Fetch & Tracking Download Progress

Summary: in this tutorial, you will learn how to download a file from a server in JavaScript using the `fetch()` method and track the download progress.

Introduction to stream in JavaScript

In JavaScript, a stream is a way of handling data that is being transferred over a network. It is a sequence of bytes processed over time instead of all at once.

By using stream, you can process data as soon as it arrives, which can be more efficient than waiting for all the available data.

Typically, streams handle data in chunks. A chunk is a small piece of data transferred in a stream. This is useful when large amounts of data are arriving gradually over time.

For example, when streaming video over the internet, the server sends video data in chunks, allowing the video to start playing before the entire video file has been downloaded.

The `fetch()` method allows you to track the download progress of a file by using the `ReadableStream`. Here are the steps:

Step 1. Call the `fetch()` to start the download:

```
const response = await fetch(url);
```

Step 2. Create a `ReadableStream` (called source stream) to read the body of the HTTP response in chunks:

```
const reader = response.body.getReader();
```

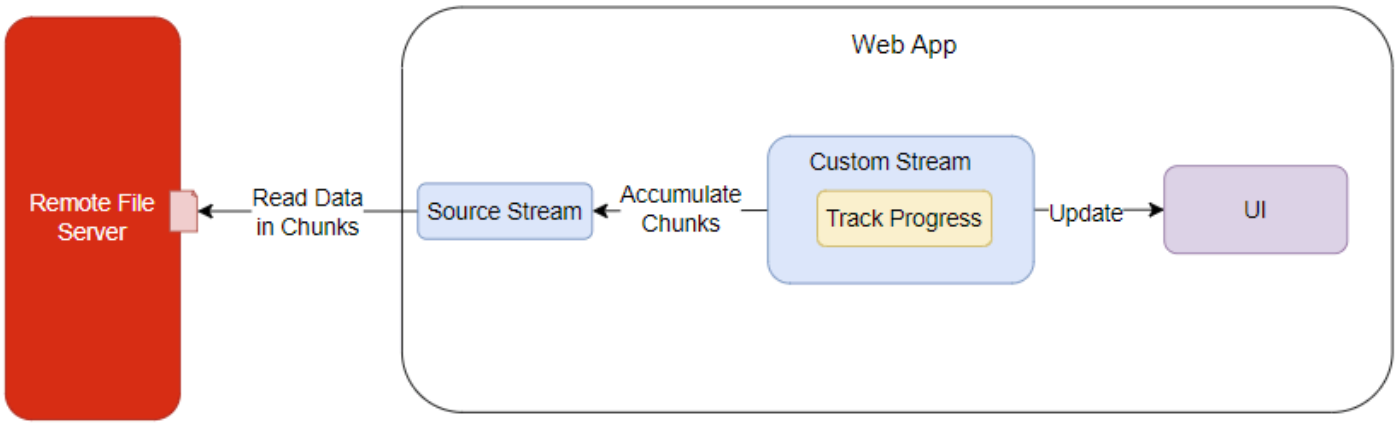
Step 3. Create another ReadableStream to accumulate data from the source stream:

```
const stream  =  new ReadableStream(start);
```

The `start` is a function that is responsible for setting up the stream and controlling how data flows through it.

Step 4. Track the progress by accumulating the size of chunks received and update the progress to the app's user interface (UI).

The following diagram illustrates how to track the download progress using stream:



Fetch Download Progress example

Step 1. Create a new project directory `fetch-progress` :

```
mkdir fetch-progress
cd fetch-progress
```

Step 2. Create a `data.txt` file in the project directory. We'll download this file from the app.

Step 3. Create an `index.html` file with the following code:

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
        <title>Download with Progress</title>
        <link rel="stylesheet" href="css/style.css">
        <script src="js/app.js" defer type="module"></script>
    </head>
    <body>
        <main>
            <h1>File Download Progress</h1>
            <button id="download-btn" class="download-btn">Download File</button>
            <div class="progress-container">
                <progress id="progress-bar" value="0" max="100">
                </progress>
                <div id="progress-text" class="progress-text">0%</div>
            </div>
        </main>
    </body>

</html>
```

Step 4. Create `css/style.css` file.

Step 5. Create the `js/app.js` file to store the JavaScript code:

```
const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

const downloadBtn = document.getElementById('download-btn');
downloadBtn.addEventListener('click', () => {
  const url = 'data.txt';
  downloadFile(url);
});

const progressBar = document.getElementById('progress-bar');
const progressText = document.getElementById('progress-text');

async function downloadFile(url) {
  downloadBtn.disabled = true;
  progressBar.value = 0;
  progressText.textContent = '0%';

  try {
```

```
// fetch the file
const response = await fetch(url);

// check if the response is ok
if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);

// get the total size of the file
const contentLength = response.headers.get('content-length');

// set the max value of the progress bar
const totalSize = contentLength ? parseInt(contentLength, 10) : 0;

// create the stream
const reader = response.body.getReader();

const stream = new ReadableStream({
  // start the stream
  async start(controller) {
    let loaded = 0;
    while (true) {
      // read the next chuck of data
      const { done, value } = await reader.read();
      //  simulate  network delay
      await delay(200);

      if (done) break;

      // calcualte the progress %
      loaded += value.length;
      const progress = totalSize ? (loaded / totalSize) * 100 : 0;

      // update the progressbar
      progressBar.value = progress;
      progressText.textContent = `${progress.toFixed(2)}%`;

      // send the data to the controller
      controller.enqueue(value);
    }
    // close the stream
    controller.close();
```

```
      },
    });

    // create the download link
    createDownloadLink(url, stream);

    // Update the progress text
    progressText.textContent = 'Download Complete!';
  } catch (error) {
    // update the progress text
    progressText.textContent = 'Download Failed!';
  } finally {
    // enable the download button
    downloadBtn.disabled = false;
  }
}

const createDownloadLink = async (url, stream) => {
  // Create a new blob URL
  const responseStream = new Response(stream);
  const blob = await responseStream.blob();
  // Create a link element, set the download attribute and trigger a download
  const link = document.createElement('a');
  link.href = URL.createObjectURL(blob);
  link.download = url.split('/').pop();
  document.body.appendChild(link);
  link.click();
  document.body.removeChild(link);
};
```

How it works.

First, define a function `delay` that simulates a network delay for a specified number of milliseconds. This is optional to test the progress in case the file is small or the network speed is fast, so you cannot see the progress.

```
const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
```

Second, select the download button element and register a click event handler:

```
const downloadBtn = document.getElementById('download-btn');
downloadBtn.addEventListener('click', () => {
  downloadFile('data.txt');
});
```

Inside the click event handler, call the `downloadFile()` function to download the `data.txt` file from the app's root directory.

Third, define the `downloadFile()` function that accepts a URL to download:

```
async function downloadFile(url)
```

In the `downloadFile` function:

1) Select the `progress-bar` and `progress-text` elements:

```
const progressBar = document.getElementById('progress-bar');
const progressText = document.getElementById('progress-text');
```

2) Disable the download button and set the progress bar value to zero and progress text to 0%:

```
downloadBtn.disabled = true;
progressBar.value = 0;
progressText.textContent = '0%';
```

3) Start downloading the file using the `fetch()` method:

```
const response = await fetch(url);
```

4) Throw an error if the request is not successful:

```
if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
```

5) If the request was successful, get the file size by retrieving the `content-length` from the header of the HTTP response:

```
const contentLength = response.headers.get('content-length');
```

6) Convert the `content-length` value to an integer and assign it to the `totalSize` variable:

```
const totalSize = contentLength ? parseInt(contentLength, 10) : 0;
```

7) Get the `ReadableStream` by calling the `getReader()` method of the `request.body` property:

```
const reader = response.body.getReader();
```

This is a source stream that reads data from the file in chunks.

8) Create a second stream (custom stream) that reads data from the source stream:

```
const stream = new ReadableStream({
    // start the stream
    async start(controller) {
// ...
```

The `start()` method will be called automatically when the custom stream starts. It is in charge of setting up the stream and controlling how data flows through it.

9) Initializes a variable `loaded` to track how much data has been read so far:

```
let loaded = 0;
```

10) Create an infinite loop that continuously reads data from the stream until the stream ends:

```
while (true) {
```

11) Read the next data chunk from the stream:

```
const { done, value } = await reader.read();
```

The `read()` method returns a Promise that resolves to an object with two properties:

- `done` : A boolean indicating if the stream has finished reading.
- `value` : The chunk of data that was read.

12) Simulate a network delay of 200ms each time reading data from the stream:

```
await delay(2000);
```

Note that you need to delete this line of code if you want to use it for a production system.

13) If no more data from the stream to read, the done is set to true, exit the loop:

```
if (done) break;
```

14) Accumulate the size of the data chunks that have been read by adding the current chunk size (`value.length`) to the `loaded` variable.

```
loaded += value.length;
```

15) Calculate the percentage of data that has been loaded relative to the total size of the data (`totalSize`).

```
const progress = totalSize ? (loaded / totalSize) * 100 : 0;
```

16) Update the progress bar and progress text:

```
progressBar.value = progress;
progressText.textContent = `${progress.toFixed(2)}%`;
```

17) Add the current data chunk (`value`) to the stream:

```
controller.enqueue(value);
```

18) Close the stream once all data has been read:

```
controller.close();
```

19) Call the `createDownloadLink()` function to download the file:

```
createDownloadLink(url, stream);
```

20) Update the progress text:

```
progressText.textContent = 'Download Complete!';
```

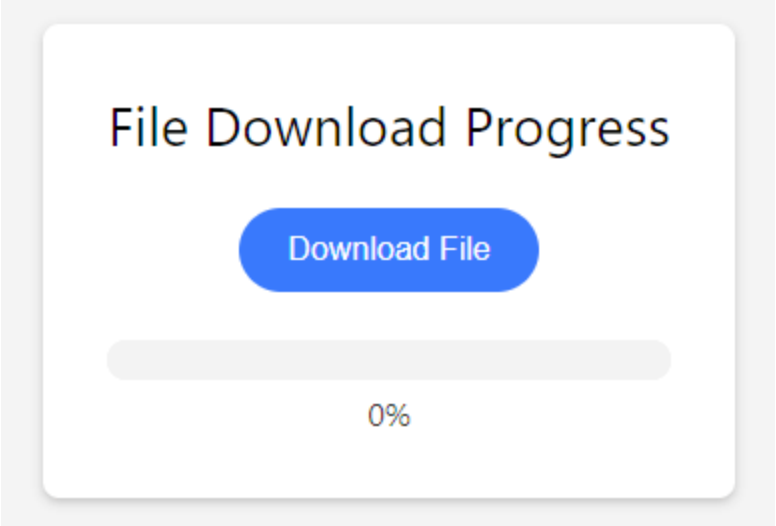
21) If an error occurs while streaming, set an error message to the `progressText` element:

```
progressText.textContent = 'Download Failed!';
```

22) Enable the download button whether the streaming is successful or not:

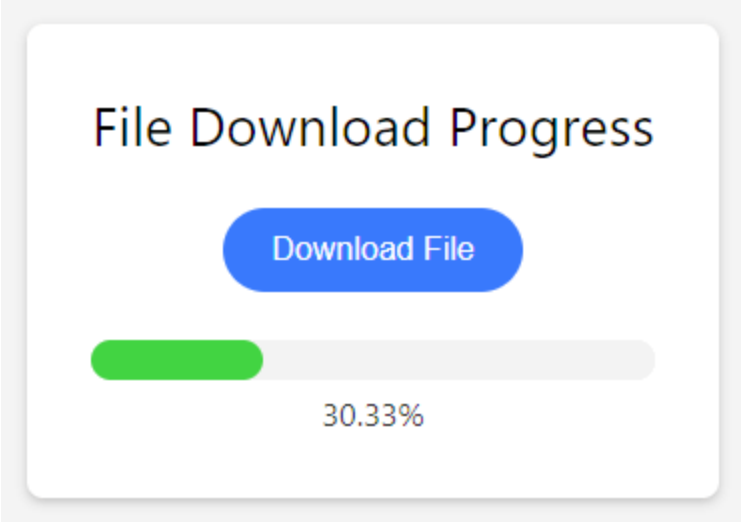
```
} finally {
  // enable the download button
  downloadBtn.disabled = false;
}
```

Step 6. Launch the `index.html` file on the web browser. It'll show the following page:

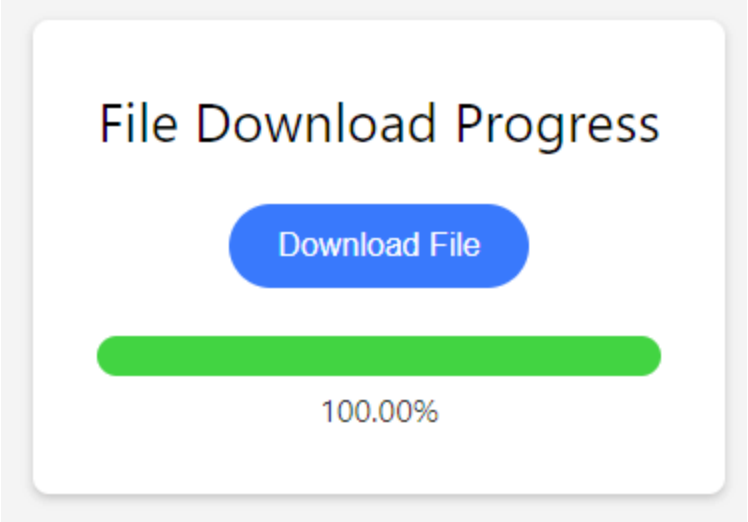


> Note that you need to host the index.html file on a web server. Alternatively, you can launch the index.html file using liveserver extension of VS Code.

If you click the `Download File` button, it'll download the `data.txt` file from the web server, displaying the progress.



Once the download is completed, you'll be prompted to save the file to a directory on your computer:



Download the project source code

Click here to download the project source code

Summary

- Streams allow you to process data transferred over the network in chunks, as soon as the first chunk of data arrives.
- Use ReadableStream API to handle stream data.