The IoC Testing Framework

The IoC Testing Framework is written using our existing jqUnit wrapper for jQuery's QUnit as a base - the kind of basic unit testing supported by these base libraries is described on the page Writing JavaScript Unit Tests. The IoC Testing Framework is both written using Fluid's IoC system, as well as being designed to test components which are themselves written using IoC. This framework aims to extend our power to write tests in various directions at the same time. As well as creating an idiomatic way of writing *integration tests* addressed at realistic-sized chunks of applications, expressed as IoC component trees, the IoC testing framework also considerably eases the task of testing *complex event sequences* - that is, sequences of application state that are derived from an alternating conversation between user interaction and application response.

NOTE: The IoC Testing framework is only for integration testing requiring asynchrony, i.e. testing that involves either

1. user interaction via the DOM, or
2. AJAX requests.

If it does not, it is much better to use more simple techniques.

Integration testing with component trees

The concept of *context* in Infusion IoC is derived from the entire collection of components held in an IoC component tree. The behaviour of each component is potentially altered by all of the other components which are *in scope* from the site of the component under consideration - for a detailed guide to the operation of scope within Infusion IoC, please consult the page on Contexts. Therefore in order to test component behaviour in context, we need a testing system whose lifecycle (in particular, the lifecycle of setup and teardown common to all testing systems) is aligned with the lifecycle of component trees - as well as a testing system which enables testing directives to be referred to any components within the tree in an IoC-natural way.

Event sequence testing

The idiom to be used when binding event listeners which are responsible for *implementing* application behaviour is very different from that to be used when *testing* the application behaviour. Implementation listeners are typically bound permanently - that is, for the entire lifecycle of the component holding the listener. This is in order to make application behaviour as regular as possible, and in order to make it as easy as possible to reason about application behaviour in the absence of race conditions. However, when writing tests directed at an event stream, typically the behaviour required for the listener to each individual event in the sequence is different - since the testing assertion(s) held in the listener will be verifying a component state against required conditions which change with each successive event. Carried to the fullest extent, this typically results in convoluted, brittle code, holding deeply nested sequences of event binding and unbinding operations held within listeners to other events. We need a system which allows such assertions to be expressed declaratively, with this sequence flattened out into a linear list of JSON elements corresponding to each successive state in the event chain.

Discussion about the Testing Framework

The framework was designed over October-December 2012, with initial call for implementation on the fluid-work mailing list at October 31st, continuing over a sequence of community meetings, and including a summary of work in progress on December 5th. The overall goals for the testing framework were presented as these:

1. To facilitate the testing of demands blocks that may be issued by integrators against components deployed in a particular (complex) context
2. To automate and regularise the work of "setup" and "teardown" in complex integration scenarios, by deferring this to our standard IoC infrastructure
3. To simplify the often tortuous logic required when using the "nested callback style" to test a particular sequence of asynchronous requests and responses (via events) issued against a component with complex behaviour
4. To facilitate the reuse of testing code by allowing test fixtures to be aggregated into what are becoming the 2 standard forms for our delivery of implementation - a) pure JSON structures which can be freely interchanged and transformed, b) free functions with minimum dependence on context and lifecycle

How to Use the IoC Testing Framework

The IoC Testing framework requires the use of two new kinds of Fluid component, which are packaged as "grades" within the implementation in the file IoCTestUtils.js. In order to make use of the framework, the tester must derive their own component types from these grades, and assemble them into various component trees corresponding to the desired integration scenarios.
The first type of component corresponds to the overall root of the component tree under test - the *test environment*, defined in the grade `fluid.test.testEnvironment`. The children of this component correspond to the entire "application segment" (the context) under test - this may be as large (as an entire application) or as small (as a single component) as required in order to comprise the desired fixture. These children are intermixed with components of the second type, the *test fixtures*, derived from the grade `fluid.test.testCaseHolder`. These fixture components typically contain no implementation code, but are simply holders for declarative JSON configuration defining the sequence and structure of a group of test cases which are to be run.

Simple Example

This simple example shows the testing of a simple component, `fluid.tests.cat` which defines one method. Firstly we define the component under test:

```
/** Component under test **/
fluid.defaults("fluid.tests.cat", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
});
fluid.tests.cat.preInit = function (that) {
    that.makeSound = function () {
        return "meow";
    };
};
```

In order to test this single component, we embed it appropriately within a *testing environment*, derived from the grade `fluid.test.testEnvironment`, together with a component to hold the test fixtures named `fluid.tests.catTester`:

```
fluid.defaults("fluid.tests.myTestTree", {
    gradeNames: ["fluid.test.testEnvironment", "autoInit"],
    components: {
        cat: {       // instance of component under test
            type: "fluid.tests.cat"
        },
        catTester: { // instance of test fixtures
            type: "fluid.tests.catTester"
        }
    }
});
```

Finally we need to define the test fixture holder itself, `fluid.tests.catTester`, derived from `fluid.test.testCaseHolder`, as well as the test fixture code itself:

```
fluid.defaults("fluid.tests.catTester", {
    gradeNames: ["fluid.test.testCaseHolder", "autoInit"],
    modules: [ /* declarative specification of tests */ {
        name: "Cat test case",
        tests: [{
            expect: 1,
            name: "Test Global Meow",
            type: "test",
            func: "fluid.tests.globalCatTest",
            args: "{cat}"
        }
        ]
    }]
});

fluid.tests.globalCatTest = function (catt) {
    jqUnit.assertEquals("Sound", "meow", catt.makeSound());
};
```

The standard structure inside a `fluid.test.testCaseHolder` shows an outer layer of containment, `modules`, corresponding to a QUnit `module`, and then a entry named `tests`, holding an array of structures corresponding to a QUnit `testCase`. Here we define a single test case which holds a single *fixture record* which executes a global function, `fluid.tests.globalCatTest` which makes one jqUnit assertion. In cases where we apply *sequence testing* , the fixture record may instead hold an entry named `sequence` which holds an array of fixture records representing sequence points to be attained by the test case.

In order to run this test case, we can either simply construct an instance of the environment tree by calling `fluid.tests.myTestTree()`, or submit its name to the global driver function `fluid.test.runTests` as `fluid.test.runTests("fluid.tests.myTestTree")`. The latter method should be used when running multiple environments within the same file to ensure that their execution is properly serialised.

Supported fixture records

The IoC testing system currently supports the following 5 types of fixture record, which can be assigned to two categories - "executors", which actively trigger an action, and "binders" which register some form of listener in order to receive an event from the tree under test. These are recognised using a "duck typing system" similar to that used in the Fluid Renderer. These records may either form the complete payload for a test held in the `tests` section of a `TestCaseHolder`, or may appear as elements of an array held in its `sequence` member, representing a sequence of actions (either executors or binders) to be performed by the test case.

Fixture name	Field name	Field type	Field description	Fixture category

Function executor	func	Function/function name	function to be executed	executor
	args [optional]	Object/Array	arguments to be supplied to function	
Event listener	event	Fluid event firer	The event to which the listener will be bound	binder
	listener	Function/function name	The listener to be bound to the event	
	args	string/Object/Array	The set of arguments to be supplied to the listener. These may include IoC references to any values that are in scope, including `arguments` to refer to the original event arguments	
	listenerMaker	Function/function name	A function which will produce a listener to be bound	
	makerArgs [optional]	Object/Array	arguments to be supplied to the listener maker function in order to produce a listener	
	priority [optional]	String/Number	the priority order for this listener to be notified, relative to other listeners to the same event. String values may be `first` or `last`, or numeric values greater than 0 to represent higher priority than the default, or less than 0 for lower priority	
	namespace [optional]	String	A namespace to be applied to this listener when attached to its event. Only one listener with the same namespace may be attached to an event at one time	
Change event listener	changeEvent	Fluid event firer corresponding to a change event (currently `modelChanged`, `guards` or `postGuards`)	Change event to be listened to - ***NOTE*** **- this record forms an unstable API and its structure will be revised when the "Old ChangeApplier" is removed from the framework**	binder
	path	string	A path specification matching the EL paths for which the listener is to be registered, as per the ChangeApplier API	
	spec	Object	A record holding a structured description of the required listener properties, as per the ChangeApplier API. This may include a `priority` field as encoded for plain events above, but note its different position within the record.	
	listener	Function/function name	The listener to be bound to the event	
	listenerMaker	Function/function name	A function which will produce a listener to be bound	
	makerArgs [optional]	Object/Array	arguments to be supplied to the listener maker function in order to produce a listener	
jQuery event trigger	jQuery Trigger	string	The name of a jQuery event (jQuery eventType) to be triggered	executor
	args [optional]	Object/Array	additional arguments to be supplied to `jQuery.trigger`	
	element	jQueryable (DOM element, jQuery, or selector)	The jQuery object on which the event is to be triggered	
jQuery event binder	jQuery Bind	string	The name of a jQuery event for which a listener is to be registered	binder
	element	jQueryable (DOM element, jQuery, or selector)	The jQuery object on which a listener is to be bound	
	args [optional]	Object/Array	additional arguments to be supplied to `jQuery.one`	
	listener	Function/function name	The listener to be bound to the event	
	listenerMaker	Function/function name	A function which will produce a listener to be bound	
	makerArgs [optional]	Object/Array	arguments to be supplied to the listener maker function in order to produce a listener	

In each case in this table, the "type" field may be taken as comprising a string holding an IoC specification (context-qualified EL path) for the type in question. Fields highlighted in red and green rows are alternatives to each other - they may not be used simultaneously within the same fixture. The fields in light grey rows are the essential "duck typing fields" which define the type of the fixture records and are mandatory.

A More Complex Example

This example shows sequence testing of a component `fluid.tests.asyncTest` with genuine asynchronous behaviour (as well as synchronous event-driven behaviour). The component under the test is a Fluid Renderer component which renders a button, and a model-bound text entry field. The component defines a listener to clicks to the button which asynchronously (via `window.setTimeout`) fires to a Fluid event named `buttonClicked`. Separately, the component binds listeners to change events from the text field, which are corresponded with the standard ChangeApplier events resulting from corresponding changes to the component's model.

```
/** Component under test **/
fluid.defaults("fluid.tests.asyncTest", {
    gradeNames: ["fluid.rendererComponent", "autoInit"],
    model: {
        textValue: "initialValue"
    },
    selectors: {
        button: ".flc-async-button",
        textField: ".flc-async-text"
    },
    events: {
        buttonClicked: null
    },
    protoTree: {
        textField: "${textValue}",
        button: {
            decorators: {
                type: "fluid",
                func: "fluid.tests.buttonChild"
            }
        }
    }
});

fluid.defaults("fluid.tests.buttonChild", {
    gradeNames: ["fluid.viewComponent", "autoInit"],
    events: {
        buttonClicked: "{asyncTest}.events.buttonClicked"
    }
});

fluid.tests.buttonChild.postInit = function (that) {
    that.container.click(function() {
        setTimeout(that.events.buttonClicked.fire, 1);
    });
}
```

Just as with the simple cat testing example above, we embed this component together with a suitable `TestCaseHolder` within an overall `testEnvironment`:

```
fluid.defaults("fluid.tests.asyncTestTree", {
    gradeNames: ["fluid.test.testEnvironment", "autoInit"],
    markupFixture: ".flc-async-root",
    components: {
        asyncTest: {
            type: "fluid.tests.asyncTest",
            container: ".flc-async-root"
        },
        asyncTester: {
            type: "fluid.tests.asyncTester"
        }
    }
});
```

This environment shows use of the optional `markupFixture` property on the `testEnvironment`. Since the IoC testing framework operates setup/teardown on the unit of overall `testEnvironment`s, we cannot (should not) make use of QUnit's standard markup setup/teardown operated on the hard-wired DOM node with id `qunit-fixture`, which is on the unit of individual test cases. The `markupFixture` property is to be used where the overall environment makes use DOM material where its markup is rendered, which naturally should be reset to its original value between runs of different `testEnvironment`s. The `markupFixture` property holds any jQueryable value, designating the overall root node of this DOM material. After the `testEnvironment` has been torn down, the framework will reset the markup within this root to the contents it enjoyed before setup of the environment.

Finally, we show the contents of the associated `TestCaseHolder`. In this case, the 1 test it defines holds a `sequence` member prescribing a sequence of 11 states for the component, which run a total of 7 jqUnit assertions. These show records of all of the 5 types defined above - the framework ensure the correct sequence of activities (including binding and unbinding of listeners registered in `binder` records) is operated, as well as showing feedback in the QUnit UI relating to the sequence point reached by the system. This can be used to diagnose the last successfully reached sequence point in the case of a "hang" caused by an unexpectedly missing event in the sequence.

The sequence first initiates rendering of the overall component with a custom global function `fluid.tests.startRendering`, which checks that the component has rendered correctly and then initiates a click on the rendered button element. The sequence then checks for the expected asynchronous Fluid event - it then synthesises a further click on the button and checks for the same event again. It then synthesises an update to the rendered text field in the UI, and listens to the expected ChangeEvent generated by this update. It changes the field again to a different value and listens for the further ChangeEvent. Next, the sequence makes a direct call to a jqUnit assertion function to verify that the component's model has been updated properly. Finally, it returns to the button, directly simulating a click event using the `jQueryTrigger` fixture type, and listening to that event itself using the `jQueryBind` fixture type.

The TestCaseHolder makes reference to a few global utility functions which are reproduced below.

```
fluid.defaults("fluid.tests.asyncTester", {
    gradeNames: ["fluid.test.testCaseHolder", "autoInit"],
    newTextValue:     "newTextValue",
    furtherTextValue: "furtherTextValue",
    modules: [ {
        name: "Async test case",
        tests: [{
            name: "Rendering sequence",
            expect: 7,
            sequence: [ {
                func: "fluid.tests.startRendering",
                args: ["{asyncTest}", "{instantiator}"]
            }, {
                listener: "fluid.tests.checkEvent",
                event: "{asyncTest}.events.buttonClicked"
            }, { // manually click on the button
                jQueryTrigger: "click",
                element: "{asyncTest}.dom.button"
            }, {
                listener: "fluid.tests.checkEvent",
                event: "{asyncTest}.events.buttonClicked"
            }, { // Issue two requests via UI to change field, and check model update
                func: "fluid.tests.changeField",
                args: ["{asyncTest}.dom.textField", "{asyncTester}.options.newTextValue"]
            }, {
                listenerMaker: "fluid.tests.makeChangeChecker",
                args: ["{asyncTester}.options.newTextValue", "textValue"],
                path: "textValue",
                changeEvent: "{asyncTest}.applier.modelChanged"
            }, {
                func: "fluid.tests.changeField",
                args: ["{asyncTest}.dom.textField", "{asyncTester}.options.furtherTextValue"]
            }, {
                listenerMaker: "fluid.tests.makeChangeChecker",
                makerArgs: ["{asyncTester}.options.furtherTextValue", "textValue"],
                // alternate style for registering listener
                spec: {path: "textValue", priority: "last"},
                changeEvent: "{asyncTest}.applier.modelChanged"
            }, {
                func: "jqUnit.assertEquals",
                args: ["Model updated", "{asyncTester}.options.furtherTextValue",
                    "{asyncTest}.model.textValue"]
            }, { // manually click on the button a final time with direct listener
                jQueryTrigger: "click",
                element: "{asyncTest}.dom.button"
            }, {
                jQueryBind: "click",
                element: "{asyncTest}.dom.button",
                listener: "fluid.tests.checkEvent"
            }
            ]
        }
        ]
    }]
});
```

```
fluid.tests.checkEvent = function () {
    jqUnit.assert("Button event relayed");
};

fluid.tests.changeField = function (field, value) {
    field.val(value).change();
};

fluid.tests.makeChangeChecker = function (toCheck, path) {
    return function (newModel) {
        var newval = fluid.get(newModel, path);
        jqUnit.assertEquals("Expected model value " + toCheck + " at path " + path, toCheck , newval);
    };
}

fluid.tests.startRendering = function (asyncTest, instantiator) {
    asyncTest.refreshView();
    var decorators = fluid.renderer.getDecoratorComponents(asyncTest, instantiator);
    var decArray = fluid.values(decorators);
    jqUnit.assertEquals("Constructed one component", 1, decArray.length);
    asyncTest.locate("button").click();
};
```

It's anticipated that such lengthy fixture lists will in practice themselves be generated from other more suitable and concise forms in JSON structures, either by Fluid's Model Transformation system, or otherwise - as well as making better use of a growing library of standardised assertion and triggering functions such as `fluid.tests.changeField`.