**Working with the Fetch API**

**What is fetch?**

The [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) is a simple interface for fetching resources. Fetch makes it easier to make web requests and handle responses than with the older [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest), which often requires additional logic (for example, for handling redirects).

**Note:** Fetch supports the [Cross Origin Resource Sharing (CORS)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS). Testing generally requires running a local server. Note that although fetch does not require HTTPS, service workers do and so using fetch in a service worker requires HTTPS. Local servers are exempt from this.

You can check for browser support of fetch in the window interface. For example:

**main.js**

if (!('fetch' in window)) {  
  console.log('Fetch API not found, try including the polyfill');  
  return;  
}  
// We can safely use fetch from now on

There is a [polyfill](https://github.com/github/fetch) for [browsers that are not currently supported](http://caniuse.com/#feat=fetch) (but see the readme for important caveats.).

The [fetch() method](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch) takes the path to a resource as input. The method returns a [promise](http://www.html5rocks.com/en/tutorials/es6/promises/) that resolves to the [Response](https://developer.mozilla.org/en-US/docs/Web/API/Response) of that request.

**Making a request**

Let's look at a simple example of fetching a JSON file:

**main.js**

fetch('examples/example.json')  
.then(function(response) {  
  // Do stuff with the response  
})  
.catch(function(error) {  
  console.log('Looks like there was a problem: \n', error);  
});

We pass the path for the resource we want to retrieve as a parameter to fetch. In this case this is **examples/example.json**. The fetch call returns a promise that resolves to a response object.

When the promise resolves, the response is passed to .then. This is where the response could be used. If the request does not complete, .catch takes over and is passed the corresponding error.

Response objects represent the response to a request. They contain the requested resource and useful properties and methods. For example, response.ok, response.status, and response.statusText can all be used to evaluate the status of the response.

Evaluating the success of responses is particularly important when using fetch because bad responses (like 404s) still resolve. The only time a fetch promise will reject is if the request was unable to complete. The previous code segment would only fall back to .catch if there was no network connection, but not if the response was bad (like a 404). If the previous code were updated to validate responses it would look like:

**main.js**

fetch('examples/example.json')  
.then(function(response) {  
  if (!response.ok) {  
    throw Error(response.statusText);  
  }  
  // Do stuff with the response  
})  
.catch(function(error) {  
  console.log('Looks like there was a problem: \n', error);  
});

Now if the response object's ok property is false (indicating a non 200-299 response), the function throws an error containing response.statusText that triggers the .catch block. This prevents bad responses from propagating down the fetch chain.

**Reading the response object**

Responses must be read in order to access the body of the response. Response objects have [methods](https://developer.mozilla.org/en-US/docs/Web/API/Response) for doing this. For example, [Response.json()](https://developer.mozilla.org/en-US/docs/Web/API/Body/json) reads the response and returns a promise that resolves to JSON. Adding this step to the current example updates the code to:

**main.js**

fetch('examples/example.json')  
.then(function(response) {  
  if (!response.ok) {  
    throw Error(response.statusText);  
  }  
  // Read the response as json.  
  return response.json();  
})  
.then(function(responseAsJson) {  
  // Do stuff with the JSON  
  console.log(responseAsJson);  
})  
.catch(function(error) {  
  console.log('Looks like there was a problem: \n', error);  
});

This code will be cleaner and easier to understand if it's abstracted into functions:

**main.js**

function logResult(result) {  
  console.log(result);  
}  
  
function logError(error) {  
  console.log('Looks like there was a problem: \n', error);  
}  
  
function validateResponse(response) {  
  if (!response.ok) {  
    throw Error(response.statusText);  
  }  
  return response;  
}  
  
function readResponseAsJSON(response) {  
  return response.json();  
}  
  
function fetchJSON(pathToResource) {  
  fetch(pathToResource) // 1  
  .then(validateResponse) // 2  
  .then(readResponseAsJSON) // 3  
  .then(logResult) // 4  
  .catch(logError);  
}  
  
fetchJSON('examples/example.json');

(This is [promise chaining](https://developers.google.com/web/ilt/pwa/working-with-promises#chaining).)

To summarize what's happening:

Step 1. Fetch is called on a resource, **examples/example.json**. Fetch returns a promise that will resolve to a response object. When the promise resolves, the response object is passed to validateResponse.

Step 2. validateResponse checks if the response is valid (is it a 200-299?). If it isn't, an error is thrown, skipping the rest of the then blocks and triggering the catch block. This is particularly important. Without this check bad responses are passed down the chain and could break later code that may rely on receiving a valid response. If the response is valid, it is passed to readResponseAsJSON.

**Note:** You can also handle any network status code using the **status** property of the **response** object. This lets you respond with custom pages for different errors or handle other responses that are not **ok** (i.e., not 200-299), but still usable (e.g., status codes in the 300 range). See [Caching files with the service worker](https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker#generic-fallback) for an example of a custom response to a 404.

Step 3. readResponseAsJSON reads the body of the response using the [Response.json()](https://developer.mozilla.org/en-US/docs/Web/API/Body/json) method. This method returns a promise that resolves to JSON. Once this promise resolves, the JSON data is passed to logResult. (Can you think of what would happen if the promise from response.json() rejects?)

Step 4. Finally, the JSON data from the original request to **examples/example.json** is logged by logResult.

**For more information**

* [Response interface](https://developer.mozilla.org/en-US/docs/Web/API/Response)
* [Response.json()](https://developer.mozilla.org/en-US/docs/Web/API/Body/json)
* [Promise chaining](https://developers.google.com/web/ilt/pwa/working-with-promises#chaining)

**Example: fetching images**

Let's look at an example of fetching an image and appending it to a web page.

**main.js**

function readResponseAsBlob(response) {  
  return response.blob();  
}  
  
function showImage(responseAsBlob) {  
  // Assuming the DOM has a div with id 'container'  
  var container = document.getElementById('container');  
  var imgElem = document.createElement('img');  
  container.appendChild(imgElem);  
  var imgUrl = URL.createObjectURL(responseAsBlob);  
  imgElem.src = imgUrl;  
}  
  
function fetchImage(pathToResource) {  
  fetch(pathToResource)  
  .then(validateResponse)  
  .then(readResponseAsBlob)  
  .then(showImage)  
  .catch(logError);  
}  
  
fetchImage('examples/kitten.jpg');

In this example an image (**examples/kitten.jpg)** is fetched. As in the previous example, the response is validated with validateResponse. The response is then read as a [Blob](https://developer.mozilla.org/en-US/docs/Web/API/Blob) (instead of as JSON), and an image element is created and appended to the page, and the image's src attribute is set to a data URL representing the Blob.

**Note:** The [URL object's](https://developer.mozilla.org/en-US/docs/Web/API/URL) [createObjectURL() method](https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL) is used to generate a data URL representing the Blob. This is important to note as you cannot set an image's source directly to a Blob. The Blob must first be converted into a data URL.

**For more information**

* [Blobs](https://developer.mozilla.org/en-US/docs/Web/API/Blob)
* [Response.blob()](https://developer.mozilla.org/en-US/docs/Web/API/Body/blob)
* [URL object](https://developer.mozilla.org/en-US/docs/Web/API/URL)

**Example: fetching text**

Let's look at another example, this time fetching some text and inserting it into the page.

**main.js**

function readResponseAsText(response) {  
  return response.text();  
}  
  
function showText(responseAsText) {  
  // Assuming the DOM has a div with id 'message'  
  var message = document.getElementById('message');  
  message.textContent = responseAsText;  
}  
  
function fetchText(pathToResource) {  
  fetch(pathToResource)  
  .then(validateResponse)  
  .then(readResponseAsText)  
  .then(showText)  
  .catch(logError);  
}  
  
fetchText('examples/words.txt');

In this example a text file is being fetched, **examples/words.txt**. Like the previous two exercises, the response is validated with validateResponse. Then the response is read as text, and appended to the page.

**Note:** It may be tempting to fetch HTML and append that using the **innerHTML** attribute, but be careful -- this can expose your site to [cross site scripting attacks](https://www.google.com/about/appsecurity/learning/xss/)!

**For more information**

* [Response.text()](https://developer.mozilla.org/en-US/docs/Web/API/Body/text)

**Note:** For completeness, the methods we have used are actually methods of [Body](https://developer.mozilla.org/en-US/docs/Web/API/Body), a Fetch API [mixin](https://developer.mozilla.org/en-US/docs/Glossary/mixin) that is implemented in the Response object.

**Making custom requests**

fetch() can also receive a second optional parameter, init, that allows you to create custom settings for the request, such as the [request method](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods), cache mode, credentials, [and more](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch).

**Example: HEAD requests**

By default fetch uses the GET method, which retrieves a specific resource, but other request HTTP methods can also be used.

HEAD requests are just like GET requests except the body of the response is empty. You can use this kind of request when all you want the file's metadata, and you want or need the file's data to be transported.

To call an API with a HEAD request, set the method in the init parameter. For example:

**main.js**

fetch('examples/words.txt', {  
  method: 'HEAD'  
})

This will make a HEAD request for **examples/words.txt**.

You could use a HEAD request to check the size of a resource. For example:

**main.js**

function checkSize(response) {  
  var size = response.headers.get('content-length');  
  // Do stuff based on response size  
}  
  
function headRequest(pathToResource) {  
  fetch(pathToResource, {  
    method: 'HEAD'  
  })  
  .then(validateResponse)  
  .then(checkSize)  
  // ...  
  .catch(logError);  
}  
  
headRequest('examples/words.txt');

Here the HEAD method is used to request the size (in bytes) of a resource (represented in the **content-length** header) without actually loading the resource itself. In practice this could be used to determine if the full resource should be requested (or even how to request it).

**Example: POST requests**

Fetch can also send data to an API with POST requests. The following code sends a "title" and "message" (as a string) to **someurl/comment**:

**main.js**

fetch('someurl/comment', {  
  method: 'POST',  
  body: 'title=hello&message=world'  
})

**Note:** In production, remember to always encrypt any sensitive user data.

The method is again specified with the init parameter. This is also where the body of the request is set, which represents the data to be sent (in this case the title and message).

The body data could also be extracted from a form using the [FormData](https://developer.mozilla.org/en-US/docs/Web/API/FormData/FormData) interface. For example, the above code could be updated to:

**main.js**

// Assuming an HTML <form> with id of 'myForm'  
fetch('someurl/comment', {  
  method: 'POST',  
  body: new FormData(document.getElementById('myForm')  
})

**Custom headers**

The init parameter can be used with the [Headers](https://developer.mozilla.org/en-US/docs/Web/API/Headers) interface to perform various actions on HTTP request and response headers, including retrieving, setting, adding, and removing them. An example of reading response headers was shown in a [previous section](https://developers.google.com/web/ilt/pwa/working-with-the-fetch-api#head). The following code demonstrates how a custom [Headers](https://developer.mozilla.org/en-US/docs/Web/API/Headers) object can be created and used with a fetch request:

**main.js**

var myHeaders = new Headers({  
  'Content-Type': 'text/plain',  
  'X-Custom-Header': 'hello world'  
});  
  
fetch('/someurl', {  
  headers: myHeaders  
});

Here we are creating a Headers object where the Content-Type header has the value of text/plain and a custom X-Custom-Header header has the value of hello world.

**Note:** Only some headers, like **Content-Type** can be modified. Others, like **Content-Length** and **Origin** are [guarded](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#Guard), and cannot be modified (for security reasons).

Custom headers on [cross-origin](https://developers.google.com/web/ilt/pwa/working-with-the-fetch-api#cors) requests must be supported by the server from which the resource is requested. The server in this example would need to be configured to accept the X-Custom-Header header in order for the fetch to succeed. When a custom header is set, the browser performs a [preflight](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Preflighted_requests) check. This means that the browser first sends an OPTIONS request to the server to determine what HTTP methods and headers are allowed by the server. If the server is configured to accept the method and headers of the original request, then it is sent. Otherwise, an error is thrown.

**For more information**

* [Headers](https://developer.mozilla.org/en-US/docs/Web/API/Headers)
* [Preflight checks](https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request)

**Cross-origin requests**

Fetch (and XMLHttpRequest) follow the [same-origin policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy). This means that browsers restrict [cross-origin](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS) HTTP requests from within scripts. A cross-origin request occurs when one domain (for example **http://foo.com/**) requests a resource from a separate domain (for example **http://bar.com/**). This code shows a simple example of a cross-origin request:

**main.js**

// From http://foo.com/  
fetch('http://bar.com/data.json')  
.then(function(response) {  
  // Do something with response  
});

**Note:** Cross-origin request restrictions are often a point of confusion. Many resources like images, stylesheets, and scripts are fetched cross-origin. However, these are exceptions to the same-origin policy. Cross-origin requests are still restricted *from within scripts*.

There have been attempts to work around the same-origin policy (such as [JSONP](http://stackoverflow.com/questions/2067472/what-is-jsonp-all-about)). The [Cross Origin Resource Sharing](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS) (CORS) mechanism has enabled a standardized means of retrieving cross-origin resources. The CORS mechanism lets you specify in a request that you want to retrieve a cross-origin resource (in fetch this is enabled by default). The browser adds an Origin header to the request, and then requests the appropriate resource. The browser only returns the response if the server returns an Access-Control-Allow-Origin header specifying that the origin has permission to request the resource. In practice, servers that expect a variety of parties to request their resources (such as 3rd party APIs) set a wildcard value for the Access-Control-Allow-Origin header, allowing anyone to access that resource.

If the server you are requesting from doesn't support CORS, you should get an error in the console indicating that the cross-origin request is blocked due to the CORS Access-Control-Allow-Origin header being missing.

You can use [no-cors](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch) mode to request opaque resources. [Opaque responses](https://fetch.spec.whatwg.org/#concept-filtered-response-opaque) can't be accessed with JavaScript but the response can still be served or cached by a service worker. Using no-cors mode with fetch is relatively simple. To update the above example with no-cors, we pass in the init object with mode set to no-cors:

**main.js**

  // From http://foo.com/  
fetch('http://bar.com/data.json', {  
  mode: 'no-cors' // 'cors' by default  
})  
.then(function(response) {  
  // Do something with response  
});