Ajax

Ajax[1] is all about using JS to grab data from a server without having to reload the whole page or pause the application: HTTP requests are made asynchronously and small parts of the page can be updated incrementally. This makes the web app seem more responsive to the user. For example, when you search Google and you see a dropdown of suggested search terms, those terms are fetched asynchronously from a server, but you are able to stay on the same search page and continue typing as the suggestions appear.

Asynchronous Programming

We've already seen asynchrony in our own JS examples with `setTimeout` and `setInterval`, which schedule events to trigger function calls at a later time. Those function calls are squeezed in-between whatever else the browser may be busy with at the time. The opposite of "asynchronous" is "blocking", where everything else has to wait until the blocking task is done.

A web application's JavaScript code is always *single-threaded*. All function calls wait to be processed in its event loop, **one by one**. At the same time JavaScript provides us with the ability to listen for events: when an event occurs, the associated handler function is scheduled and it waits for its turn to be processed in the event loop.

Roughly speaking, asynchronous programming boils down to scheduling a function to be called once we see the "I finished the thing you asked for" browser event, where the "thing you asked for" could be "downloading data from a server", etc. While we wait for that special event to occur, the rest of the event loop keeps being processed, so the web app doesn't freeze.

The XMLHttpRequest object

(It's helpful to review what HTTP requests (GET, POST) and responses look like before reading this section. See previous weeks' notes.)

To make a custom HTTP request to a server, use `XMLHttpRequest`. Its name is misleading; it's not restricted to only XML. An XMLHttpRequest object represents a two-way conversation between client and server.

(**Cross-browser note:** In IE6 and older, you achieve the same thing as `XMLHttpRequest` with `new ActiveXObject("Microsoft.XMLHTTP");`)

Synchronous (Blocking)

Here's a quick example of making a **blocking** GET request to a public API endpoint. We'll look at how async works later.

```
request = new XMLHttpRequest();
// public API query
url = "https://represent.opennorth.ca/representatives/?gender=F"
// configure the request, false means it's blocking
// not using async to simplify the example
// example error: try omitting "GET" argument
request.open("GET", url, false)
// send the request; it's a GET request so the body is null
request.send(null)

// execution is blocked here until the server responds
// because we configured the request with async=false

// the response is now available in the request object's properties
// This will print 200 OK
console.log(request.status, request.statusText);
// can look at individual headers, too
console.log(request.getResponseHeader("content-type"));

// the response body happens to be a JSON string,
// so turn it into structured object
// (more on JSON later)
y = JSON.parse(request.responseText)
// y is an object, let's access its properties
console.log(y.objects[1].email);
```

Summary of XMLHttpRequest methods and properties

- `open(method, url, async)` configures the request.
  - Always capitalize the method ("GET", "POST")
  - Ensure the URL string doesn't contain any invalid characters (see `encodeURI`, `encodeURIComponent` below)
  - The URL can be:
    - absolute (starting with a protocol like `http://www.ex.com/path/to/something`)
    - relative your application server root (starting with `/` like `/path/to/something`)
    - relative to current file's directory (no initial `/`)
- `send(body)` actually triggers the request. When there's no body to send (like in GET requests), pass in `null`. Sending POST request parameters in the body would look like `send("type=muppet&name=elmo")`

Asynchronous (non-blocking)

In the previous example, `request.open("GET", url, false)` was followed by `request.send(null)` which won't return until the request is successfully sent and the server responds. That can take a long time (due to slow connection, large request, whatever). During that time, our application is unresponsive: no other application code can execute, no event handlers can fire.

That's why we use asynchronous requests instead: `otherRequest.open("GET", url, true)`. In this case, `otherRequest.send(null)` will return right away and allow subsequent code to execute immediately, regardless of how long the server takes to respond.

The sending and receiving of data happens in the background while our web app continues its execution. When the server does respond eventually, an event fires. In order to process the server response, we have to listen for the appropriate event.

The older (and therefore cross-browser-compatible) event handler to look at is `onreadystatechange`:

```
var r = new XMLHttpRequest();
r.open("GET", url, true);
r.send(null);
r.onreadystatechange = doSomething

// process the request when it's ready
function doSomething() {
  // the response was a success
  // (status greater than 400 indicates error)
  if (r.readyState === 4 && r.status === 200) {
    // process r.responseText however you please
  }
}
```

The modern event to listen for is `load`, but it won't work on older browsers:

```
var r = new XMLHttpRequest();
r.open("GET", url, true);
r.addEventListener("load", doSomething);
r.send(null);
```

In both cases, the handler `doSomething` has to figure out whether the server responded successfully with the data we asked for, so it looks at request properties.

`request.status` refers to HTTP response status:

- Status in the 100s to 300s indicates some form of information, success or redirection respective. The code of most interest is 200, which represents "OK".
- Status in the 400s to 500s indicates an error. 400s is client-side error, 500s is server error.

`request.readyState` refers to the status of the request being sent by `send`. The sequence of states is:

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4: request finished and response is ready

Encoding safer requests

Not all characters are allowed in valid URLs (like spaces), and some characters have special meaning (like /, ? and #). What if you need to submit a request parameter that contains a restricted character like `Do cats =^.^= eat cauliflower?`

Let's say the name of the parameter is `q`. Then to send a get request with that value, I could try a URL like `http://catanswers.com?q=Do cats =^.^= eat cauliflower?`, but this would be interpreted by the server as:

- Base url: `http://catanswers.com`
- query parameter 1: `q=Do cats`
- invalid substring: `=^.^= eat cauliflower?`

A malicious user could take advantage of this behaviour to try to form a URL that causes the server to do something unexpected.

```
"To avoid unexpected requests to the server, you should call  encodeURIComponent on an
```

Let's do that. `encodeURIComponent` converts a string to use escape characters that may interfere with a URL.

```
r = new XMLHttpRequest();
r.open("GET", "http://catanswers.com?q=\"" +
    encodeURIComponent("Do cats =^.^= eat cauliflower?") +
    "\"",
    true);
```

This yields a URL like:
`http://catanswers.com?q="Do%20cats%20%3D%5E.%5E%3D%20eat%20cauliflower%3F"`. Note that it's important to use `"` not `'` in long URL parameters (e.g comment="blah blah blah").

To convert the string back to normal, you would use **decodeURIComponent**

There's also **encodeURI** and **decodeURI**, if you want to convert an entire URL, but note that this will keep `#`, `?` and `/` unconverted because they have special meaning for URLs.

More about POST requests

You can also use Ajax to send data to a server by making a POST request. As mentioned earlier, include the request body in the `send` method like `r.send("type=muppet&name=elmo")`.

In addition, you also need to tell the server what kind of data you are sending. This has to be done by setting the appropriate request header.

For example:

```
var data = "email=" + encodeURIComponent(email);
r.open('POST', 'http://www.example.com/somepage.php', true);
r.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
r.send(data);
```

Where to send requests

You can't do Ajax on local file system, need to use HTTP, for example, so need to test on a local dev server.

Request must be to server on same domain or to a server that is set up to accept cross-origin API requests.[2] Lots of organizations have public APIs (YouTube, Twitter, etc.)

If you want to experiment with using public APIs, here are a few easy-to-use APIs that came up for me in a quick search. Example requests:

- http://www.omdbapi.com/?t=Arrival
- http://netflixroulette.net/api/api.php?director=Quentin Tarantino"
- Another api (needs key): http://openweathermap.org/appid

Simplify the use of XMLHttpRequest with a help callback

Remember callback functions? We saw an example or two earlier on when we first started looking at JavaScript functions. When you use `addEventListener`, the handler you provide is an example of a callback function: `document.addEventListener("click", myHandler)`.

A callback is a function that is provided as a parameter to another function, which will "call us back" with the callback function.

Here's an example of using XMLHttpRequest with a callback function:

```
/** Read the response at `url` with `cb`
 * @param {string} url - address to send a GET request to
 * @param {string -> ?} cb - function that accepts request response text as input
 */
function readData(url, cb) {
  var req = new XMLHttpRequest();
  req.open("GET", url, true);
  req.addEventListener("load", function() {
    if (req.status === 200)
      cb(req.responseText);
    });
  req.send(null);
}
```

`readData` is a helper function that abstracts away the details of XMLHttpRequest, and `cb` can be any function that processes the server response however you like.

We could use `readData` as follows:

```
readData("http://blah.com/?something", function (text) {
  var p = document.createChild("p");
  p.textContent = "The response is " +
      text.length + " chars long.";
  document.body.appendChild(p);
});
```

Additional references

- A different explanation of Async Requests from Eloquent JavaScript
- The JavaScript Event Loop
- Guide to Ajax on MDN.
- Escape invalid characters in strings that you want to use in URL parameters with encodeURIComponent
- More about XMLHttpRequest
- More about Cross-Origin Resource Sharing (CORS)

---

1. The term Ajax comes from the abbreviation "Asynchronous JavaScript and XML", but it refers to more than just XML today. (XML and JSON are data formats. JSON is used most often today because it's actually part of JavaScript itself. We'll look at both XML and JSON in future lectures. Ajax can also be used to receive HTML or plain text.)↩
2. See CORS (Cross-Origin Resource Sharing)↩
