Tutorial - Evented Components

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

On This Page

See Also

Component Grades
Infusion Event System

Many times, you will be creating a component that works in an environment where other things are operating, and it will probably need to notify those other things of key **events** in its lifecycle. Events can be used to trigger changes in the visual appearance of a component, or actions by other components. For example:

- the Infusion Reorderer component provides drag-and-drop functionality for lists, grids, etc. Its operation has several key events, such as when a move is initiated, when it's completed, etc.
- the Infusion Uploader component, a queued multi-file uploader, has events including when a file is successfully added to the queue, when each file starts being uploaded, when each upload completes, etc.

The Infusion Framework defines its own event system. Infusion events differ from browser events in that they are not bound to the DOM or its insfrastructure. Infusion events can be used for anything, not only user-interface related events.

Declaring an Evented Component

To use events with your component is to use the **eventedComponent** grade. To do this:

- specify a grade of `fluid.eventedComponent`, and
- include an `events` property in your defaults, listing the events your component will fire.

```
fluid.defaults("tutorials.eventedComponent", {
    gradeNames: ["fluid.eventedComponent", "autoInit"],
    ...
    events: {
        onAnAction: null,
        afterAnAction: null,
        onAnotherAction: "preventable",
        afterAnotherAction: null
    }
});
```

The contents of the `events` object is a set of key-value pairs where the key is the event name and the value is the event type.

- **Event naming conventions**: You can call your events anything you like, but Infusion has adopted the convention of prefacing events with `on` or `a fter` to indicate whether or not the event is being fired before some action is taken, or after the action has completed.
- **Event types**: If the event is a standard event defined on this component, you will normally write `null` here. Event types supported by the framework are described at the Infusion Event System. Another possibility is to inject an event appearing elsewhere in the component tree by use of an IoC reference such as `{myOtherComponent}.events.myEvent`.

Example: Saving and Deleting

Suppose you're creating a component that is responsible for managing records of some kind, or editing documents. An application like that is going to allow users to save their edits or remove the record altogether. You might create the following events for these actions:

```
// Declare the events in the defaults
fluid.defaults("tutorials.recordEditor", {
    gradeNames: ["fluid.eventedComponent", "autoInit"],
    ...
    events: {
        afterSave: null,
        onRemove: "preventable",
        afterRemove: null
    }
});
```

By making the `onRemove` event `preventable`, you have the option of carrying out some kind of double-check, like a confirmation dialog, if you like.

Firing Events

When you declare your component to be an **evented** component, the Framework will automatically set up *event firers* for all of your listed events. These will be attached to an object on your component called `events` and provide an API (and reference target) for you to fire events and add or remove listeners.

Example: Saving and Deleting

Our record editor component will likely have public methods for the saving and removing of records. We will define these methods using the framework facility for invokers. These methods will be responsible for firing the events.

```
// Declare the events in the defaults
fluid.defaults("tutorials.recordEditor", {
    gradeNames: ["fluid.eventedComponent", "autoInit"],
    events: {
        afterSave: null,
        onRemove: "preventable",
        afterRemove: null
    },
    invokers: {
        save: {
            funcName: "tutorials.recordEditor.save",
            args: "{that}"
        },
        remove: {
            funcName: "tutorials.recordEditor.remove",
            args: "{that}"
        }
    }
});

// Add public methods that will fire events when they do things
tutorials.recordEditor.save = function (that) {
    // save stuff
    // ...
    // let anyone listening know the save has happened:
    that.events.afterSave.fire();
};

tutorials.recordEditor.remove = function (that) {
    // see if anyone listening objects to the removal:
    var prevent = that.events.onRemove.fire();
    if (prevent === false) {
        // a listener prevented the move, don't do it
    }
    else {
        // no one objects, go ahead and remove
        // ...
        // let listeners know that the remove has completed
        that.events.afterRemove.fire();
    }
};
```

Example: Currency Converter

Component grades can be combined, if necessary. Suppose we wish to add events to the model-bearing currency converter shown on the previous page. We can declare the component to be both a model component and an evented component by using the standard framework grade `fluid.standardComponent` which is a built-in defined as the combination of `fluid.eventedComponent` and `fluid.modelComponent`. The user can define such combinations themselves by just adding the list of grades directly into `gradeNames`.

Compatibility note:

During the transition period between Infusion 1.5 and Infusion 2.0 it is recommended that authors of new code use the names `fluid.modelRelayComponent` and `fluid.standardRelayComponent` rather than `fluid.modelComponent` and `fluid.standardComponent` respectively. See Component Grades for details.

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.standardComponent", "autoInit"],
    model: {
        rates: {
            euro: 0.712,
            yen: 81.841,
            yuan: 6.609,
            usd: 1.02,
            rupee: 45.789
        },
        currentSelection: "euro",
        amount: 0,
        convertedAmount: 0
    },
    events: {
        conversionUpdated: null
    },
    invokers: {
        updateCurrency: {
            changePath: "currentSelection",
            value: "{arguments}.0"
        },
        updateRate: {
            funcName: "tutorials.currencyConverter.updateRate",
            args: ["{that}", "{arguments}.0", "{arguments}.1"] // currency, newRate
        },
        updateAmount: {
            funcName: "tutorials.currencyConverter.updateAmount",
            args: ["{that}", "{arguments}.0"] // amount
    },
    modelListeners: {
        "convertedAmount" {
            func: "{that}.events.conversionUpdated.fire",
            args: "{change}.value"
        }
    }
});

tutorials.currencyConverter.updateRate = function (that, currency, newRate) {
    that.applier.change(["rates", currency], newRate);
};

tutorials.currencyConverter.updateAmount = function (that, amount) {
    var convertedAmount = amount * that.model.rates[that.model.currentSelection];
    that.applier.change("convertedAmount", convertedAmount);
};
```

Notes on declarative programming:

It's a long-term goal of the Infusion framework that as much logic as possible can be expressed as declarative configuration, rather than as manual JavaScript code. By comparing in this page's history, you can see that since it was first written, the majority of the implementation can now be expressed in configuration (e.g., the invoker definitions, the implementation of `updateCurrency` and the model listener). In future versions of Infusion, 100% of this component will be expressible as configuration - it can't be handled by the current framework because of the indirection into the model from itself (the fact that `updateRate` and `updateAmount` use `model.currentSelection` and `currency` as model indexes). You can track this work at FLUID-5286.

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