AJAX stands for "Asynchronous JavaScript And XML" but the name is a misleading. These days, most uses of AJAX do not involve XML. XML is a format for sending structured data. JSON, i.e., JavaScript Object Notation, is used instead of XML in most applications.

AJAX is a way for JavaScript code to request information from a web server. To do this, JavaScript in modern browsers provides a method **fetch()**. That's what we'll show here.

A Problem: Waiting for Web Servers

A web server may take several seconds or more to respond. It might never respond. If the AJAX function waited for the web server response, your web page would lock up and not respond to any user input until that response came. Things would be even worse if you need to make half a dozen queries to a web server to build a web page.

The Solution: Promises

The basic idea to allow web pages to continue to respond while waiting for answers is **asynchrony**. That means saving the code that you want to run when the results are finally returned some place where it can be run later. Meanwhile, the rest of your web page code that responds to button clicks and scrolling and so on, continues to run.

The way this is done in modern browser libraries is with a JavaScript object called a **promise**. You used to need a library like jQuery to use promises, but now they are available in JavaScript directly.

The JavaScript function to get data from a web server is **fetch()**. It takes a URL and an optional object with additional information. When you call **fetch()**, it

- sends a request to get or update data to the web server
- creates and immediately return a promise object

The rest of your code continues to run as normal. It does not wait for the server response.

When the response comes from the server, the browser runs any **success function** that has been attached to the promise.

If a server error occurs or the server doesn't respond with some period, the browser instead runs any **failure function** that has been attached to the promise.

You attach a success function to a promise by writing *promise***.then(***function***)**. The function will be called with an object containing the response sent by the server.

You attach a failure function to a promise by writing *promise***.catch(***function***)**. The function will be called with an object containing the error information sent by the server.

Here's a simple but not very useful example. Assume our server returns an array of user names when called with the URL **/users**. Then we could test that this is working with this:

```
fetch( "/users").then(function(response) {
  console.log('Request succeeded: ' + response.statusText);
}).catch(function(error) {
  console.log( "Request failed: " + error.message );
});
```

When this code is executed, it sends the URL **/users** to the server, and immediately returns a promise object.

At some point in the future, perhaps many seconds later, the browser receives a response. If the response says the request succeeded, the function in **then()** is called. If the response says the request failed, e.g., the URL is unknown, or authentication is needed, the function in **catch()** is called.

Promise chains

The above code isn't very useful. We get a response object, but not the actual data we want. To do that, you need to call a method on the response object:

- *response***.blob()** will extract the raw data sent by the server; this could be handy if you are fetching an image
- *response***.text()** will extract the data sent by the server in text string form; this could be handy if you are getting HTML;
- *response***.json()** will extract the text string and use **JSON.stringify()** on it; this is often what we want to do

These methods return promises, like **fetch()**. That's because extracting and/or parsing data from a stream of bits is not a fast process.

Since they return promises, we can't use the return value directly. Instead, we need to add a **then()** to process the return value when it's ready, and/or a **catch()** to handle errors.

Fortunately, promises have a very important property:

> If a **then()** function returns a new promise, that new promise is added to the previous promise.

This means that you can simply chain a sequence of promises together. Furthermore, you don't need to write separate **catch()** functions. One **catch()** at the end of the chain will get an error that occurs with any promise in the chain. So we can get our list of users as a JavaScript array with code like this

```
fetch( "/users").then(function(response) {
  if (response.ok) {
  // add a new promise to the chain
    return response.json();
  }
  // signal a server error to the chain
  throw new Error(response.statusText);
}).then(function(json) {
  // do something with the JSON
  // note that this does not add a new promise
  console.log(json);   //
}).catch(function(error) {
```

```
  // called when an error occurs anywhere in the chain
  console.log( "Request failed: " + error.message );
});
```

Since the same pattern would be used anytime we want to get JSON, it's worth defining a **fetchJson()** function for that task:

```
function fetchJson(url, init) {
  return fetch(url, init).then(function(response) {
    if (response.ok) {
      return response.json();
    }
    throw new Error(response.statusText);
  });
}
```

The second **init** argument is an optional JSON object that **fetch()** takes. It is where you can put extra header information, such as an API key, or specify that you want to use POST not GET as the HTTP method. An example of an **init** object appears in the next section.

fetchJson() returns a promise, so it can be used like this:

```
fetchJson('/users').then(function(json) {
  // do something with the JSON
  ...
}).catch(function(error) {
  // do something getting JSON fails
});
```

POSTing data to the server

If you need to send information to a web server to update data on the server, tell **fetch()** to use the HTTP POST method, like this:

```
fetch(url, {
  method: 'POST'
  ...
});
```

The second, optional, argument to **fetch()** is a JavaScript object that can specify many properties of a request.

- what HTTP method to use, e.g., GET or POST
- if POST, what data, e.g., a string with JSON
- special header information, such as the content type, or an API key

For example, suppose that

- there is a web app that manages a calendar of appointments,
- **/appointments/18** is the URL for the list of appointments for the user with ID 18
- posting a title and date will add an appointment with that data to the list of appointments
- the server returns either a **201 Created** status code or a failure status

Then this is how we might use **fetch()** to post an appointment to user 18. The date string is ISO-8601 standard format for 2pm on July 6, 2016.

```
fetch('/appointments/18', {
  method: 'POST',
  body: JSON.stringify({ title: 'meet John', time: '2016-07-06T14:00:00.000Z' }),
  headers: new Headers({
    contentType: 'application/json'
  })
}).then(function (response) {
  if (response.ok) {
    alert( 'Appointment saved' );
  }
  throw new Error(response.statusText);
}).catch(function(error) {
  alert('Appointment not saved: ' + error.message);
});
```

The headers of a request is a set of key-value pairs. The JavaScript **Headers** object makes it easy to construct the headers in the right syntax from a JavaScript object.

Note that the body of the request has to be a string, so we call **JSON.stringify()** to convert our JSON object into string form.

Parallel Promises

Occasionally, you need to make several asynchronous calls, that are all independent, but all must complete before you can combine the results and continue. For example, here are two web services that return data useful for a currency exchange rate web page:

- The URL https://api.fixer.io/latest returns a JSON object with today's currency exchange rates for the Euro, using ISO 4217 codes, like "EUR" for the Euro, and "USD" for the US dollar
- The URL https://openexchangerates.org/api/currencies.json returns a JSON object with the full names of those currency codes.

A web page showing currency data might want both pieces of data before it displayed anything.

If you do this with a promise chain, then each promise has to wait for the previous promise to finish. If there are N such promises, the total time to do the entire chain will be the sum of the times to get responses for each promise. It would be much faster if we could do all N promises in parallel. Then the total time would just be the time of the slowest response.

To do parallel promises, you use **Promise.all()**. This is a function that takes either an array of promise objects, or simply one or more promise objects as arguments. It returns a new promise that is fulfilled when and only when all the promises it was passed are fulfilled. Any **then()** function attached to this promise will get an array of the values received, in the same order as the promises given to **Promise.all()**.

We could define a function that fetches JSON from a list of URLs and returns a promise that waits until all the JSON has been retrieved:

```
function fetchJson(url, init) {
  return fetch(url, init).then(function(response) {
    if (response.ok) {
      return response.json();
    }
    throw new Error(response.statusText)
  });
}

function fetchJsonList(urls, init) {
  return Promise.all(urls.map(url => fetchJson(url, init)));
}
```

Note the optional **init** argument. If we leave it out, **init** will be undefined and ignored by **fetch()**.

We can use **fetchJsonList()** to get our currency exchange data like this:

```
var nameUrl = 'https://openexchangerates.org/api/currencies.json';
var rateUrl = 'https://api.fixer.io/latest';


fetchJsonList([nameUrl, rateUrl]).then(function(values) {
  var nameData = values[0];
  var rateData = values[1];
  ...
})
```

Side note: Modern JavaScript lets you write function parameters that "destructure" argument data. There are many options, but one of them is to write an array as a parameter, when we know we're getting an array with a fixed number of values. So the above could be written even more simply, like this:

```
var nameUrl = 'https://openexchangerates.org/api/currencies.json';
var rateUrl = 'https://api.fixer.io/latest';


fetchJsonList([nameUrl, rateUrl]).then(function([nameData, rateData]) {
  ...
})
```