js-test-driver - AsyncTestCase.wiki

A traditional JsTestDriver test case looks like this: ``` var MyTest = TestCase('MyTest');

MyTest.prototype.testSomething = function() { // make some assertions, etc. }; ```

We've extended the test cases to allow their test methods to accept a parameter. The parameter is a queue that accepts inline functions that represent sequential steps of the test. The test runner executes these steps in sequence, ending the test if at any point an assertion fails.

Here's an example that uses the queue but does not use any asynchronous operations. ``` var QueueTest = AsyncTestCase('QueueTest');

QueueTest.prototype.testSomething = function(queue) { var state = 0;

queue.call('Step 1: assert the starting condition holds', function() { assertEquals(0, state); });

queue.call('Step 2: increment our variable', function() { ++state; });

queue.call('Step 3: assert the variable\'s value changed', function() { assertEquals(1, state); }); }; ``` Note we are using AsyncTestCase() now instead of TestCase(). Also, the queue has a method call() that accepts an optional string to identify the step of the test, and an operation. You could also omit the strings and just call `queue.call(function() {})`. The functions passed to the queue's `call()` method usually have more than one line, but the above example is minimal.

Finally, in order to support testing asynchronous operations, as the test runner executes each step in the queue it passes a parameter to that step. The parameter is an empty pool of callback functions. You add your callback functions to the pool so the test runner can track that they are outstanding. The pool wraps your callback functions in a way that preserves their behavior but that notifies the test runner when an asynchronous system calls them. You call `var myTweakedCallback = callbacks.add(myOriginalCallback)` which returns the tweaked version. The test runner will not execute any subsequent step in the queue until all outstanding callbacks of the current step are complete. If the callbacks are not a called for an egregious amount of time, currently set to 30 seconds, the test fails.

Here's an example: ``` var AsynchronousTest = AsyncTestCase('AsynchronousTest');

AsynchronousTest.prototype.testSomethingComplicated = function(queue) { var state = 0;

queue.call('Step 1: schedule the window to increment our variable 5 seconds from now.', function(callbacks) { var myCallback = callbacks.add(function() { ++state; }); window.setTimeout(myCallback, 5000); });

queue.call('Step 2: then assert our state variable changed', function() { assertEquals(1, state); }); }; ``` Here's a more realistic example that communicates with a hypothetical HTTP server:

``` var XhrTest = AsyncTestCase('XhrTest');
```

XhrTest.prototype.testRequest = function(queue) { var xhr = new XMLHttpRequest(); xhr.open('GET', '/some/path');

var responseBody;

queue.call('Step 1: send a request to the server and save the response body', function(callbacks) { var onStatusReceived = callbacks.add(function(status) { assertEquals(200, status); });

```
var onBodyReceived = callbacks.add(function(body) {
  responseBody = body;
});


xhr.onreadystatechange = function() {
  if (xhr.readyState == 2) { // headers and status received
    onStatusReceived(xhr.status);
  } else if (xhr.readyState == 4) { // full body received
    onBodyReceived(xhr.responseText);
  }
};


xhr.send(null);
```

});

queue.call('Step 2: assert the response body matches what we expect', function() { assertEquals('hello', responseBody); }); }; ```

Advanced CallbackPool Features

Noop Callbacks

Sometimes, you may want to do your assertions in the next test step, rather than within the callback you pass to the asynchronous system. CallbackPool has a method called noop() that returns a noop function that blocks the current step until it is called.

Example: ``` var NoopTest = AsyncTestCase('NoopTest');

NoopTest.prototype.testNoop = function(queue) { var asynchronousSystem = ...; assertFalse(asynchronousSystem.wasTriggered()); queue.call('Trigger the system', function(callbacks) { asynchronousSystem.triggerLater(callbacks.noop()); });

queue.call('Assert about the system', function()
{ assertTrue(asynchronousSystem.wasTriggered()); }); }; ```

Errbacks (Unexpected Callbacks)

Sometimes, you may want to fail the test immediately if your asynchronous system calls one callback function instead of another. For instance, imagine you pass two callback functions to your asynchronous system, one for a successful outcome, and another for an unsuccessful outcome. When one callback is executed, the other will never execute. They are mutually exclusive.

CallbackPool has a method called addErrback() precisely for this situation. Use addErrback() to return a function() that will fail the test if your asynchronous system calls it. AddErrback() accepts a string to identify which errback the asynchronous system called so you may easily identify it from the test failure message.

Example: ``` var ErrbackTest = AsyncTestCase('ErrbackTest');

ErrbackTest.prototype.testErrback = function(queue) { var asynchronousSystem = ...; assertFalse(asynchronousSystem.wasTriggered()); queue.call('Trigger the system', function(callbacks) { asynchronousSystem.triggerLater(callbacks.noop(), callbacks.addErrback('Failed to trigger')); }); queue.call('Assert about the system', function() { assertTrue(asynchronousSystem.wasTriggered()); }); }; ```

Expect Multiple Invocations

Sometimes, your asynchronous system needs to call a single callback multiple times within one step of your test. Both add() and noop() accept an optional count argument that specifies how many times you expect your asynchronous system to call its callback.

Example: ``` var MultipleTest = AsyncTestCase('MultipleTest');

MultipleTest.prototype.testMultipleInvocations = function(queue) { queue.call('Expect three invocations', function(callbacks) { var count = 0; var intervalHandle; var callback = callbacks.add(function() { ++count; if (count >= 3) { window.clearInterval(intervalHandle); } }, 3); // expect callback to be called no less than 3 times intervalHandle = window.setInterval(callback, 1000); }); }; ```

js-test-driver - TestCase.wiki

Declaring a TestCase

It is necessary to tell the test runner about your tests suites. Most test runners deal with this by letting you build up test suites. This can be tedious and error prone resulting for un-run tests. In JsTestDriver we register the test case during the declaration phase. There are two ways to declare tests in the TestCase.

Using prototype: ``` MyTestCase = TestCase("MyTestCase");

MyTestCase.prototype.testA = function(){ };

MyTestCase.prototype.testB = function(){ }; ```

Using inline declaration:

`TestCase("MyTestCase", { testA:function(){ }, testB:function(){ } });`

Test life-cycle

When tests execute they follow JUnit life-cycle: 1. Instantiate new instance of TestCase for each test method. 1. Execute the `setUp()` method. 1. Execute the `testMethod()` method. 1. Execute the `tearDown()` method.

Asserts

The asserts are declared in Asserts.js. They follow JUnit assert conventions. This means that first argument is message and is optionally followed by expected and actual values.

`expectAsserts(count)`

This asserts tells the JsTestDriver how many asserts to expect. This is useful with callbacks.

`MyTestCase.prototype.testExample = function () { expectAsserts(1); var worker = new Worker(); var doSomething = {}; worker.listener = function (work){ assertSame(doSomething, work); }; worker.perform(doSomething); };`

In the above example we have a callback function which contains an `assert`. However if the `Worker` does not call the callback function than the test would pass even thought the callback did not get called. To prevent this false positive the `expectAsserts` tells the test runner that the test is only valid if one `assert` has been called.

`fail([msg])`

`assertTrue([msg], actual)`

`assertFalse([msg], actual)`

`assertEquals([msg], expected, actual)`

`assertSame([msg], expected, actual)`

`assertNotSame([msg], expected, actual)`

`assertNull([msg], actual)`

`assertNotNull([msg], actual)`

Console

Some browsers provide console object which can be forwarded to the test runner output with `-captureConsole` flag. See CommandLineFlags. In addition there is `jstestdriver.console` class provided for you which is always forwarded to the test runners standard output.

``` ConsoleTest = TestCase("ConsoleTest");

ConsoleTest.prototype.testGreet = function() { jstestdriver.console.log("JsTestDriver", "Hello World!"); console.log("Browser", "Hello World!"); }; ```

The result of the above test when run is shown (the browser message will only be shown when `-captureConsole` is used.)

```
$ java -jar JsTestDriver.jar --tests all --captureConsole . Total 1 tests (Passed: 1; Fails: 0; Errors: 0) (1.00 ms) Safari 528.16: Run 1 tests (Passed: 1; Fails: 0; Errors 0) (1.00 ms) ConsoleTest.testGreet passed (1.00 ms) [LOG] JsTestDriver Hello World! [LOG] Browser Hello World!
```

# js-test-driver - QUnitAdapter.wiki
```

QUnit to JS Test Driver adapter

Version: 1.1.0

Author: Karl O'Keeffe (`karl@monket.net`, http://monket.net)

Blog post introduction: http://monket.net/blog/2009/06/new-qunit-to-js-test-driver-adapter/

Introduction

Qunit Adapter provides a small wrapper around your QUnit tests that allows them to be run using JS Test Driver.

It works by converting each qunit test and assertion into corresponding JS Test Driver test methods and assertions. Each qunit module maps to a JS Test Driver TestCase, each qunit test maps to a test method on that TestCase. And each qunit ok, equals, or same assertion maps to a JS Test Driver assertion. Qunit lifecycles (setup and teardown) also map to JS Test Driver setUp and tearDown.

This ensures you still get assertion level error reporting when running your qunit tests with JS Test Driver.

Essentially this adapter allows you to write native JS Test Driver tests, but using the less verbose qunit syntax.

Installing the QUnit Adapter

Download `equiv.js` and `QUnitAdapter.js` from source control: http://code.google.com/p/js-test-driver/source/browse/#svn/trunk/JsTestDriver/contrib/qunit/src

Copy both the `equiv.js` and `QUnitAdapter.js` files to your project test directory (for example `tests/qunit/`).

Configuring JS Test Driver

To run your qunit tests in JS Test Driver you need to configure it to load the adapter before your qunit tests.

Update your `jsTestDriver.conf` to load the files:

``` server: http://localhost:9876

load: # Add these lines to load the equiv function and adapter in order, before the tests # (assuming they are saved to tests/qunit/) - tests/qunit/equiv.js - tests/qunit/QUnitAdapter.js

# This is where we load the qunit tests - tests/js/*.js
```

# And this loads the source files we are testing - src/js/*.js ```

Running JS Test Driver with qunit tests

Now we can run JS Test Driver and watch as it runs all our qunit tests!

The tests will run as individual JS Test Driver tests, with the format `Module Name.Test Name`.

Example output:

```
[PASSED] Module 1.test Test 1 [PASSED] Module 1.test Test 2 [PASSED]
Module 2.test Test 1 Total 3 tests (Passed: 3; Fails: 0; Errors: 0)
(1.00 ms) Safari 530.18: Run 3 tests (Passed: 3; Fails: 0; Errors 0)
(1.00 ms)
```

Limitations

There are a few limitations on which qunit tests will successfully be converted.

The tests must run synchronously (which means no use of the qunit `stop` and `start` methods).

If you need to test timeouts, intervals, or other asynchronous sections of code, consider using the jsUnit Clock object to deal with timeouts and intervals.

QUnit DOM support is not included. Consider avoiding interacting directly with the browser within your unit tests. But if you do need to, you'll need to create and remove the DOM objects yourself with each test, or the setup and teardown methods.

Release Notes

See History: http://code.google.com/p/js-test-driver/source/browse/trunk/JsTestDriver/contrib/qunit/History.txt

js-test-driver - HtmlDoc.wiki

Html Doc

So, you've got this lean and spare JsUnit test. And now, you want to test some DOM interactions and get some use out of testing on all these browsers. You could use the built in dom methods: ```

TestCase.prototype.setUp = function() { this.div = document.createElement('div'); var p = document.createElement('p'); div.appendChild(p); p.innerHTML = "bar"; div.id = 'foo'; }

```

But, it's gets complicated. What if we could just have a simple declarative process to create html?

Well, it turns out, in JsTestDriver, you get just that. Using the simplified comment syntax you can either create the html scoped to your test, or appended to body:

### Html Scoped to a Test:

```

TestCase.prototype.testFoo = function() { assertUndefined(this.foo); /**:DOC foo =**

foo

/ assertNotUndefined(this.foo); };

```

### Html Appended to the Body

```

TestCase.prototype.testFoo = function() { /*:DOC += */
assertNotNull(document.getElementById('foo')); }; ```

js-test-driver - Gateway.wiki

Introduction

JsTestDriver has the ability to gateway unrecognized requests to servers-under-test. You may find this useful for larger integration tests that need to communicate with a backend server.

Details

To activate the gateway, add a `gateway` section (or `proxy` for older versions of JsTestDriver up to 1.3.2) to your jsTestDriver.conf file:

`gateway: - {matcher: "/matchedPath", server: "http://localhost:7000"} - {matcher: "/wildcardPath/*", server: "http://localhost:8000/"} - {matcher: "*", server: "http://localhost:9000"}`

The above configuration sends requests to `/matchedPath` along to the `http://localhost:7000`, requests to `/wildcardPath/{anything}` along to `http://localhost:8000/{anything}`, and any remaining requests to `{anything}` along to `http://localhost:9000/{anything}`.

The `gateway` (or `proxy`) entry of the configuration file is a list of matchers mapped to server addresses (including an optional path). When handling unknown requests, JsTestDriver iterates sequentially through the list of matchers, finds the first matching pattern, and forwards the request along to the server URL, appending any extra path information matched by a wildcard.

Matcher patterns come in three varieties: * Literal matchers, e.g. `/matchedPath` * Suffix matchers, e.g. `/wildcardPath/*` * Prefix matchers, e.g. `*.pdf`

Path Collisions

Sometimes your server-under-test may handle HTTP requests on URLs that JsTestDriver already handles. For instance, you may handle requests on `/cache` that are vital to your service and that you would like to test.

Use the following flag `--serverHandlerPrefix jstd` to prefix all JsTestDriver-specific request paths with `/jstd` so they won't collide with your service.

You have to use this flag when starting server and running tests both.

List of all paths used by JsTestDriver can be found here.

Security

Be sure you control or trust the servers that you enter into your `gateway` (or `proxy`) configuration entry. The gateway bypasses the browser's same-origin policy and forwards the requests almost verbatim.

js-test-driver - CommandLineFlags.wiki

Help

```
$ java -jar JsTestDriver.jar --help --browser VAR : The path to the
browser executables, separated by ','. Arguments can be passed to the
executables separated by ';'. '%s' will be replaced with the initial
url. If '%s' is not included the url will be appended to the argument
list. --browserTimeout VAR : The ms before a browser is declared
dead. --captureConsole : Capture the console (if possible) from the
browser --config VAL : Loads the configuration file --dryRunFor VAR :
Outputs the number of tests that are going to be run as well as their
names for a set of expressions or all to see all the tests --help :
Help --port N : The port on which to start the JsTestDriver server --
preloadFiles : Preload the js files --requiredBrowsers VAR : Browsers
that all actions must be run on. --reset : Resets the runner --server
VAL : The server to which to send the command --serverHandlerPrefix
VAL : Whether the handlers will be prefixed with jstd --testOutput
VAL : A directory to which serialize the results of the tests as XML
--tests VAR : Run the tests specified by the supplied regular
expression. Use '#' to denote the separation between a testcase and a
test. --verbose : Displays more information during a run --plugins
VAL[,VAL] : Comma separated list of paths to plugin jars. --config
VAL : Path to configuration file. --basePath VAL : Override the base
path in the configuration file. Defaults to the parent directory of
the configuration file. --runnerMode VAL : The configuration of the
logging and frequency that the runner reports actions: DEBUG,
DEBUG_NO_TRACE, DEBUG_OBSERVE, PROFILE, QUIET (default), INFO --
serverHandlerPrefix : Prefix for all jstd paths (to avoid conflict
with proxy)
```

Server Options

`--port`

When starting up a server all you need to specify is the `--port` on which port number should the server be running. The same port number is than used to both capture the browsers as well as for the test runner to connect to.

Status

Once the server is running you can visit the server base URL with your browser to see that status of the servers. If you started the server on port 4224 than you can visit the status of the server by visiting http://localhost:4224. Here is a sample output of the server status.

Capture This Browser

Captured Browsers:

Id: 2
Name: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.0.10) Gecko/2009042315 Firefox/3.0.10
Version: 5.0 (Macintosh; en-US)
Operating System: MacIntel

Id: 1
Name: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_7; en-us) AppleWebKit/528.16 (KHTML, like Gecko)
Version/4.0 Safari/528.16
Version: 5.0 (Macintosh; U; Intel Mac OS X 10_5_7; en-us) AppleWebKit/528.16 (KHTML, like Gecko) Version/4.0 Safari/528.16

Operating System: MacIntel

Browser Capture

Visiting http://localhost:4224/capture will automatically capture the browser by the server. At this point the browser is available to be used for running tests.

The browser does not have to be on the same machine. This allows a server running on one platform to capture browsers from other platform giving the developer full access to all browsers/platform combinations.

`--browser`

If you wish to auto-capture a browser on startup you can specify a list of paths (separated by comma `','`) to the browser on the command line using this option. This will automatically launch the browser and capture it. This is useful when setting up JsTestDriver in ContinuousBuild mode. Arguments can be passed to the executables separated by ';'. '%s' will be replaced with the capture url. If '%s' is not included the capture url will be appended to the argument list.

Test Runner Options

If `--port` option is not present than we are running in a test runner mode. This client is run by the developer to run the tests in automated fashion from the IDE.

`--captureConsole`

Many browsers have a `console` object which has logging methods for debugging. When this option is specified than we try to intercept the console log messages and display them on the test runner standard out. This allows the developer to see what is going on on a remote browsers console to aid in debugging. (FireFox does not allow overwriting of the console object.)

`--config`

By default the test runner reads the `jsTestDriver.conf` file in the current directory to get its configuration. This option allows you to override the location of the ConfigurationFile.

`--reset`

Asks the server to reload the browsers. Sometimes it is passible that slave gets into an inconsistent state. For example a test has overridden some global variable. Sending a reset signal should restore the slave to standard configuration.

`--server`

Specifies the location of the server to connect to for running the tests. By default we try to use the location form the ConfigurationFile, but this option allows the developer to override the configuration file.

`--testOutput`

For ContinuousBuild it is often necessary to publish the test results in a file for later processing. This option specifies which directory the tests results should be written to. The test results are written in XML JUnit format which should be compatible with most of the continuous build systems out there. There will be one XML file written per browser captured. The format should be compatible with most continuous build systems which understand JUnit XML format.

`--tests`

Specifies which tests should be run. * `all` special keyword to run all of the tests.
* `TestCaseName` run all tests for that test case. * `TestCaseName#testName` run only the specified test, useful when debugging a single test. See DebuggingWithJsTestDriver. * Additionally, it accepts regular expressions for any part of the string. '#' is reserved, and unexpected results may happen when using # in a test name.

`--verbose`

Normally the test runner tries to run as many tests as passible before reporting status of the test to the user. This is done for performance reasons. But it i possible that a test can hang the runner in this case you have no idea which tests is causing the problem since nothing is displayed. This option makes the test runner verbose and as a result it reports on every test as it runs allowing you to pinpoint the failure.

js-test-driver - DebuggingWithJsTestDriver.wiki

We have tried hard to make sure that your debugging experience with JsTestDriver is a good one. For the most part there is nothing special about the debugging JsTetsDriver tests. Here are typical steps:

1. Start server and capture the browser which you would like to debug
2. Run the failing test using `java -jar JsTestDriver.jar --tests MyTestCase.testIWantToDebug` This should load all of your JavaScript source files into the Browser
3. Open your debugger and you should see all of your source files present in your debugger. Open the source file and place a breakpoint anywhere you wish.

4. Rerun the test using command in step #2. The debugger should stop at your breakpoint where you should have full access all of your code, variables, and DOM.

Happy debugging...

js-test-driver - ForDevelopers.wiki

Introduction

Info needed for developers and contributors.

Building the core jar

Just checkout the project, install ant, and run `ant jar` in the root folder of the project.

Releasing

Run `ant release` **You should use Java6 and Ant 1.7.1 or greater, or your codesite password will be echoed. Don't worry, the Javac task will still target 1.5 class files.**

This will build and test the code, tag the revision in svn, and publish the artifacts to the codesite, marking them Featured. You just need to remove the Featured tag from the files of the previous release.

Troubleshooting

Gah! ``` instrument: [instr] processing instrumentation path ...

BUILD FAILED com.vladium.emma.EMMARuntimeException: [UNEXPECTED_FAILURE] unexpected failure java.lang.ArrayIndexOutOfBoundsException: 14, please submit a bug report to: 'http://sourceforge.net/projects/emma' at com.vladium.emma.instr.InstrProcessorST._run(InstrProcessorST.java:784) at com.vladium.emma.Processor.run(Processor.java:54) at com.vladium.emma.instr.instrTask.execute(instrTask.java:77) at com.vladium.emma.emmaTask.execute(emmaTask.java:57) at org.apache.tools.ant.UnknownElement.execute(UnknownElement.java:288) at sun.reflect.GeneratedMethodAccessor3.invoke(Unknown Source) ``` you're running a 64-bit JVM, which emma doesn't work with. Run a 32-bit JVM.

Building the Eclipse plugin

Set some environment variables, so the Eclipse plugin will build. You may want to download the RCP/PDE plugin development version of Eclipse so that the plugin tools are available.

Otherwise you'll need to figure out the appropriate plugins to add to an Eclipse install. * ECLIPSE_LAUNCHER_JAR=/plugins/org.eclipse.equinox.launcher_1.0.101.R34x_v20081125.jar * ECLIPSE_PDE_XML=/plugins/org.eclipse.pde.build_3.4.1.R34x_v20081217/scripts/build.xml * ECLIPSE_BASE_DIR=

Then run ant.

Working on the IDEA plugin

See [JsTestDriver IntelliJ plugin Development]() page

To build a release

Point an Ant variable to the location of your IDEA installation (can't point to a "platform" IDE here, as the javac2.jar isn't included with those products). `ant -DIDEA_LIB_PATH=/Applications/IntelliJ\ IDEA\ 9.0.3\ CE.app/lib jar` or on Windows, `IDEA_LIB_PATH=C:\\devel\\idea-8\\lib\\`

Releasing the plugin

- Change the version number in META-INF/plugin.xml to match the release of the core library that's included (the IDEA plugin repo uses this to determine our version)
- Update the changelog in the same file, adding a section for your release at the top. This will be displayed inside of IDEA, which helps users decide to update
- Carefully build all the code - the core JSTD jar as well as the plugin, with the `zip` target
- Log into the update site at http://plugins.intellij.net/ username jakeherringbone, password Alex for it
- Go to http://plugins.intellij.net/plugin/edit/?pid=4468
- Fill in the form:
  - Select the zip file you just built
  - Copy the current change info into the 'Add info to RSS feed and news channels' box
  - Source code download URL http://code.google.com/p/js-test-driver/source/browse/
  - Forum URL http://groups.google.com/group/js-test-driver

js-test-driver - ContinuousBuild.wiki

To run your tests as part of the continuous build step we provide an easy way to launch the server, capture browsers, run tests, report the status and than automatically shut down the browsers and the server. The command to do all of this is below. The key is to specify `--port`and `--tests` flags together

```
java -jar JsTestDriver.jar --port 4224 --browser
broserpath1,browserpath2 --tests all --testOutput testOutputDir
```

Here is sample output of the server.
```
$ java -jar JsTestDriver.jar --port 4224 -
-browser open --tests all --testOutput . May 21, 2009 7:13:24 PM
org.slf4j.impl.JCLLoggerAdapter info INFO: Logging to
org.slf4j.impl.JCLLoggerAdapter(org.mortbay.log) via
org.mortbay.log.Slf4jLog May 21, 2009 7:13:24 PM
org.slf4j.impl.JCLLoggerAdapter info INFO: jetty-6.1.x May 21, 2009
7:13:24 PM org.slf4j.impl.JCLLoggerAdapter info INFO: Started
SocketConnector@0.0.0.0:4224 May 21, 2009 7:13:24 PM
org.slf4j.impl.JCLLoggerAdapter info INFO: Browser Captured:
Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_7; en-us)
AppleWebKit/528.16 (KHTML, like Gecko) Version/4.0 Safari/528.16
version 5.0 (Macintosh; U; Intel Mac OS X 10_5_7; en-us)
AppleWebKit/528.16 (KHTML, like Gecko) Version/4.0 Safari/528.16 (1)
Total 1 tests (Passed: 1; Fails: 0; Errors: 0) (0.00 ms) Safari
528.16: Run 1 tests (Passed: 1; Fails: 0; Errors 0) (0.00 ms)
```

And here is a sample XML output file
```
$ cat TEST-com.google.jstestdriver.1.xml
<?xml version="1.0" encoding="UTF-8"?> <testsuite
name="com.google.jstestdriver.1"> <testcase classname="GreeterTest"
name="testGreet:Safari528.16" time="0.0010"/> <system-
out><![CDATA[[LOG] JsTestDriverHello World! ]]></system-out>
</testsuite>
```

Integration with Existing CI Systems

Here are some examples of how you can integrate JsTestDriver with existing Continuous Builds. This is not an exhaustive list, but you can use this as a starting point on how to integrate it with your system.

Atlassion Bamboo & QUnit

by Mike Arvela

As I managed to come up with a solution myself, I thought it would be a good idea to share it. The approach might not be flawless, but it's the first one that seemed to work. Feel free to post improvements and suggestions.

What I did in a nutshell: * Launch an instance of Xvfb, a virtual framebuffer * Using JsTestDriver: * launch an instance of Firefox into the virtual framebuffer (headlessly) * capture the Firefox instance and run the test suite * generate JUnit-compliant test results .XML * Use Bamboo to inspect the results file to pass or fail the build

I will next go through the more detailed phases. This is what my my directory structure ended up looking like: `lib/ JsTestDriver.jar test/ qunit/ equiv.js QUnitAdapter.js jsTestDriver.conf run_js_tests.sh tests.js test-reports/ build.xml`

On the build server: * Install Xvfb (apt-get install Xvfb) * Install Firefox (apt-get install firefox)

Into your application to be built: * Install JsTestDriver: http://code.google.com/p/js-test-driver/ * add the QUnit adapters equiv.js and QUnitAdapter.js * configure JsTestDriver (jsTestDriver.conf):

``` server: http://localhost:4224

load:

## Load QUnit adapters (may be omitted if QUnit is not used)

- qunit/equiv.js
- qunit/QUnitAdapter.js

## Tests themselves (you'll want to add more files)

- tests.js ```

Create a script file for running the unit tests and generating test results (example in Bash, run_js_tests.sh):

```

!/bin/bash

directory to write output XML (if this doesn't exist, the results will not be generated!)

OUTPUT_DIR="../test-reports" mkdir $OUTPUT_DIR

XVFB=`which Xvfb` if ["$?" -eq 1]; then echo "Xvfb not found." exit 1 fi

FIREFOX=`which firefox` if ["$?" -eq 1]; then echo "Firefox not found." exit 1 fi

$XVFB :99 -ac & # launch virtual framebuffer into the background PID_XVFB="$!" # take the process ID export DISPLAY=:99 # set display to use that of the xvfb

run the tests

java -jar ../lib/JsTestDriver.jar --config jsTestDriver.conf --port 4224 --browser $FIREFOX --tests all --testOutput $OUTPUT_DIR

kill $PID_XVFB # shut down xvfb (firefox will shut down cleanly by JsTestDriver) echo "Done." ```

Create an Ant target that calls the script: ```

```
<exec executable="/bin/bash" dir="test" osfamily="unix">

   <arg value="run_js_tests.sh" />

</exec>
```

``` Finally, tell the Bamboo build plan to both invoke the test target and look for JUnit test results. Here the default "/test-reports/

# js-test-driver - GettingStarted.wiki

## Introduction

JsTestDriver aims to help javascript developers use good TDD practices and aims to make writing unit tests as easy as what already exists today for java with JUnit.

## Laying out Your Project

We have created a sample Hello-World project for you here: http://code.google.com/p/js-test-driver/source/browse/#svn/samples/hello-world

There are three things you need to set up your project with JsTestDriver: 1. **source folder:** Although not strictly required we recommend that you create a source folder for your application. In our case we have named it `src`. 1. **test folder:** Again not strictly necessary, but keeping your test code away from your production code is a good practice. In our case we have named it `src-test`. 1. **configuration file:** By default the JsTestDriver runner looks for the configuration file named `jsTestDriver.conf` in the current directory. We recommend that you name it same way, or you will have to enter it as a command line option all of the time.

## Writing your production code

You write your JavaScript production code as you normaly would. We have created a sample `Greeter` class for domenstration purposes. ``` myapp = {};
```

myapp.Greeter = function() { };

myapp.Greeter.prototype.greet = function(name) { return "Hello " + name + "!"; }; ```

Writing your test code

We have tried to follow the JUnit testing conventions as much as possible. If you are familiar with JUnit, than you should be right at home with JsTestDriver.

``` GreeterTest = TestCase("GreeterTest");

GreeterTest.prototype.testGreet = function() { var greeter = new myapp.Greeter(); assertEquals("Hello World!", greeter.greet("World")); }; ```

The JsTestDriver needs to know about all of your `TestCase` classes for this reason you declare a new test class using the notation below. Once the `TestCase` is declared it acts as a normal class declaration in JavaScript. You can use normal prototype declarations to create test methods. See TestCase for more information. `GreeterTest = TestCase("GreeterTest");`

Test methods are declared on the prototype. Optionally you can declare `setUp()` and `tearDown()` method just as in JUnit.`GreeterTest.prototype.testGreet = function() { var greeter = new myapp.Greeter(); assertEquals("Hello World.", greeter.greet("World")); };`

## Writing configuration file

If you are familiar with Java, you can think of the configuration file as classpath for JavaScript. The file contains information about which JavaScript source files need to be loaded into the browser and in which order. The configuration file is in YAML format.

``` server: http://localhost:9876

load: - src/**.js - src-test/**.js ```

The server directive tells the test runner where to look for the test server. If the directive is not specified you will need to enter the server information through a command line flag.

The load directive tells the test runner which JavaScript files to load into the browsers and in which order. The directives can contain "`*`" for globing multiple files at once. In our case we are saying to load all of the files in the src folder followed by all of the files in the srt-test folder. For more information see ConfigurationFile.

## Starting the server & Capturing browsers

Before you can run any of your tests you need to start the test server and capture at least one slave browser. The server does not have to reside on the machine where the test runner is, and the browsers themselves can be at different machines as well.

Starting server on port 9876: `java -jar JsTestDriver.jar --port 9876`

Then capture a browser by going to the URL of your JsTestDriver server, if running on localhost it should be: `http://localhost:9876`

Click the link `Capture This Browser`. Your browser is now captured and used by the JsTestDriver server.

You can also directly capture the browser by using the URL: `http://localhost:9876/capture`

You can also tell the server to autocapture the browser by providing a path to the browser exectable on the cammand line. Multiple browsers can be specified if separated by a comma ','. `java -jar JsTestDriver.jar --port 9876 --browser firefoxpath,chromepath`

For full line of command line flags refer to CommandLineFlags.

## Running the tests

Now that we have server running and at least one browser captured we can start running tests. Tests can be executed from the command line using: `java -jar JsTestDriver.jar --tests all`

As long as the jsTestDriver.conf file is present in the current directory the test runner will read it and use it to locate the server and the files which need to be loaded into the browser. It will then load any files (which have changed) into the browser and run your tests reporting the results an standard out.

```
Total 2 tests (Passed: 2; Fails: 0; Errors: 0) (0.00 ms) Safari
528.16: Run 1 tests (Passed: 1; Fails: 0; Errors 0) (0.00 ms) Firefox
1.9.0.10: Run 1 tests (Passed: 1; Fails: 0; Errors 0) (0.00 ms)
```

Automatically running tests in Eclipse

For discussion how to set up your tests to run automatically on save see: http://misko.hevery.com/2009/05/07/configure-your-ide-to-run-your-tests-automatically/

js-test-driver - ConfigurationFile.wiki

Introduction

Configuration file is written in YAML and is used to tell the test runner which files to load to browser and in which order. By default the JsTestDriver looks for the configuration file in the current directory and with the name `jsTestDriver.conf`. You can use `--config`command line option to specify a different file.

Example: ``` server: http://localhost:4224

load: - src/*.js

test: - src-test/*.js

exclude: - uselessfile.js

serve: - css/main.css

proxy: - {matcher: "*", server: "http://localhost/whatever"}

plugin: - name: "coverage" jar: "lib/jstestdriver/coverage.jar" module: "com.google.jstestdriver.coverage.CoverageModule"

timeout: 90

```

## `server:`

Specifies the default location of the server. This value can be overridden with a command line option `--server`. See CommandLineFlags.

## `load:`

List of files to load to browser before the test can be run. We support globing with `*`. The files are loaded in the same order as specified in the configuration file or alphabetically if using globing.

You can declare external scripts by adding the http address of the script as a loadedable item.

## `test:`

A list of test sources to run.

## `exclude:`

Never load this file. Used in conjunction with globing and `load`. Useful saying load everything except these files.
```

`serve:`

Load static files (images, css, html) so that they can be accessed on the same domain as jstd.

`proxy:`

Set jstd to behave as proxy. See proxy.

`plugin:`

Load jstd plugin. See plugins.

`timeout:`

Timeout in seconds.

js-test-driver - Assertions.wiki

Default Assertions

`fail([msg])`

Throws a JavaScript Error with given message string.

`assert([msg], actual)`

`assertTrue([msg], actual)`

Fails if the result isn't truthy. To use a message, add it as the first parameter.

`assertFalse([msg], actual)`

Fails if the result isn't falsy.

`assertEquals([msg], expected, actual)`

Fails if the expected and actual values can not be compared to be equal.

`assertNotEquals([msg], expected, actual)`

Fails if the expected and actual values can be compared to be equal.

`assertSame([msg], expected, actual)`

Fails if the expected and actual values are not references to the same object.

`assertNotSame([msg], expected, actual)`

Fails if the expected and actual are references to the same object.

`assertNull([msg], actual)`

Fails if the given value is not exactly null.

`assertNotNull([msg], actual)`

Fails if the given value is exactly null.

`assertUndefined([msg], actual)`

Fails if the given value is not undefined.

`assertNotUndefined([msg], actual)`

Fails if the given value is undefined.

`assertNaN([msg], actual)`

Fails if the given value is not a NaN.

`assertNotNaN([msg], actual)`

Fails if the given value is a NaN.

`assertException([msg], callback, error)`

Fails if the code in the callback does not throw the given error.

`assertNoException([msg], callback)`

Fails if the code in the callback throws an error.

`assertArray([msg], actual)`

Fails if the given value is not an Array.

`assertTypeOf([msg], expected, value)`

Fails if the JavaScript type of the value isn't the expected string.

`assertBoolean([msg], actual)`

Fails if the given value is not a Boolean. Convenience function to assertTypeOf.

`assertFunction([msg], actual)`

Fails if the given value is not a Function. Convenience function to assertTypeOf.

`assertObject([msg], actual)`

Fails if the given value is not an Object. Convenience function to assertTypeOf.

`assertNumber([msg], actual)`

Fails if the given value is not a Number. Convenience function to assertTypeOf.

`assertString([msg], actual)`

Fails if the given value is not a String. Convenience function to assertTypeOf.

`assertMatch([msg], regexp, actual)`

Fails if the given value does not match the given regular expression.

`assertNoMatch([msg], regexp, actual)`

Fails if the given value matches the given regular expression.

`assertTagName([msg], tagName, element)`

Fails if the given DOM element is not of given tagName.

`assertClassName([msg], className, element)`

Fails if the given DOM element does not have given CSS class name.

`assertElementId([msg], id, element)`

Fails if the given DOM element does not have given ID.

`assertInstanceOf([msg], constructor, actual)`

Fails if the given object is not an instance of given constructor.

`assertNotInstanceOf([msg], constructor, actual)`

Fails if the given object is an instance of given constructor.

js-test-driver - DesignPrinciples.wiki

Our goal is to have a better way to write JavaScript tests

Overview

JsTestDriver is to unit-tests what Selenium is to end-to-end tests. The goal of JS Test Driver is to make JavaScript unit test development as seamless as and easy as Java unit tests.

Features * Support TDD development model * Super fast test execution * Super easy set up * Seamless integration with existing IDEs * Debugger Support * Federated test executions across all browsers and platforms * Focus on the command line and the continuous build * Designed to be farm friendly * HTML Loading Support

Support TDD development model

Test-Driven-Development encourages to write lots of small focused tests and run them often. The goal of JsTestDriver was to enable the execution of the complete test suite on each save. This places severe constraints on latency. The goal was to enable the running of large (hundreds/thousands) of tests in under one second. If I am working in an IDE I save my files often. This means that as a user I can only tolerate a second or two for my tests results to come back.

Traditional test frameworks fall short of this goal. First it is not possible to run all tests in interactive mode. The reason is that each test case consist of HTML file and a collection of tests (usually around 10). The developer then loads the HTML into the Browser. To run the tests you have to refresh the browser. The Browser then reloads, reparses all of the resources and executes only the tests in this test case. Therefore as a developer I can only run one test case at a time. This implies that it is easy to break other test cases and not notice it until you check it in and the continuous build runs all of the tests cases in your tests suite. Constant refreshing of the browser also breaks the development flow, and makes it that the developer does not run the tests often.

In contrast JsTestDriver runs all of the tests every time you save.

Super fast test execution

If we wish to execute all of the tests in under one sec we have to rethink the way the tests are run. Lets see where the latency of running tests comes from in traditional test frameworks: * Start up of browser can take several seconds * Constant reloading of the HTML test files causes the reloading of all of the resources which puts strain on the network. * If the resource is JavaScript than reload also implies the re-parsing of the JavaScript AST

We designed the JsTestDriver for speed from ground up. To achieve super fast execution times JsTestDriver consist of server and client code. The server captures any number of web browsers and keeps them "hot" and ready for test execution. The browser loads and parses the HTML file only once. The original HTML file contains code which turns the browser into a slave listening on the server for commands to execute. Each browser then evals any code which the server sends a request for. The server then loads your production and tests code and runs the tests for you. If you change any code the server only loads the changed code into the browser. This greatly lowers the amount of reparsing which the browser needs to do. Additionally the server is eager and it loads the code aggressively into the browser even before it is ready to run. Because the browser and their documents with your application and test code are long lived running tests is in the millisecond range. Allowing JsTestDriver to finish the whole test suite in sub-second times.

Super easy set up

Writing a test should be as simple as: 1. start the JsTestsDriver 1. write test 1. write production code. The test running should be automatic on each save.

In contrast traditional test frameworks require you to write an HTML file. This file needs to contain JavaScript dependencies to the framework, production code and tests. This means that each HTML file has a complex project dependent set up. Most developers tend to solve this by cutting and pasting the initialization files between the HTML test files.

In contrast JsTestDriver does not require any HTML file to be written by the developer. It really is as easy as writing your tests in JavaScript format.

Seamless integration with existing IDEs

Traditional test frameworks don't play nice with the IDEs. This is because they do not run in the IDE but in the browser. The context switch does not seem like a lot but it is enough to make the process clunky. Our goal is to be able to right-click on the test in the IDE and run it from within the IDE in isolation (not possible now) or as part of the whole test suite (not possible now) and have the results report inside the IDE UI (not possible now) where you can directly click on the stack trace and go to the source of the exception (again not possible with existing tools).

JsTestDriver consist of a client and a server. The server can run anywhere, but usually on the same machine. The client on the other hand is an embedable code which can be run from command line or from within any tool, such as an IDE.

Debugger support

Each browser has an existing debugger support. Our goal with JsTestDriver is to make sure that we do not break any of the existing debugger workflows and that you can place breakpoint in your production or test code.

Federated test executions across all browsers and platforms

Traditional JavaScript test frameworks run inside of a single browser. This means that each developer can run the code only in one browser at a time. This means that most tests are not run on most browsers most of the time. To make the matters worse the developer usually only has a single platform available at a time, which further complicates the problem.

In JsTestDriver the server can capture any number of browsers from any number of machines and any number of platforms. We can even have multiple versions of the same browser captured at a same time. When tests are run they are executed on all browsers in parallel. This means that when the developer runs the tests the answer will include the pass or fail information from all of the browsers at once. This means that you can easily compare browser (miss)behavior.

Focus on the command line and the continuous build

Most traditional test frameworks focus on running the tests and forget that tests need to be run in automated fashion in the continuous build machine. The biggest problem being how do I get the result of my tests out of the browser and in to a JUnit compliant XML file. Often times developers write a JUnit target which starts Selenium and controls the HTML tests runner. Java then uses selenium to load individual HTML files and records the pass fail information. This is extremely slow, not to mention backwards. Why do I need to write Java code to run JavaScript tests?

Designed to be farm friendly

JsTestDriver is designed with farm availability in mind. The JsTestDriver server can run anywhere. This means that you can have a large pool of servers running all configured with the right browser and ready to run the tests for any developer. This will greatly help with the continuous build machines ability to execute the tests across multiple platforms. It will also enable the teams to easily set up continuous builds.

Future Road Map

HTML loading support

Since we no longer have HTML files we need an alternative method of loading the HTML test data into the tests. This is currently supported as a string literal, but we are working on a cleaner way to load HTML content into tests.

Support for code coverage through on the fly code instrumentation

Once the basics are in place we would like to include support for on the fly code instrumentation of the code to gather coverage numbers. Since the server sends all of the data to the server, the server can do on the fly instrumentation of the code.

Status Console/Charts

A status page for the server which would show the state of the browsers, coverage, performance, and allowed the control of the browsers remotely.

js-test-driver - IntelliJPlugin.wiki

JsTestDriver IntelliJ plugin allows you to enjoy all the benefits of JsTestDriver right from the comfort of your IDE (WebStorm, PhpStorm, IntelliJ IDEA, RubyMine, PyCharm or AppCode).

It is the open-source project under the terms of Apache License 2.0.

Features

- starting and stopping the server;
- running and rerunning tests;
- filtering and viewing test results, navigation from results to source code;
- jumping from JavaScript exception stacktrace to source code;
- support for JsTestDriver configuration file: syntax and error highlighting, basic completion, navigation to referenced files;
- capturing messages sent to `console.log()`;
- support for Jasmine, QUnit and JsTestDriver built-in assertion frameworks:
    - quick-fixes for enabling global symbol reference resolving for each assertion framework (if you have QUnit or Jasmine tests in a project, you will be prompted to install the corresponding adapter);
    - contextual code generation actions (Alt+Insert) for creating new tests, setup and teardown methods for each assertion framework;
    - declarative HTML injection support for JsTestDriver built-in assertion framework.

Installation

Please visit Installation page.

Getting started

Please visit Getting Started page.

Releases & Changelog

You can check out the plugin page for more information about the releases.

Roadmap

You can find features that we are planning to implement in the future on the Roadmap page.

Issue tracker

If you've found a bug, a glitch or anything that doesn't work well, please file an issue in the WebStorm/PhpStorm project issue tracker (select **"Plugin: JsTestDriver"** subsystem when creating an issue).

Please don't file an issue in the http://code.google.com/p/js-test-driver/issues/list?q=label:IntelliJ. Those bugs are handled normally, but it's an old way.

For developers

If you are considering participation in the development of the plugin, or just going to build it yourself, please visit Development page.

js-test-driver - Plugins.wiki

Not Currently Ready!

please contact http://groups.google.com/group/js-test-driver for information about plugins.

Very pre-alpha.

js-test-driver - UsingTheEclipsePlugin.wiki

Introduction

The eclipse plugin for JS Test Driver allows you to enjoy all the benefits of JS Test Driver right from the comfort of Eclipse. This includes starting and stopping the server, capturing browsers, running and rerunning tests, filtering and viewing results.

How to install

In Eclipse, go to **Help** -> **Install new Software** (or whatever the similar menu item is in your version of eclipse.

Add the following url as an update site : **http://js-test-driver.googlecode.com/svn/update/**

Check the JS Test Driver checkbox and hit Next.

Hit Finish in the install details, accept any agreements and when Eclipse asks you to restart / apply changes, do so.

How to use the plugin

- **Displaying the view**

Go to Window -> Show View -> Other, and select JS Test Driver view from the list. This should bring up the view from where you can interact with it.

- **Global Preferences**

Go to Eclipse preferences (Either Window -> Preferences or Eclipse -> Preferences, depending on your OS). You should see a JS Test Driver menu item on the left. When you select that, you should see the following options

- **Port** : This is the port that the JS Test Driver Server will start on
- **Browser paths** : You can set the path to your browsers so that they can be launched from within eclipse itself.
- **Server Panel**

The top part of the JS Test Driver view is the Server info panel. The green play button is to start the server. Once you click on that, it should change to a stop button, and the NOT RUNNING text should change to the capture URL. You can either copy paste this URL in a browser to capture it as a slave, or click one of the browser icons to launch the browser (only if you have set the browser path in the global preferences) to automatically capture it.

Once you have a browser captured, the icons should turn lit, and the bar should turn from yellow to green. This means you are now ready to run the tests.

- **Running the tests**

There are a few ways to run the tests. The simplest way to run all your JS Test Driver tests is to go to Run Configurations and then add a new JS Test Driver configuration. Simply select your project and config file, and you are ready to run all your tests.

All other ways to run the tests require that you have a run configuration setup first. One you have done that, another way to run the tests once you have your browsers captured is to right click inside the text editor itself, and click Run as -> JS Test Driver Test. Inside the editor, it is context sensitive. By default, it will run all the tests inside that file. If you click on a line which has a test method definition, it will run just that test. If you select a bunch of lines, it will run only the tests in the highlighted portions.

The third way to run the tests is by selecting a js test file (or multiple) in the package and right clicking, selecting Run As -> JS Test Driver test. It will run all the tests in those files.

- **Test Results View**

Now that you have run your tests, you can see the results similar to JUnit in the test results tree. It is grouped by Browser, then by Test case and finally all the test results. You can see the failure message or any log statements you had in your tests by double clicking a particular test.

The icons at the top also allow you to interact with tests. The Refresh Icon refreshes all captured browsers so that it holds a clean state. The Rerun button reruns the last launch configuration, and does nothing if you haven't previously run anything. And the last blue and red cross button filters your test results to show only failed tests.

js-test-driver - CodeCoverage.wiki

JsTestDriverCoverage

Code coverage can be a valuable tool for gauging your code health. JsTestDriver makes it easy to generate code coverage for your JavaScript.

Installation

Coverage was developed as a plugin for JsTD. So, to enable it, download the coverage.jar and place it into the JsTestDriver plugins directory. Your directory structure should look like this: `/JsTestDriver.jar /plugins/coverage.jar` Setup: To enable coverage for a test configuration, add the following entry to the configuration after your files: `plugin: - name: "coverage" jar: "plugins/coverage.jar" module: "com.google.jstestdriver.coverage.CoverageModule"` This tells JsTD that it needs to add the plugin named coverage, from the module com.google.jstestdriver.coverage.CoverageModule that resides in the jar "plugins/coverage.jar".

For a working example, look at coverage.conf.

Running: Just run the test as usual. If there is no --testOutput flag defined, coverage will be reported for each file as a percentage, with an aggregate percent displayed. With a --

testOutput defined, line level coverage is recorded in the testOuput directory as -coverage.dat in the LCOV format.

Generating Report: The jsTestDriver.conf-coverage.dat is compatible with the LCOV (http://ltp.sourceforge.net/coverage/lcov.php) visualizer. After a successful coverage run, execute

```
genhtml jsTestDriver.conf-coverage.dat.
```

Further details are here: (http://ltp.sourceforge.net/coverage/lcov/genhtml.1.php)

js-test-driver - XUnitCompatibility.wiki

Introduction

The goal of the js-test-driver is to be the test runner for JavaScript testing. It is not our goal to be the assertion framework for the JavaScript testing, instead we hope that the open-source community will integrate a wide variety of assertion framework as plugins to js-test-driver.

Yahoo UI Test

- Config file which works with YUI Test
- http://code.google.com/p/js-test-driver/issues/detail?id=4

QUnit

- QUnitAdapter

Jasmine

- http://github.com/ibolmo/jasmine-jstd-adapter

Ruby AutoTest

- http://monket.net/blog/2009/06/autotest-and-js-test-driver/

Team City

- http://blogs.7digital.com/dev/2009/08/25/running-js-test-driver-in-team-city/

Other

- http://blog.james-carr.org/2009/06/16/more-test-driven-development-with-javascript-jstestdriver/
- http://www.rallydev.com/engblog/2009/09/24/testing-javascript-is-i-mean-should-be-easy/

JS Unit Mock Timeouts

See this previous thread discussing it: http://groups.google.com/group/js-test-driver/browse_thread/thread/1393946af440ed90?hl=en

Or check out this Google Testing Blog entry: http://googletesting.blogspot.com/2007/03/javascript-simulating-time-in-jsunit.html

js-test-driver - Proxy.wiki

Introduction

JsTestDriver has the ability to gateway unrecognized requests to servers-under-test. You may find this useful for larger integration tests that need to communicate with a backend server.

Details

To activate the gateway, add a gateway section to your jsTestDriver.conf file:

`gateway: - {matcher: "/matchedPath", server: "http://localhost:7000"} - {matcher: "/wildcardPath/*", server: "http://localhost:8000/"} - {matcher: "*", server: "http://localhost:9000"}`

The above configuration sends requests to `/matchedPath` along to the `http://localhost:7000`, requests to `/wildcardPath/{anything}` along to `http://localhost:8000/{anything}`, and any remaining requests to `{anything}` along to `http://localhost:9000/{anything}`.

The `gateway` entry of the configuration file is a list of matchers mapped to server addresses (including an optional path). When handling unknown requests, JsTestDriver iterates sequentially through the list of matchers, finds the first matching pattern, and forwards the request along to the server URL, appending any extra path information matched by a wildcard.

Matcher patterns come in three varieties: * Literal matchers, e.g. `/matchedPath` * Suffix matchers, e.g. `/wildcardPath/*` * Prefix matchers, e.g. `*.pdf`

Path Collisions

Sometimes your server-under-test may handle HTTP requests on URLs that JsTestDriver already handles. For instance, you may handle requests on `/cache` that are vital to your service and that you would like to test.

Use the following flag `--serverHandlerPrefix jstd` to prefix all JsTestDriver-specific request paths with `/jstd` so they won't collide with your service.

You have to use this flag when starting server and running tests both.

List of all folders used by jstd could be found here.

Security

Be sure you control or trust the servers that you enter into your `gateway` configuration entry. The gateway bypasses the browser's same-origin policy and forwards the requests almost verbatim.