

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

Sometimes other people make our lives easier. Sometimes we pay them for this service. And sometimes we can actually get something for nothing. The Internet is full of **Error! Hyperlink reference not valid.** tools that people have made available for others to use without charge, free for the taking.

Whether the motivation behind making their labors freely available is a matter of seeking recognition, resume building, free advertising for other services, bragging rights, or just plain old-fashioned altruism, we can gratefully take advantage of these tools. jQuery is one such tool.

Introducing jQuery

jQuery, a self-professed "new type" of JavaScript library, operates from a slightly different viewpoint than other toolkits like Dojo and Prototype. It purports to change the way that you write JavaScript, and quite truly, adopting the jQuery philosophy can make a huge impact on how you develop the script for your pages.

In either case, import the jQuery script file into any pages on which you wish to use jQuery. For the purposes of this section, we'll use the uncompressed (readable) version of the script file, place it in the same folder as our example pages (for easy importing), and name it jquery.js.

jQuery Basics

Before diving into making Ajax requests with jQuery, let's take a look at some of the basic concepts that we need to have under our belts before beginning to make sense of how jQuery operates. This will by no means be a complete primer on jQuery -- that would take much more space than we have allotted here -- but it should give you an idea of the philosophy behind jQuery's modus operandi.

The jQuery wrapper

Other libraries that we have seen, particularly Prototype, operate by introducing new classes and by extending the built-in JavaScript classes in order to augment the capabilities of the script on our pages. jQuery takes a different approach.

Rather than extending classes, jQuery provides a new class, appropriately named jQuery, that serves as a wrapper around other objects in order to provide extended operations upon those objects. The concept of a wrapper object is not foreign to advanced **Error! Hyperlink reference not valid.** of object-oriented programs. This pattern is often used as an adapter to present an interface for manipulating an object that is different from the original object's interface.

In jQuery, most operations are performed by using the jQuery wrapper around a set of items and calling wrapper methods that operate upon the wrapped items. In order to make expressions and statements containing jQuery wrappers terser, the jQuery class is mapped to \$. This is not to be confused with Prototype's use of \$(), which serves a completely different purpose.

The jQuery object can wrap a number of different object types, and what it can do for us depends on what has been wrapped. For example, we can wrap an HTML snippet:

```
$("#<p>What's cooking?</p>")
```

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

This constructs a DOM fragment from the HTML that we can then operate upon with jQuery's methods. For example, if we wanted to append this fragment to the end of the document, we could use

```
$("<p>What's cooking?</p>").appendTo("body");
```

As Ajax developers who often have a need to generate new DOM elements, the advantages of this convenient and short means to effect such additions should be readily apparent.

In addition to adding new DOM elements, we often find ourselves needing to manipulate existing elements in our pages. The jQuery wrapper also allows us to wrap existing elements by passing a string to the `$()` wrapper that provides a number of ways to identify the items to be wrapped: CSS selectors, XPath expressions, and element names. We'll be using CSS selectors a great deal within our example code. Consider the following:

```
$("div")
```

This will cause all `<div>` elements in the document to be wrapped for manipulation. Another example is:

```
$("#someId")
```

This wraps the DOM element with the id of `someId` for manipulation.

Here's yet a third example:

```
$(".someClass")
```

This will wrap all elements, regardless of type, that possess the CSS class name of `someClass`.

The authors of jQuery were very clever in using CSS selectors and XPath to identify target elements as opposed to inventing some jQuery-specific syntax that users of jQuery would be forced to adopt. By using mechanisms that we, as page developers, are already familiar with, they have made it far easier for us to adopt and use jQuery to identify the elements that we wish to manipulate.

It is also possible to wrap other items such as elements and functions. We'll be seeing examples later in this section.

Chaining jQuery Operations

jQuery sensibly allows us to string together numerous operations into a single expression. Most of the jQuery wrapper methods return a reference to the jQuery wrapper object itself so that we can just keep tacking operations onto a single expression when we need to perform multiple manipulations on the wrapped object(s).

Consider the case where we might want to add a CSS class to an element (whose id is something) and then cause it to be shown (assuming it was initially hidden). Rather than

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
$('#something').addClass('someClass');  
$('#something').show();
```

we would write:

```
$('#something').addClass('someClass').show();
```

Executing Code When the Document Is Ready

Frequently on our pages, we need some initialization code to execute in order to prepare the page before the user gets a chance to interact with it. Generally we use the window's onload event handler for such initializations. This guarantees that the page has completed loading prior to executing the onload code, thereby guaranteeing that the DOM elements exist and are ready for manipulation.

But one problem with relying on onload is that not only does it wait until the document body has been loaded, but it also waits for images to load. Since images must be fetched from the **Error! Hyperlink reference not valid.** if the browser has not cached them, this can sometimes extend the point at which the initialization code runs far beyond the point at which the document has been loaded and the code is safe to execute.

jQuery solves this problem for us by introducing the concept of the "document ready handler." This mechanism causes a function to execute when the document has loaded but prior to waiting for any images and the onload event handler.

The syntax for employing this mechanism is to wrap the document element and to call the ready(). method on the wrapped document:

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

Whatever function is passed to ready() will execute when the DOM is ready for manipulation. Note that when you use both the ready mechanism and an onload event handler on a page, both handlers will execute, with the ready event handler triggered prior to the onload event handler.

A shorthand notation for a ready() handler can be used by wrapping a function in the jQuery wrapper. The code fragment

```
$(function);
```

is equivalent to the code fragment for declaring a ready() handler that was presented earlier.

Using jQuery and Prototype Together

Prototype is a very popular library, and jQuery is rapidly gaining ground. As such, it's not unlikely that page authors might wish to use the power of both libraries on the same page.

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

In general, jQuery follows best-practice guidelines and avoids polluting the global namespace; for example, by placing such constructs as utility functions within the jQuery namespace. But one area of conflict, which we've already alluded to earlier, is the use of the \$ as a global name.

jQuery, being a good library citizen, has anticipated this issue. When using Prototype and jQuery on the same page, calling the jQuery utility function `jQuery.noConflict()` any time after both libraries have been loaded will cause the functionality of the \$ name to revert to Prototype's definition.

jQuery functionality will still be available through the jQuery namespace, or you could define your own shorthand alias. For those times when you use jQuery together with Prototype, the jQuery documentation suggests the following alias:

```
var $j = jQuery;
```

That's enough preliminaries!

We'll see more use of jQuery methods within the solutions in this section. But even so, we'll only be lightly touching on jQuery's capabilities. If after reading these solutions you find yourself intrigued by jQuery's capabilities, we strongly urge you to visit <http://docs.jquery.com/> to read the extensive **Error! Hyperlink reference not valid.** and find out what other capabilities jQuery has to offer.

Asynchronous Loading with jQuery

jQuery provides a fairly large number of methods to make Ajax requests. Some are simple and useful high-level methods that initiate Ajax requests to perform some of the most commonly required tasks. Others are more low-level, providing control over every aspect of the Ajax request.

We'll employ a representative handful of these methods in the solutions within this section. First, let's tackle one of the most common of Ajax interactions: obtaining dynamic content from the **Error! Hyperlink reference not valid..**

Problem

Let's imagine that we own an eFridge -- a hypothetical high-tech refrigerator that not only keeps track of what its contents are, but also provides an Internet interface that server software can use to communicate and interact with the eFridge.

The imaginary technology used by the eFridge to keep track of its inventory is unimportant. It could be bar-code scanning, RFID (Radio Frequency Identification) tags, or some yet-to-be-imagined technology. All we care about as page authors is that we have a server component to which we can make requests in order to obtain information about the state of our food!

The page we'll focus on will present a list of the items that are in our eFridge. Upon clicking on an item in this list, more information about the item will be displayed.

For this problem, we'll assume that the page was prepopulated with the list of items by whatever server-side templating mechanism generated our page. In the next section, we'll see a technique to obtain this list dynamically from the server.

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

Solution

To begin, in order to use jQuery on a page it is necessary to import the jQuery library:

```
<script type="text/javascript" src="jquery.js"></script>
```

The list of items in the eFridge, which we're assuming was generated on our behalf by some server-side mechanism, is presented in a select element:

```
<form>
  <select id="itemsControl" name="items" size="10">
    <option value="1">Milk</option>
    <option value="2">Cole Slaw</option>
    <option value="3">BBQ Sauce</option>
    <option value="4">Lunch Meat</option>
    <option value="5">Mustard</option>
    <option value="6">Hot Sauce</option>
    <option value="7">Cheese</option>
    <option value="8">Iced Tea</option>
  </select>
</form>
```

For the purpose of this example, we're only showing eight items. The average refrigerator would probably contain more than this, but we all know fast-food junkies whose refrigerator contents are sometimes pretty sparse.

The server (perhaps some "eFridge driver") assigns each item an identification number that is used to uniquely identify each item_in this case, a simple sequential integer value. This identifier is set as the value for each <option> representing an item.

Even though we know that we need the select control to react to user input, note that no handlers are declared within the markup that creates the <select> element. This brings up another philosophy behind the design of jQuery.

One of the goals of jQuery is to make it easy for page authors to separate script from document markup, much in the same manner that CSS allows us to separate presentation from the document markup. Granted, we could do it ourselves without jQuery's help -- after all, jQuery is written in JavaScript and doesn't do anything we couldn't do -- but jQuery does a lot of the work for us, and is designed with the goal of easily separating script from document markup. So, rather than adding an onchange event handler directly in the markup of the <select> element, we'll use jQuery's help to add it under script control.

We can't manipulate the DOM elements on our page until after the document is ready, so in the <script> element in our page header, we'll institute a jQuery ready() handler as we previously discussed. Within that handler, we'll use jQuery's method to add a change handler to an element, as shown in the following code fragment:

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
$(document).ready(function(){
    $('#itemsControl').change(showItemInfo);
});
```

In the ready() handler, we create a jQuery instance that wraps the <select> element, which we have given the id of itemsControl. We then use the jQuery change() method, which assigns its parameter as the change handler for the wrapped element.

In this case, we've identified a function named showItemInfo(). It's within this function that we'll make the Ajax request for the item that is selected from the list:

```
function showItemInfo() {
    $('#div#itemData').load(      #1
        'fetchItemData.jsp',      #2
        {itemId: $(this).val()}   #3
    );
}
```

- #1 Wraps element and invokes load method
- #2 Identifies server-side resource
- #3 Obtains item id and passes as parameter

jQuery provides a fair number of different ways to make Ajax requests to the server. For the purposes of this solution, we'd like to fetch a pre-formatted snippet of HTML from the server (containing the item data) and load it into a waiting element, that is, a <div> element having an id of itemData. The jQuery load() method (#1) serves this requirement perfectly.

This method fetches a response from a URL provided as its first parameter and inserts it into the wrapped DOM element. A second parameter to this function allows us to pass an object whose properties serve as the parameters for the request. A third parameter can be used to specify a callback function to be executed when the request completes.

First, we wrap a DOM element (#1) identified by the CSS selector div#itemData, which is an empty <div> element into which we want the item data to be loaded. Then, using the load() method, we provide the URL to a JSP page (#2) that will fetch the item data identified by the itemId request parameter supplied in the second method parameter (#3).

The value of that parameter needs to be the value of the option that the user clicks on in the <select> element. Because the <select> element is set as the function context of the change handler, it is available to it via the this reference. We wrap that reference and use JQuery's val() method to obtain the current selected value of the control (#3).

Since all we want to do is to load the item data into the DOM, we have no need for a callback and omit the third parameter to the load() method.

That's all there is to it.

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

jQuery's capabilities have taken a very common procedure that might have taken a nontrivial amount of code to implement and allow us to perform it with very few lines of simple code. The JSP page that gets invoked by this handler uses the value of the itemId request parameter to fetch the info for the corresponding item and formats it as HTML to be displayed on the page.

Our finished page, shown here after selecting a refrigerator item:



Got milk?
is laid out in its entirety in this listing:

```
<html>
<head>
<title>What's for dinner?</title>
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
$(document).ready(function(){
    $('#itemsControl').change(showItemInfo);
});

function showItemInfo() {
    $('#itemData').load(
        'fetchItemData.jsp',
        {itemId: $(this).val()}
    );
}
</script>
<style type="text/css">
form,#itemData {
```

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
float: left;
}
</style>
</head>

<body>
<form>
<select id="itemsControl" name="items" size="10">
<option value="1">Milk</option>
<option value="2">Cole Slaw</option>
<option value="3">BBQ Sauce</option>
<option value="4">Lunch Meat</option>
<option value="5">Mustard</option>
<option value="6">Hot Sauce</option>
<option value="7">Cheese</option>
<option value="8">Iced Tea</option>
</select>
</form>

<div id="itemData"></div>
</body>
</html>
```

Discussion

This section introduced us to one of JQuery's means of performing Ajax requests, the `load()` method.

The JQuery `load()` method is very well suited for use with server-side templating languages such as JSP and PHP that make it a snap to format and return HTML as the response. The `fetchItemData.jsp` file, as well as the Java classes that fake the eFridge functionality, are downloadable.

A few other important JQuery features are also exposed in this solution. For example, we used a `ready()` handler to trigger the execution of code that must execute before a user is allowed to interact with the page, but after the entire DOM has been constructed for the page.

We also saw the `val()` method, which returns the value of the wrapped input element. If more than one element is wrapped, the value of the first matched element is returned by this method.

In this solution, we assumed that the original list of eFridge contents was generated by whatever server-side resource produced the page; a JSP template, for example. That's a common expectation for a web application, but in the interest of exploring more of JQuery's Ajax abilities, let's pretend that we need to fetch that list dynamically upon page load in the next problem.

Fetching Dynamic Data with jQuery

In the previous section we were introduced to the JQuery `load()` method, which made it extremely easy to perform the common task of fetching an HTML snippet to load into a DOM element. While the utility of this method cannot be dismissed, there are times when we might want to exert more control over the Ajax request process, or to obtain data (as opposed to preformatted HTML) from the **Error! Hyperlink reference not valid.**

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

In this section we'll explore more of what jQuery has to offer in the Ajax arena.

Problem

We wish to augment the code of the previous section to obtain the list of items in the eFridge from a page-initiated asynchronous request.

Solution

Reviewing the previous solution, we can readily see that in order to load the select options dynamically, the changes that we would need to make are to remove the <option> elements from the <select> element and to add code to the ready() handler to fetch and load the items. But before we embark upon that effort, we're going to change the way that we coded the showItemInfo() handler function if for no other reason than as an excuse to further explore jQuery's capabilities. Rather than using the load() method of the jQuery wrapper, we're going to use one of jQuery's utility functions:

`$.get()` .

Hey, wait a minute! What's that period character doing in there? That's not the \$() wrapper that we've been using up to now!

Not only does jQuery provide the wrapper class that we've make good use of up to this point, but it also provides a number of utility functions, many implemented as class methods of the \$ wrapper class.

If the notation \$.functionName() looks odd to you, imagine the expression without using the \$ alias for the jQuery function:

`jQuery.get()`;

Okay, that looks more familiar. The \$.get() function is defined as a class method; that is, a function property of the jQuery wrapper function. Although we know them to be class methods on the jQuery wrapper class, jQuery terms these methods utility functions and to be consistent with the jQuery terminology, that's how we'll refer to them in this section.

The \$.get() utility function accepts the same parameters as the load() method: the URL of the request, a hash of the request parameters, and a callback function to execute upon completion of the request. When using this utility function, because there is no object being wrapped that will automatically be injected with the response, the callback function, although an optional parameter, is almost always specified. It is the primary means for causing something to happen when the request completes.

It should also be noted that the callback function can be specified as the second parameter to this utility function when no request parameters need to be passed. Internally, jQuery uses some JavaScript sleight of hand to ensure that the parameters are interpreted correctly.

The rewritten showItemInfo() handler using this utility function is as follows:

```
function showItemInfo() {  
    $.get('fetchItemData.jsp',
```

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
{itemId: $(this).val()},
function(data) {
    $('#itemData').empty().append(data);
}
);
}
```

Aside from using the `$.get()` utility function, another change to the code of the previous solution was the addition of a callback function as the third parameter, which we use to insert the returned HTML into the `itemData` element.

In doing so, we make use of two more wrapper methods: `empty()`, which clears out the wrapped DOM element, and `append()`, which adds to the wrapped element the HTML snippet passed to the callback in the `data` parameter.

Now we're ready to tackle loading the `<options>` from data that we will obtain from the server when the document is loading. In this case, we're going to obtain the raw data for the options from the server in the form of a JavaScript hash object. We could return the data as XML, but we'll opt to use JSON, which is easier for JavaScript code to digest.

jQuery comes to our rescue once again with a utility function that is well suited to this common task: the `$.getJSON()` utility function. This function accepts the now-familiar trio of parameters: a URL, a hash of request parameters, and a callback function.

The advantage that the `$.getJSON()` utility function brings to the table is that the callback function will be invoked with the already-evaluated JSON structure. We won't have to perform any evaluation of the returned response. How handy!

Using this utility method, the following line gets added to the document's `ready()` handler:

```
$.getJSON('fetchItemList.jsp',loadItems);
```

A JSP page named `fetchItemList.jsp` is used as the URL, and a function named `loadItems()` (whose definition we'll be looking at next) is supplied as the callback function. Note that, since we don't need to pass any request parameters, we can simply omit the object hash and provide the callback as the second parameter.

The `loadItems()` function is defined as

```
function loadItems(itemList) {
#1
    if (!itemList) return;
    for(var n = 0; n < itemList.length; n++) {
        $('#itemsControl').get(0).add(
#2
            new Option(itemList[n].name,itemList[n].id),
#3
        );
    }
}
```

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
        document.all ? 0 : null
    );
}
}
```

- #1 Invokes callback with evaluated JSON structure
- #2 Locates select element
- #3 Adds new option

Recall that the \$.getJSON() utility function invokes the callback with the JSON response already evaluated as its JavaScript equivalent (#1). In our solution, the fetchItemList.jsp page will return a response that contains

```
[
  {id:'3',name:'BBQ Sauce'},
  {id:'5',name:'Mustard'},
  {id:'7',name:'Cheese'},
  {id:'2',name:'Cole Slaw'},
  {id:'4',name:'Lunch Meat'},
  {id:'8',name:'Iced Tea'},
  {id:'6',name:'Hot Sauce'},
  {id:'1',name:'Milk'}
]
```

When our callback is invoked, this response string will already have been converted to an array of JavaScript objects, each of which contains an id and a name property, courtesy of jQuery. Each of these objects will be used to construct a new <option> element to be added to the select control (#3). In order to add an option to the <select> element, we need a reference to that control's DOM element. We could just use document.getElementById() or \$(), but we have chosen to do it the jQuery way with the get() wrapper method:

\$('#itemsControl').get(0)

This method, when passed no parameters, returns an array of all the elements matched by the CSS selector of the jQuery wrapper on which it is invoked. If we only want one of those matches, we can specify a zero-based index as a parameter. In our case, we know that there will only be a single match to the selector because we used an id, so we specify an index of 0 to return the first matched element.

The code for the entire page, with changes from the previous solution highlighted in bold, is shown here.

```
<html>
  <head>
    <title>What's for dinner?</title>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function(){
```

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

```
$.getJSON('fetchItemList.jsp',loadItems);j
$('#itemsControl').change(showItemInfo);
});

function loadItems(itemList) {
  if (!itemList) return;
  for(var n = 0; n < itemList.length; n++) {
    $('#itemsControl').get(0).add(
      new Option(itemList[n].name,itemList[n].id),
      document.all ? 0 : null
    );
  }
}

function showItemInfo() {
  $.get('fetchItemData.jsp',
    {itemId: $(this).val()},
    function(data) {
      $('#itemData').empty().append(data);
    }
  );
}
</script>
</head>

<body>
  <form style="float:left">
    <select id="itemsControl" name="items" size="10">
    </select>
  </form>
  <div id="itemData" style="float:left"></div>
</body>
</html>
```

Discussion

This section exposed us to more of jQuery's abilities in the areas of DOM manipulation and traversal, as well as Ajax request initiation. We saw the jQuery \$.get() utility function, which made it easy for us to make Ajax requests using the HTTP GET method. A corresponding utility function named \$.post() with the exact same function signature makes it equally easy to submit POST requests via Ajax. As both utility functions use the same parameter signature -- most notably the request parameter hash -- we can easily switch between which HTTP method we'd like to use without having to get bogged down in the details of whether the request parameters need to be encoded in the query string (for GET) or as the body of the request (for POST).

Another Ajax utility function, \$.getJSON(), makes it incredibly easy for us to use the power of the server to format and return JSON notation. The callback for this operation is invoked with the JSON string already evaluated, preventing us from having to work with the vagaries of the JavaScript eval() function ourselves.

Getting Started with jQuery

By Bear Bibeault, Dave Crane, Jord Sonneveld, and Ted Goddard

For occasions where we might wish to exert more control over, and visibility into, an Ajax request, jQuery provides a versatile utility function named `$.ajax()`. The online documentation provides more details about how to use this low-level utility function.

We also saw a handful of the powerful DOM manipulation wrapper methods such as `get()`, `empty()`, `val()`, and `append()`, all geared toward making it easy for us -- as Ajax page **Error! Hyperlink reference not valid.** -- to manipulate the page DOM.

Taking the next steps

This is all just barely plumbing the depth of jQuery capabilities. For example, space prevents us from exploring the effects API, which provides fading, sliding, flashing, hovering. and even the ability to provide your own animations. You are urged to visit <http://jquery.com/> for more information on jQuery and how it can help you write powerful Ajax applications.

Additionally, advanced developers might be interested in jQuery's plug-in API. This API is one of jQuery's most powerful assets, as anyone can extend the toolkit in a snap. For more information, please see <http://docs.jquery.com/Plugins/Authoring>.