

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in](#) 

Fetching data from the server

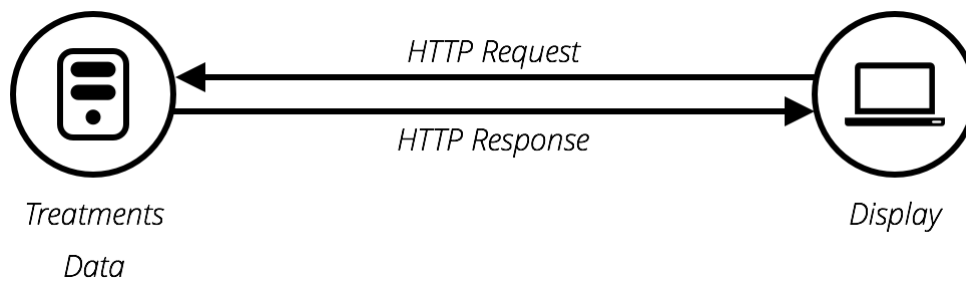
[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page. This seemingly small detail has had a huge impact on the performance and behavior of sites, so in this article we'll explain the concept and look at technologies that make it possible, such as XMLHttpRequest and the Fetch API.

Prerequisites:	JavaScript basics (see first steps , building blocks , JavaScript objects), the basics of Client-side APIs
Objective:	To learn how to fetch data from the server and use it to update the contents of a web page.

What is the problem here?

Originally page loading on the web was simple — you'd send a request for a web site to a server, and as long as nothing went wrong, the assets that made the web page would be downloaded and displayed on your computer.



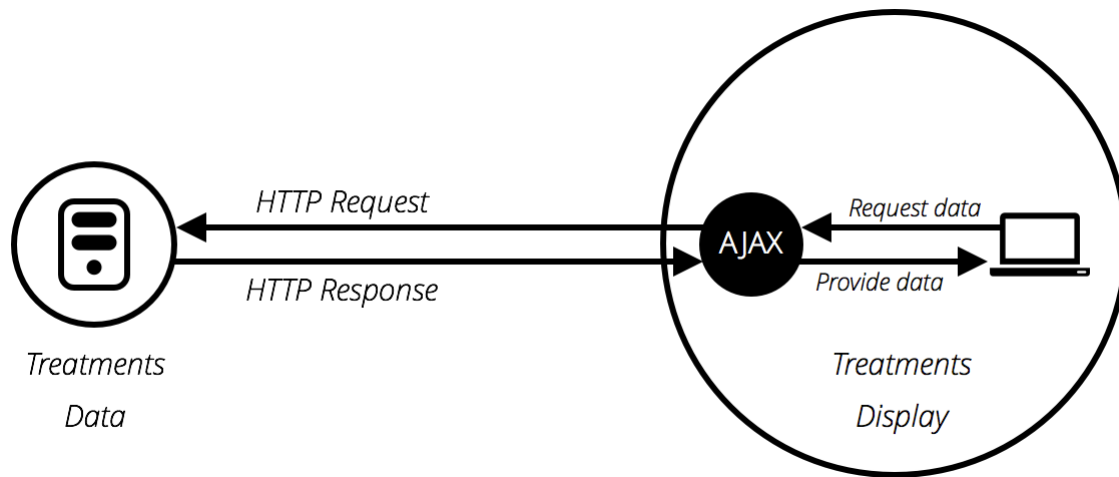
The trouble with this model is that whenever you want to update any part of the page, for example to display a new set of products or load a new page, you've got to load the entire page again. This is extremely wasteful and results in a poor user experience, especially as pages get larger and more complex.

Enter Ajax

This led to the creation of technologies that allow web pages to request small chunks of data (such as `HTML`, `XML`, `JSON`, or plain text) and display them only when needed, helping to solve the problem described above.

This is achieved by using APIs like `XMLHttpRequest` or — more recently — the `Fetch API`. These technologies allow web pages to directly handle making `HTTP` requests for specific resources available on a server, and formatting the resulting data as needed, before it is displayed.

Note: In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to use `XMLHttpRequest` to request XML data. This is not normally the case these days (you'd be more likely to use `XMLHttpRequest` or `Fetch` to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.



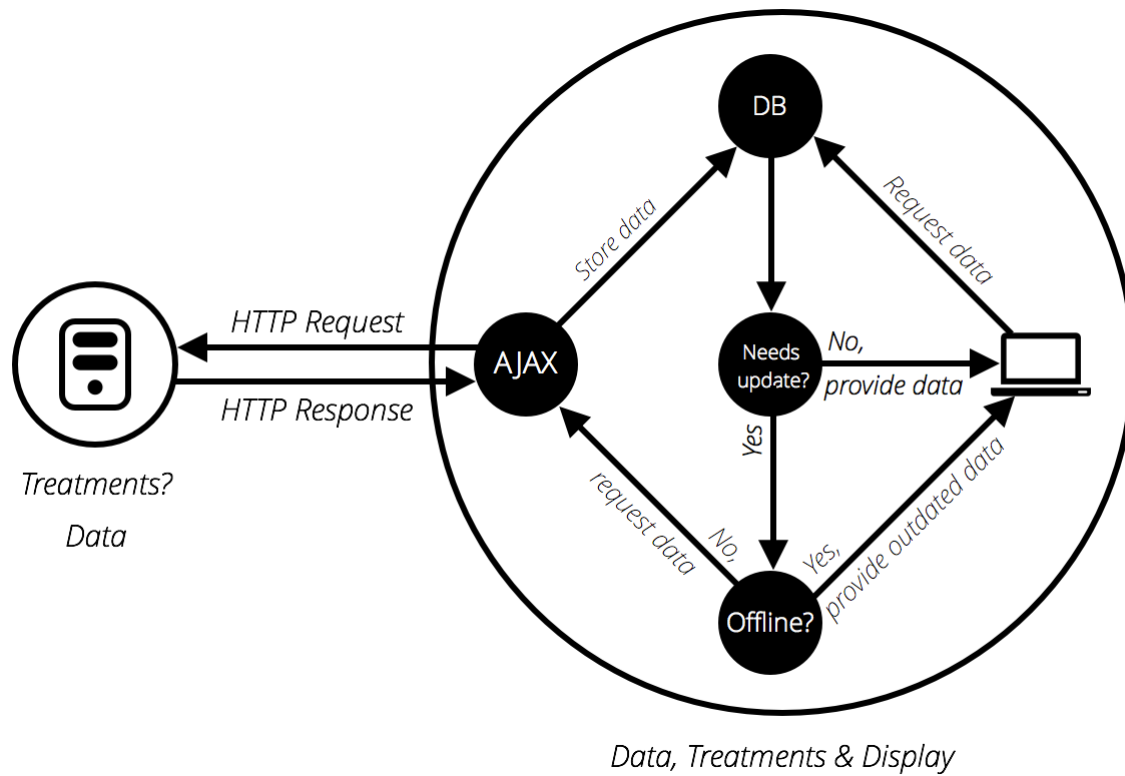
The Ajax model involves using a web API as a proxy to more intelligently request data rather than just having the browser to reload the entire page. Let's think about the significance of this:

1. Go to one of your favorite information-rich sites, like Amazon, YouTube, CNN, etc., and load it.
2. Now search for something, like a new product. The main content will change, but most of the surrounding information, like the header, footer, navigation menu, etc., will stay the same.

This is a really good thing because:

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in developing countries that don't have ubiquitous fast Internet service.

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies when the page is first loaded. The content is only reloaded from the server when it has been updated.



A basic Ajax request

Let's look at how such a request is handled, using both `XMLHttpRequest` and `Fetch`. For these examples, we'll request data out of a few different text files, and use them to populate a content area.

This series of files will act as our fake database; in a real application we'd be more likely to use a server-side language like PHP, Python, or Node to request our data from a database. Here however we want to keep it simple, and concentrate on the client-side part of this.

XMLHttpRequest

`XMLHttpRequest` (which is frequently abbreviated to `XHR`) is a fairly old technology now — it was invented by Microsoft in the late 1990's, and has been standardized across browsers for quite a long time.

1. To begin this example, make a local copy of [ajax-start.html](#) and the four text files — [verse1.txt](#), [verse2.txt](#), [verse3.txt](#), and [verse4.txt](#) — in a new directory on your computer. In this example we will load a different verse of the poem (which you may well recognize) via `XHR` when it's selected in the drop down menu.

2. Just inside the `<script>` element, add the following code. This stores a reference to the `<select>` and `<pre>` elements in variables, and defines an `onchange` event handler function so that when the select's value is changed, its value is passed to an invoked function `updateDisplay()` as a parameter.

```
1 | var verseChoose = document.querySelector('select');
2 | var poemDisplay = document.querySelector('pre');
3 |
4 | verseChoose.onchange = function() {
5 |     var verse = verseChoose.value;
6 |     updateDisplay(verse);
7 | };
```

3. Let's define our `updateDisplay()` function. First of all, put the following beneath your previous code block — this is the empty shell of the function:

```
1 | function updateDisplay(verse) {
2 |
3 | };
```

4. We'll start our function by constructing a relative URL pointing to the text file we want to load, as we'll need it later. The value of the `<select>` element at any time is the same as the text inside the selected `<option>` (unless you specify a different value in a value attribute) — so for example "Verse 1". The corresponding verse text file is "verse1.txt", and is in the same directory as the HTML file, therefore just the file name will do.

However, web servers tend to be case sensitive, and the file name hasn't got a space in it. To convert "Verse 1" to "verse1.txt" we need to convert the V to lower case, remove the space, and add .txt on the end. This can be done with `replace()`, `toLowerCase()`, and simple string concatenation. Add the following lines inside your `updateDisplay()` function:

```
1 | verse = verse.replace(" ", "");
2 | verse = verse.toLowerCase();
3 | var url = verse + '.txt';
```

5. To begin creating an XHR request, you need to create a new request object using the `XMLHttpRequest()` constructor. You can call this object anything you like, but we'll

call it `request` to keep things simple. Add the following below your previous lines:

```
1 | var request = new XMLHttpRequest();
```

6. Next, you need to use the `open()` method to specify what HTTP request method to use to request the resource from the network, and what its URL is. We'll just use the `GET` method here, and set the URL as our `url` variable. Add this below your previous line:

```
1 | request.open('GET', url);
```

7. Next, we'll set the type of response we are expecting — which is defined by the request's `responseType` property — as `text`. This isn't strictly necessary here — XHR returns text by default — but it is a good idea to get into the habit of setting this in case you want to fetch other types of data in the future. Add this next:

```
1 | request.responseType = 'text';
```

8. Fetching a resource from the network is an asynchronous operation, meaning that you've got to wait for that operation to complete (e.g., the resource is returned from the network) before you can then do anything with that response, otherwise an error will be thrown. XHR allows you to handle this using its `onload` event handler — this is run when the `load` event fires (when the response has returned). When this has occurred, the response data will be available in the `response` property of the XHR request object.

Add the following below your last addition. you'll see that inside the `onload` event handler we are setting the `textContent` of the `poemDisplay` (the `<pre>` element) to the value of the `request.response` property.

```
1 | request.onload = function() {  
2 |     poemDisplay.textContent = request.response;  
3 | };
```

9. The above is all setup for the XHR request — it won't actually run until we tell it to, which is done using the `send()` method. Add the following below your previous addition to complete the function:

```
1 | request.send();
```

10. One problem with the example as it stands is that it won't show any of the poem when it first loads. To fix this, add the following two lines at the bottom of your code (just above the closing `</script>` tag) to load verse 1 by default, and make sure the `<select>` element always shows the correct value:

```
1 | updateDisplay('Verse 1');
2 | verseChoose.value = 'Verse 1';
```

Serving your example from a server

Some browsers (including Chrome) will not run XHR requests if you just run the example from a local file. This is because of security restrictions (for more on web security, read [Website security](#)).

To get around this, we need to test the example by running it through a local web server. To find out how to do this, read [How do you set up a local testing server?](#)

Fetch

The Fetch API is basically a modern replacement for XHR — it was introduced in browsers recently to make asynchronous HTTP requests easier to do in JavaScript, both for developers and other APIs that build on top of Fetch.

Let's convert the last example to use Fetch instead!

1. Make a copy of your previous finished example directory. (If you didn't work through the previous exercise, create a new directory, and inside it make copies of [xhr-basic.html](#) and the four text files — [verse1.txt](#), [verse2.txt](#), [verse3.txt](#), and [verse4.txt](#).)
2. Inside the `updateDisplay()` function, find the XHR code:

```
1 | var request = new XMLHttpRequest();
2 | request.open('GET', url);
3 | request.responseType = 'text';
4 |
5 | request.onload = function() {
6 |     poemDisplay.textContent = request.response;
```

```
7 | };  
8 |  
9 | request.send();
```

3. Replace all the XHR code with this:

```
1 | fetch(url).then(function(response) {  
2 |     response.text().then(function(text) {  
3 |         poemDisplay.textContent = text;  
4 |     });  
5 | });
```

4. Load the example in your browser (running it through a web server) and it should work just the same as the XHR version, provided you are running a modern browser.

So what is going on in the Fetch code?

First of all, we invoke the `fetch()` method, passing it the URL of the resource we want to fetch. This is the modern equivalent of `request.open()` in XHR, plus you don't need any equivalent to `.send()`.

After that, you can see the `.then()` method chained onto the end of `fetch()` — this method is a part of **Promises**, a modern JavaScript feature for performing asynchronous operations. `fetch()` returns a promise, which resolves to the response sent back from the server — we use `.then()` to run some follow-up code after the promise resolves, which is the function we've defined inside it. This is the equivalent of the `onload` event handler in the XHR version.

This function is automatically passed the response from the server as a parameter when the `fetch()` promise resolves. Inside the function we grab the response and run its `text()` method, which basically returns the response as raw text. This is the equivalent of `request.responseText = 'text'` in the XHR version.

You'll see that `text()` also returns a promise, so we chain another `.then()` onto it, inside of which we define a function to receive the raw text that the `text()` promise resolves to.

Inside the inner promise's function, we do much the same as we did in the XHR version — set the `<pre>` element's text content to the text value.

Aside on promises

Promises are a bit confusing the first time you meet them, but don't worry too much about this for now. You'll get used to them after a while, especially as you learn more about modern JavaScript APIs — most of the newer ones are heavily based on promises.

Let's look at the promise structure from above again to see if we can make some more sense of it:

```
1 fetch(url).then(function(response) {  
2   response.text().then(function(text) {  
3     poemDisplay.textContent = text;  
4   });  
5 });
```

The first line is saying "fetch the resource located at url" (`fetch(url)`) and "then run the specified function when the promise resolves" (`.then(function() { ... })`). "Resolve" means "finish performing the specified operation at some point in the future". The specified operation in this case is to fetch a resource from a specified URL (using an HTTP request), and return the response for us to do something with.

Effectively, the function passed into `then()` is a chunk of code that won't run immediately — instead, it will run at some point in the future when the response has been returned. Note that you could also choose to store your promise in a variable, and chain `.then()` onto that instead. the below code would do the same thing:

```
1 var myFetch = fetch(url);  
2  
3 myFetch.then(function(response) {  
4   response.text().then(function(text) {  
5     poemDisplay.textContent = text;  
6   });  
7 });
```

Because the `fetch()` method returns a promise that resolves to the HTTP response, any function you define inside a `.then()` chained onto the end of it will automatically be given the response as a parameter. You can call the parameter anything you like — the below example would still work:

```
1 fetch(url).then(function(dogBiscuits) {  
2   dogBiscuits.text().then(function(text) {  
3     poemDisplay.textContent = text;
```

```
4   });  
5   });
```

But it makes more sense to call the parameter something that describes its contents!

Now let's focus just on the function:

```
1  function(response) {  
2    response.text().then(function(text) {  
3      poemDisplay.textContent = text;  
4    });  
5  }
```

The response object has a method `text()`, which takes the raw data contained in the response body and turns it into plain text, which is the format we want it in. It also returns a promise (which resolves to the resulting text string), so here we use another `.then()`, inside of which we define another function that dictates what we want to do with that text string. We are just setting the `textContent` property of our poem's `<pre>` element to equal the text string, so this works out pretty simple.

It is also worth noting that you can directly chain multiple promise blocks (`.then()` blocks, but there are other types too) onto the end of one another, passing the result of each block to the next block as you travel down the chain. This makes promises very powerful.

The following block does the same thing as our original example, but is written in a different style:

```
1  fetch(url).then(function(response) {  
2    return response.text()  
3  }).then(function(text) {  
4    poemDisplay.textContent = text;  
5  });
```

Many developers like this style better, as it is flatter and arguably easier to read for longer promise chains — each subsequent promise comes after the previous one, rather than being inside the previous one (which can get unwieldy). The only other difference is that we've had to include a `return` statement in front of `response.text()`, to get it to pass its result on to the next link in the chain.

Which mechanism should you use?

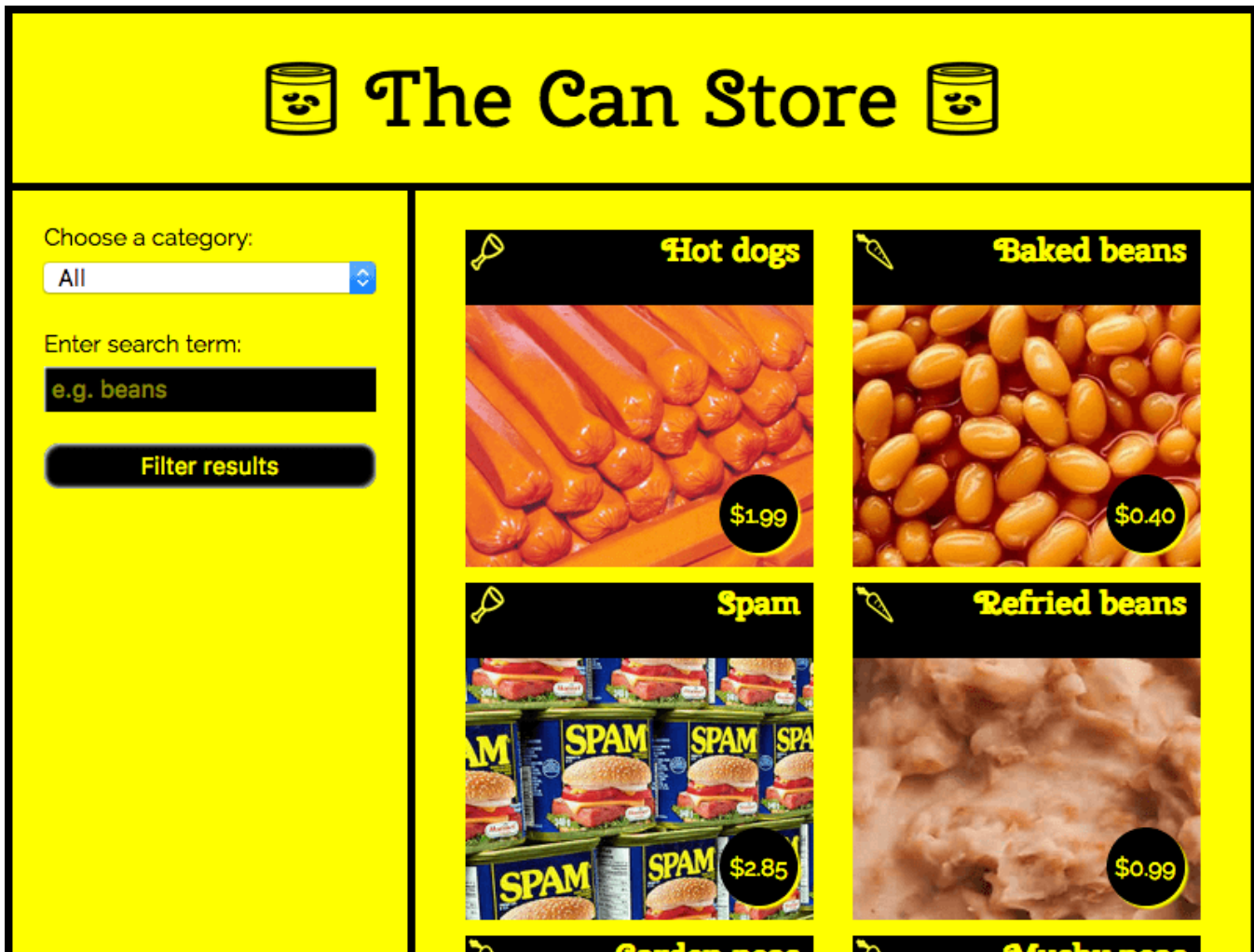
This really depends on what project you are working on. XHR has been around for a long time now and has very good cross-browser support. Fetch and Promises, on the other hand, are a more recent addition to the web platform, although they're supported well across the browser landscape, with the exception of Internet Explorer and Safari (which at the time of writing has Fetch available in its latest technology preview).

If you need to support older browsers, then an XHR solution might be preferable. If however you are working on a more progressive project and aren't as worried about older browsers, then Fetch could be a good choice.

You should really learn both — Fetch will become more popular as Internet Explorer declines in usage (IE is no longer being developed, in favor of Microsoft's new Edge browser), but you might need XHR for a while yet.

A more complex example

To round off the article, we'll look at a slightly more complex example that shows some more interesting uses of Fetch. We have created a sample site called The Can Store — it's a fictional supermarket that only sells canned goods. You can find this [example](#) live on GitHub, and [see the source code](#).



By default, the site displays all the products, but you can use the form controls in the left hand column to filter them by category, or search term, or both.

There is quite a lot of complex code that deals with filtering the products by category and search terms, manipulating strings so the data displays correctly in the UI, etc. We won't discuss all of it in the article, but you can find extensive comments in the code (see [can-script.js](#)).

We will however explain the Fetch code.

The first block that uses Fetch can be found at the start of the JavaScript:

```

1  fetch('products.json').then(function(response) {
2    if(response.ok) {
3      response.json().then(function(json) {
4        products = json;
5        initialize();
6      });
7    } else {
8      console.log('Network request for products.json failed with response '

```

```
9 |   }  
10 | });
```

This looks similar to what we saw before, except that the second promise is inside a conditional statement. In the condition we check to see if the response returned has been successful — the `response.ok` property contains a Boolean that is `true` if the response was OK (e.g. 200 meaning "OK"), or `false` if it was unsuccessful.

If the response was successful, we run the second promise — this time however we use `json()`, not `text()`, as we want to return our response as structured JSON data, not plain text.

If the response was not successful, we print out an error to the console stating that the network request failed, which reports the network status and descriptive message of the response (contained in the `response.status` and `response.statusText` properties, respectively). Of course a complete web site would handle this error more gracefully, by displaying a message on the user's screen and perhaps offering options to remedy the situation.

You can test the fail case yourself:

1. Make a local copy of the example files (download and unpack [the can-store ZIP file](#))
2. Run the code through a web server (as described above, in [Serving your example from a server](#))
3. Modify the path to the file being fetched, to something like 'produc.json' (i.e. make sure it is misspelled)
4. Now load the index file in your browser (e.g. via `localhost:8000`) and look in your browser developer console. You'll see a message along the lines of "Network request for products.json failed with response 404: File not found"

The second Fetch block can be found inside the `fetchBlob()` function:

```
1 fetch(url).then(function(response) {  
2   if(response.ok) {  
3     response.blob().then(function(blob) {  
4       objectURL = URL.createObjectURL(blob);  
5       showProduct(objectURL, product);  
6     });  
7   } else {  
8     console.log('Network request for "' + product.name + '" image failed w
```

```
    9 |     }  
    10 |   });
```

This works in much the same way as the previous one, except that instead of using `json()`, we use `blob()` — in this case we want to return our response as an image file, and the data format we use for that is `Blob` — the term is an abbreviation of "Binary Large Object", and can basically be used to represent large file-like objects — such as images or video files.

Once we've successfully received our blob, we create an object URL out of it, using `createObjectURL()`. This returns a temporary internal URL that points to an object referenced inside the browser. These are not very readable, but you can see what one looks like by opening up the Can Store app, Ctrl-/Right-clicking on an image, and selecting the "View image" option (which might vary slightly depending on what browser you are using). The object URL will be visible inside the address bar, and should be something like this:


```
1 | blob:http://localhost:7800/9b75250e-5279-e249-884f-d03eb1fd84f4
```

Challenge: An XHR version of the Can Store

We'd like you to have a go at converting the Fetch version of the app to use XHR, as a useful bit of practice. Take a [copy of the ZIP file](#), and try modifying the JavaScript as appropriate.

Some helpful hints:

- You might find the `XMLHttpRequest` reference material useful.
- You will basically need to use the same pattern as you saw earlier in the [XHR-basic.html](#) example.
- You will, however, need to add the error handling we showed you in the Fetch version of the Can Store:
 - The response is found in `request.response` after the `load` event has fired, not in a promise `then()`.
 - About the best equivalent to Fetch's `response.ok` in XHR is to check whether `request.status` is equal to 200, or if `request.readyState` is equal to 4.
 - The properties for getting the status and status message are the same, but they are found on the `request` (XHR) object, not the `response` object.

 **Note:** If you have trouble with this, feel free to check your code against the finished version on GitHub ([↗ see the source here](#), and also [↗ see it running live](#)).

Summary

That rounds off our article on fetching data from the server. By this point you should have an idea of how to start working with both XHR and Fetch.

See also

There are however a lot of different subjects discussed in this article, which has only really scratched the surface. For a lot more detail on these subjects, try the following articles:

- [Ajax — Getting started](#)
- [Using Fetch](#)
- [Promises](#)
- [Working with JSON data](#)
- [An overview of HTTP](#)
- [Server-side website programming](#)

[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

In this module

- [Introduction to web APIs](#)
- [Manipulating documents](#)
- [Fetching data from the server](#)

- Third party APIs
 - Drawing graphics
 - Video and audio APIs
 - Client-side storage
-