Advertise Here

Making Use of jQuery UI’s Widget Factory

Pavan Podila on Jan 25th 2013 with 12 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 1 Hour

For a long time, the only way to write custom controls in jQuery was to extend the `$.fn` namespace. This works well for simple widgets, however, as you start building more stateful widgets, it quickly becomes cumbersome. To aid in the process of building widgets, the jQuery UI team introduced the Widget Factory, which removes most of the boilerplate that is typically associated with managing a widget.

The widget factory, part of the *jQuery UI Core*, provides an object-oriented way to manage the lifecycle of a widget. These lifecycle activities include:

- Creating and destroying a widget
- Changing widget options

- Making "*super*" calls in subclassed widgets
- Event notifications

Let’s explore this API, as we build a simple bullet chart widget.

The Bullet Chart Widget

Before we build this widget, let’s understand some of the building blocks of the widget. The Bullet Chart is a concept introduced by Stephen Few as a variation on the bar chart.

The chart consists of a set of bars and markers overlaid on each other to indicate relative performance. There is a quantiative scale to show the actual range of values. By stacking the bars and markers this way, more information can be conveyed without compromising readability. The legend tells the kind of information we are plotting.

The HTML for this chart looks like so:

```html
<!-- Chart Container -->
<div class="chart bullet-chart">

  <!-- Legend -->
  <div class="legend" style="">
    <div class="legend-item">
      <span class="legend-symbol marker green"></span>
      <span class="legend-label">Green Line</span>
    </div>
  </div>

<!-- Chart -->
  <div class="chart-container" style="width: 86%;">

    <!-- Quantitative Scale -->
    <div class="tick-bar">
      <div class="tick" style="left: 0%;"></div>
      <div class="tick-label" style="left: 0%;">0</div>
      <div class="tick" style="left: 25%;"></div>
      <div class="tick-label" style="left: 25%;">25</div>
      <div class="tick" style="left: 50%;"></div>
      <div class="tick-label" style="left: 50%;">50</div>
      <div class="tick" style="left: 75%;"></div>
      <div class="tick-label" style="left: 75%;">75</div>
      <div class="tick" style="left: 100%;"></div>
      <div class="tick-label" style="left: 100%;">100</div>
    </div>

    <!-- Bars -->
    <div class="bar" style="left: 0px; width: 75%;" bar-index="0">
    <div class="bar blue" style="left: 0px; width: 50%;" bar-index

    <!-- Markers -->
    <div class="marker green" style="left: 80%;" marker-index="0">
    <div class="marker red" style="left: 50%;" marker-index="1"></
  </div>
</div>
```

Our widget, which we'll call `jquery.bulletchart`, will dynamically generate this HTML from the data provided. The final widget can be viewed on the demo page, which you can download from GitHub. The call to create the widget should look like so:

```
$('.chart').bulletchart({
  size: 86,
  bars: [
    { title: 'Projected Target', value: 75, css: '' },
    { title: 'Actual Target', value: 50, css: 'blue' }
  ],
  markers: [
    { title: 'Green Line', value: 80, css: 'green' },
    { title: 'Minimum Threshold', value: 50, css: 'red' }
  ],

  ticks: [0, 25, 50, 75, 100]
});
```

All of the values are in percentages. The `size` option can be used when you want to have several bullet charts placed next to each other with relative sizing. The `ticks` option is used to put the labels on the scale. The markers and bars are specified as an array of object literals with `title`, `value` and `css` properties.

Building the Widget

Now that we know the structure of the widget, let's get down to building it. A widget is created by calling `$.widget()` with the name of the widget and an object containing its instance methods. The exact API looks like:

```
jQuery.widget(name[, base], prototype)
```

For now, we will work with just the name and prototype arguments. For the bulletchart, our basic widget stub looks like the following:

```
$.widget('nt.bulletchart', {
  options: {},

  _create: function () {},
  _destroy: function () {},
  _setOption: function (key, value) {}
});
```

It's recommended that you always namespace your widget names. In this case, we are using '**nt.bulletchart**'. All of the jQuery UI widgets are under the '**ui**' namespace. Although we are namespacing the widget, the call to create a widget on an element does not include the namespace. Thus, to create a bullet chart, we would just call `$('#elem').bulletchart()`.

The instance properties are specified following the name of the widget. By convention, all private methods of the widget should be prefixed with '_'. There are some special properties which are expected by the widget factory. These include the `options`, `_create`, `_destroy` and `_setOption`.

- `options`: These are the default options for the widget
- `_create`: The widget factory calls this method the first time the widget is instantiated. This is used to create the initial DOM and attach any event handlers.
- `_init`: Following the call to `_create`, the factory calls `_init`. This is generally used to reset the widget to initial state. Once a widget is created, calling the plain widget constructor, eg: **$.bulletchart()**, will also reset the widget. This internally calls `_init`.
- `_setOption`: Called when you set an option on the widget, with a call such as: `$('#elem').bulletchart('option', 'size', 100)`. Later we will see other ways of setting options on the widget.

Creating the initial DOM with `_create`

Our bulletchart widget comes to life in the `_create` method. Here is where we build the basic structure for the chart. The `_create` function can be seen below. You will notice that there is not much happening here besides creating the top-level container. The actual work of creating the DOM for bars, markers and ticks happens in the `_setOption` method. This may seem somewhat counter-intuitive to start with, but there is a valid reason for that.

```
_create: function () {
  this.element.addClass('bullet-chart');

  // chart container
  this._container = $('<div class="chart-container"></div>')
    .appendTo(this.element);

  this._setOptions({
    'size': this.options.size,
    'ticks': this.options.ticks,
    'bars': this.options.bars,
    'markers': this.options.markers
  });

}
```

Note that the bars, markers and ticks can also be changed by setting options on the widget. If we kept the code for its construction inside `_create`, we would be repeating ourselves inside `_setOption`. By moving the code to `_setOption` and invoking it from `_create` removes the duplication and also centralizes the construction.

Additionally, the code above shows you another way of setting options on the widget. With the `_setOptions` method (note the plural), you can set mutiple options in one go. Internally, the factory will make individual calls on `_setOption` for each of the options.

The `_setOption` method

For the bullet chart, the `_setOption` method is the workhorse. It handles creation of the markers, bars and ticks and also any changes made to these properties. It works by clearing any existing elements and recreating them based on the new value.

The `_setOption` method receives both the option key and a value as arguments. The key is the name of the option, which should correspond to one of the keys in the default options. For example, to change the bars on the widget, you would make the following call:

```
$('#elem').bulletchart('option', 'bars', [{
    title: 'New Marker', value: 50
}])
```

The `_setOption` method for the bulletchart looks like so:

```
_setOption: function (key, value) {
  var self = this,
    prev = this.options[key],
    fnMap = {
      'bars': function () {
        createBars(value, self);
      },
      'markers': function () {
        createMarkers(value, self);
      },
      'ticks': function () { createTickBar(value, self); },
      'size': function () {
        self.element.find('.chart-container')
          .css('width', value + '%');
      }
    };

  // base
  this._super(key, value);

  if (key in fnMap) {
    fnMap[key]();

    // Fire event
    this._triggerOptionChanged(key, prev, value);
  }
}
```

Here, we create a simple hash of the option-name to the corresponding function. Using this hash, we only work on valid options and silently ignore invalid ones. There are two more things happening here: a call to `_super()` and firing the option changed event. We will look at them later in this article.

For each of the options that changes the DOM, we call a specific helper method. The helper methods, `createBars`, `createMarkers` and `createTickBar` are specified outside of the widget instance properties. This is because they are the same for all widgets and need not be created individually for each widget instance.

```
// Creation functions
function createTickBar(ticks, widget) {

    // Clear existing
    widget._container.find('.tick-bar').remove();

    var tickBar = $('<div class="tick-bar"></div>');
    $.each(ticks, function (idx, tick) {
      var t = $('<div class="tick"></div>')
        .css('left', tick + '%');

      var tl = $('<div class="tick-label"></div>')
        .css('left', tick + '%')
        .text(tick);

      tickBar.append(t);
      tickBar.append(tl);
    });

    widget._container.append(tickBar);

  }

  function createMarkers(markers, widget) {

    // Clear existing
    widget._container.find('.marker').remove();

    $.each(markers, function (idx, m) {
      var marker = $('<div class="marker"></div>')
        .css({ left: m.value + '%' })
        .addClass(m.css)
        .attr('marker-index', idx);

      widget._container.append(marker);
    });

  }

  function createBars(bars, widget) {

    // Clear existing
    widget._container.find('.bar').remove();

    $.each(bars, function (idx, bar) {
      var bar = $('<div class="bar"></div>')
        .css({ left: 0, width: '0%' })
        .addClass(bar.css)
        .attr('bar-index', idx)
        .animate({
          width: bar.value + '%'
        });

      widget._container.append(bar);
    });

  }
```

All of the creation functions operate on percentages. This ensures that the chart reflows nicely when you resize the containing element.

proudly towards
we are relentlessly working towards making a difference in your life everyday and this time with 3G.
3G
banglalink

The Default Options

Without any options specified when creating the widget, the defaults will come into play. This is the role of the `options` property. For the bulletchart, our default options look like so:

```
    $.widget('nt.bulletchart', {
      options: {
        // percentage: 0 - 100
        size: 100,

        //  [{ title: 'Sample Bar', value: 75, css: '' }],
        bars: [],

        //  [{ title: 'Sample Marker', value: 50, css: '' }],
        markers: [],

        // ticks -- percent values
        ticks: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
      },

      ...
  }
```

We start with a size of *100%*, no bars and markers and with ticks placed every *10%*. With these defaults, our bullet chart should look like:

So far, we have seen how to create the widget using `_create` and updating it using `_setOption`. There is one other lifecycle method, which will be called when you destroy a widget. This is the `_destroy` method. When you call `$('#elem').bulletchart('destroy')`, the widget factory internally calls `_destroy` on your widget instance. The widget is responsible for removing everything that it introduced into the DOM. This can include classes and other DOM elements that were added in the `_create` method. This is also a good place to unbind any event handlers. The `_destroy` should be the exact opposite of the `_create` method.

For the bullet chart widget, the `_destroy` is quite simple:

```
_destroy: function () {
  this.element.removeClass('bullet-chart');
  this.element.empty();
},
```

Subclassing, Events and More

Our bulletchart widget is almost feature complete, except for one last feature: *legend*. The legend is quite essential, since it will give more meaning to the markers and bars. In this section we will add a legend next to the chart.

Rather than adding this feature directly to the bulletchart widget, we will create a subclass, `bulletchart2`, that will have the legend support. In the process, we will also look at some of the interesting features of Widget Factory inheritance.

Adding a Legend

The Widget Factory supports subclassing of a widget to create more specialized versions. Earlier in the article, we saw the API for `$.widget()`, which had three arguments:

```
jQuery.widget(name[, base], prototype)
```

The second parameter allows us to pick a base-class for our widget. Our `bulletchart2` widget, which subclasses `bulletchart`, will have the following signature:

```
$.widget('nt.bulletchart2', $.nt.bulletchart, {
  options: {
    // Show/hide legend
    legend: true
  },

  // this ensures we keep the same namespace as the base
  widgetEventPrefix: $.nt.bulletchart.prototype.widgetEventPrefix,

  _create: function () { ... },

  _destroy:function(){ ... },

  _setOption: function (key, value) { ... }
})
```

There are few interesting things to note here:

- We continue to namespace our widget name: `nt.bulletchart2`.
- The widget factory automatically puts the widget under the **$.nt** namespace. Thus, to reference our previous widget, we used `$.nt.bulletchart`. Similarly if we were to subclass one of the standard jQuery UI widgets, we would reference them with `$.ui.widget-name`
- The `widgetEventPrefix` is a new property that we haven't seen before. We will get to that when we talk about events. The rest of the instance properties should be familiar.

Since we are adding more DOM elements with the legend, we will have to override the `_create` method. This also means that we need to override `_destroy`, in order to be symmetric.

```
_create: function () {
  var self = this;

```

```
    this._legend = $('<div class="legend"></div>')
      .appendTo(this.element);

    ...

    // Call the base
    this._super();

    this._setOption('legend', this.options.legend);
  },

  _destroy:function(){
    this.element.find('.legend').empty();

    ...

    this._super();
  },
```

Here, again, we see the same pattern as our earlier `_create` method. We create the container for the legend and then call `_setOption` to build the rest of the legend. Since we are overriding the `_create`, we need to make sure that we call the base `_create`. We do this with the call to `_super`. Similarly, in `_destroy`, we also see the call to `_super`.

Now you may be wondering: how does the widget know which super-method to call with a simple unqualified `_super` invocation? The smarts for that lie in the bowels of the widget factory. When a widget is subclassed, the factory sets up the `_super` reference differently for each of the instance functions. Thus, when you call `_super` from your instance method, it always points to the correct `_super` method.

Event Notifications

Since the bulletchart supports changing markers and bars, the legend needs to be in sync with those changes. Additionally, we will also support toggling the visibility of markers and bars by clicking on the legend items. This becomes useful when you have several markers and bars. By hiding a few of the elements, you can see the others more clearly.

To support syncing of the legend with the changes to markers and bars, the `bulletchart2` widget must listen to any changes happening to those properties. The base bulletchart already fires a change event every time that its options change. Here is the corresponding snippet from the base widget:

```
_setOption: function (key, value) {
  var self = this,
    prev = this.options[key];

  ...

  // base
  this._super(key, value);

  if (key in fnMap) {
    fnMap[key]();

    // Fire event
    this._triggerOptionChanged(key, prev, value);
  }
},

_triggerOptionChanged: function (optionKey, previousValue, current
  this._trigger('setOption', {type: 'setOption'}, {
    option: optionKey,
    previous: previousValue,
    current: currentValue
  });
}
```

Whenever an option is set, the `setOption` event is fired. The event data contains the previous and new value for the option that was changed.

By listening to this event in the subclassed widget, you can know when the markers or bars change. The `bulletchart2` widget subscribes to this event in its `_create` method. Subscribing to widgets events is achieved with the call to `this.element.on()`. `this.element` points to the jQuery element on which the widget was instantiated. Since the event will be fired on the element, our event subscription needs to happen on that.

```
_create: function () {
  var self = this;

  this._legend = $('<div class="legend"></div>')
    .appendTo(this.element);

  ...

  // Apply legend on changes to markers and bars
```

```
    this.element.on('bulletchart:setoption', function (event, data)
      if (data.option === 'markers') {
        createLegend(data.current, self.options.bars, self);
      }
      else if (data.option === 'bars') {
        createLegend(self.options.markers, data.current, self);
      }
    });

    // Call the base
    this._super();

    this._setOption('legend', this.options.legend);
  }
```

Note the event name used for subscribing: `'bulletchart:setoption'`. As a policy, the widget factory attaches an event-prefix for events fired from the widget. By default, this prefix is the name of the widget, but this can be easily changed with the `widgetEventPrefix` property. The base bulletchart widget changes this to `'bulletchart:'`.

```
$.widget('nt.bulletchart', {
    options: { ... },

    widgetEventPrefix: 'bulletchart:'

    ...
});
```

We also need to subscribe to `'click'` events on the legend items to hide/show the corresponding marker/bar. We do this with the `_on` method. This method takes a hash of the event signature to the handler function. The handler's context (`this`) is correctly set to the widget instance. One other convenience with `_on` is that the widget factory automatically unbinds the events on destroy.

```
_create: function () {
...

  // Listen to clicks on the legend-items
  this._on({
    'click .legend-item': function (event) {
      var elt = $(event.currentTarget),
        item = elt.data('chart-item'),
        selector = '[' + item.type + '-index=' + item.index + ']';

      this.element.find(selector).fadeToggle();
      elt.toggleClass('fade');
    }
    });

  ...
  }
```

More Tips

The Widget factory packs a few other niceties that you should be aware of.

Referencing the widget instance

So far, we have only seen one way of calling methods on the widget. We did this with `$('#elem).bulletchart('method-name')`. However, this only allows calling public methods such as 'option', 'destroy', 'on', 'off'. If you want to invoke those methods directly on the widget instance, there is a way of doing that. The widget factory attaches the widget instance to the `data()` object of the element. You can get this instance like so:

```
var widget = $('#elem').data('bulletchart');
widget.destroy();
```

Additionally, if you want to get a hold of all bulletchart widgets on the page, there is also a selector for that:

```
var allCharts = $(':nt-bulletchart');
```

Some special methods

There are a few special methods that you should be aware of, which are used less frequently: `_getCreateEventData()` and `_getCreateOptions()`. The former is used to attach event data for the 'create' event that is fired after finishing the call to `_create`.

`_getCreateOptions` is for attaching additional default options for the widget or overriding existing ones. The user-provided options override options returned by this method, which in turn overrides the default widget options.

Summary

That's a wrap! If you'd like to explore further, the references below should serve you quite well. Of course, the best source for information will always be source–code, itself. I would encourage reading the jquery.ui.widget source on GitHub.

- JQueryUI Widget Factory API
- Slides on Widget Factory

Like 116 people like this. Be the first of your friends.

Tags: jquery ui

By Pavan Podila

I am a Financial Technologist building apps for Web and Mobile Platforms using Ruby, NodeJS and iOS. You can follow me on Twitter or read my Blog

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more