AJAX = Asynchronous JavaScript and XML  
  
Ajax is a group of interrelated Web development techniques used on the client-side to create asynchronous Web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. Data can be retrieved using the XMLHttpRequest object. Despite the name, the use of XML is not required (JSON is often used), and the requests do not need to be asynchronous.

Ajax is not a single technology, but a group of technologies. HTML and CSS can be used in combination to mark up and style information. The DOM is accessed with JavaScript to dynamically display – and allow the user to interact with – the information presented. JavaScript and the XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads.

The term *Ajax* has come to represent a broad group of Web technologies that can be used to implement a Web application that communicates with a server in the background, without interfering with the current state of the page. In the article that coined the term Ajax, Jesse James Garrett explained that the following technologies are incorporated:

* HTML and CSS for presentation
* The Document Object Model (DOM) for dynamic display of and interaction with data
* XML for the interchange of data, and XSLT for its manipulation
* The XMLHttpRequest object for asynchronous communication
* JavaScript to bring these technologies together

Since then, however, there have been a number of developments in the technologies used in an Ajax application, and the definition of the term Ajax. XML is not required for data interchange and, therefore, XSLT is not required for the manipulation of data. JavaScript Object Notation (JSON) is often used as an alternative format for data interchange,[10] although other formats such as preformatted HTML or plain text can also be used.

Before you continue you should have a basic understanding of the following:

* HTML
* CSS
* JavaScript / DOM

In the early 1990s, most Web sites were based on complete HTML pages. Each user action required that a complete page be loaded from the server. This process was inefficient, as reflected by the user experience: all page content disappeared, then reappeared. Each time the browser reloaded a page because of a partial change, all of the content had to be re-sent, even though only some of the information had changed. This placed additional load on the server and used excessive bandwidth.

In 1996, the iframe tag was introduced by Internet Explorer to load or to fetch content asynchronously.

In 1998, Microsoft Outlook Web App team implemented the first component XMLHTTP by client script.

In 1999, Microsoft used its iframe technology to dynamically update the news stories and stock quotes on the default page for Internet Explorer,[5] and created the XMLHTTP ActiveX control in Internet Explorer 5, which was later adopted by Mozilla, Safari, Opera and other browsers as the XMLHttpRequest JavaScript object.[6] Microsoft has adopted the native XMLHttpRequest model as of Internet Explorer 7, though the ActiveX version is still supported. The utility of background [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests to the server and asynchronous Web technologies remained fairly obscure until it started appearing in full scale online applications such as Outlook Web App (2000)[7] and Oddpost (2002).

Google made a wide deployment of standards-compliant, cross browser Ajax with Gmail (2004) and Google Maps (2005).[8] In October 2004 Kayak.com's public beta release was among the first large-scale e-commerce uses of what their developers at that time called "the xml [http](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) thing".[9]

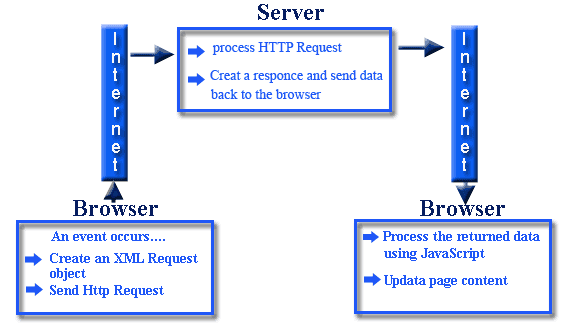
The term "Ajax" was publicly stated on 18 February 2005 by Jesse James Garrett in an article titled **"Ajax: A New Approach to Web Applications",** based on techniques used on Google pages.[3]

On 5 April 2006, the World Wide Web Consortium (W3C) released the first draft specification for the XMLHttpRequest object in an attempt to create an official Web standard.[8]

[http](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)://en.wikipedia.org/wiki/Ajax\_%28programming%29

Ajax is a collection of several technologies aiming to provide a better user experience compared to traditional web applications. End to end implementation of Ajax includes HTML, CSS, DOM, JavaScript, a Server Side Language, and XMLHttpRequest which is also called as XHR.

In traditional web applications, the browser sends a request to server, server processes the data, sends some data back to the browser and then it is rendered by the browser. But meanwhile, when server is processing, the user has to wait. This does not provide the user with a good experience. Ajax helps to get rid of the issue. It makes the user's interaction with the application asynchronous, greatly improving the user's experience.



Creative Commons image from w3reference.com

Ajax stands for Asynchronous JavaScript and XML, but JSON can replace XML when working with Ajax.

[**HTTP**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)

The dream behind the Web is of a common information space in which we communicate by sharing information. Its universality is essential: the fact that a hypertext link can point to anything, be it personal, local or global, be it draft or highly polished.

Tim Berners-Lee, *The World Wide Web: A very short personal*[*history*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2567)

The *Hypertext Transfer Protocol* is the mechanism through which data is requested and provided on the World Wide Web. This chapter describes the protocol in more detail and explains the way browser JavaScript has access to it.

**The protocol**

If you type *eloquentjavascript.net/17\_*[*http*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)*.html* into your browser’s address bar, the browser first looks up the address of the server associated with *eloquentjavascript.net* and tries to open a TCP connection to it on port 80, the default port for [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) traffic. If the server exists and accepts the connection, the browser sends something like this:

GET /17\_[http](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781).html [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1 Host: eloquentjavascript.net User-Agent: Your browser's name

Then the server responds, through that same connection.

[HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1 200 OK Content-Length: 65585 Content-Type: text/html Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT <!doctype html> ... the rest of the document

The browser then takes the part of the response after the blank line and displays it as an HTML document.

The information sent by the client is called the *request*. It starts with this line:

GET /17\_[http](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781).html [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1

The first word is the *method* of the request. GET means that we want to *get* the specified resource. Other common methods are DELETE to delete a resource, PUT to replace it, and POST to send information to it. Note that the server is not obliged to carry out every request it gets. If you walk up to a random website and tell it to DELETE its main page, it’ll probably refuse.

The part after the method name is the path of the resource the request applies to. In the simplest case, a resource is simply a file on the server, but the protocol doesn’t require it to be. A resource may be anything that can be transferred *as if* it is a file. Many servers generate the responses they produce on the fly. For example, if you open [*twitter.com/marijnjh*](http://twitter.com/marijnjh), the server looks in its database for a user named *marijnjh*, and if it finds one, it will generate a profile page for that user.

After the resource path, the first line of the request mentions [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1 to indicate the version of the [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) protocol it is using.

The server’s response will start with a version as well, followed by the status of the response, first as a three-digit status code and then as a human-readable string.

[HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1 200 OK

Status codes starting with a 2 indicate that the request succeeded. Codes starting with 4 mean there was something wrong with the request. 404 is probably the most famous [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) status code—it means that the resource that was requested could not be found. Codes that start with 5 mean an error happened on the server and the request is not to blame.

The first line of a request or response may be followed by any number of [*headers*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782). These are lines in the form “name: value” that specify extra information about the request or response. These [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) were part of the example response:

Content-Length: 65585 Content-Type: text/html Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT

This tells us the size and type of the response document. In this case, it is an HTML document of 65,585 bytes. It also tells us when that document was last modified.

For the most part, a client or server decides which [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) to include in a request or response, though a few [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) are required. For example, the Host header, which specifies the hostname, should be included in a request because a server might be serving multiple hostnames on a single IP address, and without that header, the server won’t know which host the client is trying to talk to.

After the [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782), both requests and responses may include a blank line followed by a *body*, which contains the data being sent. GET and DELETE requests don’t send along any data, but PUT and POST requests do. Similarly, some response types, such as error responses, do not require a body.

[**Browsers and HTTP**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2783)

As we saw in the example, a browser will make a request when we enter a URL in its address bar. When the resulting HTML page references other files, such as images and JavaScript files, those are also fetched.

A moderately complicated website can easily include anywhere from 10 to 200 resources. To be able to fetch those quickly, browsers will make several requests simultaneously, rather than waiting for the responses one at a time. Such documents are always fetched using GET requests.

HTML pages may include *forms*, which allow the user to fill out information and send it to the server. This is an example of a form:

edit & run code by clicking it

<form method="GET" action="example/message.html"> <p>Name: <input type="text" name="name"></p> <p>Message:<br><textarea name="message"></textarea></p> <p><button type="submit">Send</button></p> </form>

This code describes a form with two fields: a small one asking for a name and a larger one to write a message in. When you click the Send button, the information in those fields will be encoded into a *query string*. When the <form> element’s method attribute is GET (or is omitted), that query string is tacked onto the action URL, and the browser makes a GET request to that URL.

GET /example/message.html?name=Jean&message=Yes%3F [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1

The start of a query string is indicated by a question mark. After that follow pairs of names and values, corresponding to the name attribute on the form field elements and the content of those elements, respectively. An ampersand character (&) is used to separate the pairs.

The actual message encoded in the previous URL is “Yes?”, even though the question mark is replaced by a strange code. Some characters in query strings must be escaped. The question mark, represented as %3F, is one of those. There seems to be an unwritten rule that every format needs its own way of escaping characters. This one, called *URL encoding*, uses a percent sign followed by two hexadecimal digits that encode the character code. In this case, 3F, which is 63 in decimal notation, is the code of a question mark character. JavaScript provides the encodeURIComponent and decodeURIComponent functions to encode and decode this format.

console.log(encodeURIComponent("Hello & goodbye")); // → Hello%20%26%20goodbye console.log(decodeURIComponent("Hello%20%26%20goodbye")); // → Hello & goodbye

If we change the method attribute of the HTML form in the example we saw earlier to POST, the [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) request made to submit the form will use the POST method and put the query string in body of the request, rather than adding it to the URL.

POST /example/message.html [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)/1.1 Content-length: 24 Content-type: application/x-www-form-urlencoded name=Jean&message=Yes%3F

By convention, the GET method is used for requests that do not have side effects, such as doing a search. Requests that change something on the server, such as creating a new account or posting a message, should be expressed with other methods, such as POST. Client-side software, such as a browser, knows that it shouldn’t blindly make POST requests but will often implicitly make GET requests—for example, to prefetch a resource it believes the user will soon need.

The [next chapter](http://eloquentjavascript.net/18_forms.html#forms) will return to forms and talk about how we can script them with JavaScript.

**XMLHttpRequest**

The interface through which browser JavaScript can make [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests is called XMLHttpRequest (note the inconsistent capitalization). It was designed by Microsoft, for its Internet Explorer browser, in the late 1990s. During this time, the XML file format was *very* popular in the world of business software—a world where Microsoft has always been at home. In fact, it was so popular that the acronym XML was tacked onto the front of the name of an interface for [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781), which is in no way tied to XML.

The name isn’t completely nonsensical, though. The interface allows you to parse response documents as XML if you want. Conflating two distinct concepts (making a request and parsing the response) into a single thing is terrible design, of course, but so it goes.

When the XMLHttpRequest interface was added to Internet Explorer, it allowed people to do things with JavaScript that had been very hard before. For example, websites started showing lists of suggestions when the user was typing something into a text field. The script would send the text to the server over [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) as the user typed. The server, which had some database of possible inputs, would match the database entries against the partial input and send back possible completions to show the user. This was considered spectacular—people were used to waiting for a full page reload for every interaction with a website.

The other significant browser at that time, Mozilla (later Firefox), did not want to be left behind. To allow people to do similarly neat things in *its* browser, Mozilla copied the interface, including the bogus name. The next generation of browsers followed this example, and today XMLHttpRequest is a de facto standard interface.

[**Sending a request**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2785)

To make a simple request, we create a request object with the XMLHttpRequest constructor and call its open and send methods.

var req = new XMLHttpRequest(); req.open("GET", "example/data.txt", false); req.send(null); console.log(req.responseText); // → This is the content of data.txt

The open method configures the request. In this case, we choose to make a GET request for the *example/data.txt* file. URLs that don’t start with a protocol name (such as [*http*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)*:*) are relative, which means that they are interpreted relative to the current document. When they start with a slash (/), they replace the current path, which is the part after the server name. When they do not, the part of the current path up to and including its last slash character is put in front of the relative URL.

After opening the request, we can send it with the send method. The argument to send is the request body. For GET requests, we can pass null. If the third argument to open was false, send will return only after the response to our request was received. We can read the request object’s responseText property to get the response body.

The other information included in the response can also be extracted from this object. The status code is accessible through the status property, and the human-readable status text is accessible throughstatusText. [Headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) can be read with getResponseHeader.

var req = new XMLHttpRequest(); req.open("GET", "example/data.txt", false); req.send(null); console.log(req.status, req.statusText); // → 200 OK console.log(req.getResponseHeader("content-type")); // → text/plain

Header names are case-insensitive. They are usually written with a capital letter at the start of each word, such as “Content-Type”, but “content-type” and “cOnTeNt-TyPe” refer to the same header.

The browser will automatically add some request [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782), such as “Host” and those needed for the server to figure out the size of the body. But you can add your own [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) with the setRequestHeader method. This is needed only for advanced uses and requires the cooperation of the server you are talking to—a server is free to ignore [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) it does not know how to handle.

[**Asynchronous Requests**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2786)

In the examples we saw, the request has finished when the call to send returns. This is convenient because it means properties such as responseText are available immediately. But it also mean that our program is suspended as long as the browser and server are communicating. When the connection is bad, the server is slow, or the file is big, that might take quite a while. Worse, because no event handlers can fire while our program is suspended, the whole document will become unresponsive.

If we pass true as the third argument to open, the request is *asynchronous*. This means that when we call send, the only thing that happens right away is that the request is scheduled to be sent. Our program can continue, and the browser will take care of the sending and receiving of data in the background.

But as long as the request is running, we won’t be able to access the response. We need a mechanism that will notify us when the data is available.

For this, we must listen for the "load" event on the request object.

var req = new XMLHttpRequest(); req.open("GET", "example/data.txt", true); req.addEventListener("load", function() { console.log("Done:", req.status); }); req.send(null);

This forces us to use an asynchronous style of programming, wrapping the things that have to be done after the request in a function and arranging for that to be called at the appropriate time. We will come back to this [later](http://eloquentjavascript.net/17_http.html#promises).(Promises)

[**Fetching XML Data**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2787)

When the resource retrieved by an XMLHttpRequest object is an XML document, the object’s responseXML property will hold a parsed representation of this document. This representation works much like the DOM discussed in [Chapter 13](http://eloquentjavascript.net/13_dom.html#dom), except that it doesn’t have HTML-specific functionality like the style property. The object that responseXML holds corresponds to the document object. Its documentElement property refers to the outer tag of the XML document. In the following document (*example/fruit.xml*), that would would be the <fruits> tag:

<fruits> <fruit name="banana" color="yellow"/> <fruit name="lemon" color="yellow"/> <fruit name="cherry" color="red"/> </fruits>

We can retrieve such a file like this:

var req = new XMLHttpRequest(); req.open("GET", "example/fruit.xml", false); req.send(null); console.log(req.responseXML.querySelectorAll("fruit").length); // → 3

XML documents can be used to exchange structured information with the server. Their form—tags nested inside other tags—lends itself well to storing most types of data, or at least better than flat text files. The DOM interface is rather clumsy for extracting information, though, and XML documents tend to be verbose. It is often a better idea to communicate using JSON data, which is easier to read and write, both for programs and for humans.

var req = new XMLHttpRequest(); req.open("GET", "example/fruit.json", false); req.send(null); console.log(JSON.parse(req.responseText)); // → {banana: "yellow", lemon: "yellow", cherry: "red"}

[**HTTP**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)**sandboxing**

Making [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests in web page scripts once again raises concerns about [security](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2791). The person who controls the script might not have the same interests as the person on whose computer it is running. More specifically, if I visit *themafia.org*, I do not want its scripts to be able to make a request to *mybank.com*, using identifying information from my browser, with [instructions](http://www.webmastercert.com/moodle/mod/resource/view.php?r=681) to transfer all my money to some random mafia account.

It is possible for websites to protect themselves against such attacks, but that requires effort, and many websites fail to do it. For this reason, browsers protect us by disallowing scripts to make [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests to other *domains* (names such as *themafia.org* and *mybank.com*).

This can be an annoying problem when building systems that want to access several domains for legitimate reasons. Fortunately, servers can include a header like this in their response to explicitly indicate to browsers that it is okay for the request to come from other domains:

Access-Control-Allow-Origin: \*

[**Abstracting requests**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2788)

In [Chapter 10](http://eloquentjavascript.net/10_modules.html#amd), in our implementation of the AMD module system, we used a hypothetical function called backgroundReadFile. It took a filename and a function and called that function with the contents of the file when it had finished fetching it. Here’s a simple implementation of that function:

function backgroundReadFile(url, callback) { var req = new XMLHttpRequest(); req.open("GET", url, true); req.addEventListener("load", function() { if (req.status < 400) callback(req.responseText); }); req.send(null); }

This simple abstraction makes it easier to use XMLHttpRequest for simple GET requests. If you are writing a program that has to make [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests, it is a good idea to use a helper function so that you don’t end up repeating the ugly XMLHttpRequest pattern all through your code.

The function argument’s name, callback, is a term that is often used to describe functions like this. A callback function is given to other code to provide that code with a way to “call us back” later.

It is not hard to write an [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) utility function, tailored to what your application is doing. The previous one does only GET requests and doesn’t give us control over the [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) or the request body. You could write another variant for POST requests or a more generic one that supports various kinds of requests. Many JavaScript libraries also provide wrappers for XMLHttpRequest.

The main problem with the previous wrapper is its handling of failure. When the request returns a status code that indicates an error (400 and up), it does nothing. This might be okay, in some circumstances, but imagine we put a “loading” indicator on the page to indicate that we are fetching information. If the request fails because the server crashed or the connection is briefly interrupted, the page will just sit there, misleadingly looking like it is doing something. The user will wait for a while, get impatient, and consider the site uselessly flaky.

We should also have an option to be notified when the request fails so that we can take appropriate action. For example, we could remove the “loading” message and inform the user that something went wrong.

Error handling in asynchronous code is even trickier than error handling in synchronous code. Because we often need to defer part of our work, putting it in a callback function, the scope of a try block becomes meaningless. In the following code, the exception will *not* be caught because the call to backgroundReadFile returns immediately. Control then leaves the try block, and the function it was given won’t be called until later.

try { backgroundReadFile("example/data.txt", function(text) { if (text != "expected") throw new Error("That was unexpected"); }); } catch (e) { console.log("Hello from the catch block"); }

To handle failing requests, we have to allow an additional function to be passed to our wrapper and call that when a request goes wrong. Alternatively, we can use the convention that if the request fails, an additional argument describing the problem is passed to the regular callback function. Here’s an example:

function getURL(url, callback) { var req = new XMLHttpRequest(); req.open("GET", url, true); req.addEventListener("load", function() { if (req.status < 400) callback(req.responseText); else callback(null, new Error("Request failed: " + req.statusText)); }); req.addEventListener("error", function() { callback(null, new Error("Network error")); }); req.send(null); }

We have added a handler for the "error" event, which will be signaled when the request fails entirely. We also call the callback function with an error argument when the request completes with a status code that indicates an error.

Code using getURL must then check whether an error was given and, if it finds one, handle it.

getURL("data/nonsense.txt", function(content, error) { if (error != null) console.log("Failed to fetch nonsense.txt: " + error); else console.log("nonsense.txt: " + content); });

This does not help when it comes to exceptions. When chaining several asynchronous actions together, an exception at any point of the chain will still (unless you wrap each handling function in its own try/catchblock) land at the top level and abort your chain of actions.

[**Promises**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789)

For complicated projects, writing asynchronous code in plain callback style is hard to do correctly. It is easy to forget to check for an error or to allow an unexpected exception to cut the program short in a crude way. Additionally, arranging for correct error handling when the error has to flow through multiple callback functions and catch blocks is tedious.

There have been a lot of attempts to solve this with extra abstractions. One of the more successful ones is called [*promises*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789). [Promises](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789) wrap an asynchronous action in an object, which can be passed around and told to do certain things when the action finishes or fails. This interface is set to become part of the next version of the JavaScript language but can already be used as a library.

The interface for [promises](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789) isn’t entirely intuitive, but it is powerful. This chapter will only roughly describe it. You can find a more thorough treatment at [*www.promisejs.org*](https://www.promisejs.org/).

To create a promise object, we call the Promise constructor, giving it a function that initializes the asynchronous action. The constructor calls that function, passing it two arguments, which are themselves functions. The first should be called when the action finishes successfully, and the second should be called when it fails.

Once again, here is our wrapper for GET requests, this time returning a promise. We’ll simply call it get this time.

function get(url) { return new Promise(function(succeed, fail) { var req = new XMLHttpRequest(); req.open("GET", url, true); req.addEventListener("load", function() { if (req.status < 400) succeed(req.responseText); else fail(new Error("Request failed: " + req.statusText)); }); req.addEventListener("error", function() { fail(new Error("Network error")); }); req.send(null); }); }

Note that the interface to the function itself is now a lot simpler. You give it a URL, and it returns a promise. That promise acts as a *handle* to the request’s outcome. It has a then method that you can call with two functions: one to handle success and one to handle failure.

get("example/data.txt").then(function(text) { console.log("data.txt: " + text); }, function(error) { console.log("Failed to fetch data.txt: " + error); });

So far, this is just another way to express the same thing we already expressed. It is only when you need to chain actions together that [promises](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789) make a significant difference.

Calling then produces a new promise, whose result (the value passed to success handlers) depends on the return value of the first function we passed to then. This function may return another promise to indicate that more asynchronous work is being done. In this case, the promise returned by then itself will wait for the promise returned by the handler function, succeeding or failing with the same value when it is resolved. When the handler function returns a nonpromise value, the promise returned by then immediately succeeds with that value as its result.

This means you can use then to transform the result of a promise. For example, this returns a promise whose result is the content of the given URL, parsed as JSON:

function getJSON(url) { return get(url).then(JSON.parse); }

That last call to then did not specify a failure handler. This is allowed. The error will be passed to the promise returned by then, which is exactly what we want—getJSON does not know what to do when something goes wrong, but hopefully its caller does.

As an example that shows the use of [promises](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789), we will build a program that fetches a number of JSON files from the server and, while it is doing that, shows the word *loading*. The JSON files contain information about people, with links to files that represent other people in properties such as father, mother, or spouse.

We want to get the name of the mother of the spouse of *example/bert.json*. And if something goes wrong, we want to remove the *loading* text and show an error message instead. Here is how that might be done with [promises](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789):

<script> function showMessage(msg) { var elt = document.createElement("div"); elt.textContent = msg; return document.body.appendChild(elt); } var loading = showMessage("Loading..."); getJSON("example/bert.json").then(function(bert) { return getJSON(bert.spouse); }).then(function(spouse) { return getJSON(spouse.mother); }).then(function(mother) { showMessage("The name is " + mother.name); }).catch(function(error) { showMessage(String(error)); }).then(function() { document.body.removeChild(loading); }); </script>

The resulting program is relatively compact and readable. The catch method is similar to then, except that it only expects a failure handler and will pass through the result unmodified in case of success. Much like with the catch clause for the try statement, control will continue as normal after the failure is caught. That way, the final then, which removes the loading message, is always executed, even if something went wrong.

You can think of the promise interface as implementing its own language for asynchronous control flow. The extra method calls and function expressions needed to achieve this make the code look somewhat awkward but not remotely as awkward as it would look if we took care of all the error handling ourselves.

[**Appreciating HTTP**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2790)

When building a system that requires communication between a JavaScript program running in the browser (client-side) and a program on a server (server-side), there are several different ways to model this communication.

A commonly used model is that of *remote procedure calls*. In this model, communication follows the patterns of normal function calls, except that the function is actually running on another machine. Calling it involves making a request to the server that includes the function’s name and arguments. The response to that request contains the returned value.

When thinking in terms of remote procedure calls, [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) is just a vehicle for communication, and you will most likely write an abstraction layer that hides it entirely.

Another approach is to build your communication around the concept of resources and [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) methods. Instead of a remote procedure called addUser, you use a PUT request to /users/larry. Instead of encoding that user’s properties in function arguments, you define a document format or use an existing format that represents a user. The body of the PUT request to create a new resource is then simply such a document. A resource is fetched by making a GET request to the resource’s URL (for example, /user/larry), which returns the document representing the resource.

This second approach makes it easier to use some of the features that [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) provides, such as support for caching resources (keeping a copy on the client side). It can also help the coherence of your interface since resources are easier to reason about than a jumble of functions.

[**Security**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2791)**and HTTPS**

Data traveling over the Internet tends to follow a long, dangerous road. To get to its destination, it must hop through anything from coffee-shop Wi-Fi networks to networks controlled by various companies and states. At any point along its route it may be inspected or even modified.

If it is important that something remain secret, such as the password to your email account, or that it arrive at its destination unmodified, such as the account number you transfer money to from your bank’s website, plain [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) is not good enough.

The secure [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) protocol, whose URLs start with *https://*, wraps [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) traffic in a way that makes it harder to read and tamper with. First, the client verifies that the server is who it claims to be by requiring that server to prove that it has a cryptographic certificate issued by a certificate authority that the browser recognizes. Next, all data going over the connection is encrypted in a way that should prevent eavesdropping and tampering.

Thus, when it works right, HTTPS prevents both the someone impersonating the website you were trying to talk to and the someone snooping on your communication. It is not perfect, and there have been various incidents where HTTPS failed because of forged or stolen certificates and broken software. Still, plain [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) is trivial to mess with, whereas breaking HTTPS requires the kind of effort that only states or sophisticated criminal organizations can hope to make.

[**Summary**](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2792)

In this chapter, we saw that [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) is a protocol for accessing resources over the Internet. A *client* sends a request, which contains a method (usually GET) and a path that identifies a resource. The *server* then decides what to do with the request and responds with a status code and a response body. Both requests and responses may contain [headers](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2782) that provide additional information.

Browsers make GET requests to fetch the resources needed to display a web page. A web page may also contain forms, which allow information entered by the user to be sent along in the request made when the form is submitted.

The interface through which browser JavaScript can make [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) requests is called XMLHttpRequest. You can usually ignore the “XML” part of that name (but you still have to type it). There are two ways in which it can be used—synchronous, which blocks everything until the request finishes, and asynchronous, which requires an event handler to notice that the response came in. In almost all cases, asynchronous is preferable. Making a request looks like this:

var req = new XMLHttpRequest(); req.open("GET", "example/data.txt", true); req.addEventListener("load", function() { console.log(req.status); }); req.send(null);

Asynchronous programming is tricky. [*Promises*](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2789) are an interface that makes it slightly easier by helping route error conditions and exceptions to the right handler and by abstracting away some of the more repetitive and error-prone elements in this style of programming.

You will see later in the course that Javascript Libraries shield programmers from most of the low-level programming required to implement Ajax. While important to have a basic understanding of [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) and the request model, most of the time you will be working with a library that already contains the code to perform the low-level [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) communication for you. We will work with one of those libraries later in the course.

**Introduction to ToolKit**

Many different Ajax toolkits exist to help programmers create Ajax styled widgets and applications, without having to reinvent the wheel each time they start a new project. Most toolkits render the Javascript code needed to perform a particular task, and most are geared for a specific programming language. For example, there are toolkits for Ajax that concentrate on implementing the techniques using the Java programming language, the Php language, and Windows based languages.

Since this is not a class on implementing Ajax using one of those specific programming languages, we are using a small toolkit that is geared for use directly on a Web page, using HTML, that doesn't require knowledge of Java, Php or any other programming language.

The Dynamic Drive collection of Ajax Routines is perfect for our application. The site is at [http](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781)://www.dynamicdrive.com, and can also be used as a huge resource for learning all kinds of programming techniques, in addition to learning Ajax.

The folks at Dynamic Drive have written the low-level Javascript code that implements several different Ajax techniques. We'll concentrate on three of those –

* the first is basic [HTTP](http://www.webmastercert.com/moodle/mod/resource/view.php?r=2781) Requests to pull content from a file and embed it into a Web page, without the browser having to refresh and reload the content.
* The second is using Ajax to create tabbed windows that can pull content from files into the windows.
* The third is a technique that creates a slideshow effect on a page, by creating page numbers that the user can click through to display new content on the page, without any browser re-loading.

Armed with the toolkit and these effects, you'll be able to add a lot of cool functionality to your HTML projects!

**Tab Content**

http://www.dynamicdrive.com/dynamicindex17/ajaxtabscontent/