Tutorial - Renderer Components

Previous: Tutorial - View Components Up to overview **Next:** Tutorial - Subcomponents

On This Page

See Also

Renderer
Renderer Component Trees
Cutpoints

In the previous Tutorial - View Components currency converter example, you can notice two things:

- The list of currencies is duplicated: It is present in the HTML as well as in the model.
- Event listeners were attached to the UI elements to update the model when the controls were changed.

Both of these things can be avoided by using a **renderer component**, which is a view component with the addition of the Infusion Renderer.

Note that the direct use of the Infusion Renderer as described here will be withdrawn in the Infusion 2.0 release. Currently the renderer is in transition to being rewritten as a standard part of the IoC framework rather than requiring special JSON configuration in the form of renderer component trees.

The Renderer will populate an HTML template with the contents of a data model. In the case of the currency converter, this means that the currency list can be specified only in the data model, and an empty HTML `<select>` element will be populated with the data from the model, eliminating the duplication.

The Renderer can also bind the HTML to the data model, so that when users of the currency converter select a different currency or enter a different amount, the content of the data model will *automatically* be updated without any effort on your part.

Renderer Component Grade

To use the Renderer with your component, you need to use the **rendererComponent** grade. To do this:

- specify a grade of `fluid.rendererComponent`, and
- provide the following:
  - a data model
  - an HTML template
  - a component tree
  - selectors

Each of these is discussed below.

Data Model

A renderer component supports a model (just as Tutorial - Model Components and Tutorial - View Components do). The model of a renderer component contains any data that is to be rendered by the component.

Templates

A Renderer Template is an HTML file or snippet that provides guidance to the Renderer regarding how to render the data. The template can be provided in one of two ways:

- it can simply be found in the HTML document itself, or
- it can be loaded from a separate file.

Component Trees

A renderer component tree is an abstract representation of how the data is to be rendered. A *renderer component* (not to be confused with Infusion Components) is a JavaScript object that represent the contents and data binding function of a view, separate from any particular rendering of it.

The simplest way to specify a component tree is to create a function that returns the tree and specify the function name using the `produceTree` option (see the example below).

For more detailed information, see Renderer Component Trees

Selectors and Cutpoints

The Infusion Renderer provides a clean separation between model and view; the HTML template and the component tree are relatively independent. For example, if your UI is going to allow users to select from a number of choices, your component tree will define the choices, but will be independent of whether the selection is rendered using checkboxes, a pull-down list, or some other method.

The mapping between the component tree and the template is specified using Cutpoints: key/value pairs mapping the ID of the component in the tree to a selector for the element in the template. When you use the renderer component grade, the Framework automatically generates a list of cutpoints from the `selectors` specified in the component options.

For more detailed information, see Cutpoints.

Useful functions and events

In addition to the standard view component features, the renderComponent automatically synthesizes and attaches a refreshView function and an afterRender event.

refreshView:

The refreshView function is used to trigger/re-trigger the rendering of a component. If the "renderOnInit" option is set to "true", the component will automatically render without having to specifically call refreshView.

afterRender:

The afterRender event will be fired as soon as the rendering process has finished. This is useful to know when a render component is ready or to attach new events and functions to pieces of the rendered DOM.

Example: Currency Converter

The currency converter example we've been evolving over the course of this tutorial can be implemented as a rendererComponent. As in the view component version, you need to specify selectors and a model, and can define events. In this case, however, declare the grade to be `fluid.rendererComponent`:

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.rendererComponent", "autoInit"],
    selectors: {
        amount: ".tut-currencyConverter-amount",
        currency: ".tut-currencyConverter-currency-selecter",
        result: ".tut-currencyConverter-result"
    },
    model: {
        rates: {
            names: ["euro", "yen", "yuan", "usd", "rupee"],
            values: ["0.712", "81.841", "6.609", "1.02", "45.789"]
        },
        currentSelection: "0.712",
        amount: 0,
        result: 0
    },
    events: {
        conversionUpdated: null
    }
});
```

As mentioned above, you also need to provide a renderer component tree. Create a public function that will accept the component object, `that`, and returns the tree.

In general, renderer component trees contain one entry per renderer component. One renderer component is typically one element in a user interface, for example: a text entry field, a set of radio buttons, a row in a table.

In a renderer component tree, the binding to the data model is specified using a special notation: the EL path into the data model is enclosed in `"${...}"`. Some types of renderer components, such as a selection, have a particular format. For details see ProtoComponent Types. The preferred way of specifying the component tree to a renderer component is via the `protoTree` top-level option.

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.rendererComponent", "autoInit"],
    selectors: {
        amount: ".tut-currencyConverter-amount",
        currency: ".tut-currencyConverter-currency-selecter",
        result: ".tut-currencyConverter-result"
    },
    model: {
        rates: {
            names: ["euro", "yen", "yuan", "usd", "rupee"],
            values: ["0.712", "81.841", "6.609", "1.02", "45.789"]
        },
        currentSelection: "0.712",
        amount: 0,
        result: 0
    },
    protoTree: {
        amount: "${amount}",
        currency: {
            optionnames: "${rates.names}",
            optionlist: "${rates.values}",
            selection: "${currentSelection}"
        },
        result: "${result}"
    }
});
```

While it is not necessary to create event handlers to update the model when users change the controls (the renderer provides incoming data binding support automatically), you still need event handlers for:

- responding to changes in the model by updating the converted amount
- refreshing the display when the result is updated.

We can clean up the implementation of our component from the Tutorial - Evented Components still further by writing these event handlers also in a declarative form. The display update can be handled by a `modelListeners` entry of the kind we have seen before:

```
    modelListeners: {
        "result": {
            func: "{that}.refreshView"
        }
    }
```

This directs the framework to automatically refresh the rendered view whenever the computed converted currency value is updated - this is done by binding a listener onto the renderer's automatically generated `refreshView` method to trigger from any changes received to the model's `result` field.

Arranging declaratively to perform the currency conversion requires a more interesting kind of definition. Any transformation that can be expressed as part of Infusion's Model Transformation system can be used to construct a Model Relay rule which can keep two component models (or two parts of the same component model) synchronised with each other's changes, where the synchronisation automatically takes account of a transformation rule. In this case we can recognise that the transformation performed by this component is one of the standard rules supplied with the framework, fluid.transforms.linearScale (if it weren't part of the standard set, it would be easy to use any suitable free function as the transforming rule). With the current Infusion version of 1.5, this requires upgrading the base grade of our component from `fluid.rendererComponent` to `fluid.rendererRelayComponent`. This requirement will go away in Infusion 2.0, along with the use of the "old renderer". The relay rule looks like this:

```
    modelRelay: {
        target: "result",
        singleTransform: {
            type: "fluid.transforms.linearScale",
                value: "{that}.model.amount",
                factor: "{that}.model.currentSelection"
            }
        }
    }
```

This rule states that the value held in the model field `amount` will be multiplied by the value held in `currentSelection` and the result placed in `result`. Note that within the model transformation document itself we need to use fully-qualified IoC expressions of the form `{that}.model.amount` etc. in order to avoid ambiguity with referring directly to strings. Outside the transformation rule we can use short-form references to the current component's model fields such as `result` - although note that we would have had to have used the long forms here too if we had wanted to refer to a model held by a different component.

Putting it all together, you have the following:

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.rendererRelayComponent", "autoInit"],
    selectors: {
        amount: ".tut-currencyConverter-amount",
        currency: ".tut-currencyConverter-currency-selecter",
        result: ".tut-currencyConverter-result"
    },
    model: {
        rates: {
            names: ["euro", "yen", "yuan", "usd", "rupee"],
            values: ["0.712", "81.841", "6.609", "1.02", "45.789"]
        },
        currentSelection: "0.712",
        amount: 0,
        result: 0
    },
    protoTree: {
        amount: "${amount}",
        currency: {
            optionnames: "${rates.names}",
            optionlist: "${rates.values}",
            selection: "${currentSelection}"
        },
        result: "${result}"
    },
    modelListeners: {
        "result": {
            func: "{that}.refreshView"
        }
    },
    modelRelay: {
        target: "result",
        singleTransform: {
            type: "fluid.transforms.linearScale",
                value: "{that}.model.amount",
                factor: "{that}.model.currentSelection"
            }
        }
    }
    renderOnInit: true
});
```

Impressively we have succeeded in implementing all of this component design without a single line of user JavaScript code.

Previous: Tutorial - View Components Up to overview **Next:** Tutorial - Subcomponents