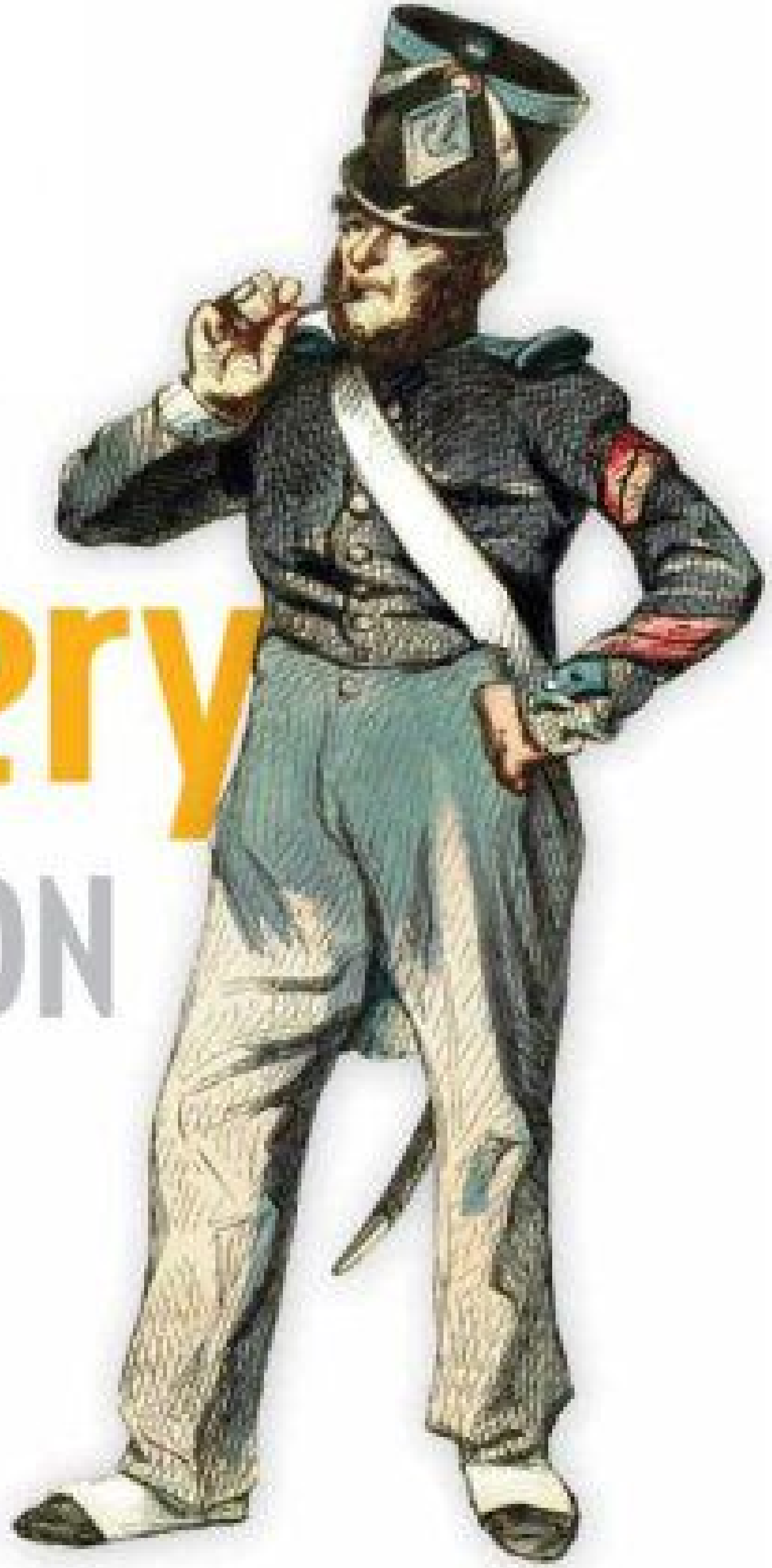
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8

jQuery IN ACTION

SECOND EDITION



MANNING

Table of Contents

8 Talk to the server with Ajax

This chapter covers

- A brief overview of Ajax
- Loading preformatted HTML from the server
- Making general `GET` and `POST` requests
- Exerting fine-grained control over requests
- Setting default Ajax properties
- Handling Ajax events

It can be successfully argued that no single technology shift has transformed the landscape of the web more than Ajax. The ability to make asynchronous requests back to the server without the need to reload entire pages has enabled a whole new set of user-interaction paradigms and made DOM-scripted applications possible.

Ajax is a less recent addition to the web toolbox than many people may realize. In 1998, Microsoft introduced the ability to perform asynchronous requests under script control (discounting the use of `<iframe>` elements for such activity) as an ActiveX control as part of the creation of Outlook Web Access (OWA). Although OWA was a moderate success, few people seemed to take notice of the underlying technology.

A few years passed, and a handful of events launched Ajax into the collective consciousness of the web development community. The non-Microsoft browsers implemented a standardized version of the technology as the `XMLHttpRequest` (XHR) object; Google began using XHR; and, in 2005, Jesse James Garrett of Adaptive Path coined the term *Ajax* (for Asynchronous JavaScript and XML).

As if they were only waiting for the technologies to be given a catchy name, the web development masses suddenly took note of Ajax in a *big* way, and it has become one of the primary tools by which we can enable DOM-scripted applications.

In this chapter, we'll take a brief tour of Ajax (if you're already an Ajax guru, you might want to skip ahead to section 8.2), and then we'll look at how jQuery makes using Ajax a snap.

Let's start off with a refresher on what Ajax technology is all about.

8.1 Brushing up on Ajax

Although we'll take a quick look at Ajax in this section, please note that this isn't intended as a complete Ajax tutorial or an Ajax primer. If you're completely unfamiliar with Ajax (or worse, think that we're talking about a dishwashing liquid or a mythological Greek hero), we encourage you to familiarize yourself with the technology through resources that are geared toward teaching you *all* about Ajax; the Manning books *Ajax in Action* and *Ajax in Practice* are both excellent examples.

Some people may argue that the term *Ajax* applies to *any* method of making server requests without the need to refresh the user-facing page (such as by submitting a request to a hidden `<iframe>` element), but most people associate the term with the use of `XMLHttpRequest` (XHR) or the Microsoft XMLHTTP ActiveX control.

A diagram of the overall process, which we'll examine one step at a time, is shown in figure 8.1.

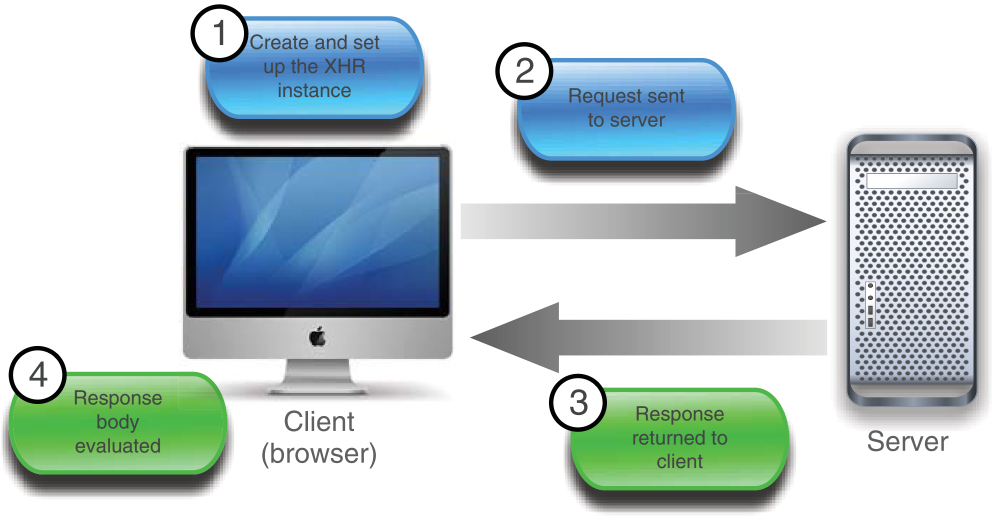


Figure 8.1 **The lifecycle of an Ajax request as it makes its way from the client to the server and back again**

Let's take a look at how those objects are used to generate requests to the server, beginning with creating an XHR instance.

8.1.1 *Creating an XHR instance*

In a perfect world, code written for one browser would work in all commonly used browsers. We've already learned that we don't live in that world, and things are no different when it comes to Ajax. There is a standard way to make asynchronous requests via the JavaScript XHR object, and an Internet Explorer proprietary way that uses an ActiveX control. With IE 7, a wrapper that emulates the standard interface is available, but IE 6 requires divergent code.

> **NOTE** jQuery's Ajax implementation—which we'll be addressing throughout the remainder of this chapter—doesn't use the Internet Explorer wrapper, citing issues with improper implementation. Rather, it uses the ActiveX object when available. This is good news for us! By using jQuery for our Ajax needs, we know that the best approaches have been researched and will be utilized.

Once created, the code to set up, initiate, and respond to the request is relatively browser-independent, and creating an instance of XHR is easy for any particular browser. The problem is that different browsers implement XHR in different ways, and we need to create the instance in the manner appropriate for the current browser.

But rather than relying on detecting which browser a user is running to determine which path to take, we'll use the preferred technique of *feature detection* that we introduced in chapter 6. Using this technique, we try to figure out what the browser's features are, not which browser is being used. Feature detection results in more robust code because it can work in any browser that supports the tested feature.

The code in listing 8.1 shows a typical idiom used to instantiate an instance of XHR using this technique.

Listing 8.1 Capability detection resulting in code that can use Ajax in many browsers

```
var xhr;
if (window.ActiveXObject) {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
else  if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
}
else {
  throw new Error("Ajax is not supported by this browser");
}
```

Tests to see if ActiveX is present

Tests to see if XHR is defined

Throws error if there's no Ajax support

After it's created, the XHR instance sports a conveniently consistent set of properties and methods across all supporting browser instances. These properties and methods are shown in table 8.1, and the most commonly used of these will be discussed in the sections that follow.

Table 8.1 `XMLHttpRequest (XHR)` methods and properties

Methods	Description
`abort()`	Causes the currently executing request to be cancelled.
`getAllResponseHeaders()`	Returns a single string containing the names and values of all response headers.
`getResponseHeader(name)`	Returns the value of the named response header.
`open(method,url,async, username,password)`	Sets the HTTP method (`GET` or `POST`) and destination URL of the request. Optionally, the request can be declared synchronous, and a username and password can be supplied for requests requiring container-based authentication.
`send(content)`	Initiates the request with the specified (optional) body content.
`setRequestHeader(name,value)`	Sets a request header using the specified name and value.
Properties	**Description**
`onreadystatechange`	The event handler to be invoked when the state of the request changes.
`readyState`	An integer value that indicates the current state of the active request as follows: 0 = UNSENT 1 = OPENED 2 = HEADERS_RECEIVED 3 = LOADING 4 = DONE
`responseText`	The body content returned in the response.
`responseXML`	If the body content is identified as XML, the XML DOM created from the body content.
`status`	The response status code returned from the server. For example: 200 for *success* or 404 for *not found*. See the HTTP specification[a] for the full set of codes.
`statusText`	The status text message returned by the response.

[a] HTTP 1.1 status code definitions from RFC 2616: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.

NOTE Want to get it from the horse's mouth? The XHR specification can be found at http://www.w3.org/TR/XMLHttpRequest/.

Now that we've got an XHR instance created, let's look at what it takes to set up and fire off the request to the server.

8.1.2 Initiating the request

Before we can send a request to the server, we need to perform the following setup steps:

1. Specify the HTTP method, such as (`POST` or `GET`)
2. Provide the URL of the server-side resource to be contacted
3. Let the XHR instance know how it can inform us of its progress
4. Provide any body content for `POST` requests

We set up the first two items by calling the `open()` method of XHR as follows:

```
xhr.open('GET','/some/resource/url');
```

Note that this method doesn't cause the request to be sent to the server. It merely sets up the URL and HTTP method to be used. The `open()` method can also be passed a third Boolean parameter that specifies whether the request is to be asynchronous (if `true`, which is the default) or synchronous (if `false`). There's seldom a good reason to make the request synchronous (even if doing so means we don't have to deal with callback functions); after all, the asynchronous nature of the request is usually the whole point of making a request in this fashion.

In the third step, we must provide a means for the XHR instance to tap us on the shoulder to let us know what's going on. We accomplish this by assigning a callback function to the `onreadystatechange` property of the XHR object. This function, known as the *ready state handler*, is invoked by the XHR instance at various stages of its processing. By looking at the settings of the other properties of XHR, we can find out exactly what's going on with the request. We'll take a look at how a typical ready state handler operates in the next section.

The final steps to initiating the request are to provide any body content for `POST` requests and send it off to the server. Both of these steps are accomplished via the `send()` method. For `GET` requests, which typically have no body, no body content parameter is passed, as follows:

```
xhr.send(null);
```

When request parameters are passed to `POST` requests, the string passed to the `send()` method must be in the proper format (which we might think of as *query string* format) in which the names and values are properly URI encoded. URI encoding is beyond the scope of this section (and, as it turns out, jQuery is going to handle all of that for us), but if you're curious, do a web search for the term `encodeURIComponent`, and you'll be suitably rewarded.

An example of such a call is as follows:

```
xhr.send('a=1&b=2&c=3');
```

Now let's see what the ready state handler is all about.

8.1.3 Keeping track of progress

An XHR instance informs us of its progress through the ready state handler. This handler is established by assigning a reference to the function to serve as the ready handler to the `onreadystatechange` property of the XHR instance.

Once the request is initiated via the `send()` method, this callback will be invoked numerous times as the request makes transitions through its various states. The current state of the request is available as a numeric code in the `readyState` property (see the description of this property in table 8.1).

That's nice, but more times than not, we're only interested in when the request completes and whether it was successful or not. Frequently we'll see ready handlers implemented using the idiom shown in listing 8.2.

Listing 8.2 Ready state handlers are often written to ignore all but the DONE state

```
xhr.onreadystatechange = function() {
  if (this.readyState == 4) {
    if (this.status >= 200 &&
        this.status < 300) {
      //success
    }
    else {
      //problem
    }
  }
}
```

Ignores all but DONE state

Branches on response status

Executes on success

Executes on failure

This code ignores all but the DONE state, and once that has been detected, examines the value of the `status` property to determine whether the request succeeded or not. The HTTP specification defines all status codes in the 200 to 299 range as success and those with values of 300 or above as various types of failure.

Now let's explore dealing with the response from a completed request.

8.1.4 Getting the response

Once the ready handler has determined that the `readyState` is complete and that the request completed successfully, the body of the response can be retrieved from the XHR instance.

Despite the moniker *Ajax* (where the *X* stands for XML), the format of the response body can be any text format; it's not limited to XML. In fact, most of the time, the response to Ajax requests is a format *other* than XML. It could be plain text or, perhaps, an HTML fragment; it could even be a text representation of a JavaScript object or array in JavaScript Object Notation (JSON) format (which is becoming increasingly popular as an exchange format).

Regardless of its format, the body of the response is available via the `responseText` property of the XHR instance (assuming that the request completes successfully). If the response indicates that the format of its body is XML by including a content type header specifying a MIME type of `text/xml`, `application/xml`, or a MIME type that ends with `+xml`, the response body will be parsed as XML. The resulting DOM will be available in the responseXML property. JavaScript (and jQuery itself, using its selector API) can then be used to process the XML DOM.

Processing XML on the client isn't rocket science, but—even with jQuery's help—it can still be a pain. Although there are times when nothing but XML will do for returning complex hierarchical data, frequently page authors will use other formats when the full power (and corresponding headache) of XML isn't absolutely necessary.

But some of those other formats aren't without their own pain. When JSON is returned, it must be converted into its runtime equivalent. When HTML is returned, it must be loaded into the appropriate destination element. And what if the HTML markup returned contains <script> blocks that need evaluation? We're not going to deal with these issues in this section because it isn't meant to be a complete Ajax reference and, more importantly, because we're going to find out that jQuery handles most of these issues on our behalf.

At this point, you might want to review the diagram of the whole process shown in figure 8.1.

In this short overview of Ajax, we've identified the following pain points that page authors using Ajax need to deal with:

- Instantiating an XHR object requires browser-specific code.
- Ready handlers need to sift through a lot of uninteresting state changes.
- The response body needs to be dealt with in numerous ways, depending upon its format.

The remainder of this chapter will describe how the jQuery Ajax methods and utility functions make Ajax a lot easier (and cleaner) to use on our pages. There are a lot of choices in the jQuery Ajax API, and we'll start with some of the simplest and most-used tools.

8.2 Loading content into elements

Perhaps one of the most common uses of Ajax is to grab a chunk of content from the server and stuff it into the DOM at some strategic location. The content could be an HTML fragment that's to become the child content of a target container element, or it could be plain text that will become the content of the target element.

Setting up for the examples

Unlike all of the example code that we've examined so far in this book, the code examples for this chapter require the services of a web server to receive the Ajax requests to server-side resources. Because it's well beyond the scope of this book to discuss the operation of server-side mechanisms, we're going to set up some minimal server-side resources that send data back to the client without worrying about doing it for real, treating the server as a "black box"; we don't need or want to know how it's doing its job.

To enable the serving of these smoke-and-mirrors resources, you'll need to set up a web server of some type. For your convenience, the server-side resources have been

continued

set up in two formats: Java Server Pages (JSP) and PHP. The JSP resources can be used if you're running (or wish to run) a servlet/JSP engine; if you want to enable PHP for your web server of choice, you can use the PHP resources.

If you want to use the JSP resources but aren't already running a suitable server, instructions on setting up the free Tomcat web server are included with the sample code for this chapter. You'll find these instructions in the chapter8/tomcat.pdf file. And don't be concerned; even if you've never looked at a single line of Java, it's easier than you might think!

The examples found in the downloaded code are set up to use either of the JSP or PHP resources, depending upon which server you have set up.

Once you have the server of your choice set up, you can hit the URL http://localhost:8080/jqia2/chapter8/test.jsp (to check your Tomcat installation) or http://localhost/jqia2/chapter8/test.php (to check your PHP installation). The latter assumes that you have set up your web server (Apache or any other you have chosen) to use the example code root folder as a document base.

When you can successfully view the appropriate test page, you'll be ready to run the examples in this chapter.

Alternatively, if you don't want to run these examples locally, you can run the example code remotely from http://bibeault.org/jqia2.

Let's imagine that, on page load, we want to grab a chunk of HTML from the server using a resource named someResource, and make it the content of a <div> element with an id of someContainer. For the final time in this chapter, let's look at how we'd do this without jQuery's assistance. Using the patterns we set out earlier in this chapter, the body of the onload handler is as shown in listing 8.3. The full HTML file for this example can be found in the file chapter8/listing.8.3.html.

> **NOTE** Again, you must run this example using a web server—you can't just open the file in the browser—so the URL should be http://localhost:8080/jqia2/chapter8/listing.8.3.html. Omit the port specification of :8080 if using Apache, and leave it in if using Tomcat.
>
> In future URLs in this chapter we'll use the notation [:8080] to indicate that the port number might or might not be needed, but be sure *not* to include the square brackets as part of the URL.

Listing 8.3 Using native XHR to fetch and include an HTML fragment

```
var xhr;

if(window.ActiveXObject) {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");
   }
else if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
}
```

```
else {
  throw new Error("Ajax is not supported by this browser");
}

xhr.onreadystatechange = function() {
  if (this.readyState == 4) {
    if (this.status >= 200 && this.status < 300) {
      document.getElementById('someContainer')
        .innerHTML = this.responseText;
    }
  }
}

xhr.open('GET','someResource');
xhr.send();
```

Although there's nothing tricky going on here, that's a non-trivial amount of code; 20 lines, without even counting the blank lines that we added for readability.

The equivalent code we'd write as the body of a ready handler using jQuery is as follows:

```
$('#someContainer').load('someResource');
```

We're betting that we know which code you'd rather write and maintain! Let's take a close look at the jQuery method we used in this statement.

8.2.1 *Loading content with jQuery*

The simple jQuery statement at the end of the previous section easily loads content from the server-side resource using one of the most basic, but useful, jQuery Ajax methods: `load()`. The full syntax description of this method is as follows:

Method syntax: load

`load(url,parameters,callback)`

Initiates an Ajax request to the specified URL with optional request parameters. A callback function can be specified that's invoked when the request completes and the DOM has been modified. The response text replaces the content of all matched elements.

Parameters

`url` (String) The URL of the server-side resource to which the request is sent, optionally modified via selector (explained below).

`parameters` (String|Object|Array) Specifies any data that's to be passed as request parameters. This argument can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.
If specified as an object or array, the request is made using the `POST` method. If omitted or specified as a string, the `GET` method is used.

`callback` (Function) An optional callback function invoked after the response data has been loaded into the elements of the matched set. The parameters passed to this function are the response text, a status string (usually "success"), and the XHR instance.
This function will be invoked once for each element in the wrapped set with the target element set as the function context (`this`).

Returns

The wrapped set.

Though simple to use, this method has some important nuances. For example, when the `parameters` parameter is used to supply the request parameters, the request is made using the `POST` HTTP method if an object hash or array is used; otherwise, a `GET` request is initiated. If we want to make a `GET` request with parameters, we can include them as a query string on the URL. But be aware that when we do so, we're responsible for ensuring that the query string is properly formatted and that the names and values of the request parameters are URI-encoded. The JavaScript `encodeURIComponent()` method is handy for this, or you can employ the services of the jQuery `$.param()` utility function that we covered in chapter 6.

Most of the time, we'll use the `load()` method to inject the complete response into whatever elements are contained within the wrapped set, but sometimes we may want to filter elements coming back as the response. If we want to filter response elements, jQuery allows us to specify a selector on the URL that will be used to limit which response elements are injected into the wrapped elements. We specify the selector by suffixing the URL with a space followed by the selector.

For example, to filter response elements so that only `<div>` instances are injected, we write

```
$('.injectMe').load('/someResource div');
```

When it comes to supplying the data to be submitted with a request, sometimes we'll be winging it with ad hoc data, but frequently we'll find ourselves wanting to gather data that a user has entered into form controls.

As you might expect, jQuery's got some assistance up its sleeve.

SERIALIZING FORM DATA

If the data that we want to send as request parameters come from form controls, a helpful jQuery method for building a query string is `serialize()`, whose syntax is as follows:

Method syntax: serialize

`serialize()`

Creates a properly formatted and encoded query string from all successful form elements in the wrapped set, or all successful form elements of forms in the wrapped set.

Parameters

`none`

Returns

The formatted query string.

The `serialize()` method is smart enough to only collect information from form control elements in the wrapped set, and only from those qualifying elements that are deemed *successful*. A successful control is one that would be included as part of a form submission according to the rules of the HTML specification.[1] Controls such as unchecked checkboxes and radio buttons, dropdowns with no selections, and dis-

[1] HTML 4.01 Specification, section 17.13.2, "Successful controls": http://www.w3.org/TR/html401/interact/forms.html#h-17.13.2.

abled controls aren't considered successful and don't participate in form submission, so they're also ignored by `serialize()`.

If we'd rather get the form data in a JavaScript array (as opposed to a query string), jQuery provides the `serializeArray()` method.

Method syntax: serializeArray

`serializeArray()`

Collects the values of all successful form controls into an array of objects containing the names and values of the controls.

Parameters

`none`

Returns

The array of form data.

The array returned by `serializeArray()` is composed of anonymous object instances, each of which contains a `name` property and a `value` property that contain the name and value of each successful form control. Note that this is (not accidentally) one of the formats suitable for passing to the `load()` method to specify the request parameter data.

With the `load()` method at our disposal, let's put it to work solving some common real-world problems that many web developers encounter.

8.2.2 *Loading dynamic HTML fragments*

Often in business applications, particularly for commerce web sites, we want to grab real-time data from the server in order to present our users with the most up-to-date information. After all, we wouldn't want to mislead customers into thinking that they can buy something that's not available, would we? In this section, we'll begin to develop a page that we'll add to throughout the course of the chapter. This page is part of a web site for a fictitious firm named The Boot Closet, an online retailer of overstock and closeout motorcycle boots. Unlike the fixed product catalogs of other online retailers, this inventory of overstock and closeouts is fluid, depending on what deals the proprietor was able to make that day and what's already been sold from the inventory. It will be important for us to always make sure that we're displaying the latest info!

To begin our page (which will omit site navigation and other boilerplate to concentrate on the lessons at hand), we want to present our customers with a dropdown containing the styles that are currently available and, when a style is selected, display detailed information regarding that style. On initial display, the page will look as shown in figure 8.2.

After the page first loads, a dropdown with the list of styles currently available in the inventory is displayed. When no style is selected, we'll display a helpful message as a placeholder for the selection: "— choose a style —". This invites the user to interact with the dropdown, and when a user selects a boot style from this dropdown, here's what we want to do:

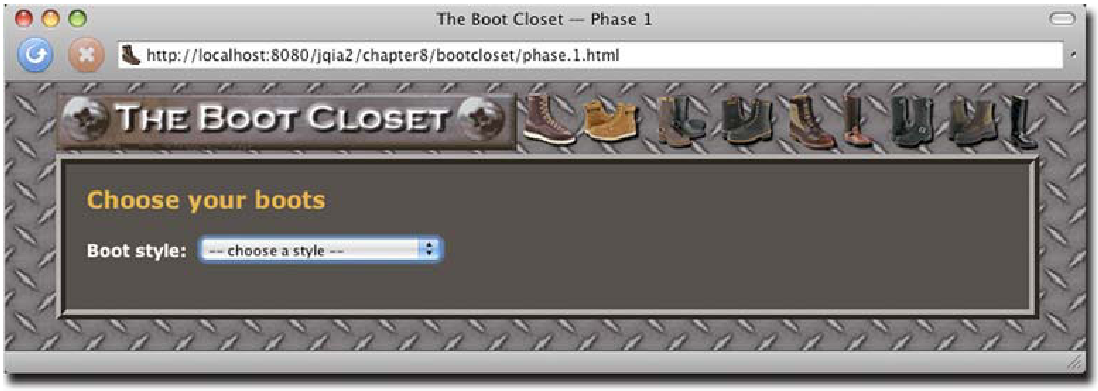

The Boot Closet — Phase 1
http://localhost:8080/jqia2/chapter8/bootcloset/phase.1.html
THE BOOT CLOSET
Choose your boots
Boot style:
-- choose a style --

Figure 8.2 The initial display of our commerce page with a simple dropdown inviting customers to click on it

- Display the detailed information about that style in the area below the dropdown.
- Remove the "— choose a style —" entry; once the user picks a style, it's served its purpose and is no longer meaningful.

Let's start by taking a look at the HTML markup for the body that defines this page structure:

```
<body>
  <div id="banner">
    <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
  </div>
  <div id="pageContent">
    <h1>Choose your boots</h1>
    <div>
      <div id="selectionsPane">
        <label for="bootChooserControl">Boot style:</label>&nbsp;
        <select id="bootChooserControl" name="bootStyle"></select>
      </div>
      <div id="productDetailPane"></div>
    </div>
  </div>
</body>
```

1 Contains selection control

2 Holds place for product details

Not much to it, is there?

As would be expected, we've defined all the visual rendition information in an external style sheet, and (adhering to the precepts of Unobtrusive JavaScript) we've included no behavioral aspects in the HTML markup.

The most interesting parts of this markup are a container 1 that holds the `<select>` element that will allow customers to choose a boot style, and another container 2 into which product details will be injected.

Note that the boot style control needs to have its option elements added before the user can interact with the page. So let's set about adding the necessary behavior to this page.

The first thing we'll add is an Ajax request to fetch and populate the boot style dropdown.

> **NOTE** Under most circumstances, initial values such as these would be handled on the server prior to sending the HTML to the browser. But there are circumstances where prefetching data via Ajax may be appropriate, and we're doing that here, if only for instructional purposes.

To add the options to the boot style control, we define a ready handler, and within it we make use of the handy `load()` method:

```
$('#bootChooserControl').load('/jqia2/action/fetchBootStyleOptions');
```

How simple is that? The only complicated part of this statement is the URL, which isn't really all that long or complicated, which specifies a request to a server-side action named `fetchBootStyleOptions`.

One of the nice things about using Ajax (with the ease of jQuery making it even nicer) is that it's completely independent of the server-side technology. Obviously the choice of server-side tech has an influence on the structure of the URLs, but beyond that we don't need to worry ourselves much about what's going to transpire on the server. We simply make HTTP requests, sometimes with appropriate parameter data, and as long as the server returns the expected responses, we could care less if the server is being powered by Java, Ruby, PHP, or even old-fashioned CGI.

In this particular case, we expect that the server-side resource will return the HTML markup representing the boot style options—supposedly from the inventory database. Our faux backend code returns the following as the response:

```
<option value="">&mdash; choose a style &mdash;</option>
<option value="7177382">Caterpillar Tradesman Work Boot</option>
<option value="7269643">Caterpillar Logger Boot</option>
<option value="7332058">Chippewa 9" Briar Waterproof Bison Boot</option>
<option value="7141832">Chippewa 17" Engineer Boot</option>
<option value="7141833">Chippewa 17" Snakeproof Boot</option>
<option value="7173656">Chippewa 11" Engineer Boot</option>
<option value="7141922">Chippewa Harness Boot</option>
<option value="7141730">Danner Foreman Pro Work Boot</option>
<option value="7257914">Danner Grouse GTX Boot</option>
```

This response then gets injected into the `<select>` element, resulting in a fully functional control.

Our next act is to instrument the dropdown so that it can react to changes, carrying out the duties that we listed earlier. The code for that is only slightly more complicated:

```
$('#bootChooserControl').change(function(event){
  $('#productDetailPane').load(
    '/jqia2/action/fetchProductDetails',
    {style: $(event.target).val()},
    function() { $('[value=""]',event.target).remove(); }
  );
});
```

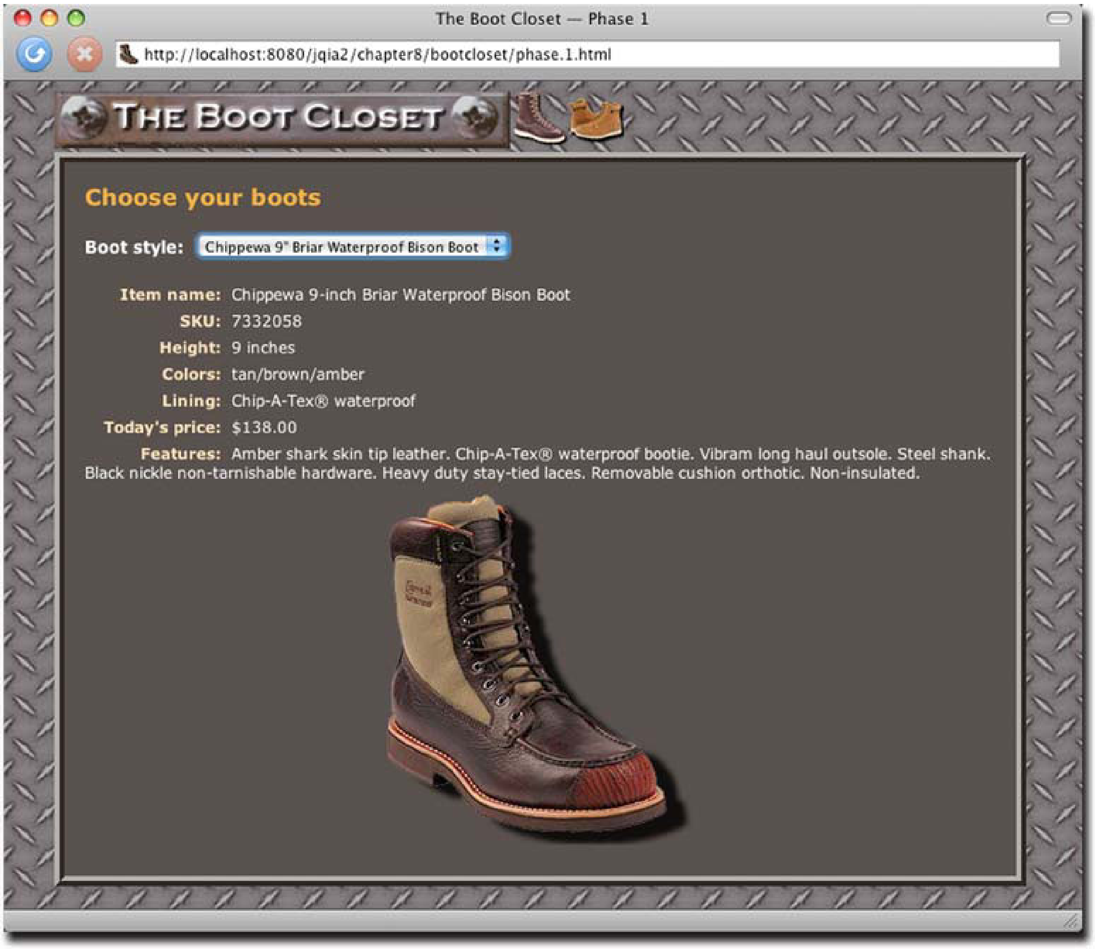
Establishes change handler ❶

❷ **Fetches and displays product detail**

In this code, we select the boot style dropdown and bind a `change` handler to it ❶. In the callback for the change handler, which will be invoked whenever a customer changes the selection, we obtain the current value of the selection by applying the `val()` method to the event target, which is the `<select>` element that triggered the event. We then once again employ the `load()` method ❷ to the `productDetailPane` element to initiate an Ajax callback to a server-side resource, in this case `fetchProductDetails`, passing the style value as a parameter named `style`.

After the customer chooses an available boot style, the page will appear as shown in figure 8.3.

The most notable operation performed in the ready handler is the use of the `load()` method to quickly and easily fetch snippets of HTML from the server and place them within the DOM as the children of existing elements. This method is extremely handy and well suited to web applications that are powered by servers capable of server-side templating with technologies such as JSP and PHP.


The Boot Closet — Phase 1
http://localhost:8080/jqia2/chapter8/bootcloset/phase.1.html
THE BOOT CLOSET

Choose your boots

Boot style: Chippewa 9" Briar Waterproof Bison Boot

Item name: Chippewa 9-inch Briar Waterproof Bison Boot
SKU: 7332058
Height: 9 inches
Colors: tan/brown/amber
Lining: Chip-A-Tex® waterproof
Today's price: $138.00
Features: Amber shark skin tip leather. Chip-A-Tex® waterproof bootie. Vibram long haul outsole. Steel shank. Black nickle non-tarnishable hardware. Heavy duty stay-tied laces. Removable cushion orthotic. Non-insulated.

Figure 8.3 The server-side resource returns a preformatted fragment of HTML to display the detailed boot information.

Listing 8.4 shows the complete code for our Boot Closet page, which can be found at http://localhost[:8080]/jqia2/chapter8/bootcloset/phase.1.html. We'll be revisiting this page to add further capabilities to it as we progress through this chapter.

Listing 8.4 The first phase of the Boot Closet commerce page

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Boot Closet &mdash; Phase 1</title>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css">
    <link rel="stylesheet" type="text/css" href="bootcloset.css">
    <link rel="icon" type="image/gif" href="images/favicon.gif">
    <script type="text/javascript"
            src="../../scripts/jquery-1.4.js"></script>
    <script type="text/javascript"
            src="../../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function() {

        $('#bootChooserControl')
          .load('/jqia2/action/fetchBootStyleOptions');

        $('#bootChooserControl').change(function(event){
          $('#productDetailPane').load(
            '/jqia2/action/fetchProductDetails',
            {style: $(event.target).val()},
            function() { $('[value=""]',event.target).remove(); }
          );
        });

      });
    </script>
  </head>

  <body>

    <div id="banner">
      <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
    </div>

    <div id="pageContent">

      <h1>Choose your boots</h1>

      <div>

        <div id="selectionsPane">
          <label for="bootChooserControl">Boot style:</label>&nbsp;
          <select id="bootChooserControl" name="bootStyle"></select>
        </div>

        <div id="productDetailPane"></div>

      </div>

    </div>

  </body>

</html>
```

The `load()` method is tremendously useful when we want to grab a fragment of HTML to stuff into the content of an element (or set of elements). But there may be times when we either want more control over how the Ajax request gets made, or we need to do something more esoteric with the returned data in the response body.

Let's continue our investigation of what jQuery has to offer for these more complex situations.

8.3 Making GET and POST requests

The `load()` method makes either a `GET` or a `POST` request, depending on how the request parameter data (if any) is provided, but sometimes we want to have a bit more control over which HTTP method gets used. Why should *we* care? Because, just maybe, our *servers* care.

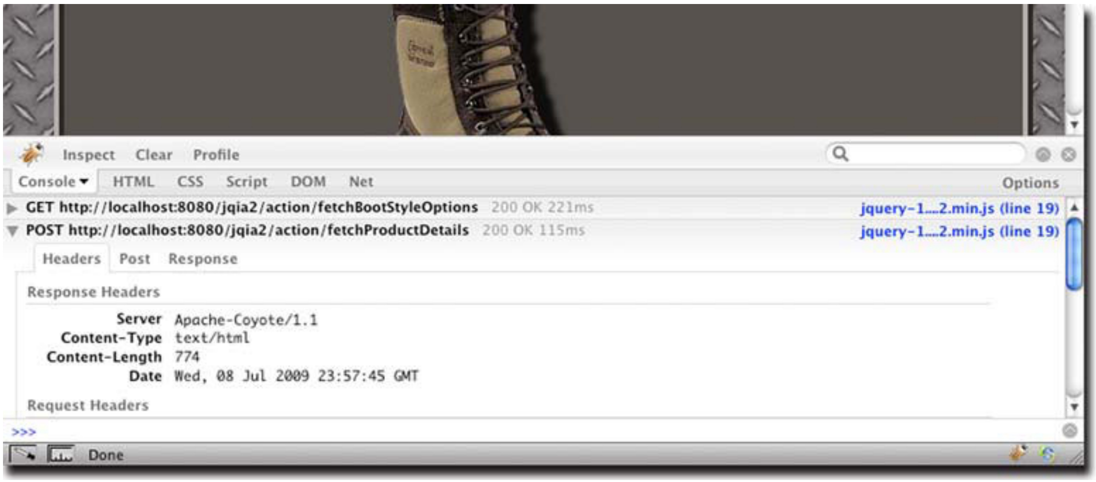
Web authors have traditionally played fast and loose with the `GET` and `POST` methods, using one or the other without heeding how the HTTP protocol intends for these methods to be used. The intentions for each method are as follows:

- `GET` *requests*—Intended to be *idempotent*; the same `GET` operation, made again and again and again, should return exactly the same results (assuming no other force is at work changing the server state).
- `POST` *requests*—Can be *non-idempotent*; the data they send to the server can be used to change the model state of the application; for example, adding or updating records in a database or removing information from the server.

A `GET` request should, therefore, be used whenever the purpose of the request is to merely get data; as its name implies. It may be required to *send* some parameter data to the server for the `GET`; for example, to identify a style number to retrieve color information. But when data is being sent to the server in order to effect a change, `POST` should be used.

> **WARNING** *This is more than theoretical.* Browsers make decisions about caching based upon the HTTP method used, and `GET` requests are highly subject to caching. Using the proper HTTP method ensures that you don't get crossways with the browser's or server's expectations regarding the intentions of the requests.
>
> This is just a glimpse into the realm of RESTful principles, where other HTTP methods, such as `PUT` and `DELETE`, also come into play. But for our purposes, we'll limit our discussion to the `GET` and `POST` methods.

With that in mind, if we look back to our phase one implementation of The Boot Closet (in listing 8.4), we discover that *we're doing it wrong*! Because jQuery initiates a `POST` request when we supply an object hash for parameter data, we're making a `POST` when we really should be making a `GET`. If we glance at a Firebug log (as shown in figure 8.4) when we display our page in Firefox, we can see that our second request, submitted when we make a selection from the style dropdown, is indeed a `POST`.



```
▶ GET http://localhost:8080/jqia2/action/fetchBootStyleOptions  200 OK 221ms    jquery-1....2.min.js (line 19)
▼ POST http://localhost:8080/jqia2/action/fetchProductDetails  200 OK 115ms     jquery-1....2.min.js (line 19)

Response Headers
         Server  Apache-Coyote/1.1
   Content-Type  text/html
 Content-Length  774
           Date  Wed, 08 Jul 2009 23:57:45 GMT

Request Headers
>>>
```

Figure 8.4 An inspection of the Firebug console shows that we're making a `POST` request when we should be making a `GET`.

Does it really matter? That's up to you, but if we want to use HTTP in the manner in which it was intended, our request to fetch the boot detail should be a GET rather than a POST.

We could simply make the parameter that specifies the request information a string rather than an object hash (and we'll revisit that a little later), but for now, let's take advantage of another way that jQuery lets us initiate Ajax requests.

Get Firebug

Trying to develop a DOM-scripted application without the aid of a debugging tool is like trying to play concert piano while wearing welding gloves. Why would you do that to yourself?

One important tool to have in your tool chest is *Firebug*, a plugin for the Firefox browser. As shown in figure 8.4, Firebug not only lets us inspect the JavaScript console, it lets us inspect the live DOM, the CSS, the script, and many other aspects of our page as we work through its development.

One feature most relevant for our current purposes is its ability to log Ajax requests along with both the request and response information.

For browsers other than Firefox, there's *Firebug Lite*, which simply loads as a JavaScript library while we're debugging.

You can get Firebug at http://getfirebug.com and Firebug Lite at http://getfirebug.com/lite.html.

Google's Chrome browser comes built in with Firebug-like debug capabilities, which you can display by opening its Developer Tools (look around the menus for this entry—it keeps moving).

8.3.1 Getting data with GET

jQuery gives us a few means to make GET requests, which unlike load(), aren't implemented as jQuery methods on a wrapped set. Rather, a handful of utility functions are provided to make various types of GET requests. As we pointed out in chapter 1, jQuery utility functions are top-level functions that are namespaced with the jQuery global name (and its $ alias).

When we want to fetch some data from the server and decide what to do with it ourselves (rather than letting the load() method set it as the content of an HTML element), we can use the $.get() utility function. Its syntax is as follows:

Function syntax: $.get

$.get(url,parameters,callback,type)

Initiates a GET request to the server using the specified URL with any passed parameters as the query string.

Parameters

url	(String) The URL of the server-side resource to contact via the GET method.
parameters	(String|Object|Array) Specifies any data that's to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose name and value properties specify the name/value pairs.
callback	(Function) An optional function invoked when the request completes successfully. The response body is passed as the first parameter to this callback, interpreted according to the setting of the type parameter, and the text status is passed as the second parameter. A third parameter contains a reference to the XHR instance.
type	(String) Optionally specifies how the response body is to be interpreted; one of html, text, xml, json, script, or jsonp. See the description of $.ajax() later in this chapter for more details.

Returns

The XHR instance.

The $.get() utility function allows us to initiate GET requests with a lot of versatility. We can specify request parameters (if appropriate) in numerous handy formats, provide a callback to be invoked upon a successful response, and even direct how the response is to be interpreted and passed to the callback. If even that's not enough versatility, we'll be seeing a more general function, $.ajax(), later on.

We'll be examining the type parameter in greater detail when we look at the $.ajax() utility function, but for now we'll let it default to html or xml depending upon the content type of the response.

Applying $.get() to our Boot Closet page, we'll replace the use of the load() method with the $.get() function, as shown in listing 8.5.

Listing 8.5 Changing the Boot Closet to use a `GET` when fetching style details

```
$('#bootChooserControl').change(function(event){
  $.get(
    '/jqia2/action/fetchProductDetails',
    {style: $(event.target).val()},
    function(response) {
      $('#productDetailPane').html(response);
      $('[value=""]',event.target).remove();
    }
  );
});
```

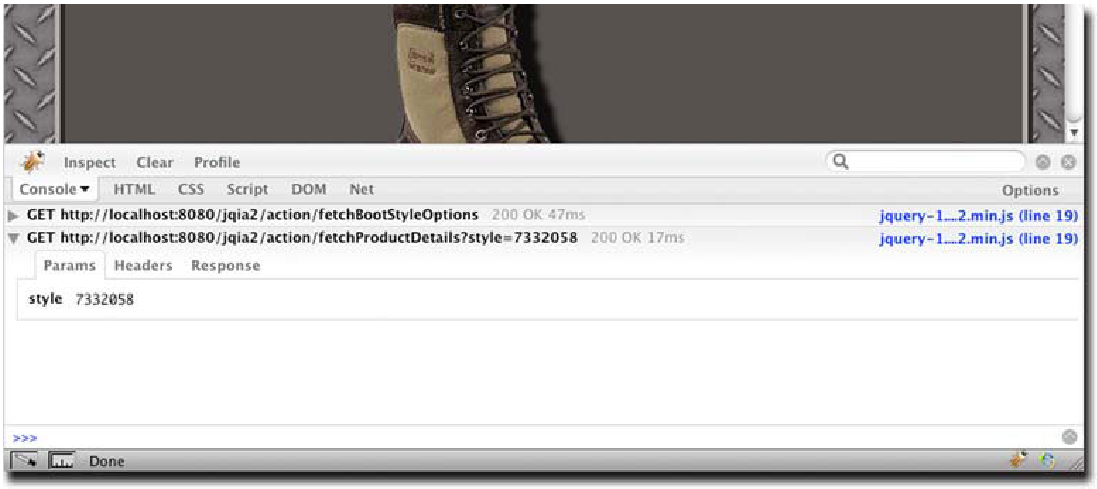
❶ Initiates GET request

❷ Injects response HTML

The changes for this second phase of our page are subtle, but significant. We call `$.get()` ❶ in place of `load()`, passing the same URL and the same request parameters. Because `$.get()` does no automatic injection of the response anywhere within the DOM, we need to do that ourselves, which is easily accomplished via a call to the `html()` method ❷.

The code for this version of our page can found at http://localhost[:8080]/jqia2/chapter8/bootcloset/phase.2.html, and when we display it and select a style dropdown, we can see that a `GET` request has been made, as shown in figure 8.5.
In this example, we returned formatted HTML from the server and inserted it into the DOM, but as we can see from the `type` parameter to `$.get()`, there are many other possibilities than HTML. In fact, the term *Ajax* began its life as the acronym AJAX, where the X stood for XML.

When we pass the `type` as `xml` (remember, we'll be talking about `type` in more detail in a little bit), and return XML from the server, the data passed to the callback is a parsed XML DOM. And although XML is great when we need its flexibility and our data is highly hierarchical in nature, XML can be painful to traverse and to digest its data. Let's see another jQuery utility function that's quite useful when our data needs are more straightforward.



```
▶ GET http://localhost:8080/jqia2/action/fetchBootStyleOptions 200 OK 47ms            jquery-1....2.min.js (line 19)
▼ GET http://localhost:8080/jqia2/action/fetchProductDetails?style=7332058 200 OK 17ms   jquery-1....2.min.js (line 19)
    style  7332058
>>>
```

Figure 8.5 Now we can see that the second request is a `GET` rather than a `POST`, as befitting the operation.

8.3.2 Getting JSON data

As stated in the previous section, when an XML document is returned from the server, the XML document is automatically parsed, and the resulting DOM is made available to the callback function. When XML is overkill or otherwise unsuitable as a data-transfer mechanism, JSON is often used in its place. One reason for this choice is that JSON is astoundingly easy to digest in client-side scripts. jQuery makes it even easier.

For times when we know that the response will be JSON, the `$.getJSON()` utility function automatically parses the returned JSON string and makes the resulting JavaScript data item available to its callback. The syntax of this utility function is as follows:

Function syntax: $.getJSON

`$.getJSON(url,parameters,callback)`

Initiates a `GET` request to the server using the specified URL, with any passed parameters as the query string. The response is interpreted as a JSON string, and the resulting data is passed to the callback function.

Parameters

`url` (String) The URL of the server-side resource contacted via the `GET` method.

`parameters` (String|Object|Array) Specifies any data that's to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.

`callback` (Function) A function invoked when the request completes. The data value resulting from digesting the response body as a JSON representation is passed as the first parameter to this callback, and the status text is passed as the second parameter. A third parameter provides a reference to the XHR instance.

Returns

The XHR instance.

This function, which is simply a convenience function for `$.get()` with a `type` of `json`, is great for those times when we want to get data from the server without the overhead of dealing with XML.

Between `$.get()` and `$.getJSON()`, jQuery gives us some powerful tools when it comes to making `GET` requests, but man does not live by `GET`s alone!

8.3.3 Making POST requests

"Sometimes you feel like a nut, sometimes you don't." What's true of choosing between an Almond Joy or a Mounds candy bar is also true of making requests to the server. Sometimes we want to make a `GET`, but at other times we want (or even need) to make a `POST` request.

There are any number of reasons why we might choose a `POST` over a `GET`. First, the intention of the HTTP protocol is that `POST` will be used for any non-idempotent requests. Therefore, if our request has the potential to cause a change in the server-side state, resulting in varying responses, it should be a `POST`. Moreover, accepted practices and conventions aside, a POST operation must sometimes be used when the data to be passed to the server exceeds the small amount that can be passed by URL in a query string—a limit that's a browser-dependent value. And sometimes, the server-side resource we contact may only support POST operations, or it might even perform different functions depending upon whether our request uses the GET or POST method.

For those occasions when a POST is desired or mandated, jQuery offers the `$.post()` utility function, which operates in a similar fashion to `$.get()`, except for employing the POST HTTP method. Its syntax is as follows:

Function syntax: $.post

`$.post(url,parameters,callback,type)`

Initiates a POST request to the server using the specified URL, with any parameters passed within the body of the request.

Parameters

`url` (String) The URL of the server-side resource to contact via the POST method.

`parameters` (String|Object|Array) Specifies any data that's to be passed as request parameters. This parameter can be a string that will be used as the query string, an object whose properties are serialized into properly encoded parameters to be passed to the request, or an array of objects whose `name` and `value` properties specify the name/value pairs.

`callback` (Function) A function invoked when the request completes. The response body is passed as the single parameter to this callback, and the status text as the second. A third parameter provides a reference to the XHR instance.

`type` (String) Optionally specifies how the response body is to be interpreted; one of `html`, `text`, `xml`, `json`, `script`, or `jsonp`. See the description of `$.ajax()` for more details.

Returns

The XHR instance.

Except for making a POST request, using `$.post()` is identical to using `$.get()`. jQuery takes care of the details of passing the request data in the request body (as opposed to the query string), and sets the HTTP method appropriately.

Now, getting back to our Boot Closet project, we've made a really good start, but there's more to buying a pair of boots than just selecting a style; customers are sure to want to pick which color they want, and certainly they'll need to specify their size. We'll use these additional requirements to show how to solve one of the most-asked questions in online Ajax forums, that of …

8.3.4 *Implementing cascading dropdowns*

The implementation of cascading dropdowns—where subsequent dropdown options depend upon the selections of previous dropdowns—has become sort of a poster child for Ajax on the web. And although you'll find thousands, perhaps tens of thousands, of solutions, we're going to implement a solution on our Boot Closet page that demonstrates how ridiculously simple jQuery makes it.

We've already seen how easy it was to load a dropdown dynamically with server-powered option data. We'll see that tying multiple dropdowns together in a cascading relationship is only slightly more work.

Let's dig in by listing the changes we need to make in the next phase of our page:

- Add dropdowns for color and size.
- When a style is selected, add options to the color dropdown that show the colors available for that style.
- When a color is selected, add options to the size dropdown that show the sizes available for the selected combination of style and color.
- Make sure things remain consistent. This includes removing the "— please make a selection —" options from newly created dropdowns once they've been used once, and making sure that the three dropdowns never show an invalid combination.

We're also going to revert to using `load()` again, this time coercing it to initiate a `GET` rather than a `POST`. It's not that we have anything against `$.get()`, but `load()` just seems more natural when we're using Ajax to load HTML fragments.

To start off, let's examine the new HTML markup that defines the additional dropdowns. A new container for the select elements is defined to contain three labeled elements:

```
<div id="selectionsPane">
  <label for="bootChooserControl">Boot style:</label>
  <select id="bootChooserControl" name="style"></select>
  <label for="colorChooserControl">Color:</label>
  <select id="colorChooserControl" name="color" disabled="disabled"></select>
  <label for="sizeChooserControl">Size:</label>
  <select id="sizeChooserControl" name="size" disabled="disabled"></select>
</div>
```

The previous style selection element remains, but it has been joined by two more: one for color, and one for size, each of which is initially empty and disabled.

That was easy, and it takes care of the additions to the structure. Now let's add the additional behaviors.

The style selection dropdown must now perform double duty. Not only must it continue to fetch and display the boot details when a selection is made, its change handler must now also populate and enable the color selection dropdown with the colors available for whatever style was chosen.

Let's refactor the fetching of the details first. We want to use `load()`, but we also want to force a `GET`, as opposed to the `POST` that we were initiating earlier. In order to have `load()` induce a `GET`, we need to pass a string, rather than an object hash, to specify the request parameter data. Luckily, with jQuery's help, we won't have to build that string ourselves. The first part of the change handler for the style dropdown gets refactored like this:

```
$('#bootChooserControl').change(function(event){
  $('#productDetailPane').load(
    '/jqia2/action/fetchProductDetails',
    $(this).serialize()
  );
  // more to follow
});
```

Provides query string

By using the `serialize()` method, we create a string representation of the value of the style dropdown, thereby coercing the `load()` method to initiate a `GET`, just as we wanted.

The second duty that the change handler needs to perform is to load the color-choice dropdown with appropriate values for the chosen style, and then enable it. Let's take a look at the rest of the code to be added to the handler:

```
$('#colorChooserControl').load(
  '/jqia2/action/fetchColorOptions',
  $(this).serialize(),
  function(){
    $(this).attr('disabled',false);
    $('#sizeChooserControl')
      .attr('disabled',true)
      .html("");
  }
);
```

❶ **Fetches and loads color options**

❷ **Enables color control**

❸ **Disables and empties size control**

This code should look familiar. It's just another use of `load()`, this time referencing an action named `fetchColorOptions`, which is designed to return a set of formatted `<option>` elements representing the colors available for the chosen style (which we again passed as request data) ❶. This time, we've also specified a callback to be executed when the `GET` request successfully returns a response.

In this callback, we perform two important tasks. First, we enable the color-chooser control ❷. The call to `load()` injected the `<option>` elements, but once populated it would still be disabled if we did not enable it.

Second, the callback disables and empties the size-chooser control ❸. But why? (Pause a moment and think about it.)

Even though the size control will already be disabled and empty the first time the style chooser's value is changed, what about later on? What if, after the customer chooses a style and a color (which we'll soon see results in the population of the size control), he or she changes the selected style? Because the sizes displayed depend upon the combination of style and color, the sizes previously displayed are no longer applicable and don't reflect a consistent view of what's chosen. Therefore, whenever the style changes, we need to blow the size options away and reset the size control to initial conditions.

Before we sit back and enjoy a lovely beverage, we've got more work to do. We still have to instrument the color-chooser dropdown to use the selected style and color values to fetch and load the size-chooser dropdown. The code to do this follows a familiar pattern:

```
$('#colorChooserControl').change(function(event){
  $('#sizeChooserControl').load(
    '/jqia2/action/fetchSizeOptions',
    $('#bootChooserControl,#colorChooserControl').serialize(),
    function(){
      $(this).attr('disabled',false);
    }
  );
});
```

Upon a change event, the size information is obtained via the `fetchSizeOptions` action, passing both the boot style and color selections, and the size control is enabled.

There's one more thing that we need to do. When each dropdown is initially populated, it's seeded with an `<option>` element with a blank value and display text along the lines of "— choose a something —". You may recall that in the previous phases of this page, we added code to remove that option from the style dropdown upon selection.

Well, we could add such code to the change handlers for the style and color dropdowns, and add instrumentation for the size dropdown (which currently has none) to add that. But let's be a bit more suave about it.

One capability of the event model that often gets ignored by many a web developer is *event bubbling*. Page authors frequently focus only on the targets of events, and forget that events will bubble up the DOM tree, where handlers can deal with those events in more general ways than at the target level.

If we recognize that removing the option with a blank value from any of the three dropdowns can be handled in the exact same fashion *regardless* of which dropdown is the target of the event, we can avoid repeating the same code in three places by establishing a *single* handler, higher in the DOM, that will recognize and handle the change events.

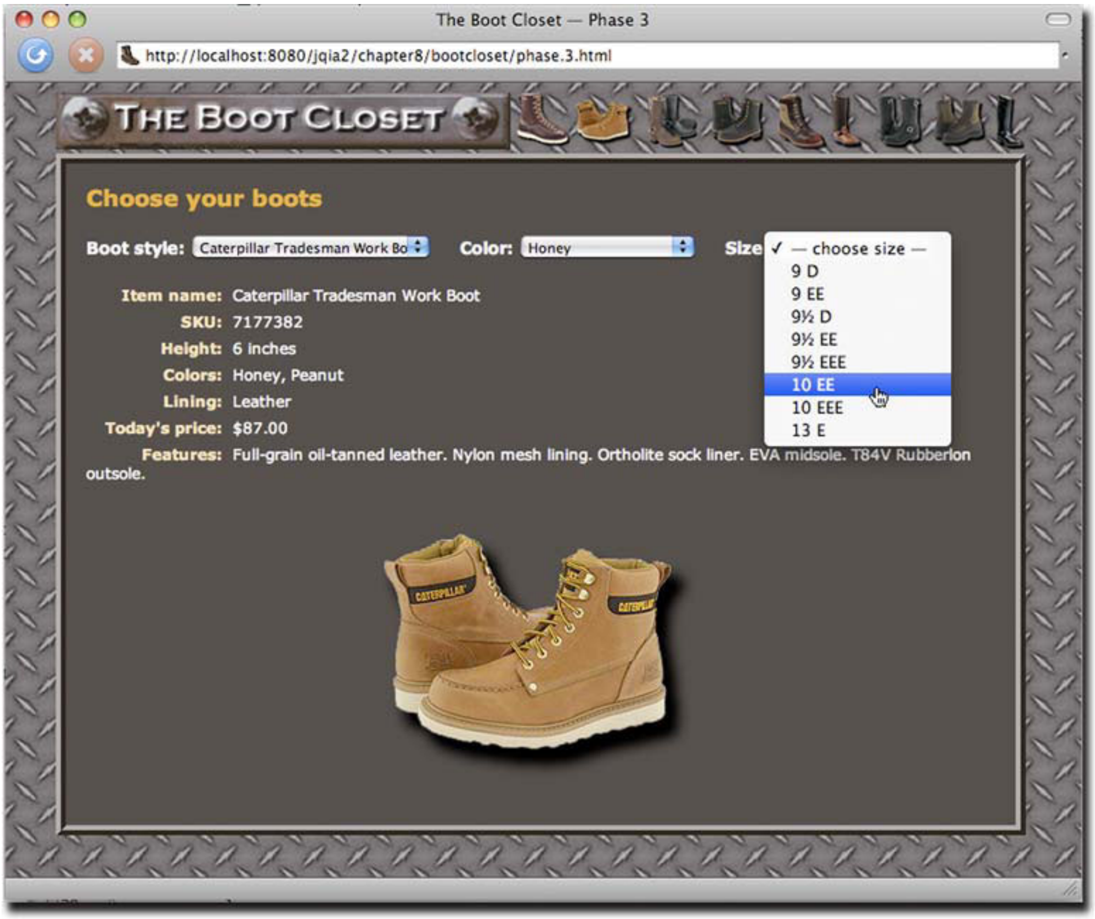
Recalling the structure of the document, the three dropdowns are contained within a `<div>` element with an `id` of `selectionsPane`. We can handle the removal of the temporary option for all three dropdowns with the following, single listener:

```
$('#selectionsPane').change(function(event){
  $('[value=""]',event.target).remove();
});
```

This listener will be triggered whenever a change event happens on any of the enclosed dropdowns, and will remove the option with the blank value within the context of the target of the event (which will be the changed dropdown). How slick is that?

Using event bubbling to avoid repeating the same code in lower-level handlers can really elevate your script to the big leagues!

With that, we've completed phase three of The Boot Closet, adding cascading dropdowns into the mix as shown in figure 8.6. We can use the same techniques in any pages where dropdown values depend upon previous selections. The page for this phase can be found at URL http://localhost[:8080]/jqia2/chapter8/bootcloset/phase.3.html.



The Boot Closet — Phase 3

http://localhost:8080/jqia2/chapter8/bootcloset/phase.3.html

THE BOOT CLOSET

Choose your boots

Boot style: Caterpillar Tradesman Work Bo Color: Honey Size ✓ — choose size —

Item name: Caterpillar Tradesman Work Boot
SKU: 7177382
Height: 6 inches
Colors: Honey, Peanut
Lining: Leather
Today's price: $87.00
Features: Full-grain oil-tanned leather. Nylon mesh lining. Ortholite sock liner. EVA midsole. T84V Rubberlon outsole.

Figure 8.6 **The third phase of The Boot Closet shows how easy it is to implement cascading dropdowns.**

The full code of the page is now as shown in listing 8.6.

Listing 8.6 The Boot Closet, now with cascading dropdowns!

```
<!DOCTYPE html>
<html>
  <head>
    <title>The Boot Closet &mdash; Phase 3</title>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css">
    <link rel="stylesheet" type="text/css" href="bootcloset.css">
    <link rel="icon" type="image/gif" href="images/favicon.gif">
    <script type="text/javascript"
            src="../../scripts/jquery-1.4.js"></script>

    <script type="text/javascript"
            src="../../scripts/jqia2.support.js"></script>
    <script type="text/javascript">
      $(function() {

        $('#bootChooserControl')
          .load('/jqia2/action/fetchBootStyleOptions');
        $('#bootChooserControl').change(function(event){
          $('#productDetailPane').load(
            '/jqia2/action/fetchProductDetails',
            $(this).serialize()
          );
          $('#colorChooserControl').load(
            '/jqia2/action/fetchColorOptions',
            $(this).serialize(),
            function(){
              $(this).attr('disabled',false);
              $('#sizeChooserControl')
                .attr('disabled',true)
                .html("");
            }
          );
        });

        $('#colorChooserControl').change(function(event){
          $('#sizeChooserControl').load(
            '/jqia2/action/fetchSizeOptions',
            $('#bootChooserControl,#colorChooserControl').serialize(),
            function(){
              $(this).attr('disabled',false);
            }
          );
        });

        $('#selectionsPane').change(function(event){
          $('[value=""]',event.target).remove();
        });

      });
    </script>
  </head>

  <body>

    <div id="banner">
      <img src="images/banner.boot.closet.png" alt="The Boot Closet"/>
    </div>

    <div id="pageContent">

      <h1>Choose your boots</h1>

      <div>

        <div id="selectionsPane">
          <label for="bootChooserControl">Boot style:</label>
          <select id="bootChooserControl" name="style"></select>
          <label for="colorChooserControl">Color:</label>
          <select id="colorChooserControl" name="color"
                  disabled="disabled"></select>
          <label for="sizeChooserControl">Size:</label>
          <select id="sizeChooserControl" name="size"
                  disabled="disabled"></select>
        </div>

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

As we've seen, with the `load()` method and the various `GET` and `POST` jQuery Ajax functions at our disposal, we can exert some measure of control over how our request is initiated and how we're notified of its completion. But for those times when we need *full* control over an Ajax request, jQuery has a means for us to get as picky as we want.

8.4 Taking full control of an Ajax request

The functions and methods we've seen so far are convenient for many cases, but there may be times when we want to take the control of all the nitty-gritty details into our own hands.

In this section, we'll explore how jQuery lets us exert such dominion.

8.4.1 Making Ajax requests with all the trimmings

For those times when we want or need to exert fine-grained control over how we make Ajax requests, jQuery provides a general utility function for making Ajax requests, named `$.ajax()`. Under the covers, all other jQuery features that make Ajax requests eventually use this function to initiate the request. Its syntax is as follows:

Function syntax: $.ajax

`$.ajax(options)`

Initiates an Ajax request using the passed options to control how the request is made and callbacks notified.

Parameters

`options` (Object) An object whose properties define the parameters for this operation. See table 8.2 for details.

Returns

The XHR instance.

Looks simple, doesn't it? But don't be deceived. The `options` parameter can specify a large range of values that can be used to tune the operation of this function. These options (in general order of their importance and the likelihood of their use) are defined in table 8.2.

Table 8.2 Options for the `$.ajax()` utility function

Name	Description
`url`	(String) The URL for the request.
`type`	(String) The HTTP method to use. Usually either `POST` or `GET`. If omitted, the default is `GET`.

Table 8.2 Options for the `$.ajax()` utility function *(continued)*

Name	Description
`data`	(String|Object|Array) Defines the values that will serve as the query parameters to be passed to the request. If the request is a `GET`, this data is passed as the query string. If a `POST`, the data is passed as the request body. In either case, the encoding of the values is handled by the `$.ajax()` utility function. This parameter can be a string that will be used as the query string or response body, an object whose properties are serialized, or an array of objects whose `name` and `value` properties specify the name/value pairs.
`dataType`	(String) A keyword that identifies the type of data that's expected to be returned by the response. This value determines what, if any, post-processing occurs upon the data before being passed to callback functions. The valid values are as follows: • `xml`—The response text is parsed as an XML document and the resulting XML DOM is passed to the callbacks. • `html`—The response text is passed unprocessed to the callback functions. Any `<script>` blocks within the returned HTML fragment are evaluated. • `json`—The response text is evaluated as a JSON string, and the resulting object is passed to the callbacks. • `jsonp`—Similar to `json` except that remote scripting is allowed, assuming the remote server supports it. • `script`—The response text is passed to the callbacks. Prior to any callbacks being invoked, the response is processed as a JavaScript statement or statements. • `text`—The response text is assumed to be plain text. The server resource is responsible for setting the appropriate content-type response header. If this property is omitted, the response text is passed to the callbacks without any processing or evaluation.
`cache`	(Boolean) If `false`, ensures that the response won't be cached by the browser. Defaults to `true` except when `dataType` is specified as either `script` or `jsonp`.
`context`	(Element) Specifies an element that is to be set as the context of all callbacks related to this request.
`timeout`	(Number) Sets a timeout for the Ajax request in milliseconds. If the request doesn't complete before the timeout expires, the request is aborted and the error callback (if defined) is called.
`global`	(Boolean) If `false`, disables the triggering of global Ajax events. These are jQuery-specific custom events that trigger at various points or conditions during the processing of an Ajax request. We'll be discussing them in detail in the upcoming section. If omitted, the default (`true`) is to enable the triggering of global events.
`contentType`	(String) The content type to be specified on the request. If omitted, the default is `application/x-www-form-urlencoded`, the same type used as the default for form submissions.
`success`	(Function) A function invoked if the response to the request indicates a success status code. The response body is returned as the first parameter to this function and evaluated according to the specification of the `dataType` property. The second parameter is a string containing a status value—in this case, always `success`. A third parameter provides a reference to the XHR instance.

Table 8.2 Options for the `$.ajax()` utility function *(continued)*

Name	Description
`error`	(Function) A function invoked if the response to the request returns an error status code. Three arguments are passed to this function: the XHR instance, a status message string (in this case, one of: `error`, `timeout`, `notmodified`, or `parseerror`), and an optional exception object, sometimes returned from the XHR instance, if any.
`complete`	(Function) A function called upon completion of the request. Two arguments are passed: the XHR instance and a status message string of either `success` or `error`. If either a success or error callback is also specified, this function is invoked after that callback is called.
`beforeSend`	(Function) A function invoked prior to initiating the request. This function is passed the XHR instance and can be used to set custom headers or to perform other pre-request operations. Returning `false` from this handler will cancel the request.
`async`	(Boolean) If specified as `false`, the request is submitted as a synchronous request. By default, the request is asynchronous.
`processData`	(Boolean) If set to `false`, prevents the data passed from being processed into URL-encoded format. By default, the data is URL-encoded into a format suitable for use with requests of type `application/x-www-form-urlencoded`.
`dataFilter`	(Function) A callback invoked to filter the response data. This function is passed the raw response data and the `dataType` value, and is expected to return the “sanitized” data.
`ifModified`	(Function) If `true`, allows a request to succeed only if the response content has not changed since the last request, according to the `Last-Modified` header. If omitted, no header check is performed. Defaults to `false`.
`jsonp`	(String) Specifies a query parameter name to override the default `jsonp` callback parameter name of `callback`.
`username`	(String) The username to be used in the event of an HTTP authentication request.
`password`	(String) The password to be used in the case of an HTTP authentication request.
`scriptCharset`	(String) The character set to be used for `script` and `jsonp` requests when the remote and local content are of different character sets.
`xhr`	(Function) A callback used to provide a custom implementation of the XHR instance.
`traditional`	(Boolean) If `true`, the traditional style of parameter serialization is used. See the description of `$.param()` in chapter 6 for details on parameter serialization.

That’s a lot of options to keep track of, but it’s unlikely that more than a few of them will be used for any one request. Even so, wouldn’t it be convenient if we could set default values for these options for pages where we’re planning to make a large number of requests?

8.4.2 Setting request defaults

Obviously the last question in the previous section was a setup. As you might have suspected, jQuery provides a way for us to define a default set of Ajax properties that will be used when we don't override their values. This can make pages that initiate lots of similar Ajax calls much simpler.

The function to set up the list of Ajax defaults is `$.ajaxSetup()`, and its syntax is as follows:

Method syntax: $.ajaxSetup

`$.ajaxSetup(options)`

Establishes the passed set of option properties as the defaults for subsequent calls to `$.ajax()`.

Parameters

`options`	(Object) An object instance whose properties define the set of default Ajax options. These are the same properties described for the `$.ajax()` function in table 8.2. This function should not be used to set callback handlers for success, error, and completion. (We'll see how to set these up using an alternative means in an upcoming section.)

Returns

Undefined.

At any point in script processing, usually at page load (but it can be at any point of the page authors' choosing), this function can be used to set up defaults to be used for all subsequent calls to `$.ajax()`.

> **NOTE** Defaults set with this function aren't applied to the `load()` method. Also, for utility functions such as `$.get()` and `$.post()`=, the HTTP method can't be overridden by these defaults. For example, setting a default `type` of `GET` won't cause `$.post()` to use the `GET` HTTP method.

Let's say that we're setting up a page where, for the majority of Ajax requests (made with the utility functions rather than the `load()` method), we want to set up some defaults so that we don't need to specify them on every call. We can, as the first statement in the header `<script>` element, write this:

```
$.ajaxSetup({
  type: 'POST',
  timeout: 5000,
  dataType: 'html'
});
```

This would ensure that every subsequent Ajax call (except as noted previously) would use these defaults, unless explicitly overridden in the properties passed to the Ajax utility function being used.

Now, what about those *global events* we mentioned that were controlled by the `global` option?

8.4.3 Handling Ajax events

Throughout the execution of jQuery Ajax requests, jQuery triggers a series of custom events for which we can establish handlers in order to be informed of the progress of a request, or to take action at various points along the way. jQuery classifies these events as local events and global events.

Local events are handled by the callback functions that we can directly specify using the `beforeSend`, `success`, `error`, and `complete` options of the `$.ajax()` function, or indirectly by providing callbacks to the convenience methods (which, in turn, use the `$.ajax()` function to make the actual requests). We've been handling local events all along, without even knowing it, whenever we've registered a callback function to any jQuery Ajax function.

Global events are those that are triggered like other custom events within jQuery, and for which we can establish event handlers via the `bind()` method (just like any other event). The global events, many of which mirror local events, are: ajaxStart, ajaxSend, ajaxSuccess, ajaxError, ajaxStop, and ajaxComplete.

When triggered, the global events are broadcast to every element in the DOM, so we can establish these handlers on any DOM element, or elements, of our choosing. When executed, the handlers' function context is set to the DOM element upon which the handler was established.

Because we don't need to consider a bubbling hierarchy, we can establish a handler on any element for which it would be convenient to have ready access via `this`. If we don't care about a specific element, we could just establish the handler on the <body> for lack of a better location. But if we have something specific to do to an element, such as hide and show animated graphics while an Ajax request is processing, we could establish the handle on that element and have easy access to it via the function context.

In addition to the function context, more information is available via parameters passed to the handlers; most often these are the `jQuery.Event` instance, the XHR instance, and the options passed to `$.ajax()`.

Exceptions to this parameter list will be noted in Table 8.3, which shows the jQuery Ajax events in the order in which they are delivered.

Table 8.3 jQuery Ajax event types

Event name	Type	Description
ajaxStart	Global	Triggered when an Ajax request is started, as long as no other requests are active. For concurrent requests, this event is triggered only for the first of the requests. No parameters are passed.
beforeSend	Local	Invoked prior to initiating the request in order to allow modification of the XHR instance prior to sending the request to the server, or to cancel the request by returning `false`.

Table 8.3 jQuery Ajax event types *(continued)*

Event name	Type	Description
ajaxSend	Global	Triggered prior to initiating the request in order to allow modification of the XHR instance prior to sending the request to the server.
success	Local	Invoked when a request returns a successful response.
ajaxSuccess	Global	Triggered when a request returns a successful response.
error	Local	Invoked when a request returns an error response.
ajaxError	Global	Triggered when a request returns an error response. An optional fourth parameter referencing the thrown error, if any, is passed.
complete	Local	Invoked when a request completes, regardless of status. This callback is invoked even for synchronous requests.
ajaxComplete	Global	Triggered when a request completes, regardless of status. This callback is invoked even for synchronous requests.
ajaxStop	Global	Triggered when an Ajax request completes *and* there are no other concurrent requests active. No parameters are passed.

Once again (to make sure things are clear), local events represent callbacks passed to `$.ajax()` (and its cohorts), whereas global events are custom events that are triggered and can be handled by established handlers, just like other event types.

In addition to using `bind()` to establish event handlers, jQuery also provides a handful of convenience functions to establish the handlers, as follows:

Method syntax: jQuery Ajax event establishers

`ajaxComplete(callback)`
`ajaxError(callback)`
`ajaxSend(callback)`
`ajaxStart(callback)`
`ajaxStop(callback)`
`ajaxSuccess(callback)`

Establishes the passed callback as an event handler for the jQuery Ajax event specified by the method name.

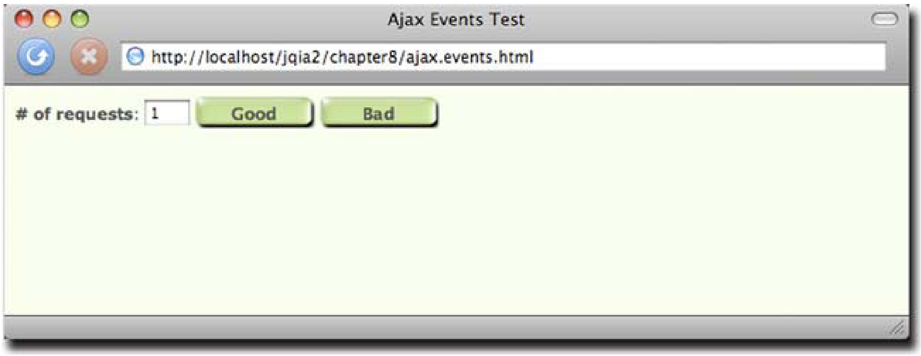
Parameters

`callback`	(Function) The function to be established as the Ajax event handler. The function context (`this`) is the DOM element upon which the handler is established. Parameters may be passed as outlined in table 8.3.

Returns

The wrapped set.

Let's put together a simple example of how some of these methods can be used to easily track the progress of Ajax requests. The layout of our test page (it's too simple to be called a Lab) is shown in figure 8.7 and is available at URL http://localhost[:8080]/jqia2/chapter8/ajax.events.html.



# of requests: 1

Figure 8.7 **The initial display of the page we'll use to examine the jQuery Ajax events by firing multiple events and observing the handlers**

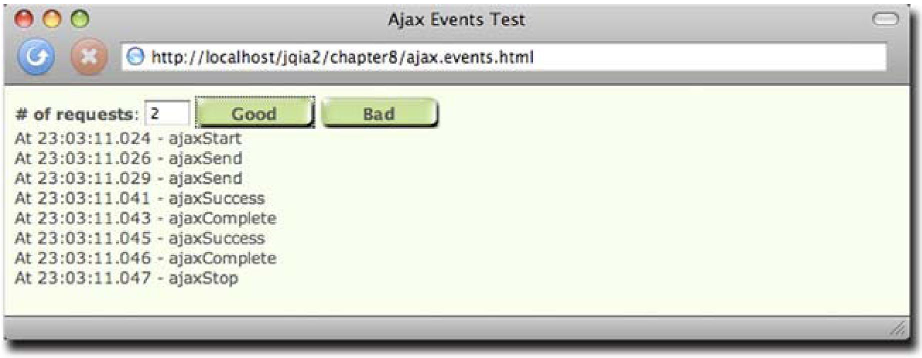
This page exhibits three controls: a count field, a Good button, and a Bad button. These buttons are instrumented to issue the number of requests specified by the count field. The Good button will issue requests to a valid resource, whereas the Bad button will issue that number of requests to an invalid resource that will result in failures.

Within the ready handler on the page, we also establish a number of event handlers as follows:

```
$('body').bind(
  'ajaxStart ajaxStop ajaxSend ajaxSuccess ajaxError ajaxComplete',
  function(event){ say(event.type); }
);
```

This statement establishes a handler for each of the various jQuery Ajax event types that emits a message to the on-page "console" (which we placed below the controls), showing the event type that was triggered.

Leaving the request count at 1, click the Good button and observe the results. You'll see that each jQuery Ajax event type is triggered in the order depicted in table 8.3. But to understand the distinctive behavior of the ajaxStart and ajaxStop events, set the count control to 2, and click the Good button. You'll see a display as shown in figure 8.8.



# of requests: 2

```
At 23:03:11.024 - ajaxStart
At 23:03:11.026 - ajaxSend
At 23:03:11.029 - ajaxSend
At 23:03:11.041 - ajaxSuccess
At 23:03:11.043 - ajaxComplete
At 23:03:11.045 - ajaxSuccess
At 23:03:11.046 - ajaxComplete
At 23:03:11.047 - ajaxStop
```

Figure 8.8 **When multiple requests are active, the ajaxStart and ajaxStop events are called around the set of requests rather than for each.**

Here we can see how, when multiple requests are active, the ajaxStart and ajaxStop events are triggered only once for the entire set of concurrent requests, whereas the other event types are triggered on a per-request basis.

Now try clicking the Bad button to generate an invalid request, and observe the event behavior.

Before we move on to the next chapter, let's put all this grand knowledge to use, shall we?

8.5 Putting it all together

It's time for another comprehensive example. Let's put a little of everything we've learned so far to work: selectors, DOM manipulation, advanced JavaScript, events, effects, and Ajax. And to top it all off, we'll implement another custom jQuery method!

For this example, we'll once again return to The Boot Closet page. To review, look back at figure 8.6 to remind yourself where we left off, because we're going to continue to enhance this page.

In the detailed information of the boots listed for sale (evident in figure 8.6), terms are used that our customers may not be familiar with—terms like "Goodyear welt" and "stitch-down construction." We'd like to make it easy for customers to find out what these terms mean, because an informed customer is usually a happy customer. And happy customers buy things! We like that.

We could be all 1998 about it and provide a glossary page that customers navigate to for reference, but that would move the focus away from where we want it—the pages where they can buy our stuff! We could be a *little* more modern about it and open a pop-up window to show the glossary or even the definition of the term in question. But even that's being terribly old-fashioned.

If you're thinking ahead, you might be wondering if we could use the `title` attribute of DOM elements to display a *tooltip* (sometimes called a *flyout)* containing the definition when customers hover over the term with the mouse cursor. Good thinking! That would allow the definition to be shown in-place without requiring customers to move their focus elsewhere.

But the `title` attribute approach presents some problems for us. First, the flyout only appears if the mouse cursor hovers over the element for a few seconds—and we'd like to be a bit more overt about it, displaying the information immediately after clicking a term. But more importantly, some browsers will truncate the text of a `title` flyout to a length far too short for our purposes.

So we'll build our own!

We'll somehow identify terms that have definitions, change their appearance to allow the user to easily identify such elements as clickable (giving them what's termed an "invitation to engage"), and instrument the elements so that a mouse click will display a flyout containing a description of the term. Subsequently clicking the flyout will remove it from the display.

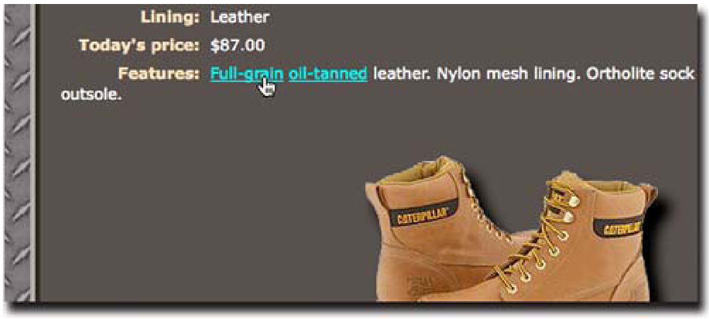
We're also going to write it as a generally reusable plugin, so we need to make sure of two very important things:

- There'll be no hard-coding of anything that's specific to The Boot Closet.
- We'll give the page authors maximum flexibility for styling and layout (within reason).

We'll call our new plugin the *Termifier*, and figures 8.9a through 8.9c display a portion of our page showing the behavior that we'll be adding.

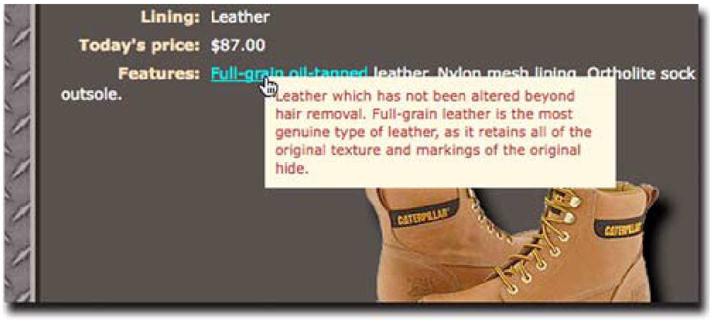
In figure 8.9a, we see the description of the item with the terms "Full-grain" and "oil-tanned" highlighted. Clicking Full-grain causes the Termifier flyout containing the term's definition to be displayed, as shown in figures 8.9b and 8.9c. In the rendition of figure 8.9b, we've supplied some rather simple CSS styling; in figure 8.9c we've gotten a bit more grandiose. We need to make sure that the plugin code allows for such flexibility.

Let's get started.



Lining: Leather
Today's price: $87.00
Features: Full-grain oil-tanned leather. Nylon mesh lining. Ortholite sock outsole.

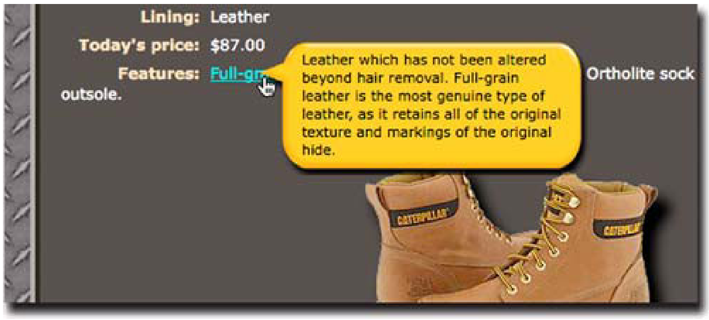
Figure 8.9a The terms "full-grain" and "oil-tanned" have been instrumented for "termifying" by our handy new plugin.



Lining: Leather
Today's price: $87.00
Features: Full-grain oil-tanned leather. Nylon mesh lining. Ortholite sock outsole.

Leather which has not been altered beyond hair removal. Full-grain leather is the most genuine type of leather, as it retains all of the original texture and markings of the original hide.

Figure 8.9b The Termifier pane deployed using simple styling specified by CSS external to the plugin.



Lining: Leather
Today's price: $87.00
Features: Full-gr Ortholite sock outsole.

Leather which has not been altered beyond hair removal. Full-grain leather is the most genuine type of leather, as it retains all of the original texture and markings of the original hide.

Figure 8.9c The Termifier pane with fancier styling—we need to give a user of our plugin this kind of flexibility.

8.5.1 Implementing the Termifier

As you'll recall, adding a jQuery method is accomplished by using the `$.fn` property. Because we've called our new plugin the Termifier, we'll name the method `termifier()`.

The `termifier()` method will be responsible for instrumenting each element in its matched set to achieve the following plan:

- Establish a `click` handler on each matched element that initiates the display of the Termifier flyout.
- Once clicked, the term defined by the current element will be looked up using a server-side resource.
- Once received, the definition of the term will be displayed in a flyout using a fade-in effect.
- The flyout will be instrumented to fade out once clicked within its boundaries.
- The URL of the server-side resource will be the only required parameter; all other options will have reasonable defaults.

The syntax for our plugin is as follows:

Method syntax: termifier

`termifier(url,options)`

Instruments the wrapped elements as Termifier terms. The class name `termified` is added to all wrapped elements.

Parameters

`url` (String) The URL of the server-side action that will retrieve term definitions.

`options` (Object) Specifies the options as follows:

- `paramName`—The request parameter name to use to send the term to the server-side action. If omitted, a default of `term` is used.
- `addClass`—A class name to be added to the outer container of the generated Termifier pane. This is in addition to the class name `termifier`, which is always added.
- `origin`—An object hash containing the properties `top` and `left` that specify an offset for the Termifier pane from the cursor position. If omitted, the origin is placed exactly at the cursor position.
- `zIndex`—The z-index to assign to the Termifier pane. Defaults to 100.

Returns

The wrapped set.

We'll begin the implementation by creating a skeleton for our new `termifier()` method in a file named `jquery.jqia.termifier.js`:

```
(function($){

  $.fn.termifier = function(actionURL,options) {
    //
    // implementation goes here
    //
    return this;
  };

})(jQuery);
```

This skeleton uses the pattern outlined in the previous chapter to ensure that we can freely use the `$` in our implementation, and creates the wrapper method by adding the new function to the `fn` prototype. Also note how we set up the return value right away to ensure that our new method plays nice with jQuery chaining.

Now, on to processing the options. We want to merge the user-supplied options with our own defaults:

```
var settings = $.extend({
  origin: {top:0,left:0},
  paramName: 'term',
  addClass: null,
  actionURL: actionURL
},options||{});
```

We've seen this pattern before, so we won't belabor its operation, but note how we've added the `actionURL` parameter value into the `settings` variable. This collects everything we'll need later into one tidy place for the closures that we'll be creating.

Having gathered all the data, we'll now move on to defining the click handler on the wrapped elements that will create and display the Termifier pane. We start setting that up as follows:

```
this.click(function(event){
  $('div.termifier').remove();
  //
  // create new Termifier here
  //
});
```

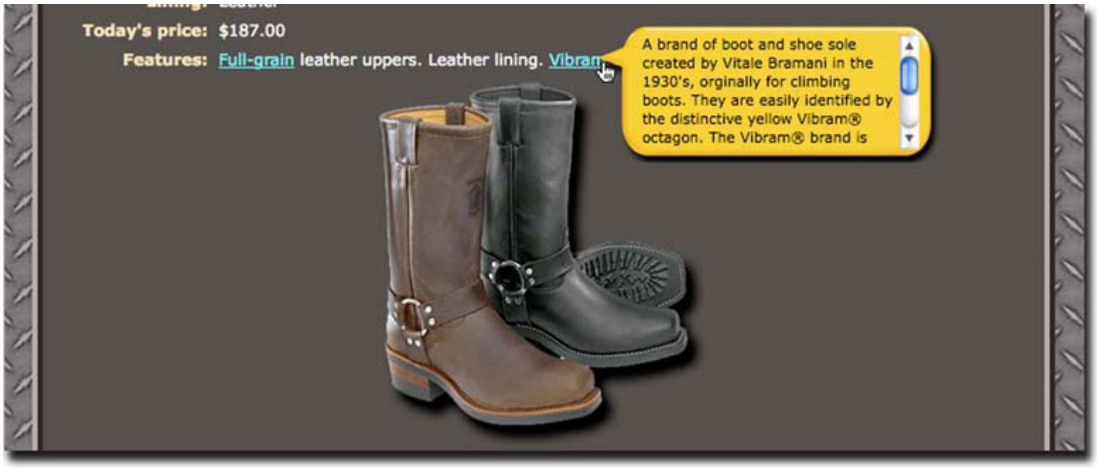
When a termified element is clicked, we want to get rid of any previous Termifier panes that are lying around before we create a new one. Otherwise, we could end up with a screen littered with them, so we locate all previous instances and remove them from the DOM.

NOTE Can you think of another approach that we might have taken to ensure that there is only one Termifier pane ever displayed?

With that, we're now ready to create the structure of our Termifier pane. You might think that all we need to do is create a single `<div>` into which we can shove the term definition, but although that *would* work, it would also limit the options that we offer the users of our plugin. Consider the example of figure 8.9c, where the text needs to be placed precisely in relation to the background "bubble" image.

So we'll create an outer `<div>`, and then an inner `<div>` into which the text will be placed. This won't only be useful for placement; consider the situation of figure 8.10, in which we have a fixed-height construct and text that's longer than will fit. The presence of the inner `<div>` allows the page author to user the `overflow` CSS rule to add scrollbars to the flyout text.

Let's examine the code that creates the outer `<div>`:



Today's price: $187.00

Features: Full-grain leather uppers. Leather lining. Vibram

A brand of boot and shoe sole created by Vitale Bramani in the 1930's, orginally for climbing boots. They are easily identified by the distinctive yellow Vibram® octagon. The Vibram® brand is

Figure 8.10 Having two `<div>` elements to play with gives the page author some leeway to do things like scroll the inner text.

```
$('<div>')
  .addClass('termifier' +
    (settings.addClass ? (' ') + settings.addClass : ''))
  .css({
    position: 'absolute',
    top: event.pageY - settings.origin.top,
    left: event.pageX - settings.origin.left,
    display: 'none'
  })
  .click(function(event){
    $(this).fadeOut('slow');
  })
  .appendTo('body')
```

❶ Creates outer <div>

❷ Adds class name(s)

❸ Assigns CSS positioning

❹ Removes from display on click

❺ Attaches to DOM

In this code, we create a new `<div>` element ❶ and proceed to adjust it. First, we assign the class name `termifier` to the element ❷ so that we can easily find it later, as well as to give the page author a hook onto which to hang CSS rules. If the caller provided an `addClass` option, it's also added.

We then apply some CSS styling ❸. All we do here is the minimum that's necessary to make the whole thing work (we'll let the page author provide any additional styling through CSS rules). The element is initially hidden, and it's absolutely positioned at the location of the mouse event, adjusted by any `origin` provided by the caller. The latter is what allows a page author to adjust the position so that the tip of the bubble's pointer in figure 8.9c appears at the click location.

After that, we establish a click event handler ❹ that removes the element from the display when clicked upon. Finally, the element is attached to the DOM ❺.

OK, so far so good. Now we need to create the inner `<div>`—the one that will carry the text—and append it to the element we just created, so we continue like this:

```
.append(                                          ❶ Appends inner to outer <div>
  $('<div>').load(
    settings.actionURL,                                         ❷ Fetches and
    encodeURIComponent(settings.paramName) + '=' +                injects definition
      encodeURIComponent($(event.target).text()),                    ❸ Provides term
    function(){
      $(this).closest('.termifier').fadeIn('slow');              Fades Termifier
    }                                                          ❹ into view
  )
```

Note that this is a continuation of the same statement that created the outer <div>—have we not been telling you all along how powerful jQuery chaining is?

In this code fragment, we create and append ❶ the inner <div> and initiate an Ajax request to fetch and inject the definition of the term ❷. Because we're using `load()` and want to force a `GET` request, we need to supply the parameter info as a text string. We can't rely on `serialize()` here because we're not dealing with any form controls, so we make use of JavaScript's `encodeURIComponent()` method to format the query string ourselves ❸.

In the completion callback for the request, we find the parent (marked with the `termifier` class) and fade it into view ❹.

Before dancing a jig, we need to perform one last act before we can declare our plugin complete; we must add the class name `termified` to the wrapped elements to give the page author a way to style termified elements:

```
this.addClass('termified');
```

There! Now we're done and can enjoy our lovely beverage.

The code for our plugin is shown in its entirety in listing 8.7, and it can be found in file chapter8/jquery.jqia.termifier.js.

Listing 8.7 The complete implementation of the Termifier plugin

```
(function($){

  $.fn.termifier = function(actionURL,options) {
    var settings = $.extend({
      origin: {top:0,left:0},
      paramName: 'term',
      addClass: null,
      actionURL: actionURL
    },options||{});
    this.click(function(event){
      $('div.termifier').remove();
      $('<div>')
        .addClass('termifier' +
          (settings.addClass ? (' ') + settings.addClass : ''))
        .css({
          position: 'absolute',
          top: event.pageY - settings.origin.top,
          left: event.pageX - settings.origin.left,
          display: 'none'
        })
        .click(function(event){
          $(this).fadeOut('slow');
        })
        .appendTo('body')
        .append(
          $('<div>').load(
            settings.actionURL,
            encodeURIComponent(settings.paramName) + '=' +
              encodeURIComponent($(event.target).text()),
            function(){
              $(this).closest('.termifier').fadeIn('slow');
            }
          )
        );
    });
    this.addClass('termified');
    return this;
  };

})(jQuery);
```

That was the hard part. The easy part should be putting the Termifier to use on our Boot Closet page—at least, if we did it right. Let's find out if we did.

8.5.2 Putting the Termifier to the test

Because we rolled all the complex logic of creating and manipulating the Termifier flyout into the `termifier()` method, using this new jQuery method on the Boot Closet page is relatively simple. But first we have some interesting decisions to make.

For example, we need to decide how to identify the terms on the page that we wish to termify. Remember, we need to construct a wrapped set of elements whose content contains the term elements for the method to operate on. We *could* use a `<span>` element with a specific class name; perhaps something like this:

```
<span class="term">Goodyear welt</span>
```

In this case, creating a wrapped set of these elements would be as easy as `$('span.term')`.

But some might feel that the `<span>` markup is a bit wordy. Instead, we'll leverage the little-used `<abbr>` HTML tag. The `<abbr>` tag was added to HTML 4 in order to help identify abbreviations in the document. Because the tag is intended purely for identifying document elements, none of the browsers do much with these tags, either in the way of semantics or visual rendition, so it's perfect for our use.

NOTE HTML 4[2] defines a few more of these semantic tags, such as `<cite>`, `<dfn>`, and `<acronym>`. The HTML 5 draft specification[3] proposal adds even more of these semantic tags, whose purpose is to provide structure rather than provide layout or visual rendition directives. Among such tags are `<section>`, `<article>`, and `<aside>`.

[2] HTML 4.01 specification, http://www.w3.org/TR/html4/.

[3] HTML 5 draft specification, http://www.w3.org/html/wg/html5/.

Therefore, the first thing we need to do is modify the server-side resource that returns the item details to enclose terms that have glossary definitions in `<abbr>` tags. Well, as it turns out, the `fetchProductDetails` action already does that. But because the browsers don't do anything with the `<abbr>` tag, you might not have even noticed, unless you've already taken a look inside the action file or inspected the action's response. A typical response (for style 7141922) contains this:

```
<div>
  <label>Features:</label> <abbr>Full-grain</abbr> leather uppers. Leather
  lining. <abbr>Vibram</abbr> sole. <abbr>Goodyear welt</abbr>.
</div>
```

Note how the terms "Full-grain", "Vibram", and "Goodyear welt" are identified using the `<abbr>` tag.

Now, onward to the page itself. Starting with the code of phase three (listing 8.6) as a starting point, let's see what we need to add to the page in order to use the Termifier. We need to bring the new method into the page, so we add the following statement to the `<head>` section (after jQuery itself has loaded):

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

We need to apply the `termifier()` method to any `<abbr>` tags added to the page when item information is loaded, so we add a callback to the `load()` method that fetches the product detail information. That callback uses the Termifier to instrument all `<abbr>` elements. The augmented `load()` method (with changes in bold) is as follows:

```
$('#productDetailPane').load(
  '/jqia2/action/fetchProductDetails',
  $('#bootChooserControl').serialize(),
  function(){ $('abbr').termifier('/jqia2/action/fetchTerm'); }
);
```

The added callback creates a wrapped set of all `<abbr>` elements and applies the `termifier()` method to them, specifying a server-side action of `fetchTerm` and letting all options default.

And that's it! (Well, almost.)

CLEANING UP AFTER OURSELVES

Because we wisely encapsulated all the heavy lifting in our reusable jQuery plugin, using it on the page is even easier than pie! And we can as easily use it on any other page or any other site. Now that's what *engineering* is all about!

But there is one little thing we forget. We built into our plugin the removal of any Termifier flyouts when another one is displayed, but what happens if the user chooses a new style? Whoops! We'd be left with a Termifier pane that's no longer relevant. So we need to remove any displayed Termifiers whenever we reload the product detail.

We *could* just add some code to the `load()` callback, but that seems wrong, tightly coupling the Termifier to the loading of the product details. We'd be happier if we could keep the two decoupled and just listen for an event that tells us when its time to remove any Termifiers.

If the ajaxComplete event came to mind, treat yourself to a Maple Walnut Sundae or whatever other indulgence you use to reward yourself for a great idea. We can listen for ajaxComplete events and remove any Termifiers that exist when the event is tied to the `fetchProductDetails` action:

```
$('body').ajaxComplete(function(event,xhr,options){
 if (options.url.indexOf('fetchProductDetails') != -1) {
   $('div.termifier').remove();
 }
});
```

Now let's take a look at how we applied those various styles to the Termifier flyouts.

TERMIFYING IN STYLE

Styling the elements is quite a simple matter. In our style sheet we can easily apply rules to make the termified terms, and the Termifier pane itself, look like the display of figure 8.9b. Looking in `bootcloset.css`, we find this:

```
abbr.termified {
  text-decoration: underline;
  color: aqua;
  cursor: pointer;
}

div.termifier {
  background-color: cornsilk;
  width: 256px;
  color: brown;
  padding: 8px;
  font-size: 0.8em;
}
```

These rules give the terms a link-ish appearance that invites users to click the terms, and gives the Termifier flyouts the simple appearance shown in figure 8.9b. This version of the page can be found at http://localhost[:8080]/jqia2/chapter8/bootcloset/phase.4a.html.

To take the Termifier panes to the next level, shown in figure 8.9c, we only need to be a little clever and use some of the options we provided in our plugin. For the fancier version, we call the Termifier plugin (within the `load()` callback) with this code:

```
$('#productDetailPane').load(
  '/jqia2/action/fetchProductDetails',
  $('#bootChooserControl').serialize(),
  function(){ $('abbr')
    .termifier(
      '/jqia2/action/fetchTerm',
      {
        addClass: 'fancy',
        origin: {top: 28, left: 2}
      }
    );
  }
);
```

This call differs from the previous example only by specifying that the class name `fancy` be added to the Termifiers, and that the origin be adjusted so that the tip of the bubble appears at the mouse event location.

To the style sheet we add this (leaving the original rule):

```
div.termifier.fancy {
  background: url('images/termifier.bubble.png') no-repeat transparent;
  width: 256px;
  height: 104px;
}

div.termifier.fancy div {
  height: 86px;
  width: 208px;
  overflow: auto;
  color: black;
  margin-left: 24px;
}
```

This adds all the fancy stuff that can be seen in figure 8.9c.

This new page can be found at http://localhost[:8080]/jqia2/chapter8/boot-closet/phase.4b.html. Our new plugin is useful and powerful, but as always, we can make improvements.

8.5.3 Improving the Termifier

Our brand-spankin'-new jQuery plugin is quite useful as is, but it does have some minor issues and the potential for some major improvements. To hone your skills, here's a list of possible changes you could make to this method or to the Boot Closet page:

- Add an option (or options) that allows the page author to control the fade durations or, perhaps, even to use alternate effects.
- The Termifier flyout stays around until the customer clicks it, another one is displayed, or the product details are reloaded. Add a timeout option to the Termifier plugin that automatically makes the flyout go away if it's still displayed after the timeout has expired.
- Clicking the flyout to close it introduces a usability issue, because the text of the flyout can't be selected for cut-and-paste. Modify the plugin so that it closes the flyout if the user clicks anywhere on the page *except* on the flyout.
- We don't do any error handling in our plugin. How would you enhance the code to gracefully deal with invalid caller info, or server-side errors?
- We achieved the appealing drop shadows in our images by using PNG files with partial transparencies. Although most browsers handle this file format well, IE 6 doesn't and displays the PNG files with white backgrounds. To deal with this, we could also supply GIF formats for the images without the drop shadows. Although support for IE 6 is waning (in fact, Google has dropped support for IE 6 as of

March 1, 2010), how would you enhance the page to detect when IE 6 is being used and to replace all the PNG references with their corresponding GIFs?

- While we're talking about the images, we only have one photo per boot style, even when multiple colors are available. Assuming that we have photo images for each possible color, how would you enhance the page to show the appropriate image when the color is changed?

Can you think of other improvements to make to this page or the `termifier()` plugin? Share your ideas and solutions at this book's discussion forum, which you can find at http://www.manning.com/bibeault2.

8.6 Summary

Not surprisingly, this is one of the longest chapters in this book. Ajax is a key part of the new wave of DOM-scripted applications, and jQuery is no slouch in providing a rich set of tools for us to work with.

For loading HTML content into DOM elements, the `load()` method provides an easy way to grab the content from the server and make it the contents of any wrapped set of elements. Whether a `GET` or `POST` method is used is determined by how any parameter data to be passed to the server is provided.

When a `GET` is required, jQuery provides the utility functions `$.get()` and `$.getJSON()`; the latter being useful when JSON data is returned from the server. To force a `POST`, the `$.post()` utility function can be used.

When maximum flexibility is required, the `$.ajax()` utility function, with its ample assortment of options, lets us control the most minute aspects of an Ajax request. All other Ajax features in jQuery use the services of this function to provide their functionality.

To make managing the bevy of options less of a chore, jQuery provides the `$.ajaxSetup()` utility function that allows us to set default values for any frequently used options to the `$.ajax()` function (and to all of the other Ajax functions that use the services of `$.ajax()`).

To round out the Ajax toolset, jQuery also allows us to monitor the progress of Ajax requests by triggering Ajax events at the various stages, allowing us to establish handlers to listen for those events. We can `bind()` the handlers, or use the convenience methods: `ajaxStart()`, `ajaxSend()`, `ajaxSuccess()`, `ajaxError()`, `ajaxComplete()`, and `ajaxStop()`.

With this impressive collection of Ajax tools under our belts, it's easy to enable rich functionality in our web applications. And remember, if there's something that jQuery doesn't provide, we've seen that it's easy to extend jQuery by leveraging its existing features. Or, perhaps there's already a plugin—official or otherwise—that adds exactly what you need!