15th January 2020

Fetch: Download progress

The `fetch` method allows to track *download* progress.

Please note: there's currently no way for `fetch` to track *upload* progress. For that purpose, please use XMLHttpRequest, we'll cover it later.

To track download progress, we can use `response.body` property. It's `ReadableStream` – a special object that provides body chunk-by-chunk, as it comes. Readable streams are described in the Streams API specification.

Unlike `response.text()`, `response.json()` and other methods, `response.body` gives full control over the reading process, and we can count how much is consumed at any moment.

Here's the sketch of code that reads the reponse from `response.body`:

```
// instead of response.json() and other methods
const reader = response.body.getReader();

// infinite loop while the body is downloading
while(true) {
  // done is true for the last chunk
  // value is Uint8Array of the chunk bytes
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  console.log(`Received ${value.length} bytes`)
}
```

The result of `await reader.read()` call is an object with two properties:

- **`done`** – `true` when the reading is complete, otherwise `false`.
- **`value`** – a typed array of bytes: `Uint8Array`.

Please note:

Streams API also describes asynchronous iteration over `ReadableStream` with `for await..of` loop, but it's not yet widely supported (see browser issues), so we use `while` loop.

We receive response chunks in the loop, until the loading finishes, that is: until `done` becomes `true`.

To log the progress, we just need for every received fragment `value` to add its length to the counter.

Here's the full working example that gets the response and logs the progress in console, more explanations to follow:

```
// Step 1: start the fetch and obtain a reader
let response = await fetch('https://api.github.com/repos/javascript-tutorial/

const reader = response.body.getReader();

// Step 2: get total length
const contentLength = +response.headers.get('Content-Length');

// Step 3: read the data
let receivedLength = 0; // received that many bytes at the moment
let chunks = []; // array of received binary chunks (comprises the body)
while(true) {
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  chunks.push(value);
  receivedLength += value.length;

  console.log(`Received ${receivedLength} of ${contentLength}`)
}

// Step 4: concatenate chunks into single Uint8Array
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
  chunksAll.set(chunk, position); // (4.2)
  position += chunk.length;
}

// Step 5: decode into a string
let result = new TextDecoder("utf-8").decode(chunksAll);

// We're done!
let commits = JSON.parse(result);
alert(commits[0].author.login);
```

Let's explain that step-by-step:

1. We perform `fetch` as usual, but instead of calling `response.json()`, we obtain a stream reader `response.body.getReader()`.

   Please note, we can't use both these methods to read the same response: either use a reader or a response method to get the result.

2. Prior to reading, we can figure out the full response length from the `Content-Length` header.

   It may be absent for cross-origin requests (see chapter Fetch: Cross-Origin Requests) and, well, technically a server doesn't have to set it. But usually it's at place.

3. Call `await reader.read()` until it's done.

We gather response chunks in the array `chunks`. That's important, because after the response is consumed, we won't be able to "re-read" it using `response.json()` or another way (you can try, there'll be an error).

4. At the end, we have `chunks` – an array of `Uint8Array` byte chunks. We need to join them into a single result. Unfortunately, there's no single method that concatenates those, so there's some code to do that:
    1. We create `chunksAll = new Uint8Array(receivedLength)` – a same-typed array with the combined length.
    2. Then use `.set(chunk, position)` method to copy each `chunk` one after another in it.
5. We have the result in `chunksAll`. It's a byte array though, not a string.

   To create a string, we need to interpret these bytes. The built-in TextDecoder does exactly that. Then we can `JSON.parse` it, if necessary.

   What if we need binary content instead of a string? That's even simpler. Replace steps 4 and 5 with a single line that creates a `Blob` from all chunks:

```
let blob = new Blob(chunks);
```

At the end we have the result (as a string or a blob, whatever is convenient), and progress-tracking in the process.

Once again, please note, that's not for *upload* progress (no way now with `fetch`), only for *download* progress.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)