Introduction to XMLHttpRequest

Relevant Links

- XMLHttpRequest Specification[1]
- MDN's page on using XMLHttpRequest[2]
- Flanagan's book, chapter 18
- Cross-Origin Resource Sharing article[3]

Notes

XMLHttpRequest

The XMLHttpRequest allows you to load some remote resource on the background, and to take appropriate action when that resource has finished loading. The name implies that it is meant to be for HTTP requests that return XML, but neither of those two facts is strictly speaking a requirement.

This protocol is the essential component of what is known as "AJAX", an acronym standing for Asynchronous Javascript and XML

In order to use XMLHttpRequest, you need to implement the following steps:

- Create a new instance of XMLHttpRequest
- Add an `onLoad` handler to handle getting the resource
- Call the instance's `open` method, providing the request's parameters
- Call the instance's `send` method.

```
var xhr = new XMLHttpRequest();
xhr.onload = function(ev) {
    // Runs when resource is loaded
    console.log("Event", ev);
    console.log("xhr object has info", xhr);
};
xhr.open("get", "xhr_intro.html", true);
xhr.send();
```

You will see that the `xhr` object has been populated with a number of useful information about the request and corresponding response, including status text, code, response etc.

Look at the documentation for more details on how to use XMLHttpRequest directly. In particular, the API offers fairly fine-grained control over the whole process, including receiving progress notifications at various stages.

For the remaining of this section we will focus on using jQuery's AJAX features.

[1] https://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html

[2] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest

[3] http://www.html5rocks.com/en/tutorials/cors/

Ajax via jQuery

jQuery provides a simple method, called just $.ajax[4], for performing asynchronous requests.

Set up an ajax request. settings is an object with properties related to the request. Here are some of them (often you do not need to specify any):

The request method (POST, GET etc)
The URL for the requested resource
Use it to tell the browser not to used cached responses.
An object to use as the "context" / "this" for the various callbacks.
When sending data to the server, specifies their MIME type.
Used with GET requests to append a query to the URL.
The type of the data you expect in response. Servers can offer send the data back in multiple formats. jQuery will process these for you. Example types are "xml", "html", "json", "text".
Other headers.
How long to wait before the request times out
A function to be called when the response has returned.

[illegible] call returns a jqXHR object. It is what is known as a "deferred" or "promise" object. You can read more about javascript promises in many sources.

This object is outfitted with some useful methods, namely:

Takes as argument a function to call when the resource is downloaded. The function is called a f(data, textStatus, jqXHR).
Takes as argument a function to call if the resource fails to load. The function is called as f(jqXHR, textStatus, errorThrown).
Incorporates both the two above.

So a typical way to make an AJAX request would be:

```
$.ajax(requestObject).then(onSuccess, onFail)
```

Consult the jQuery documentation for details if you need to use this.

The Same Origin Policy and ways around it

The **same-origin** policy is a security measure that protects the execution of Javascript code with certain permissions. The policy says that "scripts are contained in a first web page are allowed to access data that is in another web page **only** if those pages have the same *origin*".

[4] http://api.jquery.com/jquery.ajax/

For instance there are ways in HTML to load another page within a “frame” of the existing page. This way for instance we can include third-party sections in our page, like amazon ads, widgets etc.

But we do not want those widgets to have access to the possibly sensitive data that our application creates. We have no reason to trust those scripts that are coming from other sources like google, amazon etc. They can access information related to the “frame” that created them, but they are prevented from interacting with the rest of the page.

This is very critical, and it causes some problems in relation to XHR and ajax.

Simply put, when performing an ajax request, you can in general only make requests to the same host that provided the page and script in the first place. For example we can’t easily make a request to anything other than, in this case, skiadas.github.io, because that would mean our script would attempt to read data originating in a different “origin”.

You can easily try this out:

```
// Getting jQuery loaded
var jq = document.createElement('script');
jq.src = "https://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js";
document.getElementsByTagName('head')[0].appendChild(jq);
// Wait a bit. Then $ = jQuery will exist
var xhr = $.ajax({ url: "http://www.google.com" });
// You get an error
```

There are two nowadays standard ways around this problem, both requiring some setup on the server side. So these will work provided the server has enabled the necessary functionality.

JSONP JSONP stands for “JSON with padding”. The idea is as follows: Normally the site would return information as a JSON object, so something like:

```
{
    a: 4,
    b: [1, 2, 5]
}
```

When we make a JSON GET request, we add a “callback=foo” to its end. Then what will get sent back instead is the JSON document wrapped in a function call:

```
foo({
    a: 4,
    b: [1, 2, 5]
})
```

This can then be placed as the source in a <script> tag. Script tags don’t suffer the same-origin problems (or else we couldn’t load things like jQuery). So as long as we have provided a global function called foo, this will get processed properly. The end result would be the injection into the page of code like:

```
<script src="https://pathToJSONResource?callback=foo"></script>
```

This code then loads the aforementioned “script” and executes it.

jQuery has support for this feature. So we can do something like:

```
var xhr = $.ajax({
    url: "http://pathToResource",
    dataType: "jsonp"
});
```

and jQuery will take care of the rest, in most situations.

JSONP is not without its risks however. You are loading a script from another source. You are being promised that that script will be nothing more than a function wrapping around some data, but the server could send you arbitrary Javascript code instead, and you will run it. This of course is a general problem with all Javascript scripts that we load from other sources.

JSONP is really a “clever hack”. It does the trick, but is not the “ideal” solution to the problem.

Cross-Origin Resource Sharing Cross-Origin Resource Sharing, usually abreviated as CORS, is another more recent approach to cross-domain access.

CORS requires that the server provide certain headers in response to a request, to indicate that it is willing to allow this cross-domain access. For instance this would have to be something like:

```
Access-Control-Allow-Origin: yourSiteHere
```

Essentially the server looks at the “origin” header in the browser’s request, and echos it back in the response. It essentially says to the browser: “I am aware that this other origin wants to access this data I am about to send you, and I am hereby giving my permission”.