SAMPLE CHAPTER

Extending jQuery

Keith Wood

FOREWORD BY Dave Methvin

MANNING

Extending jQuery

by Keith Wood

Chapter 12

Copyright 2013 Manning Publications

brief contents

12 Extending Ajax

This chapter covers

- The jQuery Ajax framework
- Adding Ajax prefilters
- Adding Ajax transporters
- Adding Ajax converters

Support for *Ajax* (Asynchronous JavaScript and XML) is one of the key features of jQuery, making it easy to request content from the server and to process the returned data and update the current page accordingly, without requiring a full refresh. You specify the URL to access, you can provide parameters to be sent along, and you can process the returned content in a callback function, as shown here:

```
$.ajax('product.php', {data: {prod_id: 'AB1234'},
    success: function(info) {...}});
```

jQuery also contains several convenience functions that encapsulate the Ajax abilities. For a simple request and response, you can use the `get` function, or to use an alternate parameter encoding, you use the `post` function. To load specific types of data, you can use the `getScript` or `getJSON` functions for JavaScript and *JSON*

(JavaScript Object Notation) content respectively. If you want to place HTML content directly into an element on the page, you can use the `load` function on that element.

```
$.get('product.php', {prod_id: 'AB1234'}, function(info) {...});
$.getScript('product.js');
$('#mydiv').load('product.php', {prod_id: 'AB1234'}, function(info) {...});
```

jQuery offers additional abilities to set default values for all Ajax processing and to register handlers for events that occur during the Ajax lifecycle.

Although there's a lot of built-in functionality, if you retrieve content in an unusual format, you may find it necessary to process that content yourself. Fortunately, jQuery includes several extension points within its Ajax framework to let you customize the downloading and handling of the requested content.

12.1 The Ajax framework

Underlying the Ajax support of jQuery is its management of an `XMLHttpRequest` object (or the corresponding ActiveX object in some versions of IE) to perform the actual download of the remote content. jQuery carries out several steps as it executes an Ajax request. You can add your own processing at various stages to implement special requirements for retrieving information.

A request starts by applying prefilters that may affect how the call proceeds, before selecting a transport mechanism to use for the actual download. When the content is received, it may be run through a converter to obtain an alternate format for the user to work with. Figure 12.1 shows the sequence of operations for a basic Ajax call.

The whole process is driven by the data type specified for your request, the standard ones being `text`, `html`, `xml`, `script`, `json`, and `jsonp`. jQuery will try to determine the

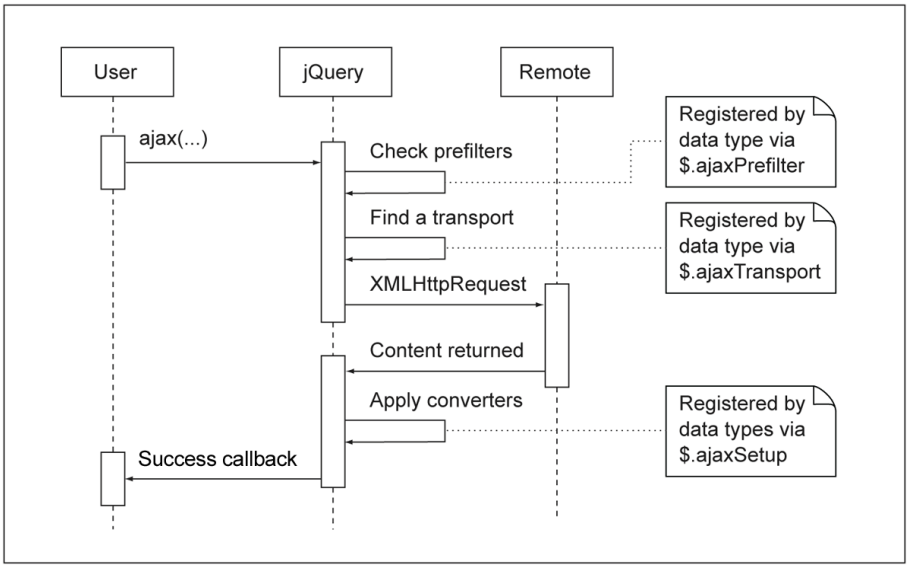


Figure 12.1 Sequence diagram for a standard Ajax call, showing extension points

data type for retrieved content, if it wasn't given, by inspecting the returned MIME type. The conversion process is invoked if the returned data type doesn't match the requested one.

Since version 1.5, jQuery wraps the native `XMLHttpRequest` object to provide additional functionality, with the enhanced object being known as a `jqXHR` object. The wrapper also acts as a `Deferred` object,[1] allowing you to add extra callbacks to be triggered when the Ajax processing succeeds or generates an error.

> **NOTE** The `success`, `error`, and `complete` functions of the `jqXHR` object have been deprecated in jQuery 1.8 and should be replaced by the corresponding `done`, `fail`, and `always` functions.

12.1.1 Prefilters

A *prefilter* is a function that's called before the actual request to the server is made. It allows you to preprocess that request and alter how it proceeds, and it's useful when creating custom data types. For example, you could add extra headers to the request, or even cancel it entirely.

The prefilters are called after parameter serialization has occurred—after any `data` option is converted to a string (assuming `processData` is `true`), but before the Ajax framework looks for an appropriate transporter. Prefilters are identified by the data type to which they apply, such as `html` or `script`, and can be set to operate on all data types by using `*`. The filters specific to a data type are executed first, before continuing on to those for all types.

You can register a new prefilter by calling the `$.ajaxPrefilter` function and providing the associated data type and the function to be called for it. The parameters to your function include the Ajax options and a reference to the `jqXHR` object being used for the remote access. To cancel a request, you call the `abort` function on the latter.

For example, to change the user agent identity for all `html` requests, you could use a prefilter like this:

```
$.ajaxPrefilter('html', function(options, originalOptions, jqXHR) {
    jqXHR.setRequestHeader('User-Agent', 'Unknown');
});
```

12.1.2 Transports

An Ajax *transport* provides the underlying mechanism to retrieve the requested data from the server. Although the `XMLHttpRequest` object is normally used, there are other means available to load content, such as the `src` attribute of an `img` element (to download an image).

New transports are registered by calling the `$.ajaxTransport` function and providing the data type to which this transport applies and the function that returns the custom retrieval object. This transport object supplies two callback functions: one to

[1] jQuery API Documentation, "Category: Deferred Object," http://api.jquery.com/category/deferred-object/.

perform the actual retrieval, and another to clean up if a request is aborted. As for prefilters, you can use * to define a transport for all data types. Only the first matching transport is used, starting with those for specific data types in preference to those for all data types.

For example, to forbid access to any xml documents, you could use a transport like this:

```
$.ajaxTransport('xml', function(options, originalOptions, jqXHR) {
    return {
        send: function(headers, complete) {
            complete('403', 'Forbidden', {});
        },

        abort: function() {}
    };
});
```

jQuery uses this ability itself to deal with cross-domain requests for the script data type.

12.1.3 Converters

Once the data is retrieved from the server, it's not necessarily in the most useful format. A *converter* implements the conversion process, accepting the text-based content as input and generating the appropriate output.

For example, when you request XML from the server, you specify a dataType of xml. This triggers a converter to parse the returned text as an XML document and produce an XML DOM as the final value of the Ajax call. You can then immediately start traversing that DOM to extract the information relevant to your current situation.

Converters are registered in the converters attribute passed to an $.ajaxSetup call. You specify the source and destination data formats in a single string to identify a converter, and associate that value with the function that performs the transformation. You can use * to represent any data type in the converter identifier. For example, the XML converter just described is defined as shown here—converting from text to XML by calling the jQuery parseXML function.

```
ajaxSettings: {
    ...
    converters: {
        ...
        // Parse text as xml
        "text xml": jQuery.parseXML
    },
    ...
}
```

You can define your own converters to transform custom data formats into alternative formats in a similar manner.

Each of these extension points is covered in greater detail in the following sections, starting with the prefilters.

12.2 Adding an Ajax prefilter

Ajax prefilters let you preprocess an Ajax call and potentially alter how it proceeds, such as by changing the `timeout` setting for a slow server, or even preventing the remote call. jQuery uses prefilters itself to handle the `json` and `jsonp` data types, allowing it to install the appropriate callbacks for these requests. You'll see two examples of prefilters next: one to change the data type for a request and the other to cancel a request altogether.

12.2.1 Changing the data type

Prefilters let you modify settings on the `XMLHttpRequest` object (wrapped as `jqXHR`) before a request is sent. In addition, they can be used to change the data type for a request by returning the desired data type as its value. This has the effect of making all subsequent operations use the new data type, including a reprocessing of the prefiltering based on that new value.

The following listing shows how to enforce a data type based on the requested URL.

Listing 12.1 Changing the data type

```
/* Set CSV data type. */
$.ajaxPrefilter(function(options, originalOptions, jqXHR) {
    if (options.url.match(/.*\.csv/)) {
        return 'csv';
    }
});
```

1 Define prefilter for all types
2 If filename is *.csv...
3 ...change data type

You call `$.ajaxPrefilter` to register a new prefilter function 1. Note that you don't have to provide a data type to match against, as `*` is assumed for all types. In this case, you want all requests for files with the `.csv` extension to be treated as the `csv` data type, so you test the provided URL to see whether it ends in the required text 2 and return the new data type if so 3.

To continue with the standard processing, you don't return anything. Section 12.4 describes how you might deal with this CSV content via a converter.

12.2.2 Disabling Ajax processing

There may be times when you don't want Ajax calls to be made at all, or perhaps you want to prevent certain types of calls. The next listing shows how a prefilter can meet this requirement by cancelling selected requests.

Listing 12.2 Disabling Ajax processing

```
/* Disable Ajax processing. */
$.ajax.disableDataTypes = [];

$.ajaxPrefilter('*', function(options, originalOptions, jqXHR) {
    if ($.inArray(options.dataType, $.ajax.disableDataTypes) > -1) {
        jqXHR.abort();
    }
});
```

1 List of disabled data types
2 Define prefilter for all types
3 Disable specified types

You should start by declaring a list of data types that are to be disabled (1). The user can then add selected types as necessary:

```
$.ajax.disableDataTypes.push('html');
```

You create the prefilter via a call to `$.ajaxPrefilter` (2), providing the data type to which the filter applies and the function to be used in those circumstances. The data type is specified as one or more values separated by spaces, such as `html` or `json jsonp`. You can also use the value `*` (or omit the data type altogether) to apply to all data types. In addition, you can indicate that the filter should be called before any others for a data type by prefixing that type with `+`.

Your prefilter function receives several parameters: `options` holds all the Ajax settings for this request, whether they've been defaulted by jQuery or provided on the Ajax call, whereas `originalOptions` only has the settings specified by the user, and `jqXHR` is a reference to the jQuery `jqXHR` object that will be used. You can examine the options given (specified or defaulted) and can modify the request accordingly through the provided object. In this case, you see whether the requested data type is one of those listed in the set to be disabled, and abort the request if so (3).

Other actions that you could take on the `jqXHR` object include adding headers to the request via the `setRequestHeader(name, value)` function or changing the requested MIME type via the `overrideMimeType(mimeType)` function.

If you need to remove a value from the list of disabled data types, you can use the `$.map` function of jQuery. For example, to remove the `html` data type, you'd write this:

```
$.ajax.disableDataTypes = $.map($.ajax.disableDataTypes, function(v) {
    return (v == 'html' ? null : v);
});
```

You could extend this prefilter to include particular types of requests (`GET` or `POST`) or other Ajax settings.

12.3 Adding an Ajax transport

Ajax transports provide the mechanism for downloading the requested content, and they default to using the standard `XMLHttpRequest` object. But you can implement alternative means for specific data types by adding your own transport function. You can also alter the retrieval procedure for standard data types and add your own functionality to it. Both of these extensions are illustrated in the examples ahead.

12.3.1 Loading image data

Suppose you wanted to preload images onto your page. You could create `Image` elements and go through the process of setting them up, initiating the load requests and reacting when they're ready. Or you could tie into the Ajax framework to get the benefits of all the functionality that jQuery provides surrounding its use, such as authentication and error handling.

To load images via an Ajax call, you could define an `image` transport function that knows how to handle this format, because it's not text-based, using the download abil-

ities of the DOM `Image` object to perform the actual transfer. The following listing shows how you might define the transport to achieve this.

Listing 12.3 Loading image data

```
/* Transport image data. */
$.ajaxTransport('image', function(options, originalOptions, jqXHR) {
    if (options.type === 'GET' && options.async) {
        var image;
        return {
            send: function(headers, complete) {
                image = new Image();
                function done(status) {
                    if (image) {
                        var statusText = (status == 200 ?
                            'success' : 'error');
                        var tmp = image;
                        image = image.onreadystatechange=
                            image.onerror = image.onload = null;
                        complete(status, statusText, {image: tmp});
                    }
                }
                image.onreadystatechange = image.onload = function() {
                    done(200);
                };
                image.onerror = function() {
                    done(404);
                };
                image.src = options.url;
            },

            abort: function() {
                if (image) {
                    image = image.onreadystatechange =
                        image.onerror = image.onload = null;
                }
            }
        };
    }
});
```

1 Define transport for image
2 Only if using GET asynchronously
3 Return transport object
4 Define send request function
5 Callback when request is completed
6 Invoke complete callback
7 Initialize image callbacks
8 Load the image
9 Handle aborted request

You start by calling `$.ajaxTransport` to define the transport function for the `image` data type 1. As for prefilters, the data type can be several types separated by spaces, can be `*` for all types, and can have a prefix of `+` per type to make it the first in the list for that type. Your function's `options` parameter contains the complete set of Ajax options for this call, including those set by jQuery as default values; the `originalOptions` parameter only contains the options specified explicitly by the user on this call. `jqXHR` holds a reference to the jQuery `jqXHR` object normally used by the request. Because you're using an alternative mechanism to load the image, this last parameter is ignored.

Your new transport only applies if the user requested the `GET` format and an asynchronous load (due to the limitations of the actual mechanism you're using), so you need to test for these conditions 2. If the transport does apply, you return a transport

object that allows jQuery to invoke the load process at the appropriate time ❸. The transport object contains two functions: `send` to initiate a download and `abort` to tidy up if it terminates in error.

The `send` function ❹ is used instead of the standard Ajax processing to request content for this data type. Its parameters are a reference to the headers for the request (`headers`) and a callback function to complete the processing (`complete`) within the Ajax framework. As you're using the inherent abilities of the `Image` element to load the data, you start by creating a new `Image` element to work with.

Define a callback function ❺ to be executed when the image load has finished. Within that function, you determine the success or failure of the load and set the status accordingly. Then clean up the internally created `Image` by clearing its callbacks and then setting the variable itself to `null`, allowing the assigned memory to be recovered and preventing memory leaks. The image is still accessible via the local `tmp` variable, but it's no longer available outside of the `done` callback.

Finally, you invoke the `complete` callback provided as a parameter to the `send` call, to inform the jQuery Ajax framework of the outcome of the request ❻. The parameters to the `complete` call are the numeric and text versions of the status, an object containing details about the response, and a string (optional) containing all the response headers—one to a line. This object must contain an attribute named for the data type being requested (`image` in this case) that refers to the actual result. Here you provide a reference to the `Image` element that was loaded.

Having defined the function to handle the load outcome, you assign functions to the standard callbacks on the `Image` element that calls it, and pass along the appropriate status code ❼. The last step is to start the loading process by setting the `src` attribute of the `Image` element to the URL supplied in the Ajax call ❽, which eventually triggers one of the registered callbacks.

The `abort` function of the returned transport object ❾ lets you tidy up if the request fails. In this case, you again clean up the internal `Image` element by setting everything to `null`.

To invoke this alternate transport, you could then make an Ajax call for the new data type, and you'd receive a reference to the loaded image as the parameter to the `success` callback.

```
$.ajax({url: 'img/uluru.jpg', dataType: 'image', success: function(image) {
    $('#img1').replaceWith(image);
}});
```

This transport illustrates how you can use alternative load mechanisms to obtain data from the server. The next example shows how you can simulate normal HTML retrieval by overriding the standard transport.

12.3.2 Simulating HTML data for testing

While testing a plugin that uses Ajax to implement its functionality, you might want to avoid loading from a live site so that you're not dependent on a remote connection. You can use the Ajax transport handling to override the default retrieval and substitute

known content inline instead. This keeps the data used for testing alongside the tests themselves, reducing the possibility of that data getting out of sync or being lost.

To provide a proper simulation of remote access, you should provide a mapping from requested files to their testing content. In addition, you can specify a delay before the content is returned, mirroring network delays during real processing. You could also control the return status of a page for additional test coverage.

Listing 12.4 shows how you might define an Ajax transport override for a `GET` request for `html` content.

Listing 12.4 Simulate HTML data for testing

```
/* Simulate HTML loading. */
$.ajax.simulateHtml = {};

$.ajaxTransport('html', function(options, originalOptions, jqXHR) {
    if (options.type === 'GET') {
        var timer;
        return {
            send: function(headers, complete) {
                var fileName = options.url.replace(
                    /.*\/([^\/]+)$/, '$1');
                var simulate = $.ajax.simulateHtml[fileName] ||
                    $.ajax.simulateHtml['default'];
                timer = setTimeout(function() {
                    complete(simulate.html ? 200 : 404,
                        simulate.html ? 'success' : 'error',
                        {html: simulate.html});
                }, Math.random() * simulate.variation + simulate.delay);
            },

            abort: function() {
                clearTimeout(timer);
            }
        };
    }
});
```

1 Define file mappings
2 Override transport for html
3 Only if a GET request
4 Return transport object
5 Define send request function
6 Extract filename and settings
7 Introduce delay
8 Invoke complete callback
9 Handle aborted request

You start by declaring an object (`$.ajax.simulateHtml`) to hold mappings between particular pages and the content that should be returned 1. The key for attributes in this object could be just the name of the file being requested, depending on whether the server or path names affect the testing outcome. Each attribute value is an object containing several fields: `html` for the actual content returned, `delay` for the minimum delay in milliseconds before the content comes back, and `variation` for a maximum additional delay in milliseconds above that (randomized for each call). If the content is set to a blank string, a 404 ("page not found") error is generated. Allow for handling any provided filename by including an entry indexed by `default`. For example, you could map the `test.html` file as follows:

```
$.ajax.simulateHtml['default'] = {delay: 500, variation: 1000, html: ''};
$.ajax.simulateHtml['test.html'] = {delay: 500, variation: 1000,
    html: '<p>Try this instead</p>'};
```

To override the default `html` transport handling, you define a new transport for that data type ❷. The parameters for the associated transport function are the same as for the previous `image` example: all the options, specified options only, and a `jqXHR` reference. You only use the alternate transport if a `GET` request is made, so you check for that condition before continuing ❸.

The transport function returns a transport object that jQuery can use to implement the Ajax processing ❹. Its `send` function ❺ is called when a request for content for the nominated data type is made, and it receives as parameters the headers for the request and a callback to complete the Ajax processing within the framework.

Within this function, you first extract the name of the file being requested ❻. You can use a regular expression to retrieve the filename—matching everything (`.*`) up to the last slash (`\/`), and then capturing the remainder without any slashes (`([^\/]+)`) up to the end of the string (`$`). The captured remainder (referred to via `$1`) then replaces the entire matched string (everything), to leave you with just the desired filename. From that filename you obtain the mapped response details from the `$.ajax.simulateHtml` object, or use the `default` settings if there is no mapping for that file.

To simulate network delays, you can use the standard JavaScript `setTimeout` function to introduce a delay ❼ made up of a random variation plus the minimum delay specified. When the time expires, you invoke the `complete` callback provided to the `send` function to notify the jQuery Ajax framework that the requested content is available ❽. As before, the parameters to the `complete` call are the numeric and text versions of the status (being `404` and `error` respectively, if there was no return content), an object containing the actual content as indexed by the data type (`html` in this case), and an optional string containing all the headers.

If the request is aborted for some reason, the `abort` function of the transport object lets you tidy up the environment ❾. Here you cancel the timer, if it's still running, via a call to `clearTimeout`.

To use this custom transport within a QUnit test (see chapter 7), you'd provide mappings as shown previously for the test.html page. If you then make an Ajax call for that page, you can test the expected content based on your definition, with the result shown in figure 12.2 for the code in listing 12.5. Remember that you must create asynchronous tests, because the process involves the use of Ajax.

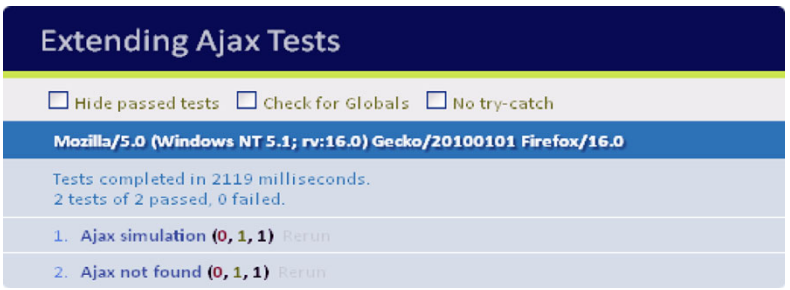


Figure 12.2 Running simulated Ajax tests

Listing 12.5 Testing with HTML simulation

```
asyncTest('Ajax simulation', function() {
    expect(1);
    $.ajax('test.html', {dataType: 'html', success: function(data) {
        equal($(data).text(), 'Try this instead', 'Ajax substitution');
        start();
    }});
});

asyncTest('Ajax not found', function() {
    expect(1);
    $.ajax('other.html', {dataType: 'html', success: function(data) {
        ok(false, 'Page found');
        start();
    }, error: function(jqXHR, textStatus, errorThrown) {
        ok(jqXHR.status == 404 && textStatus == 'error', 'Page missing');
        start();
    }});
});
```

1 Define an asynchronous test
2 Expect one assertion
3 Ajax load of test page
4 Assert correct content
5 Continue test processing
6 Define page not found test
7 Fail if page found
8 Assert correct error
9 Continue test processing

You define an asynchronous test via a call to `asyncTest` instead of `test` 1, and expect one assertion to be made in this test 2. You make the `ajax` call to load the test.html page 3, which should be substituted by the custom transporter. Note that you need to specify the `dataType` for the Ajax call to have the framework use the new functionality. Within the `success` callback, you confirm that the returned content is indeed that from the file mapping 4. Because this is an asynchronous test, you must call the QUnit `start` function to inform the testing framework that the test has finished and that its results can be shown 5.

You should add a second test to confirm that the default and error processing paths also work in the custom transporter 6, once more expecting only one assertion to be made. This time you request a page that doesn't have a mapping specified (other.html) to revert to the default mapping. Because the content specified for the default is blank, the transporter should generate a 404 error in its place. Within the `success` callback, you fail the test, as this path shouldn't occur 7. Instead, the `error` callback should be invoked, allowing you to assert that the status and status text are as expected 8. As before, you need to call `start` to resume the QUnit processing once the Ajax call has completed 9.

12.4 Adding an Ajax converter

Ajax converters let you transform a text-based document into another format that's more directly usable as the result of an Ajax call. jQuery does this already for XML and JSON content by calling the `parseXML` and `parseJSON` functions respectively. You can add your own converters to preprocess your own custom data formats.

12.4.1 Comma-separated values format

CSV (comma-separated values) is a common text format and is often used to transfer tables of information. Each line in a CSV file represents a single record, with field values for that record being separated by commas within that line (hence the name). The first line in a CSV file usually contains the names of the fields, again separated by commas, and isn't treated as a record.

```
First Name,Last Name
Marcus,Cicero
Frank,Zappa
Groucho,Marx
Jane,Austen
```

Things become more complicated when you want to include a comma within a field value. As this would normally be interpreted as the delimiter for the next field, you must indicate that you want it treated as a literal value instead. To achieve this, you surround the entire field value with quotes (`"`), but then you have a problem if you want to include a quote in your field value. The solution for this is to double up the embedded quote characters to escape them.

```
First Name,Last Name,Quote
Marcus,Cicero,"""A room without books is like a body without a soul."""
Frank,Zappa,"""So many books, so little time."""
Groucho,Marx,"""Outside of a dog, a book is man's best friend. ..."""
Jane,Austen,"""The person, be it gentleman or lady, who has not ..."""
```

Due to these additional requirements for including reserved characters within CSV files, it's not so easy to process them directly in JavaScript. To make things simpler, you could convert the CSV text format into a corresponding JavaScript object with a list of field names (`fieldsNames`) and a list of data rows (`rows`). Each row would contain a further list of field values, corresponding in position to the field names already provided. Creating a custom converter lets you integrate the transformation into the standard Ajax processing.

12.4.2 Converting text to CSV

When you request a CSV file from the server, you receive the straight text version of that file by default. To transform that text into a corresponding JavaScript object, you define an Ajax converter, as shown in the following listing.

Listing 12.6 Convert CSV text to object

```
/* Convert CSV file into a JavaScript object.
   @param  csvText  (string) the CSV text
   @return  (object) the extracted CSV with attributes
                     fieldNames (string[]) and rows (string[][]) */
function textToCsv(csvText) {                                    (1) Define conversion function
    var fieldNames = [];
    var fieldCount = 9999;
```

```
var rows = [];
var lines = csvText.match(/[^\r\n]+/g); // Separate lines
for (var i = 0; i < lines.length; i++) {
    if (lines[i]) {
        // Separate columns
        var columns = lines[i].match(/,|"([^"]|"")*"|[^,]*/g);
        var fields = [];
        var field = '';
        for (var j = 0; j < columns.length - 1; j++) {
            // Found a column delimiter
            if (columns[j] == ',') {
                // Save field
                if (fields.length < fieldCount) {
                    fields.push(field);
                }
                field = '';
            }
            else { // Remember field value
                field = columns[j].
                    replace(/^"(.*)"$/, '$1').
                    replace(/""/g, '"') || '';
            }
        }
        if (fields.length < fieldCount) { // Save final field
            fields.push(field);
        }
        if (fieldNames.length == 0) {
            // First line is headers
            fieldNames = fields;
            fieldCount = fields.length;
        }
        else {
            // Fill in missing fields
            for (var j = fields.length; j < fieldCount;j++){
                fields.push('');
            }
            rows.push(fields);
        }
    }
}
// Return extracted CSV data
return {fieldNames: fieldNames, rows: rows};
}
```

2 Separate lines of CSV text

3 Process each line

Separate fields within a line 4

Process each field 5

6 Save field value

7 Set field names from first line

Add missing fields 8

9 Return CSV object

You start by defining a new standalone function to process the conversion 1. It accepts one parameter, which is the full CSV text (`csvText`). After declaring some working variables, you separate the CSV text into individual lines 2, and then process each one in turn 3. The splitting of the text is done by a regular expression supplied to the `match` function. You look for sequences of characters that aren't line feeds or carriage returns (`[^\r\n]+`), and continue that throughout the string (`g` flag). The result of this call is an array of the matching sections (lines) from the text.

If a line isn't empty, you further separate it into its component fields ❹, taking into account quoted fields. As before, a regular expression is used to break up the line, matching with one of the following (separated by `|`):

- A comma (`,`)
- A sequence of characters delimited by double quotes (`"([^"]|"")*"`), which may include escaped quotes (`""`) within it
- A sequence of characters that doesn't contain a comma (`[^,]*`)

These sequences may be repeated throughout the string (`g` flag). The resulting array of matches will contain field values (possibly quoted) as well as comma delimiters.

For each field ❺, if the current match is a comma delimiter, you add a previously found field value (`field`) to the list of fields (`fields`) and reset the field value, but only if you have fewer fields than were identified by the first CSV line (the field names). If the current match isn't a comma, save that match as the next field value ❻ after removing any quotes surrounding it and converting embedded escaped quotes back to a single character. It's necessary to post-process the fields in this manner to cater to sequences of commas with no intervening field values. After examining all the matches from a line, save the last found field value as well.

If this is the first line from the CSV file (meaning there are no field names saved as yet), transfer the field values to the list of field names instead (`fieldNames`) ❼. Otherwise, fill in any missing field values up to the number expected from the list of field names ❽, and add the current row to the list of those already processed (`rows`).

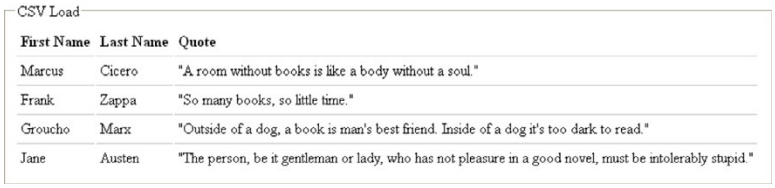
After processing all the lines from the CSV file, return the extracted field names and row contents in a JavaScript object ❾.

NOTE For a more complete text on CSV implementation, see the jQuery CSV plugin at https://code.google.com/p/jquery-csv/.

You'd register the new converter via the `$.ajaxSetup` function by providing a `converters` setting that has an attribute named for the source and destination data types, and which associates that with the conversion function:

```
$.ajaxSetup({converters: {
    'text csv': textToCsv
}});
```

To use the converter, you make an `ajax` call and specify the `dataType` that you desire as `csv`. The `success` callback is then supplied with the converted object, and you can process it directly, instead of having to deal with the CSV-formatted text. Figure 12.3 shows the results of loading the CSV data into a table, based on the code from listing 12.7.


CSV Load

First Name	Last Name	Quote
Marcus	Cicero	"A room without books is like a body without a soul."
Frank	Zappa	"So many books, so little time."
Groucho	Marx	"Outside of a dog, a book is man's best friend. Inside of a dog it's too dark to read."
Jane	Austen	"The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."

Figure 12.3 CSV data loaded into a table

Listing 12.7 Retrieving a CSV object

```
$.ajax({url: 'quotes.csv', dataType: 'csv', success: function(csv) {
    var table = '<table><thead><tr>';
    for (var i = 0; i < csv.fieldNames.length; i++) {
        table += '<th>' + csv.fieldNames[i] + '</th>';
    }
    table += '</tr></thead><tbody>';
    for (var i = 0; i < csv.rows.length; i++) {
        table += '<tr>';
        for (var j = 0; j < csv.fieldNames.length; j++) {
            table += '<td>' + csv.rows[i][j] + '</td>';
        }
        table += '</tr>';
    }
    table += '</tbody></table>';
    $('#tableResult').append(table);
}});
```

Request CSV file and convert 1

2 Create table header

3 Process each row

4 Create table rows

5 Add table to the page

You load your CSV file by making an `ajax` call and specifying the URL and `dataType` of `csv` 1. Note that if you include the prefilter from section 12.2.1 in your page, you don't need to specify the data type, as it would be set automatically based on the URL extension.

Once jQuery has loaded the file, it determines how to convert from the default format (`text`) to the requested one, finds the registered converter, and applies it. The result is the JavaScript object that contains the extracted CSV data, and that object is passed to the `success` callback for further processing.

Within the callback, you build up the new table as a string value (`table`), starting with heading cells for each field name 2. Next, you process each row 3 and add detail cells to the table to show the field values 4. Finally, you add the new table to the page 5.

12.4.3 Converting CSV to a table

Because presenting CSV content in a table is a common occurrence, you can take the conversion process one step further and transform the CSV object extracted in listing 12.7 into a table for direct inclusion in the page, as shown by the converter in listing 12.8.

Listing 12.8 Converting a CSV object to a table

```
/* Convert JavaScript CSV object into a HTML table.
   @param  csv  (object) the CSV object
   @return  (jQuery) the data in a table */
function csvToTable(csv) {
    var table = '<table><thead><tr>';
    for (var i = 0; i < csv.fieldNames.length; i++) {
        table += '<th>' + csv.fieldNames[i] + '</th>';
    }
    table += '</tr></thead><tbody>';
```

1 Define conversion function

2 Generate CSV table

```
    for (var i = 0; i < csv.rows.length; i++) {
        table += '<tr>';
        for (var j = 0; j < csv.fieldNames.length; j++) {
            table += '<td>' + csv.rows[i][j] + '</td>';
        }
        table += '</tr>';
    }
    table += '</tbody></table>';
    return $(table);                                    ❸ Return created table element
}
```

As before, you create a standalone function, but this one converts the CSV object into an HTML table ❶. The body of this function ❷ is identical to the `success` callback used in the previous converter (see listing 12.7). Generate a table as a string value by stepping through each field name for header cells, then through each row for detail cells. The resulting text is instantiated as DOM elements by jQuery, and that object is returned ❸.

Make the new converter available by calling `$.ajaxSetup` again with the new transformation listed in its `converters` option. Note that the starting data type is the CSV object created by the previous converter:

```
$.ajaxSetup({converters: {
    'csv table': csvToTable
}});
```

Now the code required to load the CSV file, convert it into a table, and display it is much shorter. Invoke `ajax` and pass it the URL of the CSV file to load. Specify the data type as a chain—first converting the text to a CSV object, then transforming that into a table. Note that using the prefilter from section 12.2.1 alongside these converters would remove the need to specify the initial `csv` data type, as that would be set automatically. The resulting `table` element is received as the parameter to the `success` callback and can be added to the page directly:

```
$.ajax({url: 'quotes.csv', dataType: 'csv table',
    success: function(table) {
        $('#tableResult').append(table);
    }
});
```

Creating your own converters lets you consistently transform one data format into another, starting from the straight text retrieved by the basic Ajax processing, and possibly progressing through one or more intermediate steps, before arriving at the format that's most appropriate for the task at hand.

12.5 Ajax plugins

The Ajax plugins described in this chapter are available for download from the book's website. Included is a web page that demonstrates how the various extensions work in conjunction with the Ajax framework.

What you need to know

jQuery simplifies accessing remote resources through its `ajax` and related functions.

Extend the Ajax processing when you have additional requirements for remote access and data formats.

You can enhance or prevent a remote request by registering a prefilter via `$.ajaxPrefilter`.

Use `$.ajaxTransport` to register a new mechanism for retrieving remote content.

Provide additional data conversion options via the `converters` attribute within an `$.ajaxSetup` call.

You can chain data types to create a conversion pipeline to obtain the data format best suited to the task at hand.

Try it yourself

Create a new convertor, similar to the CSV-to-table example, that converts a CSV object into a list. Each list entry contains multiple lines showing the label for each field followed by its value in that record. Register the new converter and apply it to the quotation data provided.

12.6 Summary

jQuery makes using Ajax simple by hiding the use of the `XMLHttpRequest` object behind an easy-to-use interface. The `ajax` function gives you complete control over the process, whereas the associated convenience functions enable simpler interactions more readily. As a request is executed, the Ajax framework first sees whether any prefilters need to be applied to it, possibly modifying or even cancelling it. Next, a transport mechanism is found that understands the requested data format and can download the content appropriately. Finally, a converter may be invoked to transform the retrieved content into a more usable format.

You can extend jQuery's Ajax processing at each of these points. Add a prefilter to customize access to remote content, or disable that access completely, as shown in this chapter. Provide an alternate download mechanism for special content with your own transport function, such as for downloading images, or replace or enhance an existing mechanism, as we did with the HTML simulation for testing purposes. Obtain data in the most useful format for the task at hand by integrating a converter into the Ajax process. Together these extension points let you make jQuery's Ajax processing work the way you want it to.

The next chapter looks at jQuery's event handling and examines how you can provide custom events for interested parties to respond to.

WEB DEVELOPMENT

Extending jQuery

Keith Wood

jQuery, the most popular JavaScript library, helps make client-side scripting of HTML easy. It offers many built-in abilities to traverse and alter the DOM, but it can't do everything. Fortunately, you can tap into jQuery's numerous extension points to create your own selectors and filters, plugins, animations, and more. This book shows you how.

Extending jQuery teaches you to build custom extensions to the jQuery library. In it, you'll discover how to write plugins and how to design them for maximum reuse. You'll also learn to write new widgets and effects for the jQuery UI. Along the way, you'll explore extensions in key areas including Ajax, events, animation, and validation.

What's Inside

- Create jQuery UI widgets and effects
- Make extensions available for distribution and reuse
- Build your own libraries

This book assumes intermediate-level knowledge of jQuery and JavaScript. No experience writing plugins or other extensions is required.

Keith Wood has developed over 20 jQuery plugins including the original Datepicker, World Calendar, Countdown, and SVG.

"Delves into just about every facet of extending jQuery's functionality."
—From the Foreword by Dave Methvin, President jQuery Foundation

"A must-read for all serious web developers."
—Ecil Teodoro, IBM

"Finally, a complete resource on the extension of jQuery and its framework."
—Daniele Midi Whitelemon Design Studio

"A well-written and technically excellent guide."
—Brady Kelly Erisia Web Development

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/ExtendingjQuery

ISBN 13: 978-1-617291-03-6
ISBN 10: 1-617291-03-X
54499
9 781617 291036

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]