Lecture 4: HTTP & AJAX

Nick Hynes

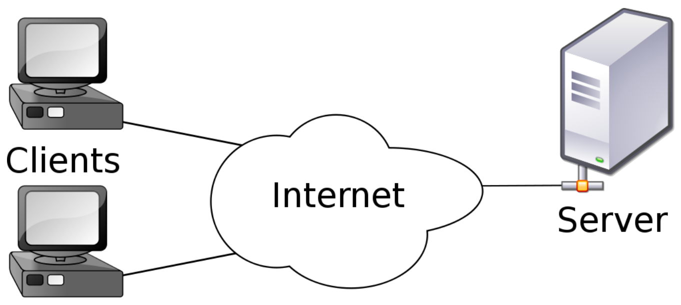
January 21, 2014

Topics

1 Introduction to Networking

Despite what the name may imply, this will not be a lesson on making business connections. Rather, we will be discussing how data is transferred across the network to produce what we all know and love as the World Wide Web!

WWW and many other Internet applications like email and centralized file hosting/sharing (as opposed to distributed, P2P architectures like BitTorrent) operate using the *client-server* model. In this model, the client—or the user's browser, in our case—sends a request for a certain file, or resource, to a server which processes the request and sends back to the client either the requested resource or a status code that explains why the request failed.



Of course, the server may perform computations based on the request parameters (e.g. request variables, request payload, cookies) and return a dynamically-generated page. If you use a web-service like Facebook or Gmail, then you're already familiar with this kind of server-side dynamism.

2 HTTP(S)

HTTP and HTTPS are the main ways HyperText is transferred over the network. HTTP stands for HyperText Transfer Protocol and, as you may already be thinking, is used to transfer documents containing HyperText. The 'S' in HTTPS means Secure; HTTPS is basically encrypted HTTP.

HTTP is a protocol used to request and send resources over the network. This is done by attaching an HTTP header to every request and response. Fortunately, in HTTP/1.1, the current version, these headers are in a great form for demonstration: human-readable text! The following examples are similar to the request and response headers generated every time you visit the course website.

Example 1 – Requesting introjsiap.com

```
1  GET / HTTP/1.1
2  Host: introjsiap.com
3  Connection: keep-alive
4  User-Agent: Mozilla/5.0 Gecko/20100101 Firefox/26.0
5  Accept: text/html,application/xhtml+xml,application/xml;q=0.5
6  Accept-Language: en-US,en;q=0.5
7  Accept-Encoding: gzip, deflate
8  If-None-Match: "80b3c-dd3-4f03251d489e7"
9  If-Modified-Since: "Fri, 17 Jan 2014 22:46:58 GMT"
10 Cache-Control: "max-age=0"
```

This header can be broken down into three main parts. On lines 1-3, the client (or, more precisely, the user-agent), specifies what resource it wants, specifically the document root of introjsiap.com (/), and how the connection should be handled after the initial transaction ends. On lines 5-7, the user-agent specifies the preferred type, language, and compression options of the requested resource; the server can use this information to *negotiate* which resource to send. Finally, the fields on lines 8-10 deal with caching; since the client already has a page identified by “80b3c-dd3-4f03251d489e7”, that was modified some time before 17/1/14, it doesn’t need the server to resend it if it hasn’t changed.

The first portion is the most important for client-side networking, so we’ll focus on that today. Line 1, itself, consists of three parts: the HTTP *verb*, which defines how the server should process the request, the requested resource, specified by by a URL, and the HTTP version, which will in most cases be 1.1 (until 2.0 is adopted). The other interesting field is `Connection`. A value of `keep-alive`, as opposed to `close`, signals to the server that the TCP (socket) connection should be reused for future requests; this feature of HTTP/1.1 significantly cuts down on networking overhead.

Example 2 – Here you go: introjsiap.com

```
HTTP/1.1 200 OK
Connection: keep-alive
Date: Tue, 21 Jan 2014 11:11:11 GMT
Etag: "80b3c-dd3-4f03251d489e7"
Keep-Alive: timeout=5
Server: Apache/2.2.16 (Debian)
Vary: Accept-Encoding
Content-Length: 3450

<!-- page contents here (has a length of 3450 bytes) -->
```

The response headers are marginally less interesting than the request headers, so we’ll be a bit more choosy with them. The most important part, by far, is on the first line; specifically the numerical status code, in this case `200`. This allows the client to determine if the request succeeded or the reason it didn’t. The `Etag` field is a document version identifier used to facilitate caching. The `Content-Length` specifies how many bytes the client should read before considering the message to be finished.

There are many other request and response header fields, but they should reveal themselves to you when the time comes (sounds almost mythical doesn't it?). The above examples should encapsulate all of the most important features of basic HTTP. In fact, now that you know how to formulate HTTP requests, you don't even have to use a web browser any more!

2.1 Methods

The following HTTP request methods, commonly referred to as verbs, are used to signal the intent of the client's request and should be implemented on the server in a reasonable (aka sane) manner.

- GET - Requests the specified resource. Should only retrieve data and nothing else. To send parameters to the server using this request method, use `http://theurl/?param1=value1&param2=value2&etc`
- POST - Requests that the server create a new sub-resource based on the request payload. The payload is urlencoded (like the GET parameters without the ?) or JSON and is included as the body of the POST request. Don't forget to set the media type and the content length!
- PUT - Requests that the server set or update the resource at the specified path to the contained data. This is similar to POST except that it creates/update the resource rather than a sub-resource.
- DELETE - Deletes the specified resources.
- HEAD - Requests only the headers (this one isn't too common)

As with most things covered thus far, this list is not complete, but is likely to be the most useful. You may satisfy your curiosity, however, by following the link in *Additional resources.*

2.2 Status codes

Status code	What it means	When it's used
200	OK	Everything went well and the server returned the requested page. Everyone is happy.
301	Moved permanently	The resource previously located at the requested URL has moved to another URL. Use this in conjunction with the `Location` header field to get the user-agent to redirect.
400	Bad request	The client sent an ill-formatted request and the server doesn't know what else to do.
401	Unauthorized	The user needs to authenticate to gain access to the resource. This is used with HTTP authentication (not form-based auth).
403	Forbidden	The user does not have permission to make the request.
404	Not found	The server couldn't find the requested resource
500	Internal server error	The server had an error and could not complete the request

There is a link to a complete list of HTTP status codes in the *Additional resources* section. While those listed should provide all of the basic, required functionality, there might be a status code that is more descriptive and appropriate for your application.

2.3 REST

The combination of HTTP methods, cookies, status codes, headers, and authentication provides a particularly rich medium for creating scalable, modular, high-performance web-applications. These may be buzzwords, but the underlying concept is sound.

REST is an acronym for Representational State Transfer. Essentially, REST defines a client-server application model in which the client sends everything the server needs to fulfill the request with each request. This is equivalent to saying that the server does not store any sort of session data for the user. It is immediately obvious that this, along with improved caching ability, reduces the server's memory and disk space usage (at the cost of increased bandwidth and possibly latency [which can be mitigated by caching]). Additionally, resources under the REST model are located at sane URLs and are operated on using the HTTP verbs. For example, if I wanted to create a new user for my social networking application, I could send a POST request with the user's data to https://cebook.com/users (drop the 'thefa', it's cleaner). The server could then return a descriptive status code that can be interpreted on the client-side to notify the user the result.

2.4 HTTP vs HTTPS

Whereas HTTP transactions are conducted using plaintext, those of HTTPS are encrypted using TLS (Transport Layer Security) or the less-secure SSL (Secure Sockets Layer). This prevents eavesdroppers from determining the contents of HTTP traffic. This is particularly useful for online shopping and banking and preventing the government from spying on you, among other things. While HTTPS has a small amount of performance overhead, the benefits provided by HTTPS, which also include verification of the server's identity, are sufficient reason to always use it. In fact, encrypting data using TLS is a requirement of HTTP/2.0

If you think of the network as a stack (like pancakes), you have, from top to bottom, the application layer, the transport layer, the internet layer, and the link layer. HTTP and HTTPS operate on the application layer, so there's really no need to deal with the rest of the IP suite at this point (although it's really, really interesting!).

The application layer is concerned with the transmission of abstract, application-specific data, as opposed to getting the data from point A to point B. HTTP operates near the top of this layer and, of course, provides a way to request and send hypertext and related resources. Here's the gist of what happens during a generic HTTP exchange:

1. The client prepares an HTTP request for a certain resource

2. The HTTP request, now treated as arbitrary data, travels down the client's networking stack and is sent over the network to the server

3. The request travels up the network stack of the server and is processed by a program that understands HTTP
4. The server generates an HTTP response
5. The response travels down the network stack of the server and up the stack of the client
6. The HTTP transaction is now complete

TLS sits at the bottom of the application layer so that when HTTP traffic passes through it from either direction, it is either encrypted or decrypted using a session secret (this is also a really good read. The Wikipedia diagram for Diffe-Hellman key exchange is one of my favorite diagrams). Practically, this means that once TLS is established, HTTP can proceed as usual.

3 AJAX

Woohoo! Now for the fun part: using JavaScript to make HTTP requests. However, before we dive into AJAX (not the detergent), let me relate to you a tale of the pre-AJAX web, circa 2006.

> "In a world where making a small change to a web page required a complete reload of the page, along with yet another request to the server, the user experience was comparatively bad by today's standards.
>
> Imagine submitting a form... and being sent a new web page. The screen might even flash as the page reloads! *gasp* Oh, the horror!"

AJAX remedies all of these problems as it allows an already-loaded page to make further requests of the server without actually reloading the page. Additionally, like JS event handlers, the responses to these AJAX requests are (read: should be) processed *asynchronously* using callback functions; this allows the web-application to maintain a responsive user-interface and submit further AJAX requests. Thus, AJAX forms the basis of the modern, responsive web-application.

The acronym AJAX technically stands for Asynchronous Javascript And XML, but basically everything *but* XML is sent using it. If you're sending XML, then that's fine, too, but my only question is "*why?*" There are infinitely less verbose serialization formats available.

3.1 JSON

Before covering the AJAX methods, it will be useful to re-introduce our old friend, the JavaScript Object, as a powerful and convenient method of serializing data, that is, converting it into a form that can be transferred over the wire and reconverted on the other end. The benefit of using JSON is that it's much, much less verbose than XML (saves bandwidth) and can be used by JavaScript much more easily since JSON can be parsed directly into a JavaScript object that can be manipulated using the usual methods. Here's an example of JSON in case you've forgotten:

Example 3 – An example on JSON

```
[
  {
    "user": "A user",
    "message": "Hey, how's it going?"
    "timestamp": 123456789
  },
  {
    "user": "Another user",
    "message": "nm, bro, just chillin",
    "timestamp": 987654321
  }
]
```

Note that the keys are double-quoted Strings and the values are JavaScript Objects (including double-quoted Strings, Arrays, Numbers, Booleans, and generic Objects).

When transferring JSON from the server, it is generally a good idea to specify the `Content-Type` of the response as `application/json` so that the client can handle it appropriately and the server can compress the file to the best of its abilities.

```
//To convert a [non-circular] Object to JSON
JSON.stringify(theObj);

//To convert JSON to an Object
JSON.parse(theObj);
```

3.2 “Native”

Example 4 – AJAX Using the XHR Object

```
var req = new XMLHttpRequest();

var contentLoaded = function() {
  //this is the response object
  if(this.status === 200) {
    console.log(this.responseText); //"Hello, world!"
  } else {
  console.log("Error: "+this.status);
  }
}

req.addEventListener("onload", contentLoaded);
//could also use req.onload = contentLoaded

req.open("GET", "helloworld.txt", true);
req.send();
console.log("Request sent"); //"Request sent"
```

This example is slightly more concise than it would have been if support for early versions of IE were needed, but if you’re lucky, this should never be the case.

On the first line, we create a new XMLHttpRequest (or XHR) object. We then assign it a listener for its *load* event, which is triggered when the response is received. Next, the XHR object is "opened," which essentially means that its request details are set. Finally, the XHR is sent to the server and patiently awaits a response.

While this is about as complicated as AJAX gets, the XHR object provides a number of methods that can be used to vastly increase its utility.

XHR object functions

- getResponseHeader(String header) – returns the value of the requested header or `null` if not set
- `open(method, URL, [Boolean async], [user], [password])` – initializes a request of type `method`, to be sent to `URL`. If the optional `async` is true, then the XHR object's callbacks will be run when the response is received. Otherwise, all scripts on the page will block until the request is handled (This is generally a BAD IDEA™). The optional Strings, `username` and `password` are mainly used for HTTP Basic auth (only transmit this over a secure network!).
- setRequestHeader(String headerFieldName, String headerValue) – sets the value of the specifieed request header field
- send([(String|Blob) data]) – Sends the request to the server with the body of the request containing the contents of the `data` argument. The Blob (Binary Large OBject) format is used when transferring binary data, like file uploads.

XHR object properties and events

- progress event – fired when download progress changes
- load event – fired when the response is loaded
- error event – fired when the request fails
- abort event – fired when the request is aborted [by the user]

- response – the response parsed according to its content-type (of either text, JSON, Document (as in HTML Document), Blob, or ArrayBuffer) or `null` if the request did not succeed

- responseType – the content-type of the response (text, JSON, Document, Blob, or ArrayBuffer)
- responseText – the response text or `null` if the request did not succeed
- responseXML – the response XML or `null`
- status – the HTTP status code
- statusText – the text returned with the status code (e.g. OK for 200 OK)

Some things to watch out for

- callback functions must be registered before calling `send`
- when sending data using the argument of the `send` function, be sure to set the `Content-Type` and `Content-Length` request headers
- the `setRequestHeader` method must be called after `open` but before `send`

3.3 jQuery

$.ajax

The jQuery AJAX function is the "low-level interface" for the rest of the jQuery AJAX functions.

```
$.ajax(URL, [settings]); //method defaults to GET

$.ajax(settings);
```

The `settings` object passed to the jQuery `ajax` function defines all of the properties and behaviors of the request. Here's a list of some of the more interesting settings. These are all optional, by the way (except maybe URL if not already set...).

- url: the request URL
- type: the HTTP method to use. Default: GET

- async: (Boolean) true if asynchronous, false if synchronous. Default: true
- beforeSend: function(jqXHR, settings) a function that can configure the jqXHR object and its settings before it is sent
- statusCode: (Object) an object of Number: Function pairs that defines which functions to run if a set status code is returned
- contentType: (String) the content type. Defaults to 'application/x-www-form-urlencoded; charset=UTF-8', which is good.
- data: (Object|String) the data to be sent to the server. An Object value will be stringified.
- headers: (Object) an object of header field: value pairs
- username: the HTTP authentication username
- password: the HTTP authentication password (send this over HTTPS if not using HTTP Digest auth!)

All of the jQuery AJAX methods return a jqXHR object which is a superset of the native XHR object, so you can use them in the same ways. However, the jqXHR object has the added benefit of being a jQuery Promise object that has allows chaining of callbacks on events like `done`, `fail`, `then`, and `always`. The latter two are called when anything happens to the Promise object or after the Promise object is rejected or resolved (failed or done'd), respectively.

Example 5 – AJAX Using jQuery's `$.ajax`

```
$.ajax("helloworld.txt")
  .done(function(data) {
  console.log(data); //"Hello, world!"
  })
  .error(function(jqXHR, statusText) {
    console.log("Error: " + statusText);
  });
```

$.get

The `$.get` function is simply a convenience method for `$.ajax`.

```
$.get(URL, [data], [success callback]);
//Chaining callbacks still works, but a success callback
//can be passed directly to the function
```

$.getJSON

This method is a convenience method for `$.get`. A convenience method of a convenience method. How convenient! The difference is that the `data` parameter passed to the callback functions is a JS object.

```
$.getJSON(URL, [data], [success callback]);
//Chaining callbacks still works, but a success callback
//can be passed directly to the function
//callback takes format of (data, textStatus, jqXHR)
```

$.getScript

This method GETs a JavaScript from the server and then executes it.

```
$.getScript(URL, [success callback]);
//Chaining callbacks still works, but a success callback
//can be passed directly to the function
//callback takes format of (script, textStatus, jqXHR)
```

$.post

Like a GET request except POST (so it's not really like a GET request at all then, is it?).

```
$.post(URL, [data], [success callback]);
//Chaining callbacks still works, but a success callback
//can be passed directly to the function
//callback takes format of (script, textStatus, jqXHR)
```

jQueryFormObject.serialize()

Serializes a jQuery object that contains a form and returns a urlencoded String. This is very useful for sending form data using `$.post`.