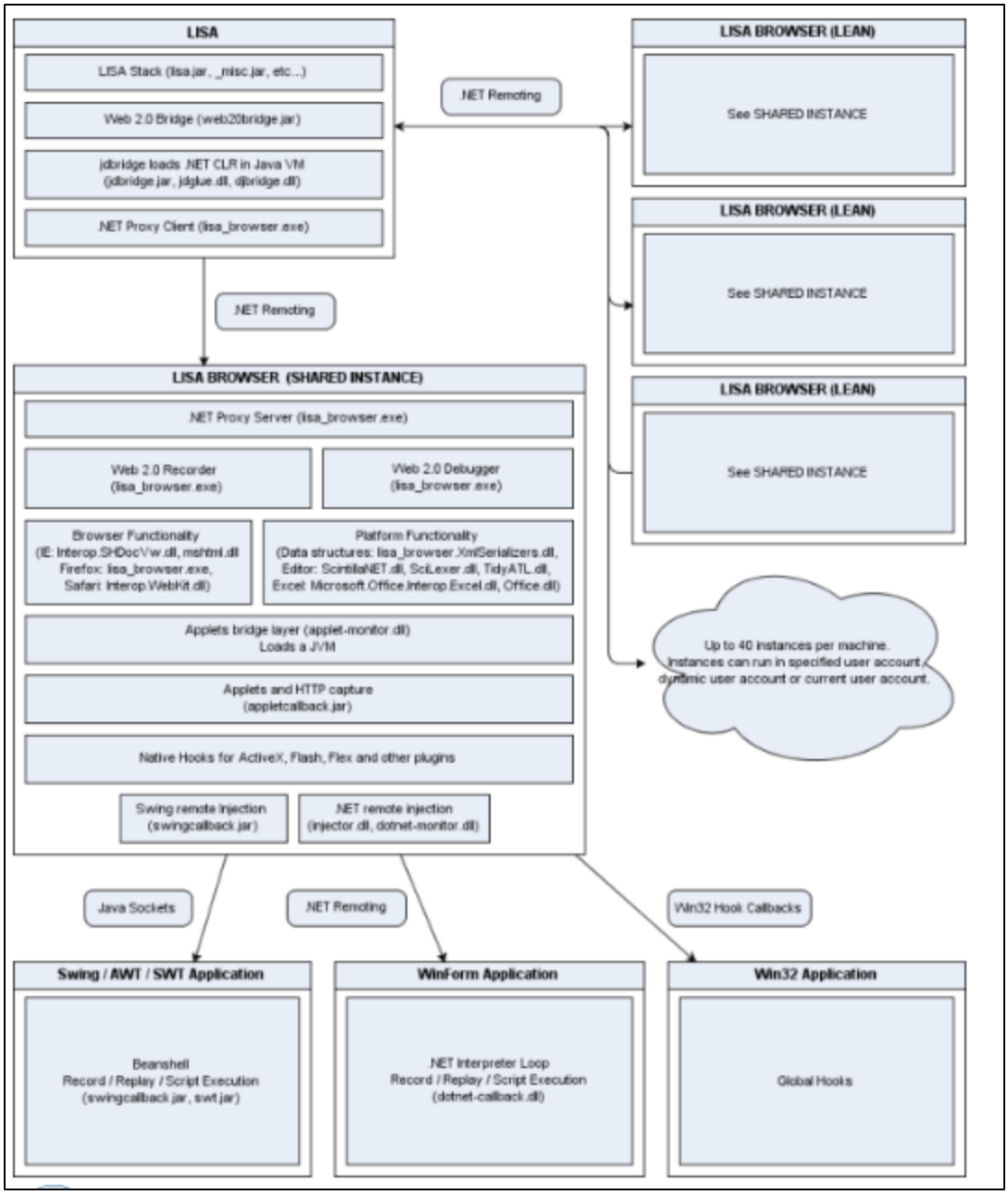
1. Select Help > Extensions Updates to open the following dialog.




2. Click Download to start the download.

## Web 2.0 Browser Architecture

The Web 2.0 architecture in 4.6 and newer versions is as diagrammed.




***FAQs:***

**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, both open-source (JDIC, JRex, and so on) and commercial, that claim to do this but our experience is that they are 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. Because 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:

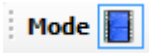
- In-process is not doable for the applets reason mentioned previously, and the required ability to run browsers under different user accounts.
- A proprietary protocol over raw sockets was too much work, especially because the communication must be bidirectional.
- Web Services over HTTP is the approach used in LISA 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 previously, which left us with the second one.

# Web 2.0 Recording Mode

Within the Web 2.0 browser, you can record many types of applications, like a simple web application, Java application, Swing application, .net application, and others. 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.

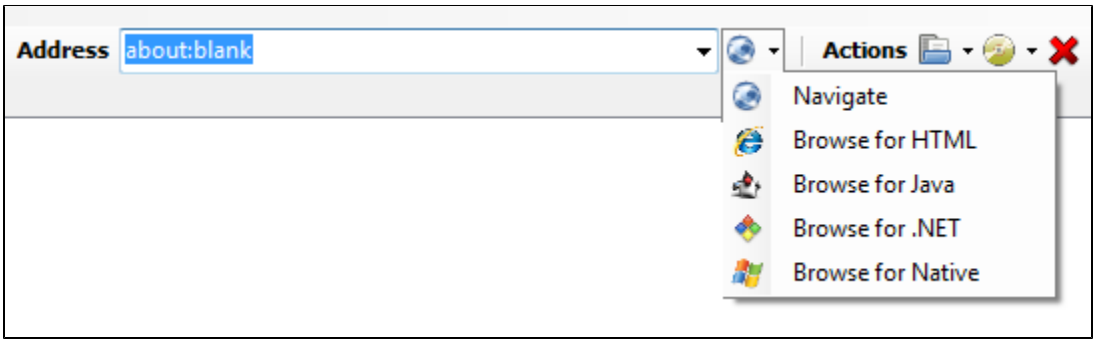

The Mode button on the LISA Browser toolbar indicates the mode of operation within the browser:

- **Recording Mode**: Where you can record the operation
- **Edit Mode**: Where you can edit the transactions
- **Playback Mode**: Where you can play back the recorded operation

You record the web page in the **Recording** mode and play back the recording by selecting 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.

Available selections are:



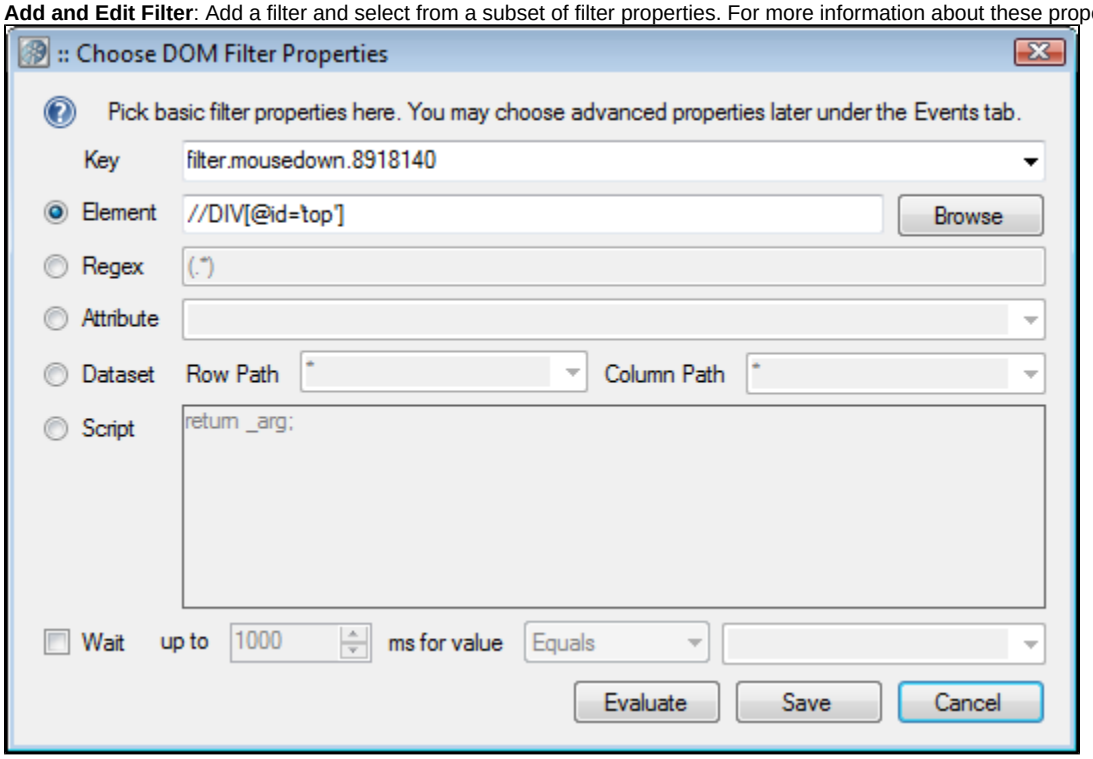

Enter the website address in the Address bar.

Navigate to the website for testing and in the background LISA will record your activity.

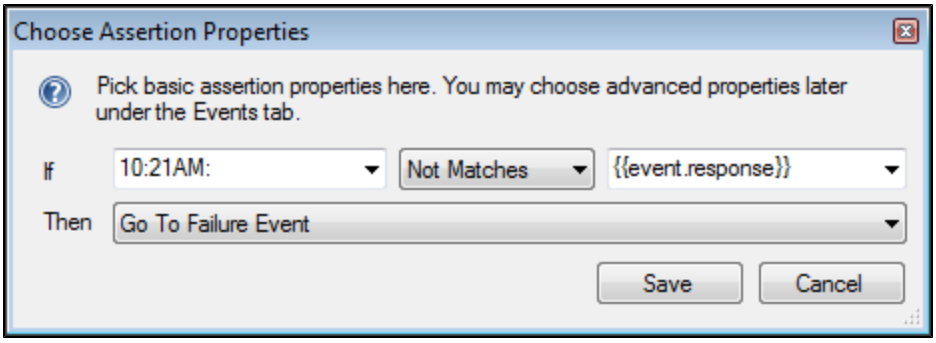
While recording, you can right-click to display a recording menu that lets you add and edit filters and assertions and to change debug mode settings.


Options are:

- **Add Quick Filter**: Select a field and select this option to auto-generate a filter, which can be seen in the Events tab.
- **Add and Edit Filter**: Add a filter and select from a subset of filter properties. For more information about these properties, see Filters.




- **Add Quick Assertion**: Select a field and select this option to auto-generate an assertion, which can be seen in the Events tab.
- **Add and Edit Assertion**: Add an assertion and select from a subset of assertion properties. For more information about these properties, see Assertions.




- **Debug**: Select to open the page in the DOM or HTML window, or to open the Debug window at the bottom of the screen.

When you are done, click Save to save the recording. This will save the recording and exit the web browser.

To play back 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:

Recording Example

Recording a Swing Test

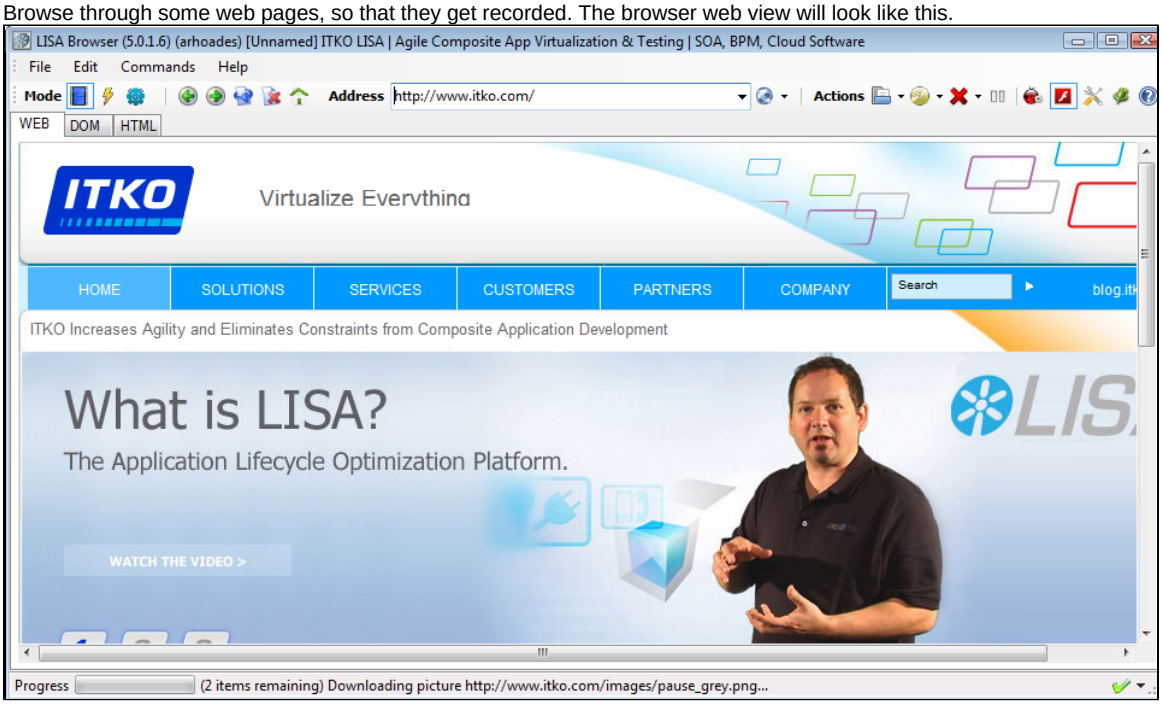
Recording an Applet Test

Different Views of a Web Page After Recording

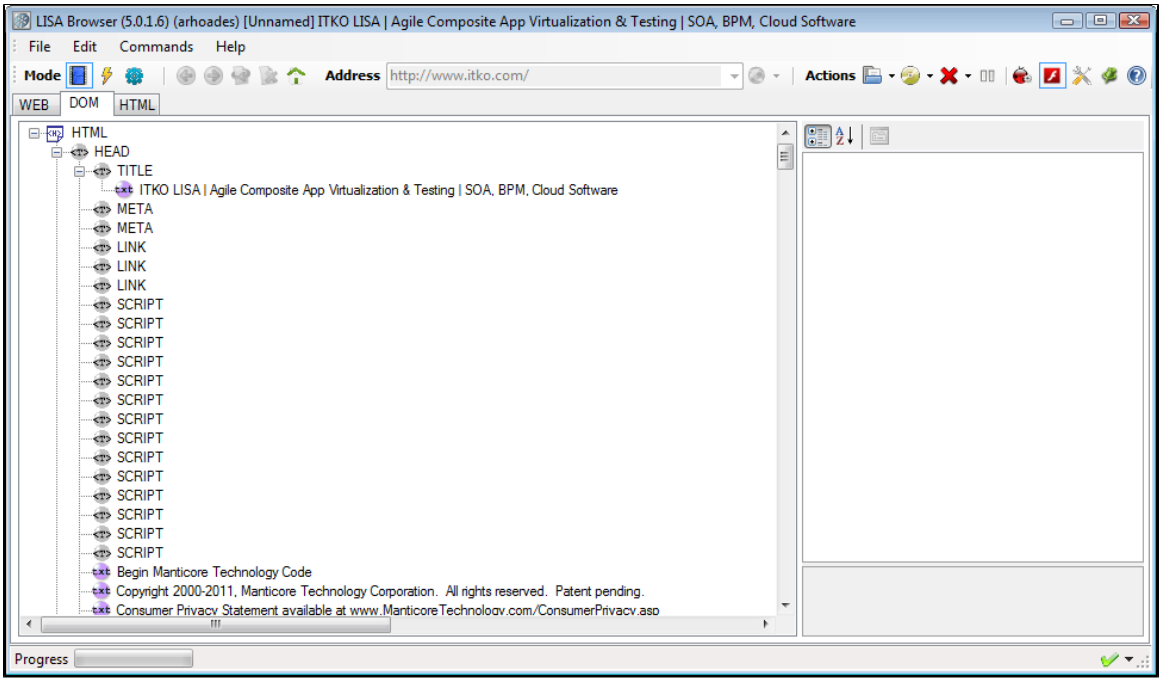
## Recording Example

**To start the recording**

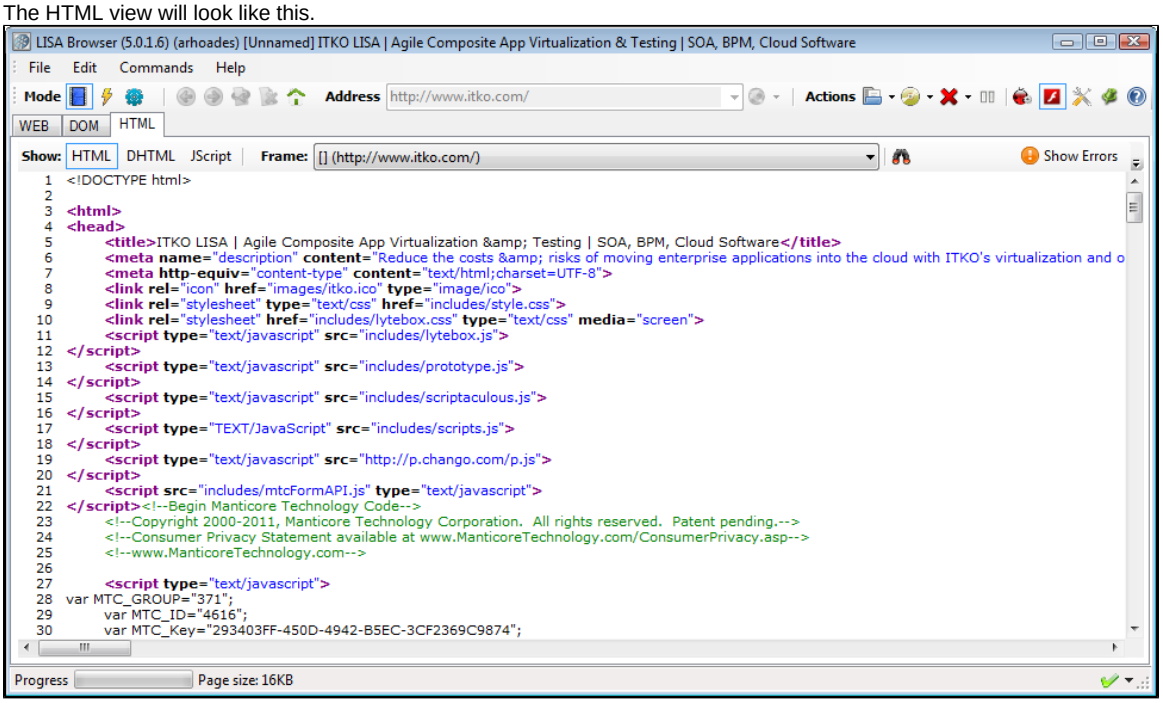
1. Type in a web address, for example: http://www.itko.com/.
2. Browse through some web pages, so that they get recorded. The browser web view will look like this.




The DOM view will look like this.

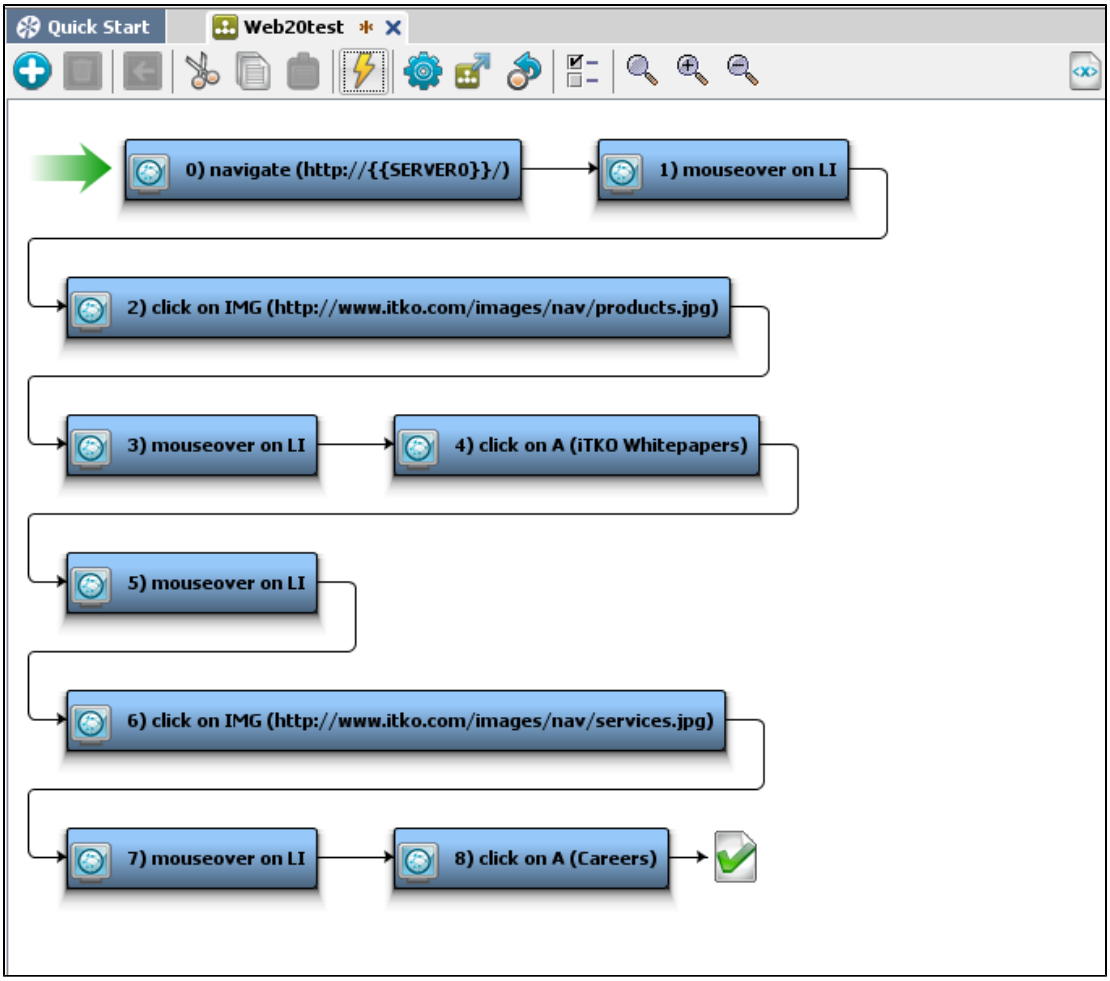



The HTML view will look like this.




3. After 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.




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

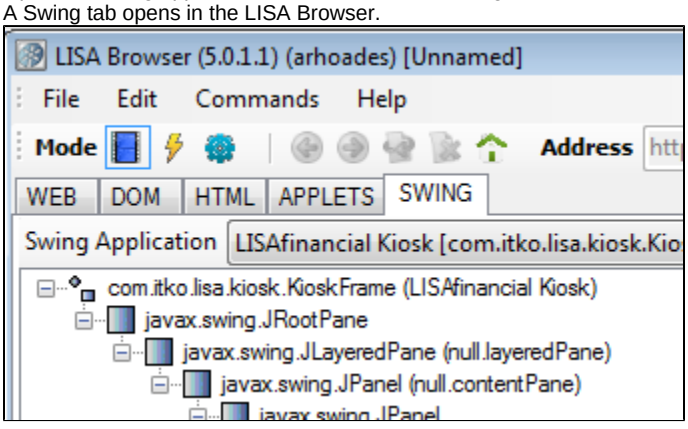
- Staging a test or
- Using Interactive Test Run (ITR)

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

## Recording a Swing Test

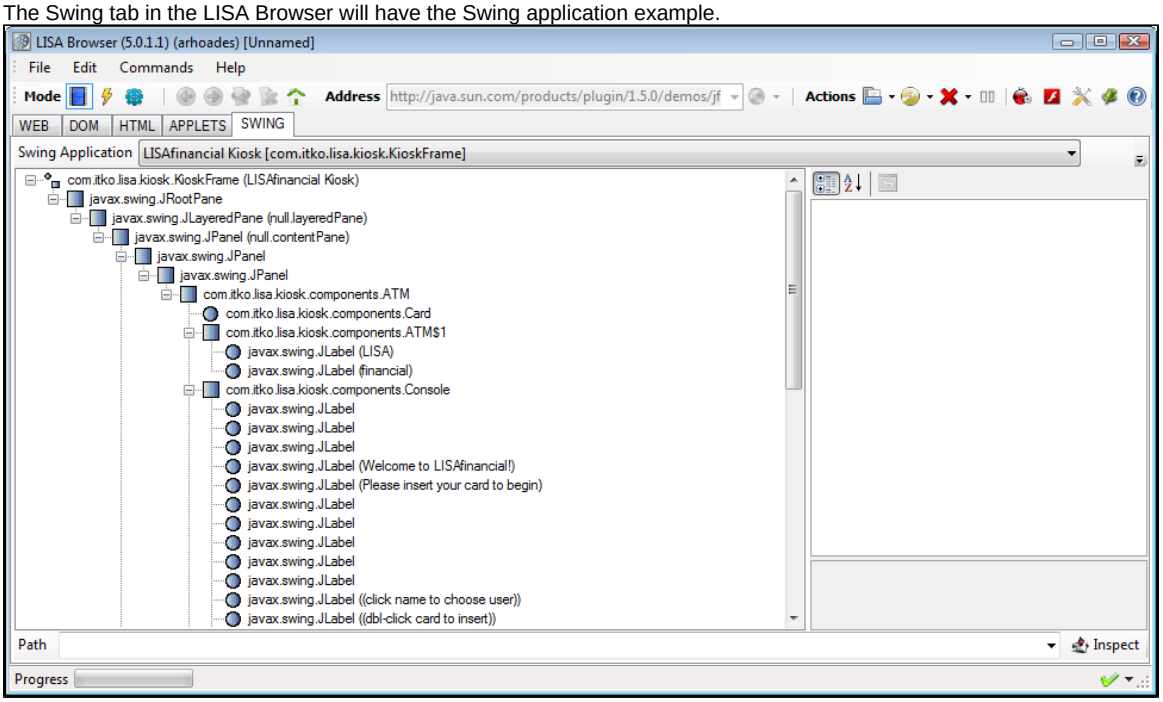
To record a Swing test, you must have a Swing application.

1. Open the LISA Browser and click **Record or Edit a Java test case by clicking here.**
2. Click **here** and browse to the path of the Swing application.
3. Open the Swing application and browse for recording.
4. A Swing tab opens in the LISA Browser.




Your Swing application will open in a separate window.

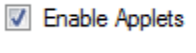

The Swing tab in the LISA Browser will have the Swing application example.



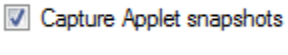
# Recording an Applet Test

To record an applet test, you must have a web page that uses an applet.

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

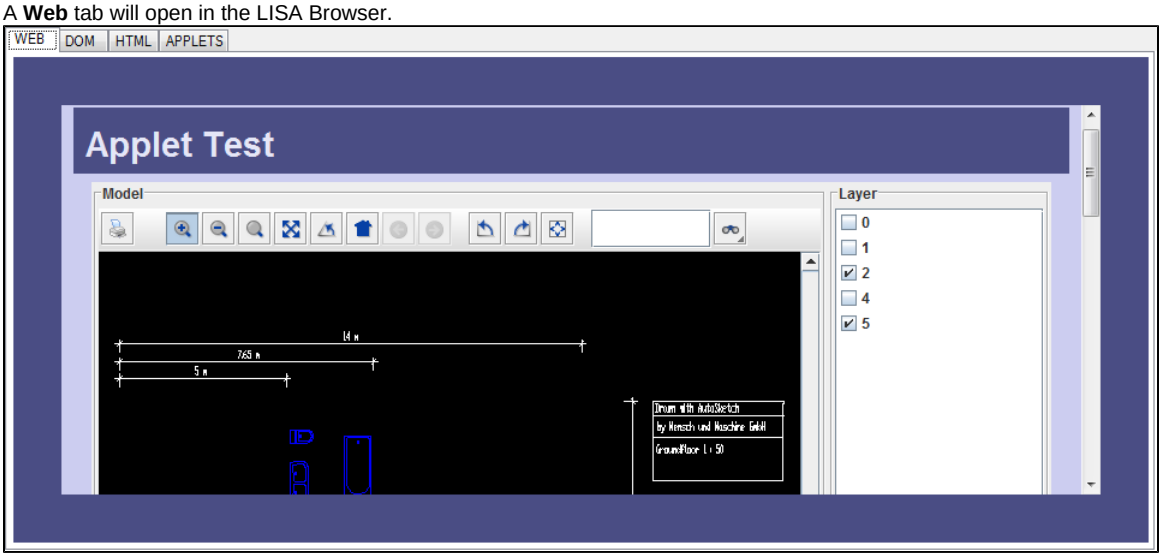



- In the General Settings tab

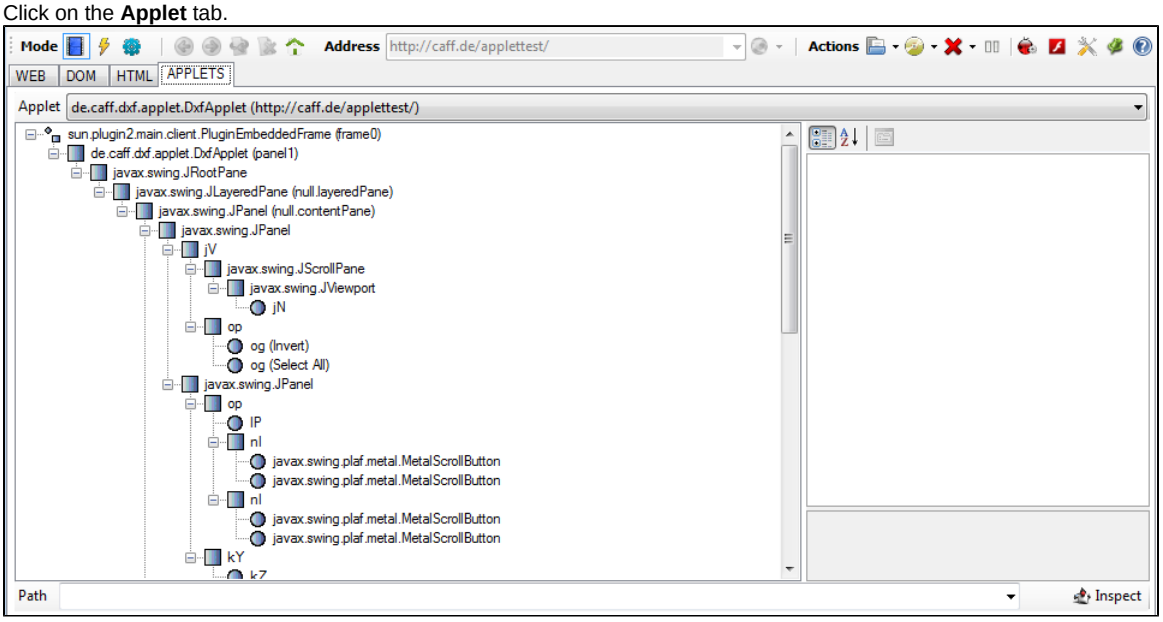



- In the Recordings Settings tab

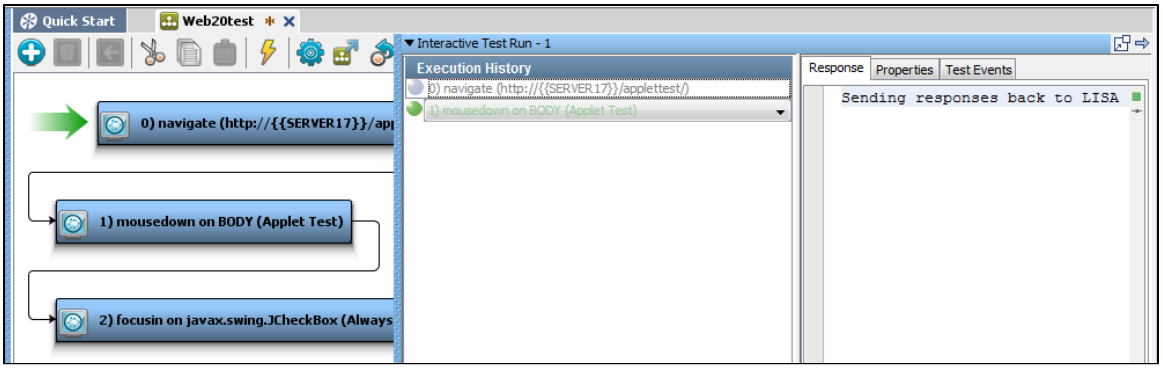
1. Open the LISA Browser and click on **Record or Edit a Java test case by clicking here**.
2. Click ***here*** and browse to the path of a web page that uses a Java applet. Example: http://caff.de/applettest/
3. Open the web page and start recording.
4. A **Web** tab will open in the LISA Browser.



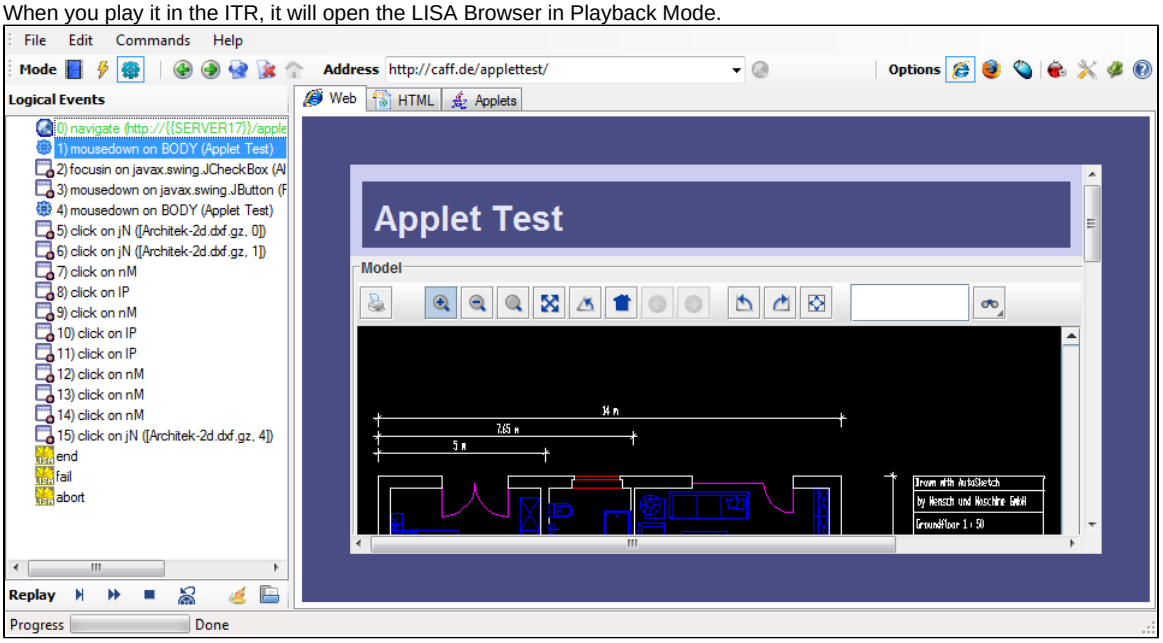

5. Click on the **Applet** tab.




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




8. When you play it in the ITR, it will open the LISA Browser in Playback Mode.

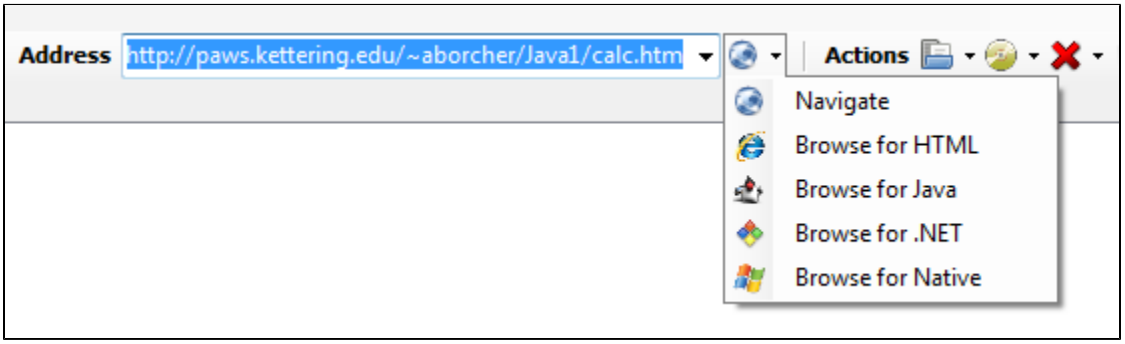



## Different Views of a Web Page

When you record a test, it is sometimes good 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, for example.

To facilitate this, the LISA Browser is capable of showing 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.




### Web View

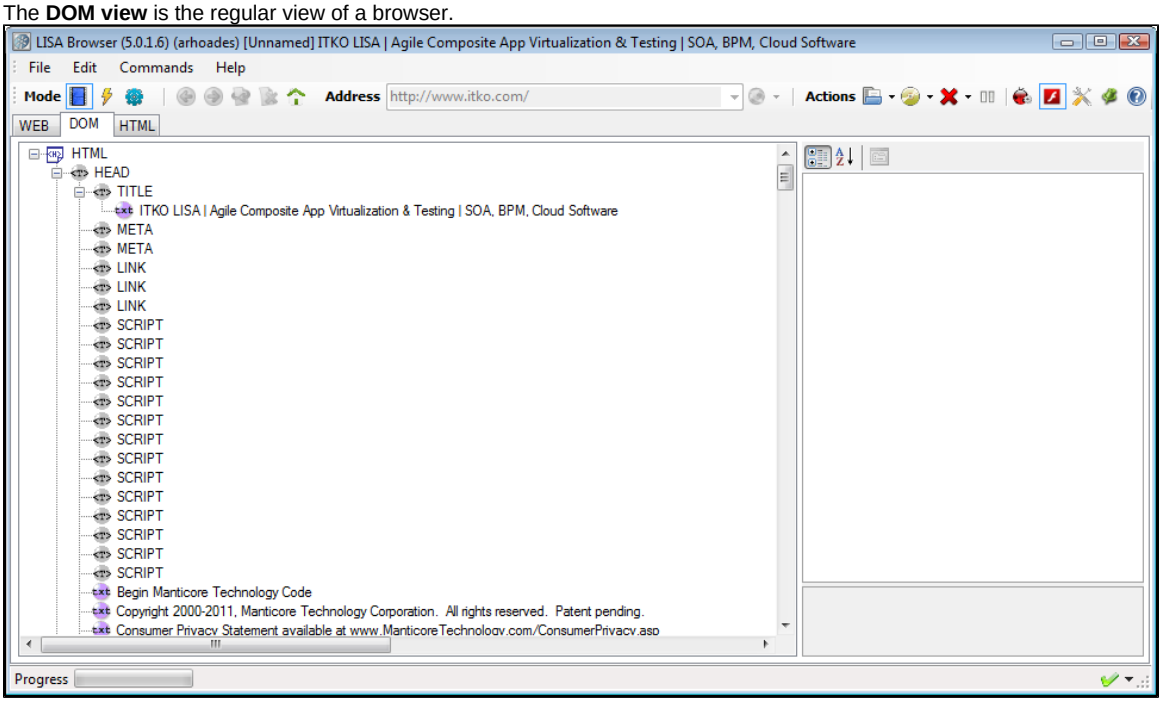
This web view is simply the representation of what is seen on the screen.


### 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 grid.

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.




### 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.

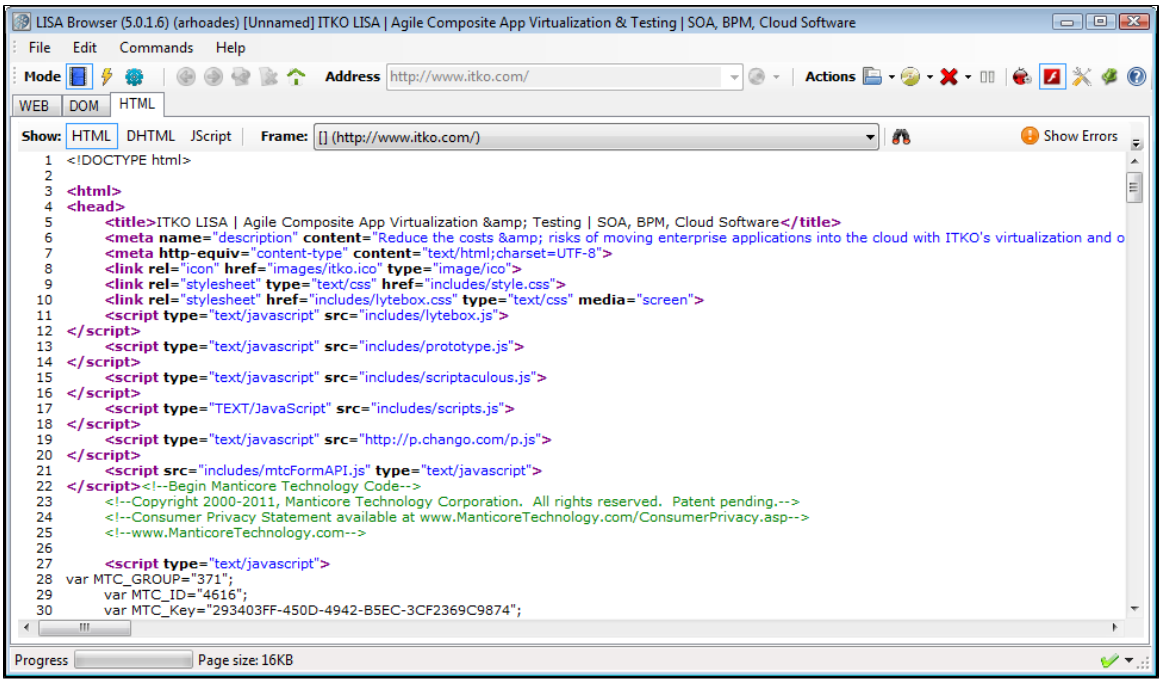
The **Frame** combo box lets you select which frame to display the source of in case of multiple frames, and the **HTML / DHTML / JScript** buttons let you select whether you are seeing the HTML that comes from the server (static HTML), or the HTML at it is currently being seen by the client

(Dynamic HTML), or the JavaScript files and snippets used by the current page and frame.

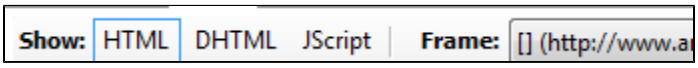
This is especially useful when pages use JavaScript or Ajax that dynamically modify the HTML in the browser without reloading a whole page from the server.

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 following page shows the HTML view of the previous web page.




In HTML view, there are some controls at the top of the window.



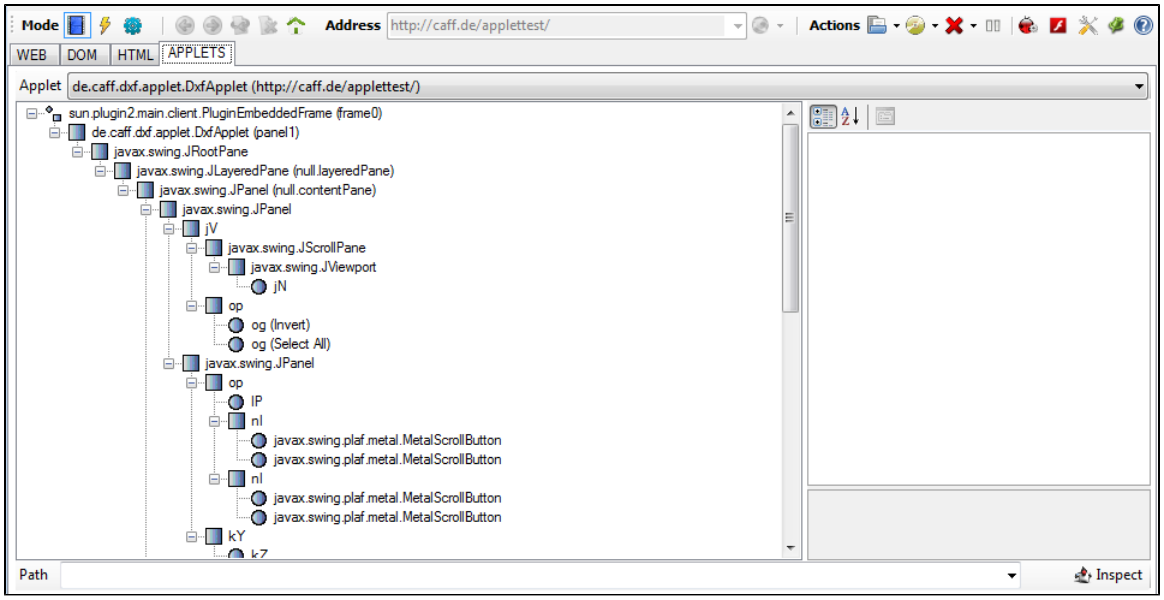

- **Show HTML/DHTML/Jscript**: You can select the required script from HTML/DHTML/Jscript. The Static/Dynamic HTML group lets you select whether you are seeing the HTML that comes from the server, or the HTML as it is currently being seen by the client. This is especially useful when pages make use of 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, if there are multiple frames.
- **Find:** Opens a Search dialog and lets you 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.

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 (for example, **java.awt.Component** properties, **javax.swing.JComponent** properties, and so on.)

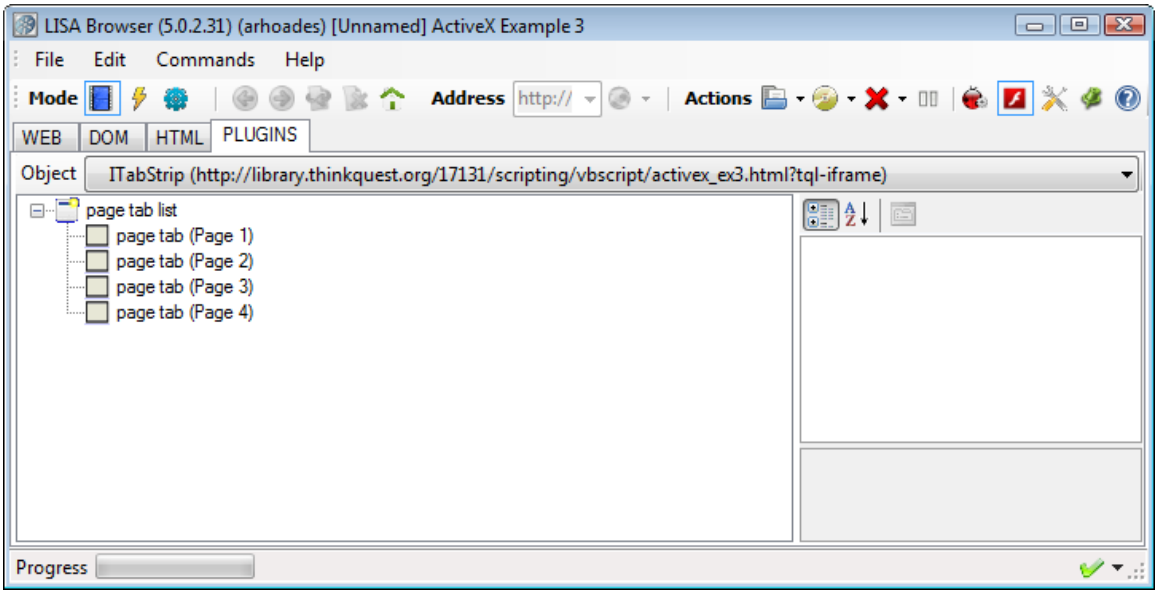
At the top of the panel, there is a drop-down box that lets you select which applet to display the properties of, in case there are several on the page.




### 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 lets you 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.



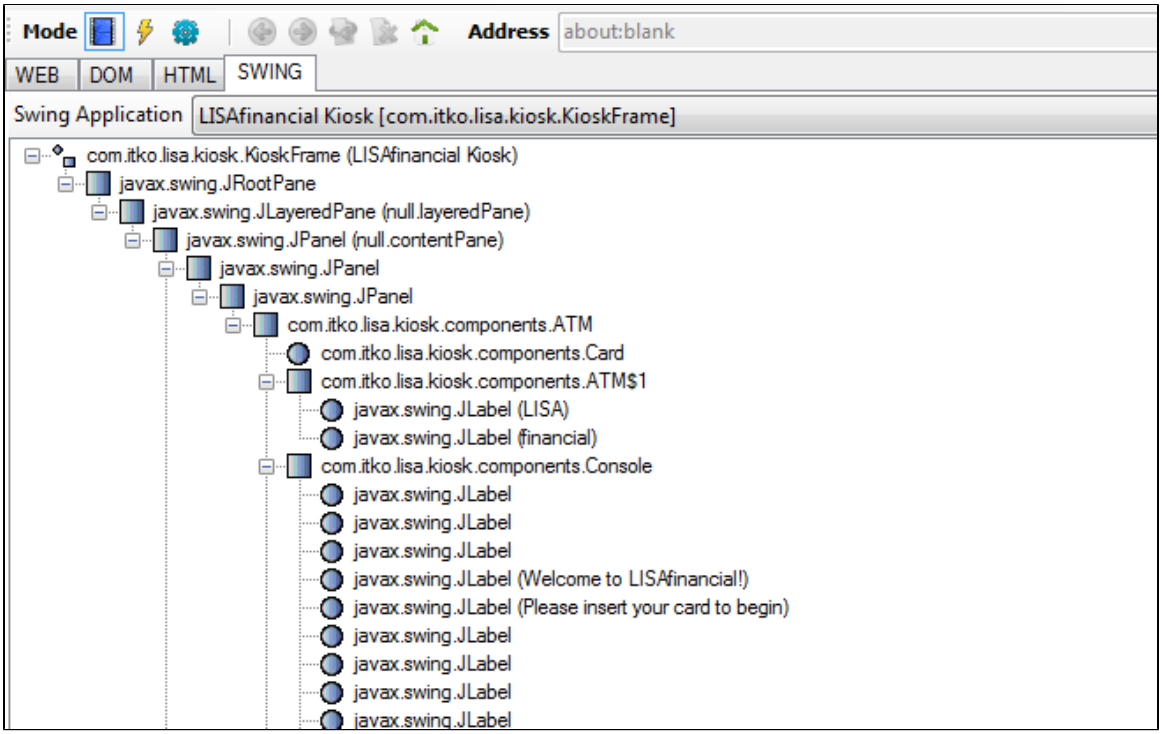

### Swing View

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.

NEED UPDATED SCREEN CAPTURE




**.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.

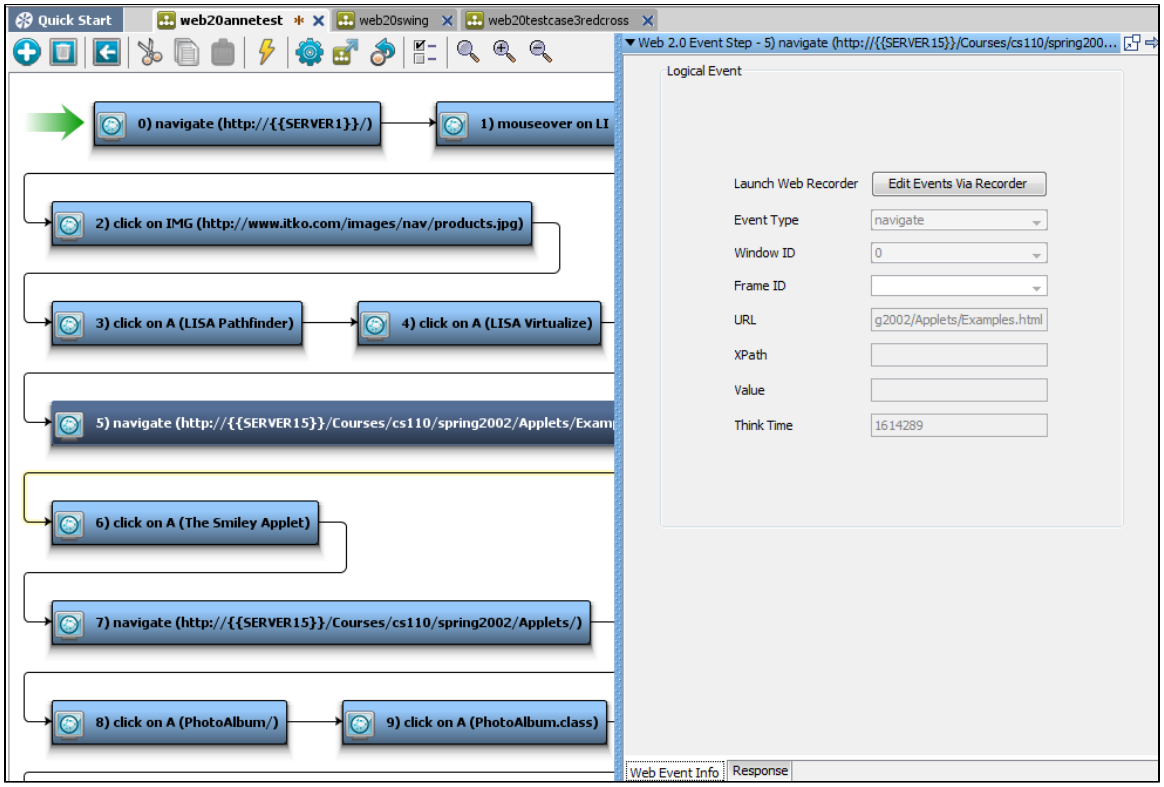
## After Recording

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

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

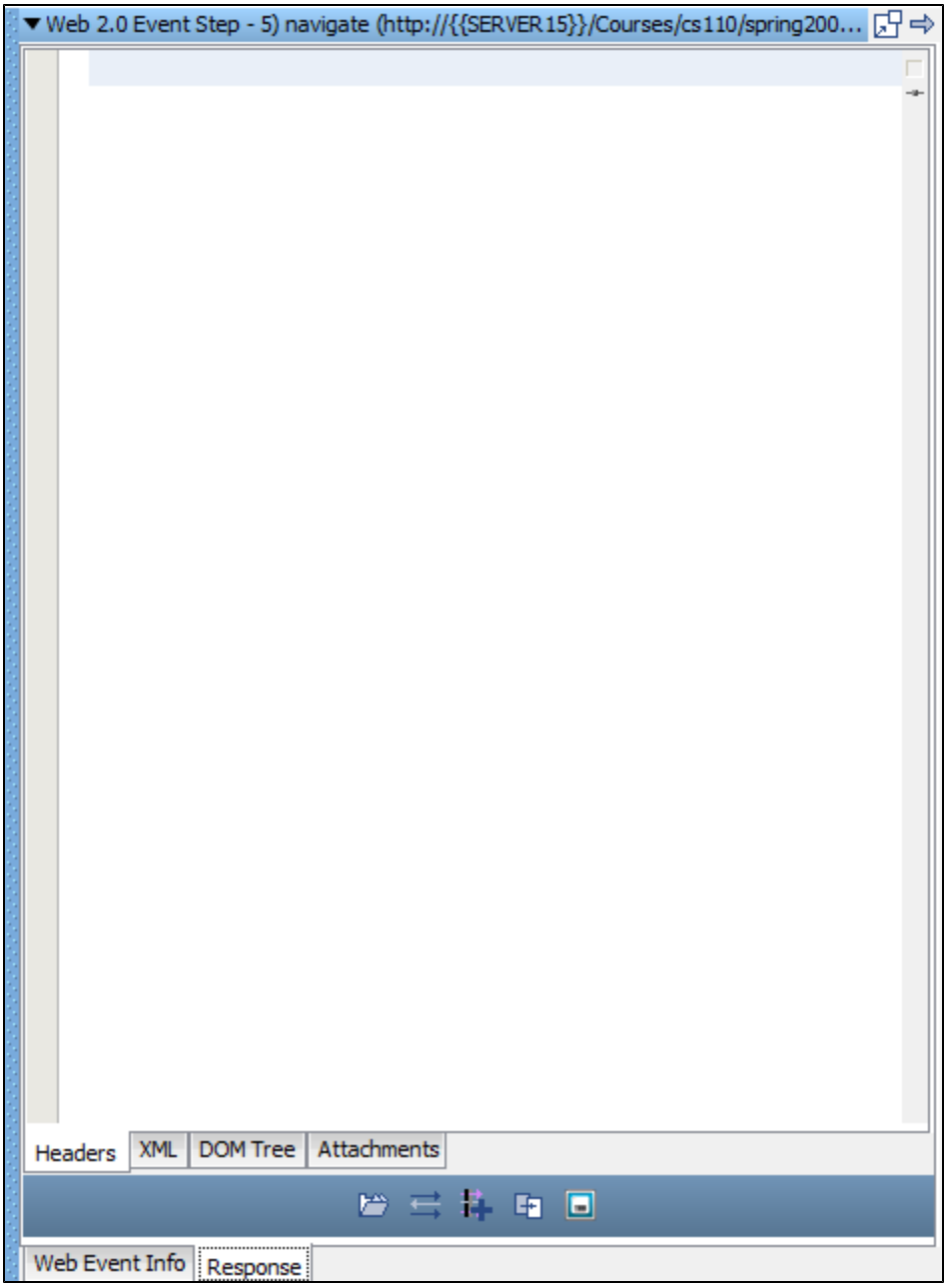
You can edit any of the test steps by double-clicking on the test step in LISA Workstation.

Select a step to open its editor in the right panel.



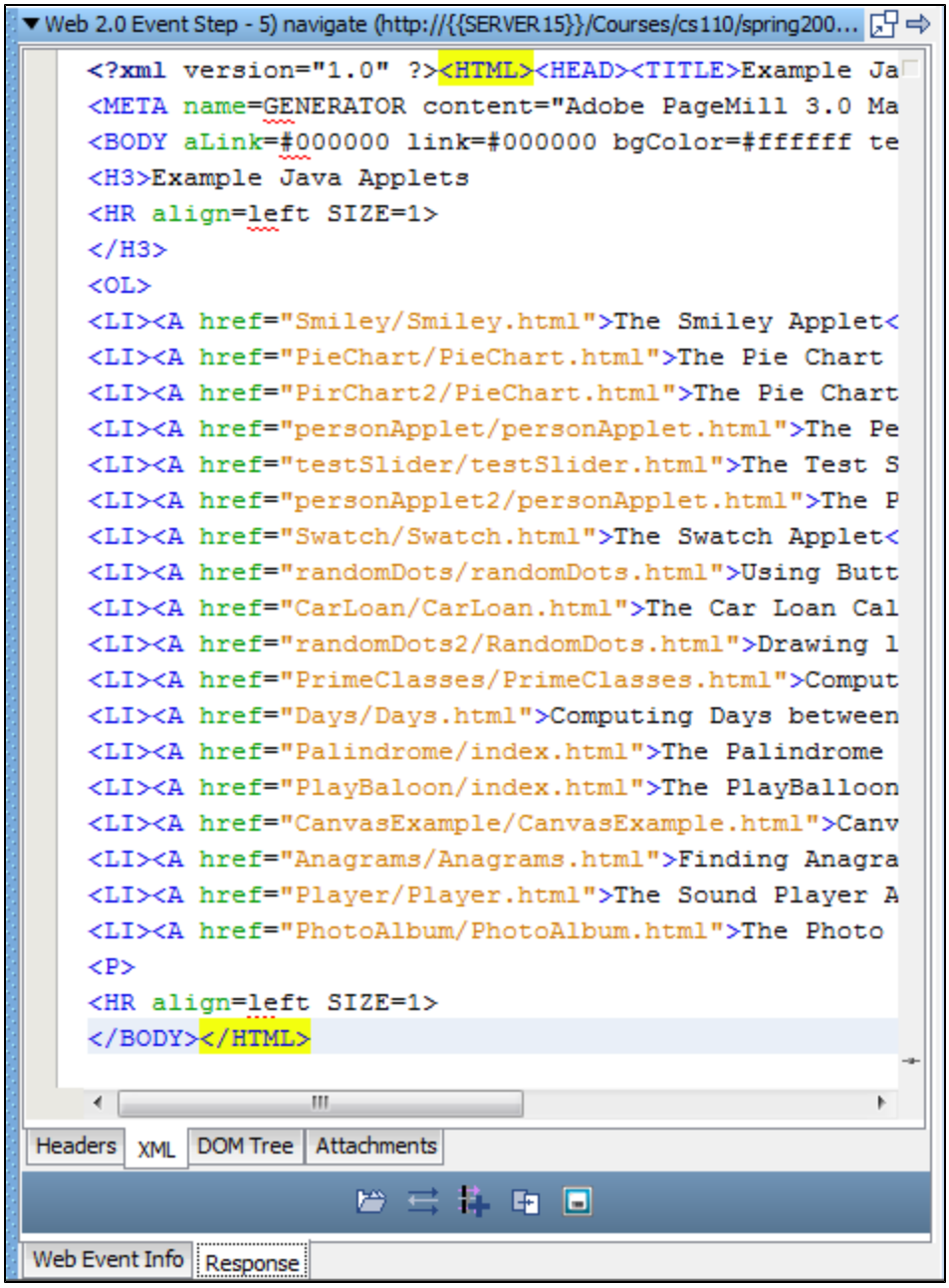

There are two tabs at the bottom in the right panel.

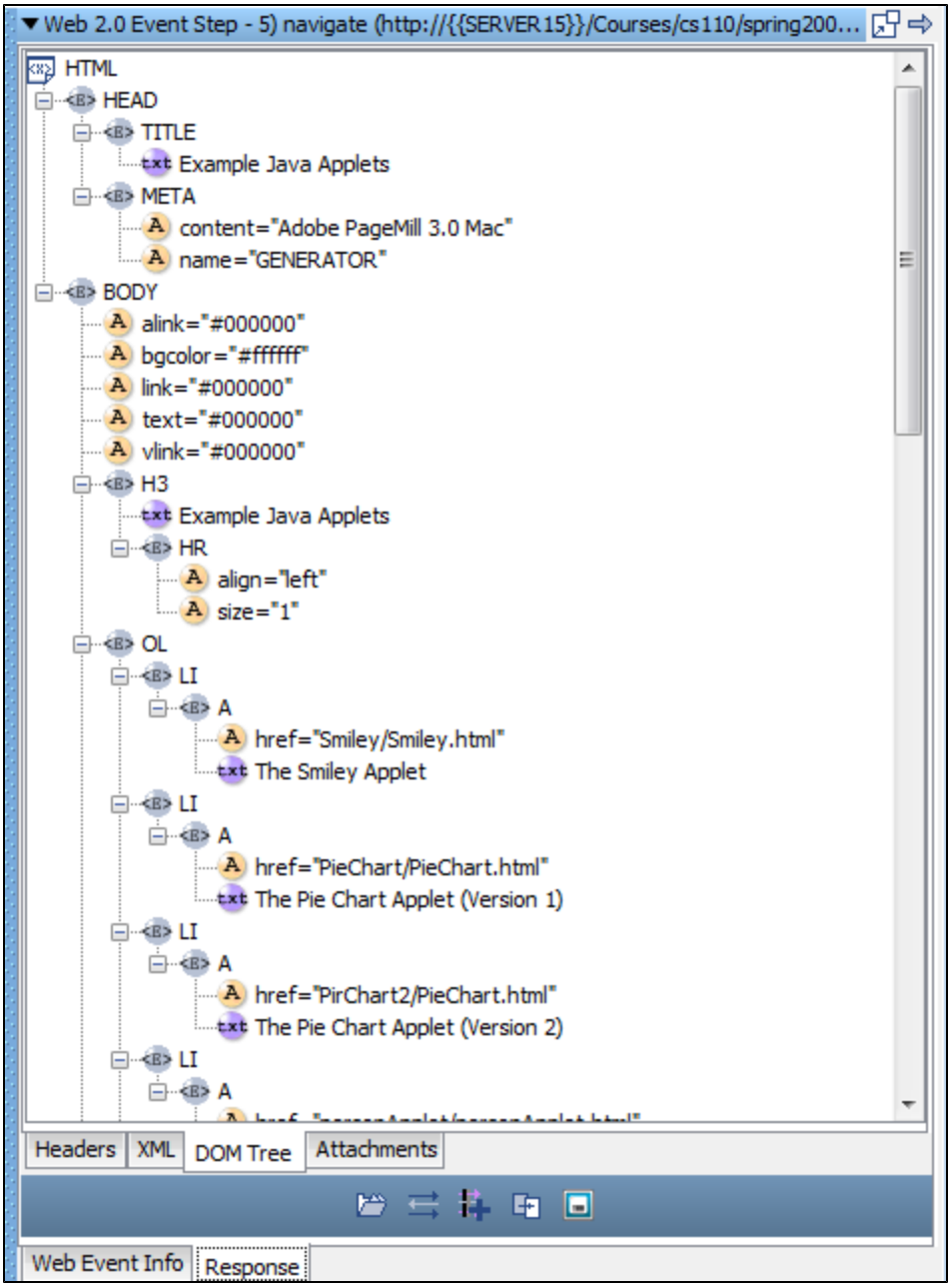
- **Web Event Info tab**: This tab opens by default, and displays information about the event or test step. Click the "Edit Events Via Recorder" button to open the LISA browser for editing.
- **Response tab**: Click the **Response** tab to see the response.




The first tab to appear is the **Headers** tab, where you can add filters and assertions.

Here, you can view the XML, DOM Tree, and Attachments tabs to see the respective outputs.

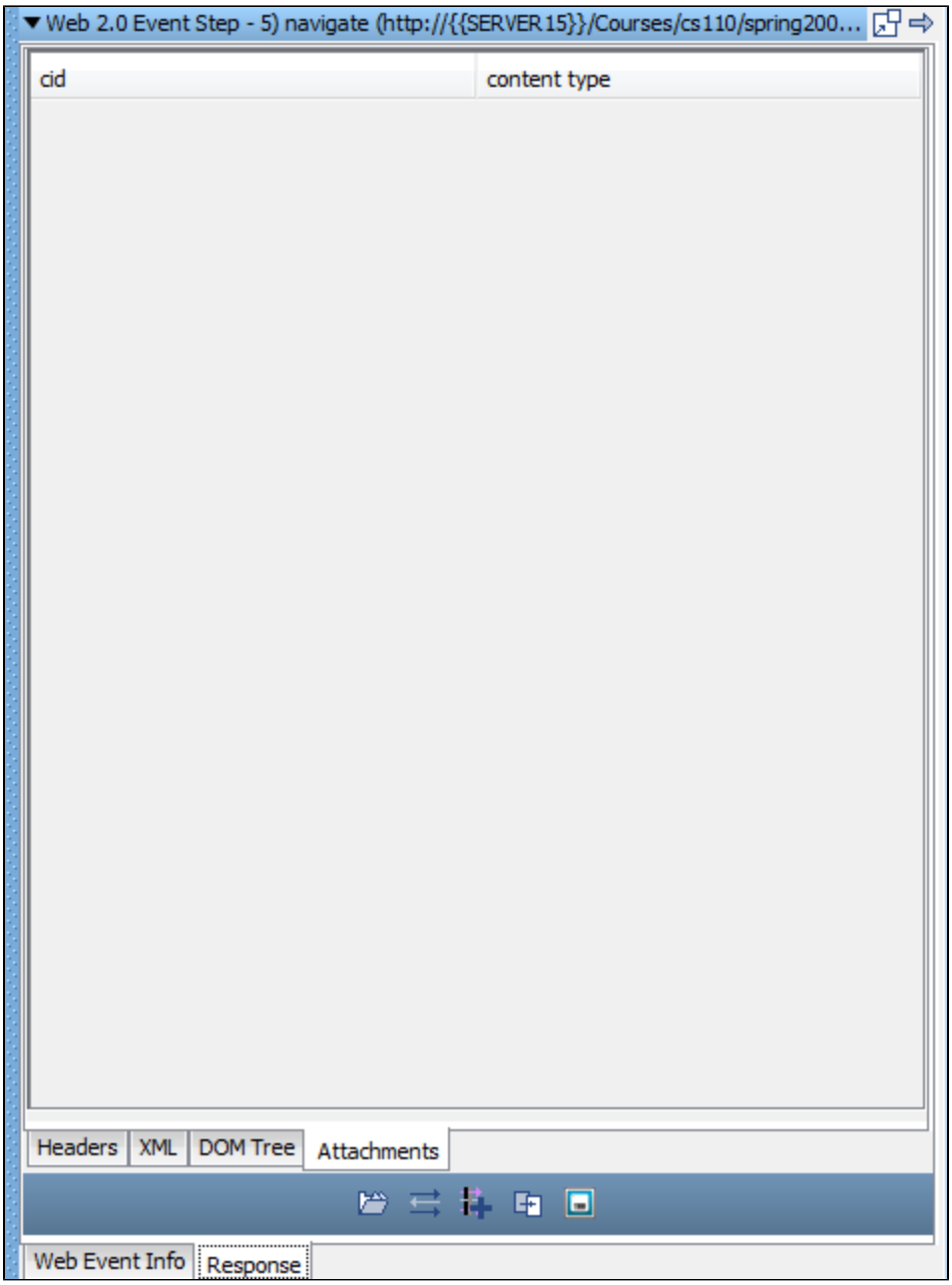

Web 2.0 Event Step - 5) navigate (http://{{SERVER15}}/Courses/cs110/spring200...
<?xml version="1.0" ?><HTML><HEAD><TITLE>Example Ja
<META name=GENERATOR content="Adobe PageMill 3.0 Ma
<BODY aLink=#000000 link=#000000 bgColor=#ffffff te
<H3>Example Java Applets
<HR align=left SIZE=1>
</H3>
<OL>
<LI><A href="Smiley/Smiley.html">The Smiley Applet<
<LI><A href="PieChart/PieChart.html">The Pie Chart
<LI><A href="PirChart2/PieChart.html">The Pie Chart
<LI><A href="personApplet/personApplet.html">The Pe
<LI><A href="testSlider/testSlider.html">The Test S
<LI><A href="personApplet2/personApplet.html">The P
<LI><A href="Swatch/Swatch.html">The Swatch Applet<
<LI><A href="randomDots/randomDots.html">Using Butt
<LI><A href="CarLoan/CarLoan.html">The Car Loan Cal
<LI><A href="randomDots2/RandomDots.html">Drawing l
<LI><A href="PrimeClasses/PrimeClasses.html">Comput
<LI><A href="Days/Days.html">Computing Days between
<LI><A href="Palindrome/index.html">The Palindrome
<LI><A href="PlayBaloon/index.html">The PlayBalloon
<LI><A href="CanvasExample/CanvasExample.html">Canv
<LI><A href="Anagrams/Anagrams.html">Finding Anagra
<LI><A href="Player/Player.html">The Sound Player A
<LI><A href="PhotoAlbum/PhotoAlbum.html">The Photo
<P>
<HR align=left SIZE=1>
</BODY></HTML>
Headers
XML
DOM Tree
Attachments
Web Event Info
Response




Within the DOM tree view, you can select an attribute and select a command to be applied to it from the toolbar at the bottom of the panel.

| Button | Function |
|---|---|
|  | Load an XML document |
|  | Generate an HTML/XML filter for attribute or text |
|  | Generate an XML/XPath filter |
|  | Generate an assertion |

| | |
|---|---|
|  | Save result |




## Web 2.0 Playback Mode

To execute or debug a Web 2.0 test, you save it to LISA and execute it through staging or in the ITR (Interactive Test Run).

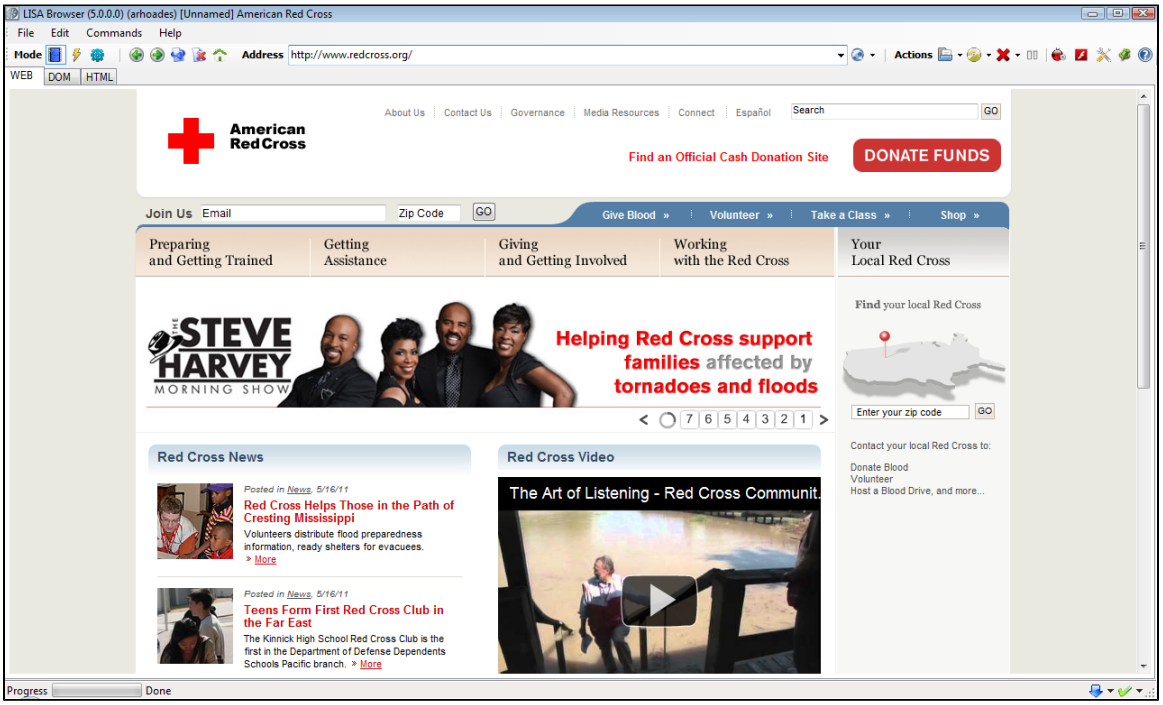
However, the Web 2.0 browser also provides some facilities to do some quick runs and debugging of your tests. It is primarily designed to work with Web 2.0 specific tests but has some powerful debugging capabilities.

Typically you would use this immediately after a recording session, doing a playback to make sure everything goes as you expect. After you have saved the recording, you can **Replay** it using the Playback mode.

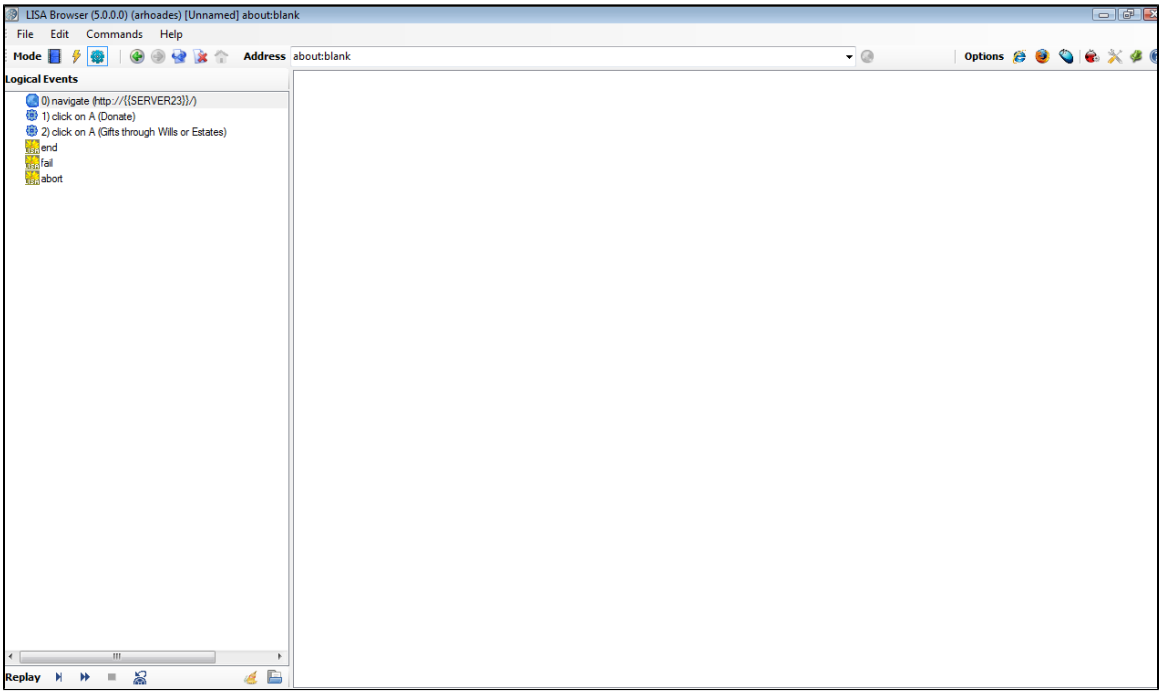
Click the Playback  icon on the toolbar to view the details of the last recording.

During playback, the browser toolbar is disabled.

This is the recording mode of a web page.




The playback of the recording is enabled by clicking on the playback mode button  on the toolbar.




## Options Toolbar

The Options toolbar 

provides options for playback.

The Internet Explorer  and Mozilla Firefox  buttons control which browser is being used to execute the web test. You can select one or both 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 (because it is automatically installed on Windows).

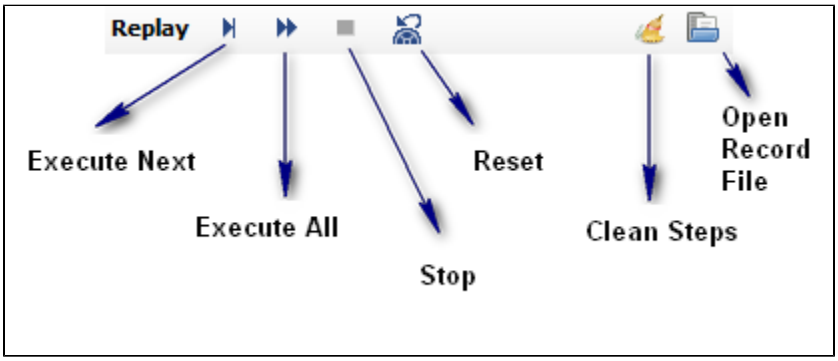
You can use the Move mouse button  to turn on or off mouse movement during playback.

## Playback Toolbar

The playback mode has a toolbar 

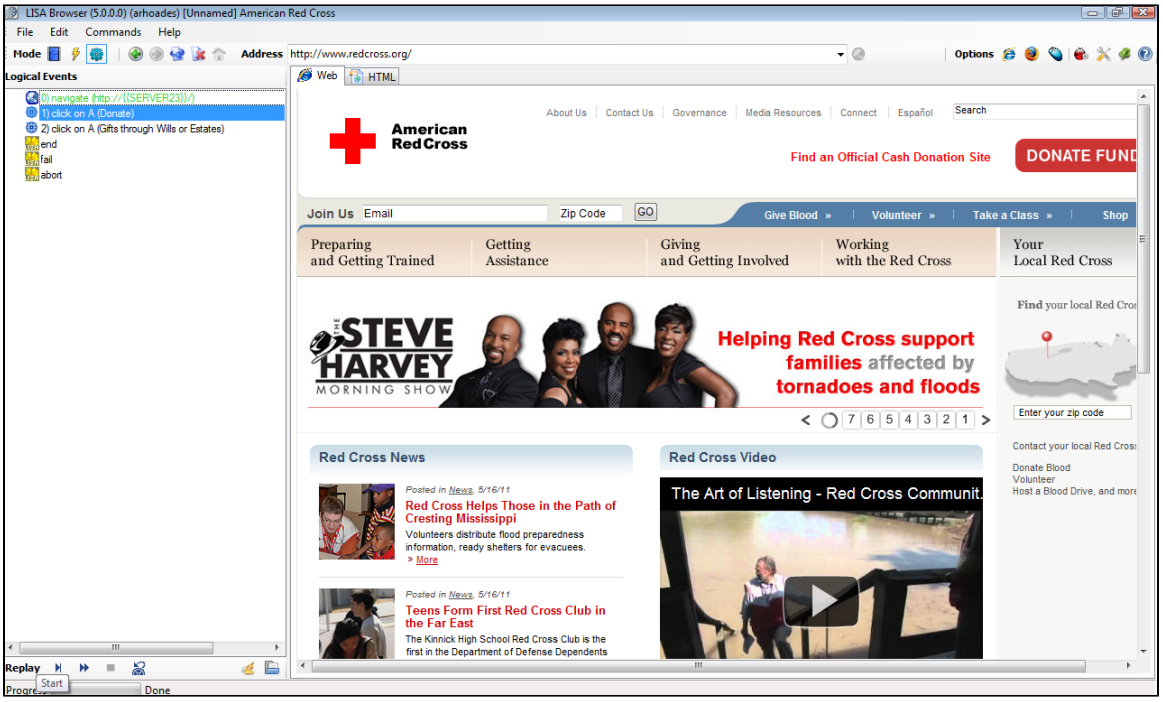
at the bottom of the Logical events pane, to control the movement of the playback activities.

Click Execute Next or Execute all steps to run the playback.




- **Execute Next**: Executes the selected event in the list.
- **Execute All**: Executes all the events starting at the selected one.
- **Stop**: Stops the playback.
- **Reset**: Resets all the variables and executes all the events in the list starting with the first one.
- **Clean Steps**: Disables all events that generated a warning in the last run.
- **Open Record File**: Opens a previously-recorded file.

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

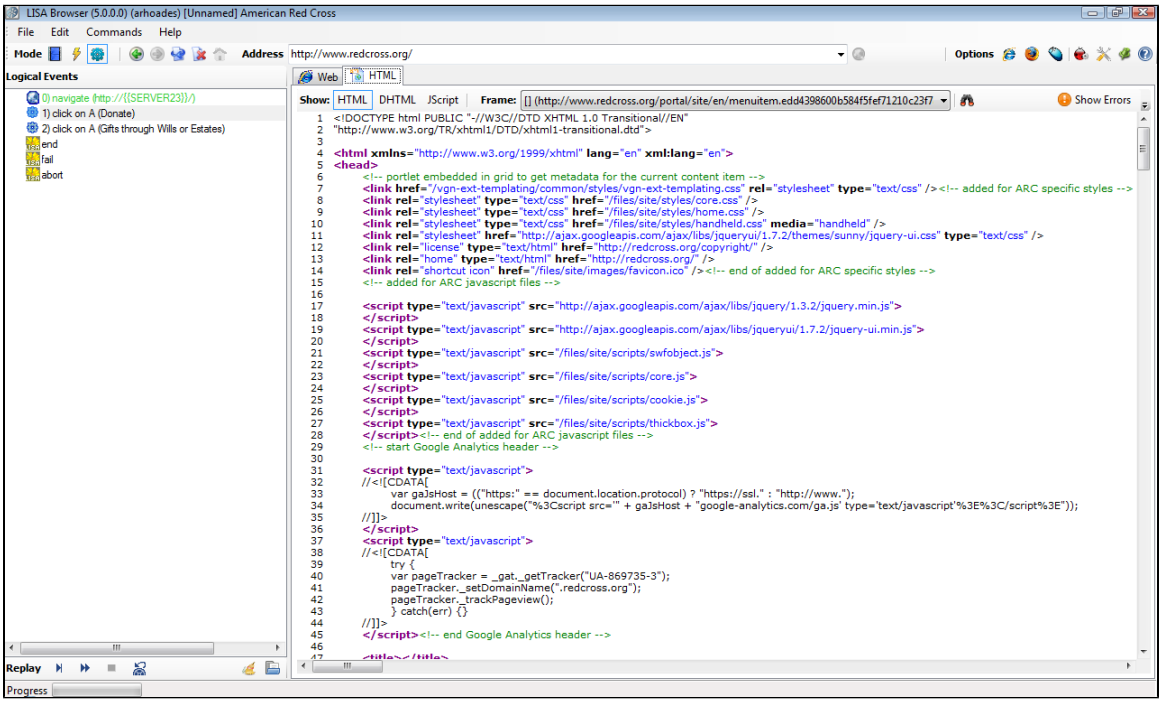



After the playback starts, the steps that have run are shown in green in the Logical Events panel.

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 the **HTML tab** within the Playback mode.




This will open a new menu bar.


- **HTML**: Displays the HTML source.
- **DHTML**: Displays the DHTML source.
- **JScript**: Displays the JavaScript source.
- **Frame**: Displays the frame name.
- **Find**: Opens a search window.
- **Tidy Output**: Shows the tidy output.
- **Show Errors**: Opens the error window.

# Web 2.0 Edit Mode

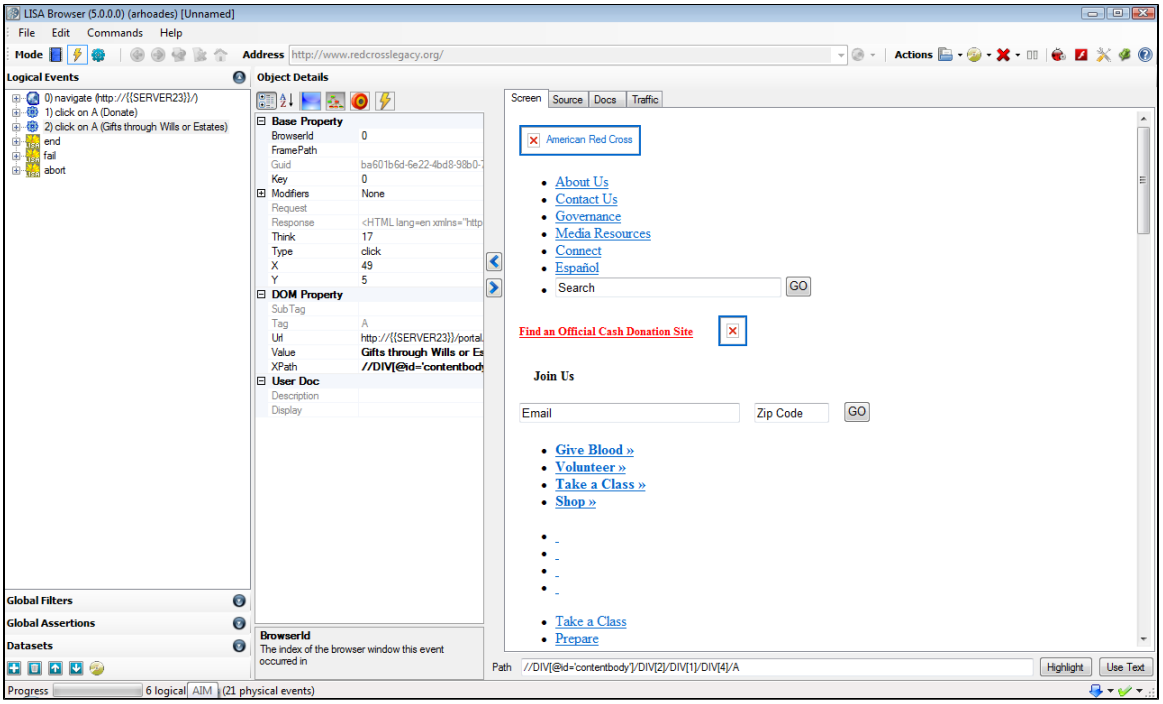
Click the Edit mode option in the toolbar  to open the browser in the Edit mode.

In 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 tab, the following tabs in Edit mode can be expanded.


The main Edit Mode window has a list of Logical Events on the left and the Object details on the right.




See these subtopics for more information.

Event Types
Logical Events
Object Details
Filters
Assertions
Data Sets
Steps_Editing Steps

# Web 2.0 Event Types

HTML pages can generate many different types of events and we will review them here to make it clearer when they are discussed in the rest of the documentation.

There are several sources for events:

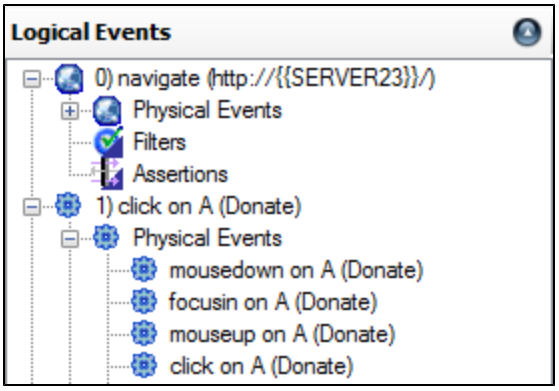
- HTML page (DOM events)
- Browser environment (Native events)
- Plugins (Applet events and others)
- Actual events that are imported by LISA from steps that make use of other technologies or are used as markers (such as the **fail** or **end** events)

There are two types of events:

- Physical Events
- Logical Events

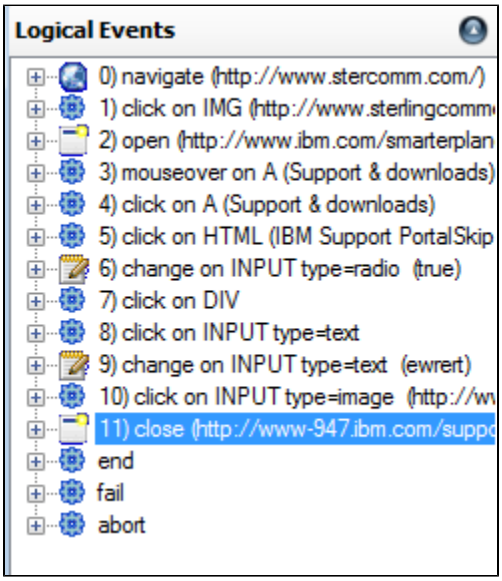
### Physical Events

When we record or play back 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 they are shown in the left panel of the editor.




### 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. They are also shown on the left panel of the editor.




For example, clicking 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 groups these physical events together into an event bucket and mark the click as the bucket's logical event.

### DOM Events

| Icon | Event | Description |
|---|---|---|
| | **Navigate** | 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** | 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** | User presses a mouse button |
| | **Mouse Up** | User releases a mouse button |
| | **Mouse Over** | Mouse hovers over a DOM element area |
| | **Mouse Out** | 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** | User presses a mouse button |
| | **Action** | AWT/Swing's notion of an action event |

| | | |
|---|---|---|
|  | **KeyPress** | 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 example) |
|  | **Mouse Down** | User presses a mouse button |
|  | **Action** | AWT/Swing's notion of an action event |
|  | **KeyPress** | 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.

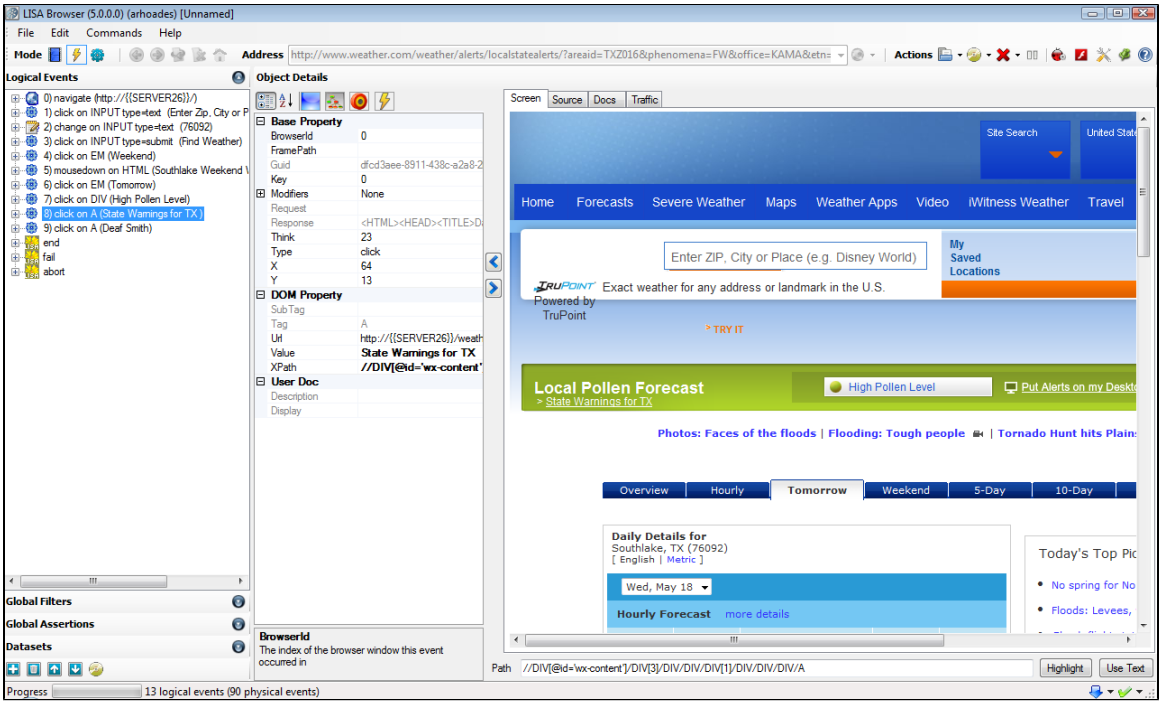
## Web 2.0 Logical Events

After a recording is complete, a LISA test case 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 you might want to inspect or modify events before committing them to a test case, or to edit them on an existing test case.

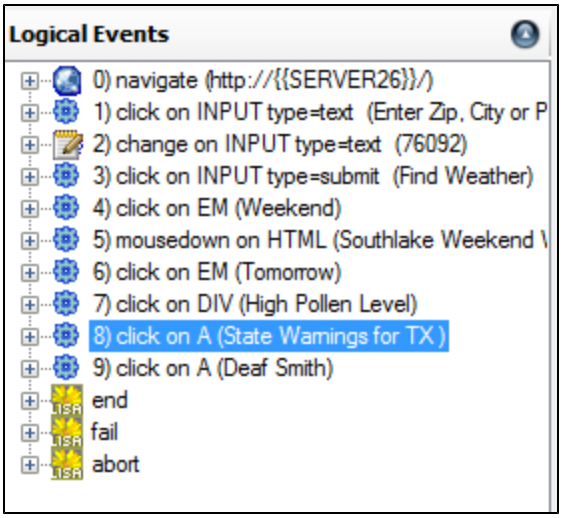
This is the purpose of the Logical Events tab in the browser.

In Editing mode, you are able to view the logical and physical events of the test case. Editing mode displays the Logical Events panel.



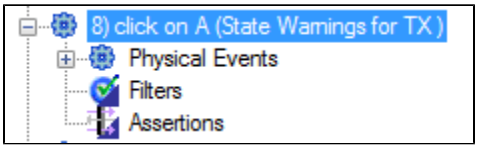

## Logical Events Panel

The **Logical Events Panel** is on the left of the browser window. In the **Logical Events Panel**, you can see the list of events recorded so far.




To view the physical events (mouse clicks, and so on), expand the event by clicking the Plus **+** 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.




You will also see tabs for global filters, assertions and datasets at the bottom.

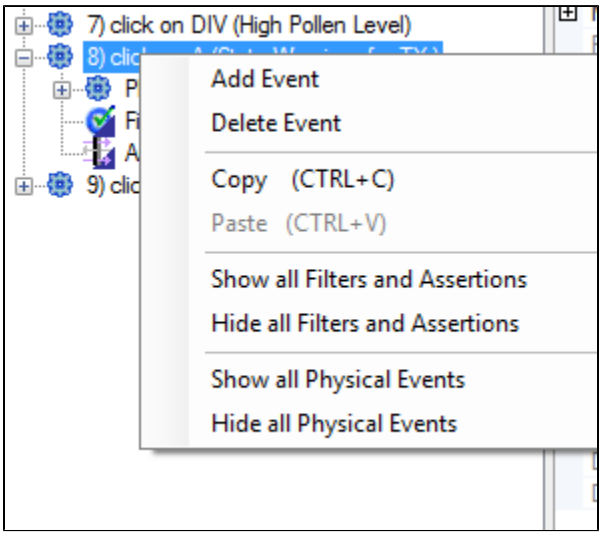

### Add/Modify/Delete Events

At the bottom, there is a toolbar that lets you add, delete, or modify an event, filter, or assertion.

| 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 the test case editor. |
|  | **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. |
|  | **Save an event** | All the changes made to an event in the detail pane (on the right side) are committed. |

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 following menu, which lets you copy, paste, add and delete events.




You can right-click any filter or assertion to add or delete it. You can also physically drag and drop a step from one location to another.

# Web 2.0 Object Details

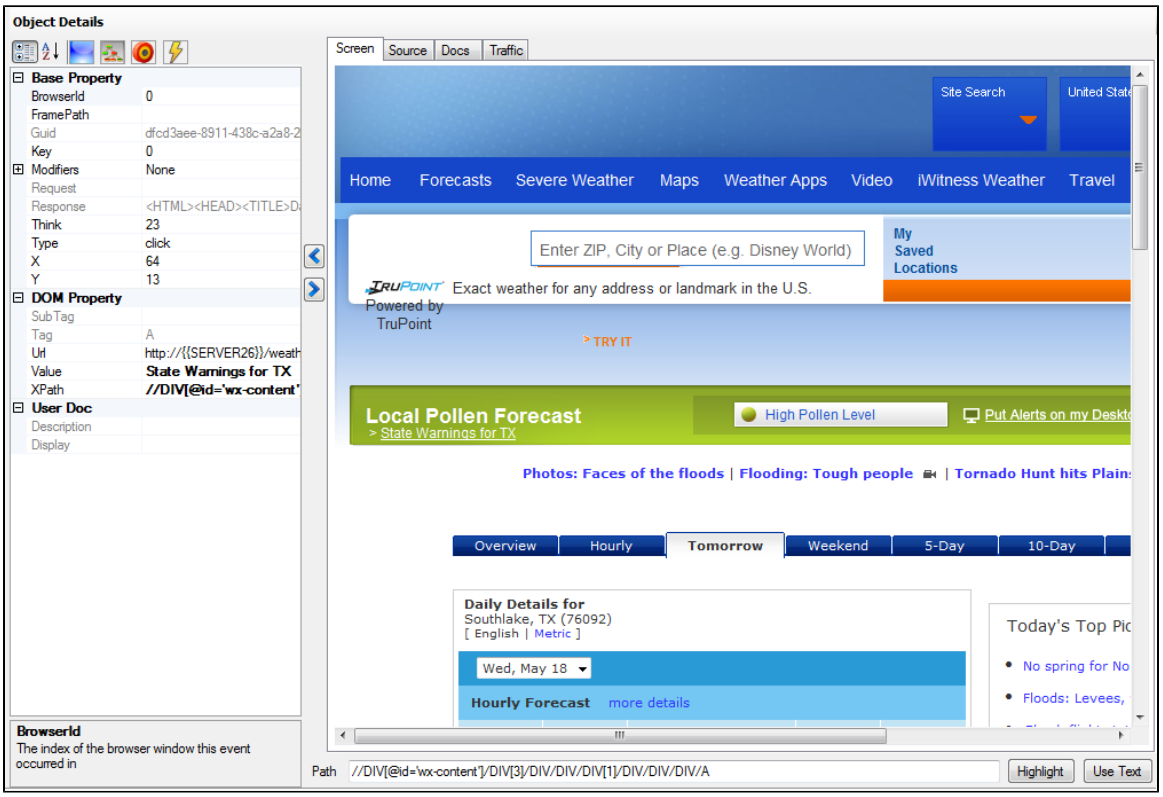
In the Edit mode, you can also see the Object Details Panel. All the details regarding the selected object are listed in this panel.

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** panel.




This panel is divided into two parts: left and right.

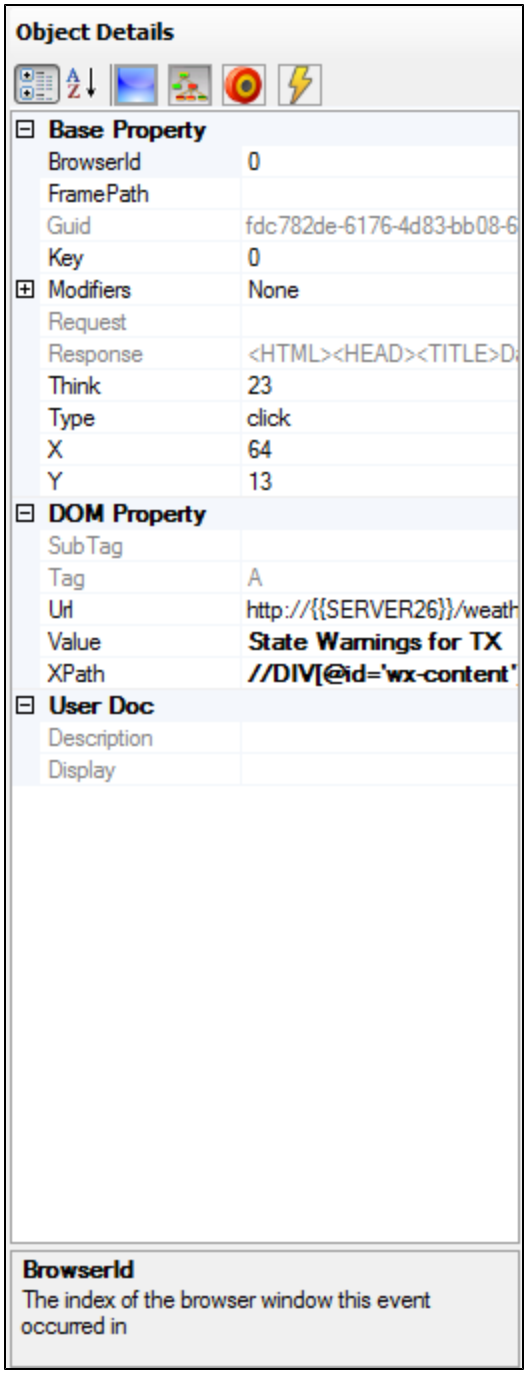
In the left part, you see all the properties specific to an object.

It is divided in two sections:

1. General properties (common to all types of objects)
2. Object-specific properties (for example, DOM/Applet or Native Properties)

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

**Object Details panel**




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

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

| Icon | Button | Description |
|---|---|---|
|  | **Categorize** | Get a categorized view of the data. |
|  | **Sort** | Sort the data alphabetically. |

| | | |
|---|---|---|
|  | **Response** | Show/hide response pane. |
|  | **Hierarchy** | Show/hide hierarchy. |
|  | **Invisible elements** | Show/hide Invisible elements. |
|  | **Scripts** | Enable/disable scripts. |

The **Base property tab** contains:

- **BrowserID**: The index of the browser window this event occurred in
- **FramePath**: The path to the frame this event occurred in
- **Guid**: The unique event identifier
- **Key**: The keyboard key associated with this event

The **Modifiers property tab** contains:

- **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** contains:

- **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 **Userdoc tab** contains:

- **Description**:
- **Display**:

The **Native properties** tab contains:

- **Button**: The label of the button the user used to dismiss the native dialog (such as Open, Save, OK, Cancel)
- **Selection**: An optional selection value that some dialogs offer (such as a file name)
- **Username**: Not used
- **Password**: Not used

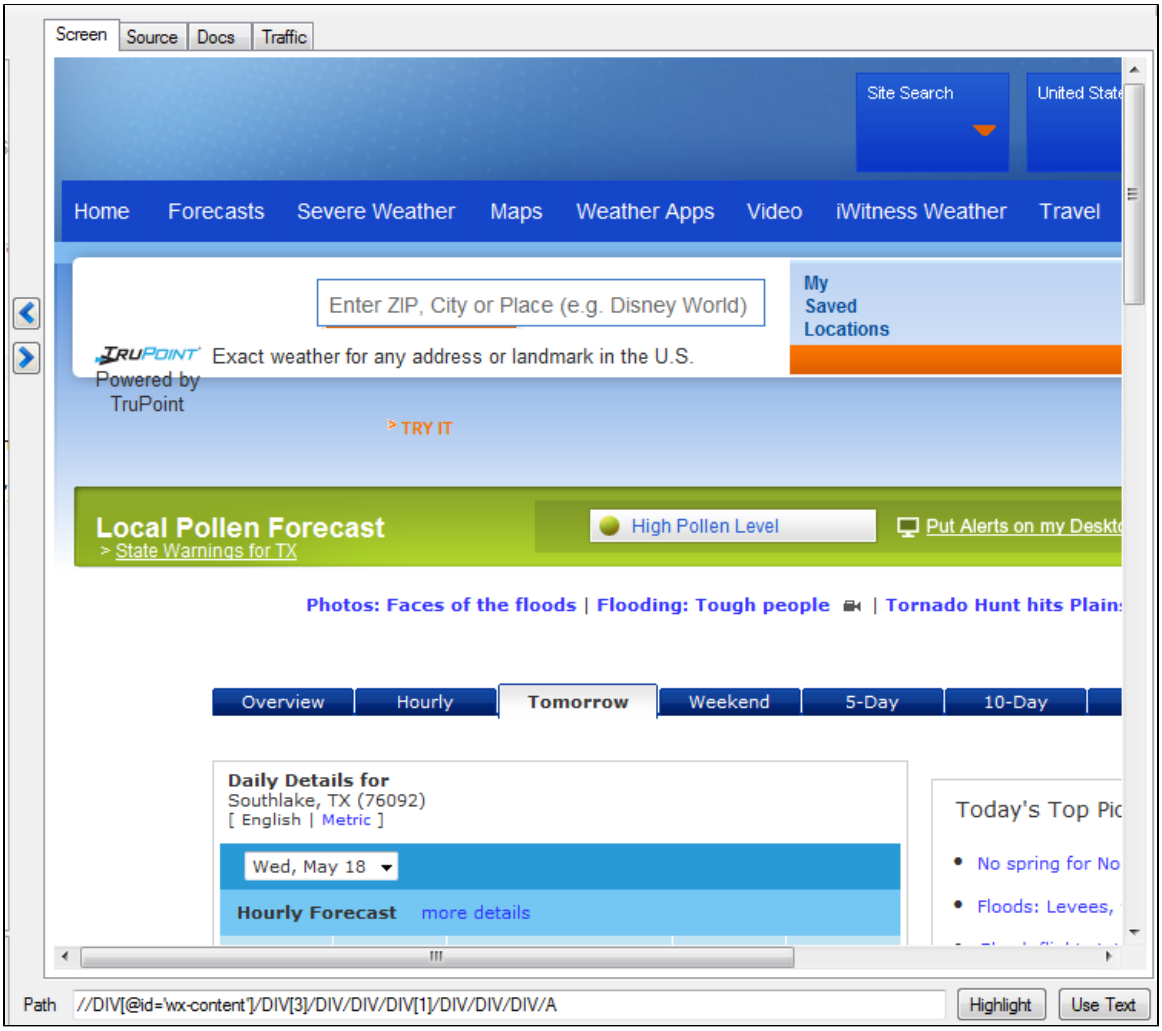
The **Applet properties** tab contains:

- **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

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. Responses are empty for Native Events.




The Response panel has four tabs.

- **Screen**: Shows the actual HTML page
- **Source**: Shows the HTML source of the page
- **Docs**: Shows the docs, if any
- **Traffic**: 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 help you, there are some Browse Buttons available next to those fields. When you click them, the browser will bring up a mode dialog window that contains a browser and helps you select those paths.

The DOM and Applet browsers have three main sections:

- An editable combo box at the top that contains an XPath expression with Select and Cancel buttons
- A tree view on the left 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 is 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 your mind, **Cancel** will discard the browser without any changes.

⚠ 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.

# Web 2.0_Filters

Authoring or executing a test, steps or events are only half 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.

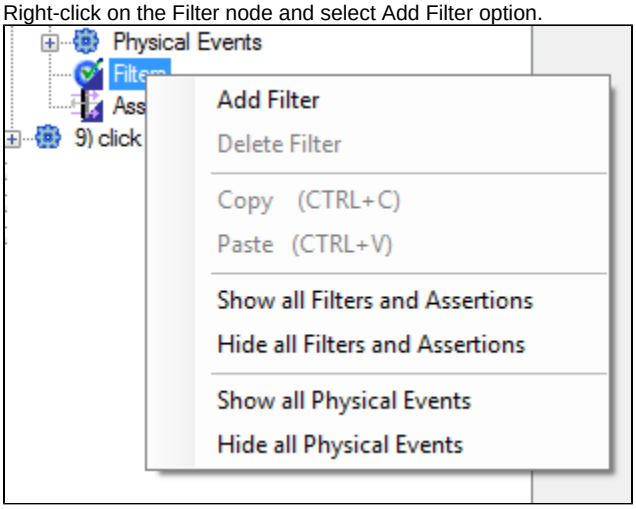
You can inspect, add, remove, or edit filters in Edit mode of the browser.

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

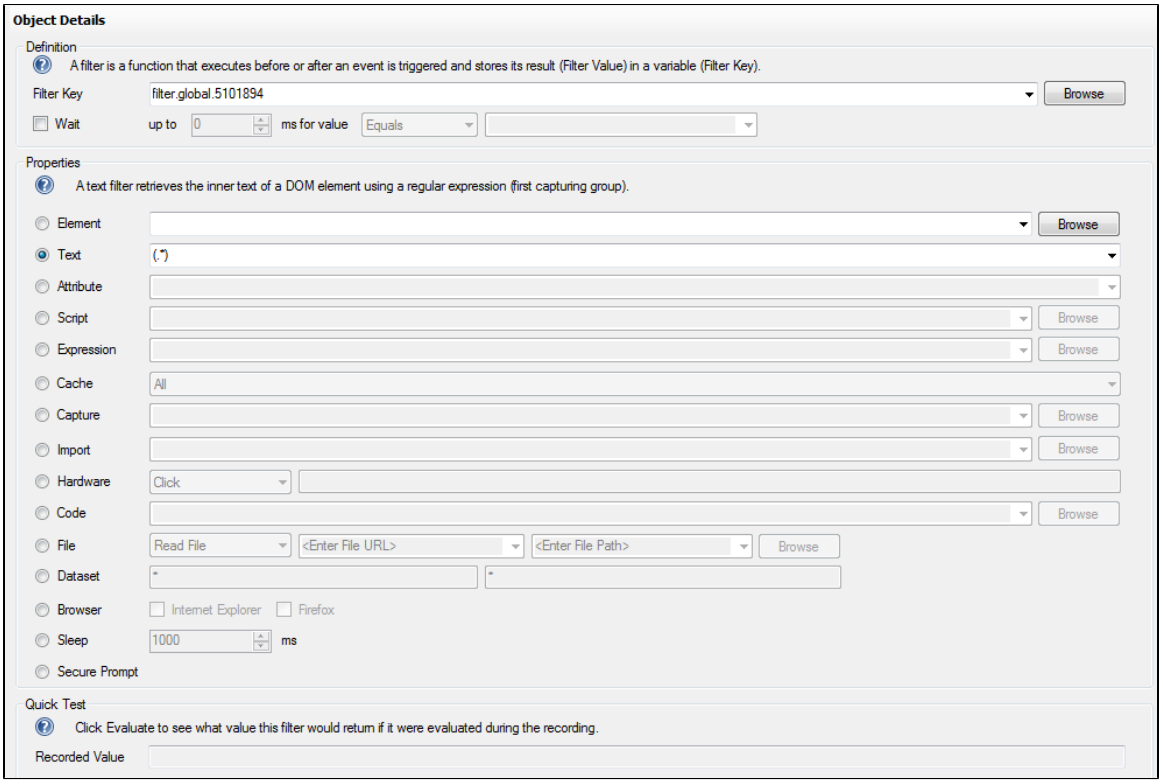
## Event Filters

**To add an event filter**

1. Select any logical event and click to expand the tree.
2. Right-click on the Filter node and select Add Filter option.

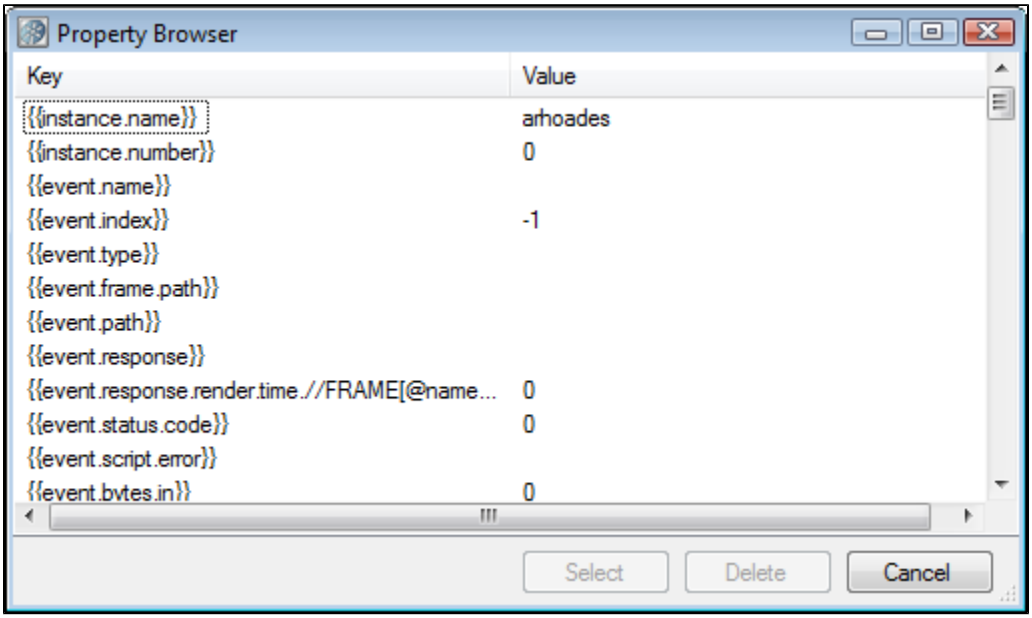



3. The filter editor will open in the right panel.




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

- **Filter Key**: The key is the name of the variable that is going to receive the filter result after it executes. You can enter a name here, or click the Browse button to open a Property Browser.

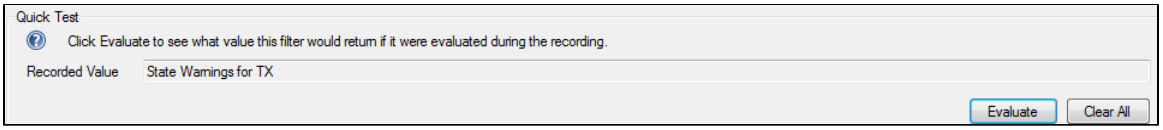



5. Find the element name, and click Select to add.
6. **Filter Properties**: the following types are available. Clicking on any of the filter properties will give its description.

- **DOM Element**: Extracts an HTML element object from its event's DOM.
- **Text**: Extracts a piece of text from its event's response.
- **DOM Attribute**: Extracts an attribute value from its event's response.
- **JavaScript**: Executes arbitrary JavaScript code in the context of its event's response.
- **Expression**: Lets you combine other filters.
- **Cache**: Clears the specified browser cache.
- **Capture**: Saves the current response to the specified location.
- **Import**: Makes the specified file (JavaScript or Java) available as a library to the running application.
- **Hardware**: Executes the selected mouse or keyboard action on the specified desktop.

- **Code**: Executes arbitrary .Net code that has access to all the LISA browser variables.
- **File**: Lets you execute a variety of operations on file, local or remote.
- **Data Set**: Automatically extracts data out of the specified element into a data set according to row and column path rules.
- **Browser**: Toggles the state of the playback window to use the selected browsers.
- **Sleep**: Sleeps for a specified amount of time.
- **Secure Prompt**: Lets you fill in some variables at runtime if you do not want them persisted.

**Quick Test**: Click to see what value this filter will return.
**Evaluate**: Click to evaluate the filter. The filter runs with the response it got during recording and displays its result in the **Recorded Value** field.




### Global Filters

1. To add a global filter, click the Global Filter 

button at the bottom of the Logical Events pane.
2. Click the Add  button to add a global filter.
3. This will open the same filter editor as for the event filter. Enter the appropriate criterion so that the filter is applied to all the events.

### Example

Imagine that a 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 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.

If you had two tables whose number of rows should always add up to the same value, you could add two filters like the previous one (say "`f1`" and "`f2`"), and add another Composite Filter "`f3`" whose value is "`f1` + `f2`".

After 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

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.

## Web 2.0_Assertions

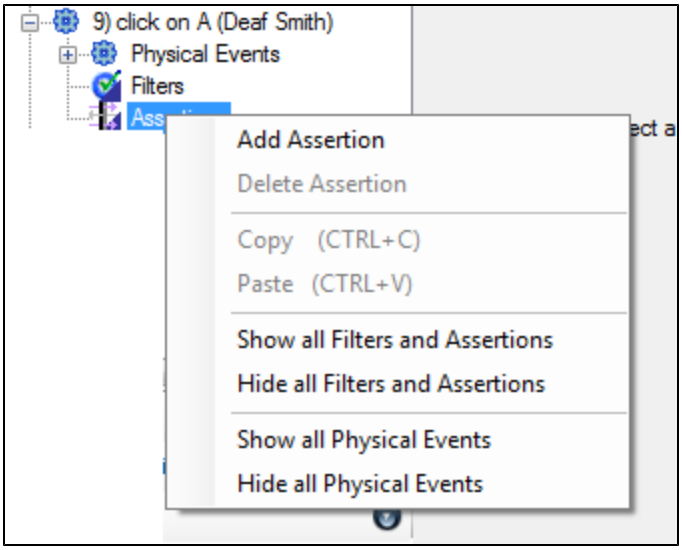
Assertions are functions that execute after an event is fired, and return a Boolean value. If the return value is true, the test proceeds normally; otherwise what happens depends on the assertion itself. It is a typical if-then-else scenario.

You can inspect, add, remove or edit assertions in the Events tab of the browser.

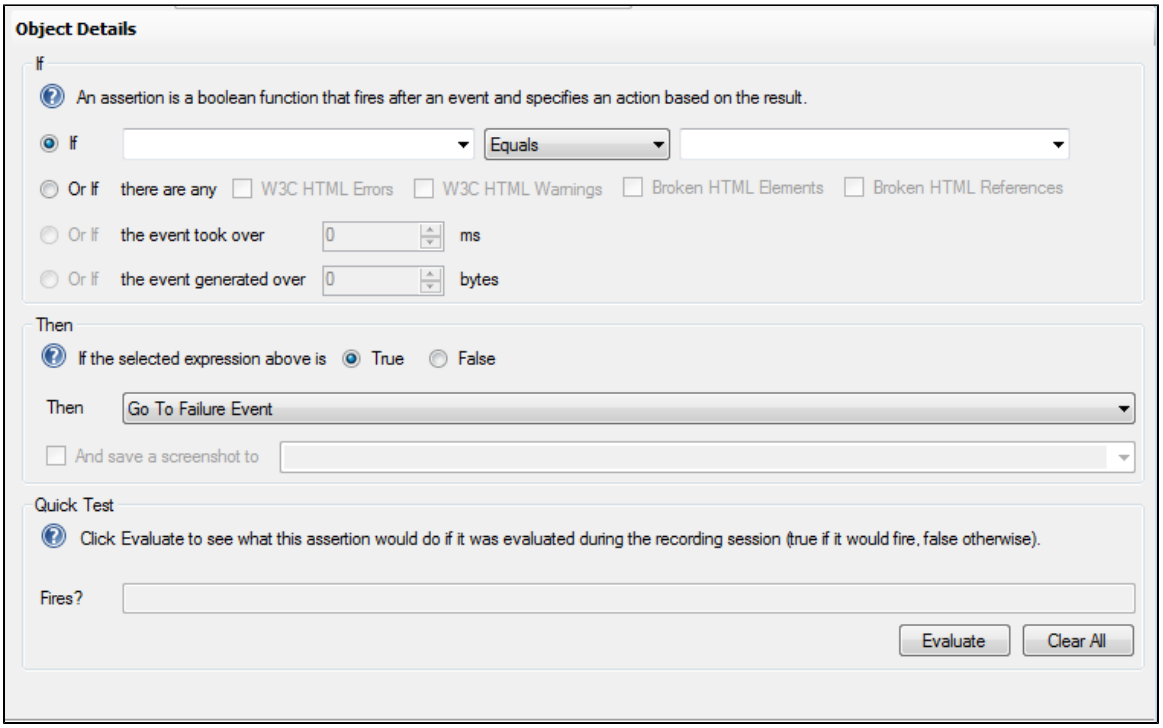
### Event Assertion

#### To add an event assertion

1. Click any logical event to expand the tree. Right-click the Assertion node and select **Add Assertion**.




2. The Assertion Editor will open in the right Object Details pane.




3. On the Assertion Object Details pane, after a description of the assertion type that is selected, you can see the following controls. You must select two parameters to be compared.

### If box

- **If**: Select the event that 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. Types of operations are:
  - **Equals**: Compare the two sides of the expression for equality.
  - **Not Equals**: Compare the two sides of the expression for non-equality.
  - **Matches**: Attempt to match (as regular expression) the left or the right side of the expression against the other side.
  - **Not Matches**: Attempt not to match (as regular expression) the left or the right side of the expression against the other side.
  - **Less Than**: Compare the two sides of the expression as numeric values for order. Returns false on non-numeric values.
  - **More Than**: Compare the two sides of the expression as numeric values for order. Returns false on non-numeric values.
- **Or if**: There are any **W3C HTML Errors**, **W3C HTML Warnings**, **Broken HTML Elements**, or **Broken HTML References**, select the ones on which you want to assert here.
- **Or if**: The event took over a specified number of milliseconds.
- **Or if**: The event generated over a specified number of bytes.

### Then box

- **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 box

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 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 a editor similar to the Event Assertion. Enter the required details to run the assertion on every event.

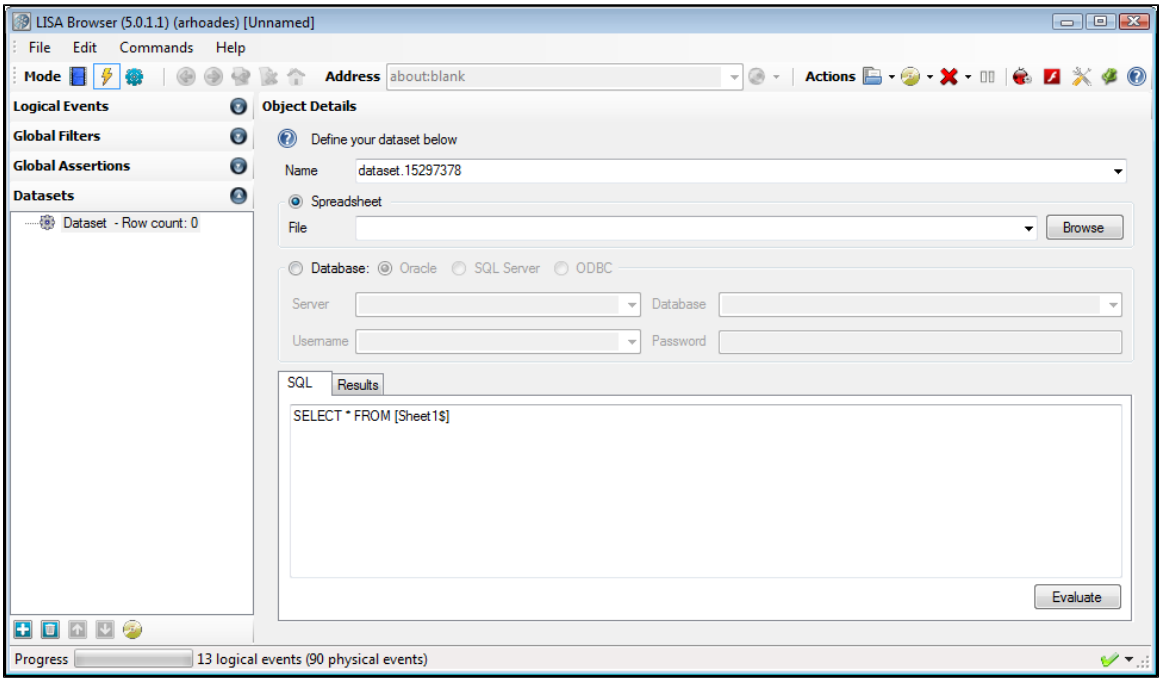
## Web 2.0 Datasets

Datasets can only be applied globally, that is, to the entire test cycle.

**To add a global dataset**

1. Click the Datasets button 

located at the bottom of the Logical events pane.
2. Click the Add button  to add a dataset. This will open the dataset editor in the right Object Details pane.




3. Define the dataset in the dataset editor.

- **Name**: Enter the appropriate name or accept the default provided by LISA.
- **Data Source**: Select the source of the dataset, either from a spreadsheet or from a database.
    - **Spreadsheet**: Select this to attach an Excel spreadsheet 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. Select the database type: Oracle, SQL Server,or ODBC. Enter all the details related to the database.

4. Enter the SQL query, and click Evaluate to evaluate the results.

## Web 2.0 Editing Steps

One of the major strengths of LISA is the outstanding ability to create and run tests that make use of a mix of different technologies (web, J2EE, web services, Swing, and so on) 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.

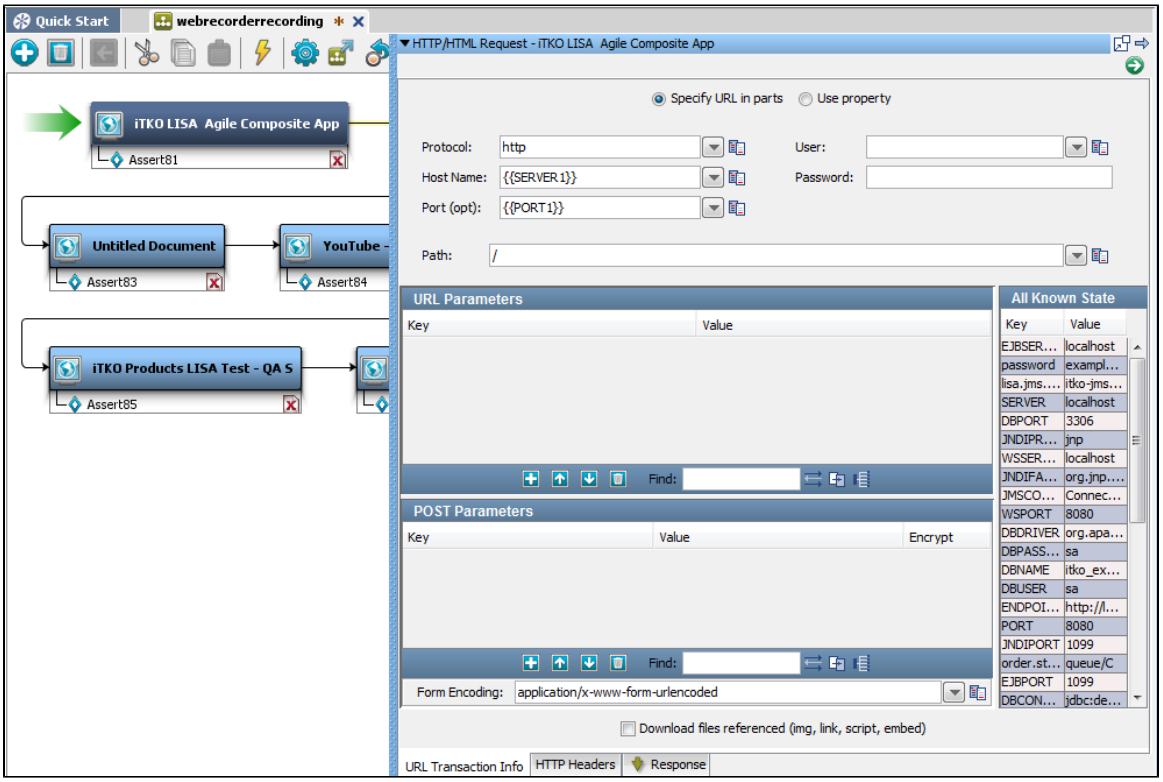
**To add steps to an existing test case**

1. Open an existing test case and then start the Web 2.0 recording.
2. After you are done, save the recording. This will close the web browser.
3. This will add all the recorded steps to the existing test case in the Model editor.

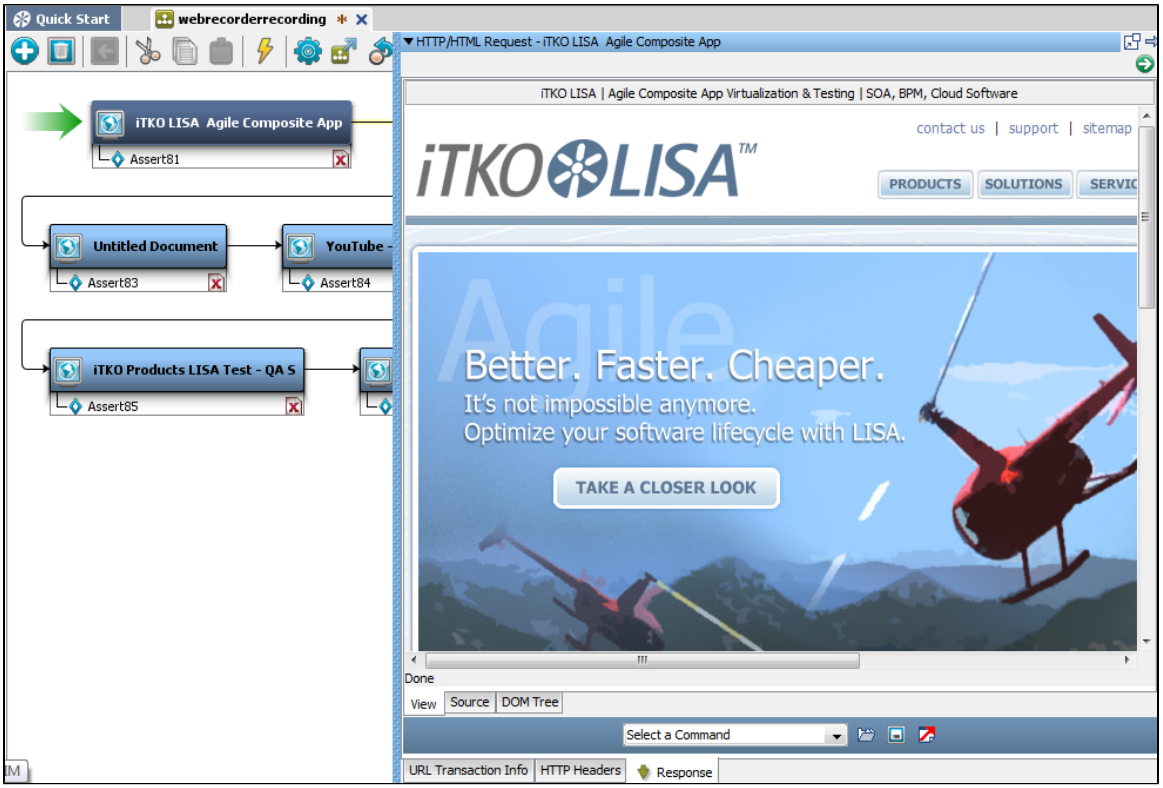
**To edit/add/delete Web 2.0 steps**

1. Request a **detailed view** of existing steps in LISA Workstation.
2. Double-click the step in the Model editor for which you need a detailed view.
3. This will launch the editor of the respective test case.

For example, we have chosen to view the detailed view of an HTML type of test step, so LISA Workstation has opened a **HTTP/HTML Request editor**.



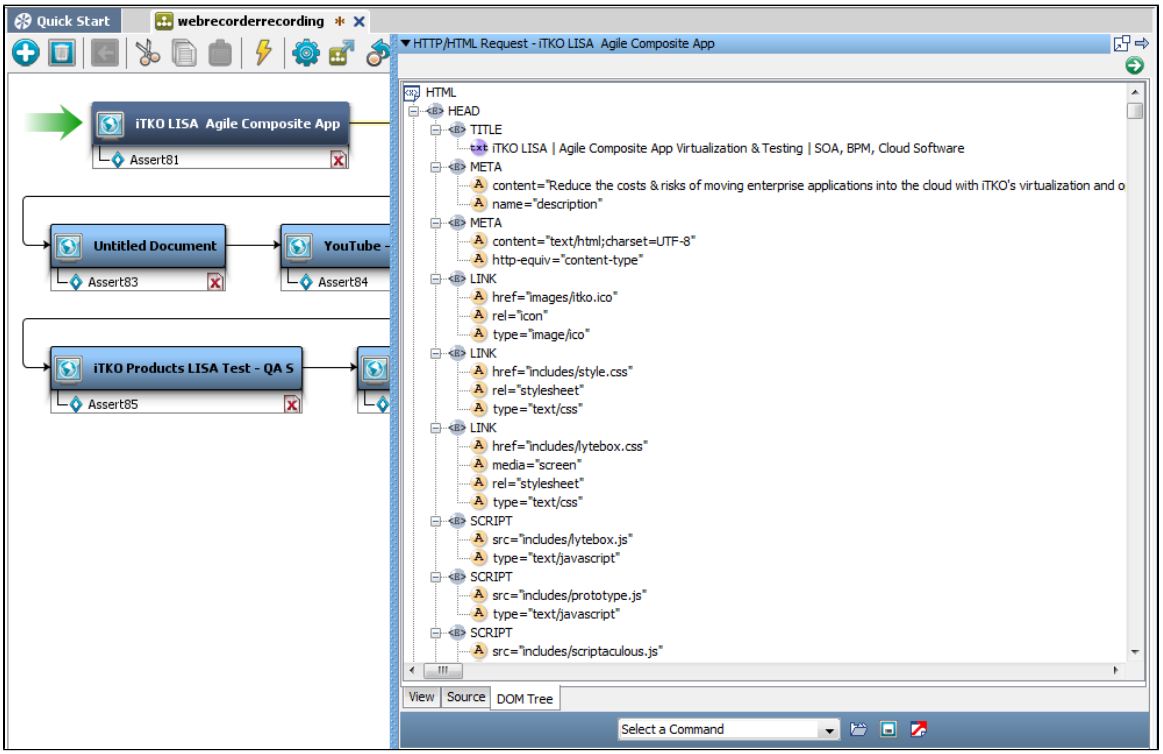

You can also view the response of the Web 2.0 steps exactly the same way you do with regular web steps, by selecting the **Response** Tab in the step's detail pane.




For HTML pages, these responses come in three types:

- Headers
- XML source
- DOM tree

The following image shows the DOM tree response.



# Web 2.0 Debugging

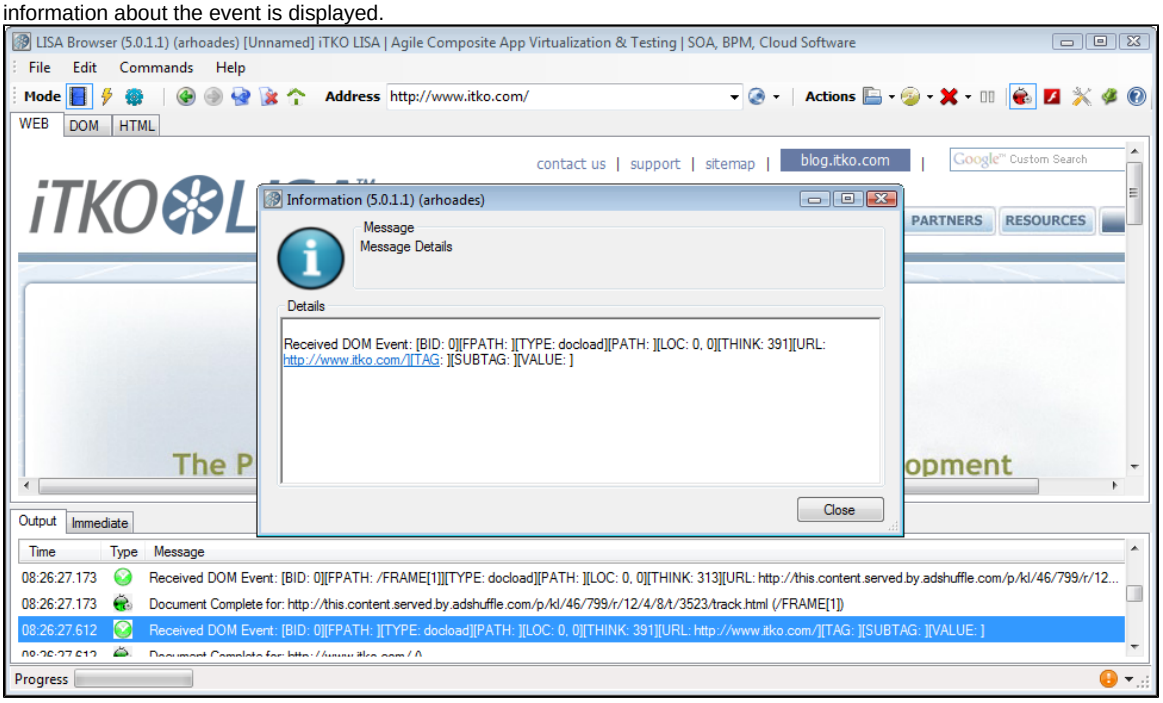
Debugging is available to debug code or see more about a problem area. Instead of clicking Next repeatedly for every step during playback of a recording, you can set one or more breakpoints in the events list and click Play or Replay. After you have reached the events of interest, you can step through one by one.

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

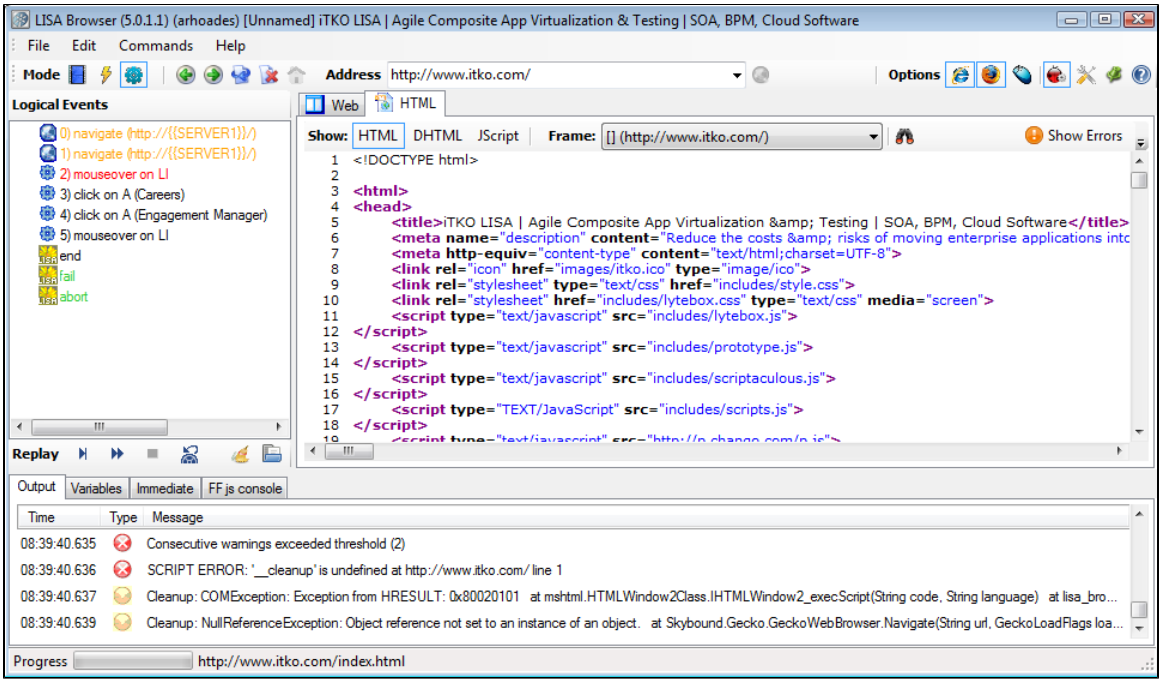
## Debugging within the LISA Browser

Click the Show/Hide Debug Window  icon in the top toolbar to open a debug window at the bottom of the screen with a set of tabs.


The **Output tab** logs information about execution and potential errors or warnings. When you double-click a message in the debug window, information about the event is displayed.




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, and so forth.




The debug window, when opened in Playback Mode, displays the same tabs as in the Recording and Edit modes, with the addition of one Variables tab.

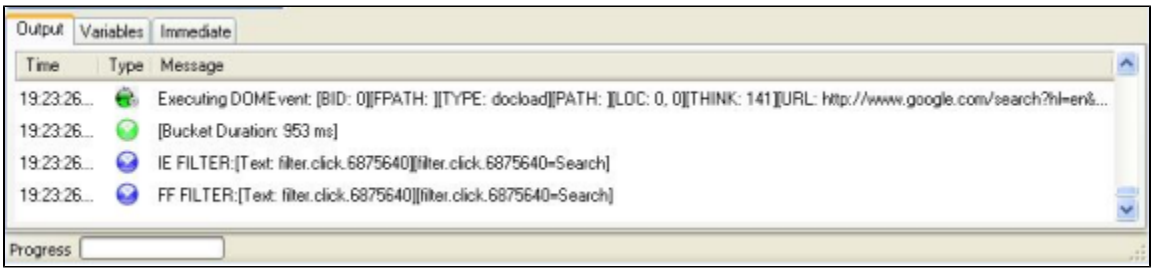
The Variables tab shows you all the variable names and values at the current step. The ones highlighted in red are the ones that were potentially modified by the last event.

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

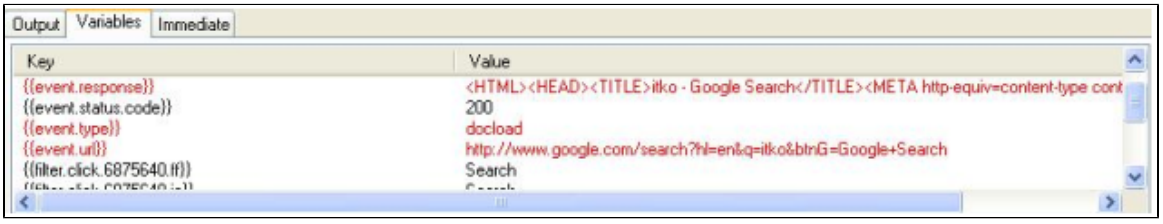
The Split toggle also applies in this source view for further comparison.

## Multiple Browsers

When multiple browsers are selected you get the logging information for each of them.



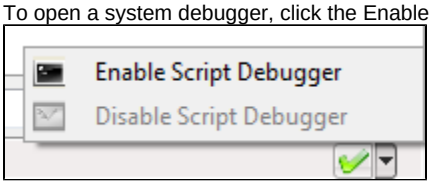

Filters also get executed for each browser and the browser abbreviation is appended to the filter key so they can be referenced individually.




Script execution is also performed for each selected browser.


### Opening a System Debugger

To open a system debugger, click the Enable Script Debugger button at the bottom of the window.

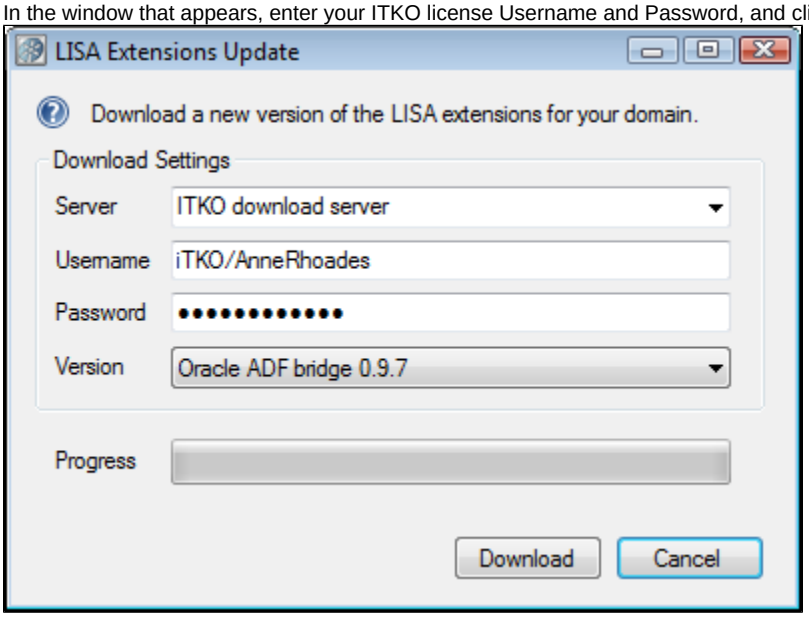



This will open a script debugger if present in the system.

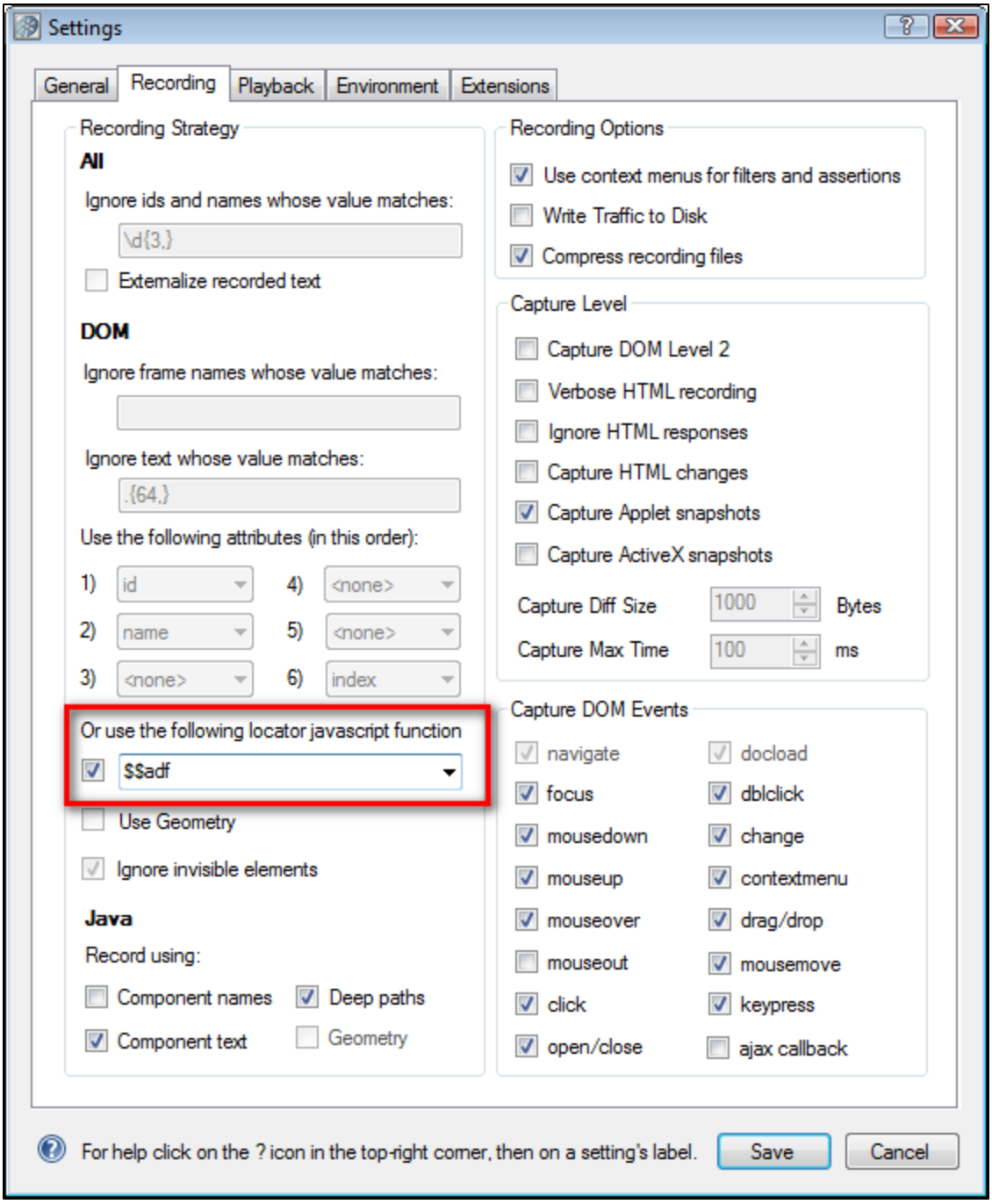
## Web 2.0 Setting up ADF Extensions

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

1. To update the extensions, open the LISA Browser and from the **Help** menu, select Extensions Update.
2. In the window that appears, enter your ITKO license Username and Password, and click Download.




3. To update the settings, open the LISA Browser.
4. From the **Edit** menu, **select Browser Settings**. This will open the browser settings window.
5. Click the **Recording** Tab.




6. In the **Or use the following locator javascript function** box, select the check box to enable it.
7. Enter **$$adf** in the drop-down list.
8. Click Save to save these settings.

## Web 2.0 Running Browser Standalone

While taking advantage of the full power of LISA (multi-technology, data sets, companions, and so forth) requires running the LISA browser 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.

# Web 2.0 Troubleshooting

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

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

- Does the scenario you are testing work in a browser, especially in Internet Explorer? 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 there?
- Have you familiarized yourself with the different settings in the browser and checked they couldn't address the problem? For example: applets not being recorded because applet support was disabled in the settings, or a site that uses mouseover events is not replaying correctly because mouseover events are not checked, or timeouts are too low for your applications (See Web 2.0 Settings)?
- Are you sure the test is not replaying correctly because of a lack of parameterization? Specifically, have you looked at the debug output window (or logs) to see what went wrong during replay (for example, "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 are likely to ask for? If the problem triggers an error dialog, you will have an option to generate an error report that will contain all this required information and you can send that to ITKO Support. 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 Test Manager 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, 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, you will probably need to contact support@itko.com.

# Web 2.0 XPath Syntax

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 and not supported functionality is as follows.

## Supported Functionality

- <Tag1>**/**<Tag2> to look for an element child (for example, TR[1]/TD[2]).
- <Tag1>**//**<Tag2> to look for an element descendant (for example, TABLE//INPUT)
- **..** to select the parent of an element (for example, TABLE//TD[@id='abc']/../TD[1])
- **>>** and **<<** to select the next and previous sibling of an element respectively - the standard axis names following-sibling or preceding-sibling can be used instead with >> and << just being shorthand for it - (for example, TABLE//TD[@id='abc']/../>>/>>/TD[1]))
- <Tag>**[@AttributeName='AttributeValue']** to look for an element with the specified tag meeting the AttributeName='AttributeValue' condition (for example, INPUT[@id='abc'])
- <Tag>**[text()='innerText']** to look for an element whose inner text equals the specified one (for example, 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 (for example, TD[matches()='\s*Hello\s*'])

- Regex search is supported with the syntax **matches()='<regex>'**.
- Support for simple string matching is provided with the **contains()** function (for example: //A[contains('Install')]).

## Unsupported Functionality

- Other XPath axes.
- 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).

# Web 2.0 Scripting Objects

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 that 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.
- **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 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 needed 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. 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 javaimport(string javafile)**: Automatically imports a Java source file into the currently running Java process, making its functions available to subsequent filters in the test. 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 encapsulates 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 object at the specified index.
- **public string toString()**: A textual representation of the dataset.

**DataRowWrapper**: A JavaScript class that encapsulates 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.

# Web 2.0 Command Line

The LISA browser command line program, 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 update**: Launches the software update dialog

And **options** are one or more of the following:

- **-a**: Autostarts the test specified by -f in playback or drive mode. The default is **false**.
- **-n**: The number of instances to autostart in drive mode. The default is **0**.
- **-f**: Specifies a recording file to load on startup. The default is **none**.
- **-c**: Specifies a config file to load on startup. The default is **none**.
- **-i**: Makes the driver console invisible in drive mode. The default is **false**.
- **-x**: Autoexits the program on test end if it was autostarted. The default is **true**.
- **-stg**: Specifies a path to an XML file that lists instances with test and config files to use.

See How To - Run Load Tests for the syntax required by the -stg option.

# Web 2.0 How Tos

This section is a list of how-to's that cover typical scenarios and how to deal with them. First read the Web 2.0 User Guide to familiarize yourself with the basic concepts of LISA Web 2.0.


How To Learn About Websites and Frameworks
How To Generate Random Data
How To Capture Dynamic HTML for later test editing
How To Deal with Time-sensitive Events
How To Parameterize dynamic data entry in loops
How To Deal with Dynamic Elements
How To Extract Complex Data from a Page
How To Ajax Auto-complete Fields
How To Write Custom Web 2.0 Steps
How To Write Cross-browser Tests
How To Use Pathfinder Integration
How To Write Java Swing and WebStart Tests
How To Debug a Test
How To Use Global Filters and Global Assertions
How To Interact with External Resources
How To Run Load Tests
How To Run in a Non-privileged Account or on 64 bit Operating Systems
How To Record and Replay against Non US-English Websites
How To Run in Crash Dump Mode

How To Set a Web 2.0 Browser Timeout


# How To Learn About Websites 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 demos at **Appcelerator Example Applications**)
- **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)
- and many others

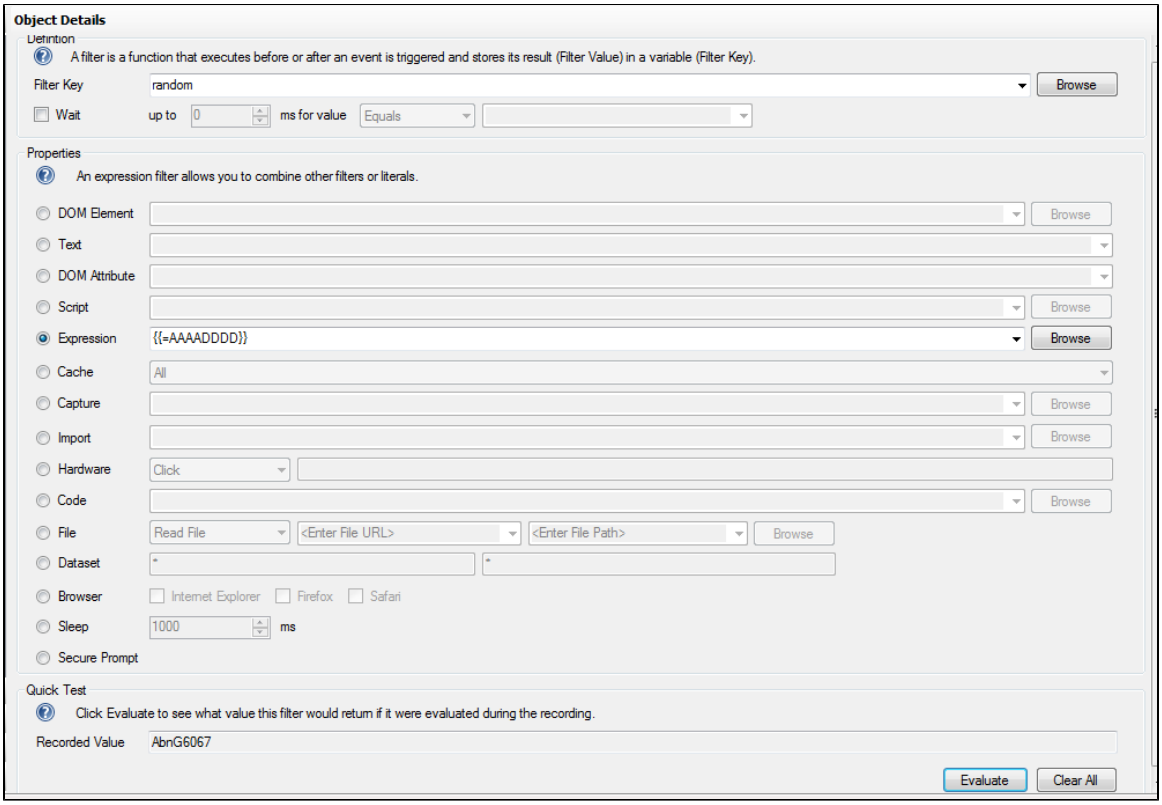
If there are any websites or frameworks that do not work in the DOM browser, contact ITKO Support.

# How To Generate Random Data

There are many places in LISA where string generation patterns can be specified. A random string can be created 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 latter.
- **A**: Replace with a random digit or letter of either case.
- **P**: Replace with a random punctuation character (.,-\/).
- **.**: Replace with a random printable character.

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, use the following syntax: `=pattern` (for example, `=LDDD`).




Use a square bracket expression 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 these, you can specify two types of modifiers. To specify an exclusion list, use a brace expression, "{}". For example, the pattern "D{0,2,4,6,8}" generates a random digit that will only ever be an odd number.

An asterisk expression can 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, the 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 alphanumeric 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 three characters, but no more than five characters long.

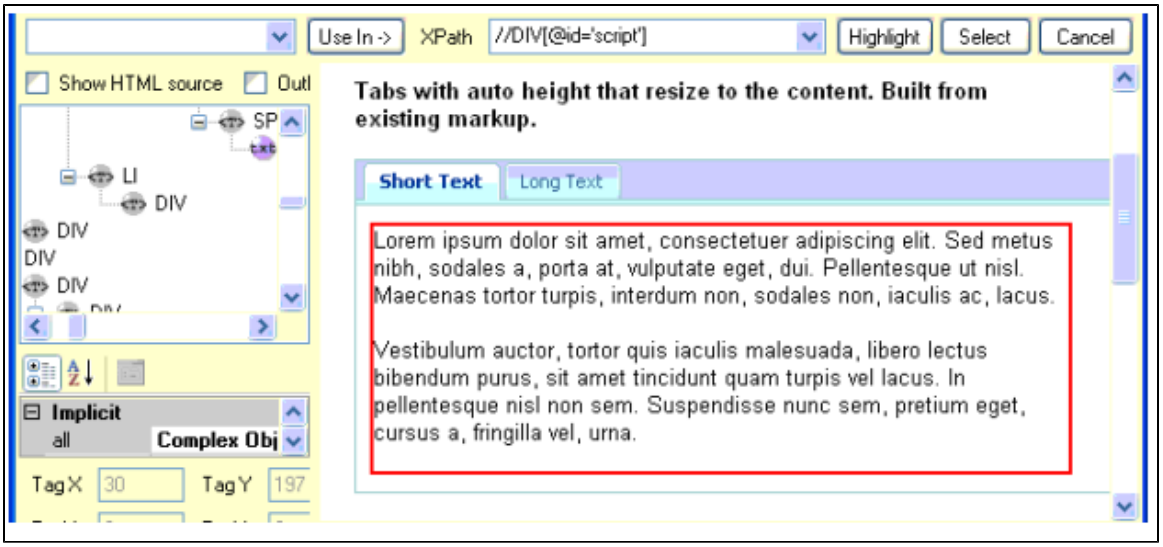
# How To Capture Dynamic HTML for later test editing

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 selected in the recording settings) so all the events between two page loads will have the same response (as you can see by looking at the events response tab).

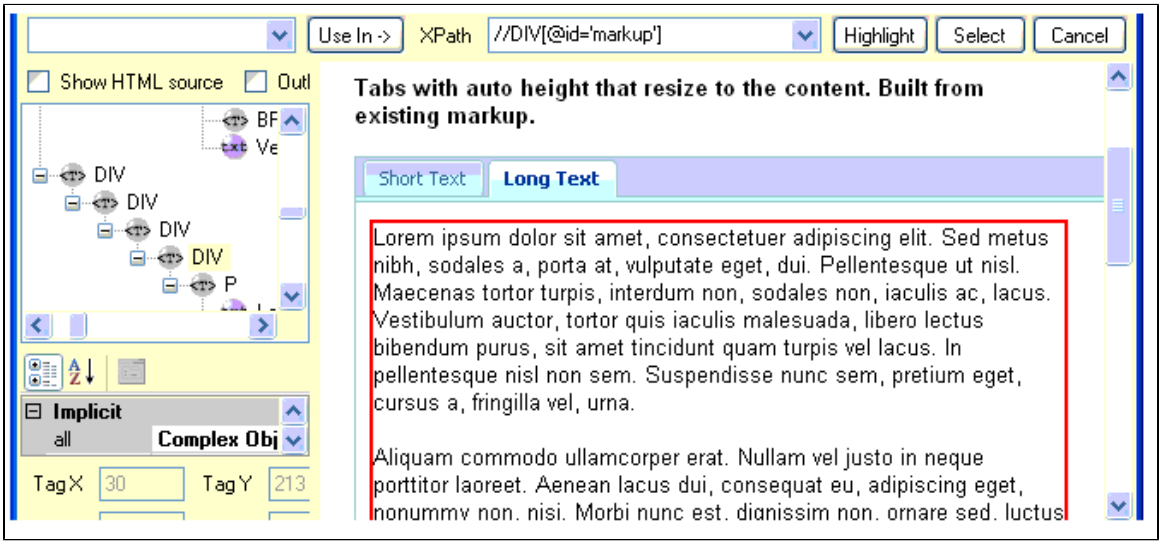
It is usually sufficient and keeps the test size relatively small but for some websites that are very JavaScript-heavy, where the page changes a lot purely on the client as a result of JavaScript executions, it can make it difficult to edit the test or add filters and assertions through browsing.

For example, if you have a page that has tabs (that are preloaded and just change visibility when you click them) and you click 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 see the first tab.

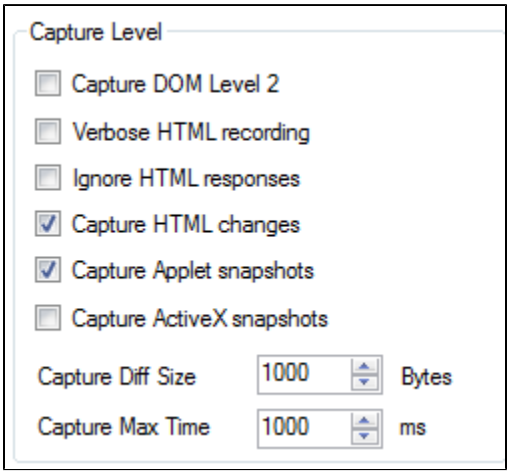
The following illustration shows browsing of a click event on the second tab without **Capture HTML changes**.




Here is browsing of a click event on the second tab with **Capture HTML changes.**




If you add **Capture HTML changes**, 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.




This would dramatically increase the test size, so to avoid this, there is another setting, **Capture Diff Size**. This 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 larger than 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 from the page load response and the diff.

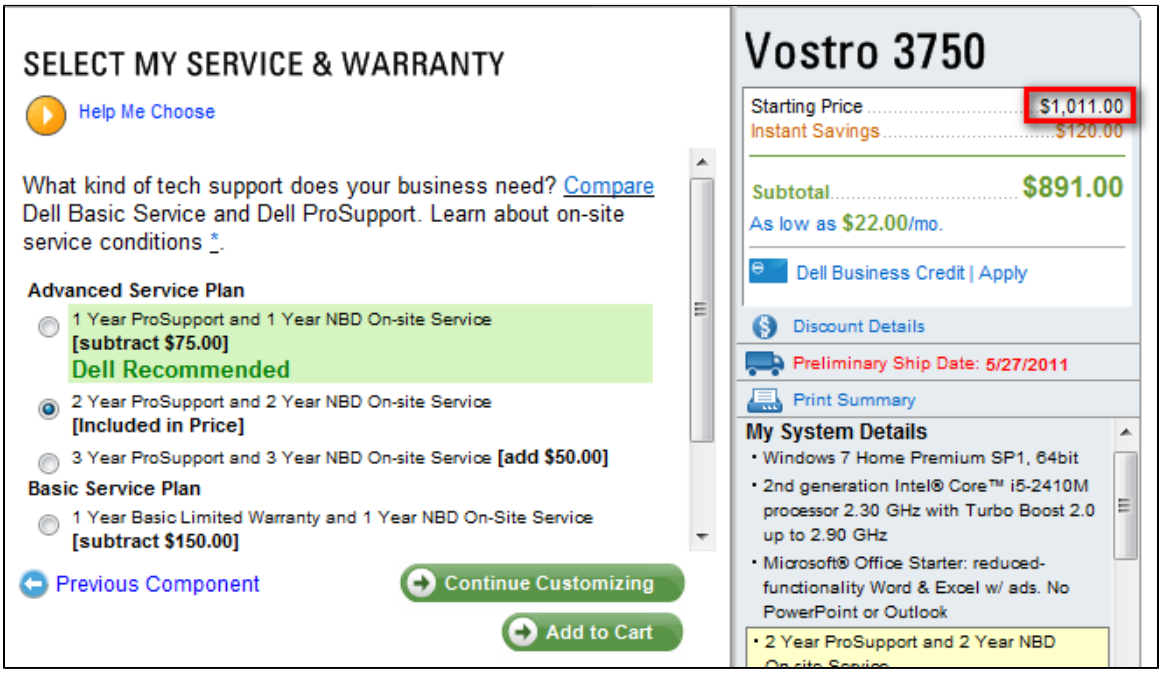
The reason it does not always do the diffs is because above a certain size they become very expensive to compute and would take too long (which is where the second setting, **Capture Max Time**, comes in).

The defaults are set to reasonable values and should not need to be changed in general. So if test size is not a concern, it is recommended that you select this to make it easy to edit the steps/filters later on.

## How To Deal with Time-sensitive Events

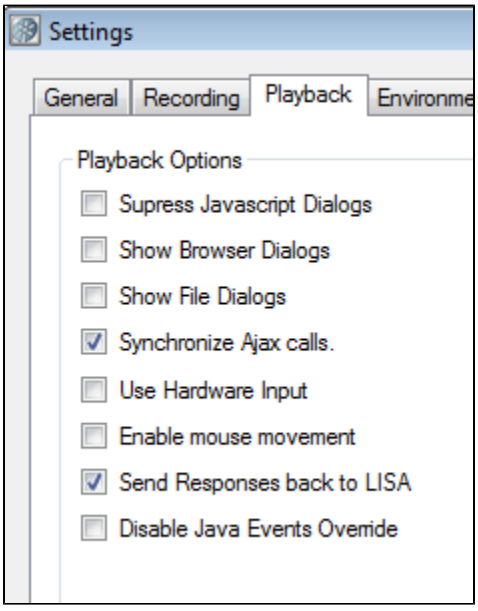
Often, what makes Web 2.0 sites difficult to test is the asynchronous nature of their interactions with the user. In a pure request/response environment, there are no race conditions or multiple threads of execution to worry about (at least from the client's perspective), but Web 2.0 environments introduce those difficulties everywhere: frames, set Timeout and particularly asynchronous Ajax calls in HTML pages and any Java applet code or Flash/Flex code make heavy use of multiple threads of execution.

The main testability problem this 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 following picture?



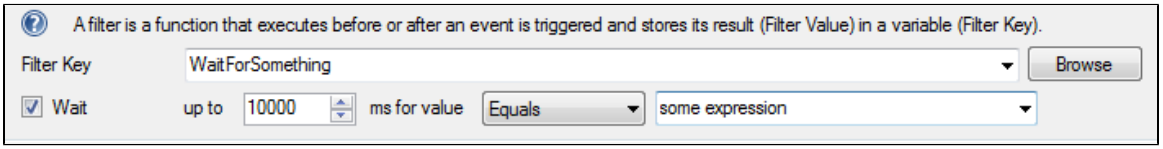

The DOM browser gives you two powerful tools to deal with those situations.

### Synchronize Ajax calls setting



The **Synchronize Ajax calls** parameter 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 Some websites that do Ajax event pushes through regular polling do not work well with this setting. If you do not know how the application is coded, try to 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 previous example.

### Wait for option of any filter



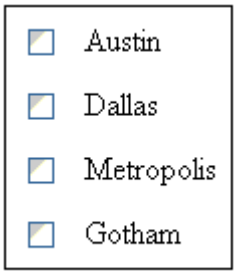

This will make the test execution wait until either the wait for condition is met, or the timeout specified expires. Another typical use of this is for pages that have splash screens. For instance, if the splash screen is a div containing the text "Loading..." pick a text filter with DOM element the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

## How To Parameterize dynamic data entry in loops

To understand this example, you should first be familiar with XPath expressions, as described in Web 2.0 - XPath Syntax.

We will examine some typical scenarios to understand how to deal with dynamic data in a loop.

First, let us say you have a list of cities in a table with a check box next to them, and you want to select the check boxes.




The code for a table row here is

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

   <td>Metropolis</td>
</tr>
```

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

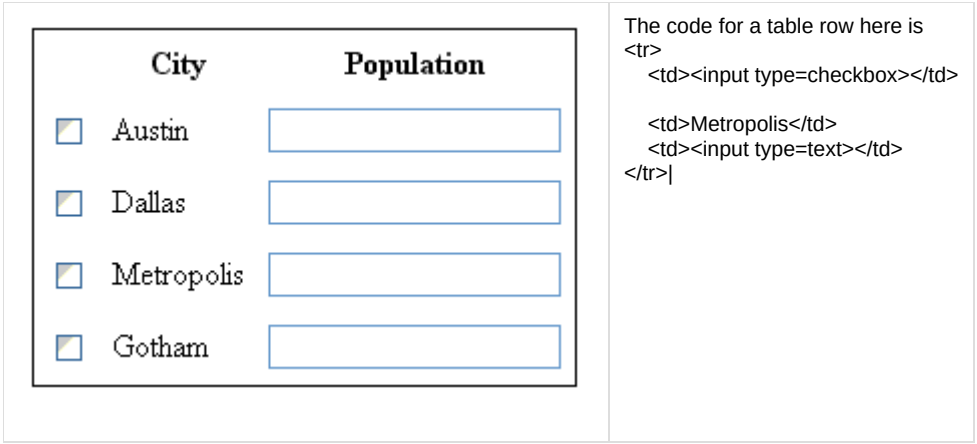
First define a looping variable. The easiest way to do this is create a filter of type **Expression** on an event before the change event. Call it **i** and set its value to **0**.

Then parameterize the XPath of the change event to: `/HTML/BODY/TABLE/TBODY/TR[{{i}/TD[1]/INPUT}}`. To increase 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 increase the value of **i** by **1**.

Finally, 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 add an assertion on the change event whose expression reads **If `i` Less Than 5 Then "Go To change event"**.

Alternatively, if you are coding-inclined, 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 .cells(0).childNodes(0).checked=true; return 0;**.

For something a little more complicated, let us say we have a list of cities coming from a dataset with population number and we want to input those values on the page.

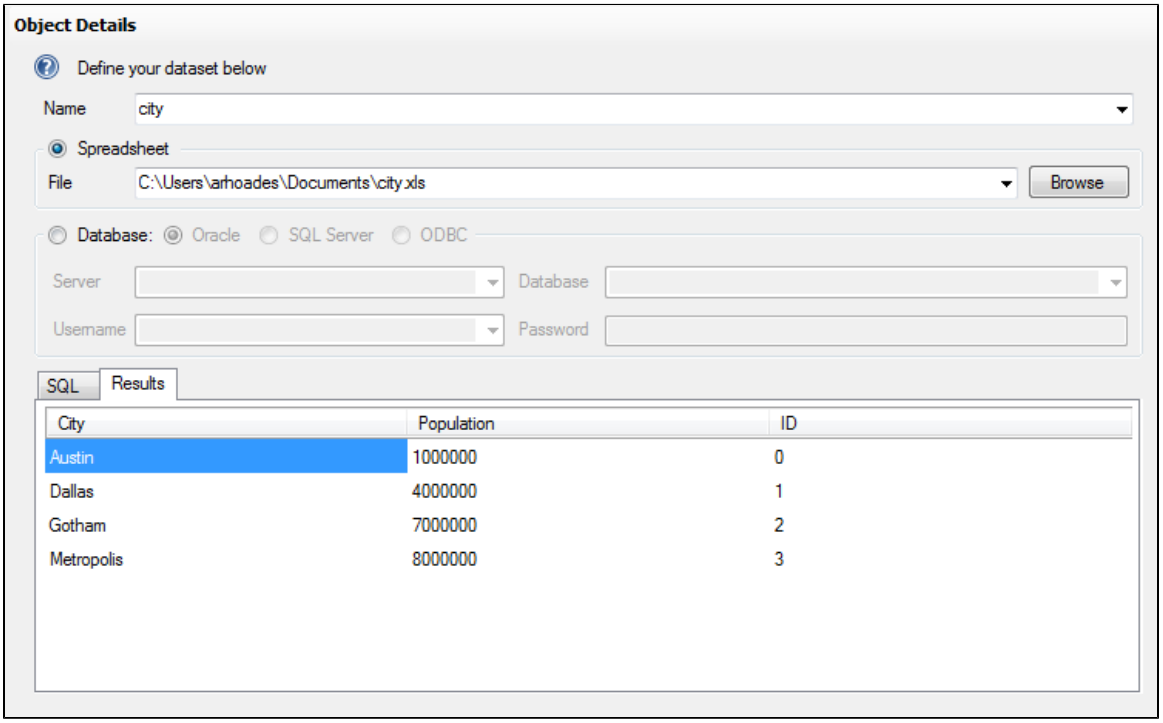



The code for a table row here is

```
<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. First create a Web 2.0 dataset using a script filter.



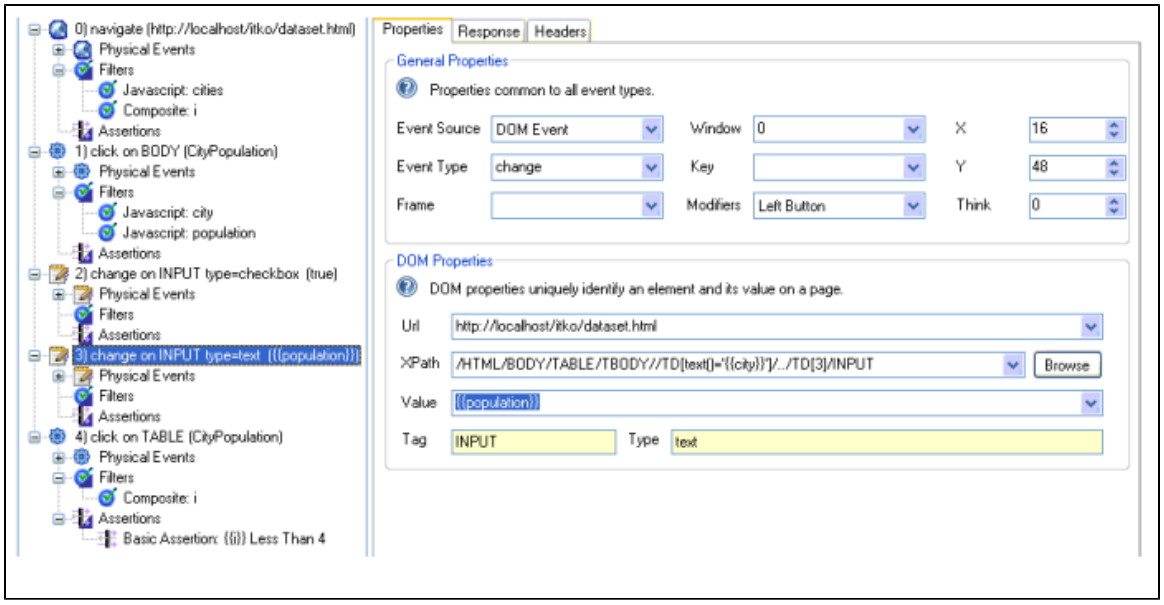

The process is very similar to the one we just followed: first record selecting one check box and entering a value in the corresponding text field. It can be any box, the goal is only to generate two changes events, one for a check box, one for a text field.

Then create the **i** looping variable before any of the two change events and parameterize 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, define a couple of script filters named **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 two 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 parameterize the value of the second change event to use the value coming from the dataset: `population`.

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




All this can also easily be accomplished in code 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;
```

## How To Deal with Dynamic Elements

To understand this example, you must be familiar with XPath expressions, as described in Web 2.0 - XPath Syntax.

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, and others, 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.

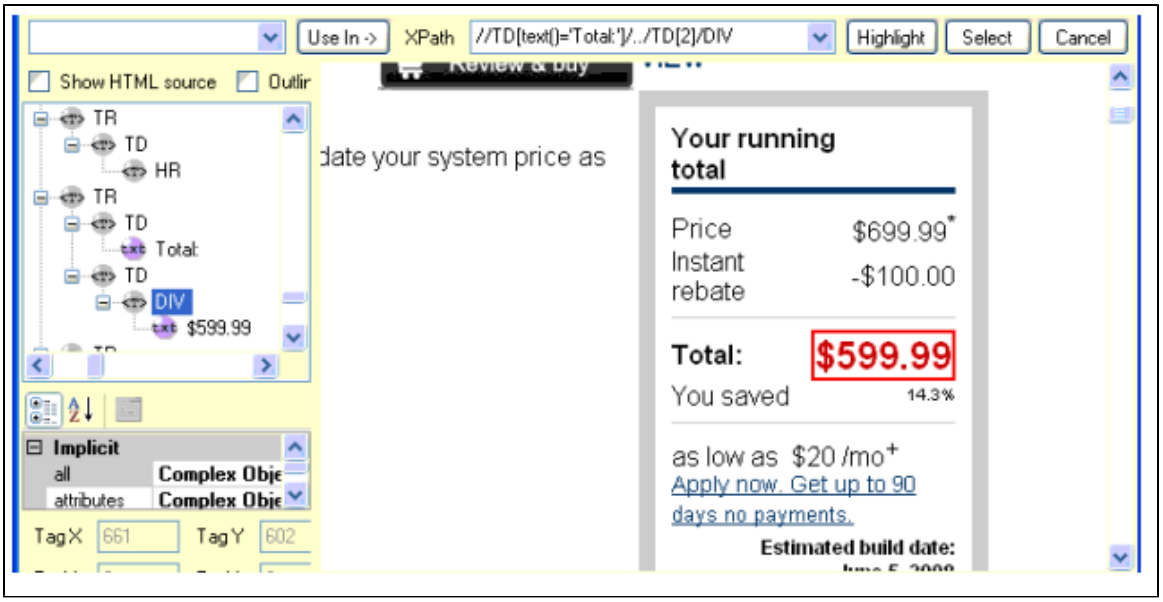
The primary means of identifying an HTML element on the page is through XPath. Those XPath expressions are automatically generated but can also be modified or parameterized to improve reliability in some cases.

In most 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 parameterize 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 three digits or more).

Generally, other typical ways to identify elements such as text or value are easily expressible as XPath too, using the syntax

```
//TAGNAME[@attributeName='value'] for example, //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 this picture, if the DIV element containing the price has a dynamic id (that is, it changes from run to run), we do not want to use it. So we browse and pick the closest element we know does not 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.

## How To Extract Complex Data from a Page

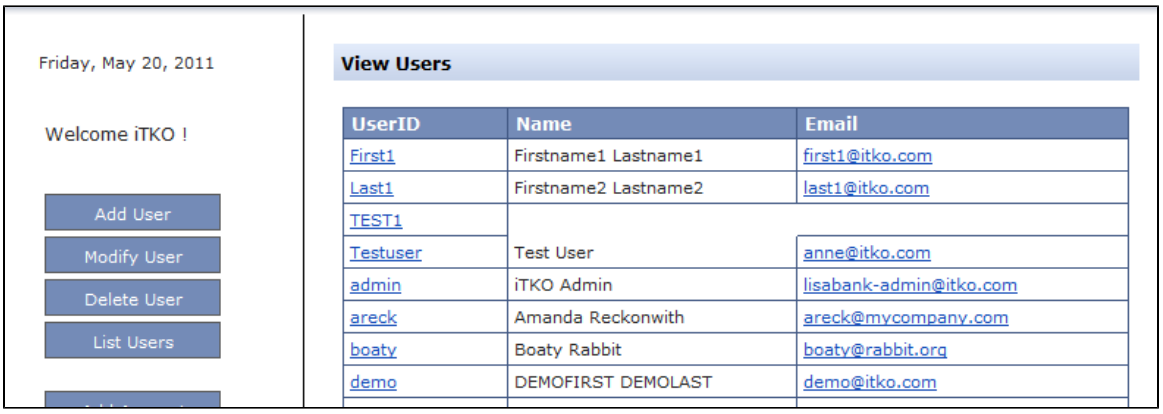
To understand this example, you must be familiar with XPath expressions, as described in Web 2.0 - XPath Syntax.

Most interesting assertions rely on data that has been extracted from pages by using filters. Almost all cases will require no configuration and will automatically pick up the data you want.

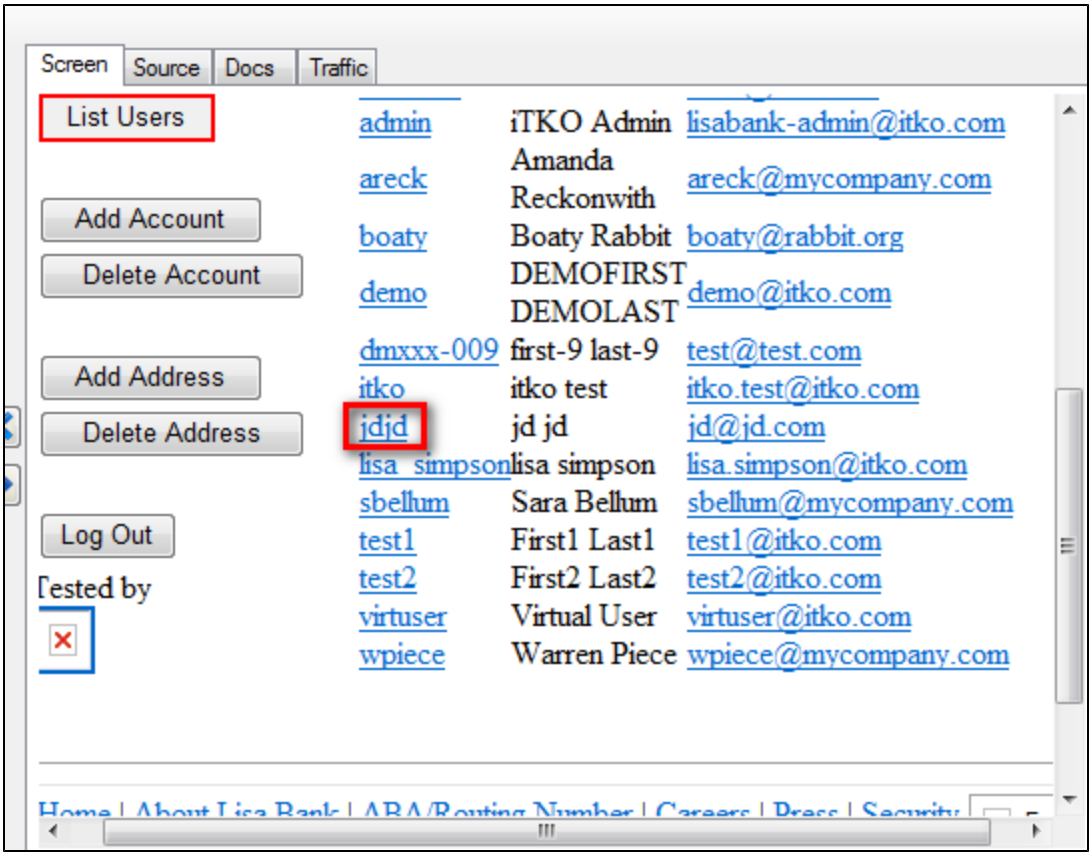
Element filters will select the chosen element (to verify it exists, for example) and fill the filter value with its XPath. The element will be already chosen for you if you use a filter added during recording (**Add Quick Filter** or **Add and Edit Filter**). If you do it post-recording, the Browse 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 pre-populated in the drop-down after an element browse is completed.

Take a look at a simple example: add a user to a list of users and it shows up in a table with user 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.




We can first add a DOM filter, and browse for the row we want in the table. We will get something like:




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

Of course, this is not reliable because the row number (493 in this instance) could change, so we need to parameterize 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']
```

The double ***//*** before the **A** specifies that 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
```

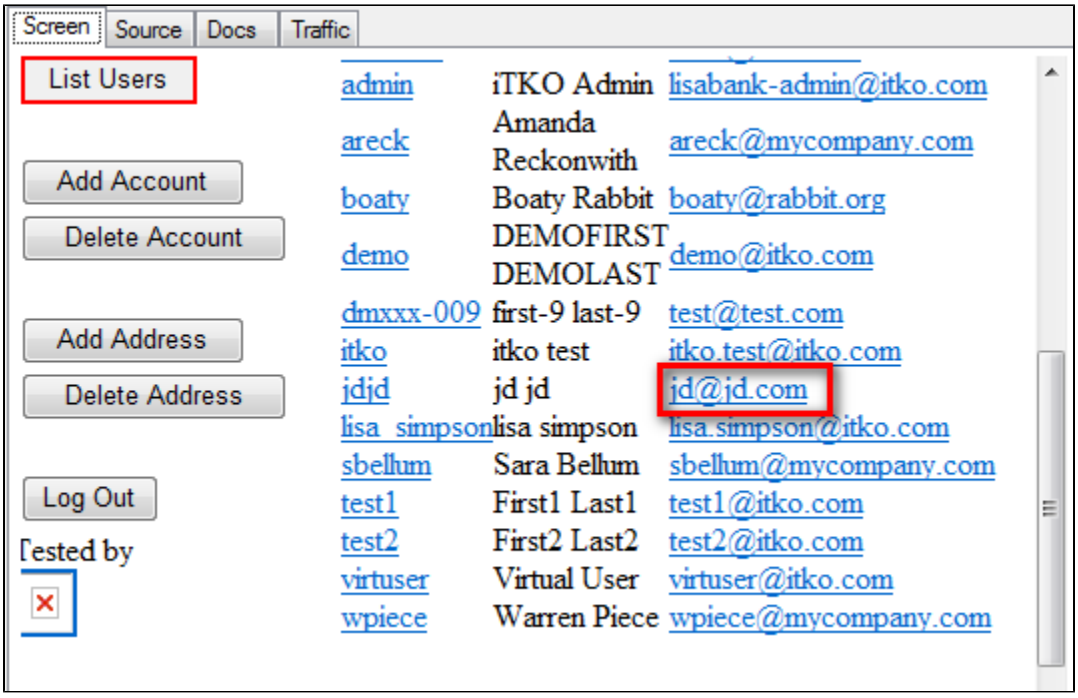
then we go up two levels to the table row:

```
//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 are selecting the right element.

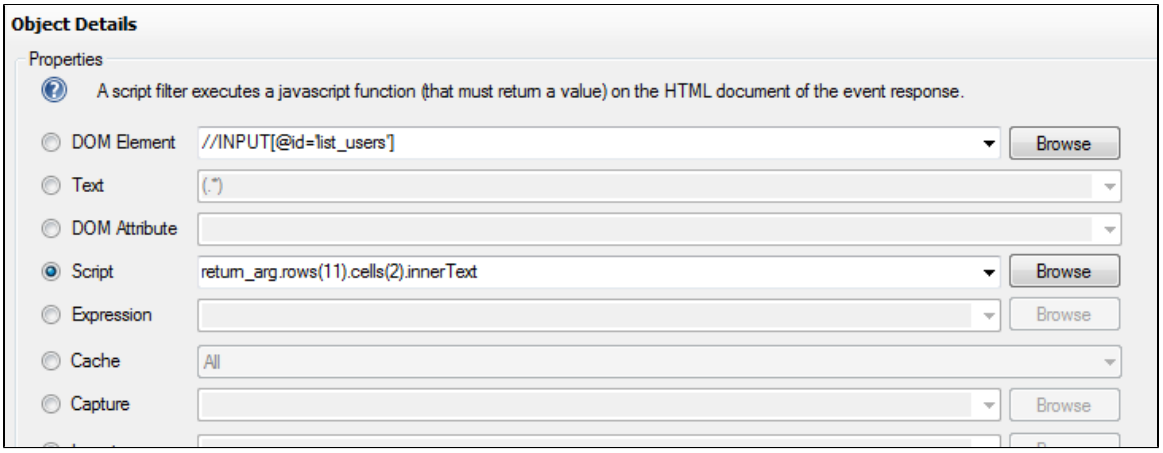



Now we add an assertion to make sure the value of this filter is indeed the email we think. Of course, all these values, such as user name, email, and so on, can be parameterized in the usual way {{ username }}, {{ email }}, and so on.

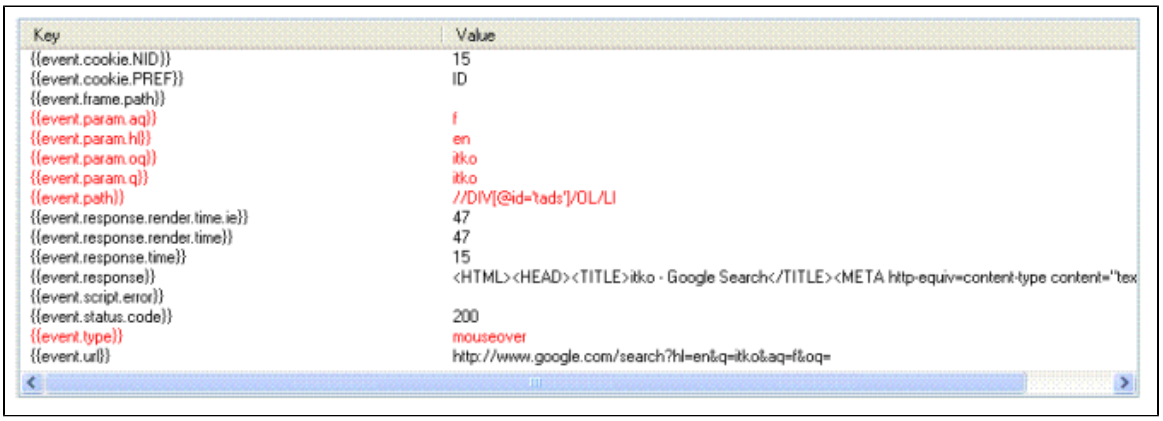
For the cases when you need finer-grained control over the retrieved data, you can use the Script Filters. They give you full control over the DOM (or the applet hierarchy when testing applets). As an illustration, reuse the previous example, where we want to extract data from the table.

The first thing you 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 Web 2.0 - Scripting Objects scripting object). Then you can easily retrieve all kinds of information.

- **Number of rows**: return _arg.rows.length
- **Text of the 3rd cell of the 12th row**: return _arg.rows(11).cells(2).innerText
- **Text of the last row**: return _arg.rows(_arg.rows.length - 1).innerText
- **Email of the user jdjd**: 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;



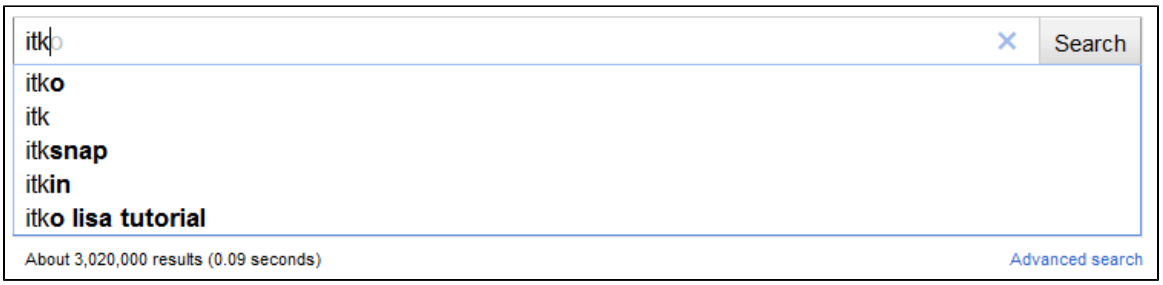

Some dynamic data is automatically extracted for you, most notably request string parameters and cookies.




String parameter names are prefixed with **event.param** and cookie names are prefixed with **event.cookie**.

## 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 as they are typing letters in a field.

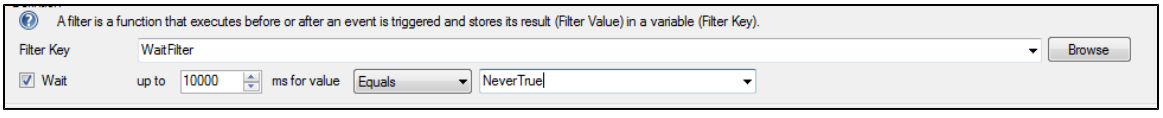



To record and replay this type of interaction, there are a couple of things to be aware of.

First, enable the keypress event recording (the default) in the settings. In most cases you can disable it and it leads to leaner test cases, but in this case it is needed.

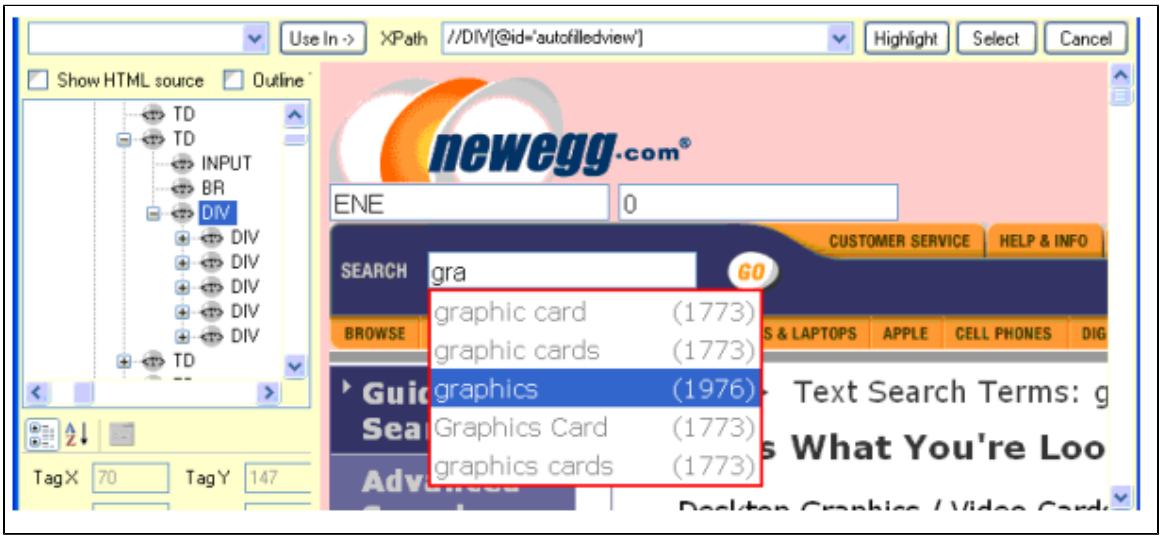
Then, when you replay the test, the drop-down is populated asynchronously, so if you take no precautions, you can execute the next event or a filter, or an assertion before the drop-down is populated. There are three ways to deal with that, as explained in the How To - Deal with time-sensitive events section. The first way is by using the **Synchronize Ajax calls** setting.

The second way 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.



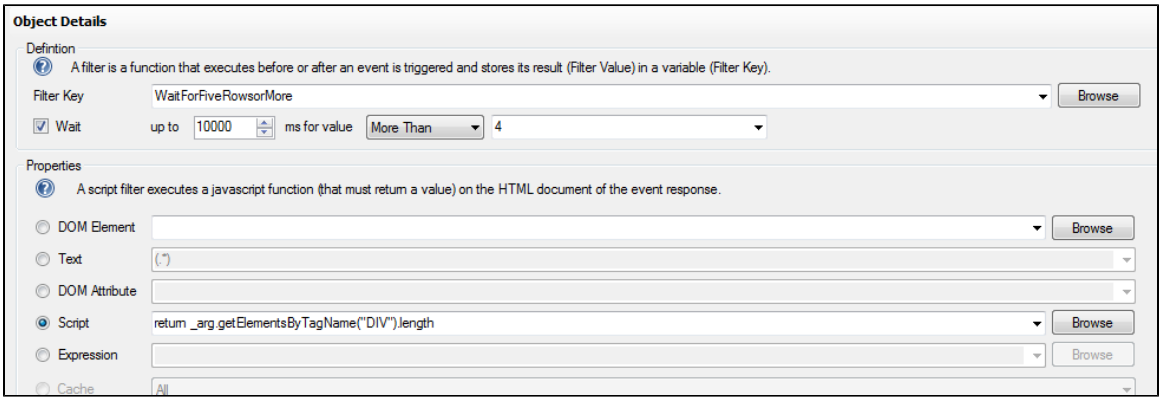
The last way, which will always work but requires a bit more work, is add a filter that waits for a specific condition to be fulfilled. In this case, the condition would be that the drop-down is visible and shows a certain number of rows.

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 dialog, as illustrated in the following picture (from the search on **www.newegg.com**).



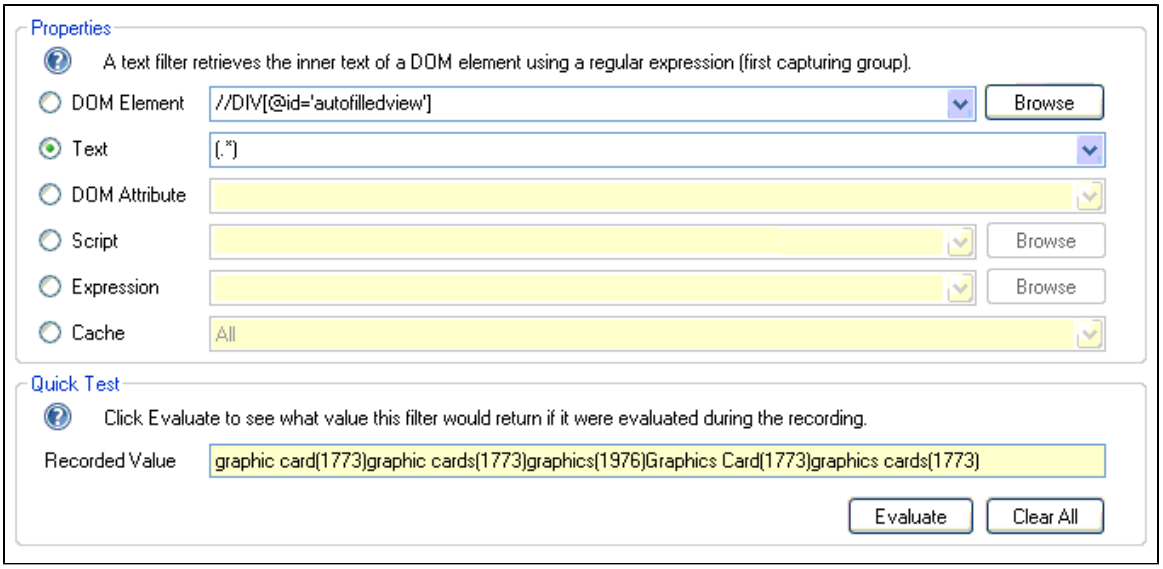

Specify the script snippet to return something meaningful for this site. In this case, you can see from the previous illustration that the drop-down is a div tag and each row is made up of a child div tag (easy to see from the DOM tree). So a meaningful filter script might return the number of rows.

Finally, because 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:




Use the Evaluate button to tell you if the script you are using is correct and returns the expected result. To see the drop-down visually in the Browse window, as pictured previously, you need to enable the **Capture HTML changes** setting in the recording settings as explained in the How To - Capture Dynamic HTML for later test editing section.

You are now ready to add more filters for data extraction and assertions for validations. This is more thoroughly covered in the How To - Extract complex data from a page 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 pop-up.

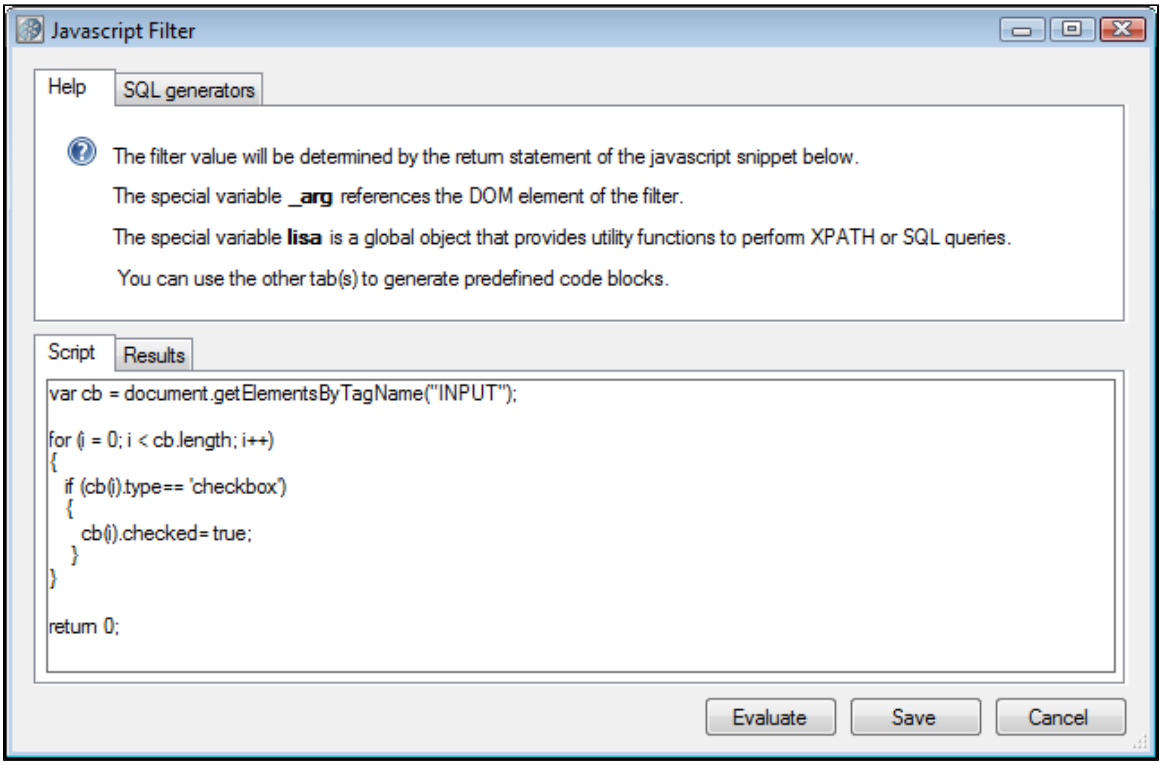



Then add an assertion that checks that some given text is contained in the value of this filter (using a Basic Match assertion).

## 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 LISA Web 2.0 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 check boxes that you want to select , the GUI way to do this would be to select one, and then to parameterize it so that it can be embedded in a loop that will visit all the check boxes with the change event (see How To - Parameterize dynamic data entry in loops). If you have two levels of loops or other constraints it will be quite tedious, but it would be easy to do in a script.



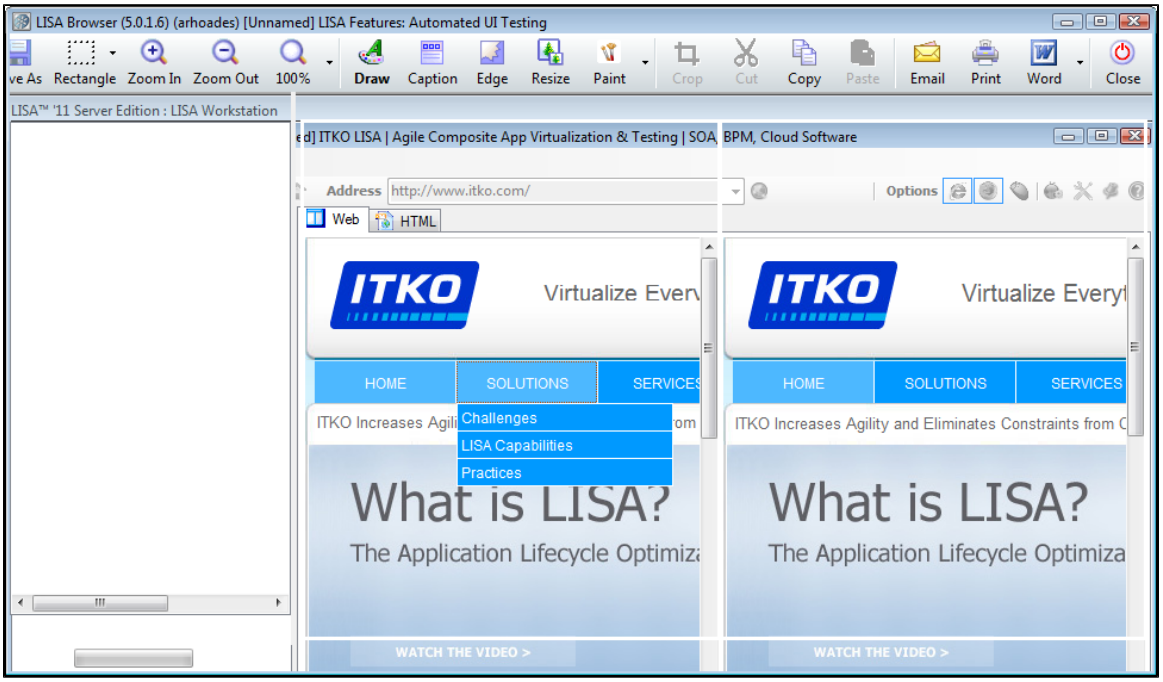

When you create a new step in the Events tab, a new event of type script gets created. It does nothing by itself but contains a script filter that can execute arbitrary JavaScript (or Java for applets). The previous image shows the script dialog that gets opened when you click the Browse button of a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

## 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 author, record and edit the test

in the exact same way you usually 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 following illustration, both 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 to run a test.



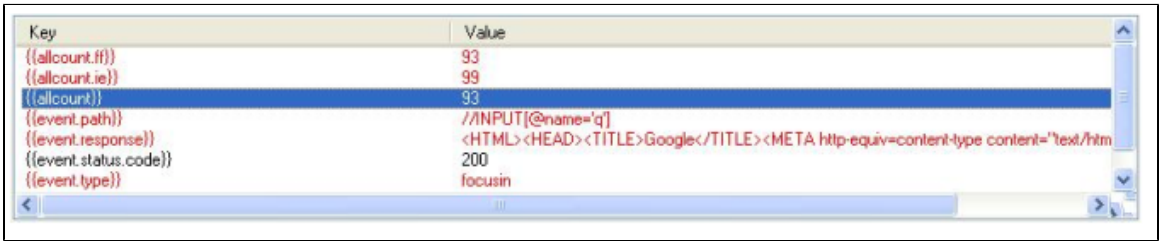

While Internet Explorer is always installed on Windows, other browsers require an initial download. Every time you attempt to use Firefox, if it has not been installed you will get prompts to download it.

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 because they do not depend on the browser; the only thing to pay attention to is filters because they highly depend on the browser.

First, 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 another one because they have the same key.

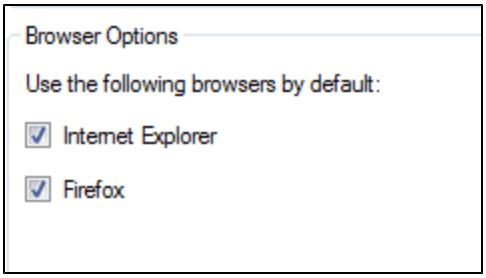
This is why when multiple browsers are used, the keys used in filters are still in use but there are automatically new filter keys being generated, one for each browser, to let us distinguish between the filter values coming from different browsers. The convention is to append two letters to the filter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

The following screenshot shows what happens when we define a JavaScript filter named allcount with code: "return document.all.length;".



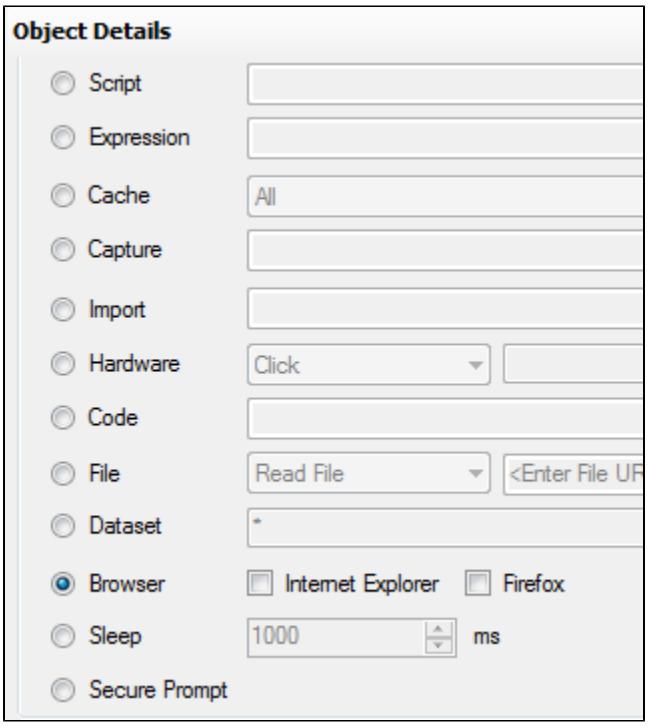

Because the different browser evaluates this JavaScript expression differently you see three values for the allcount variable: one for Internet Explorer, one for Firefox, and one default one (Internet Explorer in this instance) so as to not break assertions written for single browser mode. Now we can 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.

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 three ways do do that, one for each level of granularity. To control globally which browser(s) is (are) used by default, use the **Default Browser Mode** section in the settings.




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).

At the event level, which browser is being used is controlled by the browser filter.

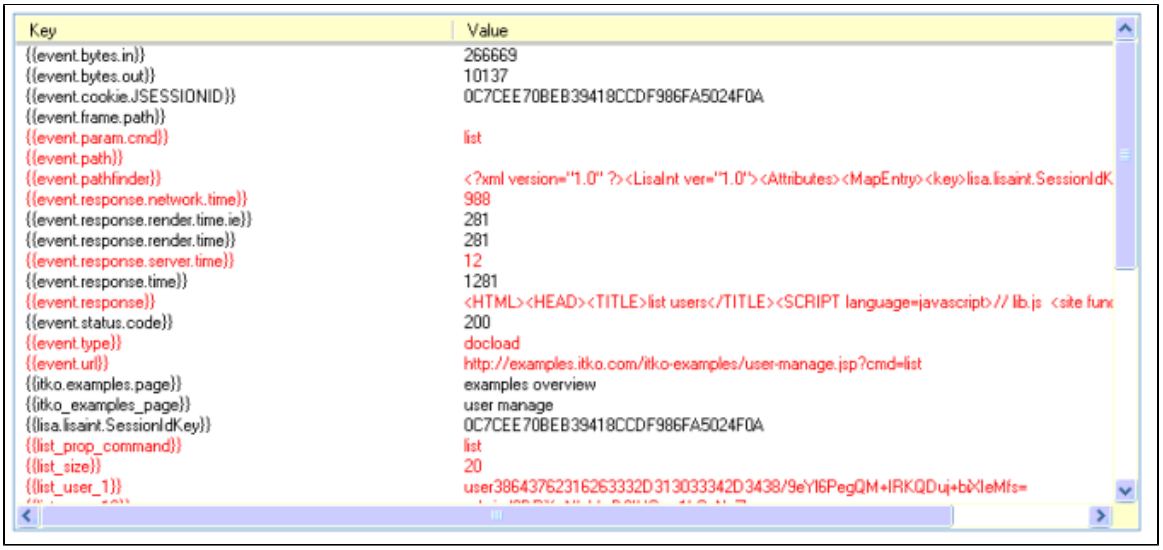



When a browser filter executes, it selects the browsers specified in the filter to run from that point on. Typically you would use this filter just before looping in a test case, so you could run a step of steps in a given browser and then again the same set of steps in an other browser to compare the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

## 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 selecting the **Enable Pathfinder Integration** check box. This will automatically turn on the **Capture HTTP traffic** option also.

The main effect of turning on this option is that several new variables will be automatically made available.



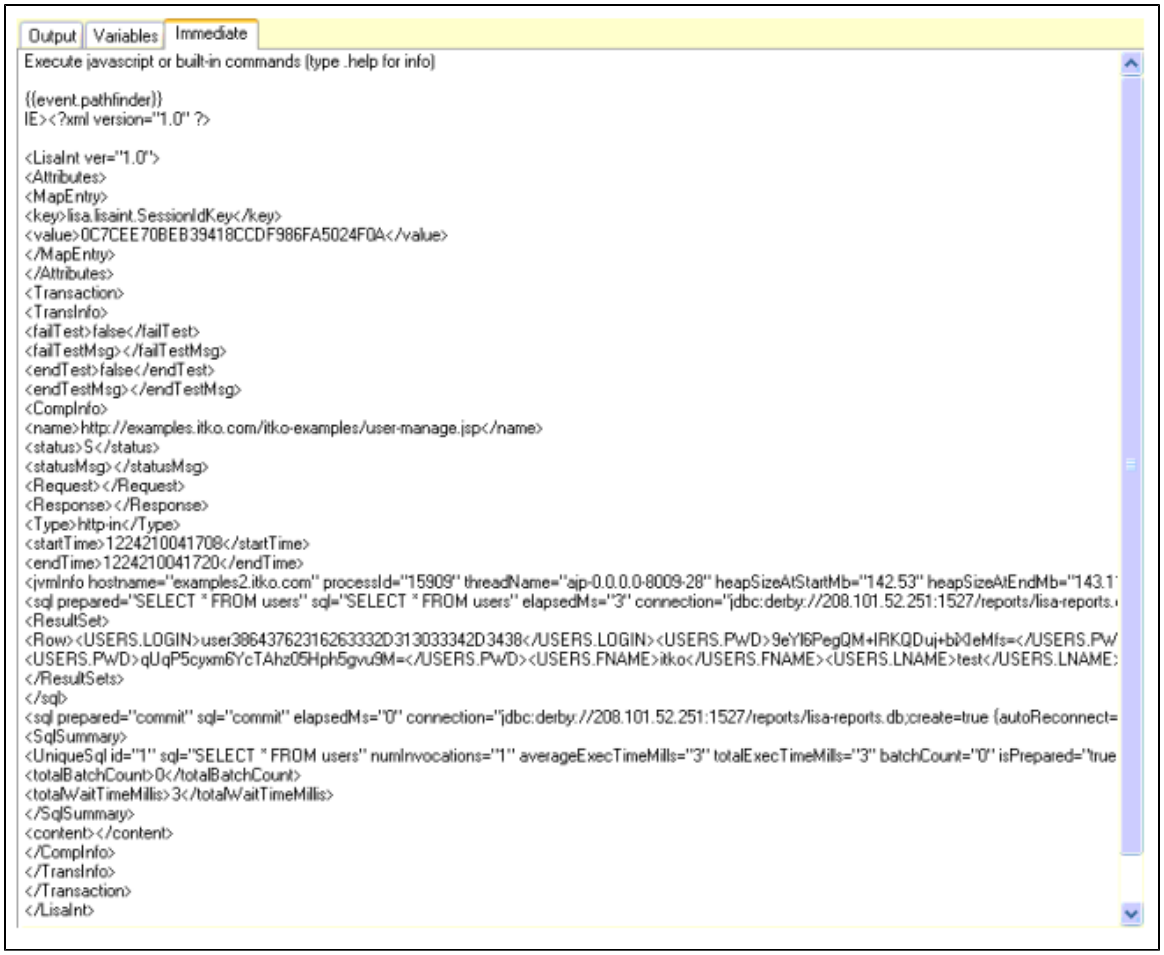
| Key | Value |
|---|---|
| {{event.bytes.in}} | 266669 |
| {{event.bytes.out}} | 10137 |
| {{event.cookie.JSESSIONID}} | 0C7CEE70BEB39418CCDF986FA5024F0A |
| {{event.frame.path}} | |
| {{event.param.cmd}} | list |
| {{event.path}} | |
| {{event.pathfinder}} | <?xml version="1.0" ?><LisaInt ver="1.0"><Attributes><MapEntry><key>lisa.lisaint.SessionIdK |
| {{event.response.network.time}} | 988 |
| {{event.response.render.time.ie}} | 281 |
| {{event.response.render.time}} | 281 |
| {{event.response.server.time}} | 12 |
| {{event.response.time}} | 1281 |
| {{event.response}} | <HTML><HEAD><TITLE>list users</TITLE><SCRIPT language=javascript>// lib.js <site func |
| {{event.status.code}} | 200 |
| {{event.type}} | docload |
| {{event.url}} | http://examples.itko.com/itko-examples/user-manage.jsp?cmd=list |
| {{itko.examples.page}} | examples overview |
| {{itko_examples_page}} | user manage |
| {{lisa.lisaint.SessionIdKey}} | 0C7CEE70BEB39418CCDF986FA5024F0A |
| {{list_prop_command}} | list |
| {{list_size}} | 20 |
| {{list_user_1}} | user38643762316263332D313033342D3438/9eYl6PegQM+IRKQDuj+bXleMfs= |

In addition to the variables listed in the User Guides Web 2.0_Filters, 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 }}**: Custom variables defined by Pathfinder to represent some variable value on the server and made available for inspection on the client.

If those variables are exposed by the SDK they will be available automatically even without Pathfinder integration.

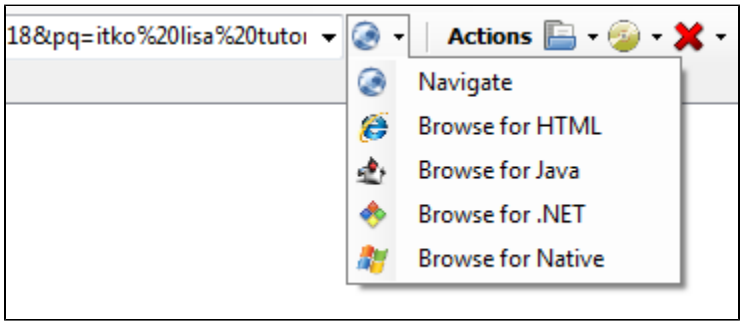
- **{{ event.pathfinder }}**: Is the most important variable. It is an XML representation of the Pathfinder payload and contains all Pathfinder information, that can then be extracted using XPath expressions, with the Web 2.0 Scripting Objects API.



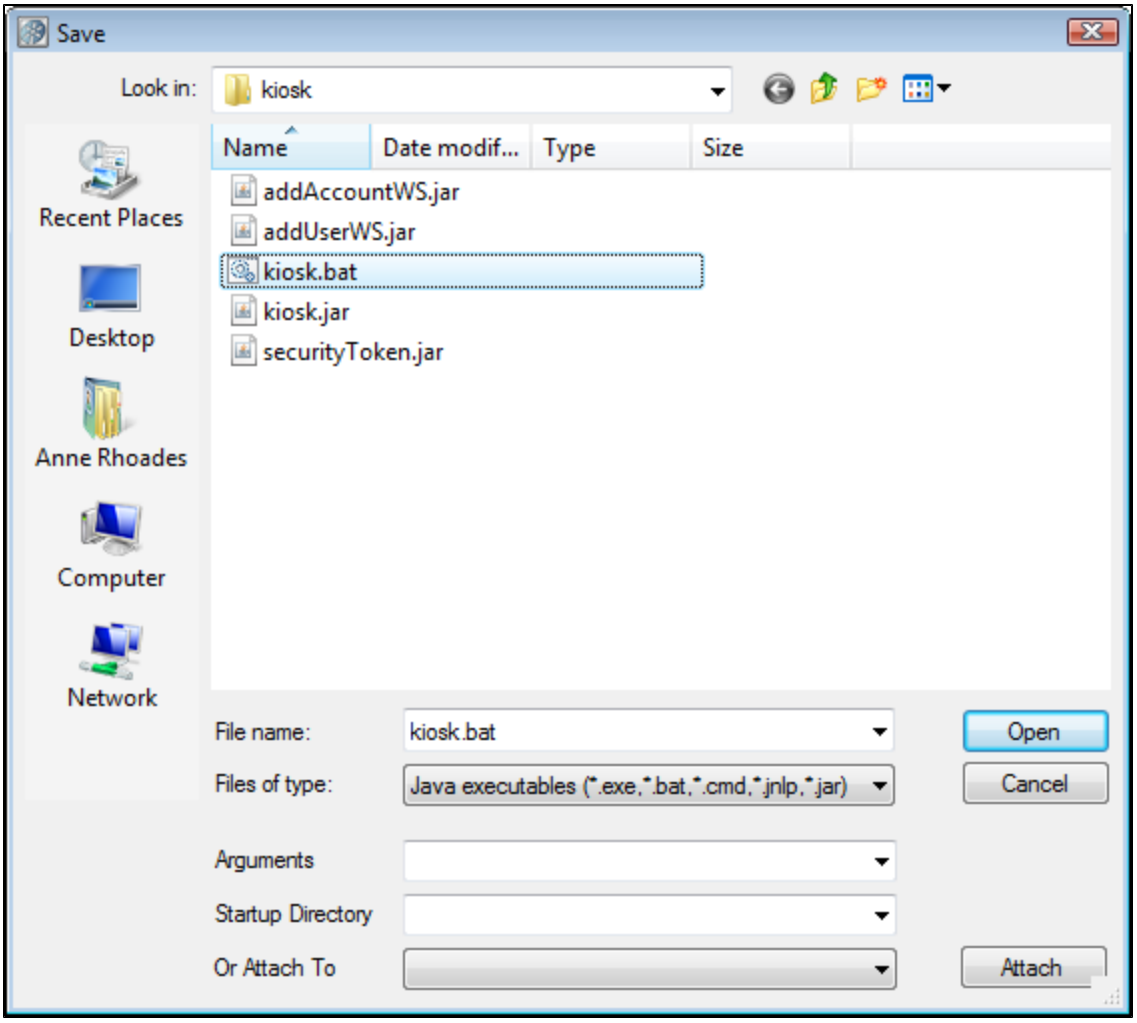

For instance, to get the query string out of the first SQL statement in the payload pictured previously, you can use return lisa.pathfind('//sql/attribute::sql'), which will return SELECT * FROM users.

## How To Write Java Swing and WebStart Tests

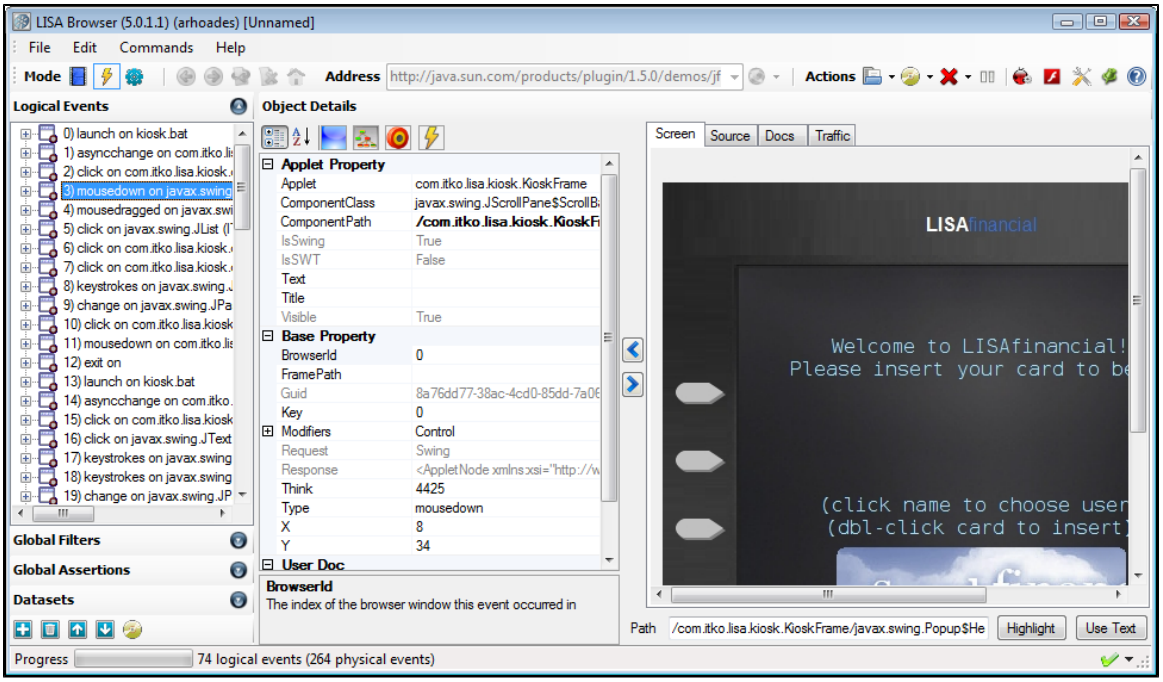
Authoring Swing tests is practically no different from writing applet tests. From a GUI perspective, the only difference is how you launch the application.




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, virtual machine arguments, and so forth) in the **Open Dialog** window, and the start directory (optionally). In other cases the application is launched from a pre-packaged executable or from a batch script. This is supported too and makes no difference.



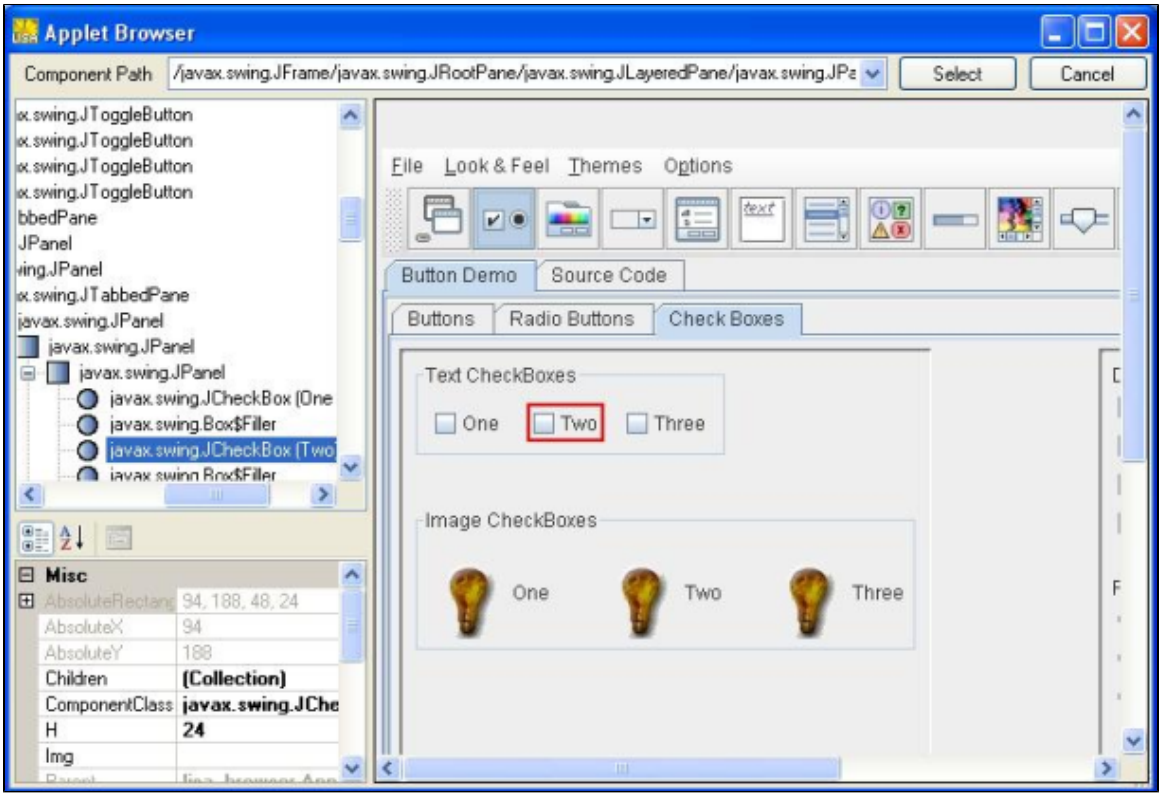

Following is a screenshot of a few events recorded against a Swing application; you will notice it looks almost identical to Applet events, except for the Applet field, which is now a JFrame.



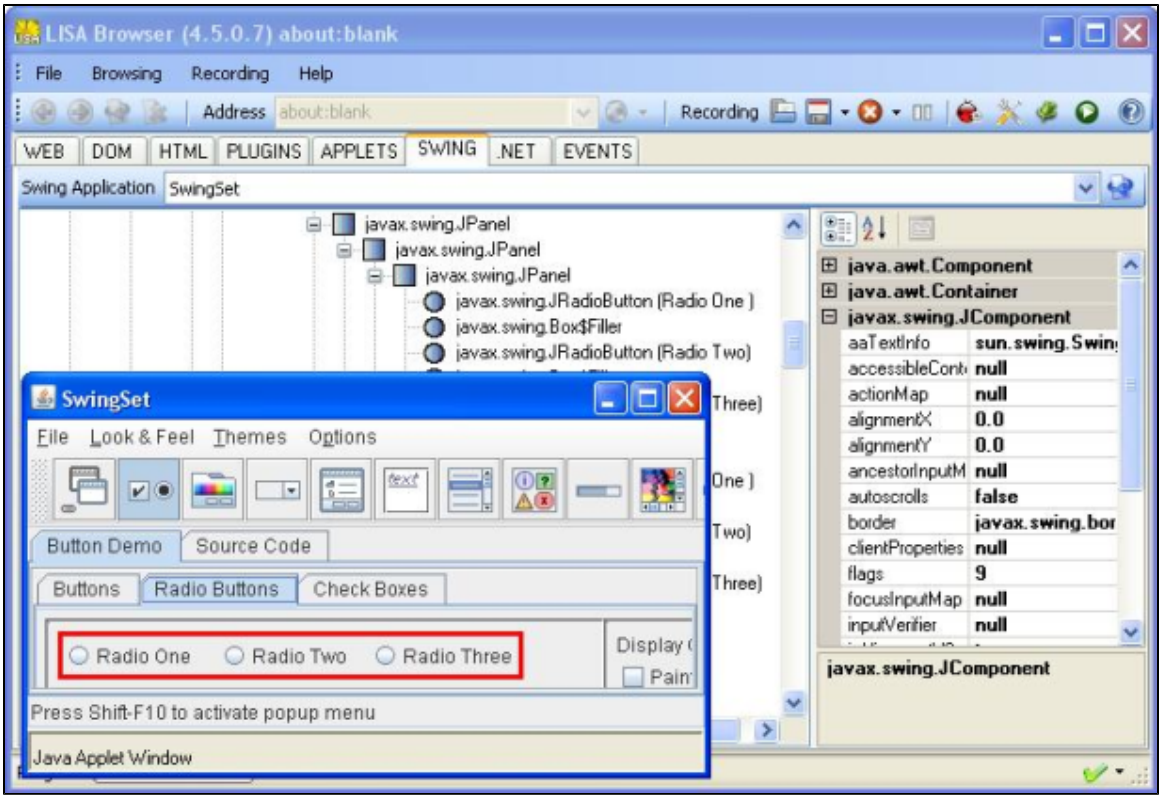

You can similarly browse for offline editing and bring up the Applet/Swing browser to change and/or parameterize event targets. The red highlight

rectangle in the following illustration is part of the application, not a screenshot artifact.

NEED UPDATED SCREEN CAPTURE



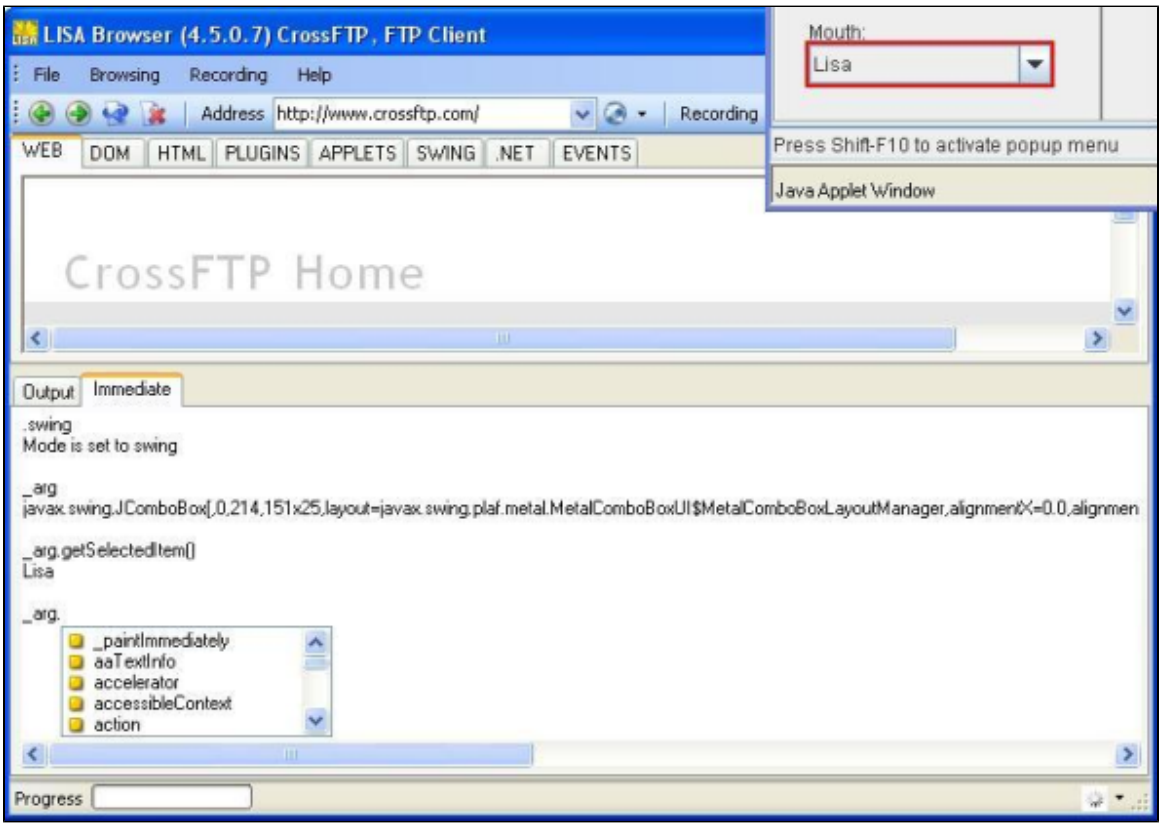

Finally, Swing applications also have their own live instance browser under the SWING tab (in both recorder and playback modes). When selecting a component in the swing hierarchy, this component will get highlighted in the live application so you can easily verify which elements you are targeting. The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




The 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 can simply select the table as the component and then write a script like:

```
return _arg.getModel().getValueAt(2,3);
```

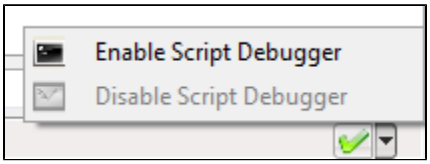
While this step's primary purpose is to do Swing or AWT testing, there is nothing that prevents you from running against headless or console Java applications. You can run "in-container" tests by specifying any scripts you want in JavaScript nodes.

## How To Debug a Test

Unexpected behavior can occur during recording or replay, and in both cases the best approach is to debug the test directly in the DOM browser, because 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 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 Debug  icon in the toolbar.

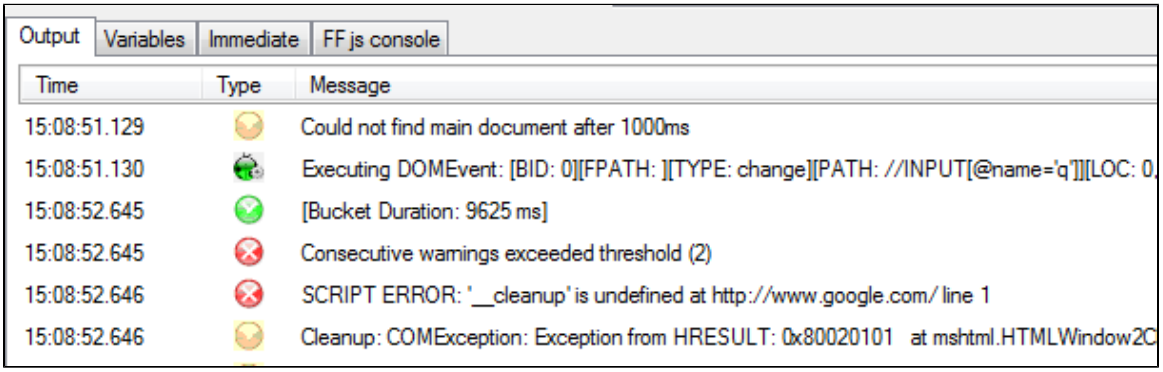
You can also toggle it by clicking the status icon in the bottom corner of the screen.



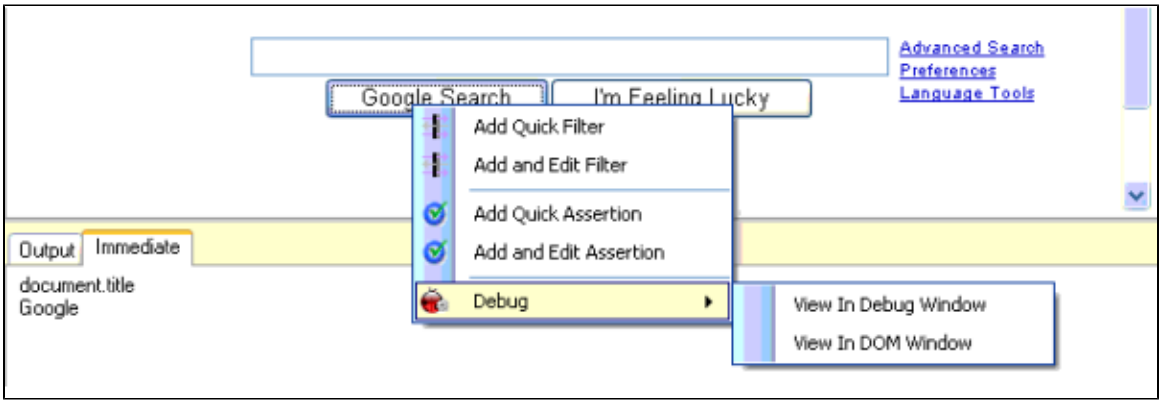

This icon has three states: a green check mark to indicate no errors, a spinning wheel to indicate a document is loading or an orange icon to indicate there have been errors: most of the time script errors, but they could also be non-critical DOM browser errors. Script errors are

sometimes difficult to debug given just the error message, hence the dialog window as seen previously to enable or disable a JavaScript debugger.

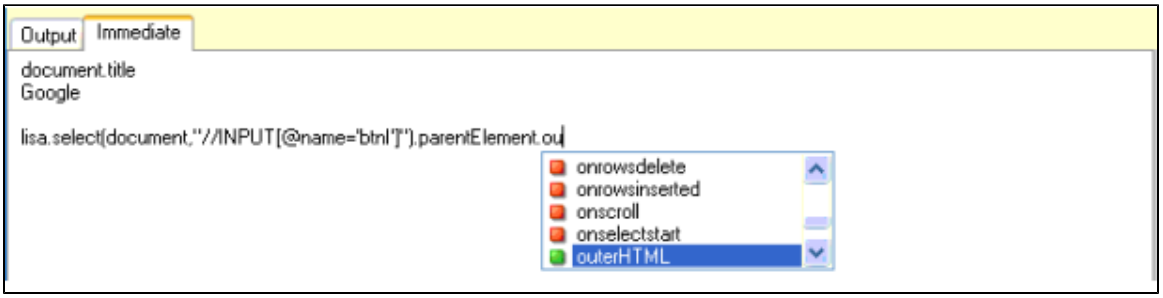
By default all debuggers are disabled to not interfere with the recording or playback, but if you enable it and there is a script error, you will see a dialog asking if you would like to debug. Clicking Yes will launch the debugger and show you all the details about the script error. Other than that, just opening the debug window will suffice in most cases.




In recording mode, two tabs are available in the debug window; the Output tab pictured previously and the Immediate tab. The Output tab 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.

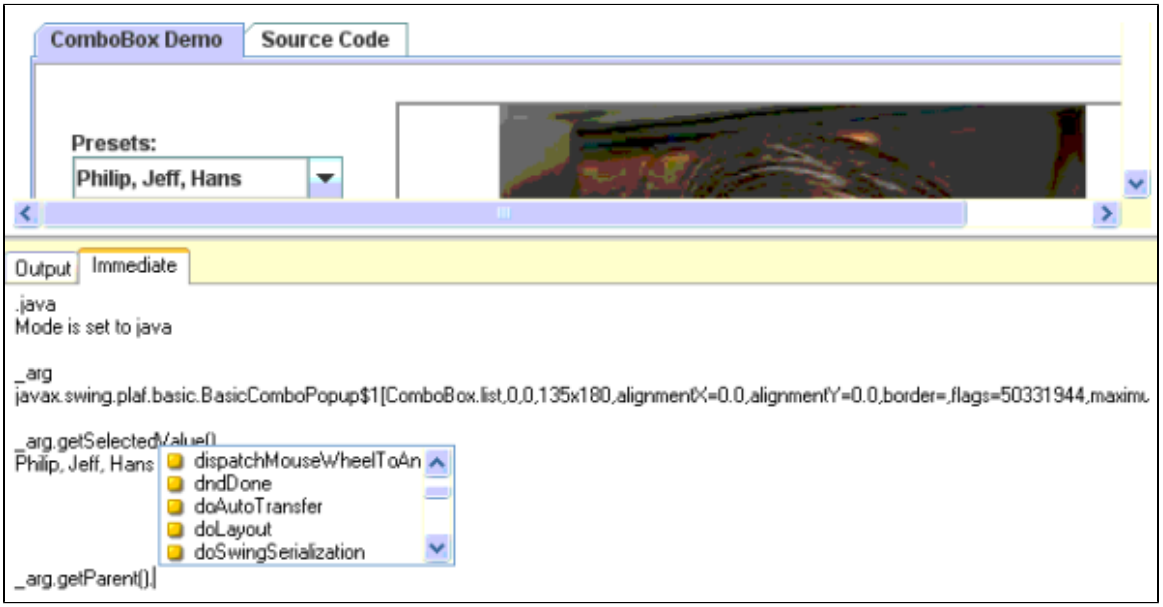



By right-clicking on an element, you get a menu that contains a **Debug** entry, to let you view the element in the DOM view or in the Immediate window. If you select Immediate, you will be able to evaluate any DOM or JavaScript expression on the element, helped by IntelliSense.



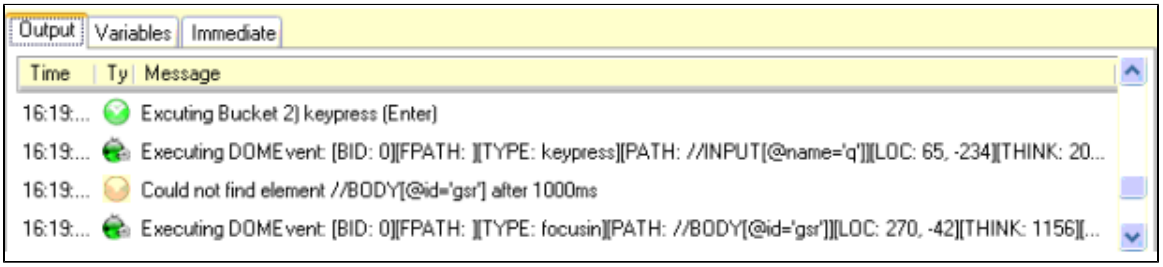

This can be helpful in writing script filters but also to inspect JavaScript event handlers. All built-in commands can be accessed by typing a dot (.) as the first character of the line. You can display what they do with the .help command.


For instance, there is a .js and a .java entries that lets you 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.



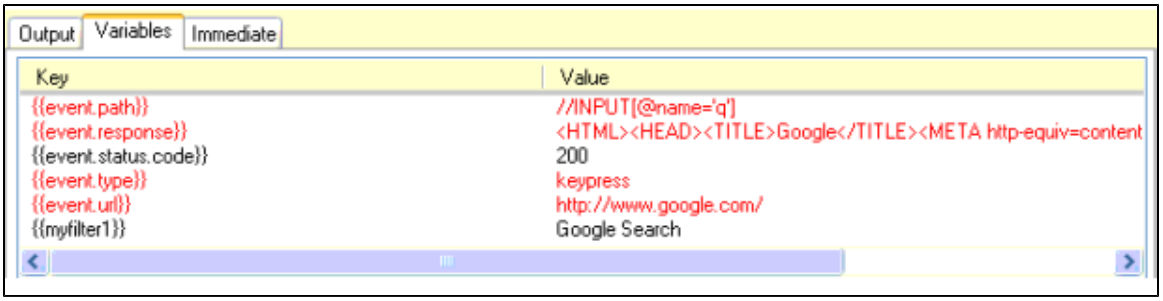

All these JavaScript and Java (BeanShell to be exact) expressions can be used in filters so this is a useful way to design your filters during recording.

Most of the debugging will probably take place during playback because 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 because of the failure of identifying an element on the page or in a control (as seen in this illustration).

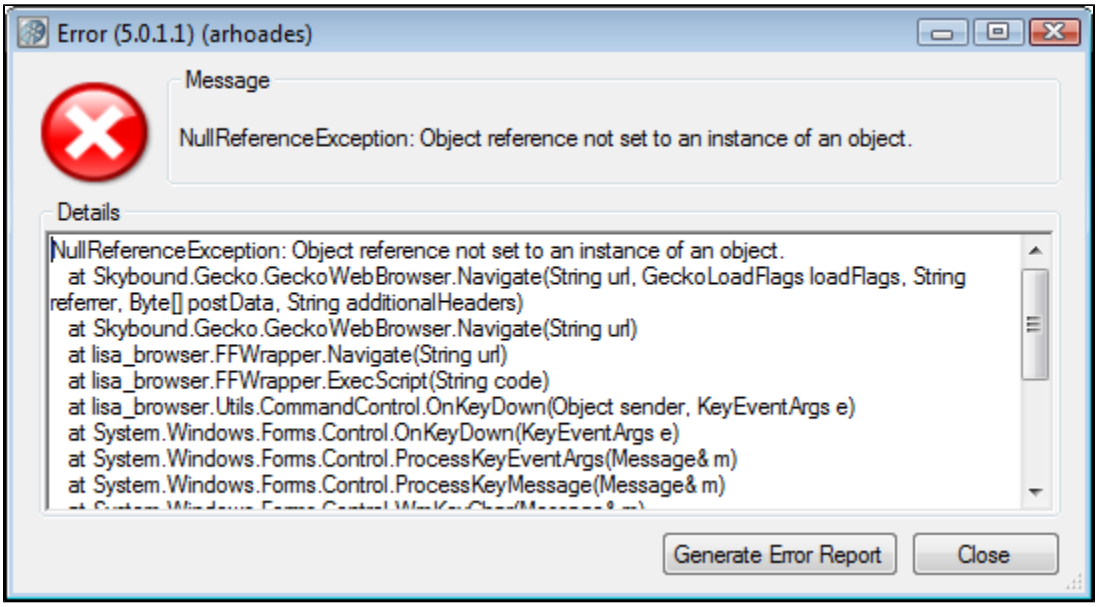



There could be several reasons for this: a genuine bug in the system under test or a failure to adequately parameterize a dynamic element. You will also see the filters and assertions being executed and their values, so you can quickly see why an assertion fired, for example.

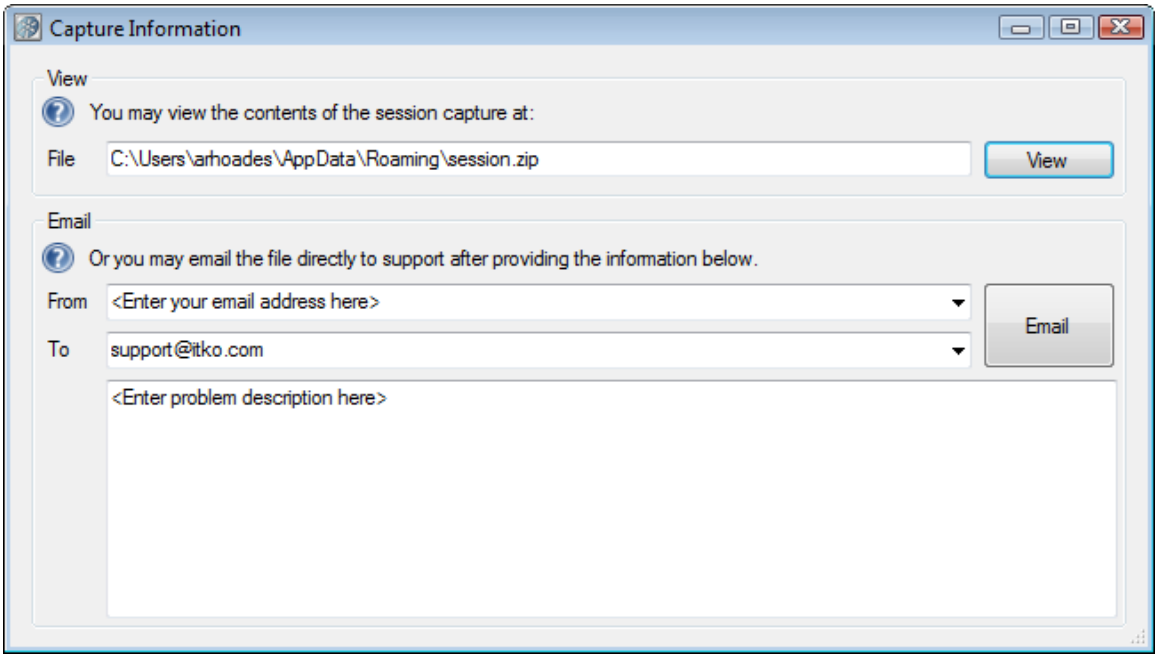
The Immediate tab behaves the same way as in recording mode. The Variables tab will list all the variable values for every step (highlighting in red the ones that were just modified). This is particularly useful when you set breakpoints or step through a test as in a debugger, you can observe the value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




Severe errors will not be caught and logged in the Output tab of the debug window, but instead will generate a dialog.




In most cases (especially during recording) you can close the dialog and proceed but not in all cases. And because 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 you with the following dialog.

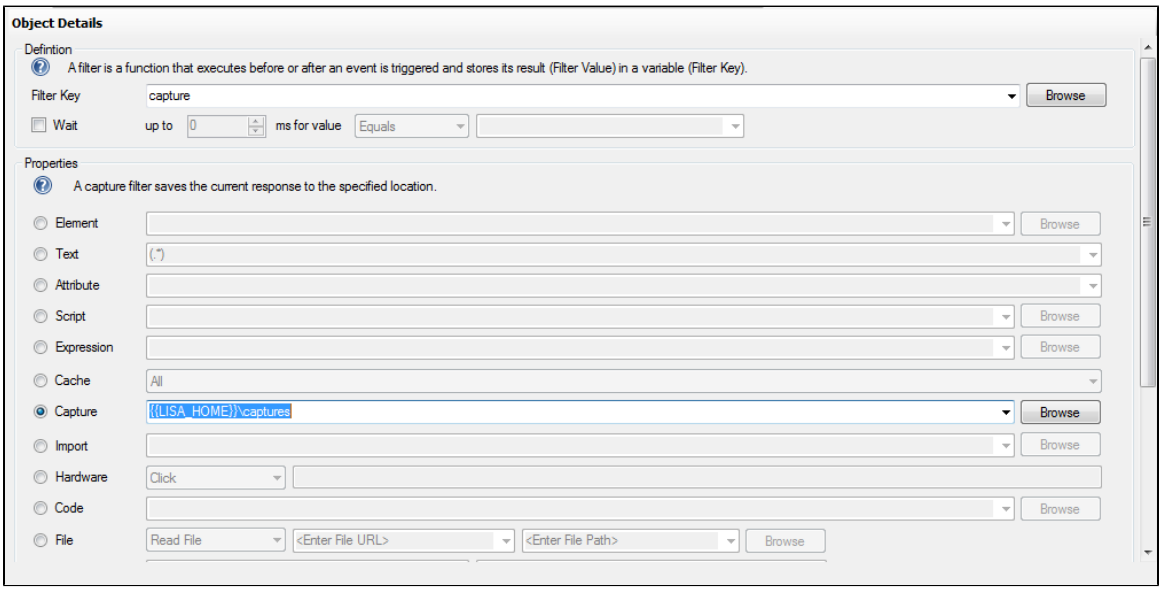



You can open the zip archive by clicking View and then email it to support by clicking Email after filling the required information for troubleshooting. If there is an SMTP server running on the computer (if IIS is installed for example), the email will be sent automatically; otherwise the default email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

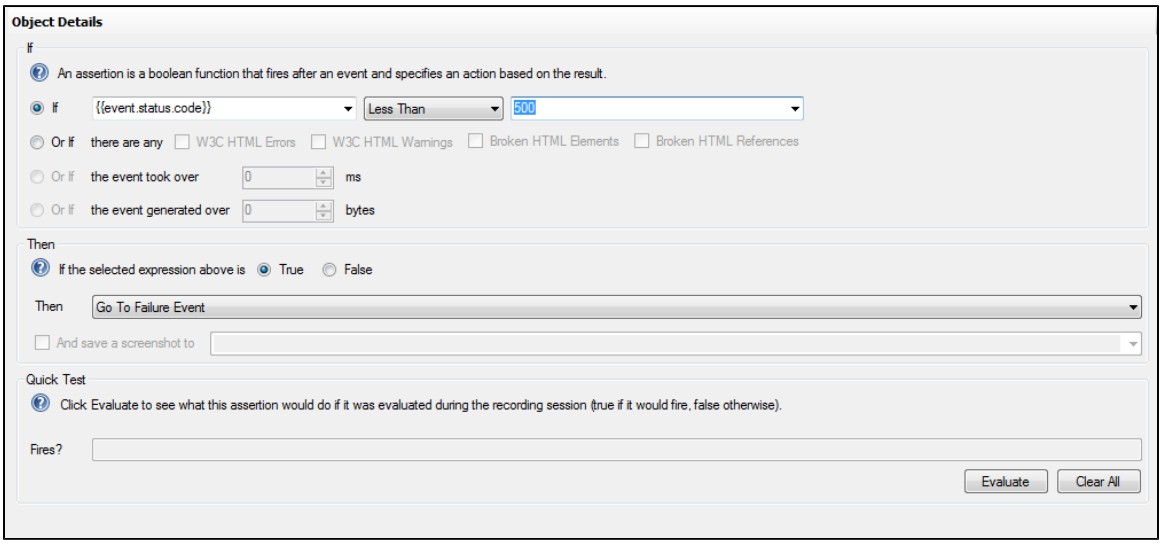
# How To Use Global Filters and Global Assertions

The Web 2.0 browser provides the ability to execute global filters and global assertions, which are filters and assertions that execute on every step. If you have repetitive actions you must take, rather than defining those manually on every step, you can 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. Go to the **Events** tab 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 in this screenshot.

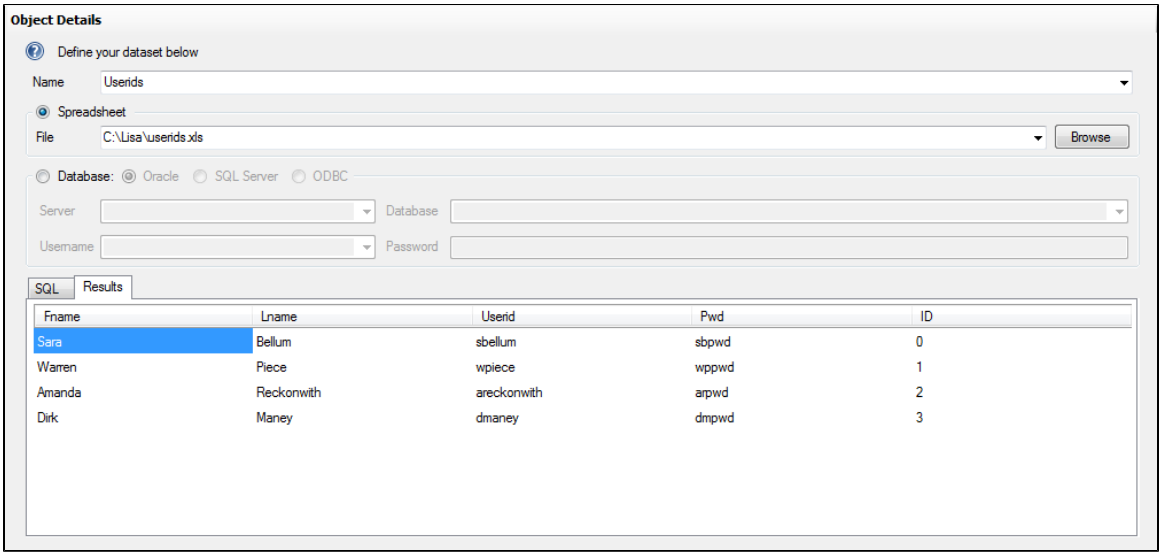



This verifies after each step that the server did not return an internal error code.

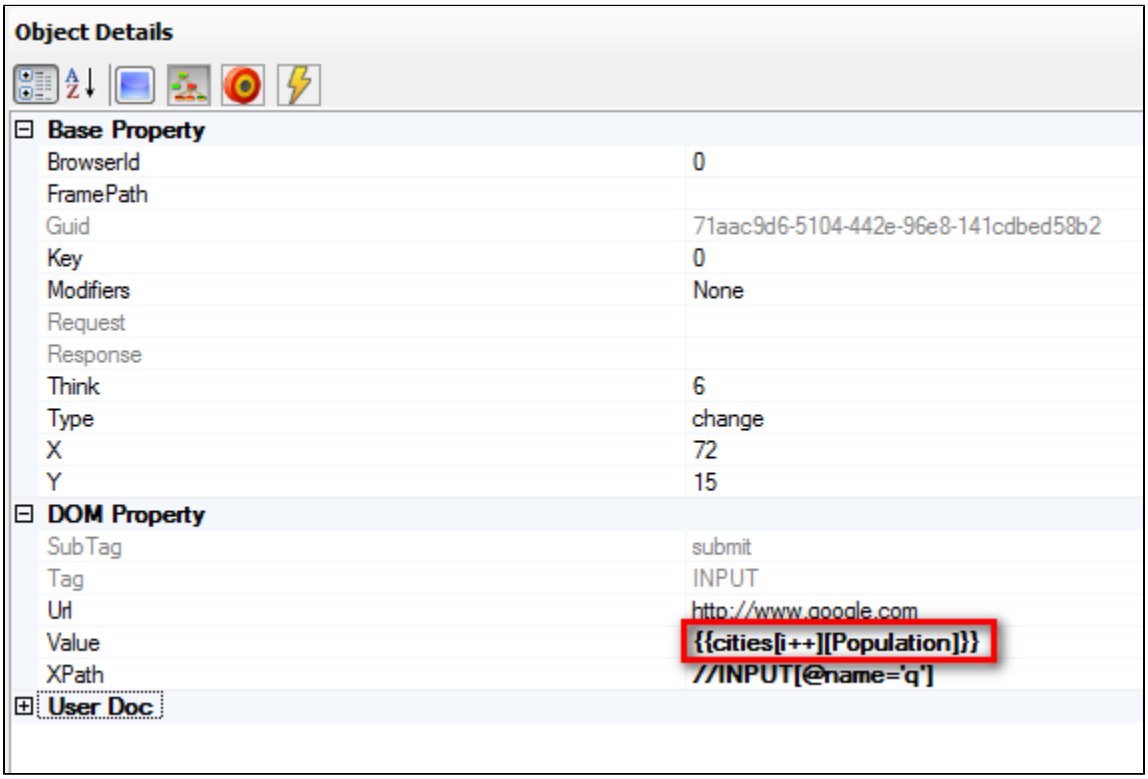
# How To Interact with External Resources

There are many ways to interact with external resources: flat files, Excel spreadsheets, databases, and more, 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 will not execute and the external data will not be available. That is why there are ways to access external data from the DOM browser too.

First, there is the concept of a Web 2.0 dataset. Those are defined directly in the browser and do not depend on LISA Workstation. To define them, go to the **Events** tab and select the **Datasets** panel, then **Add** a dataset. The details panel lets you specify its name, its data source and makes it easy to test.




After 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]`**.




This lets you 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 target row. However, if you want to automatically advance or go back, the syntax also supports the operators **++** and **--** as pictured previously.

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, one-letter variables used as integers like **i** will automatically be defined and set to 0 if they have not been defined before.

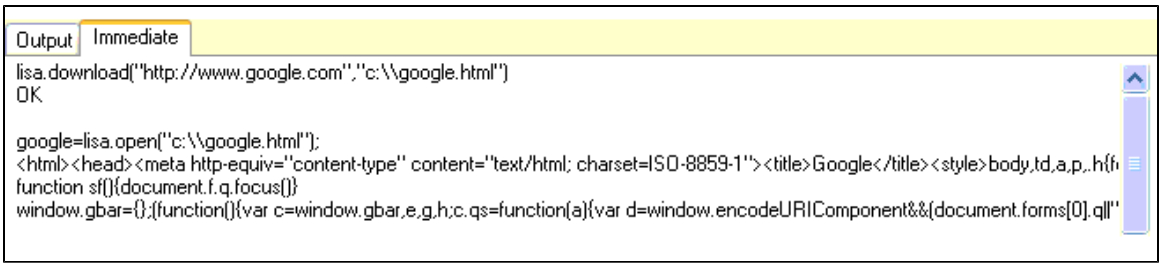
The syntax `cities.length` or `cities.count` can be used to determine the number of rows in the cities datasets. This is useful to write exit condition assertions when iterating over a dataset (for example: If `i` More Than `cities.length` Then Go To event xyz).

In addition you can use the global Web 2.0 Scripting Objects scripting object. Of interest here are the following methods: download, open, fileQuery, dbQuery.

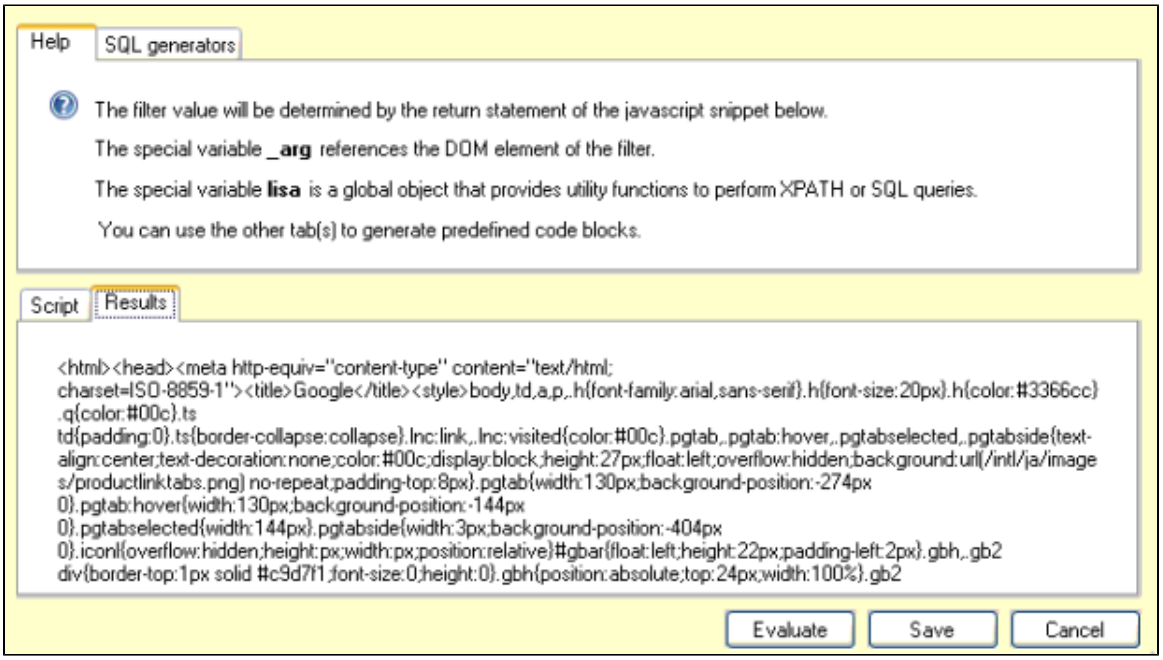
- **download**: Lets you download a resource from a given URL to the specified path, so you can interact with it.
- **open**: Reads the contents of the specified file and returns it as a string.
- **fileQuery**: Executes a SQL query against an Excel file and returns a JavaScript dataset.

- **dbQuery**: Executes a SQL query against a database file and returns a JavaScript dataset.

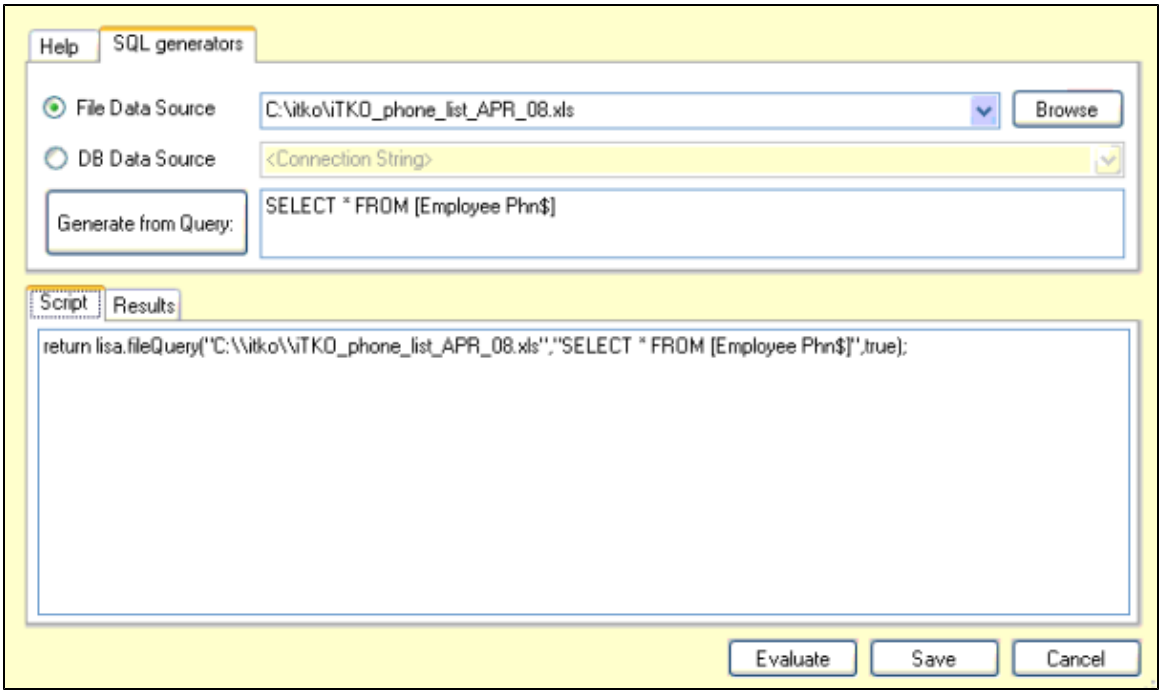
All of these functions can be used from the **Immediate** debug window or script steps or filters. For example, here is how you would use **download** and **open** from the Immediate window:



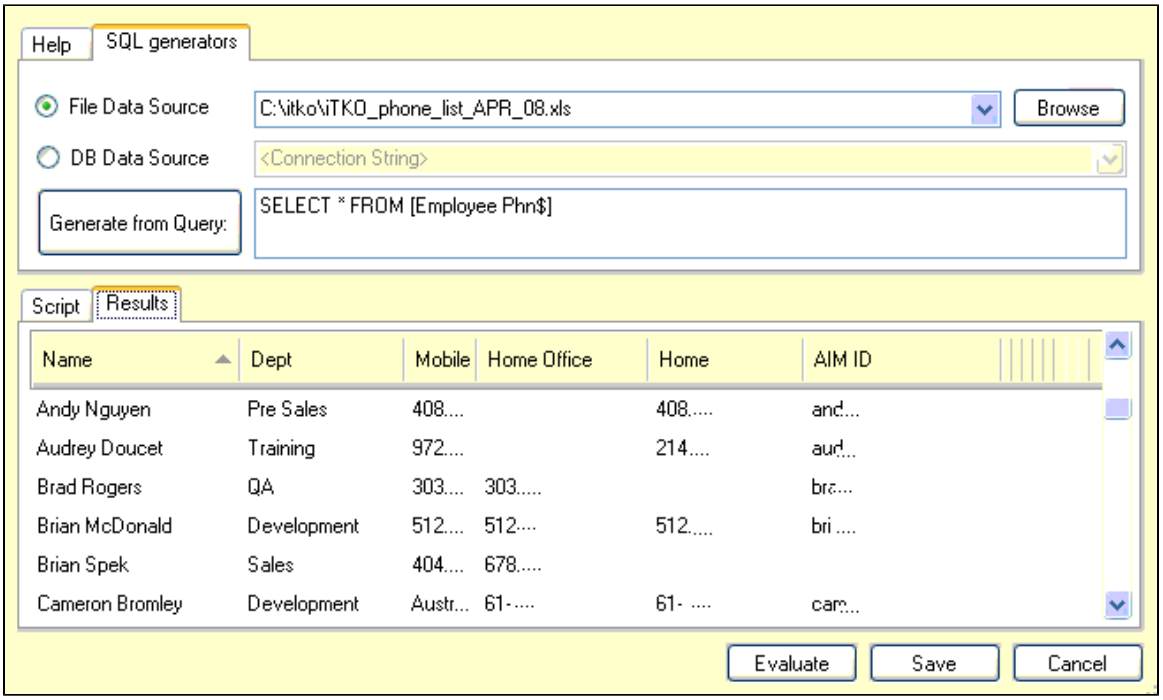

Similarly, if you created a script filter named **google**, clicked Browse to bring up the script details window, and clicked the Evaluate button after typing the script return lisa.open("c:google.html"), you see the evaluation results.




From that point on, this HTML string is available as {{ google }}. The other two functions, **fileQuery** and **dbQuery** can be used similarly, but there is a tab in the script details window named **SQL Generators** that helps you generate the calls. Filling the properties at the top and clicking **Generate From Query** yields a script snippet.

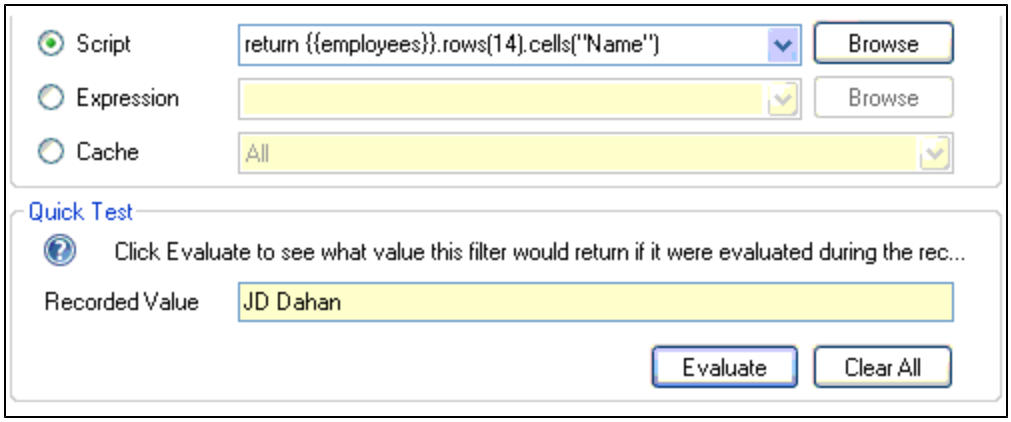



This in turns evaluates to the following dataset.




Using the API, you can create other filters and assertions to use the dataset. Assume the previous 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"), and so on.




This makes it easy to parameterize data and loop through datasets. This type of dataset is akin to a local LISA dataset because 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.

# How To 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 target 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 computer.

If your test is purely web-based, by running in dual Internet Explorer/Firefox mode, you can double that number without using many more resources.

Similar considerations apply for running other GUI technologies (Applets, Swing, .NET, Native, and others). Multiple instances are supported up to a point that is mostly determined by available hardware.

Several settings control how multiple instances run (sandboxed or shared, under which user account, and so on). Those can be seen in the Web 2.0 - Browser Architecture section.

# How To Run in a Non-privileged Account or on 64 bit Operating Systems

## Non-privileged account

The Web 2.0 step can be used on non-administrator accounts, even in Guest accounts, but it required some configuration of the environment.

1. Specify ports

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**.

> ⚠ A message box containing these instructions appears 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, and to 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`

> ⚠ A message box containing these instructions appears 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 administrator functionality.

### 64-bit operating systems

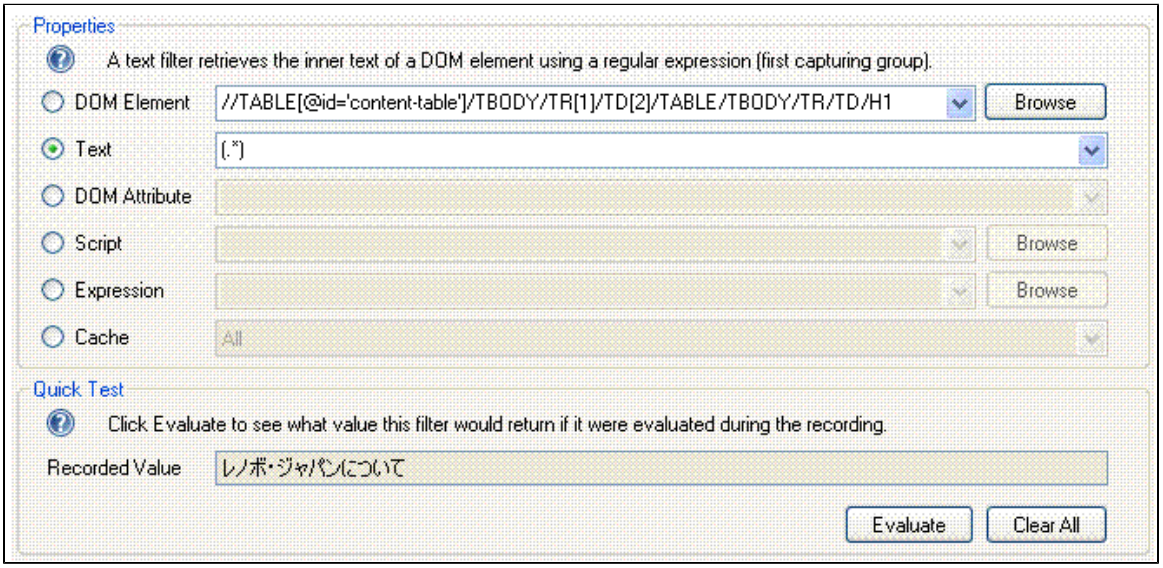
LISA Web 2.0 can run on a 64-bit operating system. The only thing to be aware of is that the component used to display the HTML source during test authoring will be disabled, but it should not affect running the test in any way.

## How To Record and Replay against Non US-English Websites

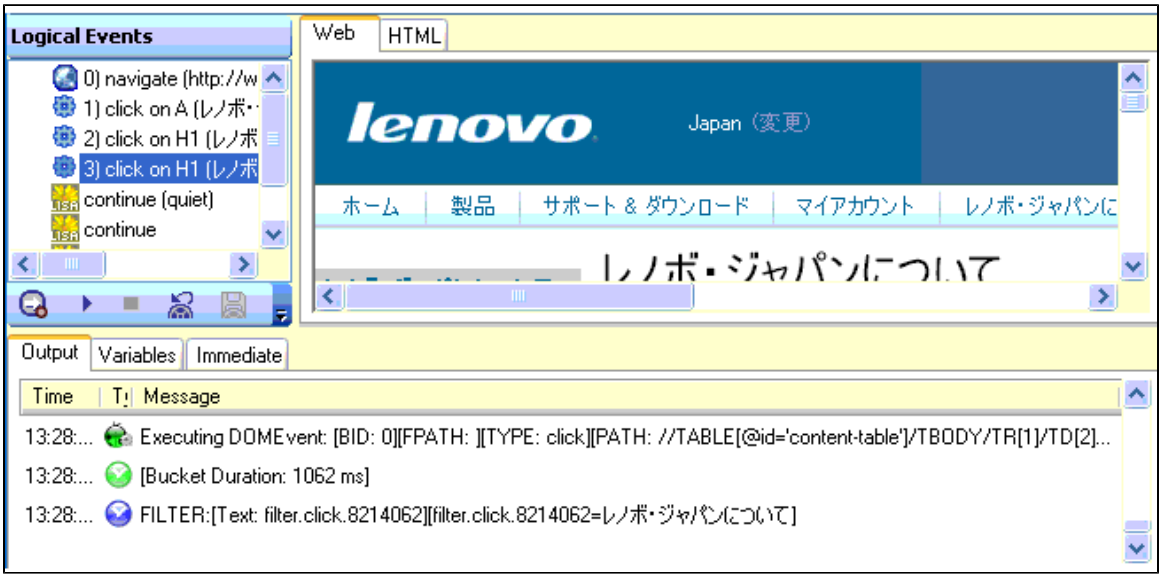
Using Web 2.0, there is nothing special to do to record and replay against websites that use any language or locale. Of course, to see the right glyphs on the screen you must download the language packs corresponding to the appropriate codepage, but if you can see them correctly in a browser, it means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

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

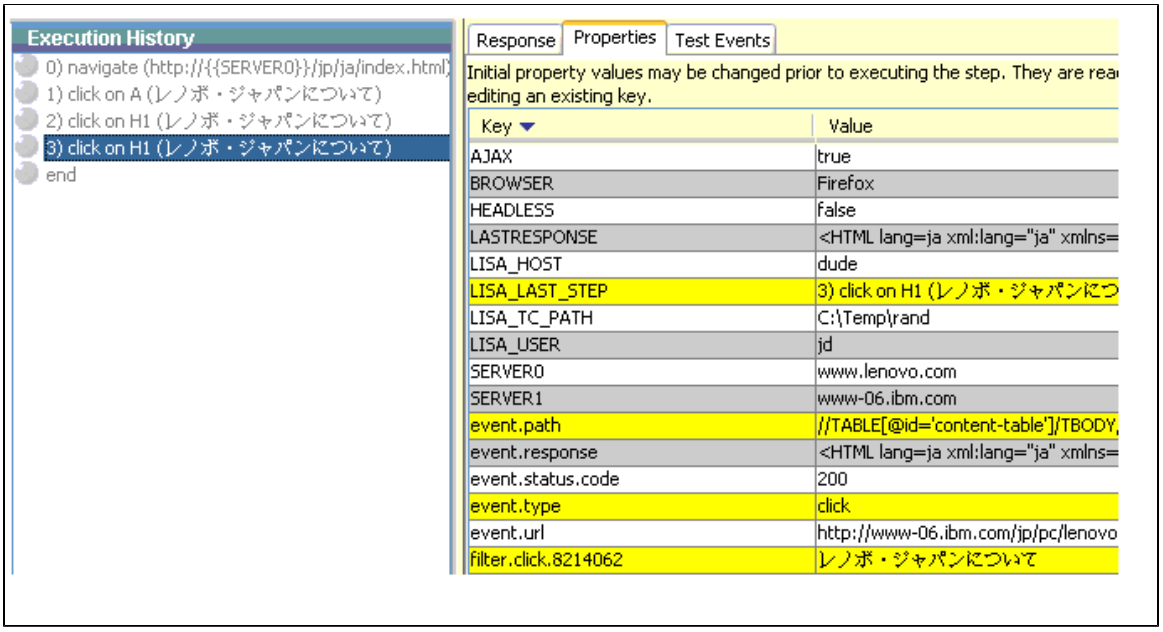
Authoring and evaluating a filter during recording:




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




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




## How To Run in Crash Dump Mode

The LISA browser runs mostly in managed code but because of external native libraries' bugs and portions running native code (for example, ActiveX, Applets, jdglue, and others), it is possible to sometimes observe crashes in certain environments and under special circumstances. These will usually manifest themselves as AccessViolation exceptions.

To make it easier to troubleshoot 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 (for example, _LISA_CRASH_DUMP=C:\Program Files\Debugging Tools for Windows (x86)).
3. Restart LISA.

When a crash occurs, LISA 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.

> ⚠ 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 reset _LISA_CRASH_DUMP to return to normal operating mode.

## How To Set a Web 2.0 Browser Timeout

To change the timeout value for the Web 2.0 browser, set the property **lisa.browser.exec.timeout** in your **local.properties** file to a large value. The value is measured in milliseconds, so, for example, something like 1000000 would set a timeout value of 15 minutes.

# Web 2.0 Videos

> ⛔ This page is under construction.

These videos detail the use of Web 2.0.

wax.html

wax.swf

### Tutorials

- Part 1 (20 minutes)
- Part 2 (24 minutes)
- Part 3 (24 minutes)
- Part 4 (13 minutes)
- Part 5 (not available)
- Part 6 (18 minutes)

### Topics

- Webstart (3 minutes)
- Win32 (Native Apps) (4 minutes)
- Filters (21 minutes)
- Datasets (5 minutes)

## Web 2.0 Tutorials Part 1

#### Part 1

## Web 2.0 Tutorials Part 2

#### Part 2

## Web 2.0 Tutorials Part 3

#### Part 3

## Web 2.0 Tutorials Part 4

#### Part 4

## Web 2.0 Tutorials Part 5

#### Part 5

## Web 2.0 Tutorials Part 6

#### Part 6

## Web 2.0 Topics - Webstart

#### Webstart

## Web 2.0 Topics - Win32 (Native Apps)

#### Win32 (Native Apps)

## Web 2.0 Topics - Filters

#### Filters

## Web 2.0 Topics - Datasets

#### Datasets

# Web 2.0 Repository Instructions

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

Download the following files to the directory **<LISA install dir>\bin\browser**. The only exceptions are:

- **web20bridge.jar** and **jdbridge.jar** goes into the <LISA install dir>\lib directory

- **djbridge.dll** and **jdglue.dll** goes into the <LISA install dir>\bin directory

You usually must download only a few files. 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 if you must revert later.

## Always update

- **lisa_browser.exe**
- **appletcallback.jar**
- **swingcallback.jar**
- **web20bridge.jar**

## 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**

## First time update

Download these files only if you do not have them.

- **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**