DOM Binder

DOM Binder Overview

The purpose of the DOM Binder is to relax the coupling between a component and its markup: A DOM Binder handles any interaction a component may have with its markup.

Any parts of a component's markup that may be accessed by the code is assigned a named selector using an option called `selectors`. The component accesses that markup through a request to the DOM Binder using the name, rather than directly. So instead of having selectors hard-coded, like this:

```
var button = jQuery(".button-classname");
```

the selector is given a name and the DOM Binder's `locate()` method is used, like this:

```
selectors: {
    button: ".button-classname"
}
...
var button = that.locate("button");
```

This allows integrators to use whatever selectors they want in the markup (and simply specify the new selector in the `selectors` option);
the component accesses the DOM through the names, not directly through selectors. 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.

The DOM Binder also ensures that a component accesses only markup specific to itself, and not to any other similar components that might be on the same page. Each Infusion View Component is scoped to a particular node in the DOM called its ***container***. The DOM Binder limits any queries to that container.

The DOM Binder also caches information, allowing efficient access for searches that are performed very frequently on material that is not changing - for example within mouse event loops. Caching means these searches are do not have to be recomputed from the DOM on every query.

On This Page

See Also

- DOM Binder API
- Options for View Components

Still need help?

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

How Infusion Components Use the DOM Binder

The Infusion Framework automatically creates a DOM Binder for any View Component as it constructs and attaches it to the component as a top-level member named `dom`. View Components specify a set of names selectors in the component's defaults called `selectors:`

```
fluid.defaults("fluid.newComponent", {
    gradeNames: ["fluid.viewComponent", "autoInit"],
    selectors: {
        uiBit1: ".className1",
        uiBit2: ".className2"
    }
});
```

DOM elements related to this component can be resolved declaratively using the `dom` member. For example, if {`newComponent`}is a reference to an instance of the above component, a reference to one of its DOM binder elements as declared above could be written as {`newComponent`}`.dom.uiBit1.`

The full programmatic API to the DOM binder also available through this `dom` member (see DOM Binder API for information about available methods). For convenience, the DOM Binder's `locate()` function is also added to the component as a top-level instance member.

Unless they are otherwise qualified, all searches performed by the DOM binder attached to a particular component will be automatically scoped to the component's own container.

Using the DOM Binder Declaratively

The preferred way of using the DOM binder to access a component's DOM elements is through declarative configuration, using the form "{`<componentRef>`}`.dom.<selectorName>`". These references typically occur in the component's default configuration specification, but references may also be made from any other component.

In the following example, the component's default configuration assigns a DOM element to another top-level component member using the `members` option:

```
fluid.defaults("fluid.tutorials.buttonHolder", {
    gradeNames: ["fluid.viewComponent", "autoInit"],
    selectors: {
        button: ".button"
    },
    members: {
        button: "{that}.dom.button"
    }
});
```

Declarative configuration is also the preferred approach for supplying arguments to Invokers and Event Listeners, as shown in the following example:

```
fluid.defaults("fluid.tutorials.buttonHolder", {
    gradeNames: ["fluid.viewComponent", "autoInit"],
    selectors: {
        status: ".holder-status",
        indicator: ".holder-ind"
    },
    events: {
        onSelect: null
    },
    listeners: {
        onSelect: {
            funcName: "fluid.tutorials.buttonHolder.selectHandler",
            args: ["{that}.dom.indicator", "arguments.0"]
        }
    },
    invokers: {
        highlightStatus: {
            funcName: "fluid.tutorials.buttonHolder.highlight",
            args: ["{that}.dom.status"]
        }
    }
});
```

Using the DOM Binder Programmatically

Standard programmatic access to the DOM binder is available using the `locate()` function:

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

The `locate()` method retrieves the specified DOM node by querying the DOM.

(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 component in the way that `locate()` is. They need to be accessed through the DOM Binder's own object available as `that.dom`.

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

The `fastLocate()` method retrieves the DOM node from the DOM Binder's cache instead of querying the DOM directly: When `fastLocate()` is used instead of `locate()`, 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);
```

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("fluid.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 supplied, it will default to the selector `.editContainer` 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 markup it is working with, or even the selectors used. When the markup changes, code doesn't break.