Subcomponent Declaration

In the Infusion IoC system, a component declares its (static) subcomponents through the `components` property of the defaults, using `fluid.defaults`:

```
fluid.defaults("my.component.name", {
    ...
    components: {
        subcomponent1: <subcomponent declaration>,
        subcomponent2: <subcomponent declaration>,
        subcomponent3: <subcomponent declaration>,
    },
    ....
});
```

Note that as with the use of dynamic grades, what could be called a kind of "dynamic subcomponent" can be added after the fact to any component simply by arranging for additional entries in its `components` options, via any of the usual routes - for example, direct arguments to its creator function, options distributed by distributeOptions, or by 2nd-level nested `components` entries in the subcomponent record of the defaults of a component grandparent. Later on in this section we will see direct framework facilities for other kinds of dynamic subcomponents, those driven by dynamic data or event firing.

Basic Subcomponent Declaration

The subcomponent declaration has the following form, holding the ***subcomponent record*** as the value corresponding to the key holding the subcomponent's ***member name*** (in this case `subcomponent1`):

```
fluid.defaults("my.component.name", {
    ...
    components: {
        subcomponent1: {
            type: "type.name"
            options: {...},
        }
    },
    ....
});
```

The properties allowed at top level in the subcomponent record are as follows:

Property	Type	Description	Example
`type`	String	This is the grade name of the type of subcomponent to be created. May be an IoC Reference.	`subcomponent1: { type: "my.component.gradename" }`

`options` (Optional)	Object	These are options to be passed to the subcomponent as "user options." Note that these are not the default options for the subcomponent, rather these options override the defaults. The defaults for the component will have already been registered by the `fluid.defaults` call(s) appropriate for its type and grade names.	`subcomponent1: { type: "fluid.mySubcomponent", options: { myOptions: "{name}.options.someOption", ... } }`
`createOnEvent` (Optional)	String	Specifies an event that will trigger the creation of the subcomponent. This option is used when the subcomponent should not be created immediately as part of the construction process of its parent, but rather at a later time signalled by the firing of the specified event. If this value is a simple string, it represents an event held on the parent component - it may also take the form of an IoC reference to an event elsewhere in the component tree. Note that if the specified event fires multiple times, the corresponding component will be destroyed and then recreated on every firing of the event after the first time.	`subcomponent1: { type: "fluid.mySubcomponent", createOnEvent: "someEvent" }`
`priority` (Optional)	Number or "first" /"last"	Specifies the order priority of the creation of the particular subcomponent. During component tree instantiation, the framework will sort the collection of subcomponents based on the priority specified. *Note that use of this option should be discouraged as it leads to fragile configuration. If you find yourself using it, please report the instance to the development team to see if a better solution can be found.*	`subcomponent1: { type: "fluid.mySubcomponent", priority: "first" }`
`container` (Required only for View components)	String	This property is a CSS-style selector identifying the container element to be used for this subcomponent. This property is required for any View components.	

Injected Subcomponent Declaration

Note that the entire subcomponent record may be replaced by a simple IoC reference to a component held elsewhere in the component tree. In this case the subcomponent is known as an ***injected component*** - the already existing component reference is simply copied into the parent component's member field. Note that as a result of growing power in the IoC framework to perform most actions "in place" with respect to an existing component in its original location without having to inject it elsewhere. Should you find yourself using injected components, it is worth flagging the issue to the development team to see if a better solution could be found.

Examples

```
fluid.defaults("cspace.admin", {
    gradeNames: ["fluid.rendererComponent", "autoInit"],
    components: {
        adminRecordEditor: { // view subcomponent declaration
            type: "cspace.recordEditor",
            container: "{admin}.dom.recordEditor",
            options: {
                csid: "{admin}.selectedRecordCsid"
            }
        }
    }
});
```

```
fluid.defaults("cspace.admin.showAddButton", {
    gradeNames: ["autoInit", "fluid.modelComponent"],
    components: { // injected component declaration
        permissionsResolver: "{permissionsResolver}"
    }
});
```

```
fluid.defaults("gpii.explorationTool.modelTransformer", {
    gradeNames: ["fluid.modelComponent", "fluid.uiOptions.modelRelay", "autoInit"],
    components: {
        highContrast: { // complex subcomponent declaration with priority and createOnEvent
            type: "gpii.explorationTool.panels.highContrast",
            container: "{uiOptions}.dom.highContrast",
            createOnEvent: "{uiOptions}.events.onUIOptionsMarkupReady",
            priority: "first",
            options: {
                resources: {
                    template: "{templateLoader}.resources.highContrast"
                }
            }
        }
    }
});
```

Dynamic components

A powerful facility known as ***dynamic (sub)components*** allows you to direct the framework to construct a number of subcomponents whose number is not known in advance from a template subcomponent record. There are two principal varieties of dynamic components. The first requires the existence of a ***source array*** for the construction - at run-time, the framework will inspect the array you refer to and construct one component from your template for each element of the array. The components which get constructed in this way can each be contextualised by both the contents of the corresponding array element as well as its index. The second requires the existence of a ***source event*** for the construction. The framework will construct one subcomponent for each firing of the event - the constructed component can be contextualised by the arguments that the event was fired with.

Both of these schemes make use of a special top-level area in a component's options, entitled `dynamicComponents`. The structure of this area is almost identical to the standard `components` area described above, with a few differences described in the dedicated subsections below.

Naming of dynamic components

The actual member names given to dynamic components follows a very straightforward scheme. The very first such component created will have the same name as the `dynamicComponents` record entry. Subsequent such components will have the name <key>-<n> where <key> represents the record entry name and <n> holds an integer, initially with value 1, which will increment for each further dynamic component constructed using the record. In practice you should not use this information to "go looking" for dynamic components, but instead should expect to observe their effects by some scheme such as injecting events down into them to which they register listeners, or broadcasting listeners down into them by use of distributeOptions or dynamic grades.

Future evolution of dynamic components

Although this facility is powerful, the reader will note the peculiar asymmetry in the construction process - the framework may be directed to construct these components in a declarative way, but they may only be destroyed procedurally through a call to the component's `destroy()` function. An improved and properly symmetric version of this facility will be delivered as part of work on the new Fluid Renderer as described by FLUID-5047 and related JIRAs, and the system described here will be withdrawn, as with previous "bridging technologies" such as the initFunction system.

Dynamic subcomponents with a source array

This scheme for declaring a dynamic component is announced by making use of the `sources` entry at top-level in the dynamic component's component record. The following defaults block defines a component which in practice will instantiate two subcomponents, one for each element in the array `values` that it declares in its own options:

```
fluid.defaults("examples.dynamicComponentRoot", {
    gradeNames: ["fluid.littleComponent", "autoInit"],
    values: [2, 3],
    dynamicComponents: {
        dynamic: {
            sources: "{that}.options.values",
            type: "fluid.littleComponent",
            options: {
                source: "{source}"
            }
        }
    }
});

var that = examples.dynamicComponentRoot();
var firstValue = that.dynamic.options.source; // 2
var secondValue = that["dynamic-1"].options.source; // 3
```

The `sources` entry will be expanded in the context of the parent component, and must hold a reference to an array. Within the configuration for the dynamic component, two special IoC context names are available. One is named **`{source}`** and holds a reference to the particular array element which was used to expand the record into a component - in the above example, successively the values 2, and 3. The other is named **`{sourcePath}`** and holds a reference to the array index which was used - in the above example, successively the values 0 and 1.

Dynamic subcomponents with a source event

The use of this scheme for dynamic components is announced by using the standard `createOnEvent` top-level member that we met earlier when writing standard `components` subcomponent blocks. The syntax is the same, but the semantic is different. For a standard subcomponent, `createOnEvent` will destroy and then recreate a component ***at the same path*** on each firing of the specified event. In contrast, for a dynamic subcomponent, createOnEvent will construct a ***fresh subcomponent*** at successive different paths on each firing of the event. The naming of these paths is described in the previous section but in practice should not be a concern for the user.

```
fluid.defaults("examples.dynamicEventRoot", {
    gradeNames: ["fluid.eventComponent", "autoInit"],
    events: {
        creationEvent: null
    },
    dynamicComponents: {
        dynamic: {
            createOnEvent: "creationEvent",
            type: "fluid.littleComponent",
            options: {
                argument: "{arguments}.0"
            }
        }
    }
});

var that = examples.dynamicEventRoot();
that.events.creationEvent.fire(2);
var firstValue = that.dynamic.options.argument; // 2
that.events.creationEvent.fire(3);
var secondValue = that["dynamic-1"].options.argument; // 3
```

In this case, rather than exposing the special context names `{source}` and `{sourcePath}` as with array-sourced dynamic components, the configuration for the dynamic components block instead just exposes the more standard context name **`{arguments}`** which we have seen used both with invokers and event listeners. In this case, the context name `{arguments}` is bound onto the argument list that was used to fire the event which triggered the creation of the particular dynamic subcomponent. The example shows the argument list successively holding the value `[2]` and then the value `[3]`.

Note that with this scheme, it is quite likely that the user will want to arrange for the destruction of the dynamic subcomponents at some time earlier than the natural destruction time of their parent and all its children. Using this scheme, they must arrange to do so using procedural code which manually schedules a call to the `destroy()` method of the dynamic subcomponent they want destroyed. As we observe above, this awkwardness will be removed when the dynamicComponents facility is replaced in a future revision of the framework that makes more powerful use of the lensing capabilities of the Model Transformations system.