CHAPTER 9

AJAX Requests: Dynamic Data and Page Updates

AJAX, or Asynchronous JavaScript and XML, is a feature provided by the web API that allows data to be updated or retrieved from a server without reloading the entire page. This capability was absent from the browser initially. The time without this feature marked the infancy of the web, and with it came along a less-than-ideal user experience that resulted in a fair amount of redundant bytes circulated between client and server. The inefficiency of this primitive model was compounded by the fact that Internet bandwidth was extremely limited by today's standards. Back in 1999 when Microsoft first introduced `XMLHTTP` as an ActiveX control in Internet Explorer 5.0,[1] about 95% of Internet users were limited by a 56 Kbps or slower dial-up connection.[2]

`XMLHTTP` was a proprietary JavaScript object implemented in Microsoft's Internet Explorer browser, and it represented a huge leap in both web development technology and user experience. It was the first full-featured built-in transport for focused client/server communication that allowed for updates without replacing the entire page. Previously, the entire page had to be reloaded even if only a small segment of the data on the page had changed. The initial API for this new transport matches its modern-day standardized cousin: `XMLHttpRequest`. Essentially, this object allows a developer to construct a `new` transport instance, send a GET, POST, PUT, PATCH, or DELETE request to any endpoint (on the same domain), and then programmatically retrieve the status and message body of the server's response. Although the aging `XMLHttpRequest` will eventually be replaced by the Fetch API,[3] it has thrived unopposed and mostly unchanged for about 15 years.

Mastering the Concepts of AJAX Communication

It is critical to understand a few key concepts when dealing with AJAX communication:

1. Asynchronous operations.
2. HyperText Transfer Protocol, also know as HTTP.
3. JSON, URL encoding, and multipart form encoding.
4. The Same Origin Policy.

The first two items will be dealt with directly in this section, in addition to an introduction to web sockets (which are not as important as some other concepts, but still potentially useful). The last two in the list will be addressed later on in this chapter.

[1] https://blogs.msdn.microsoft.com/ie/2006/01/23/native-xmlhttprequest-object/
[2] www.websiteoptimization.com/bw/0403/
[3] https://fetch.spec.whatwg.org

R. Nicholus, *Beyond jQuery*, DOI 10.1007/978-1-4842-2235-5_9

Async Is Hard

Based on my extensive experience with AJAX communication, along with observations of other developers struggling with this piece of the web API, the most attractive attribute of this feature is also its most confusing. JavaScript does not abstract asynchronous operations nearly as well as other more traditional languages, such as Java. On top of the historical lack of intuitive native support for tasks that occur out of band (such as AJAX requests), there are currently three different common ways to account for these types of asynchronous operations. These methods include callbacks, promises, and asynchronous functions. Although native support for asynchronous operations has improved over time, most developers still must explicitly deal with these types of tasks, which can be challenging due to the fact that it often requires all surrounding code to be structured accordingly. That often makes the software developer's job of accounting for asynchronous calls awkward and the resulting code complex. This, of course, adds risk and potentially more bugs to the underlying application.

Callbacks will be demonstrated in this chapter, as will promises. Both promises and callbacks are covered in much more detail in Chapter 11, along with asynchronous functions, a feature defined in the ECMAScript 2017 specification that aims to make dealing with asynchronous operations, such as AJAX requests, surprisingly easy. However, some developers don't have the luxury of using async functions (due to lack of current browser support as of 2016), so the reality of dealing with AJAX requests remains that you must embrace their asynchronous nature instead of hiding from it. This is quite mind-bending, initially. Even after you have successfully grasped this concept, expect frequent frustration in less-than-trivial situations, such as when dealing with nested asynchronous requests. If this is not already clear through previous experience, you may even come to realize this complexity as you complete this chapter. Still, this concept is perhaps the most important of all to master when working with AJAX requests.

HTTP

The primary protocol used to communicate between a browser and a server is, of course, HTTP, which stands for HyperText Transfer Protocol. Tim Berners-Lee, the father of the Web, created the first official HTTP specification[4] in 1991. This first version was designed alongside HTML and the first web browser with one method: GET. When a page was requested by the browser, a GET request would be sent, and the server would respond with the HTML that makes up the requested page, which the web browser would then render. Before AJAX was introduced as a complementary specification, HTTP was mostly limited to this workflow.

Though HTTP started with just one method—GET—several more were added over time. Currently, HEAD, POST, PUT, DELETE, and PATCH are all part of the current specification, version 2, which is maintained by the Internet Engineering Task Force (IETF) as RFC 7540.[5] GET requests are expected to have an empty message body (request payload), with a response that describes the resource referenced in the request URI (Universal Resource Indicator). It is a "safe" method, such that no changes to the resource should be made server-side as a result of handling this request. HEAD is very similar to GET, except it returns an empty message body. However, HEAD is useful in that it includes a response header—`Content-Length`—with a value equal the the number of bytes that would be transferred had the request been GET instead. This is useful to, for example, check the size of a file without actually returning the entire file. HEAD, as you might expect, is also a "safe" method.

DELETE, PUT, POST, and PATCH are not safe, in that they are expected to possibly change the associated resource on the server. Of these four "unsafe" methods, two of them—PUT and DELETE—are considered to be *idempotent*, which means that they will always produce the same result even though they are called multiple times. PUT is commonly used to replace a resource, whereas DELETE is, obviously, used to remove a resource. PUT requests are expected to have a message body that describes the updated resource content, and DELETE is *not* expected to have a payload. POST differs from PUT in that it will create a new resource. Finally, PATCH,

[4] www.w3.org/Protocols/HTTP/AsImplemented.html
[5] https://httpwg.github.io/specs/rfc7540.html

a relatively new HTTP request method,[6] allows a resource to be modified in very specific ways. The message body of this request describes exactly how the resource should be modified. PATCH differs from the PUT method in that it does not entirely replace the referenced resource.

All AJAX requests will use one of these methods to communicate dynamically with a server. Note that new methods such as PATCH may not be supported in older browsers. Later in this chapter, I go into more detail regarding proper use of these methods, and how the AJAX request specification can be used with these methods to produce a highly dynamic web application.

Expected and Unexpected Responses

An understanding of the protocol used to communicate between client and server is a fundamentally important concept. Part of this is, of course, *sending* requests, but the *response* to these requests is equally important. Requests have headers and optionally a message-body (payload), whereas responses are made up of three parts: the response message-body, headers, and a status code. The status code is unique to responses and is usually accessible on the underlying transport instance (such as `XMLHttpRequest` or `fetch`). Status codes are usually three digits and can be generally classified based on the most significant digit. 200-level status codes are indicative of success, 300 is used for redirects, while 400- and 500-level statuses indicate some sort of error. These are all formally defined in great detail in RFC 2616.[7]

Just as it is important to handle code exceptions with a `try/catch` block, it is equally important to address exceptional AJAX responses. Although 200-level responses are often expected, or at least desired, you must also account for unexpected or undesired responses, such as 400- and 500-level, or even responses with a status of 0 (which may happen if the request is terminated due to a network error or the server returns a completely empty response). I have observed that simply ignoring exceptional conditions appears to be common, and this is not limited to handling of HTTP responses. In fact, I am guilty of this myself.

Web Sockets

Web sockets are a relatively new web API feature compared to traditional AJAX requests. They were first standardized in 2011 by the IETF (Internet Engineering Task Force) in RFC 6455[8] and are currently supported by all modern browsers, with the exception of Internet Explorer 9. Web sockets differ from pure HTTP requests in a number of ways, most notably their lifetime. Whereas HTTP requests are normally very short lived, web socket connections are meant to remain open for the life of the application instance or web page. A web socket connection starts as an HTTP request, which is required for the initial handshake. But after this handshake is complete, client and server are free to exchange data at will in whatever format they have agreed upon. This web socket protocol allows for true real-time communication between client and server. Although web sockets are not explored in more depth in this chapter, I felt it was useful to at least mention them, because they *do* account for another method of JavaScript-initiated asynchronous communication.

Sending GET, POST, DELETE, PUT, and PATCH Requests

jQuery provides first-class support for AJAX requests through the appropriately named `ajax()` method. Through this method, you can send AJAX requests of any type, but jQuery also provides aliases for some standardized HTTP request methods—such as `get()` and `post()`—which save you a few keystrokes. The web API provides two objects, `XMLHttpRequest` and `fetch`, to send asynchronous requests of any type from browser to server. `XMLHttpRequest` is supported in all browsers, but `fetch` is relatively new and not supported in all modern browsers, though a solid polyfill is available which provides support for all browsers.

[6] https://tools.ietf.org/html/rfc5789
[7] www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10
[8] https://tools.ietf.org/html/rfc6455

A simple GET request to a server endpoint using jQuery, with a trivial response handler, looks something like this:

```
$.get('/my/name').then(
  function success(name) {
    console.log('my name is ' + name);
  },
  function failure() {
    console.error('Name request failed!');
  }
);
```

■ **Note** The `console` object is not available in Internet Explorer 9 and older unless the developer tools are open.

In the preceding code, we are sending a GET request to the "/my/name" server endpoint and expecting a plaintext response containing a name value, which we are then printing to the browser console. If the request fails (such as if the server returns an error status code), the `failure` function is invoked. In this case, I'm making use of the `promise-like object (or "thenable")` that jQuery's `ajax()` method and its aliases return. jQuery provides several ways to deal with responses, but I will focus specifically on the one demonstrated before. Chapter 11 talks more about promises, which are a standardized part of JavaScript.

The same request, sent without jQuery and usable in all browsers, involves a bit more typing but is certainly not too difficult:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/my/name');
xhr.onload = function() {
  if (xhr.status >= 400) {
    console.error('Name request failed!');
  }
  else {
    console.log('my name is ' + xhr.responseText);
  }
};
xhr.onerror = function() {
  console.error('Name request failed!');
};
xhr.send();
```

`onload` is called when the request completes and some response has been received. It *may* be an error response, though, so we must confirm by looking at the response status code. `onerror` is called if the request fails at some very low level, such as due to a CORS error. The `onload` property is where we can easily set our response handler. From here, we can be sure the response has completed and then can determine whether the request was successful and get a handle on the response data. All of this is available on the instance of `XMLHttpRequest` we created and assigned to the `xhr` variable. From the preceding code, you may be a bit disappointed that the web API doesn't support promises like jQuery. That *was* true, up until the `fetch` API[9] was created by the WHATWG. The Fetch API provides a modern native replacement for the aging `XMLHttpRequest` transport, and it is currently supported by Firefox, Chrome, Opera, and Microsoft Edge, with Safari support likely on the horizon. Let's look at this same example using `fetch`:

[9] https://fetch.spec.whatwg.org

```
fetch('/my/name').then(function(response) {
  if (response.ok) {
    return response.text();
  }
  else {
    throw new Error();
  }
}).then(
  function success(name) {
    console.log('my name is ' + name);
  },
  function failure() {
    console.error('Name request failed!');
  }
);
```

The code not only embraces the promises specification, it also removes a lot of the boilerplate commonly seen with `XMLHttpRequest`. You'll see more examples of `fetch` throughout this chapter. Note that any of the previous examples are identical for HEAD requests, with the exception of the method specifier. You'll also notice that `fetch` returns a `Promise`, similar to jQuery's `ajax()`. The difference is that jQuery executes the error function when a non-success status code is executed. `fetch` only does this, like `XMLHttpRequest`, when a low-level network error occurs. But we can trigger our error function when the server returns a response with an error code by throwing an exception if the response is not "ok", via the `ok` property on the `Response` object passed to our first promise handler.

Many developers who learned web development through a jQuery lens probably think that this library is doing something magical and complex when you invoke the `$.ajax()` method. That couldn't be further from the truth. All the heavy lifting is done by the browser via the `XMLHttpRequest` object. jQuery's `ajax()` is just a wrapper around `XMLHttpRequest`. Using the browser's built-in support for AJAX requests isn't very difficult, as you'll see in a moment. Even cross-origin requests are not only simple without jQuery—you'll see how they are actually *easier* without jQuery.

Sending POST Requests

I've demonstrated how to GET information from a server endpoint using jQuery, `XMLHttpRequest`, and `fetch`. But what about some of the other available HTTP methods? Instead of GETting a name, suppose you would like to add a new name to the server. The most appropriate method for this situation is POST. The name to add will be included in our request payload, which is a common place to include this sort of data when sending a POST. To keep it simple, we'll send the name using a MIME type of text/plain (I cover more advanced encoding techniques later in this chapter). I'll omit response handling code here as well so we can focus on the key concepts:

```
$.ajax({
  method: 'POST',
  url: '/user/name',
  contentType: 'text/plain',
  data: 'Mr. Ed'
});
```

We're using jQuery's generic `ajax()` method, so we can specify various parameters, such as the request's `Content-Type` header. This will send a POST request with a plaintext body containing the text "Mr. Ed". We must explicitly specify the `Content-Type` in this case because jQuery will otherwise set this header to "application/x-www-form-urlencoded", which is not what we want.

The same POST request using `XMLHttpRequest` looks like this:

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('POST', '/user/name');
3  xhr.send('Mr. Ed');
```

Sending this request without jQuery actually takes *fewer* lines of code. By default, the `Content-Type` is set to "text/plain" by `XMLHttpRequest` in this case,[10] so we don't need to mess with any request headers. We can conveniently pass the body of our request as a parameter to the `send` method, as illustrated before.

If your goal is to embrace the latest and greatest web standards, you can try sending this POST using `fetch` instead:

```
1  fetch('/user/name', {
2    method: 'POST',
3    body: 'Mr. Ed'
4  });
```

Sending this request looks similar to jQuery, but without a lot of the boilerplate. Like `XMLHttpRequest`, `fetch` intelligently sets the request `Content-Type` based on the request payload (in some cases), so we do not need to specify this request header.[11]

Sending PUT Requests

Whereas POST requests are often used to create a new resource, PUT requests typically replace an existing one. For example, PUT would be more appropriate for replacing information about an existing product. The URI of the request identifies the resource to be replaced with the new information located in the body. To simply illustrate sending a PUT request using jQuery, `XMLHttpRequest`, and `fetch`, I'll demonstrate updating a mobile phone number for an existing user record:

```
1  $.ajax({
2    method: 'PUT',
3    url: '/user/1',
4    contentType: 'text/plain',
5    data: //complete user record including new mobile number
6  });
```

This PUT request using jQuery looks almost identical to the previously illustrated POST, with the exception of the `method` property. This user is identified by their ID, which happens to be 1. You likely won't be surprised to see that sending a PUT with `XMLHttpRequest` is similar to the previous example as well:

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('PUT', '/user/1');
3  xhr.send(/* complete user record including new mobile number */);
```

The Fetch API, as expected, provides the most concise approach:

```
1  fetch('/user/1', {
2    method: 'PUT',
3    body: //complete user record including new mobile number
4  });
```

[10] www.w3.org/TR/XMLHttpRequest#dom-xmlhttprequest-send
[11] https://fetch.spec.whatwg.org/#body-mixin

Sending DELETE Requests

DELETE requests are similar to PUTs in that the resource to be acted upon is specified in the request URI. The main difference, even though this is not clearly mandated by RFC 7231,[12] DELETE requests typically do *not* contain a message body. The IETF document refers to the possibility that a message body may in fact cause problems for the requestor:[13]

> A payload within a DELETE request message has no defined semantics; sending a payload body on a DELETE request might cause some existing implementations to reject the request.

This means that the only safe way to specify parameters, apart from the resource to be acted upon, is to include them as query parameters in the request URI. Other than this exceptional case, DELETE requests are sent in the same manner as PUTs. And like a PUT request, DELETE requests are idempotent. Remember that an *idempotent* request is one that behaves the same regardless of the number of times it is called. It would be quite surprising if multiple calls to delete the same resource resulted in, for example, removal of a different resource.

A request to remove a resource, using jQuery, looks like this:

```
1  $.ajax('/user/1', {method: 'DELETE'});
```

And the same simple request, using `XMLHttpRequest`, can be achieved with only two extra lines of code:

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('DELETE', '/user/1');
3  xhr.send();
```

Finally, we can send this DELETE request natively in many modern browsers using the ever-so-elegant Fetch API (or in any browser with a polyfill for `fetch` (maintained on GitHub))[14]

```
1  fetch('/user/1', {method: 'DELETE'});
```

Sending this request with `fetch` is just as simple as `$.ajax()`; we can easily write the entire thing in one line without losing readability.

Sending PATCH Requests

As mentioned earlier, PATCH requests are relatively new on the HTTP scene and are used to update a *portion* of an existing resource. Take our previous PUT request example, where we only want to update a user's mobile phone number, but have to include all other user data in our PUT request as well. For small records, that may be fine, but for records of substantial size, it may be a waste of bandwidth. One approach may be to define specific endpoints for each part of the user's data, or identify the data to be updated based on URI query parameters, but that just clutters up our API. For this situation, it is best to use PATCH requests instead.

Let's revisit the PUT example where we need to update an existing user's mobile number, this time with PATCH. A jQuery approach—using a simple plaintext-based key-value message body to indicate the property to change along with the new property value—would look like this:

```
1  $.ajax({
2    method: 'PATCH',
```

[12] https://tools.ietf.org/html/rfc7231
[13] https://tools.ietf.org/html/rfc7231#section-4.3.5
[14] https://github.com/github/fetch

```
3    url: '/user/1',
4    contentType: 'text/plain',
5    data: 'mobile: 555-5555'
6  });
```

■ **Note** If your underlying model is JSON, it is more appropriate to send a JSON Patch Document instead.[15] But we haven't talked much about JSON yet. I cover that later in this chapter.

Remember that we can use any format we choose for the body of the PATCH request to specify the data to update, as long as client and server are in agreement. If we prefer to use `XMLHttpRequest`, the same request looks like this:

```
1  var xhr = new XMLHttpRequest();
2  xhr.open('PATCH', '/user/1');
3  xhr.send('mobile: 555-5555');
```

For the sake of completeness, I'll show you how to send the exact same request using the Fetch API as well:

```
1  fetch('/user/1', {
2    method: 'PATCH',
3    body: 'mobile: 555-5555'
4  });
```

Encoding Requests and Reading Encoded Responses

In the preceding section, I touched on the simplest of all encoding methods—text/plain—which is the Multipurpose Internet Mail Extension (MIME) for unformatted text that makes up an HTTP request or response. The simplicity of plain text is both a blessing and a curse. That is, it is easy to work with, but it's only appropriate for very small and simple cases. The lack of any standardized structure limits its expressiveness and usefulness. For more complex (and more common) requests, there are more appropriate encoding types. In this section, I discuss three other MIME types: "application/x-www-form-urlencoded", "application/json", and "multipart/form-data". At the end of this section, you will be not only familiar with these additional encoding methods, you will also be able to understand how to encode and decode messages without jQuery, especially when sending HTTP requests and parsing responses.

URL Encoding

URL encoding can take place in the request's URL or in the message body of the request/response. The MIME type of this encoding scheme is "application/x-www-form-urlencoded", and data consists of simple key/value pairs. Each key and value is separated by an equal sign (=), while each pair is separated by an ampersand (&). But the encoding algorithm is much more than this. Keys and values may be further encoded, depending on their character makeup. Non-ASCII characters, along with some reserved characters, are replaced with a percent character (%) followed by the hex value of the character's associated byte. This is further defined in the HTML forms section of the W3C HTML5 specification.[16] This description is arguably a bit over-simplistic, but it's appropriate and comprehensive enough for the purposes of this chapter. If you'd like to learn more about the encoding algorithm for this MIME type, please have a look at the spec, though it is a bit dry and may take a few reads to properly parse.

---

[15] https://tools.ietf.org/html/rfc6902
[16] www.w3.org/TR/html5/forms.html%23application/x-www-form-urlencoded-encoding-algorithm

For GET and DELETE requests, URL-encoded data should be included at the end of the URI, since these request methods typically should not include payloads. For all other requests, the body of the message is the most appropriate place for your URL-encoded data. In this case, the request or response must include a `Content-Type` header of "application/x-www-form-urlencoded"—the MIME type of the encoding scheme. URL-encoded messages are expected to be relatively small, especially when dealing with GET and DELETE requests due to real-world URI length restrictions in place on browsers and servers.[17] Although this encoding method is more elegant than text/plain, the lack of hierarchy means that these messages are also limited in their expressiveness to some degree. However, child properties *can* be tied to parent properties using brackets. For example, a "parent" key that has a value of a set of child key/value pairs—"child1" and "child2"—can be encoded as "parent[child1]=child1val&parent[child2]=child2val". In fact, this is what jQuery does when encoding a JavaScript object into a URL-encoded string.

jQuery's API provides a function that takes an object and turns it into a URL-encoded string: `$.param`. For example, if we wanted to encode a couple simple key/value pairs into a URL-encoded string, our code would look like this:

```
$.param({
  key1: 'some value',
  'key 2': 'another value'
});
```

That line would produce a string of "key1=some+value&key+2=another+value". The specification for the application/x-www-form-urlencoded MIME type *does* declare that spaces are reserved characters that should be converted to the "plus" character. However, in practice, the ASCII character code is also acceptable. So, the same couple of key/value pairs can *also* be expressed as "key1=some%20value&key%202=another%20value". You'll see an example of this when I cover URL encoding with the web API.

If we wanted to create a new user with three properties—name, address, and phone—we could send a POST request to our server with a URL-encoded request body containing the information for the new user. The request would look like this with jQuery:

```
$.ajax({
  method: 'POST',
  url: '/user',
  data: {
    name: 'Mr. Ed',
    address: '1313 Mockingbird Lane',
    phone: '555-555-5555'
  }
});
```

jQuery does admittedly make this relatively intuitive, as it allows you to pass in a JavaScript object describing the new user. There is no need to use the `$.param` method. As mentioned earlier, jQuery's `$.ajax()` API method assumes a `Content-Type` of "application/x-www-form-urlencoded", and it encodes the value of your `data` property to automatically to match this assumption. You don't have to think about encoding or encoding types at all in this case.

Even though the web API *does* require you to be conscious of the encoding type *and* it requires you encode your data before sending the request, these tasks are not overly complicated. I've already showed you how jQuery allows you to encode a string of text to "application/x-www-form-urlencoded"—using `$.param()`—and you can accomplish the same without jQuery using the `encodeURI()` and `encodeURIComponent()` methods available on global namespace. These methods are defined in the ECMAScript specification and have been available since the ECMA-262 3rd edition specification,[18] completed in 1999.

---

[17] http://stackoverflow.com/questions/417142/what-is-the-maximum-length-of-a-url-in-different-browsers

[18] http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%2C3rdedition%2CDecember1999.pdf

Both `encodeURI()` and `encodeURIComponent()` perform the same general task—URL encode a string. But they each determine *which* parts of the string to encode a bit differently, so they are tied to specific use cases. `encodeURI()` is meant to be used for a complete URL, such as *or* a string of key-value pairs separated by an ampersand (&), such as "first=ray&last=nicholus". However, `encodeURIComponent()` is meant to only be used on a single value that needs to be URL encoded, such as "Ray Nicholus" or "Mr. Ed". If you use `encodeURIComponent()` to encode the full URL listed earlier in this paragraph, then the colon, forward slashes, question mark, and ampersand will all be URL encoded, which is probably not what you want (unless the entire URL is itself a query parameter).

Looking back at the simple URL-encoding example with jQuery a little while back in this section, we can encode the same data with the web API using `encodeURI()`:

```
1  encodeURI('key1=some value&key 2=another value');
```

A couple things about the output of `encodeURI`, which produces "key1=some%20value&key%202 =another%20value". First, notice that, while jQuery replaced spaces with the plus sign (+), `encodeURI()` (and `encodeURI-Component`) replace spaces with "%20". This is perfectly valid, but a notable difference. Second, whereas jQuery allows you to express the data to be encoded as a JavaScript object, `encodeURI()` requires you separate the keys from the values with an equals sign (=) and the key/value pairs with an ampersand (&). Taking this a bit further, we can duplicate the same POST request we sent earlier that adds a new name, first using `XMLHttpRequest`:

```
1  var xhr = new XMLHttpRequest(),
2      data = encodeURI(
3        'name=Mr. Ed&address=1313 Mockingbird Lane&phone=555-555-5555');
4  xhr.open('POST', '/user');
5  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
6  xhr.send(data);
```

Another notable difference between the `XMLHTTPRequest` route and jQuery's `$.ajax()` is that we must set the request header's `Content-Type` header as well. jQuery sets it for us by default, and it is necessary to let the server know how to decode the request data. Luckily, `XMLHttpRequest` provides a method for setting request headers—the aptly named `setRequestHeader()`.

We can gain some of the same benefits seen previously with the Fetch API, but we still need to do our own encoding. No worries—that can be accomplished without much trouble:

```
1  var data =
2    encodeURI('name=Mr. Ed&address=1313 Mockingbird Lane&phone=555-555-5555');
3    fetch('/user', {
4    method: 'POST',
5    headers: {'Content-Type': 'application/x-www-form-urlencoded'},
6    body: data
7  });
```

Just as with `XMLHttpRequest`, the `Content-Type` header must also be specified here, since the default `Content-Type` for `fetch`-initiated requests with a `String` body is also "text/plain". But, again, the Fetch API allows AJAX requests to be constructed in a more elegant and condensed form, similar to the solution that jQuery has provided for some time now. `Fetch` is supported in Firefox, Chrome, Opera, and Edge, there is an open case to add support to Safari.[19] In the near future, `XMLHttpRequest` will be an artifact of history, and `fetch`, which rivals jQuery's AJAX support, will be the AJAX transport of choice.

[19] https://bugs.webkit.org/show_bug.cgi?id=151937

JSON Encoding

JavaScript Object Notation, better known as JSON, is considered to be a "data-interchange language" (as described on json.org). If this obscure description leaves you a bit puzzled, don't feel bad—it's not a particularly useful summary. Think of JSON as a JavaScript object turned into a string. There *is* a bit more to explain, but this is a reasonable high-level definition in my opinion. In web development, this is particularly useful if a browser-based client wants to easily send a JavaScript object to a server endpoint. The server can, in turn, respond to requests from this client, also supplying JSON in the response body, and the client can easily convert this into a JavaScript object to allow easy programmatic parsing and manipulation. Although "application/x-www-form-urlencoded" requires that data be expressed in a flat format (or denote parent/child relationships using non-standard bracket notation), "application/json" allows data to be expressed in a hierarchical format. A key can have many sub-keys, and those sub-keys can have child keys as well. In this sense, JSON is more more expressive than URL-encoded data, which itself is more expressive and structured than plain text.

In case you are not already familiar with this common data format, let's represent a single user in JSON:

```
{
  "name": "Mr. Ed",
  "address": "1313 Mockingbird Lane",
  "phone": {
    "home": "555-555-5555"
    "mobile": "444-444-4444"
  }
}
```

Notice that JSON gives us the power to easily express sub-keys, a feature that is lacking with plaintext and URL-encoded strings. Mr. Ed has two phone numbers, and with JSON we can elegantly associate both numbers with with a parent "phone" property. Extending our AJAX example from the previous section, let's add a new name record to our server using jQuery, this time with a JSON-encoded payload:

```
$.ajax({
  method: 'POST',
  url: '/user',
  contentType: 'application/json',
  data: JSON.stringify({
    name: 'Mr. Ed',
    address: '1313 Mockingbird Lane',
    phone: {
      home: '555-555-5555',
      mobile: '444-444-4444'
    }
  });
});
```

Notice that our code is looking a bit uglier than the previous URL-encoded example. Two reasons for this, one being that we must explicitly specify the `Content-Type` header via a `contentType` property, an abstraction that isn't really much help. Second, we must leave the comfortable and familiar jQuery API to convert our JavaScript object into JSON. Since jQuery provides no API method to accomplish this (which is odd since it *does* provide a way to convert a JSON string to an object - `$.parseJSON()`), we must make use of the `JSON` object, which has been standardized as part of the ECMAScript specification. The `JSON` object provides a couple of methods for converting JSON strings to JavaScript objects and back again. It first appeared in

the ECMAScript 5.1 specification,[20] which means it is supported in Node.js as well as all modern browsers, including Internet Explorer 8. The `JSON.stringify()` method, used in the earlier jQuery POST example, takes the user record, which is represented as a JavaScript object, and converts it into a proper JSON string. This is needed before we can send the record to our server.

If you'd like to send a GET request that receives JSON data, you can do that quite simply using `getJSON`:

```
$.getJSON('/user/1', function(user) {
  // do something with this user JavaScript object
});
```

What about sending the previous POST request without jQuery? First, with `XMLHttpRequest`:

```
var xhr = new XMLHttpRequest(),
    data = JSON.stringify({
      name: 'Mr. Ed',
      address: '1313 Mockingbird Lane',
      phone: {
        home: '555-555-5555',
        mobile: '444-444-4444'
      }
    });
xhr.open('POST', '/user');
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.send(data);
```

No surprises with XHR. This attempt looks very similar to the URL-encoded POST request we sent in the last section, with the exception of stringifying our JavaScript object and setting an appropriate `Content-Type` header. As you have already seen, we must address the same issues whether we are using jQuery or not.

But sending a JSON-encoded request is only half of the required knowledge. We must be prepared to receive and parse a JSON response too. That looks something like this with `XMLHTTPRequest`:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/user/1');
xhr.onload = function() {
  var user = JSON.parse(xhr.responseText);
  // do something with this user JavaScript object
};
xhr.send();
```

Notice that we're sending a GET request for user data and expecting our server to return this record in a JSON-encoded string. In the preceding code we're leveraging the `onload` function, which will be called when the request completes. At that point, we can grab the response body via the `responseText` property on our XHR instance. To turn this into a proper JavaScript object, we must make use of the other method on the JSON object—`parse()`. In modern browsers (except Internet Explorer), receiving JSON data with `XMLHttpRequest` is even easier:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/user/1');
xhr.onload = function() {
  var user = xhr.response;
```

[20] www.ecma-international.org/ecma-262/5.1/#sec-15.12

```
5    // do something with this user JavaScript object
6  };
7  xhr.send();
```

The preceding example assumes that the server has included a `Content-Type` header of "application/json". This lets `XMLHttpRequest` know what to do with the response data. It ultimately converts it to a JavaScript object and makes the converted value available on the `response` property of our `XMLHttpRequest` instance.

Finally, we can use the Fetch API to add this new user record to our server, again using a JSON- encoded request:

```
 1  fetch('/user', {
 2    method: 'POST',
 3    headers: {'Content-Type': 'application/json'},
 4    body: JSON.stringify({
 5      name: 'Mr. Ed',
 6      address: '1313 Mockingbird Lane',
 7      phone: {
 8        home: '555-555-5555',
 9        mobile: '444-444-4444'
10      }
11    });
12  });
```

This is unsurprising and straightforward, but what about receiving a JSON-encoded response from our server? Let's send a simply GET request using `fetch` for a user record, with the expectation that our server will respond with a user record encoded as a JSON string:

```
1  fetch('/user/1').then(function(request) {
2      return request.json();
3    }).then(function(userRecord) {
4        // do something with this user JavaScript object
5  });
```

The Fetch API provides our promissory callback with a `Request` object.[21] Since we are expecting JSON, we call the `json()` method on this object, which itself returns a `Promise`. Note that the `json()` method is actually defined on the `Body` interface.[22] The `Request` object implements the `Body` interface, so we have access to methods on both interfaces here. By returning that promise, we can chain another promissory handler and expect to receive the JavaScript object representation of the response payload as a parameter in our last success callback. Now we have our user record from the server. Pretty simple *and* elegant! Again, if promises are still a bit murky, don't worry—I cover them in great detail later in Chapter 11.

Interestingly, ECMAScript 2016 provides the ability to use an alternate syntax that makes the above (any many other code example) even more elegant. Below, I've re-written the preceding example using "arrow functions":

```
1  fetch('/user/1')
2    .then(request => request.json())
3    .then(userRecord => {
4      // do something with this userRecord object
5    });
```

[21] https://fetch.spec.whatwg.org/#request-class
[22] https://fetch.spec.whatwg.org/#body

Arrow functions are out of scope for this book, but I thought it would be nice to point this out to readers who were not already aware of this language feature. Note that not all browsers support arrow functions, so you'll likely need to use a build-time compiler that converts these to "traditional" functions. Some JavaScript compilers include TypeScript, Closure Compiler, Babel, and Bublè. Some of these are mentioned later in the book, such as in Chapter 11.

Multipart Encoding

Another common encoding scheme, usually associated with HTML form submissions, is multipart/- form-data, also known as *multipart encoding*. The algorithm for this method of passing data is formally defined by the IETF in RFC 2388.[23] The use of this algorithm in the context of HTML forms is further described by the W3C in the HTML specification.[24] Non-ASCII characters in a multipart/form-data message do *not* have to be escaped. Instead, each piece of the message is split into fields, and each field is housed inside of a multipart boundary. Boundaries are separated by unique IDs that are generated by the browser, and they are guaranteed to be unique among all other data in the request, as the browser analyzes the data in the request to ensure a conflicted ID is not generated. A field in a multipart encoded message is often an HTML `<form>` field. Each field is housed inside of its own multipart boundary. Inside of each boundary is a header, where metadata about the field lives (such as its name/key), along with a body (where the field *value* lives).

Consider the following form:

```
<form action="my/server" method="POST" enctype="multipart/form-data">
  <label>First Name:
    <input name="first">
  </label>

  <label>Last Name:
    <input name="last">
  </label>

  <button>Submit</button>
</form>
```

When the submit button is hit, any entered data will be submitted to the server. Let's say the user enters "Ray" into the first text input, and "Nicholus" into the second text input. After hitting submit, the request body may look something like this:

```
----------------------------168653674598641646212772 1994
Content-Disposition: form-data; name="first"

Ray
----------------------------168653674598641646212772 1994
Content-Disposition: form-data; name="last"

Nicholus
----------------------------168653674598641646212772 1994--
```

[23] www.ietf.org/rfc/rfc2388.txt
[24] www.w3.org/TR/html5/forms.html#multipart-form-data

The server knows how to find each form field by looking for the unique ID marking each section in the request body. This ID is included by the browser in the `Content-Type` header, which in this case is "multipart/form-data; boundary=——————————168653674598641646212772199 4". Notice that the MIME type of the message body is separated by the multipart boundary ID with a semicolon.

But HTML form submissions are not the *only* instance where we might want to encode our request message using the multipart/form-data MIME type. Since this MIME type is trivial to implement in all server-side languages, it is probably a safe choice for transmitting key/value pairs from client to server. But, above all else, multipart encoding is perfect for mixing key/value pairs with binary data (such as files). I talk more about uploading files in the next section.

So how can we send a multipart encoded request using jQuery's `$.ajax()` method? As you'll see shortly, it's ugly, and the layer of abstraction that jQuery normally provides is incomplete in this case, as you must delegate directly to the web API anyway. Continuing with some of the previous examples, let's send a new user record to our server—a record consisting of a user's name, address, and phone number:

```
var formData = new FormData();
formData.append('name', 'Mr. Ed');
formData.append('address', '1313 Mockingbird Lane');
formData.append('phone', '555-555-5555');

$.ajax({
  method: 'POST',
  url: '/user',
  contentType: false,
  processData: false,
  data: formData
});
```

To send a multipart encoded AJAX request, we must send a `FormData` object that contains our key/value pairs, and the browser takes care of the rest. There is no jQuery abstraction here; you must make use of the web API's `FormData` directly. Note that `FormData` isn't supported in Internet Explorer 9. This lack of abstraction is a hole in jQuery's blanket, though `FormData` is relatively intuitive and quite powerful. In fact, you can pass it a `<form>` element, and key/value pairs will be created for you, ready for asynchronous submission to your server. Mozilla Developer Network has a great writeup on `FormData`.[25] You should read it for more details.

The biggest problem with sending MPE requests with jQuery is the obscure options that must be set to make it work. `processData: false`? What does that even mean? Well, if you don't set this option, jQuery will attempt to turn `FormData` into a URL-encoded string. As for `contentType: false`, this is required to ensure that jQuery doesn't insert its own Content-Type header. Remember from the section introduction that the browser *must* specify the Content-Type for you as it includes a calculated multipart boundary ID used by the server to parse the request.

The same request with plain old `XMLHttpRequest` contains no surprises and, quite frankly, isn't any less intuitive than jQuery's solution:

```
var formData = new FormData(),
    xhr = new XMLHttpRequest();

formData.append('name', 'Mr. Ed');
formData.append('address', '1313 Mockingbird Lane');
formData.append('phone', '555-555-5555');

xhr.open('POST', '/user');
xhr.send(formData);
```

[25] https://developer.mozilla.org/en-US/docs/Web/API/FormData

In fact, using XHR results in *less* code, and we don't have to include nonsensical options such as `contentType: false` and `processData: false`. As expected, the Fetch API is even simpler:

```
1  var formData = new FormData();
2  formData.append('name', 'Mr. Ed');
3  formData.append('address', '1313 Mockingbird Lane');
4  formData.append('phone', '555-555-5555');
5
6  fetch('/user', {
7    method: 'POST',
8    body: formData
9  });
```

See? If you can look *Beyond jQuery*, just a bit, you'll find that the web API is not always as scary as some make it out to be. In this case, jQuery's API suffers from inelegance.

Uploading and Manipulating Files

Asynchronous file uploading, a topic that I have quite a bit of experience with,[26] is yet another example of how jQuery can often fail to effectively wrap the web API and provide users with an experience that justifies use of the library. This is indeed a complex topic, and though I can't cover everything related to file uploads here, I'll be sure to explain the basics and show how files can be uploaded using jQuery, `XMLHttpRequest`, and `fetch` both in modern browsers *and* ancient browsers. In this specific and somewhat unusual case, note that Internet Explorer 9 is excluded from the definition of "modern browsers." The reason for this will become clear soon.

Uploading Files in Ancient Browsers

Before we get into uploading files in older browsers, let's define a very important term for this section: *browsing context*. A browsing context can be a `window` or an `iframe`, for example. So if we have a `window`, and an `iframe` inside of this `window`, we have two browsing contexts: the parent `window` and the child `iframe`.

The *only* way to upload files in ancient browsers, including Internet Explorer 9, is to include an `<input type="file">` element inside of a `<form>` and submit this form. By default, the server's response to this form submit replaces the current browsing context. When working with a highly dynamic single-page web application, this is unacceptable. We need to be able to upload files in older browsers and still maintain total control over the current browsing context. Unfortunately, there is no way to prevent a form submit from replacing the current browsing context. But we can certainly create a child browsing context where we submit the form and then monitor this browsing context to determine when our file has been uploaded by listening for changes.

This approach can be implemented quite easily, simply by asking the form to target an `<iframe>` in the document. To determine when the file has finished uploading, attach an "onload" event handler to the `<iframe>`. To demonstrate this approach, we'll need to make a few assumptions in order to make this relatively painless. First, imagine our primary browsing context contains a fragment of markup that looks like this:

```
1  <form action="/upload"
2        method="POST"
3        enctype="multipart/form-data"
4        target="uploader">
5
```

[26] http://fineuploader.com

```
  <input type="file" name="file">

</form>

<iframe name="uploader" style="display: none;"></iframe>
```

Notice that the `enctype` attribute is set to "multipart/form-data". You may remember from the previous section that a form with file input elements must generate a multipart encoded request in order to properly communicate the file bytes to the server.

Second assumption: we are given a function—`upload()`—that is called when the user selects a file via the file `input` element. I'm not going to cover this specific detail now, since we haven't yet covered event handling. I discuss events in Chapter 10.

Okay, so how do we make this happen with jQuery? Like this:

```
function upload() {
  var $iframe = $('IFRAME'),
      $form = $('FORM');

  $iframe.on('load', function() {
    alert('file uploaded!')
  });

  $form.submit();
}
```

We *could* have accomplished a lot more with JavaScript/jQuery here if we needed to, such as setting the `target` attribute. If we were working with nothing more than a single file input element, we could have also dynamically created the form and moved the file input inside of it as well. But none of that was necessary since our markup contained everything we needed already. How much work and complexity has jQuery saved us from here? Let's take a look at the non-jQuery version for a comparison:

```
function upload() {
   var iframe = document.getElementsByTagName('IFRAME')[0],
       form = document.getElementsByTagName('FORM')[0]

   iframe.onload = function() {
     alert('file uploaded!');
   }

   form.submit();
}
```

jQuery hasn't done much at all for us. The web API solution is almost identical to the initial jQuery code. In both cases, we must select the iframe and form, attach an `onload` handler that does something when the upload has completed, and the submit the form. In both cases, our primary browsing context/window remains untouched. The server's response is buried in our hidden `<iframe>`. Pretty neat!

Uploading Files in Modern Browsers

There is a much more modern way to upload files asynchronously, and this is possible in all modern browsers, with the exception of Internet Explorer 9. The ability to upload files via JavaScript is possible due to the File API,[27] *and* `XMLHttpRequest` Level 2 (a small and non-breaking update to the original specification in terms of API).[28] Both of these elements of the web API are standardized by W3C specifications. jQuery does *nothing* to make uploading files easier. The modern native APIs that bring file uploading the browser are elegant, easy to use, and powerful. jQuery doesn't do much to attempt to provide a layer of abstraction here, and actually makes file uploading a bit awkward.

A typical workflow involves the following steps:

1. User selects one or more files via a `<input type="file" multiple>` element. Note that the `multiple` Boolean attribute allows the user to select multiple files, provided that the browser supports this attribute.
2. JavaScript is used to specify a "change" event listener, which is called when the user selects one or more files.
3. When the "change" listener is invoked, grab the file or files from the `<input type="file">` element. These are made available as `File` objects,[29] which extend the `Blob` interface.[30]
4. Upload the `File` objects using your AJAX transport of choice.

Because we haven't covered events yet, assume a function exists that, when called, signals that our user has selected one or more files on the `<input type="file">` that we are monitoring. The goal is to upload these files. And to keep this example focused and simple, *also* assume that the user is only able to select *one* file. This means our `<input type="file">` element will *not* include a `multiple` Boolean attribute. File uploading in modern browsers (except in IE9) with jQuery can be accomplished, given the environment I just described, like this:

```
function onFileInputChange() {
  var file = $('INPUT[type="file"]')[0].files[0];

  $.ajax({
    method: 'POST',
    url: '/uploads',
    contentType: false,
    processData: false,
    data: file
  });
}
```

The preceding code will send a POST request to an "/uploads" endpoint, and the request body will contain the bytes of the file our user selected. Again we must use the obscure `contentType: false` option to ensure that jQuery leaves the Content-Type header alone so that it may be set by the browser to reflect the MIME type of the file. Also, `processData: false` is needed to prevent jQuery from encoding the `File` object, which would destroy the file we are trying to upload. We also could have included the file in a `FormData` object and uploaded that instead. This becomes a better option if we need to upload multiple files in a single request, or if we want to easily include other form data alongside the file or files.

[27] www.w3.org/TR/FileAPI/
[28] www.w3.org/TR/XMLHttpRequest2/
[29] www.w3.org/TR/FileAPI/#dfn-file
[30] www.w3.org/TR/FileAPI/#dfn-Blob

Without jQuery, using `XMLHttpRequest`, file uploading is actually much simpler:

```
function onFileInputChange() {
  var file = document.querySelector('INPUT[type="file"]').files[0],
      xhr = new XMLHttpRequest();

  xhr.open('POST', '/uploads');
  xhr.send(file);
}
```

As with the jQuery example, we're grabbing the selected `File` from the file input's `files` property, which is included as part of the File API, and sending it to our endpoint by passing it into the `send()` method, which supports `Blob`s as of `XMLHttpRequest` level 2.

File uploading is also possible with the Fetch API. Let's take a look:

```
function onFileInputChange() {
  var file = document.querySelector('INPUT[type="file"]').files[0];

  fetch('/uploads', {
    method: 'POST',
    body: file
  });
}
```

Reading and Creating Files

Generally speaking, developers comfortable with jQuery often attempt to solve *all* of their front-end development problems with jQuery. They sometimes fail to see the web beyond this library. When developers become dependent on this safety net, it can often lead to frustration when a problem is not addressable through jQuery. This is the oppressive magic I wrote about in Chapter 1. You just saw how jQuery, at best, provides almost no help when uploading files. Suppose you want to read a file, or even create a new one or modify an existing one to be sent to a server endpoint? This is an area where jQuery has absolutely zero coverage. For reading files, you must rely on the `FileReader` interface,[31] which is defined as part of the File API. Creation of "files" browser-side requires use of the `Blob` constructor.[32]

The simplest `FileReader` example, which is sufficient for demonstration purposes here, is to read a text file to the console. Suppose a user selected this text file via a `<input type"file">`, and the text `File` object is sent to a function for output. The code required to read this file and output it to the developer tools console would involve the following code:

```
function onTextFileSelected(file) {
  var reader = new FileReader();

  reader.onload = function() {
    console.log(reader.result);
  }

  reader.readAsText(file);
}
```

[31] www.w3.org/TR/FileAPI/#dfn-filereader
[32] www.w3.org/TR/FileAPI/#dfn-Blob

No jQuery needed or even possible to read files. And why would you need it? Reading files is pretty easy. Suppose you want to take the same text file and then append some text to the end of the file before uploading it to your server? Surprisingly, this is pretty easy too:

```
1  function onTextFileSelected(file) {
2    var modifiedFile = new Blob([file, 'hi there!'], {type: 'text/plain'});
3    // ...send modifiedFile to uploader
4  }
```

The `modifiedFile` here is a copy of the selected file with the text "hi there!" added to the end. This was accomplished in a grand total of one line of code.

Cross-domain Communication: an Important Topic

As more and more logic is offloaded to the browser, it is becoming common for web applications to pull data from multiple APIs, some of which exist as part of third-party services. A great example of this is a web application (like Fine Uploader)[33] that uploads files directly to an Amazon Web Services (AWS) Simple Storage Service (S3) bucket. Server-to-server cross-domain requests are simple and without restrictions. But the same is not true for cross-domain requests initiated from the browser. For the developer who wants to develop a web application that sends files directly to S3 from the browser, an obstacle lies in the way: the same-origin policy.[34] This policy places restrictions on requests initiated by JavaScript. More specifically, requests made by `XMLHttpRequest` between domains are prohibited. For example, sending a request to `https://api.github.com` from `https://mywebapp.com` is prevented by the browser because of the same-origin policy. While this restriction is in place for added security, this seems like a major limiting factor. How can you make legitimate requests from domain A to domain B without funneling them through a server on domain A first? The next two sections cover two specific approaches to accomplish this.

The Early Days (JSONP)

The same-origin policy prevents scripts from initiating requests outside the domain of their current browsing context. Although this covers AJAX transports such as `XMLHttpRequest`, elements such as `<a>`, `<img>`, and `<script>` are *not* bound by the same-origin policy. JavaScript Object Notation with Padding, or JSONP, exploits one of these exceptions to allow scripts to make cross-origin GET requests.

If you're not familiar with JSONP, the name may be a bit misleading. There is actually no JSON involved here at all. It's a very common misconception that JSON must be returned from the server when the client initiates a JSONP call, but that's simply not true. Instead, the server returns a function invocation, which is *not* valid JSON.

JSONP is essentially just an ugly hack that exploits the fact that `<script>` tags that load content from a server are not bound by the same-origin policy. There need to be cooperation and an understanding of the convention by both client and server for this to work properly. You simply need to point the `src` attribute of a `<script>` tag at a JSONP-aware endpoint and include the name of an existing global function as a query parameter. The server must then construct a string representation that, when executed by the browser, will invoke the global function, passing in the requested data.

Exploiting this JSONP approach in jQuery is actually pretty easy. Say we want to get user information from a server on a different domain:

```
1  $.ajax('http://jsonp-aware-endpoint.com/user/1', {
2    jsonp: 'callback',
```

[33] http://fineuploader.com
[34] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

```
3    dataType: 'jsonp'
4  }).then(function(response) {
5    // handle user info from server
6  });
```

jQuery takes care of creating the `<script>` tag for us and also creates and tracks a global function. When the global function is called after the response from the server is received, jQuery passes that to our response handler mentioned earlier. This is actually a pretty nice abstraction. Completing the same task without jQuery is certainly possible, but not as nice:

```
1  window.myJsonpCallback = function(data) {
2    // handle user info from server
3  };
4
5  var scriptEl = document.createElement('script');
6  scriptEl.setAttribute('src',
7    'http://jsonp-aware-endpoint.com/user/1?callback=myJsonpCallback');
8  document.body.appendChild(scriptEl);
```

Now that you have this newfound knowledge, I suggest you forget it and avoid using JSONP altogether. It's proven to be a potential security issue.[35] Also, in modern browsers, CORS is a much better route. And you're in luck: CORS is featured in the next sub-section. This explanation of JSONP serves mostly as a history lesson and an illustration of how jQuery *was* quite useful and important before the modern evolution of web specifications.

Modern Times (CORS)

CORS, short for Cross Origin Resource Sharing, is the more modern way to send AJAX requests between domains from the browser. CORS is actually a fairly involved topic and very commonly misunderstood even by seasoned web developers. Although the W3C specification[36] can be hard to parse, Mozilla Developer Network has a great explanation.[37] I'm only going to touch on a few CORS concepts here, but the MDN article is useful if you'd like to have a more detailed understanding of this topic.

With a reasonable understanding of CORS, sending a cross-origin AJAX request via JavaScript is not particularly difficult in modern browsers. Unfortunately, the process is *not* as easy in Internet Explorer 8 and 9. Cross-origin AJAX requests are only possible via JSONP in IE7 and older, and you are restricted to GET requests in those browsers (because this is an inherent limitation of JSONP). In all non-JSONP cases, jQuery offers zero assistance.

For modern browsers, all the work is delegated to server code. The browser does everything necessary client-side for you. In the most basic case, your code for a cross-origin AJAX request in a modern browser is identical to a same-origin AJAX request when using jQuery's `ajax()` API method, or when directly using the web API's `XMLHttpRequest` transport, and even with the Fetch API. So, I won't bother showing that here.

CORS requests can be divided up into two distinct types: simple and non-simple. Simple requests consist of GET, HEAD, and POST requests, with a Content-Type of "text/plain" or "application/x-www-form-urlencoded". Non-standard headers, such as "X-" headers, are not allowed in "simple" requests. These CORS requests are sent by the browser with an `Origin` header that includes the sending domain. The server must acknowledge that requests from this origin are acceptable. If not, the request fails. Non-simple requests consists of PUT, PATCH, and DELETE requests, as well as other Content-Types, such as "application/json". Also, non-standard headers, as you just learned, will mark a CORS request as "non-simple." In fact, even a GET or POST request can be non-simple if it, for example, contains non-standard request headers.

[35] http://security.stackexchange.com/a/23439
[36] www.w3.org/TR/cors/
[37] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

A non-simple CORS request must be "preflighted" by the browser. A preflight is an OPTIONS request sent by the browser before the underlying request is sent. If the server properly acknowledges the preflight, the browser will then send the underlying/original request. Non-simple cross-origin requests, such as PUT or POST/GET requests with an X-header (for example) could not be sent from a browser pre-CORS specification. So, for these types of requests, the concept of preflighting was written into the specification to ensure servers do not receive these types of non-simple cross-origin browser-based requests without explicitly opting in. In other words, if you don't want to allow these types of requests, you don't have to make any changes to your server. The preflight request that the browser sends first will fail, and the browser will never send the underlying request.

It is also important to know that cookies are *not* sent by default with cross-origin AJAX requests. You must set the `withCredentials` flag on the `XMLHttpRequest` transport. For example:

```
$.ajax('http://someotherdomain.com', {
  method: 'POST',
  contentType: 'text/plain',
  data: 'sometext',
  beforeSend: function(xmlHttpRequest) {
    xmlHttpRequest.withCredentials = true;
  }
});
```

jQuery offers somewhat of a leaky abstraction here. We must set the `withCredentials` property on the underlying `xmlHttpRequest` that jQuery manages. This can also be accomplished by adding a withCredentials property with a value of true to an xhrFields settings object. This is mentioned in the ajax method documentation, but it may be difficult to locate unless you know exactly where to look. The web API route is familiar, and, as expected, we must set the `withCredentials` flag in order to ensure cookies are sent to our server endpoint:

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://someotherdomain.com');
xhr.withCredentials = true;
xhr.setRequestHeader('Content-Type', 'text/plain');
xhr.send('sometext');
```

The Fetch API makes sending credentials with cross-origin AJAX requests simpler:

```
fetch('http://someotherdomain.com', {
  method: 'POST',
  headers: {
    'Content-Type': 'text/plain'
  },
  credentials: 'include'
});
```

The `credentials` option used there ensures that any credentials, such as cookies, are sent along with the CORS request. Note that even for same-origin requests, `fetch` does not send cookies to the server endpoint by default. For same-origin requests, you must include a `credentials: 'same-origin'` option to ensure `fetch` sends cookies with the request. The default value of the `credentials` option is "omit", which is why `fetch` does not send cookies with any request by default.

jQuery actually becomes a headache to deal with when we need to send a cross-domain AJAX request in IE8 or IE9. If you're using jQuery for this purpose, you are truly trying to fit a square peg into a round hole. To understand why jQuery is a poor fit for cross-origin requests in IE9 and IE8, it's important to consider a couple low-level points:

1. Cross-origin AJAX requests in IE8 and IE9 can only be sent using the IE-proprietary `XDomainRequest` transport. I'll save the rant for why this was such a huge mistake by the IE development team for another book. Regardless, `XDomainRequest` is a stripped-down version of `XMLHttpRequest` and it *must* be used when making cross-origin AJAX requests in IE8 and IE9. There are significant restrictions restrictions imposed on this transport,[38] such as an inability to send anything other than POST and GET requests, and the lack of API methods to set request headers or access response headers.

2. jQuery's `ajax()` method (and all associated aliases) are just wrappers for `XMLHttpRequest`. It has a hard dependency on `XMLHttpRequest`. I mentioned this earlier in the chapter, but it's useful to point it out here again, given the context.

So, you need to use `XDomainRequest` to send the cross-origin request in IE8/9, but `jQuery.ajax()` is hard-coded to use `XMLHttpRequest`. That's a problem, and resolving it in the context of jQuery is not easy. Luckily, for those dead-set on using jQuery for this type of call, there are a few plug-ins that will "fix" jQuery in this regard. Essentially, the plug-ins must override jQuery's AJAX request sending/handling logic via the `$.ajaxTransport()` method.

Instead of wrestling with jQuery when attempting to send cross-origin AJAX requests in older browsers, stick with the web API. The following code demonstrates a simple way to determine whether you need to use `XDomainRequest` instead of `XMLHttpRequest` (use only if needed):

```
1  if (new XMLHttpRequest().withCredentials === undefined) {
2      var xdr = new XDomainRequest();
3      xdr.open('POST', 'http://someotherdomain.com');
4      xdr.send('sometext');
5  }
```

The native web not only provides a reasonable API for initiating AJAX requests, both with `XMLHttpRequest` and even more so with `fetch`, it sometimes is even *more* intuitive than jQuery in this context, especially when sending some cross-origin AJAX requests.

[38] http://blogs.msdn.com/b/ieinternals/archive/2010/05/13/xdomainrequest-restrictions-limitations-and-workarounds.aspx