Basic Component Creation - Little Components

Previous: Pick a component type Up to overview **Next:** Tutorial - Model Components

On This Page

See Also

Component Grades
fluid.defaults
Options Merging
IoC - Inversion of Control
Component Lifecycle

Regardless of which grade of component you use, the basic structure will be the same. We'll use the simplest grade, a **little component**, to illustrate what this structure is. In future pages explaining other grades, you'll see the same principles.

The definition of a component involves two things:

1. declare the component **grade** and any default values for the component's **options**. Options are used by integrators to customize the behaviour of a component.
2. define any public **functions** that the component requires to do its work.

Grade and Default Options

A component's grade and any default options are registered with the framework using a call to `fluid.defaults`, which has two parameters: the component name and an object containing the defaults. The parent grades for the component are specified in an array in the defaults called `gradeNames`. For a **little component**, specify the grade as `fluid.littleComponent`:

```
fluid.defaults("tutorials.simpleComponent", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    option1: "default value 1",
    ...
});
```

Options

Integrators can override your defaults when they instantiate the component, to customize its appearance or behaviour. The Framework will take care of merging the integrator's values with your defaults.

We'll go through some examples of options, to give you an idea of what they're all about.

Example: Currency Converter Options

Suppose you're creating a currency converter. You might wish to specify a default conversion rate:

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    exchangeRate: 1.035
});
```

Example: Inline Edit

The Infusion Inline Edit component uses a tooltip and defines (among other things) defaults for the delay before the tooltip appears, the text to display - even whether or not to enable it at all:

```
fluid.defaults("fluid.inlineEdit", {
    ...
    useTooltip: true,
    tooltipText: "Select or press Enter to edit",
    tooltipDelay: 1000, // in milliseconds
    ...
});
```

Example: Progress

The Infusion Progress component uses jQuery animations to show and hide a progress bar. The defaults include objects that are passed to jQuery to configure the animations:

```
fluid.defaults("fluid.progress", {
    ...
    showAnimation: {
        params: {
            opacity: "show"
        },
        duration: "slow"
    }, // forwarded to $().fadeIn("slow")

    hideAnimation: {
        params: {
            opacity: "hide"
        },
        duration: "slow"
    }, // forwarded to $().fadeOut("slow")
    ...
});
```

The Creator Function

All components have a *creator function*: a public function that is invoked to instantiate the component. In general, the framework will instantiate the creator function for you automatically, given the component's default options. The framework will in general also take responsibility for calling the creator function for you automatically as well, when your component is registered as a subcomponent of another. In the rare case you need to construct a component directly using a JavaScript function call, Infusion components have a standardized function signature:

- **little**, **evented** and **model** components accept a single argument: `options`
- **view** and **renderer** components accept two arguments: `container` and `options`

(We'll get into what these arguments are soon.)

Creator functions can be defined in one of two ways

1. using IoC - Inversion of Control: The framework will create the function according to your specifications
2. directly: You write the function yourself - this option is not recommended and the ability to do this will be removed in Infusion 2.0

Using IoC

Automatic creator function generation

The IoC - Inversion of Control system can automatically generate a component creator function for you. This is accomplished by added a special grade to the `gradeNames` property: "autoInit":

```
fluid.defaults("tutorials.simpleComponent", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    option1: "default value 1",
    ...
});
```

Note that in Infusion 2.0, "autoInit" will become the default for all components and will not need to be specified.

Public API methods

The standard means of adding public API functions to a component is to express them as Invokers. An invoker is a declarative record added into a components defaults, under the section `invokers`: the name of the record becomes the name of the public function which will be added. The invoker record defines the name of the public JavaScript function which should be executed when the method is called, as well as details of where the arguments that the function accepts should be sourced from - for example:

```
fluid.defaults("tutorials.simpleComponent", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    option1: "default value 1",
    ...
    invokers: {
        publicFunction: {
            funcName: "tutorials.simpleComponent.publicFunction",
            args: "{that}"
        }
    }
});

// implementation of the public function registered as an invoker above
tutorials.simpleComponent.publicFunction = function (that) {
    ...
};
```

You will note that the function `tutorials.simpleComponent.publicFunction` is a standard JavaScript function that could even be invoked directly from code if this were found relevant - it need not be necessarily bound as a component method (although most component methods tend not to make sense without being provided an instance of the relevant component).

Example: Currency Converter via IoC

So what would our currency converter example look like, create using IoC:

```
fluid.defaults("tutorials.currencyConverterAuto", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    exchangeRate: 1.035,
    invokers: {
        convert: {
            funcName: "tutorials.currencyConverterAuto.convert",
            args: ["{that}.options.exchangeRate", "{arguments}.0"] // amount
            }
        }
    }
});

// The conversion function
tutorials.currencyConverterAuto.convert = function (exchangeRate, amount) {
    return amount * that.options.exchangeRate;
};
```

You'll notice that in this case we have been able to avoid binding to the entire component instance in our public function, and so our standalone public function `tutorials.currencyConverterAuto.convert` is indeed of more general utility than just for building a component method. This has happened because its responsibilities are particularly well-defined - you should always take the opportunity to restrict the binding behaviour of your public functions in this way whenever it is appropriate.

Writing your own creator function

Note that the scheme described here is not recommended for new code, and is described here only for completeness. The ability for users to write their component creator functions directly in JavaScript code will be removed in Infusion 2.0.

Creator functions follow a few basic steps:

1. Create an object called `that` by calling the appropriate Framework component initialization function
2. Attach things to the object and otherwise initialize the component
3. Return the object

Here's what that would look like for a **little component**:

```
// The global namespace
var tutorial = tutorial || {};

(function ($, fluid) {

    // a creator function for a little component
    // creator functions are typically named by the component name itself
    tutorials.sampleComponent = function (options) {
        // call the framework component initialization function
        var that = fluid.initLittleComponent("tutorials.sampleComponent", options);

        // attach any public methods to the 'that' object
        that.publicFunction = function () {
            // ...
        };

        return that;
    };

})(jQuery, fluid_1_5);
```

Example: Currency Converter Creator Function

What would this look like for our currency converter example?

```
// creator function for the currency converter component
tutorials.currencyConverter = function (options) {
    var that = fluid.initLittleComponent("tutorials.currencyConverter", options);

    // note that component methods have access to the values stored in 'options'
    // - the ones provided in the defaults and possibly overriden by implementors
    that.convert = function (amount) {
        return amount * that.options.exchangeRate;
    }
    return that;
};
```

Previous: Pick a component type Up to overview **Next:** Tutorial - Model Components