

**Javascript** (<https://www.sitepoint.com/javascript/>) > August 21, 2015

> By [Camilo Reyes](https://www.sitepoint.com/author/creyes/) (<https://www.sitepoint.com/author/creyes/>).

## A Guide to Vanilla Ajax Without jQuery

Short for Asynchronous JavaScript and XML, **Ajax** ([https://en.wikipedia.org/wiki/Ajax\\_%28programming%29](https://en.wikipedia.org/wiki/Ajax_%28programming%29)), is a mechanism for making partial page updates. It enables you to update sections of a page with data that comes from the server, whilst avoiding the need for a full refresh. Making partial updates in this way can be effective in creating fluid user experiences and can decrease the load put on the server.

This is the anatomy of a basic Ajax request:

```
var xhr = new XMLHttpRequest();  
xhr.open('GET', 'send-ajax-data.php');  
xhr.send(null);
```

Here, we are creating an instance of the required class to make an HTTP request to the server. We are then calling its **open** method, specifying the HTTP request method as the first parameter and the URL of the page we're requesting as the second. Finally, we call its **send** method passing null as a parameter. If POST-ing the request (here we are using GET), this parameter should contain any data we want to send with the request.

And this is how we'd deal with the response from the server:

(<https://www.sitepoint.com/>).

```
xhr.onreadystatechange = function () {  
  var DONE = 4; // readyState 4 means the request is done.  
  var OK = 200; // status 200 is a successful return.  
  if (xhr.readyState === DONE) {  
    if (xhr.status === OK)  
      console.log(xhr.responseText); // 'This is the returned text.'  
    } else {  
      console.log('Error: ' + xhr.status); // An error occurred during the  
request.  
    }  
  }  
};
```

The **onreadystatechange** is asynchronous, which means it gets called at any time. These types of functions are callbacks – one that gets called once some processing finishes. In this case, the processing is happening on the server.

For those wishing to learn more about the basics of Ajax, the MDN network has [a good guide](https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started) ([https://developer.mozilla.org/en-US/docs/AJAX/Getting\\_Started](https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started)).

## To jQuery or Not to jQuery?

So, the good news is that the above code will work across all the latest major browsers. The bad news is, well, that it is quite convoluted. Yuck! I am already pining for an elegant solution.

Using jQuery, one could condense the entire snippet to:

```
(https://www.sitepoint.com/).  
$.ajax({  
  url: 'send-ajax-data.php',  
})  
.done(function(res) {  
  console.log(res);  
})  
.fail(function(err) {  
  console.log('Error: ' + err.status);  
});
```

Which is nice. And indeed for many, including yours truly, jQuery has become the de facto standard when it comes to Ajax. But, do you know what? This doesn't have to be the case. jQuery exists to get around the ugly DOM API. But, is it really *that* ugly? Or incomprehensible?

In the remainder of this article, I would like to investigate improvements made to the Ajax API in vanilla JavaScript. The entire specification can be found on the W3C (<http://www.w3.org/TR/XMLHttpRequest/>). What strikes me about this specification is the name. It is no longer “XMLHttpRequest Level 2” but “XMLHttpRequest Level 1” — a result of a 2011 merger between the two specs. Going forward, it will get treated as a single entity from a standards perspective and the living standard will be called XMLHttpRequest (<https://xhr.spec.whatwg.org/>). This shows that there is commitment by the community to stick to one standard, and this can only mean good news for developers who want to break free from jQuery.

So lets get started ...

## Setup

For this article, I am using Node.js (<https://nodejs.org/>), on the back-end. Yes, there will be JavaScript on the browser and on the server. The Node.js back-end is lean, I encourage you to download the entire demo on GitHub (<https://github.com/sitepoint-editors/VanillaAjaxNojQuery>), and follow along. Here is the meat and potatoes of what's on the server:

(<https://www.sitepoint.com/>).

```
// app.js
var app = http.createServer(function (req, res) {
  if (req.url.indexOf('/scripts/') >= 0) {
    render(req.url.slice(1), 'application/javascript', httpHandler);
  } else if (req.headers['x-requested-with'] === 'XMLHttpRequest') {
    // Send Ajax response
  } else {
    render('views/index.html', 'text/html', httpHandler);
  }
});
```

This checks the request URL to determine how the app should respond. If the request came from the **scripts** directory, then the appropriate file is served with the content type of **application/javascript**. Otherwise, if the request's **x-requested-with** headers have been set to **XMLHttpRequest** then we know we're dealing with an Ajax request and we can respond appropriately. And if neither of these is the case, the file **views/index.html** is served.

I will expand the commented out section as we dive into Ajax responses from the server. In Node.js, I had to do some heavy-lifting with the **render** and **httpHandler**:

```
// app.js
function render(path, contentType, fn) {
  fs.readFile(__dirname + '/' + path, 'utf-8', function (err, str) {
    fn(err, str, contentType);
  });
}
var httpHandler = function (err, str, contentType) {
  if (err) {
    res.writeHead(500, {'Content-Type': 'text/plain'});
    res.end('An error has occurred: ' + err.message);
  } else {
    res.writeHead(200, {'Content-Type': contentType});
    res.end(str);
  }
};
```

The `render` function asynchronously reads the contents of the requested file. It is passed a reference to the `httpHandler` function, which it then executes as a callback. The `httpHandler` function checks for the presence of an error object (which would be present, for example, if the file requested could not be opened). Providing everything is good, it then serves the contents of the file with the appropriate HTTP status code and content type.

## Testing the API

Like with any sound back-end API, let's write a few unit tests to make sure it works. For these tests, I am calling on [supertest](https://www.npmjs.com/package/supertest) (<https://www.npmjs.com/package/supertest>) and [mocha](https://www.npmjs.com/package/mocha) (<https://www.npmjs.com/package/mocha>) for help:

```
// test/app.request.js
it('responds with html', function (done) {
  request(app)
    .get('/')
    .expect('Content-Type', /html/)
    .expect(200, done);
});
it('responds with javascript', function (done) {
  request(app)
    .get('/scripts/index.js')
    .expect('Content-Type', /javascript/)
    .expect(200, done);
});
it('responds with json', function (done) {
  request(app)
    .get('/')
    .set('X-Requested-With', 'XMLHttpRequest')
    .expect('Content-Type', /json/)
    .expect(200, done);
});
```

(<https://www.sitepoint.com/>)

These ensure that our app responds with the correct content type and HTTP status code to different requests. Once you have installed the dependencies, you can run these tests from the command using `npm test`.

## The Interface

Now, let's take a look at the user interface we are building in HTML:

```
// views/index.html
<h1>Vanilla Ajax without jQuery</h1>
<button id="retrieve" data-url="/">Retrieve</button>
<p id="results"></p>
```

The HTML looks nice and neat. As you can see, all the excitement is happening in JavaScript.

## onreadystate vs onload

If you go through any canonical Ajax book, you may find **onreadystate** everywhere. This callback function comes complete with nested ifs and lots of fluff that makes it difficult to remember off the top of your head. Let's put the **onreadystate** and **onload** events head to head.

(<https://www.sitepoint.com/>).

```
(function () {  
    var retrieve = document.getElementById('retrieve'),  
        results = document.getElementById('results'),  
        toReadyStateDescription = function (state) {  
            switch (state) {  
                case 0:  
                    return 'UNSENT';  
                case 1:  
                    return 'OPENED';  
                case 2:  
                    return 'HEADERS_RECEIVED';  
                case 3:  
                    return 'LOADING';  
                case 4:  
                    return 'DONE';  
                default:  
                    return '';  
            }  
        };  
    retrieve.addEventListener('click', function (e) {  
        var oReq = new XMLHttpRequest();  
        oReq.onload = function () {  
            console.log('Inside the onload event');  
        };  
        oReq.onreadystatechange = function () {  
            console.log('Inside the onreadystatechange event with readyState: ' +  
                toReadyStateDescription(oReq.readyState));  
        };  
        oReq.open('GET', e.target.dataset.url, true);  
        oReq.send();  
    });  
})();
```

This is the output in the console:

(<https://www.sitepoint.com/>)

File: localhost:1337

Inside the onreadystatechange event with readyState: OPENED

Inside the onreadystatechange event with readyState: OPENED

Inside the onreadystatechange event with readyState: HEADERS\_RECEIVED

Inside the onreadystatechange event with readyState: LOADING

Inside the onreadystatechange event with readyState: DONE

Inside the onload event

The **onreadystatechange** event fires all over the place. It fires at the beginning of each request, at the end, and sometimes just because it really likes getting fired. But according to the spec, the **onload** event fires only when the request *succeeds*. So, the **onload** event is a modern API you can put to good use in seconds. The **onreadystatechange** event is there to be backwards compatible. But, the **onload** event should be your tool of choice. The **onload** event looks like the **success** callback on jQuery, does it not?

It's time to set the 5 lb dumbbells aside and move on to arm curls.

## Setting Request Headers

jQuery sets request headers under the covers so your back-end technology knows it is an Ajax request. In general, the back-end doesn't care where the GET request comes from as long as it sends the proper response. This comes in handy when you want to support Ajax and HTML with the same web API. So, let's look at how to set request headers in vanilla Ajax:

```
var oReq = new XMLHttpRequest();
oReq.open('GET', e.target.dataset.url, true);
oReq.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
oReq.send();
```

With this, we can do a check in Node.js:



(<https://www.sitepoint.com/>).

```
if (req.headers['x-requested-with'] === 'XMLHttpRequest') {  
  res.writeHead(200, {'Content-Type': 'application/json'});  
  res.end(JSON.stringify({message: 'Hello World!'}));  
}
```

As you can see, vanilla Ajax is a flexible and modern front-end API. There are a ton of ideas you can use request headers for, and one of them is versioning. So for example, let's say I want to support more than one version of this web API. This is useful for when I don't want to break URLs and instead provide a mechanism in which clients can choose the version they want. We can set the request header like so:

```
oReq.setRequestHeader('x-vanillaAjaxWithoutjQuery-version', '1.0');
```

And in the back-end, try:

```
if (req.headers['x-requested-with'] === 'XMLHttpRequest' &&  
    req.headers['x-vanillaajaxwithoutjquery-version'] === '1.0') {  
  // Send Ajax response  
}
```

Node.js gives you a **headers** object you can use to check for request headers. The only trick is it reads them in lowercase.

We are at the home stretch and haven't broken a sweat! You may be wondering, what else is there to know about Ajax? Well how about a couple of neat tricks.

## Response Types

You may be wondering why **responseText** contains the server response when all I'm working with is plain old JSON. Turns out, it is because I did not set the proper **responseType**. This Ajax attribute is great for telling the front-end API what type of response to expect from the server. So, let's put this to good use:

```
(https://www.sitepoint.com/).  
var oReq = new XMLHttpRequest();  
oReq.onload = function (e) {  
    results.innerHTML = e.target.response.message;  
};  
oReq.open('GET', e.target.dataset.url, true);  
oReq.responseType = 'json';  
oReq.send();
```

Awesome, instead of sending back plain text that I then have to parse into JSON, I can tell the API what to expect. This feature is available in almost all the latest major browsers. jQuery, of course, does this type of conversion automatically. But isn't it great that we now have a convenient way of doing the same in plain JavaScript? Vanilla Ajax has support for many other response types, including XML.

Sadly, in Internet Explorer the story is not quite as awesome. As of IE 11, the team has yet to add support for `xhr.responseType = 'json'` (<https://connect.microsoft.com/IE/feedback/details/794808>). This feature is to arrive on Microsoft Edge (<http://caniuse.com/#feat=xhr2>). But, the bug has been outstanding for almost two years as of the time of writing. My guess is the folks at Microsoft have been hard at work revamping the browser. Let's hope Microsoft Edge, aka Project Spartan, delivers on its promises.

Alas, if you must get around this IE issue:

```
oReq.onload = function (e) {  
    var xhr = e.target;  
    if (xhr.responseType === 'json') {  
        results.innerHTML = xhr.response.message;  
    } else {  
        results.innerHTML = JSON.parse(xhr.responseText).message;  
    }  
};
```

## Cache Busting

(<https://www.sitepoint.com/>)

One browser feature people tend to forget is the capability of caching Ajax requests. Internet Explorer, for example, does this by default. I once struggled for hours trying to figure why my Ajax wasn't working because of this. Luckily, jQuery busts the browser cache by default. Well, you can too in plain Ajax and it is pretty straightforward:

```
var bustCache = '?' + new Date().getTime();
oReq.open('GET', e.target.dataset.url + bustCache, true);
```

Per the [jQuery documentation \(http://api.jquery.com/jquery.ajax/#jQuery-ajax-settings\)](http://api.jquery.com/jquery.ajax/#jQuery-ajax-settings), all it does is append a timestamp query string to the end of the request. This makes the request somewhat unique and busts the browser cache. You can see what this looks like when you fire HTTP Ajax requests:

Result	Protocol	Host	URL
200	HTTP	localhost:1337	/?1433557462205
200	HTTP	localhost:1337	/?1433557462893
200	HTTP	localhost:1337	/?1433557463461
200	HTTP	localhost:1337	/?1433557463989
200	HTTP	localhost:1337	/?1433557464508
200	HTTP	localhost:1337	/?1433557465013
200	HTTP	localhost:1337	/?1433557465492

Tada! All with no drama.

## Conclusion

I hope you've enjoyed the 300lb bench press vanilla Ajax used to be. One upon a time, Ajax was a dreadful beast, but no more. In fact, we've covered all the basics of Ajax without the crutches, ahem shackles, of jQuery.

I'll leave you with a succinct way of making Ajax calls:

```
(https://www.sitepoint.com/).  
var oReq = new XMLHttpRequest();  
oReq.onload = function (e) {  
    results.innerHTML = e.target.response.message;  
};  
oReq.open('GET', e.target.dataset.url + '?' + new Date().getTime(), true);  
oReq.responseType = 'json';  
oReq.send();
```

And this is what the response looks like:






Don't forget, you can find the entire demo up on [GitHub](https://github.com/sitepoint-editors/VanillaAjaxNojQuery) (<https://github.com/sitepoint-editors/VanillaAjaxNojQuery>). I'd welcome hearing your thoughts Ajax with and without jQuery in the comments.



Meet the author

(h  
t

**Camilo Reyes** (<https://www.sitepoint.com/author/creyes/>),   
(<https://plus.google.com/117305015265276164398/about>), **in**  
(<https://www.linkedin.com/in/camilo-reyes-743b2069/>),   
(<https://codepen.io/beautifulcoder>),  (<https://github.com/beautifulcoder>).

Husband, father, and software engineer living in Houston Texas. Passionate about JavaScript and cyber-ing all the things.

## Stuff We Do (<https://www.sitepoint.com/>)

- [Premium \(/premium/\)](/premium/)
- [Versioning \(/versioning/\)](/versioning/)
- [Forums \(/community/\)](/community/)
- [References \(/html-css/css/\)](/html-css/css/)

## About

- [Our Story \(/about-us/\)](/about-us/)
- [Press Room \(/press/\)](/press/)

## Contact

- [Contact Us \(/contact-us/\)](/contact-us/)
- [FAQ \(https://sitepoint.zendesk.com/hc/en-us\)](https://sitepoint.zendesk.com/hc/en-us)
- [Write for Us \(/write-for-us/\)](/write-for-us/)
- [Advertise \(/advertise/\)](/advertise/)

## Legals

- [Terms of Use \(/legals/\)](/legals/)
- [Privacy Policy \(/legals/#privacy\)](/legals/#privacy)

## Connect



[\\_ \(https://www.facebook.com/sitepoint\)](https://www.facebook.com/sitepoint)



[\\_ \(http://twitter.com/sitepointdotcom\)](http://twitter.com/sitepointdotcom)



[\\_ \(/versioning/\)](/versioning/)



[\\_ \(https://www.sitepoint.com/feed/\)](https://www.sitepoint.com/feed/)



[\\_ \(https://plus.google.com/+sitepoint\)](https://plus.google.com/+sitepoint)

© 2000 – 2018 SitePoint Pty. Ltd.