Abortable fetch

By Jake Archibald
Human boy working on web standards at Google

The original GitHub issue for "Aborting a fetch" was opened in 2015. Now, if I take 2015 away from 2017 (the current year), I get 2. This demonstrates a bug in maths, because 2015 was in fact "forever" ago.

2015 was when we first started exploring aborting ongoing fetches, and after 780 GitHub comments, a couple of false starts, and 5 pull requests, we finally have abortable fetch landing in browsers, with the first being Firefox 57.

Update: Noooope, I was wrong. Edge 16 landed with abort support first! Congratulations to the Edge team!

I'll dive into the history later, but first, the API:

The controller + signal manoeuvre

Meet the `AbortController` and `AbortSignal`:

```
const controller = new AbortController();
const signal = controller.signal;
```

The controller only has one method:

```
controller.abort();
```

When you do this, it notifies the signal:

```
signal.addEventListener('abort', () => {
  // Logs true:
  console.log(signal.aborted);
});
```

This API is provided by the DOM standard, and that's the entire API. It's deliberately generic so it can be used by other web standards and JavaScript libraries.

Abort signals and fetch

Fetch can take an **`AbortSignal`**. For instance, here's how you'd make a fetch timeout after 5 seconds:

```
const controller = new AbortController();
const signal = controller.signal;

setTimeout(() => controller.abort(), 5000);

fetch(url, { signal }).then(response => {
  return response.text();
}).then(text => {
  console.log(text);
});
```

When you abort a fetch, it aborts both the request and response, so any reading of the response body (such as **`response.text()`**) is also aborted.

Note: It's ok to call **`.abort()`** after the fetch has already completed, fetch simply ignores it.

Here's a demo – At time of writing, the only browser which supports this is Firefox 57. Also, brace yourself, no one with any design skill was involved in creating the demo.

Alternatively, the signal can be given to a request object and later passed to fetch:

```
const controller = new AbortController();
const signal = controller.signal;
const request = new Request(url, { signal });

fetch(request);
```

This works because **`request.signal`** is an **`AbortSignal`**.

Note: Technically, **`request.signal`** isn't the same signal you pass to the constructor. It's a new **`AbortSignal`** that mimics the signal passed to the constructor. This means every **`Request`** has a signal, whether one is given to its constructor or not.

Reacting to an aborted fetch

When you abort an async operation, the promise rejects with a `DOMException` named `AbortError`:

```
fetch(url, { signal }).then(response => {
  return response.text();
}).then(text => {
  console.log(text);
}).catch(err => {
  if (err.name === 'AbortError') {
    console.log('Fetch aborted');
  } else {
    console.error('Uh oh, an error!', err);
  }
});
```

You don't often want to show an error message if the user aborted the operation, as it isn't an "error" if you successfully do that the user asked. To avoid this, use an if-statement such as the one above to handle abort errors specifically.

Here's an example that gives the user a button to load content, and a button to abort. If the fetch errors, an error is shown, *unless* it's an abort error:

```
// This will allow us to abort the fetch.
let controller;

// Abort if the user clicks:
abortBtn.addEventListener('click', () => {
  if (controller) controller.abort();
});

// Load the content:
loadBtn.addEventListener('click', async () => {
  controller = new AbortController();
  const signal = controller.signal;

  // Prevent another click until this fetch is done
  loadBtn.disabled = true;
  abortBtn.disabled = false;

  try {
    // Fetch the content & use the signal for aborting
    const response = await fetch(contentUrl, { signal });
    // Add the content to the page
    output.innerHTML = await response.text();
  }
  catch (err) {
    // Avoid showing an error message if the fetch was aborted
```

```
    if (err.name !== 'AbortError') {
      output.textContent = "Oh no! Fetching failed.";
    }
  }

  // These actions happen no matter how the fetch ends
  loadBtn.disabled = false;
  abortBtn.disabled = true;
});
```

Note: This example uses async functions.

Here's a demo – At time of writing, the only browsers that support this are Edge 16 and Firefox 57.

One signal, many fetches

A single signal can be used to abort many fetches at once:

```
async function fetchStory({ signal }={}) {
  const storyResponse = await fetch('/story.json', { signal });
  const data = await storyResponse.json();

  const chapterFetches = data.chapterUrls.map(async url => {
    const response = await fetch(url, { signal });
    return response.text();
  });

  return Promise.all(chapterFetches);
}
```

In the above example, the same signal is used for the initial fetch, and for the parallel chapter fetches. Here's how you'd use `fetchStory`:

```
const controller = new AbortController();
const signal = controller.signal;

fetchStory({ signal }).then(chapters => {
  console.log(chapters);
});
```

In this case, calling `controller.abort()` will abort whichever fetches are in-progress.

The future

Other browsers

Edge did a great job to ship this first, and Firefox are hot on their trail. Their engineers implemented from the test suite while the spec was being written. For other browsers, here are the tickets to follow:

- Chrome.
- Safari.

In a service worker

I need to finish the spec for the service worker parts, but here's the plan:

As I mentioned before, every **`Request`** object has a **`signal`** property. Within a service worker, **`fetchEvent.request.signal`** will signal abort if the page is no longer interested in the response. As a result, code like this just works:

```
addEventListener('fetch', event => {
  event.respondWith(fetch(event.request));
});
```

If the page aborts the fetch, **`fetchEvent.request.signal`** signals abort, so the fetch within the service worker also aborts.

If you're fetching something other than **`event.request`**, you'll need to pass the signal to your custom fetch(es).

```
addEventListener('fetch', event => {
  const url = new URL(event.request.url);

  if (event.request.method == 'GET' && url.pathname == '/about/') {
    // Modify the URL
    url.searchParams.set('from-service-worker', 'true');
    // Fetch, but pass the signal through
    event.respondWith(
      fetch(url, { signal: event.request.signal })
    );
  }
});
```

Follow the spec to track this – I'll add links to browser tickets once it's ready for implementation.

The history

Yeah… it took a long time for this relatively simple API to come together. Here's why:

API disagreement

As you can see the GitHub discussion is pretty long. There a lot of nuance in that thread (and some lack-of-nuance), but the key disagreement is one group wanted the **`abort`** method to exist on the object returned by **`fetch()`**, whereas the other wanted a separation between getting the response and affecting the response.

These requirements are incompatible, so one group wasn't going to get what they wanted. If that's you, sorry! If it makes you feel better, I was also in that group. But seeing **`AbortSignal`** fit the requirements of other APIs makes it seem like the right choice. Also, allowing chained promises to become abortable would become very complicated, if not impossible.

If you wanted to return an object that provides a response, but can also abort, you could create a simple wrapper:

```
function abortableFetch(request, opts) {
  const controller = new AbortController();
  const signal = controller.signal;

  return {
    abort: () => controller.abort(),
    ready: fetch(request, { ...opts, signal })
  };
}
```

False starts in TC39

There was an effort to make a cancelled action distinct from an error. This included a third promise state to signify "cancelled", and some new syntax to handle cancellation in both sync and async code:

👎 **Not real code** — proposal was withdrawn

```
try {
  // Start spinner, then:
  await someAction();
}
catch cancel (reason) {
  // Maybe do nothing?
}
catch (err) {
  // Show error message
}
finally {
  // Stop spinner
}
```

The most common thing to do when an action is cancelled, is nothing. The above proposal separated cancellation from errors so you didn't need to handle abort errors specifically. **`catch cancel`** let you hear about cancelled actions, but most of the time you wouldn't need to.

This got to stage 1 in TC39, but consensus wasn't achieved, and the proposal was withdrawn.

Our alternative proposal, **`AbortController`**, didn't require any new syntax, so it didn't make sense to spec it within TC39. Everything we needed from JavaScript was already there, so we defined the interfaces within the web platform, specifically the DOM standard. Once we'd made that decision, the rest came together relatively quickly.

Large spec change

`XMLHttpRequest` has been abortable for years, but the spec was pretty vague. It wasn't clear at which points the underlying network activity could be avoided, or terminated, or what happened if there was a race condition between **`abort()`** being called and the fetch completing.

We wanted to get it right this time, but that resulted in a large spec change that needed a lot of reviewing (that's my fault, and a huge thanks to Anne van Kesteren and Domenic Denicola for dragging me through it) and a decent set of tests.

But we're here now! We have a new web primitive for aborting async actions, and multiple fetches can be controlled at once! Further down the line, we'll look at enabling priority changes throughout the life of a fetch, and a higher-level API to observe fetch progress.

Last updated July 2, 2018.