Unit Testing

Servoy has built-in support for Unit Testing. The integration adds the following to the Servoy environment:

- A JSUnit node to the Solution Explorer, exposing the assert functions used in testcases
- An entry in the content menu of the Active Solution to run the Unit Tests
- A JSUnit view for viewing the results of a testrun

Creating testcases is as straightforward as creating a function with a name that is prefixed by 'test_':

```
function test_thisShouldPass() {
 jsunit.assertEquals('This test should pass', true, 5 < 10);
}

function test_thisShouldFail() {
 jsunit.assertEquals('This test should fail', true, 10 < 5);
}
```

The JSUnit node in the Solution Explorer provides easy access to the different supported assert functions

Testcases can be added on application level in the global scope of on form level in the form scope.

setUp & tearDown

Each scope can contain a setUp and/or tearDown function. The setUp function is called BEFORE running each testcase in the scope and the tearDown function is called AFTER running each test in the scope. The setUp and tearDown function allow the developer to create the right circumstances for testcases to run and cleanup afterwards. Note that the setUp() and tearDown() methods are called before and after EACH test methods, as each single test is supposed to be independant.

Example

```
function setUp() {
  //Code here to setup the environment for the testcases in this scope
}
function tearDown() {
  //Code here to cleanup the environment after running the testcases in this scope
}
```

Testing Asynchronous Code

When testing asynchronous code, for example a queued method using the queueMethod method of the Headless Client plugin with a notifyCallback method or a executeAsyncRequest on the HTTP plugin with a success/errorCallback and the UnitTest needs to test the result of the callback method, application. updateUI(Number:milliseconds) can be used inside a loop to wait for the callback to be invoked and test it's result.

Asynchronous UnitTest template

```
/**
 * @type {Number}
 */
var TIME_OUT = 1000

/**
 * @type {Number}
 */
var UPDATE_WAIT = 100

var callbackReceived = false

/**
 * @type {Object}
 */
var callbackRetval

function testLocalLinkCallback() {
        callbackReceived = false
        //Your code here that invoked something that used testCallback as callback method
        var it = 0
        while (!callbackReceived && it < TIME_OUT / UPDATE_WAIT) {
                application.updateUI(UPDATE_WAIT);
                it++
        }
        if (!callbackReceived) {
                jsunit.fail('callback not invoked within TIME_OUT period')
        } else {
                //Check the content of callbackRetval here using jsunit.assert*
        }
}

function testCallback() {
        callbackReceived = true
        callbackRetval = //Store whatever you need to complete your test in calbackRetval
```

assertEquals vs. assertSame

The jsunit object supports 2 methods that seem similar, but have a distinct difference: assertEquals and assertSame. The difference between these two function is the JavaScript compare operator they use:

- assertEquals uses the '==' equal operator
- assertSame uses the ' ===' strict equal operator

The difference between these two operators is that the equal operator does type conversion if the objects on both sides of the operator are of different types, while the strict equal operator does not.

Example

```
jsunit.assertEqual(1,'1') //true
jsunit.assertSame(1,'1') //false
```