Evented 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 `after` to indicate whether or not the event is being fired before some action is taken, or after the action has completed.
- **Event types**: By default (the `null` type), events are fired to everyone listening. You can alternatively select one of two other special cases:
  - `unicast`: Fired to only the first registered listener.
  - `preventable`: Listeners can cancel the relevant action and prevent other listeners from hearing about it. In the example above, `onAnotherAction` is a preventable event.

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 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. These methods will be responsible for firing the events. We'll add the public methods using the "finalInit" lifecycle hook:

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

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

    that.remove = function () {
        // 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:

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.modelComponent", "fluid.eventedComponent", "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
    },
    finalInitFunction: "tutorials.currencyConverter.finalInit"
});

tutorials.currencyConverter.finalInit = function (that) {

    // Add methods to the component object
    that.updateCurrency = function (newCurrency) {
        that.applier.requestChange("currentSelection", newCurrency);
    };

    that.updateRate = function (currency, newRate) {
        that.applier.requestChange("rates." + currency, newRate);
    };

    that.convert = function (amount) {
        var convertedAmount = amount * that.model.rates[that.model.currentSelection];
        that.applier.requestChange("convertedAmount", convertedAmount);
        return amount;
    };

    that.applier.modelChanged.addListener("convertedAmount", function (model, oldModel, changeRequest) {
        that.events.conversionUpdated.fire(model.convertedAmount);
    });
};
```

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