DOM Binder

DOM Binder Overview

The purpose of the DOM Binder is to relax the coupling between a component and its markup. Whilst the Fluid framework is built on jQuery, and uses that framework's selector engine throughout to perform queries on the DOM, the extra level of indirection provided by the DOM Binder allows a complete separation of concerns. Fluid components are *parameterised* by a set of named selectors, managed by the DOM Binder instance attached to each top-level component. In this way, explicit selectors never appear in component code - leaving the components free of any baked dependence on markup structure.

There are other benefits to having DOM searches (via jQuery) managed in a central location - component authors can get access to fine-grained control over caching and lifetime of search results, which might otherwise become expensive if performed repeatedly - along with its basic `locate()` method, each DOM binder has a `fastLocate()` method which will not perform a DOM search if there is a cached result.

Components access elements in the DOM through unique selector names. The component defines default values for the selectors, but implementors are free to override the defaults if they need to change the structure of the markup.

On This Page

See Also

- DOM Binder API
- How to Define a Unit
- Fluid Component API

Still need help?

Join the infusion-users mailing list and ask your questions there.

How Infusion Components Use the DOM Binder

Each standard Fluid component has a DOM binder created automatically as a result of its call to the standard framework function `initView`. This call takes a set of options from the member `selectors` from the component's top-level options, and uses them to initialise the DOM binder.

Component developers declare the component's selectors in the defaults for the component:

```
fluid.defaults("fluid.newComponent", {
    selectors: {
        uiBit1: ".className1",
        uiBit2: ".className2"
    }
});
```

When the View is initialised with `fluid.initView()`, the DOM binder is created and attached to the top-level that(the component itself) as the member named `dom`. In addition, or convenience, the DOM Binder's `locate()` function is added as a top-level instance memember on the View. For example, to get a named element, you can simply call `that.locate(name)`) (see below for more information).

The other crucial configuration passed to the DOM binder is the overall container for the component - by convention, this is passed as the first argument to the component's constructor function (recall that the standard signature for a fluid component is `fluid.componentName(container, options)`. Unless they are otherwise qualified, all searches performed by the DOM binder attached to a particular component will be automatically scoped to its own container.

Using the DOM Binder

The component must use the DOM Binder for any access to the DOM elements that make up the component, using the `locate()` function:

```
that.locate(name, localContainer);
```

(For information about parameters, for this and other DOM Binder functions, see DOM Binder API.)

Other methods on the DOM Binder - caching

The other methods on the DOM Binder are less frequently used, and are not attached to the top-level that in the way that `locate()` is. They need to be accessed through the DOM Binder's own object, which by `initView` is available as `that.dom`.

```
that.dom.fastLocate(name, localContainer);
```

The signature and function of `fastLocate` are exactly the same as that for `locate`. The difference is that, if the results of the search are already present in the DOM binder's cache, they will be returned directly without searching the DOM again. This can be very much more rapid, but runs the risk of returning stale results.

The DOM binder's cache is populated for a query, whenever a query is submitted via `locate()`.

```
that.dom.clear()
```

The `clear()` method completely clears the cache for the DOM binder for all queries. It should be used whenever, for example, the container's markup is replaced completely, or otherwise is known to change in a wholesale way.

```
that.dom.refresh(names, localContainer);
```

The `refresh()` method refreshes the cache for one or more selector names, ready for subsequent calls to `fastLocate()`. It functions exactly as for a call to `locate()` except that

- The queried results are not returned to the user, but simply populated into the cache, and
- More than one selector name (as an array) may be sent to `refresh` rather than just a single one.

Example (Inline Edit)

The Inline Edit component requires three parts in its user interface:

- a field to display the text that can be edited
- a field that can actually edit the text
- a container for the edit field

The component declares selector names for these elements, and provides defaults, in its call to `fluid.defaults()`:

```
fluid.defaults("inlineEdit", {
    selectors: {
        text: ".text",
        editContainer: ".editContainer",
        edit: ".edit"
    },
    ....
```

Here, the default selectors use class names. To use these defaults, an implementer can simply attach these class names to the relevant elements in mark-up. Alternatively, the implementer may choose to override some or all of these selectors with other selectors. For example:

```
var myOpts = {
    selectors: {
        text: "#display-text",
        edit: "#edit-field"
    }
};
var myIEdit = fluid.inlineEdit(myContainer, myOpts);
```

In this example, the implementer is using element IDs to identify the text and edit fields. Because a custom `editContainer` is not included, it will default to the class declared by the component.

To access the DOM elements, the Inline Edit component uses its DOM Binder and the selector names:

```
that.viewEl = that.locate("text");
that.editContainer = that.locate("editContainer");
that.editField = that.locate("edit", that.editContainer);
```

In this way, the Inline Edit component is completely ignorant of the mark-up it is working with, or even the selectors used. When the markup changes, code doesn't break.