How-To: Build a Web Application with Ajax Part 2

Sending a Request

Technical Note: As of this document revision, 2021.10.04.1.25.PM, most if not all modern browsers are quite capable or launching Ajax code.

In the previous portion of this document, we learned of an HTTP request (`XMLHttpRequest`) method that supports sending data from forms to browsers. We can now write a function that uses that method to make a request. We start the doReq method like this:

```
Example 2.3. ajax.js (excerpt)



this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

};
```

This first part of `doReq` calls `init` to create an instance of the `XMLHttpRequest` class, and displays a quick alert if it's not successful.

Setting Up the Request

Next, our code calls the `open` method on `this.req` — our new instance of the `XMLHttpRequest` class — to begin setting up the HTTP request:

```
Example 2.4. ajax.js (excerpt)



this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);

};
```

The `open` method takes three parameters:

1. Method – This parameter identifies the type of HTTP request method we'll use. The most commonly used methods are GET and POST.

Methods are Case-sensitive

According to the HTTP specification (RFC 2616), the names of these request methods are case-sensitive. And since the methods described in the spec are defined as being all uppercase, you should always make sure you type the method in all uppercase letters.

2. URL – This parameter identifies the page being requested (or posted to if the method is POST).

Crossing Domains

Normal browser security settings will not allow you to send HTTP requests to another domain. For example, a page served from ajax.net would not be able to send a request to remotescripting.com unless the user had allowed such requests.

3. Asynchronous Flag – If this parameter is set to `true`, your JavaScript will continue to execute normally while waiting for a response to the request. As the state of the request changes, events are fired so that you can deal with the changing state of the request.

If you set the parameter to `false`, JavaScript execution will stop until the response comes back from the server. This approach has the advantage of being a little simpler than using a callback function, as you can start dealing with the response straight after you send the request in your code, but the big disadvantage is that your code pauses while the request is sent and processed on the server, and the response is received. As the ability to communicate with the server asynchronously is the whole point of an AJAX application, this should be set to `true`.

In our `Ajax` class, the method and async properties are initialized to reasonable defaults (GET and true), but you'll always have to set the target URL, of course.

Setting Up the `onreadystatechange` Event Handler

As the HTTP request is processed on the server, its progress is indicated by changes to the readyState property. This property is an integer that represents one of the following states, listed in order from the start of the request to its finish:

- `0`: uninitialized – `open` has not been called yet.
- `1`: loading – `send` has not been called yet.
- `2`: loaded – `send` has been called, but the response is not yet available.

- `3`: interactive – The response is being downloaded, and the responseText property holds partial data.
- `4`: completed – The response has been loaded and the request is completed.

An `XMLHttpRequest` object tells you about each change in state by firing a `readystatechange` event. In the handler for this event, check the `readyState` of the request, and when the request completes (i.e., when the `readyState` changes to `4`), you can handle the server's response.

A basic outline for our `Ajax` code would look like this:

```
Example 2.5. ajax.js (excerpt)



this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);

  var self = this; // Fix loss-of-scope in inner
function

  this.req.onreadystatechange = function() {
    if (self.req.readyState == 4) {

      // Do stuff to handle response

    }

  };

};
```
We'll discuss how to "do stuff to handle response" in just a bit. For now, just keep in mind that you need to set up this event handler before the request is sent.

Sending the Request

Use the `send` method of the `XMLHttpRequest` class to start the HTTP request, like so:

```
Example 2.6. ajax.js (excerpt)



this.doReq = function() {

  if (!this.init()) {

    alert('Could not create XMLHttpRequest
object.');

    return;

  }

  this.req.open(this.method, this.url,
this.async);
```

```
  var self = this; // Fix loss-of-scope in inner
function

  this.req.onreadystatechange = function() {

    if (self.req.readyState == 4) {

      // Do stuff to handle response

    }

  };

  this.req.send(this.postData);

};
```

The send method takes one parameter, which is used for `POST` data. When the request is a simple `GET` that doesn't pass any data to the server, like our current request, we set this parameter to null.

Loss of Scope and `this`

You may have noticed that `onreadystatechange` includes a weird-looking variable assignment:

```
Example 2.7. ajax.js (excerpt)



var self = this; // Fix loss-of-scope in inner
function
```

This new variable, `self`, is the solution to a problem called "loss of scope" that's often experienced by JavaScript developers using asynchronous event handlers. Asynchronous event handlers are commonly used in conjunction with `XMLHttpRequest`, and with functions like `setTimeout` or `setInterval`.

The `this` keyword is used as shorthand in object-oriented JavaScript code to refer to "the current object." Here's a quick example — a class called `ScopeTest`:

```
function ScopeTest() {

  this.message = 'Greetings from ScopeTest!';

  this.doTest = function() {

    alert(this.message);

  };

}

var test = new ScopeTest();

test.doTest();
```

This code will create an instance of the `ScopeTest` class, then call that object's `doTest` method, which will display the message "Greetings from ScopeTest!" Simple, right?

Now, let's add some simple `XMLHttpRequest` code to our `ScopeTest` class. We'll send a simple `GET` request for your web server's home page, and, when a response is received, we'll display the content of both `this.message` and `self.message`.

```
function ScopeTest() {

  this.message = 'Greetings from ScopeTest!';

  this.doTest = function() {

    // This will only work in Firefox, Opera and
Safari.
```

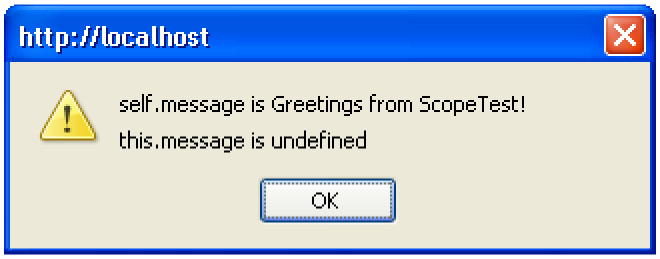
```
    this.req = new XMLHttpRequest();

    this.req.open('GET', '/index.html', true);

    var self = this;

    this.req.onreadystatechange = function() {

      if (self.req.readyState == 4) {

        var result = 'self.message is ' +
self.message;

        result += 'n';

        result += 'this.message is ' +
this.message;

        alert(result);

      }

    }

    this.req.send(null);

  };

}

var test = new ScopeTest();

test.doTest();
```

So, what message is displayed? The answer is revealed in Figure 2.1.

We can see that `self.message` is the greeting message that we're expecting, but what's happened to `this.message`?

Using the keyword `this` is a convenient way to refer to “the object that’s executing this code.” But this has one small problem — its meaning changes when it’s called from outside the object. This is the result of something called execution context. All of the code inside the object runs in the same execution context, but code that’s run from other objects — such as event handlers — runs in the calling object’s execution context. What this means is that, when you’re writing object-oriented JavaScript, you won’t be able to use the `this` keyword to refer to the object in code for event handlers (like `onreadystatechange` above). This problem is called loss of scope.

If this concept isn’t 100% clear to you yet, don’t worry too much about it. We’ll see an actual demonstration of this problem in the next chapter. In the meantime, just kind of keep in mind that if you see the variable self in code examples, it’s been included to deal with a loss-of-scope problem.



```
self.message is Greetings from ScopeTest!
this.message is undefined
```

Figure 2.1. Message displayed by ScopeTest class

Processing the Response

Now we’re ready to write some code to handle the server’s response to our HTTP request. Remember the “do stuff to handle response” comment that we left in the `onreadystatechange` event handler? We’ll, it’s time we wrote some code to do that stuff! The function needs to do three things:

1. Figure out if the response is an error or not.
2. Prepare the response in the desired format.
3. Pass the response to the desired handler function.

Include the code below in the inner function of our `Ajax` class:

```
Example 2.8. ajax.js (excerpt)



this.req.onreadystatechange = function() {

  var resp = null;

  if (self.req.readyState == 4) {

    switch (self.responseFormat) {

      case 'text':

        resp = self.req.responseText;

        break;

      case 'xml':

        resp = self.req.responseXML;

        break;

      case 'object':

        resp = req;

        break;

    }

    if (self.req.status >= 200 && self.req.status
<= 299) {
      self.handleResp(resp);

    }

    else {

      self.handleErr(resp);

    }

  }

};
```
When the response completes, a code indicating whether or not the request succeeded is returned in the status property of our `XMLHttpRequest` object. The status property contains the HTTP status code of the completed request. This could be code 404 if the requested page was missing, 500 if an error occurred in the server-side script, 200 if the request was successful, and so on. A full list of these codes is provided in the HTTP Specification (RFC 2616).

No Good with Numbers?

If you have trouble remembering the codes, don't worry: you can use the statusText property, which contains a short message that tells you a bit more detail about the error (e.g., "Not Found," "Internal Server Error," "OK").

Our `Ajax` class will be able to provide the response from the server in three different formats: as a normal JavaScript string, as an XML document object accessible via the W3C XML DOM, and as the actual `XMLHttpRequest` object that was used to make the request. These are controlled by the `Ajax` class's `responseFormat` property, which can be set to `text`, `xml` or `object`.

The content of the response can be accessed via two properties of our `XMLHttpRequest` object:

- `responseText` – This property contains the response from the server as a normal string. In the case of an error, it will contain the web server's error page HTML. As long as a response is returned (that is, `readyState` becomes 4), this property will contain data, though it may not be what you expect.
- `responseXML` – This property contains an XML document object. If the response is not XML, this property will be empty.

Our `Ajax` class initializes its `responseFormat` property to text, so by default, your response handler will be passed the content from the server as a JavaScript string. If you're working with XML content, you can change the `responseFormat` property to `xml`, which will pull out the XML document object instead.

There's one more option you can use if you want to get really fancy: you can return the actual `XMLHttpRequest` object itself to your handler function. This gives you direct access to things like the status and `statusText` properties, and might be useful in cases in which you want to treat particular classes of errors differently — for example, completing extra logging in the case of 404 errors.

Setting the Correct `Content-Type`

Implementations of `XMLHttpRequest` in all major browsers require the HTTP response's `Content-Type` to be set properly in order for the response to be handled as XML. Well-formed XML, returned with a content type of `text/xml` (or `application/xml`, or even `application/xhtml+xml`), will properly populate the `responseXML` property of an `XMLHttpRequest` object; non-XML content types will result in values of `null` or `undefined` for that property.

However, Firefox, Safari, and Internet Explorer 7 provide a way around `XMLHttpRequest`'s pickiness over XML documents: the `overrideMimeType` method of the `XMLHttpRequest` class. Our simple `Ajax` class hooks into this with the `setMimeType` method:

```
Example 2.9. ajax.js (excerpt)



this.setMimeType = function(mimeType) {

  this.mimeType = mimeType;

};
```

This method sets the `mimeType` property.

Then, in our `doReq` method, we simply call `overrideMimeType` inside a `try ... catch` block, like so:

```
Example 2.10. ajax.js (excerpt)



req.open(this.method, this.url, this.async);

if (this.mimeType) {

  try {

    req.overrideMimeType(this.mimeType);

  }

  catch (e) {

    // couldn't override MIME type  --  IE6 or
Opera?

  }

}
```

```
var self = this; // Fix loss-of-scope in inner
function
```

Being able to override `Content-Type` headers from uncooperative servers can be very important in environments in which you don't have control over both the front and back ends of your web application. This is especially true since many of today's apps access services and content from a lot of disparate domains or sources. However, as this technique won't work in Internet Explorer 6 or Opera 8, you may not find it suitable for use in your applications today.

Response Handler

According to the HTTP 1.1 specification, any response that has a code between 200 and 299 inclusive is a successful response.

The `onreadystatechange` event handler we've defined looks at the status property to get the status of the response. If the code is within the correct range for a successful response, the `onreadystatechange` event handler passes the response to the response handler method (which is set by the `handleResp` property).

The response handler will need to know what the response was, of course, so we'll pass it the response as a parameter. We'll see this process in action later, when we talk about the doGet method.

Since the handler method is user-defined, the code also does a cursory check to make sure the method has been set properly before it tries to execute the method.

Error Handler

If the status property indicates that there's an error with the request (i.e., it's outside the 200 to 299 code range), the server's response is passed to the error handler in the handleErr property. Our Ajax class already defines a reasonable default for the error handler, so we don't have to make sure it's defined before we call it.

The `handleErr` property points to a function that looks like this:

```
Example 2.11. ajax.js (excerpt)



this.handleErr = function() {

  var errorWin;

  try {

    errorWin = window.open('', 'errorWin');

    errorWin.document.body.innerHTML =
this.responseText;

  }

  catch (e) {

    alert('An error occurred, but the error message
cannot be '

      + 'displayed. This is probably because of
your browser's '

      + 'pop-up blocker.n'

      + 'Please allow pop-ups from this web site if
you want to '

      + 'see the full error messages.n'

      + 'n'

      + 'Status Code: ' + this.req.status + 'n'
      + 'Status Description: ' +
this.req.statusText);

  }

};
```

This method checks to make sure that pop-ups are not blocked, then tries to display the full text of the server's error page content in a new browser window. This code uses a `try ... catch` block, so if users have blocked pop-ups, we can show them a cut-down version of the error message and tell them how to access a more detailed error message.

This is a decent default for starters, although you may want to show less information to the end-user — it all depends on your level of paranoia. If you want to use your own custom error handler, you can use `setHandlerErr` like so:

```
Example 2.12. ajax.js (excerpt)



this.setHandlerErr = function(funcRef) {

  this.handleErr = funcRef;

}
```

Or, the One True Handler

It's possible that you might want to use a single function to handle both successful responses and errors. `setHandlerBoth`, a convenience method in our `Ajax` class, sets this up easily for us:

```
Example 2.13. ajax.js (excerpt)
```

```
this.setHandlerBoth = function(funcRef) {

  this.handleResp = funcRef;

  this.handleErr = funcRef;

};
```

Any function that's passed as a parameter to `setHandlerBoth` will handle both successful responses and errors.

This setup might be useful to a user who sets your class's `responseFormat` property to object, which would cause the `XMLHttpRequest` object that's used to make the request — rather than just the value of the `responseText` or `responseXML` properties — to be passed to the response handler.

Aborting the Request

Sometimes, as you'll know from your own experience, a web page will take a very long time to load. Your web browser has a Stop button, but what about your `Ajax` class? This is where the `abort` method comes into play:

```
Example 2.14. ajax.js (excerpt)



this.abort = function() {

  if (this.req) {

    this.req.onreadystatechange = function() { };

    this.req.abort();
    this.req = null;

  }

};
```
This method changes the `onreadystate` event handler to an empty function, calls the `abort` method on your instance of the `XMLHttpRequest` class, then destroys the instance you've created. That way, any properties that have been set exclusively for the request that's being aborted are reset. Next time a request is made, the `init` method will be called and those properties will be reinitialized.

So, why do we need to change the `onreadystate` event handler? Many implementations of `XMLHttpRequest` will fire the onreadystate event once abort is called, to indicate that the request's state has been changed. What's worse is that those events come complete with a `readyState` of 4, which indicates that everything completed as expected (which is partly true, if you think about it: as soon as we call abort, everything should come to a stop and our instance of `XMLHttpRequest` should be ready to send another request, should we so desire). Obviously, we don't want our response handler to be invoked when we abort a request, so we remove the existing handler just before we call `abort`.

Wrapping it Up

Given the code we have so far, the Ajax class needs just two things in order to make a request:

- a target URL
- a handler function for the response

Let's provide a method called `doGet` to set both of these properties, and kick off the request:

```
Example 2.15. ajax.js (excerpt)



this.doGet = function(url, hand, format) {

  this.url = url;

  this.handleResp = hand;

  this.responseFormat = format || 'text';

  this.doReq();

};
```
You'll notice that, along with the two expected parameters, `url` and `hand`, the function has a third parameter: `format`. This is an optional parameter that allows us to change the format of the server response that's passed to the handler function.

If we don't pass in a value for format, the `responseFormat` property of the `Ajax` class will default to a value of text, which means your handler will be passed the value of the `responseText` property. You could, instead, pass `xml` or `object` as the format, which would change the parameter that's being passed to the response handler to an XML DOM or `XMLHttpRequest` object.

Example: a Simple Test Page

It's finally time to put everything we've learned together! Let's create an instance of this `Ajax` class, and use it to send a request and handle a response.

Now that our class's code is in a file called `ajax.js`, any web pages in which we want to use our `Ajax` class will need to include the Ajax code with a `<script type="text/javascript"`

`src="ajax.js">` tag. Once our page has access to the Ajax code, we can create an `Ajax` object.

```
Example 2.16. ajaxtest.html (excerpt)



<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"

    "https://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">

<html xmlns="https://www.w3.org/1999/xhtml">

  <head>

    <meta http-equiv="Content-Type"

        content="text/html; charset=iso-8859-1"
/>

    <title>A Simple AJAX Test</title>

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

    <script type="text/javascript">

      var ajax = new Ajax();

    </script>

  </head>

  <body>
  </body>

</html>
```

This script gives us a shiny, new instance of the `Ajax` class. Now, let's make it do something useful.

To make the most basic request with our `Ajax` class, we could do something like this:

```
Example 2.17. ajaxtest.html (excerpt)



<script type="text/javascript">

  var hand = function(str) {

    alert(str);

  }

  var ajax = new Ajax();

  ajax.doGet('/fakeserver.php', hand);

</script>
```

This creates an instance of our `Ajax` class that will make a simple `GET` request to a page called `fakeserver.php`, and pass the result back as text to the hand function. If `fakeserver.php` returned an XML document that you wanted to use, you could do so like this:

```
Example 2.18. ajaxtest.html (excerpt)
```

```
<script type="text/javascript">

  var hand = function(str) {

    // Do XML stuff here

  }

  var ajax = new Ajax();

  ajax.doGet('/fakeserver.php', hand);

</script>
```

You would want to make absolutely sure in this case that somepage.php was really serving valid XML and that its `Content-Type` HTTP response header was set to `text/xml` (or something else that was appropriate).

Continue with Part 3: Create an Ajax Page….

Courtesy: https://www.sitepoint.com/build-your-own-ajax-web-apps/

Modified: 2021.10.04.7.10.AM

Dököll Solutions,. Inc.