Model Components

Previous: Basic Component Creation - Little Components Up to overview **Next:** Evented Components

On This Page

See Also

Component Grades
ChangeApplier
ChangeApplier API
Model Objects
Component Lifecycle and autoInit

Many components manage an internal model. For example:

- the Infusion Pager component, which allows users to break up long lists of items into separate pages, maintains a model including what the current page is, how many items are shown per page, how the list is sorted, etc.
- the Infusion (Floe) UI Options (2008-2009) component uses a slider that maintains a model including the minimum and maximum values and the current setting

The Infusion Framework provides supports for model-bearing components. When you declare a component to be a **model component**, the Framework will automatically construct a ChangeApplier, which wraps the model with special functions that can be used to query and modify the model. The ChangeApplier helps to protect the model from conflicting asynchronous changes in an environment where multiple components may be working with the same model. It also allows you to add checks that can prevent changes to the model if necessary (e.g validation).

More advanced information about the ChangeApplier is available through the links in the "See Also" sidebar.

Declaring a Model Component

To use a model with your component, you need to use the **modelComponent** grade. To do this:

- specify a grade of `fluid.modelComponent`, and
- include a `model` property in your defaults containing your component's model.

```
fluid.defaults("tutorials.modelBearingComponent", {
    gradeNames: ["fluid.modelComponent", "autoInit"],
    ...
    model: {
        ...
    }
});
```

The `model` can be any JavaScript object that can be successfully copyable using the `fluid.copy()` function. The content of your model will be entirely dictated by the needs of your application.

Using The Change Applier

The Framework will attach both your model and its ChangeApplier to the component object as top-level properties. Your methods can *read* the model directly, using `that.model.*`, but the ChangeApplier should be used to make any changes to the model, using `that.applier.requestChange();`.

Example: Dated Component

As an example, let's consider a component that need to record, for whatever reason, a date. Your `model` will include a `date` field:

```
fluid.defaults("tutorials.modelBearingComponent", {
    gradeNames: ["fluid.modelComponent", "autoInit"],
    ...
    model: {
        date: null
    }
});
```

Suppose you want the `date` initialized to the current date at the time the component is instantiated, and you want this to happen before other component initialization happens. The IoC System provides a "hook" into the beginning of the component creation lifecycle: the `preInitFunction`. You can provide a function that sets up anything required for component creation, inclucing initializing the model (if necessary) or creating any event handler functions that may be used later.

To initialize the `date` portion of the model, we can create a `preInitFunction` as follows:

```
fluid.defaults("tutorials.datedComponent", {
    gradeNames: ["fluid.modelComponent", "autoInit"],
    model: {
        date: null
    },
    preInitFunction: "tutorials.datedComponent.preInit"
});

tutorials.datedComponent.preInit = function (that) {
    // set the date in the model to ensure that it is
    // correctly set to "the date at runtime"
    that.applier.requestChange("date", new Date());
};
```

Example: Currency Converter

The currency converter example we presented on the previous page might be more helpful if it supported more than one conversion rate. We can use a model to hold the available rates and to keep track of the currently-selected rate. We define this model in the component's defaults:

```
fluid.defaults("tutorials.currencyConverter", {
    gradeNames: ["fluid.modelComponent", "autoInit"],
    model: {
        currentSelection: "euro",
        rates: {
            euro: 0.712,
            yen: 81.841,
            yuan: 6.609,
            usd: 1.02,
            rupee: 45.789
        }
    },
    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) {
        return amount * that.model.rates[that.model.currentSelection];
    };
};
```

Previous: Basic Component Creation - Little Components Up to overview **Next:** Evented Components