CUTTING EDGE

Explore Rich Client Scripting With jQuery, Part 2

Dino Esposito

Contents

Powerful Web applications require powerful client capabilities. Web developers have traditionally relied upon JavaScript to deliver that power. However, raw JavaScript has its limitations, some of which can be addressed via libraries and object orientation.

There are many JavaScript libraries available, but after a while they all look the same. If you can't decide where to begin, I would suggest you start right here—with jQuery. As I discussed last month, jQuery has some handy capabilities, including selectors, filters, wrapped sets, and the key feature—chained functions. (See "Explore Rich Client Scripting With jQuery, Part 1.") This month, I'll look at some others, including the eventing model, visual effects, caching, and AJAX capabilities.

The jQuery Eventing Model

More often than not, browsers have their own representation of events. Internet Explorer has its own eventing model, as do Firefox and Safari. Therefore, achieving cross-browser compatibility for events is no easy task without the help of a good library. Subsequently, nearly any JavaScript library must provide an abstract model for handling events. The jQuery library is no exception.

The jQuery event handling API is organized into two groups of functions. There are some general event methods to add and remove handlers, plus a long list of helper functions. General methods provide the foundation for helpers to work; and helpers make jQuery programming easy and effective.

Figure 1 lists the methods you can use to bind and unbind event handlers to the matching elements of a wrapped set.

Figure 1 General jQuery Methods for Events

Method	Description
bind	Associates the given function to one or more events for each element contained in the wrapped set.
live	Introduced with jQuery 1.3, the function binds the specified event handler to all current and future elements of the wrapped set. This means that if a new DOM element is added that matches the conditions of the wrapped set, the element will be automatically bound to the handler. The die method does the reverse and removes a live event handler from a wrapped set.
one	Works like bind, except that any event handler is automatically removed after it has been run once.
trigger	Triggers the given event for each element in the wrapped set.
triggerHandler	Triggers the given event on one element in the wrapped set and cancels the default browser actions.
unbind	Removes bound events from each element in the wrapped set.

By the way, it is worth noting that in jQuery vernacular a method is code used to process the contents of the wrapped set. A function, on the other hand, is code that performs an operation that is not specifically aimed at processing the contents of a wrapped set.

Binding Event Handlers

The bind method attaches a handler for a given event to all elements in the wrapped set. The complete signature of the bind method is this:

Copy Code

```
bind(eventName, eventData, eventHandler)
```

The first argument is a string and indicates the event to handle. The second argument represents any input data coming with the event. Finally, the third argument is the JavaScript function to bind.

Because jQuery offers an abstract eventing model, it is important to look at the list of supported events. The full list is in **Figure 2**.

Figure 2 Supported Events in jQuery

Event	Fires When
beforeunload	A browser window is unloaded or closed by the user.
blur	An element loses focus because either the user clicked outside of it or tabbed away.
change	The element loses focus and its value has been modified since it gained focus.
click	The user clicks on the element.
dblclick	The user double-clicks on the element.
error	The window object signals that an error has occurred—usually a JavaScript error has been detected.
focus	An element receives focus either via the mouse or tab navigation.
keydown	A key is pressed.
keypress	A key is pressed and released. A keypress is defined as a successive keydown and keyup events.
keyup	A key is released. This event follows keypress.
load	The element and all of its content has finished loading.
mousedown	A mouse button is pressed.
mouseenter	The mouse enters in the area of an element.
mouseleave	The mouse leaves the area of an element.
mousemove	The mouse is moved while it is over an element.
mouseout	The mouse is moved out of an element. Unlike mouseleave, this event also fires when the mouse moves into or out of child elements.
mouseover	The mouse is moved onto an element. Unlike mouseenter, this event also fires when the mouse moves into or out from child elements.
mouseup	The mouse button is released. This event follows click.
resize	An element is resized.
scroll	An element is scrolled.
select	The user selects some text in a text field.
submit	A form is submitted.
unload	A browser window is unloaded.

Because of the browser differences and the level of abstraction provided by the library, the list is less obvious than it may seem at a first glance. For example, change and select events address very distinct scenarios. The change event refers to a change in the value of an input element, including textboxes and dropdown lists. The select event simply refers to text selection in an input or textarea element.

Subtle differences also exist between the pairs of events mouseover/mouseenter and mouseout/mouseleave. They have nearly the same description and differ only because mouseover and mouseout are also fired when the user moves in and out of child elements. For elements with no children, these events are equivalent.

It is possible for you to register the same JavaScript handler for multiple events. You can do that by separating event names with a blank space. The following example toggles a CSS style when the mouse enters or leaves a DIV tag with a given style:

Copy Code

```
$("div.sensitiveArea").bind("mouseenter mouseleave",
```

```
   function(e) {
        $(this).toggleClass("hovered");
    }
);
```

The second argument of the bind method is optional and, if specified, indicates any user-defined data to be passed to the handler. **Figure 3** illustrates how you can toggle the CSS style of a textbox using a rather generic JavaScript handler.

Figure 3 Toggling Textbox Style

Copy Code

```
<script type="text/javascript">
  $(document).ready(
     function() {
       $("#TextBox1").bind("focus",
                           {cssStyle: "focusedTextBox"},
                           setCSS);
       $("#TextBox1").bind("blur",
                           {cssStyle: "focusedTextBox"},
                           setCSS);
     }
  );

  function setCSS(e)
  {
     var name = "#" + e.target.name;
     $(name).toggleClass(e.data.cssStyle);
  }
</script>
```

Note that the preceding code is purely illustrative of the bind function and may not work correctly as a way to toggle styles. It is preferable to use the :focus pseudo-class to add a special style to an element while the element has focus. Most recent browsers will support it. JavaScript handler is a function that declares and receives an event data structure. This object has the members listed in **Figure 4**.

Figure 4 Members of the jQuery Event Object

Member	Description
type	Returns the name of the event, such as "click"
target	Returns a reference to the DOM element that issued the event
pageX	Returns the X mouse coordinate relative to the document
pageY	Returns the Y mouse coordinate relative to the document
preventDefault	Cancels the default action the browser would take after the event
stopPropagation	Stops the bubbling but doesn't prevent the browser's action

It is worth noting that the target property returns a DOM reference object, not a jQuery wrapped set. To figure out the ID of the element, you must invoke its id or name property. The handler retrieves any custom data you pass through its data expando property.

Two methods complete the definition of the event data object. The preventDefault method stops the browser from taking the default action it would normally take after the event. For example, if you call preventDefault in a submit handler, no form submission would occur. The preventDefault method doesn't stop the bubbling of the event through the object's stack, however.

In contrast, the stopPropagation method stops the event bubbling but doesn't prevent the action. If you want to stop event propagation and prevent the default action, do not call either of these methods; just return false from the event handler.

Any handlers attached through the bind method can be detached using the unbind method. The method takes two optional parameters—the name of the event and the handler. If none are specified, all handlers are removed from all elements in the wrapped set.

Triggering Events Programmatically

The method trigger (see **Figure 1**) instructs jQuery to invoke any handlers registered with the specified event. To simulate the user clicking on a given button, use this:

Copy Code

```
$("#Button1").trigger("click");
```

The code won't do anything if no JavaScript handler is registered with the click event of element.

The method triggerHandler differs from trigger in two ways. First, the method triggerHandler also prevents the browser's default action. Second, the method affects only one element one element, not the whole the wrapped set. If the wrapped set contains multiple matching elements, only the first will have the handler fired for the specified event.

If you use the triggerHandler method to programmatically trigger the focus event on an input field, as you will see next, any registered handler executes but the default action of moving the input focus to the field does not happen.

Copy Code

```
$("#TextBox1").triggerHandler("focus");
```

Figure 5 shows what that looks like.

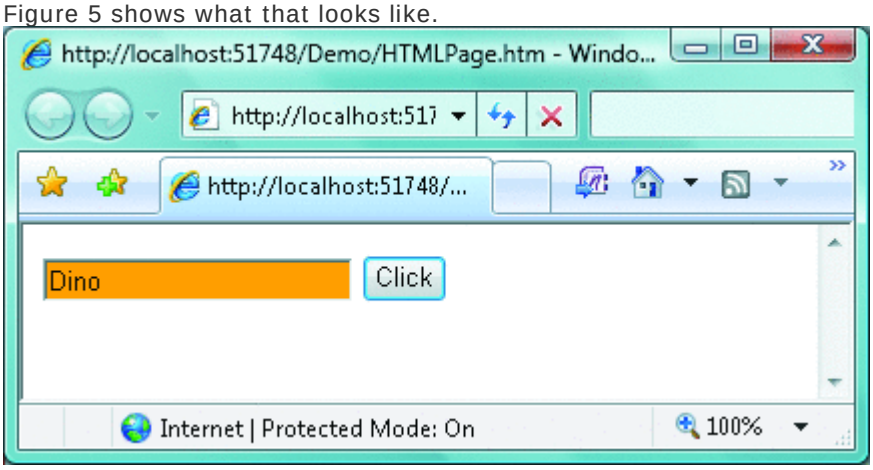

Figure 5 **Triggering the Focus Event Programmatically**

Special Event-Related Methods

The jQuery library supplies three commonly used event methods: ready, hover, toggle. The ready method takes a function and runs it when the DOM is ready to be traversed and manipulated by code:

Copy Code

```
$(document).ready(
  function() {
...
  }
);
```

Libraries need this ability, which replaces the onload event of the browser's window object. The onload event happens too late—when all images have also been loaded. The ready event, on the other hand, fires when the page and the library have been fully initialized.

In the hover function, you place the code you want to execute when the user enters and leaves a set of page elements. The hover function accepts two handlers. The first runs when the mouse hovers over an element in the wrapped set, and the second runs when the mouse leaves the element's area.

Finally, the toggle function performs an even smarter task. It takes two or more handlers and runs them alternately as the user clicks. In other words, the first click on a matching element runs the first handler, the second click runs the second handler, and so on. When the bottom of the handler list is reached, all subsequent clicks run back from the first function in the list.

Event Helpers

To reinforce the idea of its extreme usability, the jQuery library comes with a list of helper methods to simplify the binding of handlers to common events. Helpers come in two forms: with and without a function parameter.
Helpers that accept a parameter receive a JavaScript function to execute when the event is fired. If no parameter is specified, the method just triggers the given event on any element in the wrapped set. **Figure 6** shows the list of supported helpers and distinguishes between binder methods and trigger methods.

Figure 6 Binder and Trigger Event Helpers

Trigger Method (Triggers said event on the wrapped set.)	Binder Method (Binds the specified function to the associated event of matching elements.)
blur	blur(fn)
change	change(fn)
click	click(fn)
dblclick	dblclick(fn)
error	error(fn)
focus	focus(fn)
keydown	keydown(fn)
keypress	keypress(fn)
keyup	keyup(fn)
	load(fn)
	mousedown(fn)
	mousemove(fn)
	mouseout(fn)
	mouseover(fn)
	mouseup(fn)
	resize(fn)
	scroll(fn)
select	select(fn)
submit	submit(fn)
	unload(fn)

The following code registers a click event for a given button. The binding between the event and handler occurs as soon as the document is fully loaded and ready to be manipulated programmatically.

Copy Code

```
$(document).ready( function() {
   $("#btnProcess").click(
       function(e) {
this.text("Please wait ... ");
       }
   );
});
```

Some events such as the scroll, resize, and mouse events make sense only if triggered by an explicit user action. Such events lack a trigger method, as shown in **Figure 6**.

Visual Effects

One of the factors most responsible for the popularity of the jQuery library is the built-in engine for visual effects. In the library, you find an effective engine for building custom animations plus a few facilities for quickly implementing common effects such as fading and sliding.
Predefined effects can be divided into three groups according to the CSS attributes they act on: visibility, sliding, and fading. **Figure 7** lists all effects natively available.

Figure 7 Visual Effects

Visibility effect Description

show	Turns on the visibility of any elements in the wrapped set
hide	Turns off the visibility of any elements in the wrapped set
toggle	Toggles the visibility of any elements in the wrapped set
Sliding effect	Description
slideDown	Displays any matching elements by increasing their height progressively
slideUp	Hides any matching elements by decreasing their height progressively
slideToggle	Shows or hides all matching elements inverting the current sliding setting
Fading effect	Description
fadeIn	Fades any matching elements in by reducing their opacity progressively
fadeOut	Fades any matching elements out by increasing their opacity progressively
fadeTo	Fades the opacity of all matching elements to a specified opacity

All methods in **Figure 7** apply to any matching element in a wrapped set. Visibility methods act on the display CSS attribute and show or hide elements using a nice built-in animation. For example, the following code unveils an HTML panel as the user clicks the button:

Copy Code

```
$("btnShowOrders").click(
   function(e) {
      $("#panelOrders").show(2000);
   }
);
```

The animation takes two seconds to complete. Optional arguments you can specify to visibility methods include the speed of the animation and a callback to invoke upon completion. The type of animation is hardcoded and progressively uncovers the content from the top-left corner.

Sliding methods work on the CSS height attribute of matching elements. The slideDown method increases the height of an element in a loop so that the display happens by revealing the element progressively. The slideUp method does the reverse and hides the element by sliding it up, from the actual height to zero.

Fading methods follow a model analogous to sliding methods. They support optional speed and completion callback and show/hide elements looping on the CSS opacity attribute (see **Figure 8**).

Figure 8 Fading

Copy Code

```
$("#btnShowOrders").click(
   function(e)
   {
      // Hide the current panel.
      $("#panelOrders").fadeOut(1000);

      populateOrderPanel();

      // Show new content and when done
      //apply some CSS styles
      // to denote the new content.
      $("#panelOrders").fadeIn(2000,
         function() {
            $("#panelOrders").css(...);
         }
      );
   }
);
```

The preceding example shows how to fade out an existing HTML panel, refresh it while hidden, and then display it again using fade-in animation. Note that you can indicate the desired speed using an explicit duration in milliseconds or resort to a string representing one of three predefined speeds: slow, normal, or fast.

Custom Animations

All visual effects listed in **Figure 6** are implemented on top of the jQuery animation engine. The heart of this engine is the animate function:

Copy Code

```
function animate( prop, speed, easing, callback )
{
  …
}
```

The first argument to the function is an array of property/value pairs where the property refers to a CSS attribute. The function simply animates the CSS property of each matching element from its current value to the specified value.

Additional, optional arguments you can specify include the speed of the animation and the completion callback. The easing argument indicates the name of the easing effect that you want to use during transitions. There are two built-in values: linear and swing. Other easing options can be added using a plug-in.

Here's a sample call for the animate method:

Copy Code

```
$("#Element1").animate(
   { width: "70%",
     opacity: 0.4,
     borderWidth: "10px"
   },
   3000);
```

At the end of the animation, the matching element will have the specified width, opacity, and border width. The animation will complete in three seconds and vary CSS attributes from their current values up or down to the target value.

Note that CSS properties must be specified using camelCase, namely the first character is lower case and the first character of each following word is upper case. As an example, to animate the border of an element you should use "borderWidth" instead of the CSS original property name of border-width.

The animate method also supports relative animations, such as increasing (or decreasing) a property value by a percentage or by a fixed value. For example, the following code snippet shows how to increase the width of an element by 20 percent and darken its content by a value of 0.4. I've also specified that all changes should be applied in two seconds.

Copy Code

```
$("#Panel1").animate(
   { width: "+=20%",
     opacity: "-=0.4"
   },
   2000);
```

Finally, note that all animations in jQuery are automatically queued and execute in order, one after the next. Queued animations are the necessary precondition to be able to chain multiple calls on elements in a wrapped set. Queued animations, however, are just the default behavior. As a developer, you are given the tools to make some animations run in parallel.

To gain some parallelism, you create an animation queue using an overload of the animate method:

Copy Code

```
function animate( prop, options )
{
   …
}
```

As in the previous signature, the first argument indicates the set of style attributes that you wish to animate and the values you want to reach.

The second argument indicates a set of options with which to configure the animation. Options include duration, easing, completion callback, and a Boolean value to indicate whether it is a queued animation—true is the default.

By setting the queue attribute to false in the options of the animate call, you run the animation immediately without queuing. Let's consider the following code snippet:

Copy Code

```
$("#div1").animate({ width: "90%" }, {queue:false, duration:5000 });
$("#div1").animate({ fontSize: '10em' }, 1000);
$("#div1").animate({ borderWidth: 5 }, 1000);
```

The first animation is not queued and runs immediately. The second is not queued but it is considered the first in the queue so it runs as well. The net effect is that the first two animations start together. The third animation is queued and begins as soon as the second animation terminates—one second later. Because the first animation takes five seconds, the border animation also runs in parallel with the animation that changes the width of the element.

AJAX Capabilities

What would a modern JavaScript library be without a solid infrastructure for AJAX asynchronous calls? In jQuery, the AJAX support is based on the ajax function, through which you can control all aspects of a Web request. Here's a common way of invoking the ajax function:

Copy Code

```
$.ajax(
 {
   type: "POST",
   url: "getOrder.aspx",
   data: "id=1234&year=2007",
   success: function(response) {
     alert( response );
   }
 }
);
```

The function accepts a list of parameters grouped in a single object. Feasible options are type, url, data, dataType, cache, async, username, password, timeout, and ifModified.

In particular, the dataType parameter indicates the type of the expected response, whereas the cache parameter (if set to false) forces requested resources not to be cached by the browser. Other parameters such as type, password, username, and url are self-explanatory.

The options for the ajax function also include a number of optional callbacks to be invoked before any of the most relevant steps in the lifecycle of the underlying XMLHttpRequest object. The success callback indicates the completion callback. The callback function receives the response as its sole argument. Other callbacks are error, beforeSend, and complete. The complete callback runs at the end of the request when either the success or error callbacks have been called.

Interestingly, the ajax function lists a callback to preprocess the Web response before it is returned to the calling code. This callback is dataFilter and handles the raw response downloaded by XMLHttpRequest. The purpose of the callback is to filter the response so that only sanitized data compliant with the expected data type is returned to the caller. The callback function receives two arguments: the raw data returned from the server and the value assigned to the dataType parameter.

When using the jQuery library, you hardly use the ajax function directly. You would more often end up using some of the AJAX helpers such as getScript, load, or getJSON.

The following code shows how to load a script file on demand. The script is automatically executed upon loading:

Copy Code

```
$.getScript("sample.js");
```

Another quite useful piece of code is the load function. The load function downloads HTML markup and automatically injects it in the current DOM. The following code shows how to populate a menu programmatically:

Copy Code

```
$("#menu").load("menu.aspx");
```

The content of the URL is attached to the DOM subtree rooted in any elements that match the selector. The load method defaults to a GET request, but you can change it to POST by simply adding a data argument, as you see here:

Copy Code

```
$("#links").load(
   "menu.aspx",
   { topElement: "Books" },
   function()
   {
       // completion callback code
   }
);
```

As shown above, a callback can also be specified to execute upon completion of the download.

It is also possible to specify a jQuery selector in the URL so that the incoming markup is pre-filtered to select only matching elements. The syntax simply entails that you add a selector expression to the URL. Here's an example that extracts all <li> elements from a <ul> element named menuItems:

Copy Code

```
$("#links").load("/menu.aspx ul#menuItems li");
```

Finally, you have get, post, and getJSON functions to perform direct GET and POSTs and to get JSON content from a URL.

Client Caching

A client-side cache is essential in nontrivial JavaScript code. In this context, a cache is an array where developers can store data that relates to a given DOM element. In the following code, a URL is invoked to determine whether the content of a textbox is valid, and then the response is cached in an array element named IsValid.

Copy Code

```
var url = "...";
var contentIsValid = $.get(url);
$("#TextBox1").data("IsValid", contentIsValid);
```

Each DOM element may have its own local cache. In jQuery, you use the data method on the elements in a wrapped set.

To read back the content of the store, you use the same data function with just one argument—the name of the element:

Copy Code

```
alert($("#grid").data("Markup"));
```

Elements added to the cache can be removed by using the removeData function.

It would be nice to know why it's better to use the data function than expando properties. An expando property adds custom information to a DOM element using a nonstandard HTML attribute. It is definitely a type of client-side caching, but some browsers might not like custom attributes. In this case, you typically resort to using standard attributes such as alt or rel in a nonstandard way. A client-side caching framework such as the data function stores data in plain old JavaScript objects and maintains a link to the target DOM element using a dictionary. Done this way, no conflict with browsers is possible.

Summing Up

JavaScript remained nearly unchanged for about ten years. While a significant overhaul of the language is in order, it's anybody's guess when it will be fully defined and, more importantly, whether browsers will support it. Last year, a consensus was reached around a version of the language that sits in between the current language and the proposed new ECMAScript 4 language which was significantly different from today's JavaScript. The new project that should generate JavaScript 2.0 is called Harmony.

Whatever way you look at it, cross-browser libraries seem to be the only reliable way to plan efficient and effective client-side development, at least for the next few years. In this context, jQuery is an excellent library that will be integrated into upcoming Microsoft Web development tools. Take a close look at it; you'll be glad you did! Happy jQuery coding!

Send your questions and comments for Dino to cutting@microsoft.com.

Dino Esposito is an architect at IDesign and the co-author of *Microsoft .NET: Architecting Applications for the Enterprise (Microsoft Press, 2008)*. Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos