

Programmer to Programmer™



Professional

Ajax

2nd Edition

Nicholas C. Zakas, Jeremy McPeak, Joe Fawcett



Updates, source code, and Wrox technical support at www.wrox.com

Contents

Chapter 1: What Is Ajax?1
Chapter 2: Ajax Basics17
Chapter 3: Ajax Patterns65
Chapter 4: Ajax Libraries99
Chapter 5: Request Management127
Chapter 6: XML, XPath, and XSLT149
Chapter 7: Syndication with RSS and Atom193
Chapter 8: JSON237
Chapter 9: Comet273
Chapter 10: Maps and Mashups299
Chapter 11: Ajax Debugging Tools337
Chapter 12: Web Site Widgets351
Chapter 13: Ajax Frameworks407
Chapter 14: ASP.NET AJAX Extensions (Atlas)437
Chapter 15: Case Study: FooReader.NET471
Chapter 16: Case Study: AjaxMail509
Appendix A: Licenses for Libraries and Frameworks575
Index583

4

Ajax Libraries

With the popularity of Ajax applications exploding in 2005, developers and companies began looking for ways to streamline the process. As with many common programming practices, Ajax involves a lot of repetitive procedures that can be identified and simplified for common use. It wasn't long before JavaScript developers started introducing libraries to ease the redundant and sometimes quirky behavior of Ajax communication techniques. These libraries sought to break outside of the hidden frame and XHR modalities of communication and introduce their own methods (which typically are just wrappers for already accepted forms of Ajax communication). All of the libraries discussed in this chapter use interfaces that resemble but do not mimic the techniques discussed in Chapter 2. Remember, the goals of such libraries are to free the developer from worrying about cross-browser Ajax issues by hiding the details.

The Yahoo! Connection Manager

In late 2005, Yahoo! introduced its Yahoo! User Interface (YUI) library to the open source community. Available under a BSD-style license at <http://developer.yahoo.com/yui>, the YUI comprises several JavaScript libraries used within the company to aid in the rapid development of web applications such as Yahoo! Mail and Yahoo! Photos. One of these libraries is the Yahoo! Connection Manager.

With Ajax making heavy use of XHR, many developers are looking for ways to equalize the differences between browser implementations. The Yahoo! Connection Manager does this by handling all of the processing behind the scenes, exposing a simple API that frees developers from cross-browser concerns.

Setup

Before beginning, download the YUI library at <http://sourceforge.net/projects/yui>. A single ZIP file contains all of the JavaScript files necessary to use the Connection Manager. For basic Connection Manager usage, you need two required JavaScript files: `YAHOO.js`, which sets up

Chapter 4

the `YAHOO` namespace (this file is used by all Yahoo! JavaScript components), and `connection.js`, which contains the Connection Manager code. The files must be included in this order:

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

With these files included in your page, you are now ready to begin using the Connection Manager.

Basic Requests

The Yahoo! Connection Manager uses a different interface for sending XHR requests than the default one provided by modern browsers. Instead of creating objects, the Connection Manager exposes several static methods to handle requests. The method you'll use most often is `asyncRequest()`, which has the following signature:

```
transaction=YAHOO.util.Connect.asyncRequest(request_type, url, callback, postData);
```

The first argument is the type of HTTP request to make: "GET" or "POST" (these are case-sensitive). The second argument is simply the URL of the request. The third argument is a callback object containing methods to handle the response from the request. The final argument of `asyncRequest()` is the data to post to the server. For POST requests, this value is a string of URL-encoded values to be sent; for GET requests, this value can either be omitted or set to `null`.

When the call is completed, `asyncRequest()` returns a transaction object that can be used to monitor and interact with the currently executing request.

The Callback Object

The most important Connection Manager concept to understand is the role of the callback object. As opposed to having a simple event handler assignment, the callback object allows you to specify a number of options. In its simplest form, the callback object provides two methods: a `success()` method that is called when a valid response has been received and a `failure()` method that is called when an invalid response is received. For example:

```
var oCallback = {
  success: function (oResponse) {
    //handle a successful response
  },
  failure: function (oResponse) {
    //handle an unsuccessful request
  }
};
```

Each of the two methods is passed an object (`oResponse`) containing response information from the server and/or the Connection Manager itself. The response object has the following properties:

- ❑ `argument`: A developer-defined value to be returned with the response
- ❑ `responseText`: The text response from the server

- ❑ `responseXML`: An XML DOM containing XML from the server (if the content type is "text/xml")
- ❑ `status`: The HTTP status code returned from the server or an error code provided by the Connection Manager
- ❑ `statusText`: The HTTP status description or an error message provided by the Connection Manager
- ❑ `tId`: The transaction ID uniquely assigned to this request by the Connection Manager

Additionally, the response object has two methods:

- ❑ `getAllResponseHeaders()`: Returns a string containing all header information
- ❑ `getResponseHeader(name)`: Returns the value of the named HTTP header

Some of these properties and methods are direct carryovers from the XHR object.

The Connection Manager's goal is to free developers from worrying about small details, and part of that is in determining when a response was successfully received and when it was not. If the `status` of the response is between 200 and 300, the `success()` method is called; otherwise, the `failure()` method is called. Unlike using XHR directly, the developer needn't be bothered by checking the `status` property to take an appropriate action. Here's a simple example:

```
var oCallback = {
  success: function (oResponse) {
    alert("Response received successfully.");
  },

  failure: function (oResponse) {
    alert("The request failed.");
  }
};

YAHOO.util.Connect.asyncRequest("GET", "test.php", oCallback);
```

This example sends a GET request to `test.php`, passing in the callback object. When the response is received, either `success()` or `failure()` is called. A POST request can be sent in a similar fashion, just by changing the first argument of the `asyncRequest()` method and appending the post data:

```
var oCallback = {
  success: function (oResponse) {
    alert("Response received successfully.");
  },

  failure: function (oResponse) {
    alert("The request failed.");
  }
};

var sPostData = "title=Professional%20Ajax&authors=Zakas%20McPeak%20Fawcett";
YAHOO.util.Connect.asyncRequest("POST", "test.php", oCallback, sPostData);
```

Chapter 4

Here, the post data is added as a fourth argument. Note that the post data must be URL encoded before being passed to the method. Connection Manager handles setting the appropriate content type for the request, which by default is the HTTP form submission content type of “application/x-www-form-urlencoded”. It’s possible to turn this header off for submission of non-form data by calling `setDefaultPostHeader(false)`:

```
var sPostData = "raw text data";
YAHOO.util.Connect.setDefaultPostHeader(false);
YAHOO.util.Connect.asyncRequest("POST", "test.php", oCallback, sPostData);
```

The callback object isn’t limited to just two methods; a few additional properties are provided for ease of use.

The argument Property

Suppose that there’s some additional information necessary to process either a successful or an unsuccessful HTTP request. Using the techniques described in Chapter 2 would require some additional thought and planning. The Connection Manager makes this easy by using the `argument` property on the callback object. This property can be set to any value or object, and that same value or object is passed into both the `success()` and `failure()` methods as a property on the response object. For example:

```
var oCallback = {
    success: function (oResponse) {

        //retrieve the argument
        var sArg = oResponse.argument;

        alert("Request was successful: " + sArg);

    },
    failure: function (oResponse) {

        //retrieve the argument
        var sArg = oResponse.argument;

        alert("Request failed: " + sArg);

    },
    argument: "string of info"
};
```

In this code, the `argument` property is specified as a string on the callback object. The `success()` and `failure()` methods access the `argument` property on the response object and use it as part of a message displayed to the user.

Note that the `argument` property is client side only, so the server never sees the value.

The scope Property

You may have a case when you want the `success()` and `failure()` methods to call methods on another object. To facilitate this case, the callback object offers the `scope` property.

Suppose you have an object that is responsible for executing server requests.

Further suppose you have an object `oAjaxObject` that has the methods `handleSuccess()` and `handleFailure()` that should be called for `success()` and `failure()`, respectively:

```
var oAjaxObject = {
  name : "Test Object",

  handleSuccess : function (oResponse) {
    alert(this.name + " Response received successfully.");
  },

  handleFailure : function (oResponse) {
    alert(this.name + " An error occurred.");
  }
};
```

One might think of creating a callback object such as this:

```
var oCallback = {
  success: oAjaxObject.handleSuccess,
  failure: oAjaxObject.handleFailure
};
```

This code would work if the methods didn't both reference the `this` object. Since `this` always refers to the scope of the function being called, it would evaluate to `oCallback` if this callback object were used. In order to execute the function in the proper scope, as a method of `oAjaxObject`, add the `scope` property to the callback object:

```
var oCallback = {
  success: oAjaxObject.handleSuccess,
  failure: oAjaxObject.handleFailure,
  scope: oAjaxObject
};
```

The `scope` property says, "run the `success()` and `failure()` functions as methods of this object." Since good object-oriented design requires all functions to be methods of an object, this ends up being a very common case.

The timeout Property

There is one last optional property for a callback object: `timeout`. The `timeout` property specifies how long, in milliseconds, it should take for the response to be received. If the response is not received within that time period, the request is cancelled and the `failure()` method is called. Only if the response is received within the specified time period will the `success()` method be called. For instance, if a request must return within 5 seconds to be considered successful, the following callback object may be used:

```
var oCallback = {
  success: oAjaxObject.handleSuccess,
  failure: oAjaxObject.handleFailure,
  scope: oAjaxObject,
  timeout: 5000
};
```

Chapter 4

If the `failure()` method is called due to a timeout, the `status` property of the response object is set to `-1` and the `statusText` property is set to "transaction aborted."

Though this property is helpful, you can create a race condition using it. Since the Connection Manager uses a timeout to periodically check the condition of requests, a response may have been received but not registered before the timeout expires. For this reason, make sure that the timeout specified is large enough to allow ample time for a response to be received and recognized.

Monitoring and Managing Requests

One of the limitations of XHR is the lack of a built-in method to monitor and manage multiple requests. The Yahoo! Connection Manager has implemented features that allow the monitoring of multiple requests as well as the ability to abort a request that has not yet completed.

As mentioned previously, the `asyncRequest()` method returns an object representing the request transaction. This object can be used to determine if the request is still pending by passing it to the `isCallInProgress()` method, like this:

```
var oTransaction = YAHOO.util.Connect.asyncRequest("GET", "info.htm", oCallback);
alert(YAHOO.util.Connect.isCallInProgress(oTransaction)); //outputs "true"
```

The `isCallInProgress()` method returns `true` if the transaction hasn't completed yet or `false` otherwise.

You might have a case when a request was initiated but should not be allowed to complete. In this case, the Connection Manager provides an `abort()` method. The `abort()` method expects the transaction object to be passed in:

```
var oTransaction = YAHOO.util.Connect.asyncRequest("GET", "info.htm", oCallback);
if(YAHOO.util.Connect.isCallInProgress(oTransaction)) {
    YAHOO.util.Connect.abort(oTransaction);
}
```

Calling `abort()` stops the current request and frees the resources associated with it. Of course, it only makes sense to abort requests that haven't received a response yet, so it's good practice to call `isCallInProgress()` prior to calling `abort()`.

Form Interaction

It is becoming more and more common to submit form values through an Ajax request instead of using the traditional form posting technique. The Yahoo! Connection Manager makes this easy by allowing you to set a form whose data should be sent through the request. For instance, suppose that you have a form with the ID of "frmInfo". A POST request to submit the data contained in the form can be created like this:

```
var oForm = document.getElementById("frmInfo");
YAHOO.util.Connect.setForm(oForm);
YAHOO.util.Connect.asyncRequest("POST", "datahandler.php", oCallback);
```

Using the `setForm()` method, the Connection Manager creates a string of data to be sent in the next request. Because of this, there is no need to specify the fourth argument for the `asyncRequest()` method, since all the data is already retrieved from the form.

It's important to note that the data string to post is constructed when you call `setForm()`, not when `asyncRequest()` is called. The data being sent is the data at the time when `setForm()` was called, so this method should be called only right before a call to `asyncRequest()` to ensure that the data is the most recent available.

File Uploads

Unlike XHR, the Connection Manager allows file uploads through forms. Before using this feature, be sure to include the Yahoo! Event utility:

```
<script type="text/javascript" src="/js/YAHOO.js"></script>
<script type="text/javascript" src="/js/event.js"></script>
<script type="text/javascript" src="/js/connection.js"></script>
```

Next, set the second argument of `setForm()` to `true` to indicate that a file upload needs to occur:

```
var oForm = document.getElementById("frmInfo");
YAHOO.util.Connect.setForm(oForm, true);
YAHOO.util.Connect.asyncRequest("POST", "datahandler.php", oCallback);
```

When supplying this argument, the Connection Manager switches to using an `iframe` to send the request. This means that the URL receiving the POST (`datahandler.php` in the previous example) must output HTML code. Since the transaction takes place in an `iframe`, status codes aren't available for file upload operations, and thus, `success()` and `failure()` can't be used to monitor the status of the request. Instead, define an `upload()` method on the callback object:

```
var oCallback = {
    upload: function (oResponse) {
        alert(oResponse.responseText);
    }
};
```

A response object is passed into `upload()`, just as it is for `success()` and `failure()`. The `responseText` property of the response object is then filled with the text contained within the resulting page's `<body/>` element (the `status` and `statusText` properties are not available when uploading a file). If the text returned in the `iframe` is valid XML code, the `responseXML` property of the response object is filled with an XML document. It is, however, up to you to determine from `responseText` or `responseXML` whether the upload was successful or not.

As with `success()` and `failure()`, you can also use the `scope` and `argument` properties to provide additional information for the `upload()` method.

Chapter 4

As a final note, if you are uploading a file over SSL, set the third argument of `setForm()` to `true`:

```
var oForm = document.getElementById("frmInfo");
YAHOO.util.Connect.setForm(oForm, true, true);
YAHOO.util.Connect.asyncRequest("post", "datahandler.php", oCallback);
```

This is necessary due to an issue in Internet Explorer when unloading the `iframe` used for the transaction, but is good to use regardless of the browsers you intend to support.

GET Example

By revisiting the XHR GET example from Chapter 2, you can see how the Yahoo! Connection Manager simplifies the JavaScript code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Connection Manager GET Example</title>
  <script type="text/javascript" src="yahoo.js"></script>
  <script type="text/javascript" src="connection.js"></script>
  <script type="text/javascript">
    <![CDATA[
      function requestCustomerInfo() {
        var sId = document.getElementById("txtCustomerId").value;
        var oCallback = {
          success: function (oResponse) {
            displayCustomerInfo(oResponse.responseText);
          },
          failure: function (oResponse) {
            displayCustomerInfo("An error occurred: " +
              oResponse.statusText);
          }
        };
        YAHOO.util.Connect.asyncRequest("GET",
          "GetCustomerData.php?id=" + sId, oCallback);
      }

      function displayCustomerInfo(sText) {
        var divCustomerInfo = document.getElementById("divCustomerInfo");
        divCustomerInfo.innerHTML = sText;
      }
    </script>
  </head>
  <body>
    <p>Enter customer ID number to retrieve information:</p>
    <p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>
    <p><input type="button" value="Get Customer Info"
      onclick="requestCustomerInfo()" /></p>
    <div id="divCustomerInfo"></div>
  </body>
</html>
```

Using the same `displayCustomerInfo()` function and updating the `requestCustomerInfo()` function, the example works perfectly. The major difference is that the code doesn't have to check for a failure case; the Connection Manager handles that. Since the response object returns the same information as an XHR object, the success and error messages are handled using the `responseText` and `statusText` properties, respectively, mimicking the original example.

POST Example

When you are using the Connection Manager for POSTing information back to the server, the simplification of the JavaScript code is even more dramatic. Consider the second XHR example from Chapter 2, which involves adding a customer record to a database. In that example, code had to be written to encode the form's values, which included a large function designed specifically for that task. Since the Connection Manager handles that for you, the code becomes very simple:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Connection Manager POST Example</title>
  <script type="text/javascript"src="yahoo.js"></script>
  <script type="text/javascript"src="connection.js"></script>
  <script type="text/javascript">
    //
      function sendRequest() {
        var oForm = document.forms[0];

        var oCallback = {
          success: function (oResponse) {
            saveResult(oResponse.responseText);
          },

          failure: function (oResponse) {
            saveResult("An error occurred: " + oResponse.statusText);
          }
        };

        YAHOO.util.Connect.setForm(oForm);
        YAHOO.util.Connect.asyncRequest("POST", oForm.action, oCallback);
      }

      function saveResult(sMessage) {
        var divStatus = document.getElementById("divStatus");
        divStatus.innerHTML = "Request completed: " + sMessage;
      }
    //]]&gt;
  &lt;/script&gt;
&lt;/head&gt;
&lt;body&gt;
  &lt;form method="post" action="SaveCustomer.php"
    onsubmit="sendRequest(); return false"&gt;
    &lt;p&gt;Enter customer information to be saved:&lt;/p&gt;
    &lt;p&gt;Customer Name: &lt;input type="text" name="txtName" value="" /&gt;&lt;br /&gt;</pre>
</div>
<div data-bbox="903 939 956 960" data-label="Page-Footer">107</div>
```

Chapter 4

```
Address: <input type="text" name="txtAddress" value="" /><br />
City: <input type="text" name="txtCity" value="" /><br />
State: <input type="text" name="txtState" value="" /><br />
Zip Code: <input type="text" name="txtZipCode" value="" /><br />
Phone: <input type="text" name="txtPhone" value="" /><br />
E-mail: <input type="text" name="txtEmail" value="" /></p>
<p><input type="submit" value="Save Customer Info" /></p>
</form>
<div id="divStatus"></div>
</body>
</html>
```

What previously took more than 90 lines of JavaScript code using XHR now takes only 19 lines of code. Most of the savings comes with the use of `setForm()` to encode the form values. When completed, this example behaves exactly the same as its XHR counterpart.

Additional Features

As mentioned previously, the Connection Manager uses a polling mechanism to check the status of request transactions it initiates. If you find that the default polling interval isn't good enough for your needs, the `setPollingInterval()` method can be called to reset the interval as desired:

```
YAHOO.util.Connect.setPollingInterval(250);
```

This method should be called before any requests have been sent, since this new setting takes effect on all requests, both those that are in process and all those that have yet to be initiated.

Another method, `initHeader()`, allows specification of request headers:

```
YAHOO.util.Connect.initHeader("MyName", "Nicholas");
YAHOO.util.Connect.asyncRequest("get", "info.php", oCallback);
```

In this example, an extra header with a name of "MyName" and a value of "Nicholas" is sent to the server. Note that this header is good for only one request; all headers reset to default values after each call to `asyncRequest()`.

Limitations

While the Yahoo! Connection Manager does make some requests easier, it does have its limitations.

- ❑ Currently, only asynchronous requests are supported, so you'll be stuck using old school XHR if you need to make a synchronous request. Though many argue that synchronous requests should never be used, sometimes there are practical reasons for using them.
- ❑ It is also worth noting that as of this writing the current version of the Connection Manager is 0.12.0, so undoubtedly there will be some additions and changes in the future. However, for the time being, it remains one of the most compact libraries for cross-browser Ajax communication.



Programmer to Programmer™

[BROWSE BOOKS](#)

[P2P FORUM](#)

[FREE NEWSLETTER](#)

[ABOUT WROX](#)

Get more Wrox at **Wrox.com!**

Special Deals

Take advantage of special offers every month

Unlimited Access. . .

. . . to over 70 of our books in the Wrox Reference Library. (see more details on-line)

Meet Wrox Authors!

Read running commentaries from authors on their programming experiences and whatever else they want to talk about

Free Chapter Excerpts

Be the first to preview chapters from the latest Wrox publications

Forums, Forums, Forums

Take an active role in online discussions with fellow programmers

Browse Books

[.NET](#)

[SQL Server](#)

[Java](#)

[XML](#)

[Visual Basic](#)

[C#/C++](#)

Join the community!

Sign-up for our free monthly newsletter at
newsletter.wrox.com