https://developer.mozilla.org/en/AJAX/Getting_Started

See also https://developer.mozilla.org/en/AJAX#Examples

Getting Started

- Edit
- Watch

Table of contents

Page Notifications Off

This article guides you through the AJAX basics and gives you two simple hands-on examples to get you started.

What's AJAX?

AJAX stands for Asynchronous JavaScript and XML. In a nutshell, it is the use of the `XMLHttpRequest` object to communicate with server-side scripts. It can send as well as receive information in a variety of formats, including JSON, XML, HTML, and even text files. AJAX's most appealing characteristic, however, is its "asynchronous" nature, which means it can do all of this without having to refresh the page. This lets you update portions of a page based upon user events.

The two features in question are that you can:

- Make requests to the server without reloading the page
- Receive and work with data from the server

Step 1 – How to make an HTTP request

In order to make an HTTP request to the server using JavaScript, you need an instance of a class that provides this functionality. Such a class was originally introduced in Internet Explorer as an ActiveX object, called `XMLHTTP`. Then Mozilla, Safari, and other browsers followed, implementing an `XMLHttpRequest` class that supports the methods and properties of Microsoft's original ActiveX object.

As a result, in order to create a cross-browser instance (object) of the required class, you can do the following:

view plainprint?

```
1. var httpRequest;
```

```
2. if (window.XMLHttpRequest) { // Mozilla, Safari, ...
3.     httpRequest = new XMLHttpRequest();
4. } else if (window.ActiveXObject) { // IE 8 and older
5.     httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
6. }
```

Note: For illustration purposes, the above is a somewhat simplified version of the code to be used for creating an XMLHTTP instance. For a more real-life example, see step 3 of this article.

Next, you need to decide what you want to do after you receive the server response to your request. At this stage, you just need to tell the HTTP request object which JavaScript function will handle processing the response. This is done by setting the `onreadystatechange` property of the object to the name of the JavaScript function that should be called when the state of the request changes, like this:

```
httpRequest.onreadystatechange = nameOfTheFunction;
```

Note that there are no parentheses after the function name and no parameters passed, because you're simply assigning a reference to the function, rather than actually calling it. Also, instead of giving a function name, you can use the JavaScript technique of defining functions on the fly (called "anonymous functions") and define the actions that will process the response right away, like this:

view plainprint?

```
1. httpRequest.onreadystatechange = function(){
2.     // process the server response
3. };
```

Next, after you've declared what will happen as soon as you receive the response, you need to actually make the request. You need to call the `open()` and `send()` methods of the HTTP request class, like this:

view plainprint?

```
1. httpRequest.open('GET', 'http://www.example.org/some.file', true);
2. httpRequest.send(null);
```

- The first parameter of the call to `open()` is the HTTP request method – GET, POST, HEAD or any other method you want to use and that is supported by your server. Keep the method capitalized as per the HTTP standard; otherwise some browsers (like Firefox) might not process the request. For more information on the possible HTTP request methods you can check the W3C specs.
- The second parameter is the URL of the page you're requesting. As a security feature, you cannot call pages on 3rd-party domains. Be sure to use the exact domain name on all of your pages or you will get a "permission denied" error when you call `open()`. A common pitfall is accessing your site by `domain.tld`, but attempting to call pages with `www.domain.tld`. If you really need to send a request to another domain, see HTTP access control.
- The optional third parameter sets whether the request is asynchronous. If `TRUE` (the default), the execution of the JavaScript function will continue while the response of the server has not yet arrived. This is the A in AJAX.

The parameter to the `send()` method can be any data you want to send to the server if `POST`-ing the request. Form data should be sent in a format that the server can parse easily. This can be as a query string, like:

```
"name=value&anothername="+encodeURIComponent(myVar)+"&so=on"
```

or in several other formats, including JSON, SOAP, etc.

Note that if you want to `POST` data, you may have to set the MIME type of the request. For example, use the following line before calling `send()` for form data sent as a query string:

view plainprint?

```
1. httpRequest.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

Step 2 – Handling the server response

Remember that when you were sending the request, you provided the name of a JavaScript function that is designed to handle the response.

view plainprint?

```
1. httpRequest.onreadystatechange = nameOfTheFunction;
```

Let's see what this function should do. First, the function needs to check for the state of the request. If the state has the value of 4, that means that the full server response has been received and it's OK for you to continue processing it.

view plainprint?

```
1. if (httpRequest.readyState === 4) {
2.     // everything is good, the response is received
3. } else {
4.     // still not ready
5. }
```

The full list of the `readyState` values is as follows:

- 0 (uninitialized)
- 1 (loading)
- 2 (loaded)
- 3 (interactive)
- 4 (complete)

(Source)

The next thing to check is the response code of the HTTP server response. All the possible codes are listed on the W3C site. In the following example, we differentiate between a successful or unsuccessful AJAX call by checking for a `200 OK` response code.

view plainprint?

```
1. if (httpRequest.status === 200) {
2.     // perfect!
3. } else {
4.     // there was a problem with the request,
5.     // for example the response may contain a 404 (Not Found)
6.     // or 500 (Internal Server Error) response code
7. }
```

Now after you've checked the state of the request and the HTTP status code of the response, it's up to you to do whatever you want with the data the server has sent to you. You have two options to access that data:

- `httpRequest.responseText` – returns the server response as a string of text
- `httpRequest.responseXML` – returns the response as an `XMLDocument` object you

can traverse using the JavaScript DOM functions

Note that the steps above are only valid if you used an asynchronous request (third parameter of `open()` was set to `true`). If you used an **synchronous** request you don't need to specify a function, you can access the data return by the server right after calling `send()`, because the script will stop and wait for the server answer.

Step 3 – A Simple Example

Let's put it all together and do a simple HTTP request. Our JavaScript will request an HTML document, `test.html`, which contains the text "I'm a test." and then we'll `alert()` the contents of the `test.html` file.

view plainprint?

```
<span id="ajaxButton" style="cursor: pointer; text-decoration: underline">
  Make a request
</span>
<script type="text/javascript">
(function() {
  var httpRequest;
  document.getElementById("ajaxButton").onclick = function() { makeRequest('test.html'); };

  function makeRequest(url) {
    if (window.XMLHttpRequest) { // Mozilla, Safari, ...
      httpRequest = new XMLHttpRequest();
    } else if (window.ActiveXObject) { // IE
      try {
        httpRequest = new ActiveXObject("Msxml2.XMLHTTP");
      }
      catch (e) {
        try {
          httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
        }
        catch (e) {}
      }
    }

    if (!httpRequest) {
      alert('Giving up :( Cannot create an XMLHTTP instance');
      return false;
    }
    httpRequest.onreadystatechange = alertContents;
    httpRequest.open('GET', url);
    httpRequest.send();
  }

  function alertContents() {
    if (httpRequest.readyState === 4) {
```

```
    if (httpRequest.status === 200) {
      alert(httpRequest.responseText);
    } else {
      alert('There was a problem with the request.');
    }
  }
}
)();
</script>
```

In this example:

- The user clicks the link "Make a request" in the browser;
- The event handler calls the `makeRequest()` function with a parameter – the name `test.html` of an HTML file in the same directory;
- The request is made and then (`onreadystatechange`) the execution is passed to `alertContents();`
- `alertContents()` checks if the response was received and it's an OK and then `alert()`s the contents of the `test.html` file.

Note: If you're sending a request to a piece of code that will return XML, rather than to a static XML file, you must set some response headers if your page is to work in Internet Explorer in addition to Mozilla. If you do not set header `Content-Type: application/xml`, IE will throw a JavaScript error, "Object Expected", after the line where you try to access an XML element.
Note 2: If you do not set header `Cache-Control: no-cache` the browser will cache the response and never re-submit the request, making debugging "challenging." You can also append an always-diferent aditional GET parameter, like the timestamp or a random number (see bypassing the cache)
Note 3: If the `httpRequest` variable is used globally, competing functions calling `makeRequest()` may overwrite each other, causing a race condition. Declaring the `httpRequest` variable local to a closure containing the AJAX functions prevents the race condition.
Note 4: In the event of a communication error (such as the webserver going down), an exception will be thrown in the `onreadystatechange` method when attempting to access the `status` field. Make sure that you wrap your if...then statement in a try...catch. (See: bug 238559).
view plainprint?

```
function alertContents(httpRequest) {
  try {
    if (httpRequest.readyState === 4) {
      if (httpRequest.status === 200) {
        alert(httpRequest.responseText);
      } else {
        alert('There was a problem with the request.');
      }
    }
  }
  catch( e ) {
    alert('Caught Exception: ' + e.description);
```

```
13.  }
14.}
```

Step 4 – Working with the XML response

In the previous example, after the response to the HTTP request was received we used the `responseText` property of the request object, which contained the contents of the `test.html` file. Now let's try the `responseXML` property.

First off, let's create a valid XML document that we'll request later on. The document (`test.xml`) contains the following:

view plainprint?

```
1. <?xml version="1.0" ?>
2. <root>
3.     I'm a test.
4. </root>
```

In the script we only need to change the request line to:

view plainprint?

```
1. ...
2. onclick="makeRequest('test.xml')">
3. ...
```

Then in `alertContents()`, we need to replace the line `alert(httpRequest.responseText);` with:

view plainprint?

```
1. var xmldoc = httpRequest.responseXML;
2. var root_node = xmldoc.getElementsByTagName('root').item(0);
3. alert(root_node.firstChild.data);
```

This code takes the `XMLDocument` object given by `responseXML` and uses DOM methods to access some of the data contained in the XML document. You can see the `test.xml` here and the updated test script here.

Step 5 – Working with data

Finally, let's send some data to the server and receive a response. Our JavaScript will request a dynamic page this time, `test.php`, which will take the data we send and return a "computed" string - "Hello, [user data]!" - which we'll `alert()`.

First we'll add a text box to our HTML so the user can enter their name:

view plainprint?

```
1. <label>Your name:
2.   <input type="text" id="ajaxTextbox" />
3. </label>
4. <span id="ajaxButton" style="cursor: pointer; text-decoration: underline">
5.   Make a request
6. </span>
```

We'll also add a line to our event handler to get the user's data from the text box and send it to the `makeRequest()` function along with the URL of our server-side script:

view plainprint?

```
1. document.getElementById("ajaxButton").onclick = function() {
2.       var userName = document.getElementById("ajaxTextbox").value;
3.       makeRequest('test.php',userName);
4.   };
```

We need to modify `makeRequest()` to accept the user data and pass it along to the server. We'll change the request method from `GET` to `POST`, and include our data as a parameter in the call to `httpRequest.send()`:

view plainprint?

```
1. function makeRequest(url, userName) {
2.
3.     ...
4.
5.     httpRequest.onreadystatechange = alertContents;
6.     httpRequest.open('POST', url);
7.     httpRequest.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
8.     httpRequest.send('userName=' + encodeURIComponent(userName));
9.   }
```

The function `alertContents()` can be written the same way it was in Step 3 to alert our computed string, if that's all the server returns. However, let's say the server is going to return both the computed string and the original user data. So if our user typed "Jane" in the text box, the server's response would look like this:

```
{"userData":"Jane","computedString":"Hi, Jane!"}
```

To use this data within `alertContents()`, we can't just alert the `responseText`, we have to parse it and alert `computedString`, the property we want:

view plainprint?

```
1. function alertContents() {
2.     if (httpRequest.readyState === 4) {
3.       if (httpRequest.status === 200) {
4.         var response = JSON.parse(httpRequest.responseText);
5.         alert(response.computedString);
6.     } else {
7.       alert('There was a problem with the request.');
8.     }
9. }
```

For more on DOM methods, be sure to check Mozilla's DOM implementation documents.

Source: https://alexbosworth.backpackit.com/pub/67688

Ajax Mistakes

24 Jul 2007

Ajax is also a dangerous technology for web developers, its power introduces a huge amount of UI problems as well as server side state problems and server load problems. I've compiled a list of the many mistakes developers using Ajax often make.

This list is copied from my blog post: Alex Bosworth's Weblog: Ajax Mistakes

Disagree? Have your own mistake? This document has a collaborative page @ SWiK.net

Also check out the wiki page for places to use Ajax

Using Ajax for the sake of Ajax 6 Jun 2005

Sure Ajax is cool, and developers love to play with cool technology, but Ajax is a tool not a toy. A lot of the new Ajax applications are really just little toys, not developed for any real purpose, just experiments in what Ajax can do or trying to fit Ajax somewhere where it isn't needed. Toys might be fun for a little while, but toys are not useful applications.

Breaking the back button 18 May 2005

The back button is a great feature of standard web site user interfaces. Unfortunately, the back button doesn't mesh very well with Javascript. Keeping back button functionality is a major reason not to go with a pure Javascript web app.

Not giving immediate visual cues for clicking widgets 18 May 2005

If something I'm clicking on is triggering Ajax actions, you have to give me a visual cue that something is going on. An example of this is GMail loading button that is in the top right. Whenever I do something in GMail, a little red box in the top right indicates that the page is loading, to make up for the fact that Ajax doesn't trigger the normal web UI for new page loading.

Leaving offline people behind 6 Jun 2005

As web applications push the boundaries further and further, it becomes more and more compelling to move all applications to the web. The provisioning is better, the world-wide access model is great, the maintenance and configuration is really cool, the user interface learning curve is short.

However with this new breed of Ajax applications, people who have spotty internet connections or people who just don't want to switch to the web need to be accomodated as well. Just because technology 'advances' doesn't mean that people are ready and willing to go with it. Web application design should at least consider offline access. With GMail it's POP, Backpackit has SMS integration. In the Enterprise, it's web-services.

Don’t make me wait for Ajax 6 Jun 2005

With FireFox tabs I can manage various waits at websites, and typically I only have to wait for a page navigation. With AJAX apps combined with poor network connectivity/bandwidth/latency I can have a really terrible time managing an interface, because every time I do something I have to wait for the server to return a response. God help me if it has to go to the server’s disk before I can continue. Apps like that might even make me think Ajax wasn’t cool.

Sending sensitive information in the clear 6 Jun 2005

The security of AJAX applications is subject to the same rules as any web application, except that once you can talk asynchronously to the server, you may tend to write code that is a very chatty in a potentially insecure way. All traffic must be vetted to make sure security is not compromised.

Assuming AJAX development is single platform development. 6 Jun 2005

Ajax development is multi-platform development. Ajax code will run on IE’s javascript engine, Rhino (Mozilla’s js engine), or other minor engines that may grow into major engines. So it’s not enough just to code to JavaScript standards, there needs to be real-world thorough testing as well. A major obstacle in any serious Javascript development is IE’s buggy JS implementation, although there are tools to help with IE JS development.

Too much code makes the browser slow 18 May 2005

Ajax introduces a way to make much more interesting javascript applications, unfortunately interesting often means more code running. More code running means more work for the browser, which means that for some javascript intensive websites, especially poorly coded ones, you need to have a powerful CPU to keep the functionality zippy. The CPU problem has actually been a limit on javascript functionality in the past, and just because computers have gotten faster doesn’t mean the problem has disappeared.

Not having a plan for those who do not enable or have JavaScript. 6 Jun 2005

According to the W3C browser usage statistics, which if anything are skewed towards advanced browsers, 11% of all visitors don’t have JavaScript. So if your web application is wholly dependent on JavaScript, you’ve immediately cut a tenth of your audience.

Inventing new UI conventions 18 May 2005

A major mistake that is easy to make with Ajax is: ‘click on this non obvious thing to drive this other non obvious result’. Sure, users who use an application for a while may learn that if you click and hold down the mouse on this div that you can then drag it and permanently move it to this other place, but since that’s not in the common user experience, you increase the time and difficulty of learning your application, which is a major negative for any application.

Changing state with links (GET requests) 18 May 2005

As I’ve referenced in a previous posting, Ajax applications introduce lots of problems for users who assume GET operations don’t change state. Not only do state changing links cause problems for robots, users who are accustomed to having links drive navigation can become confused when links are used to drive application state changes.

Blinking and changing parts of the page unexpectedly 18 May 2005

The first A in Ajax stands for asynchronous. The problem with asynchronous messages is that they can be quite confusing when they are pop in unexpectedly. Asynchronous page changes should only ever occur in narrowly defined places and should be used judiciously, flashing and blinking in messages in areas I don’t want to concentrate on harkens back to days of the html blink tag.

Not using links I can pass to friends or bookmark 18 May 2005

Another great feature of websites is that I can pass URLs to other people and they can see the same thing that I’m seeing. I can also bookmark an index into my site navigation and come back to it later. Javascript, and thus Ajax applications, can cause huge problems for this model of use. Since the Javascript is dynamically generating the page instead of the server, the URL is cut out of the loop and can no longer be used as an index into navigation. This is a very unfortunate feature to lose, many Ajax webapps thoughtfully include specially constructed permalinks for this exact reason.

Not cascading local changes to other parts of the page 18 May 2005

Since Ajax/Javascript gives you such specific control over page content, it’s easy to get too focused on a single area of content and miss the overall integrated picture. An example of this is the Backpackit title. If you change a Backpackit page title, they immediately replace the title, they even remember to replace the title on the right, but they don’t replace the head title tag with the new page title. With Ajax you have to think about the whole picture even with localized changes.

Asynchronously performing batch operations 18 May 2005

Sure with Ajax you can make edits to a lot of form fields happen immediately, but that can cause a

lot of problems. For example if I check off a lot of check boxes that are each sent asynchronously to the server, I lose my ability to keep track of the overall state of checkbox changes and the flood of checkbox change indications will be annoying and disconcerting.

Scrolling the page and making me lose my place 18 May 2005

Another problem with popping text into a running page is that it can effect the page scroll. I may be happily reading an article or paging through a long list, and an asynchronous javascript request will decide to cut out a paragraph way above where I'm reading, cutting my reading flow off. This is obviously annoying and it wastes my time trying to figure out my place.

Blocking Spidering 6 Jun 2005

Ajax applications that load large amounts of text without a reload can cause a big problem for search engines. This goes back to the URL problem. If users can come in through search engines, the text of the application needs to be somewhat static so that the spiders can read it.