# Fetch API

A brownbag deep-dive at


by Seth House

@whiteinge
seth@eseth.com

# Prerequisites

(Topics we won't be covering today.)

# Promises

# Using Fetch



The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network.

# Using Fetch

```
fetch(resource [, init])
```

# fetch(resource [, init])

```
fetch('/path/here')
    .then(console.log)
    .catch(console.error)
```

# fetch(resource [, init])

```
» fetch('/')
      .then(console.log)
      .catch(console.error)
← ▸ Promise { <state>: "pending" }
  ▾ Response
    ▸ body: ReadableStream { locked: false }
      bodyUsed: false
    ▸ headers: Headers {  }
      ok: true
      redirected: false
      status: 200
      statusText: "OK"
      type: "basic"
      url: "http://localhost:8000/"
    ▾ <prototype>: ResponsePrototype
      ▸ arrayBuffer: function arrayBuffer()
      ▸ blob: function blob()
        body: »
        bodyUsed: »
      ▸ clone: function clone()
      ▸ constructor: function ()
      ▸ formData: function formData()
        headers: »
      ▸ json: function json()
        ok: »
        redirected: »
        status: »
        statusText: »
      ▸ text: function text()
        type: »
        url: »
        Symbol(Symbol.toStringTag): "Response"
```

# fetch(resource [, init])

```
fetch('https://api.github.com/users', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})
    .then(x => x.json())
    .then(console.log)
    .catch(console.error)
```

# fetch(resource [, init])

```
fetch('https://api.github.com/users', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})
    .then(x => x.json())
    .then(console.log)
    .catch(console.error)
```

# fetch(resource [, init])

```
fetch('/path/here', {
    method: 'POST',
    body: JSON.stringify({foo: 'Foo!'}),
})
    .then(console.log)
    .catch(console.error)
```

# Contrast: XMLHttpRequest

```
const req = new XMLHttpRequest();
req.addEventListener('load', function() {
    console.log(JSON.parse(this.responseText));
});
req.open('GET', 'https://api.github.com/users');
req.setRequestHeader('Accept', 'application/json');
req.send();
```

# Fetch API

(Sister interfaces and mixins.)

# Request

# `new Request(input[, init])`

Exact same API as `fetch()`!

```
const req = new Request('https://api.github.com/users', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})
```

# `new Request(input[, init])`

Exact same API as `fetch()`!

```
const req = new Request('https://api.github.com/users', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})
```

```
const req = new Request('/path/here', {
    method: 'POST',
    body: JSON.stringify({foo: 'Foo!'}),
})
```

# `fetch` accepts `Request` instances

```
const req = new Request('https://httpbin.org/get', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})

fetch(req).then(console.log).catch(console.error)
```

# `fetch` accepts `Request` instances

```
const req = new Request('https://httpbin.org/get', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})

fetch(req).then(console.log).catch(console.error)
```

Useful for augmenting a request before sending it.

(Stay tuned.)

# `Request` accepts `Request` instances

```
const req1 = new Request('https://httpbin.org/post', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})

const req2 = new Request(req1, {
    method: 'POST',
    body: JSON.stringify({foo: 'Foo!'}),
})

console.log(req2.headers.get('Accept'))
// => application/json
```

# `Request` accepts `Request` instances

```
const req1 = new Request('https://httpbin.org/post', {
    mode: 'cors',
    headers: {'Accept': 'application/json'},
})

const req2 = new Request(req1, {
    method: 'POST',
    body: JSON.stringify({foo: 'Foo!'}),
})

console.log(req2.headers.get('Accept'))
// => application/json
```

Makes a new copy of the instance.

```
console.log(req1.method)
// => GET
```

# Headers

# new Headers(init)

```
const headers = new Headers({
    'Accept': 'application/json',
    'Content-Type': 'application/json',
})
```

# Normalize lookups

```
const headers1 = {'content-type': 'application/json'}
const headers2 = new Headers({'content-type': 'application/json'})

if (headers1['Content-Type'] === 'application/json') {
    JSON.parse(someJsonString)
}
// Miss!

if (headers2.get('Content-Type') === 'application/json') {
    JSON.parse(someJsonString)
}
// Hit.
```

# `Request` accepts `Headers`

```
const req = new Request('/some/path', {
    headers: new Headers({'Accept': 'application/json'}),
})
```

# `Request` creates `Headers`

```
const req = new Request('https://api.github.com/users', {
    headers: {'Accept': 'application/json'},
})

req.headers.get('accept')
// => application/json
```

# Blob

# `new Blob( array[, options])`

- Represents a file-like object.
- Consists of the concatenation of the values in the parameter array.
- Isn't necessarily in a JavaScript-native format.

# `new Blob( array[, options])`

- Represents a file-like object.
- Consists of the concatenation of the values in the parameter array.
- Isn't necessarily in a JavaScript-native format.

A tad abstract outside of niche use-cases (binary data, images, etc).

(Stay tuned.)

# Encapsulate data in a Blob

```
const fileObj = new Blob(['Hello, world!'])

console.log(fileObj.size)
// => 13
```

# Associate a mime type

```
const fileObj = new Blob(['{"foo": "Foo!"}'], {
    type: 'application/json',
})

console.log(fileObj.type)
// => application/json
```

# `Request` accepts `Blob`

```
const req = new Request('/some/path', {
    method: 'POST',
    body: new Blob([JSON.stringify({foo: 'Foo!'})], {
        type: 'application/json',
    }),
})

console.log(req.headers.get('content-type'))
// => application/json
```

# `Request` accepts `Blob`

```
const req = new Request('/some/path', {
    method: 'POST',
    body: new Blob([JSON.stringify({foo: 'Foo!'})], {
        type: 'application/json',
    }),
})

console.log(req.headers.get('content-type'))
// => application/json
```

(!)

# Response

# `new Response(body, init)`

Maybe useful for mocking an HTTP response in a unit test. Usually will come from `fetch()`.

# Response body file-like objects

# More promises

`Response` is available as soon as the response headers finish.

The response body may yet still be streaming in, thus another promise.

# A note about `ok`

If `status` is in the range of 200-299.

# A note about `ok`

If `status` is in the range of 200-299.

Does not throw an error for non-200 responses.

# A note about `ok`

If `status` is in the range of 200-299.

Does not throw an error for non-200 responses.

```
request('/some/path')
    .then(rep => {
        if (rep.ok) {
            return rep.json();
        } else {
            // A common question:
            // - throw new Error('Oh noes!')
            // - return rep.json();
        }
    })
    .catch(console.error)
```

# A note about `ok`

If `status` is in the range of 200-299.

Does not throw an error for non-200 responses.

```
request('/some/path')
    .then(rep => {
        if (rep.ok) {
            return rep.json();
        } else {
            // A common question:
            // - throw new Error('Oh noes!')
            // - return rep.json();
        }
    })
    .catch(console.error)
```

Many error types. Don't conflate them!

- Network errors (timeout, blocked, cache failure).
- HTTP non-success status codes (user error, server error, upstream proxy error, many more).
- Uncaught JavaScript errors (application bugs, unparseable JSON).

# REST APIs and success or failure

```
HTTP/1.0 200 OK
Content-Type: application/json

{"error": true, "message": "Bad Request", "data": "Username taken."}
```

vs.

```
HTTP/1.0 400 Bad Request
Content-Type: application/json

{"data": "Username taken."}
```

# REST APIs and success or failure

```
HTTP/1.0 200 OK
Content-Type: application/json

{"error": true, "message": "Bad Request", "data": "Username taken."}
```

vs.

```
HTTP/1.0 400 Bad Request
Content-Type: application/json

{"data": "Username taken."}
```

But that's another rant...

# Interlude

Why using file-like objects is a genius idea.

# Uploads

`File` interface is based on `Blob`

```
<input type="file" onchange="((ev) => {
    fetch('https://httpbin.org/anything', {
        method: 'POST',
        body: ev.target.files[0],
    })
})(event)">
```

# Uploads

`File` interface is based on `Blob`

```
<input type="file" onchange="((ev) => {
    fetch('https://httpbin.org/anything', {
        method: 'POST',
        body: ev.target.files[0],
    })
})(event)">
```

Associates mime type automatically.

Note: this is based on a small, hard-coded list of file types in the browser and then falls back to file associations in the OS. For example, you may get a different mime type for a CSV file on Windows if Microsoft Excel is installed or not.

# Pop-ups

```
<button type="button" onclick="((ev) => {
    fetch('https://httpbin.org/image', {
        headers: {accept: 'image/webp'}})
    .then(x => x.blob())
    .then(blob => window.open(URL.createObjectURL(blob)))
})(event)">Show me!</button>
```

# Pop-ups

```
<button type="button" onclick="((ev) => {
    fetch('https://httpbin.org/image', {
        headers: {accept: 'image/webp'}})
    .then(x => x.blob())
    .then(blob => window.open(URL.createObjectURL(blob)))
})(event)">Show me!</button>
```

Associates mime type with window contents automatically.

PDFs will open in default PDF viewer, images will render, HTML content will be parsed, text content will display, etc.

# Downloads

```
<button type="button" onclick="((ev) => {
    fetch('https://httpbin.org/image', {
        headers: {accept: 'image/webp'}})
    .then(x => x.blob())
    .then(blob => {
        const el = document.createElement('a');
        el.setAttribute('href', window.URL.createObjectURL(blob));
        el.setAttribute('download', 'myimage.webp');
        el.click();
    })
})(event)">Download me!</button>
```

# Downloads

```
<button type="button" onclick="((ev) => {
    fetch('https://httpbin.org/image', {
        headers: {accept: 'image/webp'}})
    .then(x => x.blob())
    .then(blob => {
        const el = document.createElement('a');
        el.setAttribute('href', window.URL.createObjectURL(blob));
        el.setAttribute('download', 'myimage.webp');
        el.click();
    })
})(event)">Download me!</button>
```

Download from the server, store in-memory, then prompt the browser to save the file.

# Form Submission

```
<form
    onsubmit="((ev) => {
        ev.preventDefault();
        fetch('https://httpbin.org/post', {
            method: 'POST',
            body: new FormData(ev.target),
        });
    })(event)"
    method="POST"
>
    <input type="text" name="foo" value="Foo!" />
    <input type="text" name="bar" value="Bar!" />
    <button type="submit">Submit</button>
</form>
```

# Form Submission

```
<form
    onsubmit="((ev) => {
        ev.preventDefault();
        fetch('https://httpbin.org/post', {
            method: 'POST',
            body: new FormData(ev.target),
        });
    })(event)"
    method="POST"
>
    <input type="text" name="foo" value="Foo!" />
    <input type="text" name="bar" value="Bar!" />
    <button type="submit">Submit</button>
</form>
```

Associates mime time as `multipart/form-data` automatically.

# Get string data out of a Blob

tl;dr: modern APIs are in working draft (and also seem incomplete).

# Get string data out of a Blob

tl;dr: modern APIs are in working draft (and also seem incomplete).

Current API:

```
const myBlob = new Blob(['Foo!'])
const f = new FileReader()
f.onload = () => { console.log(f.result) }
f.readAsText(myBlob)
// => Foo!
```

# Get string data out of a Blob

tl;dr: modern APIs are in working draft (and also seem incomplete).

Current API:

```
const myBlob = new Blob(['Foo!'])
const f = new FileReader()
f.onload = () => { console.log(f.result) }
f.readAsText(myBlob)
// => Foo!
```

New API (some support in evergreens):

```
const myBlob = new Blob(['Foo!'])
myBlob.text().then(console.log)
// => Foo!
```

# Conclusion

# Fetch API

The Fetch API is a tight ecosystem of classes and mixins, not only `fetch()`.

# Ajax wrappers

No real application operates without a custom wrapper around the ajax lib...

- Abstract boilerplate request formatting:
  - Add common request headers.
  - Send request as JSON.
  - Authenticate requests: (add auth header or opt-in to sending cookies).
  - Enable CORS.
  - Send XSRF token.
- Consistent response parsing:
  - Parse as JSON.
  - Handle redirects: 301 (Permanent), 302 (Temporary).
  - Handle no-content responses: 201 (Created), 202 (Accepted), 204 (No Content).
  - Handle authorization responses: 401 (Unauthorized), 403 (Forbidden).
  - Success vs error responses: 400 (Client Error), 404 (Not Found), 409 (Conflict), 500 (Server Error), 502 (Gateway Unavailable), 503 (Service Unavailable).
- Response caching & conditional-GET requests: If-None-Match/If-Modified-Since, 304 (Not Modified).

# Fetch API

...but `fetch()` is the most minimal one I've seen.

# Fetch API

...but `fetch()` is the most minimal one I've seen.

*If* you embrace the Fetch API norms and ecosystem:

```
// Avoid this common pattern:
myajaxwrapper('/some/path',
    {foo: 'Foo!', /* automatically JSONify this arg */})

// Instead encode outside the wrapper as a file-like object:
myajaxwrapper('/some/path', tojson({foo: 'Foo!'}))
```

Where:

```
const tojson = data => new Blob([JSON.stringify(data)],
    {type: 'application/json'})
```

# Yet Another Fetch Wrapper

```
const request = (...args) => {
  const req = new Request(...args, { credentials: 'include' })
  req.headers.set('X-CSRF-Token', 'secret!')

  return fetch(req)
    .catch(console.error /* log network error to HB */)
    .then(redirect401sToLoginPage)
}
```

# Yet Another Fetch Wrapper

```
const request = (...args) => {
  const req = new Request(...args, { credentials: 'include' })
  req.headers.set('X-CSRF-Token', 'secret!')

  return fetch(req)
    .catch(console.error /* log network error to HB */)
    .then(redirect401sToLoginPage)
}
```

Input API is *identical* to `fetch()`!

The wrapper is robust and flexible yet simple.

# Yet Another Fetch Wrapper (usage)

```
request('/some/path', { method: 'POST', body: tojson(someData) })
  .then(rep =>
    rep.json().then(data => {
      if (rep.ok) {
        // Do success thing with data.
      } else {
        // Do error thing with data.
      }
    }),
  )
  .catch(console.error /* log uncaught JS error to HB */)
```

# Yet Another Fetch Wrapper (usage)

```
request('/some/path', { method: 'POST', body: tojson(someData) })
  .then(rep =>
    rep.json().then(data => {
      if (rep.ok) {
        // Do success thing with data.
      } else {
        // Do error thing with data.
      }
    }),
  )
  .catch(console.error /* log uncaught JS error to HB */)
```

...the output API lacks that same elegance.

Handling varied return types is cumbersome, though those APIs are being worked on.

Several design flaws in Promises also contribute (non-lazy, silently swallow errors, forgetting to return a nested promise).