Component Lifecycle

This page outlines the sequence of events in the instantiation of a Fluid component. Some of these events are the responsibility of the runtime, some, of the user, and some, of the Fluid framework.

Setting up a Fluid component

Defining a component requires two steps:

1. Registering the component's default options with the framework by issuing a call to `fluid.defaults`.
2. Defining a suitably namespaced creator function which constructs the component.

These two actions may be performed in any order, since none of the code in the creator function will execute until the component is instantiated. As well as general default options, `fluid.defaults` also sets up the structure for particularly meaningful component configuration such as events, DOM binding selectors and subcomponents. This section of the process is covered by the Fluid Component API page.

On This Page

See Also

- How to Define a Unit
- Infusion Framework
- Fluid Component API
- Events for Component Developers

Actions within the creator function

In theory, code within the creator function itself is largely unconstrained. However, to take full advantage of the facilities of the Fluid Framework, there are some standard calls that should be made, of which the most important is a call to `fluid.initView`.

`fluid.initView`

A call to `fluid.initView` should be amongst the first, or perhaps the very first statement within the creator function. For example, the start of a creator function for `fluid.reorderer` might look as follows:

```
fluid.reorderer = function (container, options) {
    var that = fluid.initView("fluid.reorderer", container, options);
    options = that.options; // The live, fully merged options are available at that.options
```

`fluid.initView` takes care of a number of responsibilities for the user. Its overall workflow, in the abstract, is as follows:

1. Instantiate a new, empty Object to form the overall component `that`.
2. Initialise the component container:
    - Evaluate the `container` argument by performing a jQuery search, if necessary.
    - If the container does not represent a single unique DOM node, throw an error.
3. Initialise the component options:
    - Look up any default options that may have been stored previously using `fluid.default`
    - Merge the options specified by the user with a clone of the default options.
    - Attach the newly merged options structure as the member `options` of the top-level `that`.
4. Initialise the component events:
    - For each entry discovered in the `events` property of the resulting options structure, instantiate an event firer which will be attached to the newly instantiated component, available at the top-level that under the subobject `events`.

    - For each entry discovered in the `listeners` member of the resulting options structure, register the listener with the appropriate event object.
5. Initialise the DOM Binder:
    - Instantiate a new DOM Binder with the `selectors` property from the resulting options structure.
    - Fuse the binder's `locate` method onto the top-level that, and the DOM binder itself as the member `dom`.
6. Return the resulting initialised `that` object.

The stereotypical nature of this workflow provides a stable convention for the layout and function of a Fluid component, offloading a good deal of tedious checking and wiring code off the developer. After making this single, simple signatured API call, a developer is given an object initialised with a considerable amount of helpful functionality, laid out in standard convention.

`fluid.initSubcomponents`

If the component has a complex structure, composed of various independent parts (generally the case for a sufficiently mature component), these *subcomponents* may now be initialised by calls to the standard Infusion Framework function `fluid.initSubcomponents`. Similar to `fluid.initView`, this call delegates a considerable quantity of boring lookup, instantiation and wiring code to the framework. `fluid.initSubcomponents` and actually amounts to a mini-Inversion of Control system.

The set of subcomponents for a component are organised, from the point of view of the developer writing the top-level creator function, according to subcomponents of equal *instantiation signature*. That is, the caller of `initSubcomponents` specifies, along with a name identifying the *signature class* of the components, a number of *stereotypical arguments* that will be supplied to any subcomponents that are instantiated by this call. `initSubcomponents` returns an array of all of the components which were instantiated; their number and type is actually configured by the overall user of the component in the options structure, and so this decision is decoupled from code in the component implementor.

If there is, by design, just a single component in the signature class, a call to `initSubcomponent` may be made instead, which will return just one component.

For example, to continue with our Reorderer sample,

```
that.layoutHandler = fluid.initSubcomponent(thatReorderer,
    "layoutHandler", [container, options, dropManager, that.dom]);
```

This directive explains that there is one member of the signature class, which it has named `"layoutHandler"`. The single `layoutHandler` which the user has configured, will be instantiated with the arguments supplied in the argument list in the final position - you can see the reuse of the DOM Binder object `dom` that was created by `initView` earlier.

Mature life, for a component

The last line of a component creator function should read `return that;`. After this, the component is released "out into the wild", to live its life. In general, the framework has stepped out of the loop at this point, but the facilities it has endowed the component with will live on. For example, users of the component will know that listeners may be added and removed from the component at the position `that.events`, and DOM binding directives may be issued at `that.dom` etc.

There is no particular destruction semantics for a Fluid Component – since Javascript is a garbage-collected language, and does not really allow any form of resource usage other than DOM elements, this is generally appropriate.

A component may express a stronger contract by representing itself as model-bearing, which is described in Fluid Component API and Component Model Interactions and API. It is probable that a future version of the Infusion framework will assist this by supplying an initialisation directive `initModel` to accompany `initView`, in association with the ChangeApplier