Tips for Debugging

The current incarnation of the Infusion framework makes debugging quite difficult without a good knowledge of the principles of the framework and even a working knowledge of a few aspects of its source code. Work is underway to produce a set of integrated visual and non-visual debugging tools, which the declarative form of framework-enabled code will allow to be much more powerful than the source-level debugging we are familiar with. This is summarised in the following mailing list posting and the JIRA, FLUID-4884 linked there.

In the meantime, here are a few hints for users who find themselves trying to investigate peculiar behaviour or non-behaviour by the framework.

How framework ids are assigned

Every component (as well as many other framework-managed entities) in the component tree is assigned a permanent and stable id by the framework. This is done by the framework function fluid.allocateGuid - this allocates ids starting with a textual prefix, a hyphen, and finishing with an increasing integer. For example, an id of the form `a3z4bdwt-25` represents the 25th component (or other managed object) constructed by this instance of Infusion. The Infusion instance is identified by the prefix `a3z4bdwt` which is a string of alphanumeric characters assigned at random when the instance starts up. The part after the hyphen is deterministic and so is useful for consistently identifying component instances between runs. For example, setting conditional breakpoints in Firebug or the Chrome Dev Tools of the form `id.endsWith("-25")` is a useful debugging technique for stopping execution at a point in the workflow where a component you identified on a previous run has entered some interesting state.

Getting at the instantiator

The ***instantiator*** is the central record-keeping structure of the IoC system, and so is usually an enormous help in debugging when encountering a situation where the framework's understanding of a situation seems to disagree with yours. In current incarnations of the framework, this is very easy to come by, since the global instantiator holding records of all currently instantiated components is kept in the well-known global `fluid.globalInstantiator`. It's very helpful to permanently keep a "watch" expression in whatever debugger you are using pointed at this value.

What to see inside the instantiator

The most useful instantiator contents are the two giant lookup tables `pathToComponent` and `idToShadow`. The first can be used to quickly see the global layout of the entire component tree of which this component is a part - it is keyed by the EL path of each component with respect to the component root, and the values are the components themselves. The second is less generally useful to users, but is a lookup keyed by the component's `id` to its ***shadow record.*** This table is useful in cases where the framework has logged a diagnostic mentioning the id of a component - information about the component can be looked up by id here. Each component has such a bookkeeping record associated with it, most of the contents of which are not very helpful for debugging. However, generally useful members include `that`, holding a reference back to the original component, `path`, holding its EL path as appearing in `pathToComponent`, and `contextHash`, holding a hash of all of the context names of the component to `true`. Another useful member of the shadow is `injectedPaths` which lists all the paths in the component tree to which this component has been *injected*.

How to deal with timing issues during construction

The modern framework in general is "self-timed" - that is, it does not have well-defined and named lifecycle points that each component passes through in sequence. This implies that the resolution to every "race condition" encountered during construction is the same - which is to not to try to refer directly to component members using nested named in code (e.g. `that.memberA.memberB`) but instead to "hoist" such references into EL references appearing in the configuration which binds the component onto the global function in question. That is, for example, don't write this:

```
myNamespace.myRiskyFunction = function (that) {
    return that.memberA;
}

fluid.defaults("myNamespace.myComponent", {
....
    members: {
        myMember: "myNamespace.myRiskyFunction({that})"
});
```

but instead write this:

```
myNamespace.myBetterFunction = function (member) {
    return member;
}

fluid.defaults("myNamespace.myComponent", {
....
    members: {
        myMember: "myNamespace.myRiskyFunction({that}.memberA)"
});
```

This is a good practice in any case, since it restricts the visibility of unnecessary members of your component to the utility function, increasing the chance that it is reusable. By hoisting up these references, the framework will automatically take care of any construction timing issues, unless there is a hard cyclic dependency between members in which case it will terminate with a diagnostic.

How to read error reports

Every diagnostic reported by the framework passes through the central utility `fluid.fail` which supplies a standardised log record and workflow for these failures. The method of triggering reporting is a little peculiar, for historical reasons - many browsers (mostly IE) will tend to suppress the logging of failures (or suppress their capability of launching the debugger) if these are triggered by what could be called "soft exception" - for example of the form `throw new Error("message")`. Instead we trigger a hard exception by trying to access a nonexistent member of a `String` value - and in fact, the `String` value we choose is the one holding the message that we wish to log. This consistently and satisfyingly on every platform triggers a logged error as well as tripping the debugger if it happens to be open and appropriately configured. Unfortunately it does result in a slightly peculiar console message as a result of the invalid property access, of the form `Cannot access member "Assertion failure - check console for details" of object ....`

After this console message, if logging has been enabled by `fluid.setLogging(true)`, there will follow a "activity trace" through the framework's functions which are responsible for the activity which failed. This is in addition to the standard stack trace supplied by the JavaScript runtime itself. This takes the form of a series of lines beginning with the word "while" - here is an example:

```
ASSERTION FAILED: Failed to resolve reference {uiEnhancer}.applier - could not match context with name
uiEnhancer from component leaf Object { typeName="fluid.uiOptions.constructed", id="13z4bdwt-25", options=
{...}, more...}
Current activity:
    while resolving context value {uiEnhancer}.applier
    while constructing component of type fluid.uiOptions.constructed with arguments ["body"]
```

in general the top of the "while" stack indicates the most recent activity engaged by the framework, and the one which is most proximately responsible for the error. You can read down towards the bottom of the stack to discover the more and more distant causes for the framework's activity.