|  |  |  |
| --- | --- | --- |
| **Chapter 11. Custom Events: Part 2** | | |
| [Prev](javascript:window.parent.location.href=window.parent.prev;) |  | [Next](javascript:window.parent.location.href=window.parent.next;) |

**Part 11. Custom Events: Part 2**

**Introducing Custom Events**

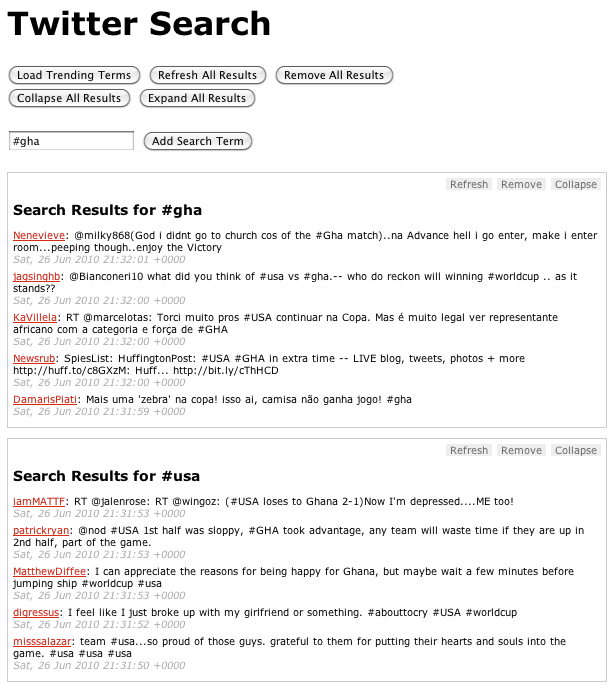
**A Sample Application**

To demonstrate the power of custom events, we’re going to create a simple tool for searching Twitter. The tool will offer several ways for a user to add search terms to the display: by entering a search term in a text box, by entering multiple search terms in the URL, and by querying Twitter for trending terms.

The results for each term will be shown in a results container; these containers will be able to be expanded, collapsed, refreshed, and removed, either individually or all at once.

When we’re done, it will look like this:

**Figure 11.1. Our finished application**



**The Setup**

We’ll start with some basic HTML:

<h1>Twitter Search</h1>

<input type="button" id="get\_trends"

value="Load Trending Terms" />

<form>

<input type="text" class="input\_text"

id="search\_term" />

<input type="submit" class="input\_submit"

value="Add Search Term" />

</form>

<div id="twitter">

<div class="template results">

<h2>Search Results for

<span class="search\_term"></span></h2>

</div>

</div>

This gives us a container (#twitter) for our widget, a template for our results containers (hidden via CSS), and a simple form where users can input a search term. (For the sake of simplicity, we’re going to assume that our application is JavaScript-only and that our users will always have CSS.)

There are two types of objects we’ll want to act on: the results containers, and the Twitter container.

The results containers are the heart of the application. We’ll create a plugin that will prepare each results container once it’s added to the Twitter container. Among other things, it will bind the custom events for each container and add the action buttons at the top right of each container. Each results container will have the following custom events:

refresh

Mark the container as being in the “refreshing” state, and fire the request to fetch the data for the search term.

populate

Receive the returned JSON data and use it to populate the container.

remove

Remove the container from the page after the user verifies the request to do so. Verification can be bypassed by passing true as the second argument to the event handler. The remove event also removes the term associated with the results container from the global object containing the search terms.

collapse

Add a class of collapsed to the container, which will hide the results via CSS. It will also turn the container’s “Collapse” button into an “Expand” button.

expand

Remove the collapsed class from the container. It will also turn the container’s “Expand” button into a “Collapse” button.

The plugin is also responsible for adding the action buttons to the container. It binds a click event to each action’s list item, and uses the list item’s class to determine which custom event will be triggered on the corresponding results container.

$.fn.twitterResult = function(settings) {

return $(this).each(function() {

var $results = $(this),

$actions = $.fn.twitterResult.actions =

$.fn.twitterResult.actions ||

$.fn.twitterResult.createActions(),

$a = $actions.clone().prependTo($results),

term = settings.term;

$results.find('span.search\_term').text(term);

$.each(

['refresh', 'populate', 'remove', 'collapse', 'expand'],

function(i, ev) {

$results.bind(

ev,

{ term : term },

$.fn.twitterResult.events[ev]

);

}

);

// use the class of each action to figure out

// which event it will trigger on the results panel

$a.find('li').click(function() {

// pass the li that was clicked to the function

// so it can be manipulated if needed

$results.trigger($(this).attr('class'), [ $(this) ]);

});

});

};

$.fn.twitterResult.createActions = function() {

return $('<ul class="actions" />').append(

'<li class="refresh">Refresh</li>' +

'<li class="remove">Remove</li>' +

'<li class="collapse">Collapse</li>'

);

};

$.fn.twitterResult.events = {

refresh : function(e) {

// indicate that the results are refreshing

var $this = $(this).addClass('refreshing');

$this.find('p.tweet').remove();

$results.append('<p class="loading">Loading ...</p>');

// get the twitter data using jsonp

$.getJSON(

'http://search.twitter.com/search.json?q=' +

escape(e.data.term) + '&rpp=5&callback=?',

function(json) {

$this.trigger('populate', [ json ]);

}

);

},

populate : function(e, json) {

var results = json.results;

var $this = $(this);

$this.find('p.loading').remove();

$.each(results, function(i,result) {

var tweet = '<p class="tweet">' +

'<a href="http://twitter.com/' +

result.from\_user +

'">' +

result.from\_user +

'</a>: ' +

result.text +

' <span class="date">' +

result.created\_at +

'</span>' +

'</p>';

$this.append(tweet);

});

// indicate that the results

// are done refreshing

$this.removeClass('refreshing');

},

remove : function(e, force) {

if (

!force &&

!confirm('Remove panel for term ' + e.data.term + '?')

) {

return;

}

$(this).remove();

// indicate that we no longer

// have a panel for the term

search\_terms[e.data.term] = 0;

},

collapse : function(e) {

$(this).find('li.collapse').removeClass('collapse')

.addClass('expand').text('Expand');

$(this).addClass('collapsed');

},

expand : function(e) {

$(this).find('li.expand').removeClass('expand')

.addClass('collapse').text('Collapse');

$(this).removeClass('collapsed');

}

};

The Twitter container itself will have just two custom events:

getResults

Receives a search term and checks to determine whether there’s already a results container for the term; if not, adds a results container using the results template, set up the results container using the $.fn.twitterResult plugin discussed above, and then triggers the refresh event on the results container in order to actually load the results. Finally, it will store the search term so the application knows not to re-fetch the term.

getTrends

Queries Twitter for the top 10 trending terms, then iterates over them and triggers the getResults event for each of them, thereby adding a results container for each term.

Here's how the Twitter container bindings look:

$('#twitter')

.bind('getResults', function(e, term) {

// make sure we don't have a box for this term already

if (!search\_terms[term]) {

var $this = $(this);

var $template = $this.find('div.template');

// make a copy of the template div

// and insert it as the first results box

$results = $template.clone().

removeClass('template').

insertBefore($this.find('div:first')).

twitterResult({

'term' : term

});

// load the content using the "refresh"

// custom event that we bound to the results container

$results.trigger('refresh');

search\_terms[term] = 1;

}

})

.bind('getTrends', function(e) {

var $this = $(this);

$.getJSON('http://search.twitter.com/trends.json?callback=?', function(json) {

var trends = json.trends;

$.each(trends, function(i, trend) {

$this.trigger('getResults', [ trend.name ]);

});

});

});

So far, we’ve written a lot of code that does approximately nothing, but that’s OK. By specifying all the behaviors that we want our core objects to have, we’ve created a solid framework for rapidly building out the interface.

Let’s start by hooking up our text input and the “Load Trending Terms” button. For the text input, we’ll capture the term that was entered in the input and pass it as we trigger the Twitter container’s getResults event. Clicking the “Load Trending Terms” will trigger the Twitter container’s getTrends event:

$('form').submit(function(e) {

e.preventDefault();

var term = $('#search\_term').val();

$('#twitter').trigger('getResults', [ term ]);

});

$('#get\_trends').click(function() {

$('#twitter').trigger('getTrends');

});

By adding a few buttons with the appropriate IDs, we can make it possible to remove, collapse, expand, and refresh all results containers at once, as shown below. For the remove button, note how we’re passing a value of true to the event handler as its second argument, telling the event handler that we don’t want to verify the removal of individual containers.

$.each(['refresh', 'expand', 'collapse'], function(i, ev) {

$('#' + ev).click(function(e) { $('#twitter div.results').trigger(ev); });

});

$('#remove').click(function(e) {

if (confirm('Remove all results?')) {

$('#twitter div.results').trigger('remove', [ true ]);

}

});

**Conclusion**

Custom events offer a new way of thinking about your code: they put the emphasis on the target of a behavior, not on the element that triggers it. If you take the time at the outset to spell out the pieces of your application, as well as the behaviors those pieces need to exhibit, custom events can provide a powerful way for you to “talk” to those pieces, either one at a time or en masse. Once the behaviors of a piece have been described, it becomes trivial to trigger those behaviors from anywhere, allowing for rapid creation of and experimentation with interface options. Finally, custom events can enhance code readability and maintainability, by making clear the relationship between an element and its behaviors.

You can see the full application at demos/custom-events.html and demos/js/custom-events.js in the sample code.