Renderer Decorators

Overview

Decorators allow users of the Renderer to attach various things, such as functions, class names, etc., to the components at render time. A number of different types of decorators are currently supported.

Using Decorators

To use a decorator, include it in the component tree for the component in question, using the `decorators` field. This field contains an array of objects providing configuration information for the desired decorators. The contents of each object will vary based on the decorator type. For example, the `addClass` decorator will specify a string of class names, the `jQuery` decorator will specify a function name and a list of parameters.

Decorators are specified using a notation similar to that of Subcomponents in an `options` structure. They include a `type` field and whatever other fields are necessary, based on the type:

```
{
  ID: "my-id",
  value: aValue,
  decorators: [
      {type: "typeName",
       field: value,
       field: value
      }
  ]
}
```

On This Page

See Also

- Renderer
- Renderer Data Binding
- Renderer Component Trees
- Renderer API
- Fluid Renderer - Background

Still need help?

Join the fluid-talk mailing list and ask your questions there.

Supported Decorators

The following table provides an overview of the currently-supported decorators. The sections that follow discuss each decorator in turn.

Decorator type	Field name	Field type	Field Description	Example
jQuery / $	func	string	jQuery function to be invoked	decorators: [{type: "jQuery", func: "click", args: function() { $(this).hide(); } }]
	args	(array of) object	Arguments to the jQuery function	
addClass	classes	string	Space-separated list of CSS class names	decorators: [{type: "addClass", classes: "fl-widget fl-centred" }]
removeClass	classes	string	Space-separated list of CSS class names	decorators: [{type: "removeClass", classes: "fl-hidden" }]
fluid	func	string	global function name to be invoked	decorators: [{ type: "fluid", func: "fluid.componentName", container: container, options: options }]
	container	jQuery-able	Designator for the container node at which to base the component	
	options	free Object	Configuration options for the component	
	args	Array	raw argument list to override container and options	
identify	key	string	The key, or nickname for the decorated node - its allocated id will be stored in idMap under this key	decorators: [{type: "identify", key: "mySpecialName" }]
attrs	attributes	object	The attribute map to be applied to the rendered node	decorators: [{type: "attrs", attributes: "" }]
event	event	string	Name of event handler to be bound	decorators: [{type: "event", event: "click", handler: myHandler }]
	handler	function	Handler function to be bound	

jQuery decorator

Perhaps the most frequently used decorator is the jQuery decorator. This will accept any jQuery function and its arguments, and invoke that function, as the rendered markup is placed into the document. Here is an example of specifying a UILink component together with a jQuery-bound `click()` handler:

```
decorators: [
    {type: "jQuery",
     func: "click",
     args: function() { $(this).hide(); }
    }]
```

Any number of decorators of any types could be accommodated in the `decorators` list.

An alternative name for the jQuery decorator is `$` - this can be used interchangeably for `jQuery` as a type name.

addClass decorator

The `addClass` decorator allows a CSS class to be attached to the rendered node. It has just one argument/member, which is a space-separated list of CSS classes in just the same form that would have been accepted by `jQuery.addClass`.

Here is a simple component which has been decorated with two CSS classes:

```
{ID: "my-menu",
 value: "Cheeses",
 decorators: {
    type: "addClass",
    classes: "fl-listmenu fl-activemenu"
    }
  }
```

removeClass decorator (New in v1.1)

The `removeClass` decorator allows a CSS class to be removed from the rendered node. It has just one argument/member, which is a space-separated list of CSS classes. It is identical in syntax to the `addClass` decorator, but opposite in function.

Here is a simple component for which we will remove a CSS class:

```
{ID: "my-menu",
 value: "Cheeses",
 decorators: {
    type: "removeClass",
    classes: "fl-listmenu"
    }
  }
```

identify decorator

Useful in more intricate scenarios, where the rendered nodes need to be easily and quickly retrievable, perhaps where events bound to one node need to manipulate another, or when nodes are part of a wider relation, such as table cells and their headers. The model behind the `identify` decorator, is that the node is given a free "nickname" by the user, by which its final HTML id, and hence the node itself, can be quickly looked up later. This works in conjunction with a lookup table named `idMap` which is passed in the #options structure to the renderer driver. As rendering progresses, the final HTML id allocated to the node is stored in `idMap` under the key provided to the `identify` decorator.

Here is a short sequence showing a possible use of `identify`:

```
  var tree = {
      ID: "data-row:",
      decorators: {
          identify: "this-catt",
          }
      };
  var idMap = {};
  fluid.selfRender($(".paged-content-2"), tree, {idMap: idMap});
  fluid.jById(idMap["this-catt"]).show();
```

Whilst component tree nodes are allocated a `fullID` in a regular way by a stable algorithm involving their `ID` values and structure, this may not always relate them in a stable way in the global document - firstly, trees may be processed and reaggregated, which might change their ID or containment structure, and secondly, they may come to collide with already existing IDs in the document and hence come to be relabelled further. The "identify nickname" system lets developers to get at exactly the nodes they are interested in, in a simple, stable and efficient way.

fluid decorator (New in v1.1)

This is a highly powerful decorator, that completes the "active" functionality supplied by the `jQuery` and `identify` decorators. Use of the `fluid` decorator allows any Fluid Component to be scheduled to be instantiated against the rendered markup, as it is added to the target document. These decorators promote markup agnosticism, as well as developer efficiency – without them, one would be left to rescan the just-rendered markup once again, in order to convert it from raw markup to an active interface. With these decorators and the component tree, one has a surface with which to talk about the **function** of the interface whilst leaving design and markup issues in their own space.

The full form of the decorator takes three members, `func`, `container` and `args`, mirroring the instantiation syntax of a standard Fluid Component - as described in Fluid Component API, this takes the form

```
fluid.componentName = function(container, options);
```

In this case, the equivalent decorator instantiation takes the form:

```
{decorators: {
   type: "fluid",
   func: "fluid.componentName",
   container: container,
   options: options
   }
}
```

Note that rather than specifying `container` and `options` separately, one can instead set the member `args` to consist of the entire argument list - this might be useful for instantiating a non-Fluid component that does not conform to the general syntax. For example, the decorator above could be given a member `args: [container, options]`. The `args` member takes precendence if specified.

There is no specially dehydrated form for the `fluid` decorator – however, like all renderer decorators it may be dehydrated to the extent of having its `type` field folded onto a key field on `decorators` if there is just one decorator of a particular type.

attrs decorator

The `attrs` director is simple and crude - it allows freeform access to all of the attributes of the rendered node. Since this is not likely to result in a very markup-agnostic relationship, its use is only recommended in special situations. Its direct equivalent on the server-side was UIFreeAttributeDecorator. The decorator takes a value named `attributes` which is a free hash of keys to values, which will be applied "on top of" the target node as it is rendered, overwriting any values which were inherited from the original markup template.

```
{ID: "component-names",
 value: "Reorderer",
 decorators: {
    attrs: {title: "Reorderer Component"}
    }
  }
```

New in v1.1: Specifying a value of "`null`" will remove the attribute.

event decorator

The final implemented decorator, `event`, allows direct access to the functionality of binding a raw browser event to the rendered node. This is not generally recommended, since this is more safely and portably achieved using `jQuery`. However, it is possible this might be a useful function in some special situation. The decorator has a member called `handler` which is directly assigned to be the native event handler for the event named `event`.

For example, this decorator:

```
decorators: {
    type: "event",
    event: "onClick",
    handler: function() {alert("You are using some grubby browser-level functionality");}
}
```

This could be attached to bind (and hence overwrite) the onClick handler for the target rendered node. Don't do this at home.